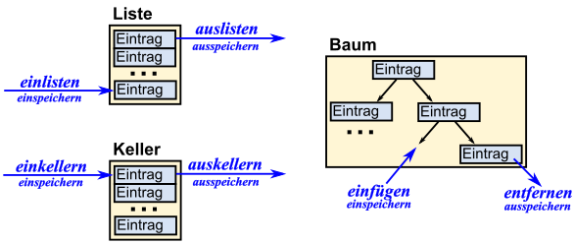
Informatik

für die Sekundarstufe I + II

- Programmieren mit Python -

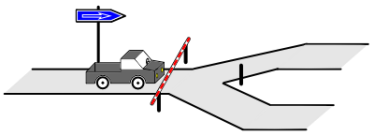
Autor: L. Drews



Grüner Baum-Python
(s) Morelia viridis
Q: de.wikipedia.org (Mwx)

python™ >>>

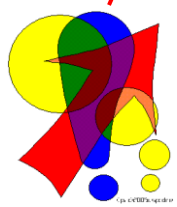
```
while eval(input("?:")) != 0:
    print("Stoppen",end='')
```



teilredigierte Version 0.9c (2020)

(c,p) 2015 - 2020 lern-soft-projekt: drews

Legende:
mit diesem Symbol werden zusätzliche Hinweise, Tips und weiterführende Ideen gekennzeichnet



Nutzungsbestimmungen / Bemerkungen zur Verwendung durch Dritte:

(1) Dieses Skript (Werk) ist zur freien Nutzung in der angebotenen Form durch den Anbieter (lern-soft-projekt) bereitgestellt. Es kann unter Angabe der Quelle und / oder des Verfassers gedruckt, vervielfältigt oder in elektronischer Form veröffentlicht werden.

(2) Das Weglassen von Abschnitten oder Teilen (z.B. Aufgaben und Lösungen) in Teildrucken ist möglich und sinnvoll (Konzentration auf die eigenen Unterrichtsziele, -inhalte und -methoden). Bei angemessen großen Auszügen gehört das vollständige Inhaltsverzeichnis und die Angabe einer Bezugsquelle für das Originalwerk zum Pflichtteil.

(3) Ein Verkauf in jedweder Form ist ausgeschlossen. Der Aufwand für Kopierleistungen, Datenträger oder den (einfachen) Download usw. ist davon unberührt.

(4) Änderungswünsche werden gerne entgegen genommen. Ergänzungen, Arbeitsblätter, Aufgaben und Lösungen mit eigener Autorenschaft sind möglich und werden bei konzeptioneller Passung eingearbeitet. Die Teile sind entsprechend der Autorenschaft zu kennzeichnen. Jedes Teil behält die Urheberrechte seiner Autorenschaft bei.

(5) Zusammenstellungen, die von diesem Skript - über Zitate hinausgehende - Bestandteile enthalten, müssen verpflichtend wieder gleichwertigen Nutzungsbestimmungen unterliegen.

(6) Diese Nutzungsbestimmungen gehören zu diesem Werk.

(7) Der Autor behält sich das Recht vor, diese Bestimmungen zu ändern.

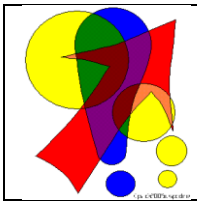
(8) Andere Urheberrechte bleiben von diesen Bestimmungen unberührt.

Rechte Anderer:

Viele der verwendeten Bilder unterliegen verschiedensten freien Lizenzen. Nach meinen Recherchen sollten alle genutzten Bilder zu einer der nachfolgenden freien Lizenzen gehören. Unabhängig von den Vorgaben der einzelnen Lizenzen sind zu jedem extern entstandenen Objekt die Quelle, und wenn bekannt, der Autor / Rechteinhaber angegeben.

public domain (pd)	Zum Gemeingut erklärte Graphiken oder Fotos (u.a.). Viele der verwendeten Bilder entstammen Webseiten / Quellen US-amerikanischer Einrichtungen, die im Regierungsauftrag mit öffentlichen Mitteln finanziert wurden und darüber rechtlich (USA) zum Gemeingut wurden. Andere kreative Leistungen wurden ohne Einschränkungen von den Urhebern freigegeben.
gnu free document licence (GFDL; gnu fdl)	
creative commens (cc) creative commons	BY: od. neu … Namensnennung … nichtkommerziell … in der gleichen Form … unter gleichen Bedingungen

Die meisten verwendeten Lizenzen schließen eine kommerzielle (Weiter-)Nutzung aus!



Bemerkungen zur Rechtschreibung:

Dieses Skript folgt nicht zwangsläufig der neuen **ODER** alten deutschen Rechtschreibung. Vielmehr wird vom Recht auf künstlerische Freiheit, der Freiheit der Sprache und von der Autokorrektur des Textverarbeitungsprogramms microsoft ® WORD ® Gebrauch gemacht.

Für Hinweise auf echte Fehler ist der Autor immer dankbar.

Inhaltsverzeichnis:

Seite

0. Einleitung 12

1. Einstieg und Grundlagen 14

1.1. Geschichte und Namensgebung 14

1.2. Warum Python? 15

grundlegende Python-Konzepte 19

2. Vorbereitung (Installation) 21

2.1. Python auf Windows-Rechnern 21

2.2. Python auf Linux-Rechnern 23

2.3. Python auf dem Raspberry Pi 23

2.5. Python auf Android-Systemen 23

2.5.1. Pydroid3 IDE 23

2.5. Python auf dem MacOS 24

2.6. Python auf dem Taschenrechner 24

2.7. Python auf Microcontrollern 25

3. Zugriff auf das Python-System 26

3.1. die Python-Shell 26

3.1.1. Eingaben an der Shell 27

3.1.2. IDLE als Python-Konsole 27

3.1.2. fortgeschrittene Mathematik 29

3.1.3. mehrzeilige Eingaben an der Shell 30

3.1.4. mehrere Befehle in eine Zeile 31

3.1.2.1. Mathematik für Informatiker – binäres Rechnen 32

3.1.3. Eingaben und Daten merken - Variablen 35

3.1.3.1. besondere Variablen und spezielle Möglichkeiten für Variablen in Python 37

3.2. Arbeiten mit Scripten 40

3.2.1. Grundlagen DOS bzw. Komandozeile (Eingabeaufforderung, Terminal) 40

3.2.2. Aufruf fertiger Python-Skripte 41

3.3. die interne Benutzer-Oberfläche 43

3.3.x. Hilfe(n)! 43

3.4. Nutzung anderer Benutzer-Oberflächen 45

3.4.1. gut geeignete Editoren für die Verwendung mit Python 45

3.4.1.1. Sublime Text 45

3.4.1.2. Geany 46

3.4.1.3. Notepad++ 46

3.4.1.4. Komodo Edit 46

3.4.x. Eclipse 47

3.4.x. Spyder 48

3.4.x. LiClipse 49

3.4.x. Anaconda 49

3.4.x. WinPython 49

3.4.x. Komodo IDE 50

3.4.x. Thonny 50

3.4.x. SciTE 50

3.4.x. TigerJython 51

3.4.x. Editoren im Internet – online-Editoren 52

3.4.x.1. w3schools.com 52

3.4.x.2. TigerJython 52

3.4.x.3. repl.it 53

3.4.x.3. ??? 53

3.5. Snap for Python 54
Windows 54
Linux 54
MacOS 54

4. erste einfache Programme mit Python 55
4.1. Kommentare 58
4.2. Planung eines Programms und Umsetzung in Python 59

5. Was passiert mit dem Quelltext? 65
5.1. Und es geht doch! – aus dem Python-Quelltext eine EXE erstellen 67
5.2. Fehlersuche 68
5.3. Stil-Regeln für Python-Programmierer 71

6. grundlegende Sprach-Elemente von Python 73
6.1. Ausgaben 73
6.1.1. Anpassen von Zahlen für Ausgaben 77
6.1.2. formatierte Ausgaben 78
6.1.2.1. formatierte Ausgaben mit der format-Funktion 79
6.1.2.2. Verwendung von Platzhaltern in Ausgabetexten 81
6.1.2.3. Kombination von Platzhaltern und format-Funktion 82
6.2. Eingaben 84
6.2.1. unschöne Eingabe-Effekte in Python-Programmen 86
6.3. Verarbeitung 89
6.4. Kontrolle(n) 93
6.4.1. Verzweigungen 94
6.4.1.1. einfache Verzeigungen 94
einseitige Auswahl / bedingte Ausführung 94
zweiseitige Auswahl / vollständige Verzweigung 97
6.4.1.2. geschachtelte Alternativen 104
6.4.1.3. Mehrfach-Verzeigungen 107
6.4.2. Schleifen 114
6.4.2.1. bedingte Schleifen 114
Berechnung der Kreiszahl Pi mit der Methode von ACHIMEDES 118
Berechnung der Quadratwurzel von x nach der Formel von HERON 120
Berechnung der n-ten Wurzel 121
6.4.2.2. Sammlungs-bedingte Schleifen 123
6.4.2.3. Zähl-Schleifen 127
6.4.2.4. besondere Kontrollstrukturen in Schleifen 129
6.4.2.5. Und was ist mit nachprüfenden / Fuß-gesteuerten Schleifen? 132
6.5. Unterprogramme, Funktionen usw. usf. 135
6.5.1. Allgemeines zu Funktionen in Python 137
6.5.2. Funktionen ohne Rückgabewerte 138
6.5.3. echte Funktionen – Funktionen mit Rückgabewerten 140
6.5.4. Funktionen mit Funktionen als Parameter 141
6.5.4. Generator-Funktionen – Funktionswerte schrittweise 142
6.5.5. Interator-Funktionen – Funktionswerte noch wieder anders 144
6.6. Vektoren, Felder und Tabellen 146
6.6.1. Felder mit unterschiedlichen Datentypen 150
6.7. ein bisschen Statistik 151
6.7.1. Zufallszahlen 151
6.8. die Python-Schlüsselwörter im Überblick 155
Python-Spicker 161
Eingabe: 161
(formatierte) Ausgabe: 161
Verzweigung: 161

Schleifen: 161
Funktion: 162
Bibliotheken:....162
Objekt / Klasse: 162

7. Problem-Lösen mit Python 163
7.0. Aufgaben versus Probleme 163
7.0.1. Programm-Entwicklungs-Strategien 165
7.0.2. Strategien zur Lösung von (echten) Problemen....168
kleine Programm-Beispiele....174
7.0.**3,14** Python am Pi-Day 175
Pi-Berechnung durch Monte Carlo Simulation....175
Pi-Berechnung über Verhältnis der Flächen von äußeren und inneren Vieleck....175
(einfache) besondere Zahlen....176
über Teilersummen definierte besondere Zahlen....176
sonstige (ganz) besondere Zahlen....177
seltene oder ungewöhnliche Zahlen und Zahlensysteme (in der Schule)....177

8. Python für Fortgeschrittene 178
8.1. Strings – Zeichenketten....178
8.1.1. einzelne Symbole / Zeichen / Charaktere....178
8.1.2. Sequenzen von Zeichen - Zeichenketten / Strings....179
8.1.1. Objekt-orientierte Nutzung von Strings....181
8.1.2. besondere Möglichkeiten für Strings in Python....182
8.2. Datentypen und Typumwandlungen....183
8.2.1. Zahlen....185
8.2.1.1. ganze Zahlen....185
8.2.1.2. Fließkommazahlen / Gleitkommzahlen....185
8.2.1.3. Wahrheitswerte....186
8.2.2. Strings und Co als Datentypen....187
8.2.2.1. einzelne Zeichen....187
8.2.2.2. Sequenzen von Zeichen - Zeichenketten....187
8.2.3. Listen, die I. – einfache Listen....189
8.2.3.0. theortische Vorbetrachtungen....190
8.2.3.0. 1. Listen – eine Form der Datensammlung....190
8.2.3.0.2. Daten-Struktur: Liste....192
8.2.3.1. Definition und Zuweisung von Listen in Python....197
8.2.3.2. Listen-Operationen (Built-in-Operatoren)....199
8.2.3.3. Listen-Indexierung....201
8.2.3.4. Listen-Bearbeitung....203
8.2.3.5. Listen-Abschnitte (Slicing)....207
8.2.3.6. Listen-Erzeugung – fast automatisch....208
8.2.3.7. Listen - extravagant....209
erweitertes Listen-Generieren....209
erweitertes Slicing....210
8.2.3.8. Ringe – geschlossene Listen....212
8.2.3. Dictionarys - Wörterbücher....214
8.4. Interation oder Rekursion? – das ist hier die Frage! 215
8.4.1. Interation....216
8.4.1.1. typische Interations-Anwendungen....217
8.4.1.1.1. Summen-Bildung....217
8.4.1.1.2. Produkt-Bildung....219
8.4.2. Rekursion....220
8.4.2.1. Rekursions-Beispiele: Summen- und Produkt-Bildung....222
8.4.2.2. weitere typische Anwendungen für Rekursionen....224
8.4.2.2.1. Überführung einer Dezimal-Zahl in eine Dual-Zahl....224
8.4.2.2.2. die Fakultät....224
8.4.2.2.3. die FIBONACCHI-Folge....225
8.4.2.2.4. das ggT – der Größte gemeinsame Teiler....226

8.4.2.2.5. Erkennung von Palindromen ... 227
8.4.2.2.x. weitere klassische Rekursions-Probleme ... 228
8.4.2.2. direkte Gegenüberstellung von interativen und rekursiven Algorithmen ... 237
8.4.2.2.1. GGT – größter gemeinsamer Teiler ... 237
8.4.2.2.2. Palindrom-Prüfung ... 238
8.4.2.2.2. Potenz-Prüfung ... 238
8.4.3. komplexe Programmier-Aufgaben: ... 239
8.5. Umgang mit Dateien ... 241
8.5.0. Dateien und Ordner ... 241
8.5.1. Dateien lesen ... 242
8.5.1.1. Lesen von Text-Dateien ... 242
8.5.1.1.1. Lesen von CSV- bzw. strukturierten TXT-Dateien ... 242
8.3.1.1.2. Lesen von XML-Dateien ... 243
8.1.1.1.3. Lesen von JSON-Dateien ... 243
8.5.1.2. Lesen von Binär-Dateien ... 243
8.5.2. Dateien schreiben ... 244
8.5.2.1. Schreiben von Text-Dateien ... 244
8.5.2.1.1. Schreiben einer neuen Datei ... 244
8.5.2.1.2. anhängendes Schreiben ... 244
8.5.2.1.3. Schreiben von CSV- bzw. strukturierten TXT-Dateien ... 244
8.5.2.1.4. Schreiben von XML-Dateien ... 245
8.5.2.1.5. Schreiben von JSON-Dateien ... 245
8.5.2.2. Schreiben von Binär-Dateien ... 245
8.5.3. gepickelte Dateien – Dateien mit gemischten Daten ... 245
8.5.3.1. Schreiben von Dateien mit gemischten Daten ... 245
8.5.3.2. Lesen von Dateien mit gemischten Daten ... 245
8.6. Module ... 246
8.6.1. "built-in"-Funktionen ... 248
8.6.2. wichtige interne Module ... 249
8.6.2.x. Verschiedenes zum Modul: sys ... 249
8.6.2.x. Verschiedenes zum Modul: time ... 250
8.6.2.x. Verschiedenes zum Modul: os ... 251
8.6.3. externe Module installieren und nutzen ... 254
8.6.3.x. Package-Installer PIP ... 254
8.7. Graphik ... 255
8.8. Turtle-Graphik – ein Bild sagt mehr als tausend Worte ... 256
8.8.1. Turtle auf der Shell ... 256
8.8.2. Turtle-Programme und Sequenzen ... 259
8.8.3. Schleifen ... 261
8.8.4. Verzweigungen ... 263
8.8.5. Funktionen ... 265
8.8.6. Rekursion ... 268
8.8.7. Eingaben mit der Maus ... 269
8.8.8. Und wie geht es weiter? ... 270
Windrad aus Rechtecken ... 270
Parkettierung (mit Rhomben) ... 270
Zeichnen eines Strauches ... 271
Baum mit Früchten ... 272
Python-Stern ... 272
8.8.9. Turteln bis zu Umfallen - rekursive Probleme schrittweise Lösen ... 273
8.8.10. Animationen mittels turtle-Grafik ... 286
8.9. Musik mit python-sonic ... 290
8.10. das Modul "pygame" ... 291
8.10.0. Quellen und Installation ... 291
8.10.1. Ausprobieren / Testen / Grundlagen ... 292
8.10.1. Sound mit pygame ... 294
8.10.1.1. Sound-Dateien abspielen ... 294

8.10.1.2. Sound-Dateien erzeugen / aufnehmen ... 295
8.10.1.3. Musik aus dem Synthesizer ... 295
8.10.2. Grafik mit pygame ... 295
8.11. Objekt-orientierte Programmierung ... 297
Design pattern – Entwurfsmuster ... 302
8.11.x. Objekt-orientierte Programmierung mittels Turtle-Grafik ... 303
8.11.x. Klassen – selbst erstellen ... 306
8.11.x.1. Erstellen einer Klasse ... 307
8.11.x.1.1. der Konstruktor ... 307
8.11.x.2. Attribute einer Klasse ... 309
8.11.x.3. Methoden einer Klasse ... 310
8.11.x.4. Speicher-Bereinigung ... 312
8.11.x.4.1. der Destruktor ... 312
8.11.x.6. eine "Auto"-Klasse ... 320
8.11.x.6.1. Erweiterung der "Auto"-Klasse um LKW's ... 320
8.11.x.7. eine "Personen"-Klasse ... 321
8.11.x.7.1. Erweiterung der "Personen"-Klasse auf eine Familie ... 322
8.11.x.7. eine "Nachrichten"-Klasse ... 323
8.11.x.y. eine Graphik-Beispiel-Klasse ... 324
8.11.x.2. Polymophismus und Vererbung ... 329
8.11.x.y. Tips und Tricks zu Objekt-orientierten Programmen / Klassen-Definitionen ... 333
8.11.x. OOP-Programmbeispiele ... 334
8.12. GUI-Programme mit Tkinter ... 337
8.12.1. … und der erste Programmierer sprach: "Hallo Welt!" ... 339
8.12.2. Nutzung verschiedener Bedienelemente ... 340
8.12.2.1. Button's - Schaltflächen ... 341
8.12.2.1.1. eine eigene Button-Aktion erstellen ... 342
8.12.2.1.2. Button gestalten / formatieren ... 344
8.12.2.2. Nachrichten-Felder / Text-Felder ... 345
8.12.2.3. Eingabe-Felder / Eingabezeilen ... 346
8.12.2.4. Nachrichten-Boxen ... 348
8.12.2.5. Checkbutton-Wdget's – Options-Felder ... 350
8.12.2.6. Radiobutton-Widget – Options-Auswahl ... 351
8.12.2.7. Text-Fenster / Text-Widget ... 354
8.12.2.8. Frames – Group-Box's – Gruppen-Boxen ... 356
8.12.2.9. Menüs / Menu-Widget ... 357
8.12.2.9.2. eine Tool-Bar einbauen ... 359
8.12.2.9.3. eine Status-Zeile (Status-Bar) einbauen ... 360
8.12.2.10. Umgang mit Standard-Dialogen ... 361
8.12.2.11. Listbox-Widget – Auswahl-Listen – List(en)-Boxen ... 362
8.12.2.12. Options-Menüs – Auswahl-Schaltflächen ... 363
8.12.2.13. Scale-Widget – Gleiter / Regler ... 365
8.12.2.14. Scrollbar-Widget - Bildlaufleisten ... 366
8.12.2.15. Widget x ... 366
8.12.x. Tkinter – stark, stärker, noch stärker Objekt-orientiert ... 367
8.12.x.1. nochmal "Hello Welt!" ... 367
8.12.x. diverse Tkinter-Beispiele ... 372
8.13. Internet ... 373
8.13.x. Python und das http-Protokoll ... 373
Variante 1 ... 373
Variante 2 ... 373
8.13.x. einfacher Web-Server ... 375
8.13.x. Python und die eMail-Protokolle (smtp, pop3, imap) ... 376
8.14. besondere mathematische Möglichkeiten in Python ... 377
8.14.1. imaginäre Zahlen ... 377

8.14.2. Matrizen (Matrixes) ... 377
8.14.3. Python numerisch, Python für Big Data ... 379
Numpy ... 379
Scipy ... 379
Mathplotlib ... 379
Pandas ... 380
8.15. Behandlung von Laufzeitfehlern – Exception's ... 381
try … except … else ... 381
try … except … finally ... 382
try … finally ... 382
raise 382
pass 382
8.16. Sortieren – eine Wissenschaft für sich ... 383
8.16.x. Bubble-Sort ... 383
8.16.x. Quick-Sort ... 384
8.16.x. Tree-Sort ... 386
8.16.x. Merge-Sort ... 386
8.16.x. Selection-Sort ... 387
8.16.x. Insertion-Sort ... 387
8.16.x. Gnome-Sort ... 388
8.16.x. Counting-Sort ... 388
8.16.x. Radix-Sort ... 389
8.16.x. Tim-Sort ... 389
8.16.x. Heap-Sort ... 389
8.16.x. Bucket-Sort ... 390
8.16.x. -Sort ... 390
8.16.x. Vergleich ausgewählter Sortier-Algorithmen ... 391
8.16.x. das Häufigste Element finden – der Modus ... 392
Beispiel-Implementierung ... 393
8.17. Nutzung weiterer (/ besonderer) graphischer Benutzer-Oberflächen ... 394
8.18. (die hohe Kunst der) Spiele-Programmierung ... 395
8.19. Python im Geheimen - Kryptologie ... 395
8.19.0. Grundlagen ... 395
8.19.0.1. Codierung ... 395
8.19.0.2. Chiffrierung ... 395
8.19.1. symmetrische Verschlüsselung ... 397
8.19.1.x. CÄSAR-Verschlüsselung ... 397
8.19.1.x. ROT13 ... 398
8.19.1.x.1. ROT13 mit einer Funktion ... 401
8.19.1.x.2. Häufigkeits-Analyse ... 403
8.19.1.x. Umsetzung der CÄSAR-Verschlüsselung ... 405
8.19.1.x. moderne CÄSAR-Verschlüsselung mit Schlüssel ... 406
8.19.1.x. POLYBIOS-Verschlüsselung ... 407
8.19.1.x. bifid-Verschlüsselung ... 408
8.19.1.x. ADFGX-Verschlüsselung ... 410
8.19.1.x. trifid-Verschlüsselung ... 412
8.19.1.x. Four-Square-Verschlüsselung ... 416
8.19.2. asymmetrische Verschlüsselung ... 418
8.20. Code verbessern und optimieren ... 419
8.21. Test-gestütztes Programmieren mit Python ... 420
9. Python, informatisch – Datenstrukturen, Klassen, Automaten, …... 421
9.1. Tupel ... 423
9.2. Mengen ... 424
9.2.1. Mengen – einfach ... 424
9.2.1.1. Mengen-Erstellung ... 424

9.2.1.2. Mengen-Operationen ... 425
einfache Operationen ... 425
typischen Mengen-Operationen ... 427
Bearbeitung in Schleifen etc. ... 428
9.2.1.x. automatische Mengen-Generierung ... 428
9.2.2. Mengen – objektorientiert ... 429
9.2.3. Anwendung von Mengen ... 430
9.2.3.1. ein bißchen Graphen ... 430
9.3. Dictonary's - Wörterbücher ... 432
9.3.1. Erfassen von unbekannten Objekten und Zählen der Objekte in einem Wörterbuch ... 436
9.3.2. Objekt-orientierte Operationen mit Dictonary's ... 437
9.7. Listen, die II. – objektorientierte Listen ... 438
9.8. Keller ... 441
9.9. Warteschlangen ... 445
9.10. Bäume ... 448
9.11. Graphen ... 449
9.12. endliche Automaten ... 450
9.13. TURING-Automaten ... 451
9.14 ... 452
10. Python für spezielle Fälle ... 453
10.1. Python in Zusammenarbeit mit anderen Anwender-Programmen ... 453
10.2. Steuerung externer Hardware (RaspberryPi, Arduino) ... 454
10.2.1. Raspberry Pi und Verwandte ... 454
10.2.1.0. Kurzbeschreibung und allgemeine Einführung zu Raspberry Pi ... 454
10.2.1.1. die GPIO-Schnittstelle ... 454
10.2.1.2. Steuerung über die GPIO-Schnittstelle ... 456
10.2.1.3. direkte Steuerung der IO-Port ... 456
10.2.1.4. Objekt-orientiertes Programmieren ... 457
10.2.1.5. GUI mit Tkinter ... 457
10.2.1.6. programmiertes Spielen mit microsoft Minecraft ... 459
10.2.2. Aduino und Verwandte ... 471
10.2.2.0. Kurzbeschreibung und allgemeine Einführung zu Arduino ... 471
10.2.2.1. Einrichtung einer Umgebung für Programmierung eines Arduino mit Python ... 471
10.2.2.x. Spezialfall UDOO ... 474
10.2.3. FRANZIS – Experimentierplatine mit FT232R ... 474
10.2.4. TI-Innovator ... 475
10.2.4.y. externe Hardware ... 475
RGB-Array ... 475
10.3. Datenbank-Zugriff mit Python ... 478
10.3.1. SQLite 3 ... 478
10.3.1.0. Verbindung herstellen ... 478
10.3.1.1. Erstellen einer Tabelle ... 478
10.3.1.2. Hinzufügen von Datensätzen zu einer Tabelle ... 479
10.3.1.3. Aktualisieren eines Datensatzes in einer Tabelle ... 479
10.3.1.4. Löschen eines Datensatzes aus einer Tabelle ... 479
10.3.1.5. Löschen einer Tabelle ... 479
10.3.1.z. Beenden der Verbindung ... 479
weitere Beispiele: ... 480
10.4. Web-Server-Anwendungen mit dem (Micro-)Framework Flask ... 481
10.4.0. Erzeugung einer Web-Seite mit Python (Wiederholung) ... 481
10.4.1. das Framework Flask ... 481
10.4.2. die Flask-Erweiterung bootstrap ... 484

10.4.3. Programmierung der Web-Oberfläche und Darstellung von Meßwerten 484
10.5. Web-Applikationen mit Django 488
10.6. MicroPython für Microcontroller 489
10.6.x. MicroPython für microbit 492
10.6.x. MicroPython für ESP-32-Microcontroller 493
10.6.x.0. Vorbereiten des ESP für MicroPython 493
10.6.x.0.1. das Tool uPyCraft 496
Installation und Beschreibung des Hilfs-Programms uPyCraft 497
10.6.x.0.2. Nutzung eines ESP mit microPython unter Linux 503
10.6.x.0.3. Esp-Tool 505
10.6.x.1. Arbeiten mit MicroPython 508
10.6.x.y.1. interaktiver Modus - REPL 509
10.6.x.y.2. interaktiver und Internet-fähiger Modus - WebREPL 510
10.6.x.y.3. "Autostart"-Modus 511
10.6.x.4. elementare Programmierung mit MicroPython 514
10.6.x.4.1. Ausgaben 514
10.6.x.4.2. Variablen, Zuweisungen und Berechnungen 517
10.6.x.4.3. Eingaben 518
10.6.x.4.4. Alternativen, Verzweigungen 519
10.6.x.4.5. Wiederholungen, Schleifen 519
10.6.x.4.6. eingebaute und mitgelieferte Funktionen 519
10.6.x.5. klassische Programmierung mit MicroPython 519
10.6.x.5.1. Listen und Listen-Verarbeitung 519
10.6.x.5.2. Wörterbücher, Dictonary's 519
10.6.x.5.3. Lesen und Schreiben von Dateien, Datei-Verarbeitung 519
10.6.x.5.4. 520
10.6.x.6. spezielle Programmierung mit MicroPython 520
10.6.x.4. weitere spezielle Programm-Beispiele und -Schnipsel 520
10.6.x.5. spezielle Module für ESP-32-Microcontroller 532
10.6.x.5.1. Modul "machine" 532
Deep-sleep-Modus () 532
RTC (realtime clock) 532
Zähler / Timer 533
Pin's / GPIO 533
PWM (pulse width modulation) 534
ADC (analog to digital conversion) 534
SPI-Bus (serial peripheral interface) 535
I2C-Bus 536
OneWire-Treiber () 536
LED-Leisten bzw. -Ringe (NeoPixel) 537
Touch-Eingabe (capacitive touch) 537
DHT (Umweltsensoren, Temperatur-Luftfeuchte-Sensor) 538
10.6.x.5.2. Modul "esp" 538
10.6.x.5.3. Modul "esp32" 539
10.6.x.5.4. Modul "network" 539
10.6.x.5.5. Modul "time" 540
10.6.x.y. Sprach-Elemente vom MicroPython (Kurz-Übersicht / Spicker) 541
(formatierte) Ausgabe: 541
Verzweigung: 541
Schleifen: 541
10.7. Python auf und mit Taschenrechnern 542
10.7.x. Texas Instruments-Rechner 542
11. Üben, üben und nochmals üben 543
11.x. Aufgaben aus der Abiturprüfung Informatik MV 544
11.x.y. Abitur 2010 544
11.x. Aufgaben der Landesolympiade Informatik MV 544
11.x.y. 2014/2015 544
11.x.y.z. Sekundarstufe II 544

Literatur und Quellen: 545

0. Einleitung

Dieser Kurs orientiert sich weniger an den speziellen Sprach-Elementen oder informatischen Objekten, sondern mehr an einem sinnvollen Weg, erste einfache Programme zu schreiben. Nichts ist langweiliger als sich mit theoretischen Strukturen und abgesetzten informatischen Ideen zu beschäftigen. Wer programmieren lernen will, muss so schnell wie möglich, auch wirklich Programme schreiben. Programmieren – der Theorie willen – ist für akademische Kurse interessant, aber für den einfachen Einsteiger meist überfordernd. Der normale Anfänger möchte praktisch arbeiten.

Wer's anders möchte und andere Herangehensweisen bevorzugt, dem seien einige unten aufgeführte (unvollständige!!!) Tutorial's empfohlen. Jedes hat für sich Vorteile und Nachteile, Stärken und Schwächen. Wobei, wirkliche Schwächen haben wohl die wenigsten! Sie haben nur andere Konzepte und Leitlinien. Einfach mal reinschauen und prüfen, ob der Stil und die Vorgehensweise zum eigenen Anspruch passt.
Viele Anfänger wollen erst einmal nur das Programmieren lernen. Dafür reichen die Kapitel 1 bis 6 – eventuell noch 7. In diesen Kapiteln werden die grundlegenden Python-Anweisungen und –Techniken besprochen. Wer dann Geschmack an der Sache oder an Python gefunden hat, dem werden die anderen Kapitel nach und nach gefallen. Aber auch hier sollte jeder schön vorsichtig und selektiv vorgehen – besser Klasse als Masse.
Die ersten Kapitel sind sozusagen der Minimalteil dieses Skriptes, wobei auch hier schon einzelne Seiten oder kleinere Abschnitte entfallen können. Für ein ersten Programmieren reicht es in jedem Fall.
Später im Skript stehen bestimmte Python-Elementen und informatische Strukturen im Zentrum. Dabei gehen wir dann auch auf allgemeine Aspekte, Strukturen und Modelle in der Datenverarbeitung ein. Nur so werden die Feinheiten von Python deutlich, der Problemblick geschärft und einem effektiven Programmieren stehen dann alle Tore offen.
Dadurch dass viele Themen nun nochmals vertieft und erweitert behandelt werden, ergibt sich eine nicht ganz schöne Skript-Struktur. Die Alternative wären zwei Skripte gewesen, die dann aber wieder unhandlicher wären, wenn man mal was nachlesen, nachschlagen usw. usf. muss.

Dieses Skript bietet in den hinteren Teilen viele Beschäftigungs-Möglichkeiten mit Python und informatischen Sachverhalten. Ich habe versucht, die einzelnen Themen so zu besprechen, dass immer jeweils nur die Anfänger-Voraussetzungen benötigt werden. Sollte doch mal das eine oder andere Rüstzeug benötigt werden, dann wird in der Einleitung zum Kapitel oder Abschnitt darauf hingewiesen. In dem Fall empfiehlt sich eine vorherige Bearbeitung des oder der erwähnten Skript-Teile.
Ich weise an dieser Stelle noch einmal explizit darauf hin, dass sich niemand das ganze Skript ausdrucken muss, auch nicht, wenn es in einer Bildungs-Einrichtung als Arbeits-Material benutzt werden soll. Jeder kann sich aber sein persönliches Skript zusammenstellen, wenn er es dann wirklich ausgedruckt braucht. Beachten Sie aber die Lizenz-Hinweise auf der 2. Seite.
Trotz alledem wird dieses Skript nicht alle Möglichkeiten von Python darstellen können. Wenn aber etwas wichtiges fehlt, dann melden Sie sich einfach bei mir. Ich bin immer für Neues aufgeschlossen, und wenn es etwas für viele Python-Nutzer bringt, dann nehme ich es gerne in das Skript auf.

andere Tutorials, …:
http://wspiegel.de/pykurs/pykurs.htm
http://www.a-coding-project.de/python/
http://www.python-kurs.eu/
http://www.cl.uni-heidelberg.de/kurs/skripte/prog1/html/
http://www.physik.uzh.ch/lectures/informatik/python/python-start.php
http://py-tutorial-de.readthedocs.org/de/python-3.3/index.html
https://www.hdm-stuttgart.de/~maucher/Python/html/index.html

weitere Links, …:
http://cscircles.cemc.uwaterloo.ca/dev/0-de/
https://www.python-forum.de
https://www-user.tu-chemnitz.de/~hot/PYTHON/ (viele Kniffe und extravagante Beispiele)
https://docs.python.org/ (offizielle Python-Dokumentation (engl.))

aus goole-books:
https://books.google.de/books?id=oLTyCQAAQBAJ

didaktische Vorzüge von Python

kleiner Sprachumfang / wenige Konstrukte
leicht und schnell erlernbar

einfacher / nachvollziehbarer Synthax

gute Lesbarkeit des Quell-Textes
einzeilige Anweisungen

gewisse intuitive Nutzung / Programmierung möglich

zwingt zur optisch strukturierten Programmierung
notwendige Einrückungen für "Blöcke"

vorlaufende Deklarierungen sind nicht notwendig
Variablen werden dort eingeführt, wo sie gebraucht werden

manche dieser Vorzüge werden von anderen klassischen Programmiersprachen (z.B.) ebenfalls realisiert, sie sind aber selten so konsequent und in dieser Kombination vorhanden

1. Einstieg und Grundlagen

sprich peiten oder im Deutschen auch püton

universelle, interpretierende höhere Programmiersprache

derzeit von der gemeinnützigen Python Software Foundation betreutes Entwicklungsmodell
erstellt Referenz-Umsetzung, diese heißt CPython und ist die verbreiteste Version des Python-Interpreters

1.1. Geschichte und Namensgebung

in den ersten 1990iger Jahren von Guido VAN ROSSUM entwickelt

aus einem Hobby-Projekt für die Weihnachts-Ferien entstanden
Python war als Arbeitstitel gedacht

Name wurde von VAN ROSSUM als Fan und aus Verehrung der Commedy-Truppe "Monty Python's Flying Circus" gewählt

zuerst als Fortsetzung / Verbesserung der Sprache ABC entwickelt und für verteilte Rechner-Architekturen gedacht

Version 1.0 war 1994 fertig

es folgten diverse Updates

im Jahr 2000 wurde Version 2.0 veröffentlicht

die Version 3.0 (auch Python 3000 genannt) erschien 2008
ist von tiefgreifenden Änderungen, Anpassungen, Vereinheitlichungen und Optimierungen geprägt und deshalb auch nur teilweise mit der Version 2.x kompatibel

damit alte Programme (die unter Version 2.x) entwickelt wurden weiter lauffähig zu halten, wird derzeit die Versions-Serie 2.x noch weiterentwickelt und geupdated

1.2. Warum Python?

Reicht nicht eigentlich eine Programmiersprache, z.B. BASIC? Es gibt doch noch so viele andere! Muss es noch eine mehr sein? Da sieht doch nachher keiner mehr durch!
Jede Programmiersprache hat ihr Für und Wider. Die klassische BASIC-Version ist sehr einfach (zu lernen), wäre aber heutigen Programmier-Aufgaben kaum noch gewachsen. Schon beim strukturierten Programmieren schwächelt das Programm mit seinem oft getadelten GOTO-Befehl.
Viele Programmiersprachen entstanden, um spezifische Probleme mit ihnen zu lösen oder die Programmiersprache sollte speziellen (oft akademischen) Konstruktions-Regeln folgen. Heute werden noch wieder andere Kriterien bei der Bewertung einer Programmiersprache mit hinzugezogen. Der Quellcode soll offen und erweiterbar sein und natürlich erwartet man die Verfügbarkeit einer freien (kostenfreien) Version für Jedermann.
Hier sind einige ausgewählte Argumente (nach: /3, S. 21; /4, S. 18ff./, die für Python sprechen. Gegner der Sprache werden sicher genauso viele Gegenargumente finden. Dazu weiter hinten ein paar Bemerkungen.
Zuerst einmal spricht für Python der kleine Umfang reservierter Wörter (Befehle usw.).

```
False       def         if          raise
None        del         import      return
True        elif        in          try
and         else        is          while
as          except      lambda      with
assert      finally     nonlocal    yield
break       for         not
class       from        or
continue    global      pass
```

Diese rund 30 Wörter sind schnell gelernt bzw. im Blick behalten.

Python ist klein.
Python ist leicht zu lernen.

Integer-Zahlen (ganze Zahlen) können in Python beliebig groß oder auch klein werden. In anderen Sprachen muss man Umwege gehen oder externe Zusatz-Module (Bibliotheken) dazu installieren bzw. in sein Programm integrieren.

Auch ansonsten sind viele gute Merkmale und Realisierungen aus anderen Sprachen übernommen worden. Die Summe vieler guter Merkmale macht Python schon so zu einer zukunfts-weisenden Programmiersprache.
Mit PASCAL gemeinsam hat es die klare Struktur. Genauso, wie PERL kann es von sich aus mit Listen und assoziativen Feldern als ureigene Datentypen umgehen.
Weiterhin können in Python Operatoren überladen werden. Da zieht es mit C++ u.ä. Programmen gleich.

Rolle von Python in der Programmier-Welt

Python als Skriptsprache für andere Programme, z.B. OpenOffice.org, Blender, GIMP

Python-Programme lassen sich in andere (Programmier-)Sprache einbauen

mit Python lässt sich auf Datenbanken zugreifen (Nutzung von SQL in Python)

andere Programmiersprachen lassen sich in Python-Skripten verwenden
CGI-Programmierung

sehr flexible – weil nicht Typ-gebundene – Programmierung löst viele allgemeine Probleme

unterstütze Programmier-Paradigmen in Python:

- **imperativ / prozedural**
  charakterisiert durch einfachen Code
  gut für die Manipulation von Daten
- **funktional**
  alle Aussagen werden als mathematische Gleichungen betrachtet
  dieses Paradigma ist eine gute Basis für
  geht in Richtung Rekursion und Lambda-Kalkül
- **Aspekt-orientiert**
- **Objekt-orientiert**
  Daten werden als Objekte mit Eigenschaften (Attributen) gesehen
  Veränderungen werden über Methoden vorgenommen
  Code ist i.A. gut wiederverwendbar
- **Verfahrens-orientiert**
  Aufgaben werden Schritt für Schritt als Interationenabgearbeitet
  häufige Operationen werden in Unterprogramme, Prozeduren bzw. Funktionen abgelegt
  gegünstigt Squenzierungen, Iterationen, Auswahl und Modularisierung

Python ist somit eine Multi-Paradigmen-Sprache (multi-paradigm language)

Vorteilhaft ist die automatische Speicher-Bereinigung (garbage collection) am Ende der Python-Nutzung. Schon innerhalb der Programmnutzung werden die nicht mehr gebrauchten und irgendwo neu definierten Programmier-Objekte aus dem Speicher entsorgt.
automatische Daten-Müllvermeidung

Python hat von sich aus weniger strenge Programmier-Kontrollen. Dem Programmierer gibt das einige zusätzliche Freiheiten.
Blöcke werden in Python nicht – wie sonst häufig üblich – in BEGIN-END-Blöcke (oder geschweifte Klammer etc.) notiert. Die Blockbildung erfolgt einfach durch Einrückung – also strukturiertes Schreiben des Quelltextes. Eine Empfehlung anderer Programmiersprachen wird hier zum Prinzip erhoben.
Die Schleifen-Konstrukte sind auf die Wiederholungs-Schleife und die vorprüfende Schleife eingeschränkt. Das reicht völlig aus und ist auch besser zu durchschauen. Natürlich lässt sich leicht eine nachlaufend-prüfende Schleife zusammenstellen. Aber wir werden sehen, man kann prinzipiell mit nur einem Schleifentyp alle anderen simulieren / ersetzen – auch wenn es nicht immer schön aussieht.
Der – von einer anderen Sprache – wechselnde Programmierer wird bei dem Begriff der Wiederholungs-Schleife aufhorchen. Heißen die nicht eigentlich Zähl-Schleifen. Nein hier

sind wirklich Wiederholungs-Schleifen gemeint, die nicht durch eine vorbestimmte Zahl an Durchläufen beschränkt ist.

Konstanten und Variablen müssen nicht vor der Benutzung deklariert (bekanntgegeben) werden. Sie werden einfach benutzt. Echte Konstanten sind nicht im Konzept enthalten. Lediglich pi und e sind definiert.

Die Datentypen werden locker gehandhabt. Prozeduren können mit unterschiedlichen Datentypen aufgerufen werden. Für den Programmierer ist es eher lästig z.B. eine Addition für Ganzzahlen und für (Gleit-)Kommazahlen zu schreiben. Einmal definiert funktioniert sie für beide Datentypen.
Einen exklusiven Datentyp für Wahrheitswerte gibt es in Python nicht. Jedem Zahlen- oder Zeichenketten-Wert wird ein Wahrheitswert zugeordnet. Damit entfällt die sonst notwendige Um-Rechnung bzw. Um-Deutung.
Für jede Anweisung wird in Python eine eigene Zeile benutzt. Dadurch werden Programme übersichtlicher, aber leider auch (Seiten-)länger. Das fehlende Semikolon am Ende jeder Programmier-Anweisung in PASCAL ist der überwiegende Anfängerfehler für Programmier-Starter.

Ausnahme-Behandlung

verteilte Objekte (CORBA, ILU, COM (Component-Object-Model))

Netz-Protokolle

kann mit Threads und Prozessen umgehen

funktionale Programmierung

Integration externer (C-)Bibliotheken

Python ist sehr flexibel.
Python vereint die Vorteile vieler Programmiersprachen in sich.
Python macht einfach Spaß.

Schwächen
gestandene Programmierer finden so manche gewohnte Programmier-Struktur nicht in Python wieder

wie gerade erwähnt machen notwendige Einrückungen und Einbefehls-Zeilen den Quelltext aufgebauscht und auch ein bisschen unübersichtlicher und vor allem lang. Da gehen andere Empfehlungen Funktionen, Unterprogramme usw. nur eine Seite lang zu machen – ein wenig in die Leere.

teilweise Probleme mit Multithreading – also dem parallelen Abarbeiten von mehreren Programmen auf einem Mehrkern-Prozessor
so etwas gehört heute zur modernen Programmierung einfach dazu

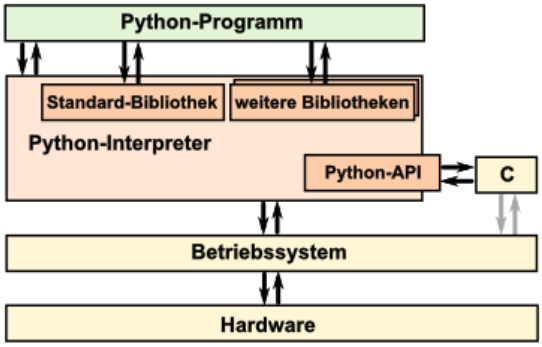
relativ langsam im Vergleich zu anderen Skript- bzw. Interpreter-Sprachen

nicht ganz übliche / moderne Programm-Strukturen, wenn objektorientiert programmiert werden soll

Python verleitet zur und unterstützt Trick-Programmierung.
Python ist in der Grundausstattung unhandlich.
Python ist relativ langsam.
Python hat so seine speziellen Strukturen.

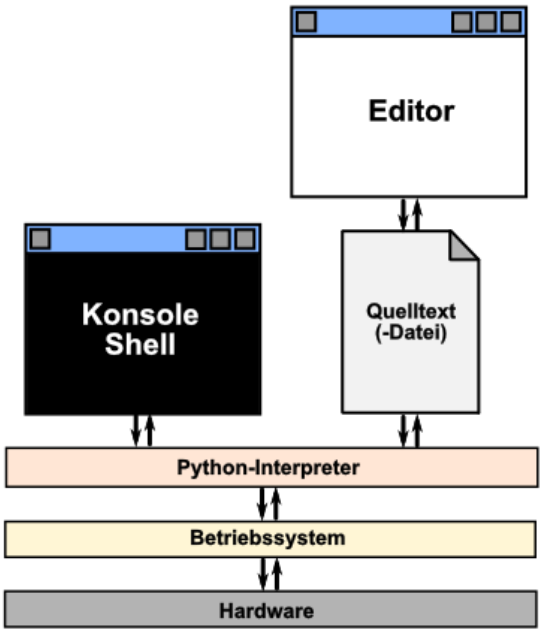
grundlegende Python-Konzepte

Ein Python-Programm wird vom Interpreter ausgeführt. Es kann dabei neben den internen Funktionen auch auf solche aus verschiedenen Bibliotheken zugreifen. Die Bibliotheken beinhalten dabei häufig gebrauchte und allgemeine Funktionen. Man kann sich dsie Bibliotheken als Erweiterungen von Python vorstellen. Der Python-Interpreter greift – je nach auszuführenden Programm – auf verschiedene Teile (Schnittstellen, Funktionen) des Betriebssystem zurück. Das Betriebssystem bedient dann wieder die Hardware. Im Normalfall kapselt das Betriebssystem vom Programmiersystem ab. Dadurch sind keine direkten Hardware-Manipulationen möglich.



Python-Konzept
Q: geänd. aus /7, S. 33/

Python-Programme sind dadurch aber universell auf verschiedenen Geräten (Hardware) und Betriebssystemen lauffähig.



Zen of Python

```
Python 2.7.10 (default, May 29 2015, 22:02:48)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2. Vorbereitung (Installation)

Download der Version, die zum genutzten Rechner-Typ und / oder Betriebssystem passt.
Die PythonSoftware Foundation bietet die verschiedenen Versionen über die Webseite www.python.org an.
Unter Downloads finden Sie die klassischen Betriebssystem-Varianten und Versionen für Betriebssysteme, von denen die meisten Computer-Nutzer noch nie etwas gehört haben. Da die meisten verfügbaren Versionen untereinander kompatibel sind, können einmal entwickelte Programme auch auf völlig anderen Betriebssystemen genutzt werden. Und alle normalen Python-Versionen sind frei und kostenlos.
Neben der originalen Pythen-Version gibt es auch andere Umsetzungen der Sprache. D.h. die Systeme sprechen das gleiche Python, aber die Arbeits- und / oder Umsetzungs-Programme sind in anderen Programmiersprachen geschrieben. Ein Beispiel dafür ist Jython. Es ist ein Python, dass auf einer JAVA-Umsetzung basiert. Dieses Programmier-System stellen wir später bei den IDE's ausführlicher vor.

2.1. Python auf Windows-Rechnern

Als freie Programmiersprache einschließlich einer einfachen graphischen Benutzer-Oberfläche (IDE) ist Python für jederman verfügbar.
Zu empfehlen ist immer eine lokale Installation auf dem Arbeitsrechner. Das ist der Standard. So wird es nur wenige Probleme geben und die Arbeit geht flott.
Man benötigt allerdings Administrator-Rechte. Wer diese nicht hat und auch nicht bekommen kann, muss auf eine portable Version ausweichen. Hier wird dann nichts installiert.
Meist ist dann allerdings keine Zuordnung der Datei-Typen (.py, .pyc und .pyo) vorhanden. Man muss dann die Dateien über das "Datei" "Öffnen" aufrufen. Ein Doppel-Klick auf die betreffenden Dateien funktioniert nicht. In WinPython gibt es aber eine Funktion, die die fehlenden Zuordnungen realisiert. Eine echte Installation auf dem Arbeitsrechner ist das aber nicht.
Den Download und einige Hinweise zum portablen Charakter von WinPython findet man unter → https://winpython.github.io/#portable. Weiter hinten (→ 3.4.x. WinPython) gehen wir dann auch noch auf einige Besonderheiten von WinPython ein.
Eine weitere – und wohl auch die ursprüngliche – Realisierung eines portablen Python's ist portablePython.com. Unter portablePython.com findet man leider nur noch (versteckt) eine ältere Version (→ http://portablepython.com/wiki/Download/). Das Projekt wird wohl nicht mehr fortgesetzt.
Die portablen Versionen können separat oder in spezielle Menü-Systeme (PortableApps-com, IoStick, …) auf einem USB-Stick kopiert werden. Mit dem USB-Stick kann man dann an beliebigen Windows-Rechnern arbeiten und hat auch seine Daten immer mit dabei (natürlich nur, wenn man sie auch auf dem Stick speichert).
Bei SourceForge.net gibt es Seite mit einer sehr aktuellen Version eines Portablen Python's (→ https://sourceforge.net/projects/portable-python/). Die Version muss heruntergeladen und entpackt werden. Anschließend lässt sich der entpackte Ordner auf den Io-Stick oder einen PortableApps-Stick kopieren. Das portable Apps-System erkennt die neuen Programme automatisch. Wer will, kann sie sich in die Kategorie "Entwicklung" verschieben.
Beim PStart-Menü des IoStick's ist das nicht so einfach.
Bitte vorm Editieren die alte Datei (PStart.xml) zusätzlich als PStart.org kopieren. Dann kann man diese später wieder reaktivieren.

Ins Menü habe ich nur den IDLE- und den PyScripter-Starter aufgenommen. Das sollte für schulische Zwecke reichen.

2.2. Python auf Linux-Rechnern

praktisch eigentlich immer dabei
gehört zur guten Ausstattung einer Linux-Distribution

2.3. Python auf dem Raspberry Pi

Standard-Programmiersprache im Raspberry Pi

also sofort nutzbar

vollständige Implementierung

besonders interessant für Steuerungs- und Sensorik-Aufgaben
da viele Schnittstellen relativ leicht zugänglich sind, von denen der Raspberry Pi auch sehr viele bietet
desweiteren sind diverse Ergänzungen (Zusatz-Boards, Sensoren, …) verfügbar

weiter hinten spezielle Möglichkeiten (→ 10.1. Steuerung der Hardware (RaspberryPi, Arduino))

Auf dem Raspberry Pi gibt es in einigen Linux-Distributionen das Spiel "Minecraft" von microsoft in einer kostenfreien Version. Diese Version lässt sich auch mittels Python programmieren. Einige Möglichkeiten stellen wir weiter hinten vor (→). Da die Python-Programme praktisch recht einfach sind, bieten sich viele Möglichkeiten für das Experimentieren, Spielen und Spaß-Haben.

Weiterhin gibt es auch eine Realisierung von Jython (TigerJython →) für den Raspberry Pi.

2.5. Python auf Android-Systemen

2.5.1. Pydroid3 IDE

frei nutzbar

typisch ist die etwas gewöhnungs-bedürftige Touch-Tastatur (zumindestens für Programmierer mit Real-Tastatur-Feeling)
Speichern - und später auch das Öffnen funktioniert über das Ordner-Symbol. Hier muss man dann für das Speichern einen geeigneten Platz suchen. Ich habe den "internen Spei-

cher" ("InternalStorage") und dort meinen "Dokumenten"-Ordner ("Documents") ausgewählt und hier einen "Python"-Ordner angelegt. Wenn man sich darin befindet, dann kann mittels "Ordner benutzen" eine Quellcode-Datei angelegt werden.
Das Ausführen funktioniert über das Dreick-Symbol an der Status-Zeile unten. Es öffnet sich die Python-Konsole und das Programm läuft ab.

Wer gleich auf der Python-Konsole arbeiten möchte, kann das über das -Menü erledigen. Dort gibt es in der Rubrik "Run" einen entsprechenden Punkt.

Für das einfache Komprimieren des Ordners (oder einzelner Dateien) für den Versand nutze ich ZAchiver von ZDevs. Dort muss man nur den passenden Ordner auswählen, länger auf ein Odner-Symbol drücken und dann aus dem Kontext-Menü "Komprimieren zu *.zip" auswählen und fertig.

2.5. Python auf dem MacOS

Python3IDE

benötigt neue(ste) OS-Version

für einfache Programmier-Versuche unterwegs ausreichend

2.6. Python auf dem Taschenrechner

für den Casio FX-CG50 – ein Taschenrechner – steht neuerdings auch eine Python-App zur Verfügung
Umsetzung von Micro-Python

da Groß- und Klein-Schreibung in Python unterscjhieden wird, muss viel in Kleinbuchstaben geschrieben werden
ein längerfristige Umschaltung ist mit [SHIFT] [ALPHA] A ←→ a
(Standard wieder herstellen mit einem weiteren [ALPHA]

für mathematische Aufgaben muss die Math-Bibliothek geladen werden
from math import *
ist im Katalog schon vordefiniert, Aufruf mit [SHIFT] 4

jetzt auch für den TI-Innovator
extra Abschnitt, weil er als eher Microcontroller ein deutlich anderes Leistungs-Spektrum hat (→ 10.2.4. TI-Innovator)
es können TI-Sensoren und -Aktoren, Grove-Sensoren und –Aktoren sowie auch andere elektronische Schaltungen (z.B.: via Steckbrett) angeschlossen werden

in Frankreich ist Python verpflichtend im Unterricht, deshalb hier auch extra Material verfügbar

→ TI-83 Premium CE Edition Python

https://online.flipbuilder.com/wera/jstv/ (Vorschau: Büchlein zur Python-Programmierung im naturwissenschaftlichen Unterricht (franz.))

http://online.flipbuilder.com/wera/tvxt/ (Vorschau: Büchlein zur Python-Programmierung mit TI-83 (franz.))

2.7. Python auf Microcontrollern

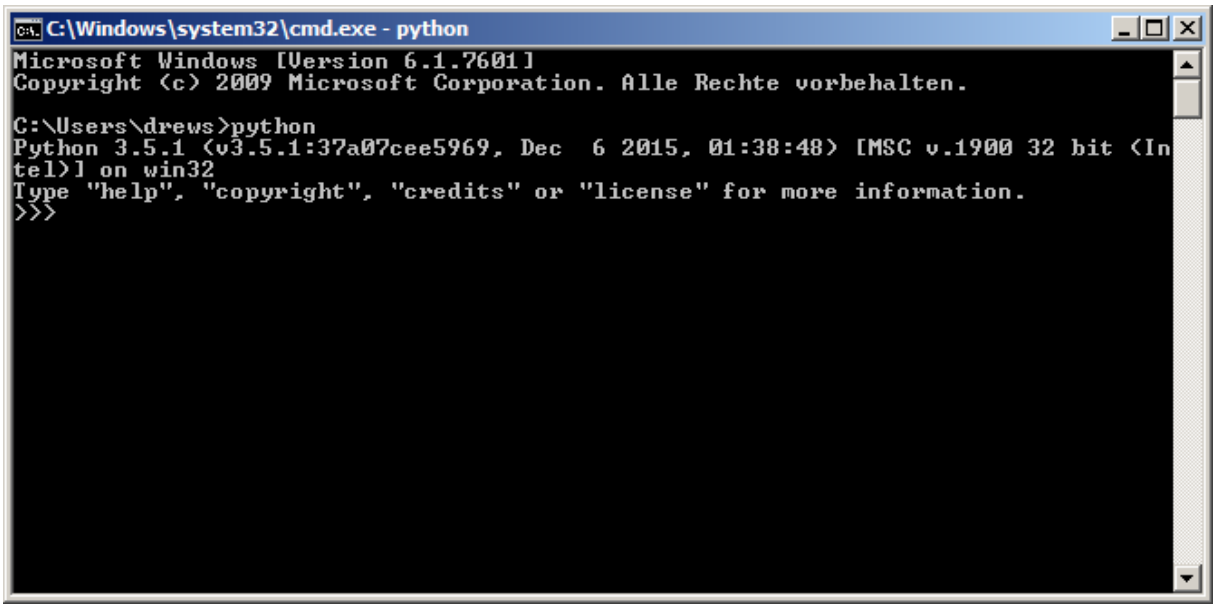
Was vor wenigen Jahren undenkbar war, ist mit der superschnellen Entwicklung von Microcontrollern Wirklichkeit geworden. Python lässt sich zur einfachen Programmierung dieser Geräte-Klasse verwenden. Voraussetzung ist allerdigs ausreichend RAM auf den Bausteinen. Hier liegt meist das Problem. Die Dinger haben einfach zu wenig. Deshalb ging es mit Arduino und Co auch noch nicht. Aber mit der breiten Verfügbarkeit und den ungemein günstigen Preisen von neuen Microcontrollern steht dem Einsatz von Python - allerdings in einer abgespeckten Version – nicht mehr viel im Wege.
Da bei MicroPython (µPython) doch einiges sehr speziell ist, besprechen wir den Einsatz auch erst weiter hinten als extra Kapitel (→ 10.6. MicroPython für Microcontroller). Einsteiger sollten sich erst mit dem "normalen" Python auseinandersetzen und dann später auf diesen modernen Zug (IoT, Automatisierung, …) aufspringen.

3. Zugriff auf das Python-System

3.1. die Python-Shell

Aufruf über das "Start"-Menü oder über eine "Eingabeaufforderung"
das Startmenü ist ev. schnell durchsucht und der passende Menüpunkt gefunden.

Eine Eingabeaufforderung erhält man ebenfalls direkt über einen entsprechenden Menü-Eintrag im "Start"-Menü bzw. – je nach Windows-Version über "Ausführen …" oder "Suchen …" im Start-Menü. Dort gibt man "cmd" ein und bestätigt mit [Enter]. Die Freunde der altbewärten Tastatur-Kürzel benutzen die Kombination [⊞] + [R].



der Phyton-Interprter in einer Eingabeaufforderung von Windows

Die Eingabeaufforderung ist quasi eine Rudiment aus alten DOS-Zeiten. Damals mussten alle Befehle oder Programm über die sogenannte Befehlszeile – oder auch Prompt genannt – gestartet werden. Die Ausgaben der gestarteten Programme sahen meist nicht besser aus.
Der Start-Befehl für den Python-Interpreter lautet "**python**".
Das Python-System ist nun im klassischen Kommandozeilen-Modus (CLI .. command line; Command Line Interpreter) gestartet. Auch wenn wir ein typisches Windows-fester sehen, es handelt sich um ein DOS-ähnliches Konsolen-Programm ohne eigene Fenster-Funktionen.
Das Python-System meldet sich mit einer kurzen Versions-Anzeige und einem eigenem Prompt. Diese besteht aus drei "Größer als"-Zeichen(">>>"). Jede Eingabe bzw. die fertige Befehlszeile muss mit [Enter] zur Ausführung gebracht werden. Fehlende Angaben oder fehlerhaftes Schreiben quittiert die Eingabeaufforderung mit einer Fehlermeldung. Natürlich kann auch ein nicht gewollter Befehl ausgeführt werden. Die Befehle sind sehr mächtig. Also vorsicht und lieber einmal genauer prüfen, was man dort eingetippt hat. Auf der Ebene der Eingabeaufforderung gibt es kein "Rückgängig" oder einen "Papierkorb". Da lässt sich sich nichts rückgängig machen oder wieder hervorzaubern!
Bevor wir richtig durchstarten noch einige Bemerkungen zum Verlassen bzw. Beenden der Shell. Zum Einen steht uns eine Funktion dafür zur Verfügung. Hinter dem Prompt geben wir einfach **exit()** ein und die Shell wird nach dem obligatorischen [Enter] geschlossen. Auch ein

quit() führt zum gleichen Ziel. Alternativ kann man die Tasten-Kombination [Strg] + [Z] (oder [Strg] + [Q] oder auch, wie bei jedem Fenster [Alt] + [F4]) benutzen.
Vergisst man das Klammerpaar hinter den Befehlen exit bzw. quit, dann erhält man den freundlichen Hinweis, wie die Shell bzw. IDLE ordnungsgemäß geschlossen wird. Danach befindet man sich wieder auf der Konsolen-Ebene (Eingabeaufforderung) von Windows. Man erkennt dieses am einfachen "Größer als"-Zeichen - dem Standard-Prompt der Eingabeaufforderung.
Auf der Konsolen-Ebene lassen sich kleine Skripte abarbeiten. Jeder Befehl, jede Zeile bzw. jeder Befehls-Block muss allerdings einzeln eingegeben werden. Die Befehls-Eingaben eines Skriptes lassen sich auch nicht abspeichern. D.h. bei einer erneuten Anwendung müssen wieder alle Anweisungen erneut eingegeben werden.
Es kommt also zu einem ständigen Wechsel zwischen Eingabe und Ausgabe. Der Nutzer interagiert mit dem System. Wir sprechen auch vom interaktiven Modus.

3.1.1. Eingaben an der Shell

Die Shell ist quasi die Schnittstelle, um Befehle direkt an den Computer abzugeben. Die Anweisungen werden in einer höheren Programmiersprache – hier eben Python – formuliert und eingegeben. Die Shell übernimmt sie und übergibt sie dem Übersetzer, damit dieser sie in Maschinen-Code – also reine Nullen und Einsen – umwandelt. Die Nullen und Einsen sind die einzigen Arbeits-Anweisungen, die eine Computer versteht. Geht irgendetwas bei der Eingabe oder beim Übersetzen schief, dann erhalten wir eine Fehlermeldung. In dem Fall, dass alles ok ist, erledigt der Computer die befohlene Aufgabe – zumindestens so wie er sie "verstanden hat".

3.1.2. IDLE als Python-Konsole

Statt der Windows-Eingabeaufforderung kann auch gleich das Programm IDLE benutzt werden. Es wird mit Python ausgeliefert und installiert. Es ist zumindestens erst einmal auch eine Konsole.
Der große Vorteil von IDLE ist, dass wir später von hier schnell in die Programmier-Ebene hineingelangen. Die brauchen wir, um länge Anweisungs-Sequenzen abzuspeichern. IDLE ist dann auch gleich nocht ein Programm-Starter. Damit können wir gespeicherte Anweisungs-Sequenzen starten / laufen lassen. Bei IDLE handelt es sich um also um einen sehr einfachen Programm-Editor und einen Programm-Starter.
Echte Viel-Programmierer nutzen spezielle Oberflächen (GUI's → 3.4. Nutzung anderer Benutzer-Oberflächen), die neben den Editor- und Start-Funktionen noch spezielle Unterstützungen anbieten. Für Programm-Einsteiger sind sie aber erst einmal nicht notwendig.

Aufgaben:

1. *Starten Sie eine Python-Shell (z.B. IDLE)!*
2. *Lassen Sie sich die folgenden Ausdrücke berechnen? Wird eigentlich mit den in der Mathematik üblichen Vorrangregeln gearbeitet? Was bedeuten diese Zeichen?:* `*  /  .  ,  //  **`

a)	`4 * 12`	b)	`16 / 8`
c)	`34 + 21 - 54 * (10 + 5)`	d)	`10 + 2 * 10 + 5`
e)	`12 / 5`	f)	`5 / 2 / 3`
g)	`12 * 0.25`	h)	`12 * 1,5`
i)	`12 // 5`	j)	`2 ** 3`
k)	`120/4*10/5/2`	l)	`20 * 0.5 * (((13 - 3) + 10) * 2)`
m)	`3**3+3**4-3**2`	n)	`20 * 0.5 * {[(13 - 3) + 10] * 2}`

3. *Wenn sich i) und j) – oder auch andere Teilaufgaben – nicht so einfach für Sie erschließen, dann variieren Sie einfach die Zahlen in kleinen Schritten!*
4. *Versuchen Sie die Aufgabe k) mit Leerzeichen zwischen den Zahlen und Operatoren aus! Welches Ergebnis erhalten Sie nun? Welche Variante empfinden Sie für besser?*
5. *Versuchen Sie zu erklären, warum l) und n) – obwohl sie doch scheinbar mathematisch gleichwertig sind – zu unterschiedlichen Ergebnissen / Ausgaben führen! Welche Schlussfolgerung muss man hier für die weitere Arbeit mit Python ziehen?*

3.1.2. fortgeschrittene Mathematik

Verfügbarmachen für die Verwendung in der Konsole bzw. im selbstgeschriebenem Programm

from math import *	alle Funktionen aus math importieren
import math	alle Funktionen aus math importieren die Funktionen können nur mit vorgesetztem Modulnamen benutzt werden → math.sqrt(…)
import math as M	alle Funktionen aus math werden importiert und dem Namen M zugeordnet → Aufruf über M.sqrt(…) möglich
from math import sqrt	nur die Funktion sqrt (Quadrat-Wurzel) importieren die Funktion kann unter dem Namen direkt benutzt werden
from math import pi	nur die Konstante aus dem Modul math importieren
from math import pi as PIEH	nur die Konstante aus dem Modul math importieren und sie unter dem Namen PIEH zur Verfügung stellen

help(math) bzw. help(M), wenn ein Import unter einem anderen Namen erfolgt (hier: M)
um sich z:B. die Beschreibung / Hilfe zu den Funktionen und Definitionen (hier: e und π) anzusehen

Aufgaben:

1. ***Finden Sie mit der Hilfe zum Modul math heraus, wie die Konstanten e und pi (für π) genau definiert sind!***
2. ***Was macht z.B. die Funktion gcd(…)?***

3.1.3. mehrzeilige Eingaben an der Shell

Im Augenblick sind unsere Eingaben an der Shell noch überschaubar. Sollen aber auf dieser Ebene komplexere Dinge gemacht werden, dann kommt man um mehrzeilige Eingaben nicht herum.

Geben Sie das nebenstehende Beispiel – wie angezeigt – ein! Achten Sie auf den Doppelpunkt am Ende der ersten Zeile! Jede Zeile wird wie üblich mit **[ENTER]** abgeschlossen.

```
>>> for i in range(10):
        i
        i*i
```

Die Einrückungen für die 2. und 3. Zeile werden dann automatisch vorgenommen. Geben Sie dann einmal zusätzlich ein [ENTER] ein, dann folgt eine mehrzeilige Aufgabe.
Alle drei Zeilen werden jetzt in einem Komplex abgearbeitet. Was auch immer das bedeutet, auch sehr komplexe Aufgaben sind schon ander Shell realisierbar. Deshalb lieben viele Administratoren Python auch so – es ist einfach und effektiv.
Die Farbigkeit der einzelnen Zeilen-Teile erklären wir später (→). Die besprochene Struktur setzen wir ab und zu mit hellgelben Hintergrund, um die entscheidende Stelle schneller zu finden.

Sehr lange Zeilen können durch einen **Backslash** (\) – den umgekehrten Schrägstrich auf der ß-Taste – quasi abgebrochen und auf der nächsten Zeile fortgesetzt werden. Es wird wieder automatisch eine Einrückung zur Kennzeichnung der Zusammengehörigkeit gemacht.

```
>>> print("langer Text"+\
          "weiterer langer Text")
```

Bei Texten sollte man immer die Text-Begrenzer mit schreiben, ansonsten muss mit zusätzlichen Leerzeichen durch den Zeilenbruch gerechnet werden.
Der Backslash kann entfallen, wenn man Ausdrücke benutzt, die Klammern enthalten. Dann müssen die Zeilen allerdings auch mit [⇧] + [ENTER] umgebrochen werden (weicher Zeilenumbruch). Die endgültige Bestätigung und Ausführung erfolgt dann erst nach einem echten [ENTER].

In echten Quelltexten wird diese Notation gerne bei Funktionen mit komplizierten oder vielen Argumenten benutzt. Man schreibt jedes Argument in eine neue Zeile und kann diese dann auch schön kommentieren.

```
…
y=testfunktion(
        0,    # Anfangswert
        100,  # Endwert
        0.5)  # Schrittweite
…
```

Aufgaben:

1. *Probieren Sie das obige Beispiel in der Konsole aus! Können Sie die Ausgaben erklären?*
2. *Wandeln Sie das Miniprogramm so ab, dass die Variable d benutzt wird und die 2. eingerückte Zeile: d*"d" lautet! Was wird dieses Programm machen? Probieren Sie es aus?*
3.

3.1.4. mehrere Befehle in eine Zeile

Die Notierung mehrerer Befehle in eine Zeile ist eigentlich nicht Python-like. Jeder Befehl bzw. jede Anweisung sollte in einer extra Zeile stehen.
U.U. werden Quell-Texte so besonders lang oder unübersichtlich. Oftmals sind die Anweisungen so zusammengehörend, dass sie schon wie eine Anweisung wirken.

In solchen Fällen kann man Anweisungen eines Blocks / einer Gruppe auch **Semikolon**-getrennt (**;**) notieren.

```
>>> print("Hallo ");print("Welt!")
```

Anweisungen, die auf Doppelpunkte folgen – also z.B. nach Einleitungen von Verzweigungen oder Schleifen – können ebenfalls in der gleichen Zeile weitergeschrieben werden.

Bei Schleifen oder Verzweigungen die später erweitert werden sollen oder könnten, sollte man diese Notation unbedingt vermeiden.

```
>>>
```

Aufgaben:

1. ***Welche Ausgabe erwarten Sie beim obigen Einzeiler (2x print)? Probieren Sie den Einzeiler aus! Wenn Ihre Voraussage nicht eingetroffen ist, erklären Sie die veränderte Ausgabe!***
2. ***Wandeln Sie das unten angegebene Programm in den drei Abschnitten in Einzeiler für die Konsole um und lassen Sie diese Zeile dann ausführen!***

```
1   a="#"
    print(a)
2   for i in range(5,10):
        print(i)
        a=a+"*"
        print(a)
3   print(i)
    print(a)
    print(a+a)
```

3.

3.1.2.1. Mathematik für Informatiker – binäres Rechnen

Na gut, eigentlich ist die Überschrift etwas hochgestapelt, aber irgendwie ist doch wieder passend.
Im Computer werden die Zahlen im Binär-Code abgelegt. Dies gilt ganz besonders für die ganzen Zahlen. In einer ersten Überlegung nehmen wir uns nur die natürlichen Zahlen vor, und tun so, als würde es keine negativen geben. In Python gibt es keinen direkt dazu passenden Datentyp. In PASCAL ist es z.B. der Datentyp **Byte**, der Zahlen von 0 bis 255 darstellen kann. Für etwas größere Zahlen (0 bis 65535) gibt es dann noch den Typ **Word**.
In diesen Datentypen werden den 8 Bits Binär-Werte zugeordnet, praktisch äquivalent zum dezimalen Zahlensystem.

Basis: 2 (binäres, duales Zahlen-System)

Position	7	6	5	4	3	2	1	0
Potenz	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Positions-Wert	128	64	32	16	8	4	2	1

An jeder Bit-Stelle kann nun das Bit gesetzt sein – also eine 1 beinhalten – oder eben 0 sein.
Die resultierende Zahl (im Dezimal-System) ist dann die Summe der Bit-gesetzten Positionswerte.
Ähnlich, wie im Dezimal-System, wo Multiplikationen und Divisionen mit 10 sehr einfach sind, so sind im Binär-System genau diese Berechnungen mit der 2 sehr schnell realisierbar.
Dazu verwendet man sogenannte Schiebe-Befehle.

Beispiel	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**
Potenz	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Positions-Wert	128	64	32	16	8	4	2	1
Anwendung	128	0	0	16	8	0	2	1

Beispiel **= 155**

Diese Befehle beziehen sich auch direkte Bit-Verschiebungen in den sogenannten Registern der CPU, in der die Zahlen zur Verarbeitung zwischengespeichert werden. Nehmen wir uns ein sehr einfaches Beispiel – die Berechnungen mit der Zahl 16.
Durch eine Links-Verschiebung kommt es praktisch zur Verdopplung (Multiplikation mit 2) der codierten Zahl.

16 =	0	0	0	**1**	0	0	0	0
Links-Verschiebung	0	0	**1**	0	0	0	0	0

16 * 2= **= 32**

Neue Bits – hier also auf der rechten Seite – werden mit 0 gefüllt.
Nimmt man dagegen eine Rechts-Verschiebung der Bits vor, dann kommt es zur Division durch 2 (Halbierung).

16 =	0	0	0	**1**	0	0	0	0
Rechts-Verschiebung	0	0	0	0	**1**	0	0	0

16 / 2= **= 8**

Hier werden auf der linken Seite 0-Bits aufgefüllt.
Die Schiebe-Befehle sind direkt im Maschinen-Programm (Mikro-Code) der CPU realisiert und deshalb besonders bei sehr großen Zahlen mit vielen Bits sehr effektiv. Typische Registerbreiten heutiger CPU's liegen bei 32, 64 und 128 Bit.

Aufgaben:

1. *Codieren Sie die dezimale Zahl 4952 im Binär-System. Welche Art von CPU (8, 16, 32 od. 64 Bit Registerbreite) wäre optimal?*
2. *Führen Sie 3 Rechts-Verschiebungen durch! Welcher Multiplikation entspricht dies? Prüfen Sie durch Rückcodieren der Binärzahl in das Dezimal-System, ob die Berechnung exakt ist!*
3. *Führen Sie mit der Zahl 4952 nun 2 Links-Verschiebungen durch! Prüfen Sie auch hier auf Exaktheit der Berechnung! Erklären Sie das auftretende Phänomen! Um welche Art Division handelt es sich also praktisch?*

Nun zur Realisierung in Python. Schiebe-Operationen werden mit doppelten Kleiner- bzw. Größer-Zeichen (<< bzw. >>) umgesetzt. Hinter den Winkel-Zeichen folgt die Anzahl der Verschiebungen.
Somit wäre eine Multiplikation mit Vielfachen von 2 z.B. so möglich:

```
>>> 864<<4
13824
>>>
```

bedeutet: 864 * (2 * 2 *2 * 2) = 864 * 16
oder: 864 * 2 * 2 * 2 * 2

Die dreifache Teilung durch 2 erfolgt dann z.B. so:

```
>>> 864>>3
108
>>>
```

bedeutet: 864 / (2 * 2 * 2) = 864 * 16
oder: 864 / 2 / 2 / 2

In den Python 3-Versionen werden beliebig große ganze Zahlen berechnet.
In älteren Versionen muss dies nicht zwangsläufig auch so erfolgen. Vielfach sind noch Varianten mit der klassischen Umsetzung des Minus-Vorzeichens im Umlauf. Dabei wir der höchste Bit-Wert nicht als entsprechender Wert genutzt, sondern als Kennzeichnung des negativen Vorzeichens. Bei Verwendung von Schiebe-Befehlen und auch anderen Rechnungen mit verschiedenen Zahlen sollte man deshalb auch immer die Typ-Grenzen mit austesten.
Zu beachten ist bei der genauen Betrachtung der Werte-Belegung, das die Nicht-Vorzeichen-Stellen (Magnitude) nicht die Zahl darstellen, sondern deren 2er-Komplement – also die Bit-Vertauschung!

Zahlentyp shortint (aus PASCAL)

Position	7	6	5	4	3	2	1	0
Potenz	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	0	0	0	0	1
2 =	0	0	0	0	0	0	1	0
3 =	0	0	0	0	0	0	1	1
4 =	0	0	0	0	0	1	0	0
5 =	0	0	0	0	0	1	0	1
6 =	0	0	0	0	0	1	1	0
7 =	0	0	0	0	0	1	1	1
	. . .							
125 =	0	0	0	0	0	0	0	1
126 =	0	1	1	1	1	1	1	0
127 =	0	1	1	1	1	1	1	1
-127 =	1	0	0	0	0	0	0	0
-126 =	1	0	0	0	0	0	0	1
-125 =	1	0	0	0	0	0	1	0
-124 =	1	0	0	0	0	0	1	1
	. . .							
-3 =	1	1	1	1	1	1	0	0
-2 =	1	1	1	0	1	1	0	1
-1 =	1	1	1	1	1	1	1	0
0 =	**0**	0	0	0	0	0	0	0

Solche Zahlen-Kodierungen stellen also einen in sich geschlossenen Kreis dar!

Besonders beim Umsetzen von Python-Programmen in andere Programmiersprachen, muss die Ganzzahlen-Darstellung in der Zielsprache beachtet werden! Die meisten Programmiersprachen benutzen begrenzte Zahlen-Typen.

Aufgaben:

1. ***Überlegen Sie sich, welche Ergebnisse bei den folgenden Berechnungen zu erwarten sind, wenn der Datentyp shortint verwendet wird!***

a)	2 + 4	b)	18 + 33	c)	125 + 2
d)	120 + 13	e)	7 - 2	f)	78 - 32
g)	126 – 2	h)	-124 – 2	i)	-124 - 5

2. ***Erkunden Sie, was die Operatoren &&, || und ! bewirken! Arbeiten Sie dazu mit kleinen Zahlen und stellen Sie sich die Eingaben und Ausgaben in Ihren Mitschriften binär (untereinander) dar! (&& und || sind zweistellige, innere Operatoren; ! ist ein einstelliger Präfix-Operator)***
3. ***Probieren Sie die Operatoren mit selbstgewählten Zahlen aus und stellen Sie diese Beispiele mit Erklärung(en) dem Kurs vor!***
   (binäre / duale Zahlen lassen sich in der folgenden Form eingeben: `0b`*dualziffern* ***)***

3.1.3. Eingaben und Daten merken - Variablen

Bestimmte Zahlen sollen in unseren Python-Skripten vielleicht häufiger verwendetet werden, aber sich auch von Skript-Aufruf zu Skript-Aufruf ändern.

Für solche Zwecke kennen wir in der Mathematik die Variablen. Die berühmtesten sind sicher x und y. Man versteht darunter beschriftete "Behälter" oder "Container" in denen etwas aufbewahrt wird. In der Informatik heißen Variablen exakt Bezeichner.

Apotheker-Schrank
Q: www.flickr.com (Leanne McCauley)

Man kann sich Variablen auch gut als beschriftete Schubladen in einem Apotheker-Schrank vorstellen. In den Schubladen wird etwas aufbewahrt – wir sagen es wird gespeichert.

Im Allgemeinen können und werden sich die Inhalte der Schubladen ständig verändern. Natürlich kann aber auch nichts oder irgendwelcher Müll in den Schubladen sein.

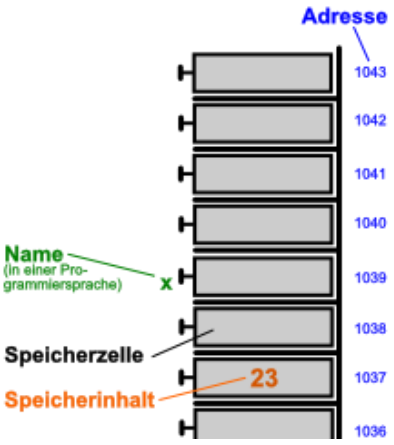
Wichtig ist, dass wir es mit zwei Dingen zu tun haben, einmal den beschrifteten Schubladen mit irgendeinem Namen und zum zweiten mit dem Inhalt der Schublade.

Natürlich gibt es Variablen auch in Python. Jeder Variable muss zuerst einmal ein Name zugeordnet werden. Anders als in verschiedenen anderen Programmiersprachen braucht man die Variablen vorher zu deklarieren (definieren). D.h. man muss nicht vorher sagen bzw. zuerst festlegen, was man in der Variable abspeichern möchte (z.B. Zahlen od. Texte) und wie groß der Inhalt werden könnte (z.B. nur ein Buchstabe oder eine Zahl mit 30 Stellen). Man benutzt die Variablen in Python sofort.

Als Namen darf man in Python alle Namen verwenden, die mit einem Buchstaben oder einem Unterstrich beginnen. Es können zwar einige wenige Sonderzeichen eingebaut werden. Das sollte man aber genauso vermeiden, wie die deutschen Umlaute und das "ß".

Üblicherweise sollte man passende Namen verwenden. Besonders Anfänger neigen dazu immer die typischen Variablen-Namen – sowas wie x, y, a, b und i – zu verwenden. Für einfache, kleine und übersichtliche Programme mit klarem mathematischen Konstrukten ist das auch ok. Ansonsten sollte man sich gleich von Anfang an angewöhnen, aussagekräftige Namen zu benutzen. In der Programmierung nennen wir solche aussagekräftigen Variablen-Namen **sprechende Bezeichner**. Später - in komplizierteren Programmen – wird man das zu schätzen wissen. Erst später – in größeren Programmen – mit solchen ausgeschriebenen Variablen-Namen – zu beginnen, ist mit großen Umstellungs-Problemen verbunden. Schlechte Angewohnheiten wird man nicht so schnell wieder los.

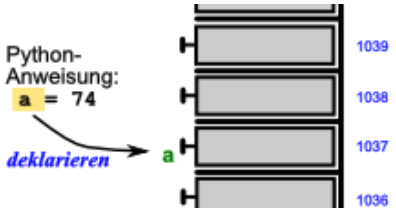
Ein weiterer wichtiger Grund für aussagekräftige Variablen-Namen ist die Notwendigkeit, auch später mal das eigene Programm oder ein fremdes Programm zu pflegen, zu erweitern, zu dokumentieren oder zu berichtigen. Das Alles gehört heute zu den wichtigen Tätigkeiten eines Programmierers.Python reserviert für jede angegebene Variable einen Stück vom Speicher. In diesen wird der zugewiesene Wert eingespeichert. Die verschiedenen Arten von Variablen – also solche für Texte und Zahlen werden getrennt voneinander gespeichert. Das muss uns aber nicht interessieren. Für unsere Zwecke reicht es, sich den Speicher als riesigen Stapel von Schubladen vorzustellen. Jede Schublade (Speicherzelle) hat eine einzigartige, fortlaufende Adresse. Die Schubläden sind quasi durchnummeriert.



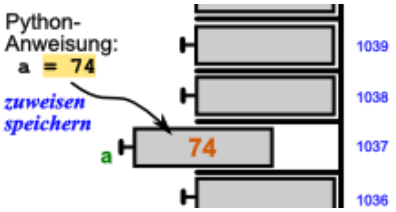
Python gibt bestimmten Speicherzellen nun den internen Variablen-Namen und bei der Wertzuweisung mit "=" wird festgelegt, welche Zahl oder welcher Text in die Schublade getan werden soll (Initialisierung). Besser spricht man statt "ist gleich" bei einer Wertzuweisung von "ergibt sich aus"! Das trifft den Kern genauer und später werden wir sehen, dass es sich nicht wirklich um eine "ist gleich"-Aktion handelt.
(Praktisch können auch mehrere Zellen (Schubläden) zusammen für eine (große) Zahl oder längere Texte benutzt werden. Das ändert aber nichts am Prinzip. Später werden wir uns dann auch mal anschauen, wieviel Speicher für bestimmte Daten genutzt werden.)
Schauen wir uns kurz ein paar Beispiele an, um das Verfahren der Variablen-Erzeugung und der Wert-Abspeicherung zu verstehen.
Als Beispiel wollen wir der Variable a den Wert 74 zuweisen.

```
>>> a = 74
>>>
```

Der Python-Übersetzer prüft bei einer Eingabe a = 74, ob es schon eine Speicherzelle mit dem Namen a gibt.
Falls ja, dann wird natürlich diese benutzt. In unserem Fall gibt es sie noch nicht.
Python legt nun eine Speicherzelle – irgenwo im Speicher – mit dem Namen a an. Diesen Teil nennen wir Deklaration. In vielen Programmiersprachen muss die Deklaration extra und vor der Benutzung erfolgen.
Unser System ist hier flexibler.



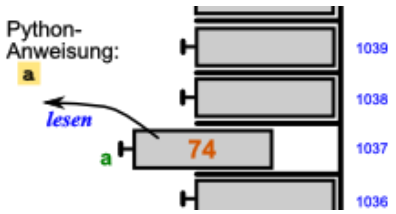
Nun wird der zweite Teil der Anweisung ausgeführt. In die Speicherzelle a wird der Wert 74 eingespeichert. Man spricht auch von einer Zuweisung.
Im Fall des Hauptspeichers (RAM's) bleibt dieser Inhalt nun solange erhalten, wie der Speicher mit Strom versorgt wird oder in die Speicherzelle ein neuer Wert geschrieben wird.



Vom Python-System kommt kein Feedback zurück, wenn man mal davon absieht, dass keine Fehlermeldung auch schon ein (eben positves) Feedback ist.
Mit einem Aufruf der Variable zeigt Python uns den eingespeicherten Wert an.

```
>>> a
74
>>>
```

Dazu wird die Speicherzelle einfach einmal ausgelesen. Der Wert in der Zelle bleibt beim Lesen erhalten.
Diese Grundfunktionen müssen wir uns vergegenwärtigen, wenn wir später über das Arbeiten mit Variablen reden. Mehr als das Anlegen, Belegen und Auslesen einer Variable ist nicht drin.



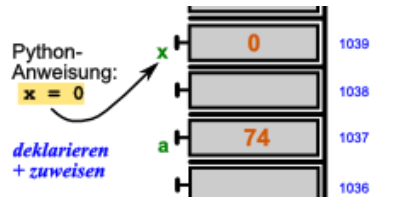
Unter bestimmten Bedingungen sorgt das Python-System dafür, dass nicht mehr gebrauchte Variablen aus dem Speicher entfernt werden. Praktisch wird auch nur der Name aus der Namensliste gestrichen, so dass hierüber kein Zugriff mehr erfolgen kann. Der Inhalt der Speicherzelle bleibt erhalten, ist aber nicht mehr direkt zugänglich.
Rufen wir einen Vriablen-Namen auf, der noch nicht vom Python-System angelegt wurde, dann bekommen wir eine Fehler-Meldung. Diese sagt aus, dass "x" noch nicht definiert ist. Die Variable x wurde noch nicht ordnungsgemäß initialisiert / deklariert.

```
>>> x
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>>
```

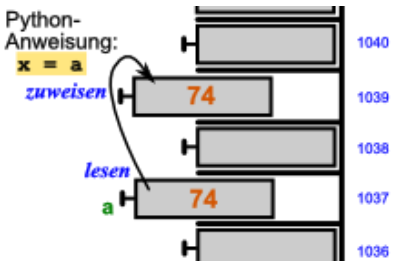
Mit der nebenstehenden Anweisung wird die Variable x im Speicher angelegt und ihr der Wert 0 zugewiesen.
Solcheine initiale Belegung (Anfangsbelegung) sollte man sich für jede Variable angewöhnen.

```
>>> x = 0
>>>
```

Der genaue Ort der Speicherzelle x ist nicht wirklich vorhersehbar. Meist erfolgt die Speicherung direkt neben den älteren Variablen.
Nun lässt sich x auch benutzen, also auslesen oder neu belegen.
Das soll nun auch getan werden. Die Variable x soll den gleichen Wert bekommen, wie die Variable a.
Die Anweisung besteht aus zwei Teilen. Zum Ersten aus dem Auselesen von a und dem folgenden Einspeichern in x.
Viele Anweisungen werden zuerst auf der rechten Seite vom Zuweisungs-Zeichen geklärt und dann die eigentliche Zuweisung erledigt.
Eine einmal angelegt Variable kann innerhalb einer Shell-Sitzung oder innerhalb eines Programmes an beliebiger Stelle wiederbenutzt werden. Dabei kann man den Wert ändern, indem einfach eine neue Zuweisung gemacht wird. Natürlich lässt sich der Wert jeder Variable so oft, wie gewünscht abrufen.
Bis zur nächsten Zuweisung bleibt der Wert erhalten.
Die Benutzung der Variablen für Berechnungen verändert den Wert der Variable nicht.




```
>>> x = a
>>>
```




```
>>> aaa=25
>>> aaa
25
>>> x
74
>>> x=100
>>> x
100
>>> x+x
200
>>> x
100
```

3.1.3.1. besondere Variablen und spezielle Möglichkeiten für Variablen in Python

_-Variable
im interaktiven Modus verfügbar, beinhaltet sie den letzten ausgegebenen Ausdruck

mehrere Variablen können gleichzeitig einen Wert erhalten
a = b = c = 7

Anzeige aktuell benutzter Variablen-Namen
print(dir())

x,y = pos()
einige Funktionen liefern zwei Informationen, typisch ist das bei Funktion, die Koordinaten als Rückgabewerte liefern
hier ist es so, dass sowohl x als auch y einen eigenen Wert bekommt
solche Komma-getrennte Variablen bzw. Werte werden Tupel genannt, dazu später mehr (→ 9.1. Tupel)

Aufgaben:

1. ***Prüfen Sie (quasi wie ein Python-System), ob die nachfolgenden Ausdrücke als Variablen-Namen zugelassen wären! Sollte dieses nicht so sein, dann erklären Sie, warum der Ausdruck kein gültiger Variablen-Name ist!***

a)	x	b)	_Input	c)	4.Zahl
d)	Eingabe	e)	eingabe	f)	_Eingabe_
g)	X_1	h)	Text.1	i)	Tätigkeit
j)	Anna Maria	k)	mAx	l)	E
m)	=x	n)	x-3	o)	C_?

2. ***Lassen Sie nun Python (auf der Konsolenebene / Shell) die Gültigkeit der Variablen von 1. prüfen! Dazu geben Sie zuerst eine beliebige Zuweisung (Text, Zahl, vorher benutze Variable) ein und anschließend fragen Sie den Wert der Variable durch die Eingabe des Variablen-Namens ab! (Siehe nebenstehendes Beispiel!)***

```
>>> x=5.5
>>> x
5.5
```

3. ***Prüfen Sie (nach Python-Art), ob die nachfolgenden Ausdrücke ordnungsgemäße Wertzuweisungen und / oder Variablen-Benutzungen sind! Wir gehen davon aus, dass vorher noch keine Eingaben gemacht worden sind (ev. IDLE oder Shell neu starten). Machen Sie jeweils Voraussagen dazu, welche Werte die einzelnen Variablen nach der Eingabe haben müssten!***

a)	X = 100	b)	X + X	c)	y = X + a
d)	a = (12 +3) * 2	e)	bc23 = X + X + X	f)	x=12
g)	a = X + x	h)	a = 12k – 4k	i)	45 + 55 = Hundert
j)	Tausend = 40 + 60	k)	Eingabe = 2	l)	Ausgabe = Eingabe
m)	Masse	n)	_28T = _13T * 7	o)	Hundert = Tausend *1

4. ***Starten Sie eine neue Shell! Lassen Sie nun Python (auf der Konsolenebene / Shell) die Gültigkeit der einzelnen Ausdrücke von 3. prüfen! (Behalten Sie die Reihenfolge unbedingt bei!)***

Etwas Verwirrung erzeugen solche Ausdrücke, wie in der nebenstehenden Shell-Ansicht zu sehen. Sie machen mathematisch keinen Sinn – sind ja eigentlich sogar falsch.
Wenn wir uns den aktuellen Wert von x anzeigen lassen, dann werden wir das dahinterliegende Arbeitsprinzip verstehen.
In Python muss man solche Ausdrücke etwa so lesen und verstehen:

```
>>> x = x + 1
>>>
```

```
>>> x
201
>>>
```

Der (neue – zu speichernde) Wert von x <u>ergibt sich aus</u> dem (alten / derzeitigen / aktuellen - ausgelesenen) Wert von x addiert mit 1.

x[zu speichern] = x[aktuell] + 1

x[zu speichern] = 200 + 1

x[zu speichern] = 201

Zuerst wird uns dieses das eine oder andere Mal ungewöhnlich vorkommen, aber nach ein, zwei Programmen geht einem diese Denkweise ins (Programmierer-)Blut über. Das einfach

Gleichheitszeichen ist in Python also ein Zuweisungs-Zeichen (entspricht: "ergibt sich aus") und kein mathematisches Gleichsetzungs-Zeichen!

Ähnlich kryptisch sieht der folgende Konstrukt aus. Wir erzeugen uns eine Variable buchstaben und weisen der z.B. den Text "abc" zu. Nun können wir auch eine Operation mit dem Sternchen und einer Zahl ausführen.

```
>>> buchstaben = "abc"
>>> buchstaben = buchstaben * 3
```

Python akzeptiert dies seltsamerweise – für Daten von zwei verschiedenen Typen (Text und Zahl verrechnen?) schon etwas ungewöhnlich.

Die Ausgabe zeigt das Ergebnis der Sternchen-Operation – es kommt zu entsprechend vielen Wiederholungen.

```
>>> buchstaben
'abcabcabc'
>>>
```

Aufgaben:

1.

2. Probieren Sie mal die folgenden Anweisungen an der Konsole! Lassen Sie sich immer die beiden Variablen zwischendurch anzeigen! Sie können die Anweisungen (c bis e)auch mehrfach hintereinander aufrufen!

a) `x = 2`
b) `a = 1`
c) `x += 1`
d) `a *= 2`
e) `a -= x`

Was machen diese "kryptischen" Anweisungen (Operationen)?

3. Geben Sie nun wieder die Anweisungen a und b von 2. ein! Was erwarten Sie, wenn Sie vor einer Ausgabe dann noch die Operationen c bis e von 2. ausführen? Begründen Sie Ihre Vermutung!

3.2. Arbeiten mit Scripten

Sequenzen von Python-Anweisnungen lassen sich in einer Text-Datei zusammenfassen dazu ist praktisch jeder Editor geeignet.
Damit der Python-Interpreter mit den Text-Dateien arbeitet, müssen sie die Datei-Endung .py bekommen.
Viele Text-Editoren bieten tolle Möglichkeiten der Textbearbeitung. Wir werden einige noch kennen lernen bzw. vorstellen (→ 3.4.1. gut geeignete Editoren für die Verwendung mit Python). Es bleibt die freie Entscheidung des Programmierers, welcher Editor für ihn am Besten ist.
Um die Programme zu starten, müssen sie dem Python-Interpretor (unter Windows heisst er: py.exe) übergeben werden. Der Dateityp *.py wird bei der Installation des Python-Systems mit dem Programm py.exe verknüpft. (So wie z.B. die docx-Dateien mit dem Programm WORD oder xlsx-Deteien mit EXCEL verbunden sind.
Läuft das Programm odnungsgemäß, ist alles ok. Sind aber Fehler im Programm, dass muss wieder im Editor der Quell-Text geändert und gespeichert werden und dann die datei wieder mit dem Interpreter ausprobiert werden. Das Wechseln zwischen Editor und Interpreter ist nicht sehr praktisch, aber es funktioniert. Schöner wäre natürlich ein Programm, dass sowohl das Editieren als auch das Testen des Programms zulässt. Solche Programme schauen wir uns ebenfalls noch an (→ 3.4. Nutzung anderer Benutzer-Oberflächen).

3.2.1. Grundlagen DOS bzw. Komandozeile (Eingabeaufforderung, Terminal)

besondere Zeichen usw.	**Bedeutung / Verwendung**		**Zeichen in Linux**	
*	Joker-Zeichen für **alle** möglichen (zugelassenen) Zeichen			
?	Joker-Zeichen für **ein** möglichen (zugelassenen) Zeichen			
\	Backslash über: [Alt Gr]+[ß] Trenner zwischen Ordnern / Verzeichnissen		/	
:	Laufwerks-Kennzeichen (mit Buchstabe davor)			
>	Umleitungs-Kennzeichen z.B. in eine Datei			
|more	Begrenzung der Anzeige auf Display-übliche Zeilen und Warten auf eine Eingabe			
"***name***"				

ein Pfad ist die Kombination von Laufwerk und allen Ordnern und Unterordnern, die zu einer Datei od.ä. führen.

Pfade können auch beim aktuellen Ordner starten, dann beginnt er mit ./

Befehl	**Funktionsumfang**		**Befehl in Linux**	
`cd` ***name***	Wechsel eines Ordners / Verzeichnisses mit dem angegebenen Namen			
`cd\`	Wechsel in den Basis- / Wurzel-Ordner des Laufwerkes			
`cd..`	Wechsel in den höheren Ordner / in das übergeordnete Verzeichnis			
`dir`	(ausführliche) Anzeige der Dateien und Ordner im aktuellen Ordner / Verzeichnis		`list`	
`dir /w`	Anzeige der Dateien und Ordner im aktuellen Ordner / Verzeichnis in Spalten (funktioniert nur bei durchgehend kurzen Namen)			
`dir *.py`	Anzeige aller py-Dateien (Python-Dateien) im aktuellen Ordner			
`md` ***name***	(auch: makedir) Erstellen eines neuen Unter-Ordner / Unter-Verzeichnis mit dem angegebenem Namen			

3.2.2. Aufruf fertiger Python-Skripte

direkt in Windows z.B. im "Arbeitsplatz" oder dem "Windows Explorer" (Datei-Explorer)
Shell braucht dabei nicht schon vorher gestartet werden
es gibt intern eine Verknüpfung der Datei-Endung / dem Dateityp ***.py*** mit dem Python-Programm (Python-Interpreter)
Programm wird zuerst automatisch gestartet und dieses benutzt dann als nächstes die geklickte Datei

praktisch reicht der Doppelklick auf eine *.py-Datei, um sie dem Programm py.exe (- dem Python-Interpreter -) zu übergeben. Die py.exe übernimmt die Datei und führt sie aus – besser gesagt, es interpretiert die *.py-Datei.

in der Shell durch Aufruf: import skriptname
vorher u.U. das richtige Laufwerk und die richtigen Verzeichnisse und Unterverzeichnisse auswählen

starten der geöffneten Skripte mit Taste [F5] oder über "Run" "Run Module" möglich

Aufgaben:

1. ***Starten Sie die Python-Shell!***
2. ***Starten Sie die nachfolgenden Skripte aus dem vorgegebenem Ordner (wird vom Kursleiter an die Tafel geschrieben!)!***
   `hello.py` `nutzer.py`
3. ***Rufen Sie das Skript*** *`nutzer.py`* ***noch einmal auf und beantworten Sie die Eingabeaufforderung anders! Warum hat sich das System die alte Eingabe nicht gemerkt?***

Auch auf der Komandozeile ist das Aufrufen von py-Dateien (Python-Quelltexten) möglich. Sollte das Fenster der Eingabeaufforderung bzw. der Konsole gleich wieder verschwinden, dann geben Sie beim Quelltext am Schluß einfach ein **input()** ein. Dieser Befehl bewirkt ein Warten auch ein [Enter]. Alles weitere zum Befehl **input()** dann später genauer (→ 6.2. Eingaben).

3.3. die interne Benutzer-Oberfläche

GUI (Graphic User Interface) heißt IDLE (sprich: eidel) steht für "**I**ntegretad **D**eveLopment **E**nviroment" (dt.: integregrierte / eingebaute Entwicklungs- / Programmier-Umgebung) bei anderen Programmiersprachen wird auch nur von der IDE od. eben der GUI gesprochen

Aufgaben:

1. ***Starten Sie die Python-GUI IDLE!***
2. ***Erstellen Sie sich ein neues Eingabe-Fenster! Speichern Sie dieses sofort in Ihrem eigenen Ordner oder auf Ihrem persönlichen Datenträger ab!***

Für dokumentarische Zwecke kann man sich das Ausgabe-Fenster auch abspeichern. Dieses lässt sich aber nicht ausführen!

über die GUI bekommen wir ein Windows-typisches Bediensystem für Python
man kann die Programmtexte Öffnen, speichern, editieren und starten

3.3.x. Hilfe(n)!

Hilfe zu einzelnen Befehlen / Schlüsselwörtern durch
help(*Schlüsselwort*)

Hilfe-Modus mit **help()** ohne Argument
keywords um die Schlüsselwörter abzufragen

```
False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

Hilfe-Texte zu den einzelnen Schlüsselwörtern durch Eingabe des Schlüsselwortes
verlassen des Hilfe-Modus mit **quit**

Aufgaben:

1.

x. Warum ergibt die Eingabe "help(keywords)" eine Fehlermeldung und nicht die Liste der Schlüsselwörter?

3.4. Nutzung anderer Benutzer-Oberflächen

Wer bei der Python-eigenen IDE bleiben möchte oder muss (weil er nichts anderes installiert bekommt oder die Einarbeitung zu langwierig wäre), der überspringt einfach den Rest dieses Abschnitts und liest bei Abschnitt 4. weiter (→ 4. erste einfache Programme mit Python)
Für diejenigen, die öfter Programmtexte eintippen und korrigieren müssen, biete ich hier ein paar geeignete Editoren an. Also vielleicht in den ersten Abschnitt dieses Kapites noch reinschauen.
Später (ab → 3.4.x. Eclipse) stellen wir auch noch echte GUI's vor, die z.T. sehr Leistungsfähig sind und sogar bei der Fehlersuche helfen.

3.4.1. gut geeignete Editoren für die Verwendung mit Python

Bitte beachten!:
Die Auswahl und die konkrete Bewertung der nachfolgenden Editoren ist rein subjektiv. Wer einen Lieblings-Editor hat, sollte nur kurz gegenprüfen, ob er genauso Leistungsfähig, wie die nachfolgend vorgestellten ist. Ansonsten gilt der Leitspruch vieler Datenverarbeiter:

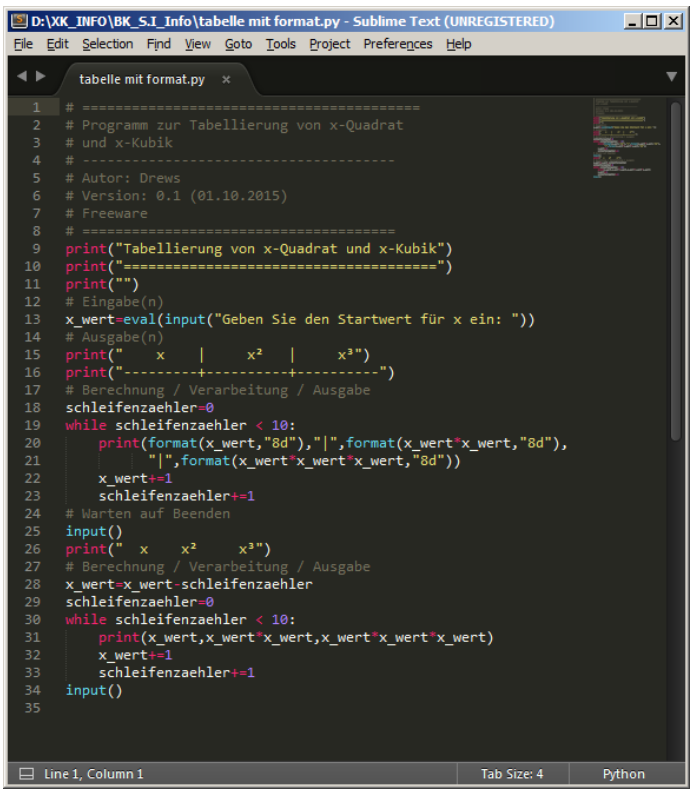
Never touch a running system.

Klar sollte sein, dass NotePad, WordPad oder Word keine wirklich geeigneten Code-Editoren sind. Natürlich lassen sich die entsprechenden Dateien mit ihnen erzeugen und bearbeiten, aber richtiges Programmieren geht anders. Als Schnell- und Ausnahmsweise-Ersatz ist aber nichts gegen die genannten Programme zu sagen. Und manchmal geht es eben nicht anders.

3.4.1.1. Sublime Text

Vielfach als Allzweck-Editor gelobt. Er kann und bietet fast alles, was man sich als Programmierer und System-Betreuer wünscht.
Einen bearbeiteten Python-Code kann man mit [Strg] + [b] an den Python-Interpreter übergeben und ausführen lassen.
Es gibt vom "Sublime Text"-Editor Varianten für Windows, Linux und den Mac. Weiterhin stehen auch Downloads für "portable Apps"-Umgebungen bereit

Links / Download:
www.sublimetext.com



3.4.1.2. Geany

Text-Editor
schnelle, kleine IDE
automatische Code-Vervollständigung
automatische Syntax-Hervorhebung, Formatierung
für unzählige andere Sprachen usw. geeignet

3.4.1.3. Notepad++

schlanker, schneller, freier Text-Editor
automatische Code-Vervollständigung
automatische Syntax-Hervorhebung, Formatierung
für unzählige andere Sprachen usw. geeignet
die gewünschte Sprache kann über das "Sprachen"-Menü zugewiesen werden
dadurch wird der Syntax farblich dargestellt und die Datei-Endung (Datei-Typ) für das Abspeichern vorbelegt
auch als portableApp verfügbar
auf dem IoStick (unter Tools) enthalten

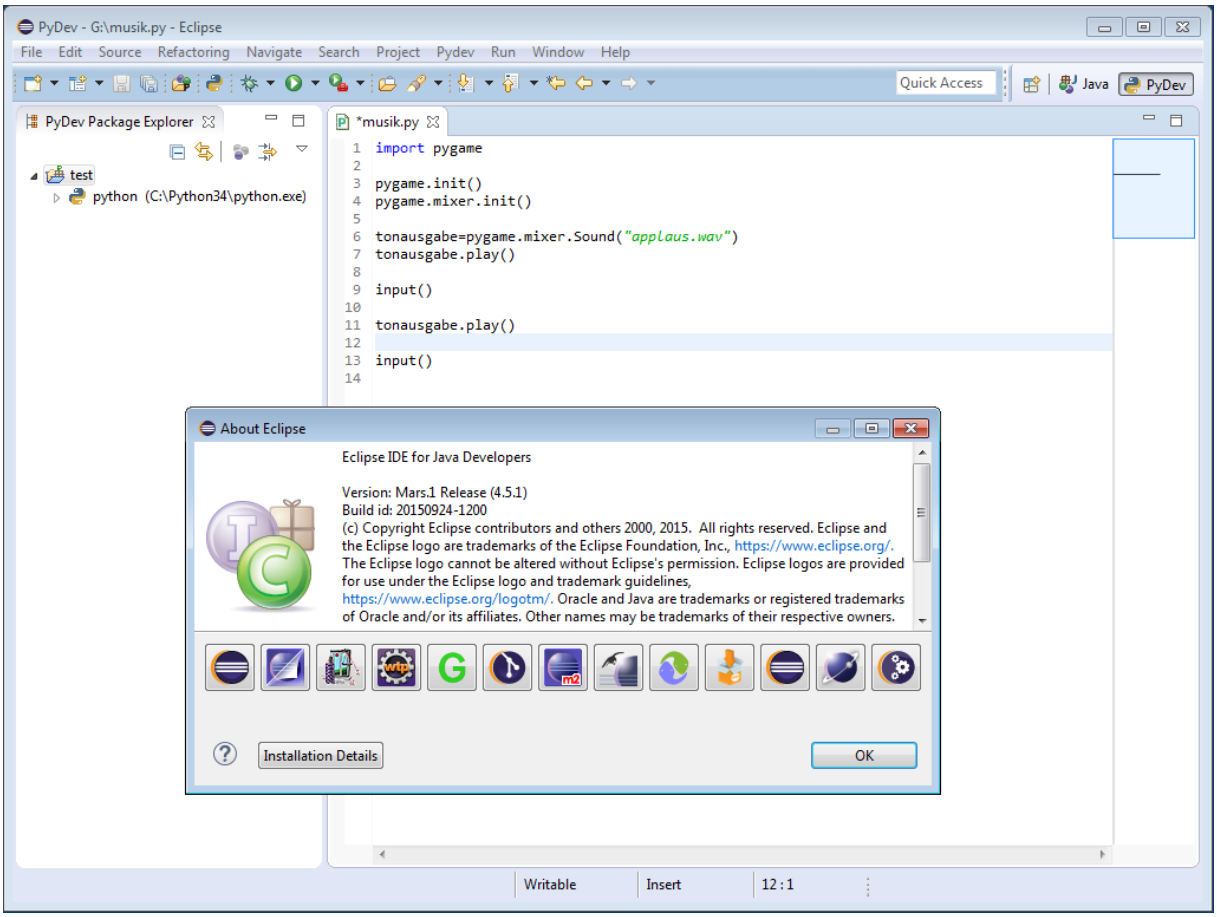
kein Debugger

3.4.1.4. Komodo Edit

3.4.x. Eclipse

Ein der weit verbreitesten universellen Entwicklungs-Umgebungen ist "Eclipse". Ursprünglich für Entwicklungen mit Java erstellt, ist die IDE heute für eine Vielzahl von Programmiersprachen nutzbar. Auch für die Verwendung mit Python lässt sie sich einrichten.

notwendige Erweiterung heißt PyDev
sowohl Eclipse als auch PyDev sind freie Produkte



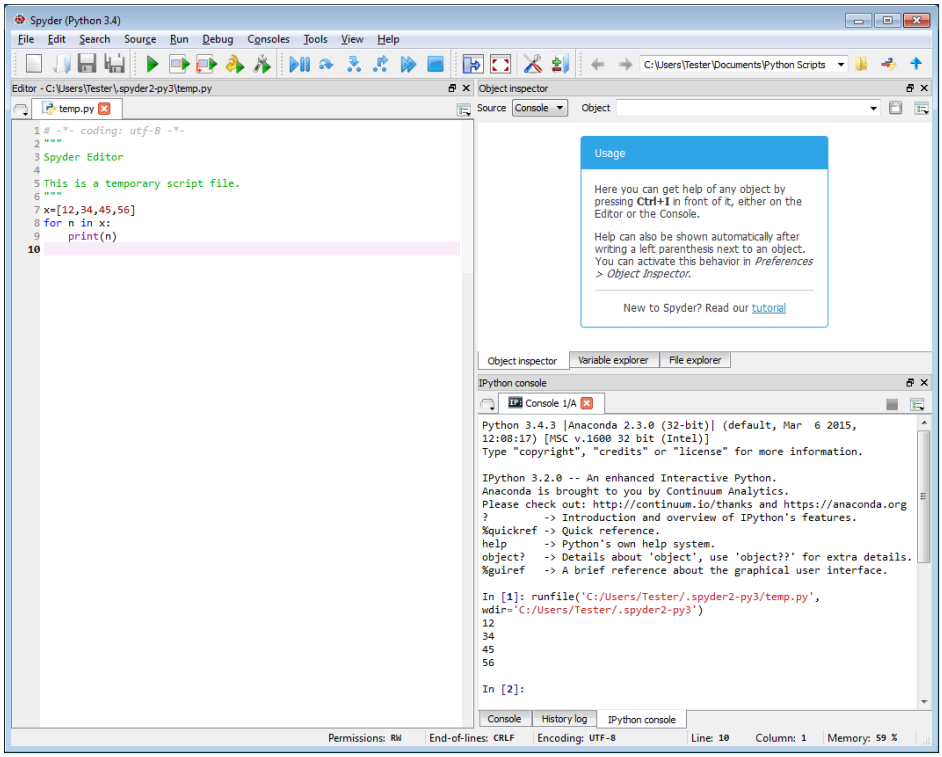
Eclipse in der Version Mars1 mit installiertem PyDev

ist selbst in Java geschrieben und steht dadurch auf fast allen Betriebssystemen zur Verfügung
für Normalnutzer ergeben sich kaum Unterschiede auf den einzelnen Plattformen
benötigt für die Installation und das Nutzen eine Java-Runtime- oder -Entwicklungs-Umgebung, was bei älteren Systemen zu Performance-Problemen führen kann
Java gilt zudem nicht unbedingt als ein sehr sicheres System, Java ist sehr mächtig und eben auf allen Plattformen zuhause, zwar gibt es sehr regelmäßig Updates, aber ein Restrisiko bleibt

Viele der typischen Programmierer-Tätigkeiten lassen sich mit Eclipse effektiver erledigen. Aber solche mächtigen graphischen Benutzer-Oberflächen haben auch ihre Nachteile. Die Funktions-Vielfalt und die sehr komplexe Oberfläche überfordern vielleicht den einen oder anderen Einsteiger.

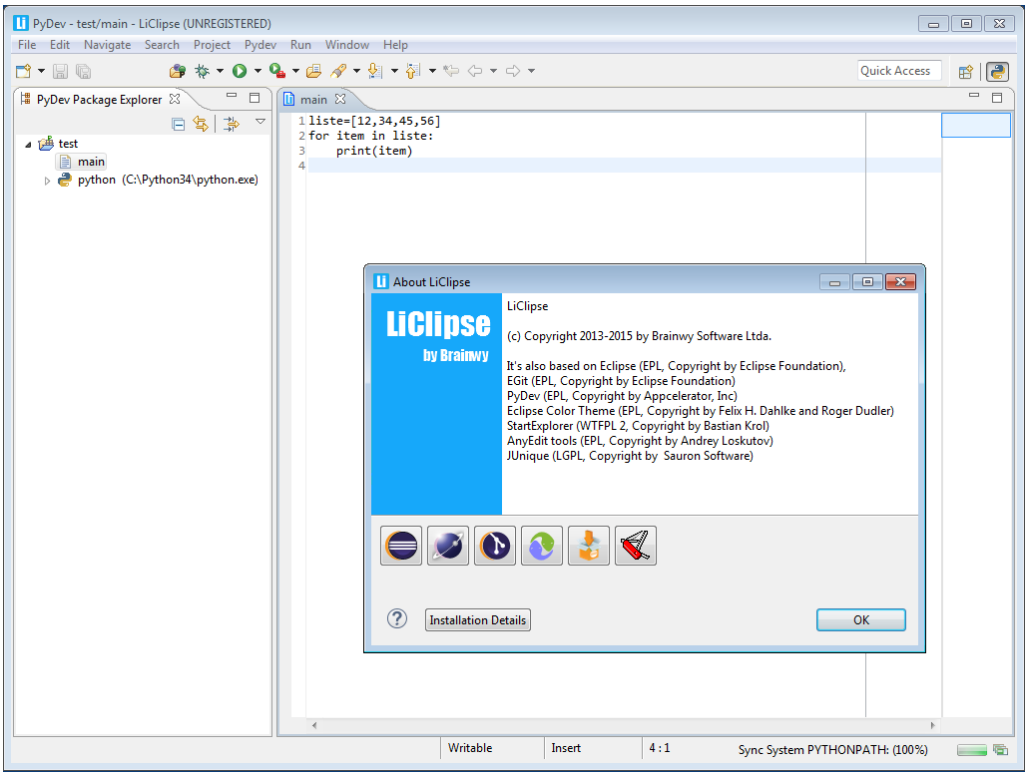
Links:
https://www1.ethz.ch/foss/news/course_python/configEclipse

3.4.x. Spyder



Editor und Konsole in einem gemeinsamen Fester
editieren und ausprobieren lassen sich so schneller und übersichtlicher durchführen

3.4.x. LiClipse



abgespecktes Eclipse

3.4.x. Anaconda

enthält viele verschiedene Python-Bibliotheken und Hilfs-Mittel
diese sind für Anfänger erst einmal nicht so interessant
Entwicklungs-Umgebung für die wissenschaftliche Programmierung
stellt Spyder als IDE zur Verfügung
weitere - sehr verbreitete - Bereitstellung von Programmen und Kommentaren sowie Abfolgen / Protokolle sind die sogenannten Jupyter-Notebook's (lauf Browser-basiert, lassen Programme, Texte, Einn- und Ausgaben zu, die auch zur Dokumentation als Ganzes gespeichert werden können)
es lassen sich mehrere unabhängige Programmier-Umgebungen definieren, die quasi eine Programmierung in einer Sandbox erlauben
reibungslose Installation
kann neben dem originalen Python installiert und betrieben werden

3.4.x. WinPython

mit Debugger

stellt auch Spyder zur Verfügung
kleine Probleme mit Installation und den ausführbaren Programmen

3.4.x. Komodo IDE

IDE zum Komodo Editor
kostenpflichtige Lizenz

3.4.x. Thonny

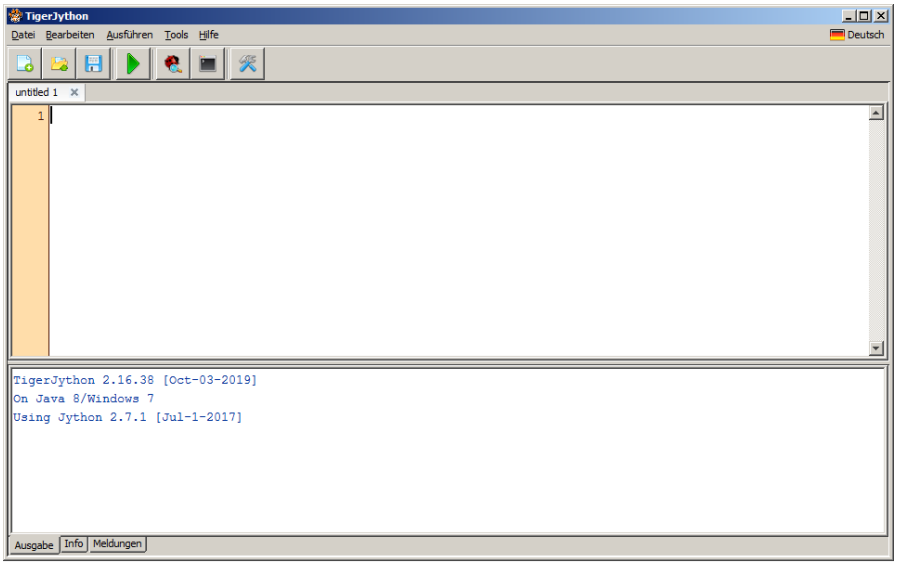
http://thonny.org/
für Windows, Mac und Linux

es gibt auch Backend für bbc micro:bit

3.4.x. SciTE

Q: https://www.scintilla.org/SciTE.html
Q: https://www.heise.de/download/product/scite-10783

3.4.x. TigerJython



Jython ist nicht etwa falsch geschreiben – sondern ein Kunstwort aus JAVA und Python. Dies soll die spezielle Version von Python charakterisieren. Bei TigerJython – einer Jython-Version – handelt es sich um eine vollständige und voll kompartible Python-Version. D.h. man kann in TigerJython genau so Python-Programme schreiben, wie im originalen Python-System von python.org (IDLE).
Die Übersetzung der Python-Quellcode's und die Arbeitsumgebung sind in JAVA programmiert worden. Jython verfügt deshalb über ein universelles Zwischen-Programm, das auf allen Geräten (Computer, Tablet, Smartphon, …) laufen kann, die JAVA können. Ein weiterer Vorteil ist die in das Programm direkt eingebaute JAVA-virtuelle-Maschine. Das Programmier-System TigerJython läuft damit unanbhängig von einer lokalen JAVA-Installation.

Nachteilig ist weitgehende Orientierung von Jython an der älteren Python-2-Version.
es soll in 2020 eine Python-3-Version geben

Die Input's werden in einen Dialog ausgelagert. Das hat schon den Anstrich von Programmierung einer graphischen Oberfläche.
Ausgaben im unteren –Terminal-ähnlichen Bereich – etwas ungewöhnlich. Man kann die Ausgaben aber auch auf Message-Dialoge auslagern. Damit sind die Programme dann nicht immer 100%ig übertragbar.

Quellen und Links:
→ http://www.tigerjython.ch (offizielle Seite zu TigerJython; u.a. auch Download's)
Q: http://letscode-python.de/links.php (Link-Liste zu TigerJython; Begleitbuch zu TigerJython)
Q: http://www.python-exemplarisch.ch/index_de.php?inhalt_links=navigation_de.inc.php&inhalt_mitte=home/de/home.inc.php (Arbeits-Material zu / mit TigerJython (u.a. mit Robotik, Microcontrollern, IoT, MachineLearning, BigData, …)
Q: http://www.tigerjython4kids.ch ()
→ http://www.python-online.ch (online-Programmier-Umgebung für Python)

3.4.x. Editoren im Internet – online-Editoren

benötigt keine Installation, außer einem aktuellen Browser
werden ständig aktualisiert und sind praktisch immer auf dem ajtuellsten Stand

nachteilig (hinsichtlich Datenschutz) ist, dass häufig eine Anmeldung notwendig ist
konsequent Schulaccounts / Schul-eMail-Adressen nutzen und alle persönlichen Angaben anonymisieren

3.4.x.1. w3schools.com

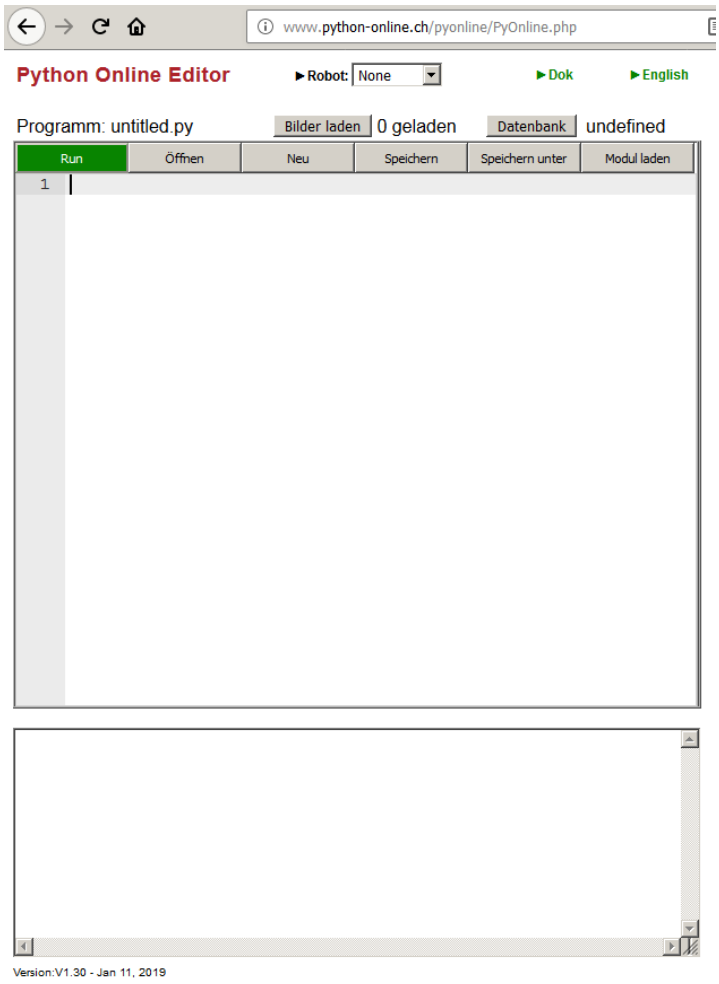
eine der besten Seiten; natürlich / leider alles englisch
https://www.w3schools.com/ (sehr viele Sprachen, …) ! keine Anmeldung notwendig

3.4.x.2. TigerJython

schlicht, aber alles, was man zum eigentlichen Programmieren braucht

Zu beachten ist, dass es sich aktuell noch um eine Umsetzung von Python 2 handelt.

→ http://www.python-online.ch (online-Programmier-Umgebung für Python / TigerJython)



3.4.x.3. repl.it

bietet neben Python auch online-Programmier-Umgebungen für viele andere Sprachen
immer Editor plus Übersetzer (Interpreter)

3.4.x.3. ???

3.5. Snap for Python

Python-Interface für Snap

benötigt 64bit-Betriebssystem und eine aktuelle Version von Python
Installation von Snap.py mittels

pip install snap-stanford

ev. vorher pip aktualisieren

python –m pip

für jede Betriebssystem-Plattform gibt es eine spezielle Download-Datei
in dieser ist auch eine setup.py enthalten, mit der ebenfalls eine Installation möglich ist

Windows

in der Konsole
cd in den Ordner, in dem sich das entpackte Download-Paket befindet
python setup.py install

Linux

in der Konsole
entpacken des Download-Paket's mit
tar zxvf snap-stanford-?.?.?-?.?-ubuntu?.?-x64-py3.?.tar.gz
cd in das Verzeichnis
sudo python3 setup.py install

MacOS

in der Konsole
entpacken des Download-Paket's mit
tar zxvf snap-stanford-?.?.?-?.?-macosx?.?-x64-py3.?.tar.gz
cd in das Verzeichnis
python3 setup.py install

Links:
https://snap.stanford.edu/snappy/index.html

4. erste einfache Programme mit Python

Programme sind Zusammenstellungen von Anweisungen (/ Befehlen), die auf einem Computer die Lösung einer Aufgabe ermöglichen sollen. Man könnte Programme auch als Umsetzungen von Algorithmen auf Computer verstehen. Alles was algorithmierbar ist, kann auch in ein Programm umgesetzt werden. Genauso, wie der Algorithmus, benötigt das Programm dann auch noch bestimmte Hardware / Werkzeuge zum Hantieren der bearbeiteten Objekte. Algorithmen und Programme sind quasi die Arbeitsvorschriften zur Erfüllung einer Aufgabe.
Die Folgen von Anweisungen werden i.A. in spezielle Dateien geschrieben, die Quellcode (Quell-Texte) genannt werden. Diese werden dann zur weiteren Bearbeitung, Korrektur usw. erst einmal abgespeichert.

Definition(en): Programm
Ein Programm (Maschinen-Programm, Computer-Programm) ist die für die Maschine bzw. den Computer nutzbare / ausführbare Folge von Befehlen, Anweisungen usw. zur gezielten Bearbeitung von Daten oder die Steuerung von Aktoren.

Definition(en): Algorithmus
Ein Algorithmus ist eine eindeutige, zum Ziel führende Handlungs-Vorschrift zur Bearbeitung einer Aufgabe.
Ein Algorithmus ist eine Sammlung / Folge systematischer und logischer Regeln und Vorgehensweisen, die zur Lösung einer Aufgabe führen.

Definition(en): Quelltext (eines Programms)
Ein Quelltext ist eine spezielle Umsetzung eines oder mehrerer Algorithmen in eine Programmiersprache. Neben den Anweisungen für die Maschine enthalten (gute) Quelltexte auch zusätzliche Hinweise / Kommentare für den menschlichen Bearbeiter / Leser. (Der Quelltext muss vor der Benutzung durch die Maschine / den Computer zuerst in eine für ihn verständliche Codierung umgesetzt werden (mit → Compiler oder Interpreter).)

Um ein Programm zu schreiben bzw. einen Quelltext einzugeben, müssen wir uns in IDLE ein neues Fenster ("File" "New File") öffnen. Jetzt sind wir quasi auf der Editor-Ebene. Den Text unseres Programms können wir in beliebiger Reihenfolge und Art und Weise erstellen. Am Schluß des Editieren (Veränderns) kommt dann die Stunde der Wahrheit und wir lassen Python testen, ob der Programmtext als Programm taugt.
Bevor aber nun wild editiert und programmiert wird, kümmern wir uns zu allererst um das Abspeichern des Quelltextes. Mit "File" "Save As …" kommen wir zu einem klassischen Datei-speichern-Dialog. Die Datei sollte – wie üblich – in einem gesonderten Ordner – z.B. einem privaten Ordner – abgelegt werden. Für das regelmäßig Speichern ist jeder selbst verantwortlich. Als schnelle Tasten-Kombination kann man sich hierfür [Strg] + [s] merken.

Jedes Programm folgt dem EVA-Prinzip. EVA steht hier nicht für eine freundliche Mitschülerin, sondern für den grundlegenden informatischen Dreiklang:

Eingabe – Verarbeitung – Ausgabe

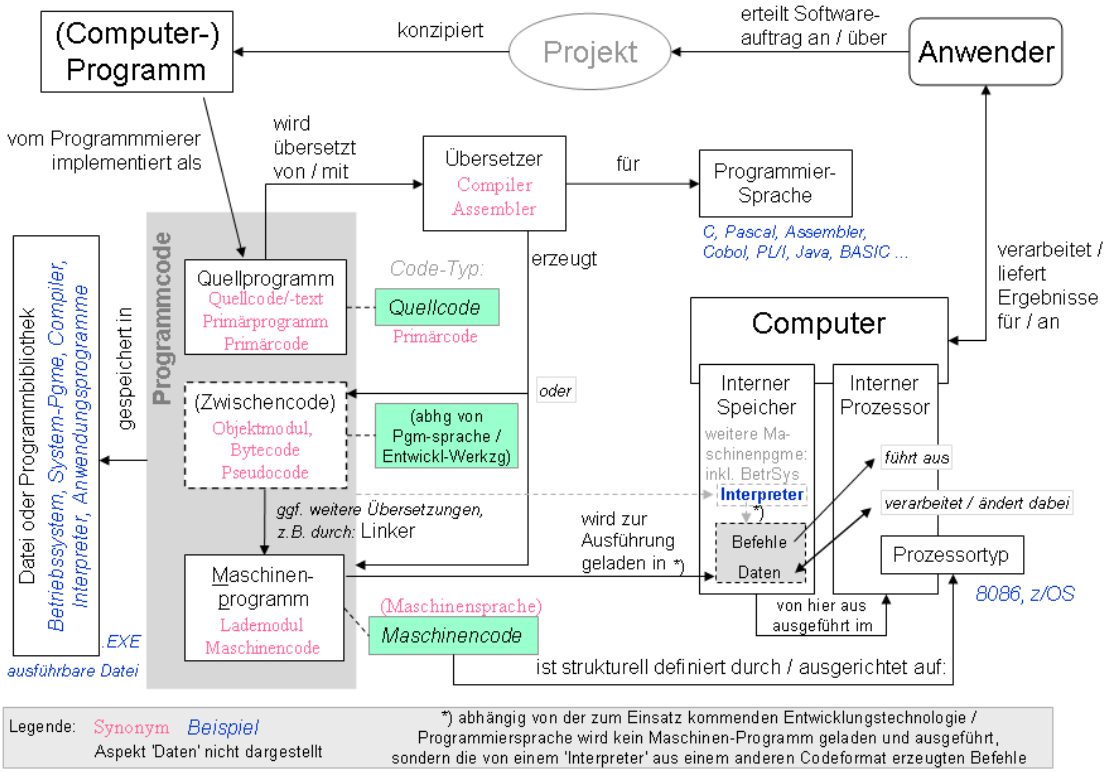
Gute Programmierer bauen diese Struktur auch im Quelltext ihrer Programme nach.
Der Ablauf eines Programms sollte immer vorgeplant werden. Zumindestens bei etwas komplizierteren Programmen kommt man dann nicht mehr ohne Vorplanung aus. U.U. werden bei größeren Programm-Projekten nur bestimmte kritische Abschnitte in passenden Schemata skizziert.
In der Programmier-Praxis gibt es zwei unterschiedliche Skizzentypen für geplante Programm-Verläufe. Die erste Variante sind sogenannte **P**rogramm-**A**blauf-**P**läne – kurz **PAP** genannt. Wegen ihres großen Platzbedarfs beim Skizzieren werden sie heute seltener verwendet. Es gibt für **Start** und **Stop**, sowie **Eingaben**, **Berechnungen**, **Entscheidungen** und **Ausgaben** unterschiedliche Symbole, die durch Verlaufs-Linien verbunden werden. Nebenstehend ist ein sehr PAP für das EVA-Prinzip dargestellt. Man liesst sich in diese Pläne recht schnell ein und die Wege sind auch gut erkennbar.

Eine moderne Alternative zu den Programm-Ablauf-Plänen sind **Struktogramme**.
Bei Struktogrammen wurden unwichtige Elemente, wie Start uznd Stopp weggelassen und alle Elemente werden in Blöcke (Rechtecke) gebracht. Für Eingaben und Ausgaben gibt es Block-Symbole mit rein- bzw. rauszeigenden Dreiecken.

Struktogramme sind schön kompakt und orientieren sich an der gewünschten Modul-Struktur im modernen Software-Design. Ein Kästchen / Block kann dann später durch immer speziellere / kompliziertere Blöcke ersetzt werden. Diese Entwicklungs-Technik von Programmen – vom Allgemeinen zum Speziellen (von oben nach unten) – wird **Top-down**-Strategie genannt. Sie entspricht der Deduktion (Denktechnik).
Bei der entgegengesetzten Entwicklungs-Technik geht man von fertigen / funktionierenden Befehlen / Blöcken aus und setzt sie zu immer umfangreicheren / ziel-orientierten Programmen zusammen (quasi: von unten nach oben). Diese Technik wird **Bottom-up** genannt und entspricht der Induktion.
In der Programmier-Praxis werden beide Strategien verwendet. Oft passiert das auch gleichzeitig. Die Top-down-Technik ergibt schnell übersetzbare Programme, auch wenn diese meist noch nicht viel leisten. An den Details muss dann Schritt für Schritt gearbeitet werden.

Die Welt der Programmierung war früher eine elitäre Sonderwelt für Freaks, Nerds oder Geeks. Damals entwickelten sich die ersten Züge einer – für Laien fas unverständlichen – Fachsprache. Heute ist die Programm-Entwicklung eine weitverbreitete Kulturtechnik. Trotzdem sind viele Begriffe und Zusammenhänge für Nicht-Profis schnell unverständlich. Einige der wichtigen Zusammenhänge und Begriffserklärungen aus dieser Begriffswelt sind in der folgenden Abbildung zu entnehmen.

Begriffe zu 'Programmcode': Zusammenhänge, Synonyme



Q: de.wikipedia.org (VÖRBY)

4.1. Kommentare

Man kann sich ruhig angewöhnen, Programme gleich von Anfang an, in die drei Abschnitte zu teilen und mit passenden Überschriften zu versehen. Natürlich sind diese nicht wirklich Teil des Programms. Man nennt Hilfstexte in Quelltexten, die zur Beschreibung von Befehlszeilen oder Programm-Strukturen dienen – **Kommentare**. In Python werden Kommentare durch die Raute begonnen und nehmen dann den Rest der Zeile ein. Bei größeren und wichtigen Programmier-Projekten werden an den Anfang des Quelltextes auch Inhalts-, Urheber- und Versions-Angaben notiert.
Kommentare werden – zu mindestens in der Stadard-Einstellung von IDLE – rot gedruckt.

kommentierter Quelltext	Erläuterungen

`# ========================================`
`# Programm zur Berechnung einer Summe`
`# ----------------------------------------`
`# Autor: Drews`
`# Version: 0.1 (01.09.2015)`
`# Freeware`
`# ========================================`
`# Eingabe(n)`
 	
`# Berechnung der Summe (Verarbeitung)`
 	
`# Ausgabe(n)`
	← hier könnte echter Quell-Text stehen
	
	← hier auch wieder

Für besondere Zwecke kann man sich auch der mehrzeiligen Kommentare bedienen.
Mehrzeilige Kommentare beginnen und enden mit drei Anführungszeichen (**" " "**). Alles zwischen diesen wird nicht von Python ausgewertet. Die Anführungszeichen werden deshalb auch gerne benutzt, um kleinere oder größere Quelltext-Stück von der Python-Interpretation auszuschließen.
Wenn ich z.B. einen Quelltext verbessern möchte, dann will ich vielleicht den alten zuerst einmal noch (sicherheitshalber) behalten. Ich setze einfach davor und dahinter die Dreifach-Anführungszeichen und kann meinen neuen Quelltext davor oder dahinter eingeben. Ein kleiner Kommentar hilft dann auch später zu erkennen, was alter und neuer Quelltext war und ist. Dieser muss aber innerhalb der beiden Dreifach-Anführungszeichen-Gruppen stehen.

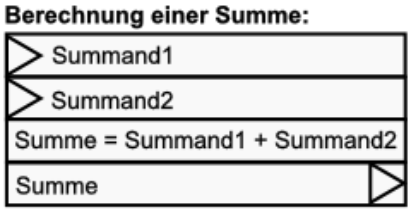
```
…
# Ausgabe(n)
for i in range(4):
    print("")
…
```

```
…
""" alter Quellcode
# Ausgabe(n)
for i in range(4):
    print("")
"""
# Ausgabe(n)
for _ in range(4):
    print("")
…
```

Mehrzeilige Kommentare werden in der Standard-Editor-Einstellung grün eingefärbt.
Bis jetzt macht unser obiges Programm noch nichts. Trotzdem können wir es schon mal testen. Dazu speichern wir erst einmal ab (z.B. mit **[Strg] + [s]**) und starten den Programmaufruf über "Run" "Run Module" oder mit der **[F5]**-Taste.
Geht alles glatt bei der Übersetzung und Ausführung des Programms, meldet sich IDLE ohne einen Fehlerhinweis.

4.2. Planung eines Programms und Umsetzung in Python

Ausgehend vom allgemeinen EVA-Struktogramm überlegt man sich nun, welche konkreten Eingaben, Verarbeitsschritte und Ausgaben für ein spezielles Problem notwendig sind.
Bei der Summierung wissen wir, dass wir zwei Summanden brauchen und diese zur Summe über den +-Operator zusammengefügt werden.
Das Struktogramm ist denkbar einfach:



Struktogramm für die Summenbildung

An dieser Stelle soll darauf hingewiesen werden, dass Struktogramm praktisch an keine spezielle Programmiersprache gebunden ist.
Ob wir das Struktogramm in JAVA, BASIC, PASCAL oder eben Python umsetzen, ist sachlich egal. Vielfach werden die Programme ganz ähnlich aussehen. Die Feinheiten jeder Programmiersprache sind dann schnell dazugelernt.

Umsetzung des Struktogramm "Summenbildung in die Programmiersprache …"

… BASIC

```
DIM AS INTEGER summand1, summand2
DIM AS INTEGER summe

INPUT "1. Summand:", summand1
INPUT "2. Summand:", summand2

summe=summand1+summand2

PRINT "Summe: " &summe
END
```

… PASCAL

```
program summe;

var summand1, summnad2: integer;
var summe: integer;

begin
  write("1. Summand: ");
  readln(summand1);
  write("2. Summand: ");
  readln(summand2);

  summe:=summand1+summand2;

  writeln("Summe: ",summe);
end.
```

… JAVA

```
public class SummeBerechnung
{
    public static void main()
```

… C

!!! falsches Programm!!!

… PROLOG

```
/* ProgrammSumme */
summe(1,1).
summe(N,S):- N > 1, M is N -1,
           summe(M,ZS), S is ZS + N.
```

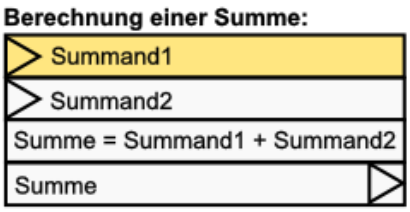
… FORTH

…	… Python
	summand1=int(input("1. Summand: ") summand2=int(input("2. Summand: ") summe=summand1+summand2 print("Summe: ",summe)

Aufgaben:

1. ***Übernehmen Sie den BASIC- und den PASCAL-Quelltext jeweils auf die linke Seite eines Blattes! Lassen Sie etwas Platz zwischen den Zeilen! Kennzeichnen Sie die Abschnitte, die jeweils zu den 4 Blöcken des Struktogramms gehören! Schreiben Sie zu den einzelnen Anweisungen auf, was diese aus Ihrer Sicht machen! (Praktisch: Kommentieren Sie die Programme!)***
2. ***Vergleichen Sie die Umsetzungen miteinander!***

Nun können wir in unseren Programm-Rumpf (mit den Kommentaren) schrittweise Befehle ergänzen. Sinnigerweise fängt man bei den Eingaben an und endet bei den Ausgaben. Aber auch andere Vorgehensweisen sind denkbar und obliegen dem Gutdünken des Programmiers. Jeder muss da seinen eigenen Stil finden.



Die Eingabe realisieren wir mit der **input()**-Funktion. Diese fragt eine Eingabe auf der Konsole bzw. der IDLE-Oberfläche ab.
Zuerst würde die Zeile:

```
…
# Eingabe(n)
summand1 = input()
…
```

(unter dem Eingabe-Kommentar) völlig ausreichen. Der Nutzer sieht auf der Konsole allerdings nur einen blinkenden Cursor und weiss gar nicht, was das Programm von ihm will. Besser ist es einen kleinen Begleittext mit anzugeben.

```
>>>
```

Diesen kann man in das Klammerpaar von **input()** notieren. Der Text selbst muss in Anführungszeichen (**" "**) oder einfachen Hochkommata (**' '**) gesetzt werden.

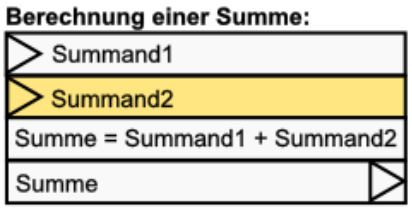
```
…
# Eingabe(n)
summand1 = input("Geben Sie den ersten Summanden ein: ")
…
```

An dieser Stelle bietet sich ein erster echter Test unseres Programms an. Also schnell abspeichern (mit [Strg] + [S]) und die Abarbeitung (mit [F5]) aufrufen. Nun sollt auf der Konsole der Eingabe-Hinweistext zu sehen sein und wir wissen, was wir zu tun haben.

```
>>>
Sie den ersten Summanden ein:
```

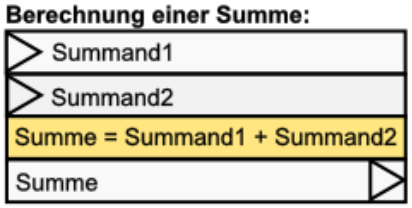
Im Fehlerfall müssen wir wieder zum Quelltext wechsel und die Fehler beseitigen. Dann wird wieder gespeichert und ausprobiert. Dieses muss man solange wiederholen, bis dieser Teil des Programms funktioniert.

Dann kann man sich an den nächsten Programm-Abschnitt machen.



Also praktisch das Gleiche noch mal für den zweiten Summanden. Am Einfachten geht das über das Kopieren der letzten Programmzeile. Wichtig ist bei Kopier-Aktionen immer, sofort die notwendigen Änderungen vorzunehmen. Ansonsten hätten wir zwei Programmzeilen, die sich um die Eingabe des ersten Summanden kümmern.

Der nächste Programmschritt – die eigentliche Verarbeitung der eingegebenen Daten – folgt dann unter dem Kommentar " Berechnung der Summe (Verarbeitung)".



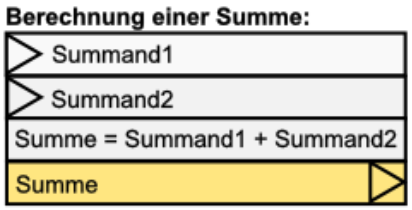
Die Gleichung ist sofort verständlich. Wichtig ist hier, dass das Ergebnis immer links stehen muss. Das Gleichheitzeichen wird unter Programmierern meist als Ergibt-Zeichen (in Pascal z.B.: **:=**) bezeichnet.

Erfahrene Programmierer geben natürlich gleich mehrere Zeilen ein und testen dann.

Ein praktische Strategie ist es auch, vor der internen Verarbeitung der eingegebenen Werte eine kleine Kontroll-Ausgabe zu programmieren. Diese kann noch ohne Texte und Formatierungen erfolgen – es geht nur darum die Korrektheit der Eingaben zu prüfen.

```
…
# Eingabe(n)
summand1 = input("Geben Sie den ersten Summanden ein: ")
summand2 = input("Geben Sie den zweiten Summanden ein: ")

# Berechnung der Summe (Verarbeitung)
summe = summand1 + summand2
…
```

Da die Berechnung nicht so kompliziert erscheint, gehen wir gleich auch noch die Ausgabe an. Wer aber unbedingt will kann wieder einen Programmlauf starten. Allerdings wird er noch kein Ergebnis zu sehen bekommen.



Als erstes reicht uns mal die Ausgabe des Variablen-Wertes von summe. Die Ausgabe-Funktion heißt **print()**.

```
…
# Ausgabe(n)
print(summe)
…
```

Hier ist wieder eine gute Gelegenheit das Programm zu testen, ansonsten gehen wir gleich ans Verfeinern.

Die einfache Ausgabe einer Zahl ist wenig informativ. Zwar können wir vielleicht aus den beiden Eingaben so ungefähr ableiten,

```
>>>
```

was berechnet wird, aber

Wer das Programm getestet hat, wird meist eine böse Überraschung erleben. Das Programm berechnet irgendwas, aber nicht die Summe. Mathematisch scheint aber doch alles richtig zu sein. Warum es zu scheinbar falschen Berechnungen kommt, klären wir gleich.

Auch bei der spärlichen Ausgabe bietet sich also ein kleiner Begleittext an, damit der Nutzer auch genau weiss, was die Ausgabe bedeutet.

```
…
# Ausgabe(n)
print("Die Summe ist gleich: ",summe)
…
```

Für echte Konsolen-Programme ergänzen wir ganz unten immer noch ein **input()**. Damit die Konsole nicht gleich nach der Ausgabe geschlossen wird. Dieses input hat keinen anderen Zweck! Es wird einfach auf ein Enter gewartet und die Konsole schließt danach.
Somit sieht unser erstes Programm insgesamt so aus:

```
# =====================================
# Programm zur Berechnung einer Summe
# -------------------------------------
# Autor: Drews
# Version: 0.1 (01.09.2015)
# Freeware
# =====================================
# Eingabe(n)
summand1 = input("Geben Sie den ersten Summanden ein: ")
summand2 = input("Geben Sie den zweiten Summanden ein: ")

# Berechnung der Summe (Verarbeitung)
summe = summand1 + summand2

# Ausgabe(n)
print("Die Summe ist gleich: ",summe)

# Warten auf Beenden
input()
```

In der Gesamtansicht erkennt man auch gut das sogenannte Highlighting der verschiedenen Programm-Elemente. Dadurch wird der Programm-Text übersichtlicher und Fehler lassen sich etwas schneller finden.

Aufgabe:

1. *Testen Sie das Summen-Programm auch mit Komma-Zahlen und Texten – auch in Kombination untereinander und mit ganzen Zahlen! Was erhalten Sie für Ergebnisse? Was sagt das über die Leistungen von Python aus?*
2. *Speichern Sie das aktuelle Programm noch einmal ab und erstellen Sie sich dann eine weitere Kopie mit "Speichern unter …"! Verwenden Sie den Namen "Subtraktion.py"!*
3. *Verändern Sie das Programm nun so, dass es eine Subtraktion durchführt! Verändern Sie auch alle Variablen-Namen, Ausgaben usw. usf. für das neue Programm!*

Spätestens jetzt fällt uns auf, dass das Programm gar nicht exakt rechnet. Es kann zwar mit Ganzzahlen, Kommazahlen und Texten umgehen, aber statt die Summe zu berechnen, werden die Eingaben nur einfach aneinander gehängt.
Das Problem liegt nicht an der Berechnung der Summe, wie man vielleicht tippen würde. Das Problem ergibt sich daraus, dass Windows i.A. und Python im Speziellen bei Eingaben zuerst einmal immer einen Text liefert. Texte werden beim Summieren einfach nur hintereinandergehängt – wir sagen auch verkettet.
Diesem Problem kann man nun auf zwei verschiedenen Wegen Paroli bieten. In der ersten Variante überlassen wir Python die Arbeit der Erkennung, was eingegeben wurde. Die Funktion **eval()** macht genau dies. Der Name steht für evaluieren / überprüfen. Die Funktion **eval()** ermittelt mit einer recht guten Treffsicherheit, ob es sich bei den Eingaben und der späteren Verarbeitung um ein Text-Ding oder um Zahlen-Verknüpfung handelt. Den Variablen wird dabei ein passender Datentyp (z.B. Text, Ganzzahl, Kommazahl) zugeordnet. Dazu später noch mehr und auch, wie man die Datentypen gezielt verändern kann (→ 8.2. Datentypen und Typumwandlungen).

```
…
# Eingabe(n)
summand1 = eval(input("Geben Sie den ersten Summanden ein: "))
summand2 = eval(input("Geben Sie den zweiten Summanden ein: "))
…
```

Aufgaben:

1. *Erstellen Sie ein Struktogramm für die Produkt-Bildung von drei Faktoren!*
2. *Schreiben Sie ein Programm, dass aus drei einzugebenen Zahlen das Produkt berechnet! Orientieren Sie sich an dem Struktogramm von 1.! Korrigieren Sie eventuell das Struktogramm, wenn es Probleme beim Testen des Programms gibt!*
3. *Konzipieren und realisieren Sie ein Programm, das zu einer einzugebenden Masse in kg die Massen in mg, g und t ausgibt!*
4. *Erstellen Sie Struktogramm und Programm zur Berechnung von cm^2, dm^2, ar, ha und km^2 aus einer Angabe in m^2! Die Einheiten dürfen ausgeschrieben werden!*
5. *Erstellen Sie ein Struktogramm und dann das Programm zur Umrechnung einer °C-Temperatur in die zugehörige KELVIN-Temperatur!*
6. *Ergänzen Sie das Programm von 5. noch um die Ausgabe der Temperatur in °Ra (RANKINE) und °Ré (REAUMUR; sprich: [reo'mü:r]!*

für die gehobene Anspruchsebene:

7. *Gibt es eigentlich noch andere Temperatur-Skalen? Wenn JA welche und, wenn NEIN, warum nicht. Wenn es weitere Skalen gibt, dann erweitern Sie das Programm von 6. um diese Skalen!*

Testwerte für die Temperatur-Umrechnungs-Programme:

°C	°F	K	°Ra	°Ré
-273	-459,7	0	0	-218,5
-12	10,4	261	469,8	-9,7
126	258,8	399	718	100,7
37,8	100,0	311	560	30,3
-51	-59,7	222	400	41
25	76,7	298	536,4	20

°C	°F	K	°Ra	°Ré
0	10,4	273	491,4	0
-17,8	0	255,4	459,7	-218,5
20	67,7	293	527,4	16
0	32	273,1	491,7	0
-217,6	-359,6	55,6	100	-174
125	257	398,1	716,7	100

Für die Temperaturen in °Ra und °Ré werden, je nach Quelle auch °R als Einheits-Zeichen verwendet. Da dies zu Verwechslungen führen kann, werden hier die ausführlichen und damit eindeutigen Einheiten-Symbole benutzt.

Der Anwender steht immer im Mittelpunkt –
und dort steht er jedem –
und vor allem dem Programmierer – im Weg!

5. Was passiert mit dem Quelltext?

Python ist eine höhere bzw. Problem-orientierte Programmiersprache
vom Menschen gut lesbar
muss für die Nutzung auf dem Computer in Maschinencode (Nullen und Einsen) übersetzt werden, dies kann aber ein Computer wieder auch selbst realisieren; Übersetzung benötigt Zeit und muss relativ universell erfolgen, deshalb ist der Vorgang recht langsam
meist sind die fertigen Programme dann auf verschiedenen Computer-Typen und Betriebssystem-Welten nutzbar

Maschinen-orientierte Programmiersprachen (z.B. Assembler od. Bytecode) sind vom Menschen nur sehr schwer lesbar und wenig verständlich
kaum Übersetzung notwendig, deshalb meist sehr schnell und effektiv
meist auf einzelne Computer-Typen und eine Betriebssystem-Welt zugeschnitten

Übersetzung einer Programmiersprache in Maschinen-Code kann auf zwei Arten erfolgen:
entweder mit

- **Interpreter** Übersetzung erfolgt während der Nutzung; es werden Zeile für Zeile (bzw. Blöcke) einzeln übersetzt und ausgeführt; bei nochmaliger Nutzung muss wieder neu interpretiert werden; immer Quell-Text und Interpreter (bei Python die Shell) zur Ausführung notwendig

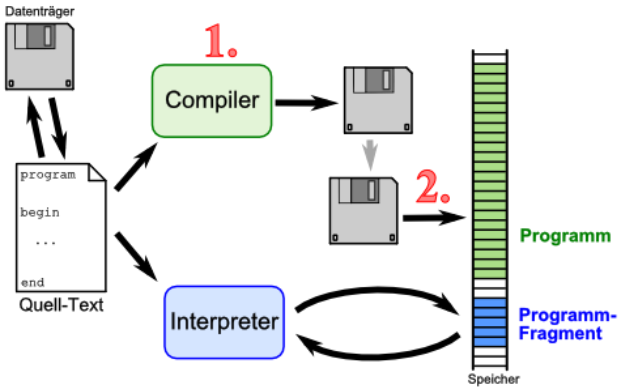
oder

- **Compiler** Übersetzung erfolgt hier in einem Stück vor der Nutzung; es wird zumeist ein echtes ausführbares Programm (EXE-Datei) erzeugt; das ausführbare Programm kann beliebig oft und ev. auch parallel ausgeführt werden; Nutzung ohne den Quell-Text und den Compiler möglich

prinzipiell ist Python eine Interpreter-Sprache, der Quell-Text wird also während der Benutzung / dem Aufruf in Maschinen-Befehle umgesetzt

Der Quell-Text wird vom Interpreter zuerst in einen Zwischen-Code (Byte-Code) übersetzt
ist Maschinen-unabhängig, d.h. jeder Interpreter erzeugt den gleichen Bytecode aus einem Quell-Text; für den Nutzer normalerweise unsichtbar
exakterweise müsste man hier eigentlich sagen, der Python-Interpreter compiliert (kompiliert) den Quell-Text in den Bytecode
der Zwischen-Code wird dann vom Interpreter, der nun als virtuelle Maschine fungiert, Maschinen-abhängig abgearbeitet

wenn der ***py***-Quelltext eines Moduls oder aus einer anderen py-Datei importiert wird, dann wird eine (sichtbare) Bytecode-Datei abgelegt
hat die Endung ****.pyc***; zumeist liegen die Dateien im Unterordner __**pycache**__ des Python-Systems



Definition(en): Interpreter
Ein Interpreter ist ein Programm, das den Quelltext zur Laufzeit einliest, analysiert und ausführt. Der Interpreter wird bei jeder Abarbeitung des Programms (Quelltextes) gebraucht.

Definition(en): Compiler
Ein Compiler ist ein Programm, das den Quelltext in ein für sich ausführbares Maschinen-Programm übersetzt. Der Compiler wird zur Abarbeitung des Programms nicht mehr gebraucht.

eval(ausdruck)
interpretiert den angegebenen Ausdruck

exec(text)
interpretiert den angegebenen Text (der ein vollständiges Python-Programm sein kann) und führt den Code aus
text kann also eine Folge von Python-Anweisungen, Importen, Funktion(sdefinietion)en usw. usf sein

compile()

die Interpreter-Technik bringt auch einige Probleme mit sich
ein solches Problem ist die Fehl-Interpetation von Eingaben; im interaktiven Modus für den Nutzer nachvollziehbar, bringt es den Endnutzer (ohne Kenntnis des Quell-Textes und vielleicht auch ohne Python-Erfahrung) leicht um den Verstand
genaueres dazu später bei den Eingaben (→ 6.2.1. unschöne Eingabe-Seiten-Effekte in Python-Programmen)

5.1. Und es geht doch! – aus dem Python-Quelltext eine EXE erstellen

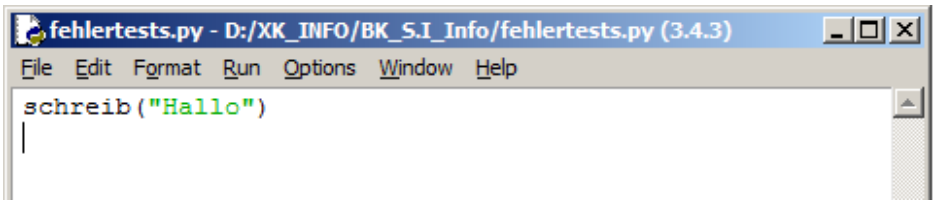
PyInstaller
erzeugt eine exe-Datei, die eine Python-Laufzeitumgebung und das eigene Script / Programm enthält
gibt es für Windows und für Linux
http://www.pyinstaller.org/

aus dem PyInstaller weiterentwickelt wurde der
McMillan's Installer
entwickelt; also gleiches Prinzip
Weiterentwicklung ???

py2exe-Modul
sehr häufig benutzt
es gibt aber bestimmte Einschränkungen, die ev. beim Programmieren beachtet werden müssen
hat manchmal auch Probleme mit bestimmten Modulen / Bibliotheken

5.2. Fehlersuche

Gleich bei den ersten Übungen tauchen erfahrungsgemäß die ersten Fehler beim Interpretieren des Quellcodes auf.



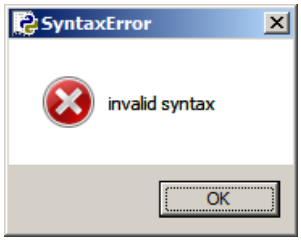
zu interpretierender Quelltext

```
>>>
Traceback (most recent call last):
  File "D:/XK_INFO/BK_S.I_Info/fehlertests.py", line 1, in <module>
    schreib("Hallo")
NameError: name 'schreib' is not defined
>>>
```

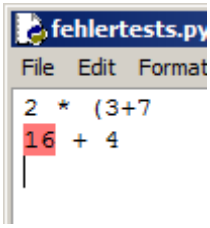
angezeigte Fehlermeldung

Manche Fehler sind offensichtlich. Vor allem dann, wenn noch Fehler-Informationen mit ausgegeben werden.
Hier z.B. wurde in der Zeile 1 (**`line 1`**) in der interpretierten Datei ein Namensfehler (**`NameError:` …**) gefunden. Der Name "schreib" ist nicht definiert.

Aber manchmal ist nur ein rotes Viereck am Anfang der Zeile zu erkennen. Die Fehlermeldung besagt aber, es handle sich um einen Syntax-Fehler.



Besonders verzwickt wird es, wenn man auch noch sicher ist, dass in dieser Zeile alles richtig ist. In so einem Fall lohnt immer ein Blick in die Zeile davor. Meist ist hier der Fehler zu finden. Für den Interpreter ist die vorherige Zeile syntaktisch noch nicht abgeschlossen. Die neue Zeile – in der der Fehler angezeigt wird – ist aber eben nicht passend zur vorherigen Zeile und somit ergibt es eine Fehlermeldung.



Die häufigsten Fehler sind fehlende Operatoren (+, -, *, /, …) oder Operanden (Zahlen bzw. Variablen).
Als nächste Fehlerquelle kommen fehlende schließende Klammern oder zu wenig öffnende in Frage.
Das Schöne ist, das uns Python die Klammernpaare im Editor kurzzeitig nach dem Eintippen anzeigt. Sollte da mal in einer Funktion nicht alles grau werden, dann fehlt zumeist irgendwo eine Klammer.
Eine unschöne Sache ist auch für uns Deutsche die Umsetzung des Dezimaltrenners **Komma** in einen **Punkt**. Ein Komma hat völlig andere Wirkungen – dazu später mehr. Hier ist erst einmal wichtig, darauf zu achten.
Ähnlich, wie bei den Klammern müssen Texte immer mit den beginnenden Zeichen – am Besten doppelte Anführungszeichen – abgeschlossen werden.

Fehlermeldungen können aber auch sehr kryptisch sein. Sie sind aus der Sicht des Interpreter eindeutig, aber eben nicht aus der Sicht des Programmierers, der ausversehen einen Tipp-Fehler gemacht hat.

```
>>>
Traceback (most recent call last):
  File "D:/XK_INFO/BK_S.I_Info/fehlertests.py", line 1, in <module>
    9+4(2*6)
TypeError: 'int' object is not callable
>>>
```

es handelt sich bei diesem Fehler nicht um einen "Typ"-Fehler sondern um eine fehlendes Mal-Zeichen vor der Klammer

Aufgaben:

1. ***Suchen Sie die Fehler in den folgenden Code-Ausschnitten! Berichtigen Sie diese und probieren Sie in Python aus, ob der Interpreter den Code akzeptiert!***

a)
```
23 * (3 + 2
12 + 83 -
```

b)
```
9 + 4 3
7 * 23+4) + 8
```

c)
```
17 \ 4 * 12
21,5 * 3
```

d)
```
.5 + 4 * 3
-3*+5*,333333
```

e)
```
print('Hallo Nutzer!")
```

f)
```
PRINT(21 + "E")
```

2. ***Erstellen Sie ein kleines Python-Programm mit mindestens 5 Fehlern! Drucken Sie das Programm zweimal aus und korrigieren Sie auf einem Blatt die Fehler (→ Lösungsblatt)!***
3. ***Tauschen Sie das 2. Blatt einem Nachbarn und korriegieren Sie dessen fehlerhaftes Programm!***
4. ***Vergleichen Sie die gefundenen Fehler mit dem Lösungsblatt!***

Den Prozess der Fehlerbereinigung nennt man auch Debuggen (dt.; engl.: debugging). Der Begriff besagt, dass die Läuse / Käfer (engl.: bugs) aus dem Quelltext bzw. dem fertigen Programm entfernt werden.
Man unterscheidet technisch zwischen:

- **Syntax-Fehlern** — praktisch Fehler in der Rechtschreibung des Quelltextes
- **Laufzeit-Fehlern** — sind Fehler, die erst beim Ausführen des Programm auftreten
- **logische und semantische Fehler** — sind inhaltliche Fehler oder auch grammatikalische Fehler im Quell-Text

Debugging ist echte Detektiv-Arbeit. Manche Fehler sind wahre Künstler im Verstecken und Verschleiern der wirklichen Fehlerstellen. Ab und zu muss man sich einfach aus der Arbeitsebene lösen und von oben auf das Programm schauen. Auch das Pause-Machen oder Schlafen-Gehen hat sich als Wunderwaffe gegen Programmierfehler herausgestellt.
Heute gibt es sogenannte Debugger – Programme, die zu mindestens eine große Menge klassischer Programmierfehler in Quelltexten finden. Letztendlich wird aber immer der Mensch die letzte prüfende Instanz sein. Somit obliegt es immer ihm, ob ein Programm fehlerarm ist. Wirklich richtig fehlerfrei wird man wohl kein komplexes Programm je hinbekommen.
Jedes Programm sollte immer erst frei gegeben werden, wenn seine Funktionsweise vom Programmierer verantwortet werden kann.
Leider planen viele Programmierer für das Debuggen zu wenig Zeit ein. Ein guter Orientierungswert sind ein Drittel der gesamten Programmierzeit. Das erste Drittel geht für die Konzeption und den Algorithmen-Entwurf drauf und das nächste für die Quelltext-Erstellung.
Bleibt die Dokumentation übrig. Dafür muss man noch mal ein Drittel einplanen. Und damit sind wir da, wo fast alle Programmier-Projekte enden – in der deutliche Überschreitung der Zeit-Vorgaben.

**Ein Programm ist dann korrekt,
wenn es zu jeder zulässigen Eingabe
eine korrekte Ausgabe produziert.**

Das impliziert, dass man beim Testen alle zulässigen Eingaben durchlaufen müsste und die vom Programm produzierten Ausgaben mit anderen Referenzwerten vergleichen müsste.
So etwas ist aber maximal bei kleinen Programmen mit wenigen Eingaben / Ausgaben machbar. In der Praxis wird man sich mit einer gut gewählten Menge von Eingaben und Ausgaben zufrieden geben müssen. Besonders effektiv ist die Test-Menge an Eingabe- und Ausgabe-Paaren, wenn diese zufällig ausgewählt werden. Dann besteht zu mindestens eine gewissen Wahrscheinlichkeit, dass das Programm ordnungsgemäß funktioniert.

Aufgaben:

1. ***Prüfen Sie zuerst auf dem Papier die folgenden Programme! Finden Sie die Fehler und berichtigen Sie diese sinnvoll!***

a)
```
print "Sternenreihe"
for i in range 1, 11
    print(i = , i)
    print(i#*
Programmende
print(ende)
```

b)
```
input(Dein Name:
print " " #Eingabe lautete:
print(name)
print "fertig"
input()
```

2. ***Diskutieren Sie die Fehler und Berichtungen mit einem Partner aus dem Kurs!***
3. ***Tippen Sie nun den korrigierten Quelltext ein und probieren Sie ihn aus! Finden Sie noch weitere Fehler des Originaltextes, Ihrer Korrekturen oder durch die (durch Sie getätigte) Eingabe?***

5.3. Stil-Regeln für Python-Programmierer

Warum denn nun schon Vorschriften zum richtigen Schreiben von Python-Prgrammen – wir haben doch noch gar nicht richtig programmiert?! Natürlich ist der Einwand richtig. Viele der nachfolgenden Regeln und Vorschriften hören sich für einen Anfänger völlig abgehoben an. Aber – wer nicht von Anfang an die wichtigen Regeln einhält, der wird sich später nur schwer umgewöhnen.
Im Normalfall wird die Einhaltung der Vorschriftung und Stil-Regeln von den Kursleitern bei der Bewertung von Programmen mit beachtet. Also wundern Saie sich nicht, wenn ein super funktiuonierendes Programm nicht die volle Punktzahl bekommt, weil es einfach nicht lesbar und verständlich ist.
Versetzen Sie sich in die Lage Ihres späteren Arbeitgebers oder eines Arbeitgebers. Da programmiert ein Programmierer einfach drauf los und erzeugt auch funktionierende Programme. Nun muss irgendetwas umgestellt werden oder ein Fehler ist aufgetaucht und muss korrigiert werden. Und nun liegt da ein Quelltext vor Ihnen, mit dem niemand etwas anfangen kann, weil er unübersichtlich oder kryptisch unverständlich geschrieben ist. Die übliche Reaktion ist: Das Programm wird neu geschrieben, das dauert genausolange, wie den alten Code aufzubröseln. Was ist das für eine Resourcen-Verschwendung. Kosten über Kosten, nur weil so ein Neunmalklug den Angeber spielen will. So geht es also nicht.
In der PEP 8 ("Python Enhancement Proposal #8) sind viele Stil-Regeln empfohlen (→ https://www.python.org/dev/peps/pep-0008/). Für uns Anfänger gelten Sie als Gesetze.
Hier die wichtigen Regeln:

Leer-Räume:

- für Einrückungen Leer-Zeichen verwenden (statt Tabulator)
- Einrückungen immer um 4 Zeichen
- Zeilen-Länge unter 80 Zeichen
- definierte Klassen und Funktionen werden mit 2 Leerzeilen von einander getrennt
- innerhalb einer Gruppe (z.B. Funktionen einer Klasse, zusammengehörende Funktionen) wird nur eine Leerzeile benutzt
- Funktions-Aufrufe und Feld-Indizies ohne Leerzeichen innerhalb der Klammern (maximal ein Leerzeichen zwischen den Einzel-Objekten)
- ein Leerzeichen vor und hinter dem Zuweisungszeichen (=)

Bezeichner:

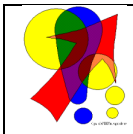
- Variablennamen, Funktionen, Attribute in Kleinbuchstaben (ev. mit Unterstrichen)
- geschützte Attribute mit führendem Unterstrich
- private Attribute mit führendem doppelten Unterstrich
- Klassen und Exceptions mit Großbuchstaben beginnend (und ev. auch intern für Wörterbeginn)
- Modul-weite Konstanten vollständig in Großbuchstaben

Ausdrück und Anweisungen:

- keine einzeiligen if-Anweisungen, for- und while-Schleifen
- keine einzeiligen Exceptions
- Vorrang für from xxx import yyy (statt import yyy)
- Reihenfolge der Importe (zuerst Standard-Module, dann Fremdanbieter-Module, zuletzt eigene Module); möglichst alphabetisch sortiert
- Test auf eine leere Liste, leere Felder, leere Werte mit **somelist** (nicht mit **len**(..) == 0 usw.)

weitere Empfehlungen / Regeln:

- gleichartige Programm-Abschnitt (besonders, wenn sie häufiger verwendet werden) als Funktionen auslagern
- komplizierte Ausdrücke in Funktionen auslagern
- Teil-Listen und Elementzugriff auf Listen über :-Notierung (Slicing)
- möglichst keine else-Blöcke nach for- oder while-Schleifen
- alle Blöcke in Exceptions nutzen



Bemerkungen zur verwendeten Schreibung in diesem Skript:
Ich bin bemüht die Regeln so gut wie möglich einzuhalten. Viele Programme sind schon älter und werden nach und nach umgestellt. Das kostet viel Zeit, Zeit, die ich derzeit erst einmal lieber für inhaltliche Erweiterung aufwenden möchte. Das andersartige Schreiben von Bezeichnern etc. ist zuerst einmal ein Schönheitsfehler.
Die Exaktheit eines Programms geht bei mir vor Schönheit.
Für Hinweise auf dringend notwendige Umstellungen und echte Fehler bin ich immer dankbar.

Aufgaben:
… folgen später, hier Verweise für Voreilige:

→
→
→

6. grundlegende Sprach-Elemente von Python

Nicht dass Sie beim Betrachten des ersten Unterabschnittes denken, ich habe das EVA-Prinzip schon wieder vergessen oder nicht richtig verstanden. Nein, die Umsortierung für die einzelnen Kapitel (**E**ingabe – **V**erarbeitung – **A**usgabe) erfolgt hier aus praktischen Gründen. Damit man mit dem Nutzer kommunizieren kann, müssen z.B. Informationen auf dem Bildschirm erscheinen. Die Ausgaben sind dafür passende Programm-Elemente. Eigentlich immer soll irgendetwas (zum Testen) ausgegeben werden. Und das sind zuerst auch nur Zwischenwerte und Rohergebnisse. Später werden die Daten dann Nutzer-freundlich präsentiert.
Deshalb verwende ich hier also die unorthodoxe Reihenfolge: **A**usgaben (→ 6.1. Ausgaben) – **E**ingaben (→ 6.2. Eingaben) – **V**erarbeitung (→ 6.3. Verarbeitung).

6.1. Ausgaben

Einige Möglichkeiten der Ausgabe mit dem Befehl bzw. der Funktion print haben wir uns bei den ersten einfachen Programmen (→ 4. erste einfache Programme mit Python) schon angesehen.
Nun wollen wir uns weitere Feinheiten ansehen und auch einige zusätzliche Ausgabe-Möglichkeiten kennenlernen.
Trotzdem werden hier nur die wichtigsten / praktischsten Möglichkeiten aufzeigen. Für mehr empfehle ich die Hilfe, die viele andere Variationen und Möglichkeiten beschreibt. Der **print()**-Befehl bietet viele Gelegenheiten, sich eine gut verständliche, ordnungsgemäß gestaltete Ausgabe zu produzieren.
So lassen sich viele einzelne Teilbereiche (quasi die Argumente) durch Kommata abgrenzen. Jeder ist für sich wieder weiter differenzierbar. Letztendlich läuft es darauf hinaus, entweder eben nur eine (**print** mit nur einem Argument) oder mehrere Texte zu erstellen.
Zahlen und Berechnungen lassen sich Kommagetrennt ebenfalls in einer Ausgabe unterbringen. Es kann dabei frei gemischt werden.

```
>>> print("Hallo ", "Nutzer!")
Hallo Nutzer!
>>>
```

```
>>> name="Klaus"
>>> print("Hallo ", name, ", wie geht's?")
Hallo Klaus, wie geht's?
>>>
```

```
>>> a=5
>>> print("Hallo ", name, ", 2*",a," ist ",2*a)
Hallo Klaus, wie geht's?
>>>
```

```
>>> print("Hallo ", name, ", 2*",a," ist ",2*a)
Hallo Klaus, 2* 5 ist 10
>>>
```

Jetzt könnte man natürlich auf die Idee kommen, die etwas gewöhnungsbedürftigen Ausgaben aus Texten und Zahlen in einzelne **print()**-Anweisungen zu stecken.

Überdichtlicher ist das in jedem Fall:

```
>>> print("Hallo "); print(name); print(", 2*"); …
Hallo
Klaus
, 2*
5
ist
10
>>>
```

Aber – und da steckt der Teufel im Detail – jede print-Anweisung erzeugt am Ende den schon erwähnten Zeilenumbruch.

Die **print()**-Anweisungen erzeugen immer eigenständige Zeilen. Dagegen gibt es zwar einen Trick, den schauen wir uns genauer an, wenn wir die Standard-Möglichkeiten besprochen haben.

Durch die Komma-Trennung werden die einzelnen Ausgaben separiert und einzeln abgearbeitet. Texte eben sofort angezeigt, Zahlen ausgegeben und Berechnungen erst ausgeführt und dann ausgedruckt.

Häufig möchte man sich die Ausgabe-Zeile in Ruhe zusammenbasteln und dann als Ganzes über den Variblen-Aufruf oder von der **print()**-Anweisung anzeigen lassen.

Dazu müssen alle Teile in Text-Form vorliegen – auch Zahlen oder Berechnungen (gemeint sind natürlich die Ergebnisse).

```
>>> nutzer="Klaus"
>>> zeile="Hallo " + nutzer + "!"
>>> zeile
'Hallo Klaus!'
>>> print(zeile)
Hallo Klaus!
>>>
```

Die Zahlen oder eben Berechnungs-Ergebnisse lassen sich mittels **str()**-Anweisung in eine Zeichenkette umwandeln.

Das ist später in graphischen Benutzungsoberflächen ebenfalls so gefordert. Die verschiedenen Zeichenketten werden dann mittels **+**-Operator verketten, aneinanderreihen oder konkatenieren.

Der ***-Operator sorgt für eine Wiederholung der Zeichenkette.

```
>>> aufgabe="5 x 2= "
>>> zeile=aufgabe + str(5*2)
>>> print(zeile)
5 * 2 = 10
>>>
```

```
>>> strichzeile="- "*10
>>> print(strichzeile)
- - - - - - - - - -
>>>
```

Aufgaben:

1. Lassen Sie in der Konsole die folgenden Anzeigen erscheinen!

a) Der eigene Name als nutzer gespeichert und in der folgenden Form ausgegeben:
`nutzer, hallo nutzer!`

b) Die folgende Zeile aus einzelnen Wörtern zusammengesetzt und mit der print()-Anweisung angezeigt!
`Hallo lieber Nutzer, jetzt geht es los!`

c) Die Zeile aus Einzelnworten (jeweils eine Variable!) zusammengesetzt gespeichert als zeile! Die Wiederholungen der Pluszeichen vorher mit dem *-Operator bilden! Die Variable zeile dann 2x mit sich selbst konkateniert!
+ + + aktuelle Nachricht + + +

d) Aufgabe und Ergebnis: 10 + 7 * 3

e) Aufgabe und Ergebnis: 22 (34-18) – (4 + 6) / 2

2. Legen Sie sich einen "Python-Spiker" an! Auf diesem können Sie den Syntax notieren! (Es sind aber keine längeren Programm-Beispiele erlaubt!)

Jede Ausgabe mit **print()** bewirkt ja mit der schließenden Klammer einen Zeilen-Umbruch. Das ist bei vielen Programmen aber gar nicht erwünscht. Vielfach will man mehrere Teil-Ausgaben in einer Zeile hintereinander machen.
Im nachfolgenden – etwas vorgreifenden (!) – Programm-Beispiel (mit einer Schleife / Wiederholung) wird mehrfach ein **print()** gemacht.

```
# FABONACCHI-Funktion bis 10 mit Tupeln
f1, f2 = 0, 1
while f2 < 10:
    print(f2)
    f1, f2 = f2, f1 + f2
input()
```

Uns interessiert hier nur die Zeile mit der **print**-Anweisung. Die Berechnungen wurden hier mit Absicht extra kryptisch gewählt, um nicht anderen Besprechnungen vorzugreifen.

Jedes Mal nach der Ausgabe von f2 wird eine neue Zeile begonnen. Bei sehr vielen Ausgaben ist schnell der untere Rand des Ausgabe-Bildschirms erreicht und die oberen Zahlen verschwinden am oberen Rand.
Um diese platzaufwändige Ausgabeform zu verhindern, dass nach jeder print-Ausgabe ein Zeilenumbruch gemacht wird, kann im **print()**-Befehl die **end**-Anweisung eingebaut werden.

```
>>>
1
1
2
3
5
8
```

Sie verhindert den Zeilenumbruch und stellt eine Möglichkeit bereit, zwischen den verschieden Print-Ausgaben einen Zwischentext auszugeben

```
# FABONACCHI-Funktion bis 1000 mit Tupeln
f1, f2 = 0, 1
while f2 < 1000:
    print(f2, end=' .. ')
    f1, f2 = f2, f1 + f2
input()
```

```
>>>
1 .. 1 .. 2 .. 3 .. 5 .. 8 .. 13 .. 21 .. 34 .. 55 .. 89 .. 144 .. 233
.. 377 .. 610 .. 987 ..
```

Man muss nun aber auch beachten, dass im obigen Programm bisher niemals ein Zeilenumbruch gemacht wird. Den braucht man aber auch das eine oder andere Mal – z.B. eben nach Schleifen, die selbst keine Ausgaben mit Zeilen-Umbrüchen enthalten.

Mit der **end**-Angabe nehmen wir praktische eine Steuerung der Ausgabe vor.
Auf Wusch kann aber auch ein Zeilchen-Umbruch eingesteuert werden. Dazu benutzt man als Steuersequenz **\n**.

\' bzw. **\"**, um die Zeichen selbst innerhalb von Texten / Zeichenketten ausgeben zu können
\n für die Erzeugung mehrzeiliger Texte, quasi als Zeilenumbruch (#D)
Zeichenketten können auch in Paare von drei Hochkommata bzw. Anführungszeichen gesetzt werden, dann ist eine extra Notierung von Zeilenumbrüchen (**\n**) nicht notwendig.
So können also Texte über mehrere Zeilen notiert und ausgegegeben werrden.

```
print("""\
Hauptmenü:
      - [O]ptionen
      - [S]tart
      - [E]nde
""")
```

Aufgaben:

1. *Verändern Sie das FIBONACCHI-Programm so, dass statt der zwei Punkte zwischen den Reihen-Gliedern nun die Zeichen-Sequenz " `==` " ausgegeben wird!*
2. *Im nebenstehenden Programm fehlen (an den Fragezeichen-Positionen) die print-Befehle. Erweitern Sie das Programm so, dass ein Ausdruck:*

   `Summen-Reihe: 0, 1, 3, … fertig!`

   entsteht!

```
summe = 0
n = 0
?
while n < 100:
    summe += n
    ?
    n += 1
?
input()
```

3.

6.1.1. Anpassen von Zahlen für Ausgaben

hierzu gehört z.B. das Runden von Zahlen
auf die genaue Beschreibung der Zahlen-Typen kommen noch (→ 8.2.1. Zahlen)

round()
Runden einer Gleitkommazahl (Zahl mit Nachkomma-Stellen) → Ergebnis bleibt eine Gleitkommazahl

int()
Erzeugen einer Ganzzahl (Zahl ohne Komma-Stellen) aus einer Fließkomma-Zahl (Gleitkomm-Zahl) oder einer Zeichenkette (Achtung!: Zeichenkette muss die exakte Notierung einer Zahl (also z.B. einen Punkt als Dezimal-Trenner) enthalten, sonst gibt es einen Laufzeitfehler)
Strategie zu Abfangen solcher Laufzeitfehler später (→ 8.14. Behandlung von Laufzeitfehlern – Exception's)

float()
Erzeugen einer Gleitkomma-Zahl aus einer Zeichenkette (oder aus einer Ganzzahl)
Achtung! Laufzeitfehler bei Zeichenketten möglich!

str()
Umwandlung einer Zahl in eine Zeichenkette (z.B. zum Verketten von Texten mit berechneten Zahlen)

Aufgaben:

1. *Erstellen Sie ein Programm, in dem jede Ziffer als Text-Variable gespeichert ist! Weiter soll dann eine Verkettung der Variablen erfolgen! Dazu werden zuerst die geraden Ziffern-Text-Variablen verkettet und dann als Text sowie als umgewandelte Ganzzahl ausgegeben!*
2. *In einer erweiterten Version des Programms soll nun eine Zeichenkette aus den ungeraden Ziffern-Variablen gebildet werden! Die Zeichen-Kette der geraden Ziffern soll dann hinter die ungeraden Ziffern konkateniert werden! Zwischen beiden Ketten soll ein Punkt gesetzt werden! Die gebildete Zeichenkette sowie die Umwandlung in eine Fließkommazahl soll ausgegeben werden!*
3. *Zuguterletzt soll noch die Fließkommazahl wieder zurück in einen Text gewandelt werden! Gibt es da Veränderungen in der Ausgabe?*

6.1.2. formatierte Ausgaben

In Python gibt es – wie in vielen Maschinen-näheren Programmiersprachen – mindestens zwei prizipiell unterschiedliche Ausgabe-Formatierungs-Systeme. Bei der einen werden in den Ausgabetext einfach an der passenden Stelle sogenannte Platzhalter eingesetzt. Weiter hinten in der Ausgabe-Anweisung folgen dann in einer Liste die auszugebenen Variablen oder Berechnungen. Diese Variante stellen wir gleich (→ 6.1.1.2. Verwendung von Platzhaltern in Ausgabetexten) vor. Problematisch ist diese Variante bei Erweiterungen oder Änderungen der Ausgaben. Da geht schnell was durcheinander. Da sie aber sehr kompakt ist, wird sie von vielen Programmierern gerne benutzt.
Bei der zweiten Variante der Ausgabe-Formatierung werden die auszugebenen Teile – wenn gewünscht – einzeln über eine spezielle Funktion (→ 6.1.1.1. formatierte Ausgaben mit der format-Funktion) formatiert. Diese **format()**-Funktion ist sehr Leistungs-fähig und kann schnell rein und raus genommen werden. Auch spezielle Anpassungen sind leicht gemacht und ausprobiert und das ohne die anderen Ausgaben-Teile zu beeinflussen. So kann man z.B. erst einmal mit "normalen" Ausgaben arbeiten und diese dann später schrittweise verbessern.
Quasi eine Kombination aus beiden obigen Varianten ist Variante 3 in Python. Hier definiert man sich zuerst einen Text mit speziellen Platzhaltern. Dies sind jetzt geschweifte Klammern **{ }** . Irgendwann später kann man dann den "Lücken"-Text mit Inhalten (Lücken-Füllung) ausgeben (→ 6.1.2.3. Kombination von Platzhaltern und format-Funktion).

6.1.2.1. formatierte Ausgaben mit der format-Funktion

Mir persönlich erscheint die **format()**-Funktion die übersichtlichste Form der Formatierung. Man weiss, wo was steht und wie es genau formatiert werden soll. Zudem ist der Quelltext noch gut lesbar.
Bei der **format()**-Funktion wird der auszugebene Ausdruck als erstes Argument übergeben und als zweites ein Format-Text. Der Format-Text beschreibt die Formatierung der Ausgabe.
Die fertig zusammengestellt format()-Funktion steht dann anstelle der einfachen Ausgabe im print()-Befehl. Dadurch kann man auch zuerst einmal ohne format() auskommen und diese Funktion dann später für eine schönere Ausgabe ergänzen.

Ausgabedaten	**Format-Text** **Format-Spezifizierer**	**Erklärung**	**Bemerkungen**
Text **String** **str**	**"15s"**	der Text wird linksbündig geschrieben und es sind 15 Zeichen dafür reserviert	wenn es mehr Zeichen werden, dann werden diese ausgeschrieben, die Formatierung ist dann aber quasi hinfällig
ganze Zahlen **int**	**"8d"** **"8n"**	die Zahl wird rechtsbündig geschrieben, dafür sind 8 Ziffern-Positionen reserviert	
Komma-Zahlen **Gleitkomma-Zahlen** **float**	**"12.3f"**	die Zahl wird rechtsbündig notiert, insgesamt sind 12 Zeichen reserviert, wobei 3 Positionen Nachkomma-Stellen sind	voreingestelte Genauigkeit liegt bei 6 Nachkommastellen
Binärzahl	**"4b"**		
Oktalzahl	**"10o"**		
Hexadezimal-Zahl **hex**	**"x"**		
Hexadezimal-Zahl **hex**	**"X"**	wie oben, nur dass Großbuchstaben verwendet werden	
Zeichen **Charakter** **char**	**"c"**		
	"8"	meint ganze Zahl, sonst wie oben	
Exponenten-Zahl **wiss. Zahl**	**"e"**		
	g		
	G		
	F		

```
from math import sqrt
fkt_name="Wurzel"
argument=2
fkt_wert=sqrt(argument)
print("Die",format(fkt_name,"12s"),
      "von",format(argument,"6d"),
      "ist gleich",format(fkt_wert,"12.3f"),".")
```

```
>>>
Die Wurzel       von      2 ist gleich        1.414 .
```

Weiterhin lassen sich mit der **format()**-Funktion die Reihenfolgen von Elementen manipulieren. Exakterweise müssten wir hier eigentlich gleich die Objekt-orientierte Schreibung .format() benutzen. Ein Objekt-orientierter Zugriff wird hier aber noch nicht so offensichtlich, so dass wir hier erst einmal darüber hinwegsehen. Später – nach dem Einstieg in die Objekt-orientierte Programmierung (→ 8.11. objektorientierte Programmierung) – wird dann das Spezifische dieser Notierung auch eher klar.

```
print("Ein {1} steht in der {2} des {0}.".format("Hauses" ,"Baum", "Nähe"))
```

Die in den geschweiften Klammern angegebenen Nummern verweisen auf die in der Argumentliste von **.format()** angegebenen Texte.

Somit ergibt sich der gesamte Ausgabetext aus den verschiedenen Textteilen.

```
>>>
Ein Baum steht in der Nähe des Hauses.
```

Sind die Argumente so angeordnet, wie sie eingesetzt werden sollen, dann kann sogar auf die Nummerierung verzichtet werden.
In den geschweiften Klammern dürfen nach der Positionsangabe auch mit Doppelpunkt getrennt weitere Formatierungs-Texte folgen. So bedeutet **{4:8d}** , dass die Ausgabe des vierten format()-Argumentes gemeint ist und für diesen 8 Zifferstellen reserviert werden.
Eine weitere Variation ist die Benutzung von Funktions-internen Variablen / Referenzen.

```
print("Die {subjekt} {praedikat} die {objekt}.".format(subjekt="Katze", praedikat= "frisst", objekt="Maus"))
```

```
>>>
Die Katze frisst die Maus.
```

Praktisch ist diese Anwendung der **format()**-Funktion schon ein Mischding zwischen der funktionallen Formatierung und der im nächsten Abschnitt besprochen Ausgabe mit Platzhaltern.

Aufgaben:

1. *Gegeben ist eine Aufgabe, die in Textform vorliegt "3 + 5 * 4". Mittels einer print()-Anweisung soll der folgenden Text formatiert (mit Platzhaltern) ausgeben werden. An die passenden Stellen sind die Aufgabe und das Ergebnis zu integrieren!*

   Das Ergebnis zur Aufgabe ??? lautet ???.

2. *Erstellen Sie ein Programm, dass ein fixes Bank-Guthaben von 100 Euro für die nächsten drei Jahre mit 0,5425% p.a. verzinst! Der Zins soll jeweils ausgezahlt werden. Die Ausgabe wird im Geld-typischer Format erwartet! Für jedes Jahr soll die Ausgabe separat mit allen relevaten Angaben in einer Zeile erfolgen!*
3. *Verändern Sie das Programm so, dass der Zins auf das Konto gutgeschrieben wird!*

6.1.2.2. Verwendung von Platzhaltern in Ausgabetexten

Eine ältere Ausgabe-Technik arbeitet mit dem sogenannten **%**-Operator. Er wird auch **String-Modulo-Operator** genannt.
Für diese Ausgabe-Formatierung werden innerhalb des Ausgabetextes Platzhalter untergebracht, die dann im hinteren Teil der **print()**-Anweisung durch konkrete Variablen oder Ausdrücke ersetzt werden.

```
from math import sqrt
fkt_name="Wurzel"
arg=2
fkt_wert=sqrt(argument)
print("Die %12s von %6d ist gleich %12.3f."  % (fkt_name,arg,fkt_wert))
```

```
>>>
Die       Wurzel von      2 ist gleich        1.414.
>>>
```

Diese Art der Text-Ausgabe – also die Nutzung des **%**-Operators – sollte aber nicht mehr eingesetzt werden. Irgendwann soll der **%**-Operator aus dem Funktions-Umfang von Python entfernt werden.

Aufgaben:

1. *Gegeben ist eine Aufgabe, die in Textform vorliegt "3 + 5 * 4". Statt der Zahlen sollen natürlich Variablen eingesetzt werden. Mittels einer print()-Anweisung soll der folgenden Text formatiert (mit %-Operator) ausgeben werden. An die passenden Stellen sind die Aufgabe und das Ergebnis zu integrieren!*

   Das Ergebnis zur Aufgabe ??? lautet ???.

2.

6.1.2.3. Kombination von Platzhaltern und format-Funktion

Für die mehrfache Verwendung ein und desselben Textes für Ausgaben, bei der nur bestimmte Werte aktualisiert werden müssen, bietet sich die dritte Variante für formatierte Ausgaben an.
Dazu definiert man sich einen Text, wie dass im nachfolgenden Quelletext mit der Variable text gemacht wurde.

```
text = "Das Ergebnis lautet: {}."
```

An die Stelle mit den beiden geschweiften Klammern (**{ }**) soll später dann ein konkreter Wert ausgegeben werden.
Ein kleines Test-Programm könnte also z.B. so aussehen:

```
text = "Das Ergebnis lautet: {}."
erg = 4 + 31
print(text.format(erg))
neuesErgebnis = erg * erg
print(text.format(neuesErgebnis))
```

```
>>>
Das Ergebnis lautet: 35.
Das Ergebnis lautet: 1225.
>>>
```

Wir verwenden den vordefinierten Text für zwei Ausgaben von zwei unterschiedlichen Berechnungen (sehr informativ ist das im Beispiel natürlich nicht!).
Diese Variante ist auch sehr praktisch, wenn man ein Programm für unterschiedliche Nutzer-Sprachen erstellen will.
Auch Korrekturen / Verbesserungen am Ausgabe-Text lassen sich schnell an einer einzigen Stelle erledigen.
Mittels mehrerer geschweifter Klammern und entsprechen vilen Argumenten in der format()-Funktion können beliebig viele Sachverhalte aussgegeben werden. Der Text lässt sich universell für verschiedene Ausgaben benutzen.

```
text = "Die herausgesuchten Daten sind {}, {} und {}."
erg = "aaa"
print(text.format(erg,erg+"a",erg+"b"))

…
wert = 1
print(text.format(wert-1,wert,wert+1))
```

In die geschweiften Klammern kann man auch noch die (minimale) Anzahl von Zeichen für die Ausgabe festlegen. Diese Anzahl wird in die geschweifte Klammer hinter einem Doppelpunkt notiert.
{:4} ist somit ein Platzhalter für exakt vier Zeichen.

Aufgaben:

1. *Erweitern Sie Ihren "Python-Spicker" um die Möglichkeiten von formatierten Ausgaben!*
2. *Lassen Sie Python auf der Console die nebenstehende Pseudografik erstellen! Dabei dürfen die anzuzeigenden Text-Teile immer nur die gleichen Symbole enthalten sein (siehe oberste Zeile: 3 Texte (verschieden unterlegt)).*

```
    ^
   / ^ \
  /  #  \  \
 /   |   \  \
/    |    \  \
|    |    |  |
|    |    |  |
|    |    |  |
#  #  #  #  #  #  #  #  #
```

3. *In einem Programm sollen 3 Zahlen als x1, x2 und x3 vorgegeben (oder eingegeben werden) – z.B.: 7, 24, 285! Für diese Zahlen soll dann die nachfolgende Tabelle erstellt werden!*

```
  x   |   x * x   | x * x + x
------+-----------+----------
    7 |        49 |        56
   24 |       576 |       600
  285 |     81225 |     81510
```

4. *Passen Sie Ihr Programm von 3. so an, dass eine saubere Trennung zwischen Eingaben (Vorgaben), Verarbeitung (Berechnungen) und Ausgaben eingehalten wird! (also keine Berechnungen in den Ausgaben oder im Ausgabebereich!)*

für die gehobene Anspruchsebene:

5. *In einem Programm sollen 3 Zahlen als x1, x2 und x3 vorgegeben (oder eingegeben werden) – z.B.: 7, 24, 285! Für diese Zahlen soll dann die nachfolgende Tabelle erstellt werden! (PI definieren wir uns mit 3,14159)*

```
  x   | x * 2.5 |  x * x * PI
------+---------+-------------
    7 |    17.5 |      153.938
   24 |    60.0 |     1809.556
  285 |   712.5 |   255175.648
```

6. *Erstellen Sie ein Programm nach dem Muster von Aufgabe 2, bei dem neben der Tür links und rechts noch ein Fenster zu sehen ist! (Das Dach kann auf der gebenen Höhe flach ausgeführt werden → Satteldach.)*

6.2. Eingaben

Eingaben dienen zur Entgegennahme von Nutzer-Interaktionen. Im Normalfall wird ein Programm zuerst einmal anzeigen (ausdrucken), was der Nutzer nun als nächstes eingeben soll bzw. welche Interaktion von ihm erwartet wird.
Obwohl Eingaben ohne jedwede Anzeige auf dem Bildschirm funktionieren, gehört es zum guten Programmierstil die Eingabe mindestens mit einer kurzen Ausgabe zu kombinieren. Bei der Vielzahl von Programmen ist es einfach eine Notwendigkeit mit dem Nutzer sinnvoll zu kommunizieren. Nur ein blinkender Cursor (Prompt) kann alles bedeuten und lässt zu viele Möglichkeiten für eine "Fehlbedienung" des Programms.
In unserem Einführungs-Beispiel (→) tauchte schon der allgegenwärtige **input()**-Befehl auf. Für die Konsole ist er quasi die einzige Möglichkeit direkt mit dem Nutzer zu interagieren.
In graphischen Programmen kommen dann die Maus-Aktionen und die verschiedenen Bedien-Elemente der Benutzer-Oberflächen (Options-Kästchen, Auswahllisten, Schaltflächen, …) dazu.
Praktisch jede Eingabe muss einer Variable zugewiesen werden. Damit ergibt sich folgendes Schema:

```
variable = input()
```

Auf der linken Seite vom Zuweisungs-Operator steht eine Variable, deren ursprünglicher Wert nun durch den Wert aus einer Eingabe überschrieben wird. Python unterscheidet nicht nach den Daten-Typen. Eingaben werden werden praktisch in Roh-Form gespeichert.
Ausnahmen sind Input's, bei denen einfach nur auf eine (beliebige) Eingabe gewartet wird. So etwas haben wir schon am Schluss des Beispiel-Programms verwendet.
Im folgenden Programm-Schnipsel wird zwar für den Programmierer klar, wofür die Eingaben dienen sollen, aber der Nutzer sieht nichts anderes als den Prompt.

```
…
wert_x = input()
wert_y = input()
…
```

```
3
4
```

Als Argument kann und muss man – zumindestens aus kommunikativen Gründen – einen Aufforderungstext mit angeben.

```
wert_x = input("Geben Sie den x-Wert ein: ")
wert_y = input("Geben Sie den y-Wert ein: ")
```

Jetzt wird jedem Nutzer klar(er), was er zu tun hat.

```
Geben Sie den x-Wert ein: 3
Geben Sie den y-Wert ein: 4
```

Eine indirekte Eingabe von Daten ist z.B. über Dateien möglich. Diese werden z.B. zu geeigneten Zeitpunkten eingelesen und ausgewertet.

Aufgaben:

1. *Lassen Sie in einem Programm den Umfang eines beliebigen Vierecks aus den 4 einzugebenen Seiten-Längen berechnen! Verwenden Sie die Bezeichnungen a, b, c und d für die Seiten!*
2. *Für die Berechnung der Fläche eines rechtwinckligen Dreieck's sollen die Seiten mittels sinnvoller Eingabe erfasst werden und das Ergebnis in einem ordentlichen Satz angezeigt werden!*
3. *Berechnen Sie für eine Kreis mit einem einzugebenen Radius den Umfang und die Fläche! Verwenden Sie eine passende Nutzer-Führung!*
4. *Realisieren Sie ein Programm, dass für einen unbedarften Nutzer das Volumen eine zylindrischen Tank's mit Halbkugel-Enden aus den Abmessungen des Tank's berechnet! Versuchen Sie mit möglichst wenigen Eingaben auszukommen!*

für die gehobene Anspruchsebene:

5. *Von einem zylindrischen Tank mit Halbkugel-Enden sind die Höhe bzw. Länge und der Durchmesser bekannt. Der Besitzer (– vielleicht ein einfacher Bauer –) möchte wissen, wieviele Liter der Tank enthält, wenn er:*
   a) flach liegt und halb gefüllt ist
   b) flach voll befüllt ist
   c) senkrecht steht und nur die untere Halbkugel voll ist
   d) außer der oberen Halbkugel voll befüllt ist
   e) der zylindrische Teil zu einem Drittel befüllt ist
   (Die Eingabe und Aussagen sollen für den Besitzer verständlich formuliert sein.)
6.

6.2.1. unschöne Eingabe-Effekte in Python-Programmen

Betrachten wir ein kleines Beispiel-Programm. Es soll eine Eingabe entgegennehmen und mit einem Korrekturfaktor k multiplizieren und dann ausgeben. Mit unseren Programmier-Kenntnissen bekommen wir das schon hin:

```
k=5
x=input("Geben Sie einen Wert für x ein: ")
y=x*k
print("Ihr korrigiertes x ist : ",y)
```

Ein erster Programm-Test zeigt gleich ein Problem: die Multiplikation von 9 und 5 ergibt eigentlich 45 und nicht – wie angezeigt – 99999.

```
Geben Sie einen Wert für x ein: 9
Ihr korrigiertes x ist: 99999
>>>
```

Benutzt man z.B. 7 als Faktor k, dann bekommen wir als Ergebnis z.B.: 9999999. Statt die Eingabe als Zahl zu verwenden, ist die Eingabe scheinbar ein Text, der mit k eben k-mal wiederholt / konkateniert wird (s.a. →).

Aufgaben:

1. ***Testen Sie das Programm mit verschiedenen Ganz- und Fließkomma-Zahlen für k und bei den Eingaben! Dokumentieren Sie k, die Eingaben und die Ergebnisse / Fehlermeldungen (nur Fehler-Typ) in einer Tabelle!***

Nun gibt es grundsätzlich zwei Methoden, um wirklich Zahlen "einzugeben". Bei der ersten Methode nehmen wir den Text und wandeln ihn gezielt in eine Zahl von dem Typ um, den wir brauchen. Dazu benutzen wir z.B. die Funktion **int()**. Diese erwartet als Klammerwert einen Text – also z.B. unsere Eingabe – und liefert eine ganze Zahl zurück.

```
k=5
x=input("Geben Sie einen Wert für x ein: ")
y=int(x)*k
print("Ihr korrigiertes x ist : ",y)
```

Nun stimmt das Ergebnis – zumindestens entsprechend unseren Erwartungen.
Für die Umwandlung in eine Fließkommazahl benutzt man **float()**.

```
Geben Sie einen Wert für x ein: 9
Ihr korrigiertes x ist: 45
>>>
```

Bei dieser Methode gibt es zwei Probleme: Zum Ersten müssen wir schon vorher wissen, welchen Zahlentyp wir benötigen. Zum Anderen können bei fehlerhaften Eingaben Laufzeitfehler eintreten, die das Programm zum Absturz bringen. Als Lösung gibt es das try…except-Konstrukt, welches wir später behandeln (→ 8.14. Behandlung von Laufzeitfehlern – Exception's).
für einen übersichtlichen Code können die Typ-Wandlungs- und die Eingabe-Funktion auch ineinander geschachtelt notiert werden. Jedem Programmierer wird dann sofort klar, dass hier z.B. eine Fließkommazahl eingegeben wird.

```
k=5
x=float(input("Geben Sie einen Wert für x ein: "))
y=x*k
print("Ihr korrigiertes x ist : ",y)
```

Bei der zweiten Umwandlungs-Variante überlassen wir Python die Arbeit. Die Funktion **eval()** übernimmt – zumindesten für Zahlen – die ordnungsgemäße Interpretation von Eingaben.

```
k=1.25
x=eval(input("Geben Sie einen Wert für x ein: "))
y=x*k
print("Ihr korrigiertes x ist : ",y)
```

Der erste Programm-Test mit einem neuen Gleitkomma-k läuft ordnungsgemäß. Die Zahlen werden so verrechnet, wie wir uns das gedacht haben.

```
Geben Sie einen Wert für x ein: 12
Ihr korrigiertes x ist: 15.0
>>>
```

Nun testen wir unser Programm mit einer bewußten Fehl-Eingabe. Ein Buchstabe ist so eine Fehl-Eingabe.

```
Geben Sie einen Wert für x ein: m
Traceback (most recent call last):
  File "D:/XK_INFO/BK_S.I_Info/EingabeSeitenEffekte.py", line 2, in <module>
    x=eval(input("Geben Sie einen Wert für x ein: "))
  File "<string>", line 1, in <module>
NameError: name 'm' is not defined
>>>
```

Die Eingabe z.B. des Buchstaben **m** bringt eine Fehlermeldung. Angezeigt wird die etwas unverständliche Meldung, dass **m** nicht definiert sei,
Was passiert aber, wenn man nun eine Eingabe tätig, die schon eine interne Variable darstellt?

Jetzt akzeptiert Python das **k** scheinbar und rechnet auch irgendwas aus. Beim genauen Hinsehen bemerken wir, dass Python jetzt die vorweg definierte Konstante k (als zu verwendender Faktor) auch in der Eingabe akzeptiert.

```
Geben Sie einen Wert für x ein: k
Ihr korrigiertes x ist : 1.5625
>>>
```

Der Nutzer weiss gar nichts von seinem Glück. Auf seinem Bildschirm ist niemals ein Hinweis auf die Konstante k und deren Wert aufgetaucht.
Was sagt uns das nun für unserer weiters Arbeiten? Man sollte niemals nur einzelne Buchstaben als Variablen-Namen verwenden. Längere – sprechende – Namen sind weniger störanfällig. Sie werden wohl kaum unbewußt oder als einfacher Eingabe-Fehler vom Nutzer verwendet.

```
faktor_k=1.25
eingabe_x=eval(input("Geben Sie einen Wert für x ein: "))
ausgabe_y = eingabe_x * faktor_k
print("Ihr korrigiertes x ist : ",ausgabe_y)
```

Jetzt müsste ein Nutzer schon den Ausdruck **faktor_x** eingeben, damit der Seiten-Effekt auftritt. Wenn er das tut, dann wohl mit voller Absicht und dann können wir auch davon ausgehen, das der Nutzer genau diese Eingabe im Programm haben will.
Auch müssen wir uns immer sehr genau um die Interpretation der Eingaben kümmern. Eingaben sollten immer gleich auf ihre Gültigkeit überprüft werden. In den Tiefen eines verarbeitenden Programms später noch Daten-Typ-Fehler zu finden, ist dann sehr aufwendig.

Aufgaben:

1. *Erstellen Sie ein Programm, dass zuerst zwei (ganze) Zahlen abfragen soll und dann ein einfaches Opperationszeichen (Rechenopperation: + - * /)! Das Programm soll dann (nur!) die vollständige Rechenaufgabe - einschließlich dem Gleichheitszeichen - ausgeben! Speichern Sie sich das Programm gut ab, wir wollen es später noch um die Berechnung des Ergebnisses ergänzen!*
2. *Durch ein Programm sollen drei Paare von Werten - immer jeweils eine ganze und eine reele Zahl - eingegeben werden. Die Zahlen sollen in einer geeigneten Pseudografik-Tabelle angezeigt werden. Unter der Tabelle - mit in die Tabelle eingebunden - sollen die Spalten-Summen und -Durchschnitte berechnet und in jeweils einzelnen Tabellen-Zeilen angezeigt werden! Achten Sie darauf, dass ein unbedarfter Nutzer die Tabelle verstehen kann! Machen Sie sich vorher eine Skizze, wie die Ausgabe aussehen soll!*
3.

6.3. Verarbeitung

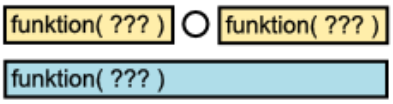
Der Verarbeitungs-Teil eines Programmes enthält alle Operationen, welche die eingegebenen Daten in die Ergebnisse umwandelt. Saclich liegt der Verarbeitungs-Teil nach dem EVA-Prinzip zwischen Eingaben und Ausgaben.
Häufig sind Verarbeitung und Ausgabe stark miteinander verwoben, so dass keine exakte Trennung vorgenommen wird. Im Vorgriff auf spätere Programme und eine Funktionsorientierte Programmierung sollte man sich zwingen, Verarbeitung und Ausgabe bestmöglich voneinander zu trennen.
Die einfachste Form von Daten-Verarbeitungen sind Funktionen. In IDLE erkennen wir sie an der violetten Text-Hervorhebung.
Sachlich lassen sich Funktionen in mehrere Gruppen einteilen. Für die Programmierung ist vor allem wichtig, ob eine Funktion Argumente benötigt oder hat. Nur wenn die Anzahl und Art (Datentyp) der Argumente stimmt, kann das Programm funktionieren. Entweder findet die Syntax-Prüfung des Übersetzers schon Fehler, ansonsten gibt es u.U. einen Laufzeitfehler.
Die zweite wichtige – ebenfalls Syntax-relevante Unterscheidung ergibt sich aus möglichen Rückgabewerten einer Funktion. Manche liefern keine Ergebnisse zurück – sie können separat in einer Befehlszeile stehen. Funktionen mit Rückgabewerten benötigen einen Abnahmer für ihre zurückgelieferten Daten. Das kann entweder eine Variable sein, oder der Rückgabewert wird direkt (als Argument oder Verknüpfungswert) weitergenutzt.

		an Funktion übergebene Daten	
		ohne Argument	**mit Argument(en)**
von Funktion (zurück) gelieferte Daten	**ohne Rückgabewert**	Funktion führt einfache Aufgabe aus	Funktion führt einfache Aufgabe in Abhängigkeit von einem oder mehreren Argumenten aus
		Aufruf: **funktion()**	Aufruf: **funktion(**argument **{** , argument **}** **)**
		Beispiel(e):	Beispiel(e):
	mit Rückgabewert	Funktion führt einfache Aufgabe aus und liefert einen Rückgabewert Rückgabewert muss direkt benutzt oder einer Variable übergeben werden	Funktion führt einfache Aufgabe in Abhängigkeit von einem oder mehreren Argumenten aus und liefert einen Rückgabewert Rückgabewert muss direkt benutzt oder einer Variable übergeben werden
		Aufruf: **variable = funktion**()	Aufruf: **var = funktion(**arg **{** , arg **}** **)**
		Beispiel(e):	Beispiel(e):

{ inhalt **}** … ev. beliebige Wiederholung von inhalt möglich (0 bis x-mal)

Funktionen lassen sich vielfältig kombinieren. Zur Veranschaulichung benutzen wir gerne Rechtecke oder Blöcke. Eine Funktion für sich ist ein Block.

Mehrere Funktionen lassen sich durch Verknüpfungen – das sind eben Addition, Subtraktion, Multiplikation und Division – gefühlt zu einer Funktion vereinen.



In einigen Programmiersprachen kommen weitere Verknüpfungs-Funktionen hinzu. Statt des Verknüpfungs-Symbol (Kreis) kommen +, - , * und / zum Einsatz.

Eine weitere Kombinations-Möglichkeit ist die Schachtelung. Dabei wird eine Funktion anstelle eines Argumentes eingesetzt. Natürlich muss dies eine Funktion sein, die einen passenden Rückgabewert besitzt.

Dagegen ist eine Überschneidung von Funktionen nicht zulässig. Praktisch ist diese auch kaum eingebbar. Sie existiert nur gefühlt für den Programmierer. Der Programm-Übersetzer wird die "Gesamt-"Funktion falsch zusammensetzen und warscheinlich wird der Konstrukt auch fehlerhaft reagieren.

Einfache Berechnungen – also die Verknüpfungen – erfordern immer saubere Anweisungen. Die meisten Programmiersprachen orientieren sich an üblichen mathematischen Ausdrücken. Nur die Zuweisung zu einer Variable zum Speichern des Ergenisses oder die Ersetzung der Berechnung in einer **print()**-Anweisung sind aus mathematischer Sicht nicht ganz logisch.

In der/den nachfolgenden Tabelle(n) sind die wichtigsten Operatoren zusammengestellt.

Operator	Name	Beschreibung	Beispiel	Ergebnis
+	Plus	Addition		
-	Minus	Subtraktion		
*****	Mal	Multiplikation		
@		Matrizen-Multiplikation		
/	Durch	(echte) Division Division ohne Rest		
//	Durch	ganzzahlige Division Division mit Rest		
%		String-Formatierung		
+/-		Vorzeichenwechsel / Vorzeichen		
~		Bit-weise NOT / NICHT		
+=	Inkrement	Addition von 1 zu einer Zahl		
-=	Dekrement	Subtraktion von 1 von einer Zahl		
%	Modulo Modulus	Rest der ganzzahligen Division		
******	Exponent "hoch"	Potenz-Rechnung Potenzierung		
=	ergibt sich aus	Zuweisung		

Ope-rator	Name	Beschreibung	Beispiel	Ergebnis
if .. else		WENN-DANN-SONST Verzweigung		
or		OR / ODER (BOOLEsche Logik)		
and		AND / UND (BOOLEsche Logik)		
not x		NOT / NICHT (BOOLEsche Logik)		
in **not in**		Inklusion bzw. Nicht-Inklusion		
is **is not**		Identität (Übereinstimmung) bzw. Nicht-Identität		
<		kleiner als		
<=		kleiner oder gleich		
>		größer als		
>=		größer oder gleich		
!=		ungleich		
==		Gleichheit / gleich		
|		Bit-weise OR / ODER		
^		Bit-weise XOR / XODER		
&		Bit-weise AND / UND		
<<		Bit-weise Links-Verschiebung		
>>		Bit-weise Rechts-Verschiebung		

Sie lassen sich entsprechend der klassischen Rangordnung kombinieren. Dadurch ergibt sich folgende aufsteigende Rangfolge (Operator-Prezedenz):

lambda → **if**..**else** → **or**, **and** → **not** → **in**, **not in**, **is**, **is not**, **<=**, **<**, **>**, **>=**, **<=**, **==**, **!=** → **|**, **^**, **&** → **<<**, **>>** → **+**, **-** → *****, @, **/**, **//**, **%** → **+/-**x, **~**x → ******, x^y, → **await** x → x**(**..**)**, x**[**..**]** , x**.**attribute → **(**..**)**, **[**..**]**, **{**..**}**

Der Syntax beschreibt die zulässigen Anweisungs-Konstrukte. In der Programmierung haben sich verschiedene Syntax-Darstellungen (Syntax-Diagramme) durchgesetzt. Eine – die EBNF (erweiterte BACKUS-NAUR-Form) lässt sich relativ gut verstehen.
Dabei sind die Zeilen immer Definitionen, die mit einem Begriff und dem Ergibt-Symbol ::= beginnen. Dann folgt die syntaktische Definition. Diese kann weitere zu definierende Begriffe, Anweisungen der Programmiersprache und Steuerzeichen (Metasprachsymbole) enthalten. Anweisungen der Programmiersprache sind sogenannte Terminale. Sie müssen genau so geschrieben werden. in den folgenden EBNF-Zeilen sind die Terminale immer klein geschrieben, so wie die Sprachelemente von Python genutzt werden müssen. Zusätzlich benutze ich noch eine blaue Schriftfarbe.
Die zu definierenden Begriffe nennt man Nicht-Terminale. Für sie muss es in der EBNF-Darstellung mindestens eine Definitions-Zeile geben. In den folgenden EBNF-Zeilen schreibe ich die Nicht-Terminale immer mit einem Großbuchstaben beginnend.
In der EBNF sind noch bestimmte zusätzliche Symbole zugelassen, die Alternativen, Optionen und Wiederholungen kennzeichnen.
Eine erste EBNF-Zeile könnte lauten:

```
Anweisung ::= break | Term
```

Diese Zeile wird folgendermaßen gelesen:

(Das Nicht-Terminal / Der Platzhalter) ***Anweisung*** *ergibt sich aus (dem Terminal / Schlüsselwort)* ***break*** ***oder*** *dem / einem (Nicht-Terminal / Platzhalter)* ***Term****. (****Term*** *ist dabei noch zu definieren!)*

Der senkrechte Strich (**|**) steht also für eine Alternative. Eines der aufgezählten Elemente muss es aber sein.

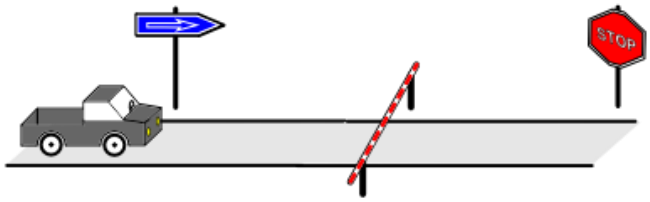
```
Anweisung ::= Variable = Variable
Anweisung ::= Variable = Variable Operator Variable | Term

Operator ::= "+" | "-" | "*" | "/" | "%"
```

Aufgaben:

1.
2. *Jetzt ist es eine gute Gelegenheit den "Python-Spiker" in eine EBNF-Form zu bringen!*
3.

6.4. Kontrolle(n)

In den seltensten Fällen ist ein Programm ein glatter Durchlauf – also eine Sequenz. Häufig müssen Entscheidungen gefällt oder bestimmte Programmteile häufiger wiederholt werden. So etwas wird in Programmen durch sogenannte Kontroll-Strukturen erledigt. Praktisch kennt man in der Programmierung zwei grundsätzliche Arten:



- **Verzweigungen oder Alternativen** — der Programm-Ablauf spaltet sich – ev. nur zeitweise – in mindestens zwei verschiedene Verarbeitungswege auf
- **Wiederholungen oder Schleifen** — bestimmte Abschnitte des Programm-Ablaufs können / müssen mehrfach abgearbeitet werden

Zu den Kontroll-Strukturen gehören wohl auch die **Exception**s. Diese spezielle Art von Verzweigungen für das Abfangen von Laufzeitfehlern besprechen wir weiter hinten (→ 8.15. Behandlung von Laufzeitfehlern – Exception's). Für Anfänger reichen zuerst einmal die anderen Kontroll-Strukturen.
Die Grund-Strukturen sind in den verschiedenen Programmier-Sprachen – ganz unterschiedlich – meist durch sehr spezielle Varianten untersetzt. Wie wir sehen werden, kann man die Strukturen anderer – vielleicht lieb-gewordener Programmier-Sprachen – durch die wenigen Python-eigenen alle ersetzen. Vielleicht bietet die aktuelle Programmier-Sprache ja auch mehr, als man bisher gewohnt war?

6.4.1. Verzweigungen

Wenn zwei alternative Wege in einem Programm zur Verfügung stehen, dann muss eine Entscheidung gefällt werden, welcher der Wege nun genommen werden soll / muss.
Genau nach diesem Prinzip werden Verzweigungen in Programm realisiert. Zur Verdeutlichung schreibe ich den einleitenden Satz dieses Abschnittes noch mal etwas Entscheidungs-betont:

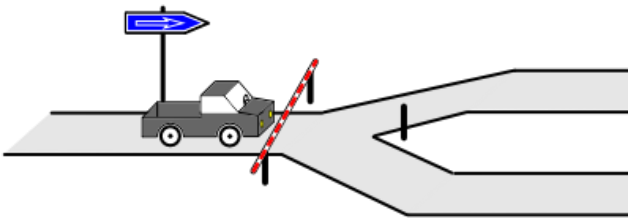
WENN zwei alternative Wege in einem Programm zur Verfügung stehen,
DANN muss eine Entscheidung gefällt werden, welcher der Wege nun genommen werden soll / muss.

Und für die erfahreneren Computer-Nutzer auch ganz ausführlich:

WENN zwei alternative Wege in einem Programm zur Verfügung stehen,
DANN muss eine Entscheidung gefällt werden, welcher der Wege nun genommen werden soll / muss,
SONST wird der andere genommen.

In fast jeder Programmier-Sprache lautet die Programm-Struktur für Alternativen deshalb auch:

IF Bedingung
THEN Alternative1
ELSE Alternative2

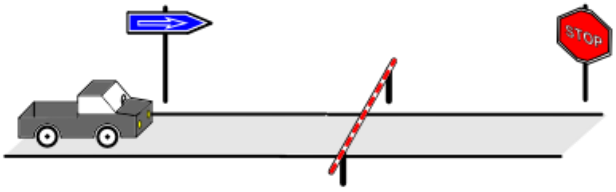


wobei der ELSE-Zweig i.A. optional ist – also bei Bedarf einfach weggelassen werden kann. In dem Fall geht es dann gleich hinter der Alternative1 weiter.
Eine vollständige Verzweigung wird auch zweiseitig genannt, eine ohne ELSE-Zweig heißt einseitige Verzweigung.

6.4.1.1. einfache Verzeigungen

einseitige Auswahl / bedingte Ausführung

Weil die weitere Ausführung des Programms bzw. seiner Teile von der Bedingung abhängt, spricht man bei Verzweigungen auch von bedingter Ausführung.
Wieder andere sprechen von einseitiger Auswahl.
Python vereinfacht für uns die Struktur ein wenig. Da hinter der Bedingung immer der THEN-Zweig kommt, wird auf die gesonderte Schreibung von THEN verzichtet.



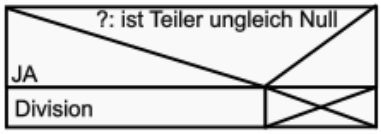
Wollen wir z.B. eine Division programmieren, dann ist das sicher kein allgemeines Problem:

```
…
# Division
zaehler = 4
teiler = 0

print("Division von",zaehler," und",teiler,": ",zaehler/teiler)
```

Bei diesem Beispiel kommt es aber zu einem Laufzeit-Fehler - ev. sind jetzt alle unseren anderen Daten verloren.
Da die Division durch Null nicht definiert ist, müssen wir diese abfangen. Wir führen die Division dazu nur durch, wenn der Teiler ungleich Null ist. In Python ist das zugehörige Symbol für ungleich **!=**.

```
…
# Division
zaehler = 4
teiler = 0

# bedingte Ausführung
if teiler != 0:
    print("Division von",zaehler," und",teiler,": ",zaehler/teiler)
…
```

Ein Struktogramm würde eine solche bedingte Ausführung so darstellen. Dabei steckt in der Symbolik auch schon der Teil, der im alternativen Fall ausgeführt werden soll. Dieser existiert aber bei bedingten / einseitigen Ausführungen / Alternativen nicht.



Der Block besteht also aus mindestens zwei Zeilen. Der Kopf-Teil (obere Zeile) enthält die Frage (Bedingung sowie die Antwort-Möglichkeiten. Die schrägen Linien grenzen die Antwort-Möglichkeiten ab. In der zweiten Zeile folgt der Block mit den Anweisungen für den beschriebenen Fall. Dieser Block kann intern wieder beliebig weiter struktoriert werden. Ein nicht benutzter Teil bzw. Block (, der auch nicht erreichbar ist), wird durchgestrichen / gekreuzt.
Der Nutzer weiss nun allerdings nichts davon, dass die Division gar nicht durchgeführt wurde. Eventuell muss er darüber informiert werden. Dies machen wir natürlich nur, wenn der Teiler gleich Null ist. Das Python-Symbol für die Gleichheit ist **==**.

```
…
# Division
zaehler = 4
teiler = 0

# bedingte Ausführung
if teiler != 0:
    print("Division von",zaehler," und",teiler,": ",zaehler/teiler)
if teiler == 0:
    print("Division (durch Null) nicht möglich!")
…
```

Ganz ähnlich lassen sich alle diskreten Fragestellungen programmieren. Als Beispiel hier mal die Unterscheidung in positive und negative Zahlen:

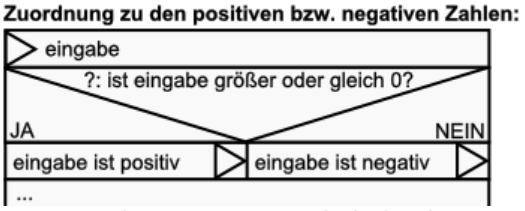
```
…
# Alternative
if eingabe>=0:
    print("Die Zahl ist positiv.")
if eingabe<0:
    print("Die Zahl ist negativ.")
…
```

Aufgaben:

1. *Zeichnen Sie das Struktogramm für die vollständige Bearbeitung der Division mit zwei if-Anweisungen!*
2. *Erstellen Sie ein Programm, das für eine einzugebene (ganze) Zahl prüft, ob es sich um eine gerade oder ungerade Zahl handelt!*
3. *Erstellen Sie ein Programm, das die Teilbarkeit einer einzugebenen Zahl durch die Teiler 2 bis 5 testet! (Es wird Wert auf eine klare Nutzerführung und –Information gelegt!)*
4.

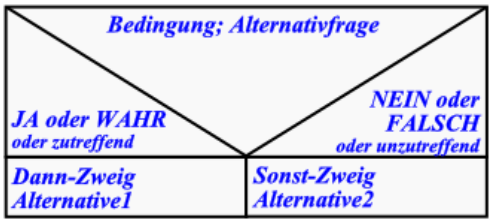
zweiseitige Auswahl / vollständige Verzweigung

Schauen wir uns einige Beispiele an, um das Prinzip und die Notation in Python zu verstehen.
Im ersten Fall wollen wir einfach testen, ob eine eingegebene Zahl positiv oder negativ ist.
Die Bedingung ist also klar, wir müssen nur testen, ob die Eingabe ≥0 ist. In Python werden bei Kombinationen von Vergleichs-Operatoren, wie "<" bzw. ">" und "=" die Zeichen so hintereinander weg geschrieben, wie man es spricht. Ansonsten reichen die einfachen Kleiner- oder Größer-Operatoren. Für Gleichheit muss ein doppelte Gleichheitszeichen verwendet werden damit keine Verwechslung mit dem einfachen Gleichheitszeichen als Zuweisung entstehen kann.



Struktogramm-Ausschnitt für den Test auf eine positive Zahl

Für Ungleichheit verwendet Python die Zeichen-Kombination "!=".
Natürlich testen wir nachfolgend nicht auch noch ab, ob die Eingabe negativ ist. Dieses ergibt sich automatisch. Lediglich wenn die 0 noch extra herausgefiltert werden soll, dann ist ein weiterer Test notwendig. Alternativ kann man auch eine Mehrfachverzweigung (→ 6.4.1.2. Mehrfach-Verzeigungen)benutzen.
Aus dem oberen Struktogramm-Ausschnitt und dem nebenstehend abgebildeten Block können wir die allgemeine Symbolik erkennen.
Im unteren Bereich sind die Blöcke bzw. Blockfolgen für die Alternativen angeordnet. Darüber – über beide hinweg - thront der Entscheidungs-Block. Dieser Block ist durch schräge Linien in drei Bereiche geteilt.



Der obere Bereich, der sich nach unten verengt, enthält die Entscheidungs-Frage oder auch die Bedingung bzw. die Alternativfrage. Diese muss für den Computer immer so gestellt werden, dass eine eindeutige "JA / NEIN"- oder "WAHR / FALSCH"-Entscheidung getroffen werden kann. In vielen Büchern oder Skripten finden sich auch die englisch-sprachigen Entsprechungen "YES / NO" bzw. "TRUE / FALSE".
Die meisten Programmiersprachen testen nur auf das Zutreffen der WAHR-Bedingung, alle anderen Fälle sind dann automatisch FALSCH.
Die Verzweigungs-Struktur beginnt in Python mit **if** gefolgt von der Bedingung. Der Bedingungsteil wird dann durch einen Doppelpunkt abgeschlossen.

```
…
# Alternative
if eingabe>=0:
    print("Die Zahl ist positiv.")
else:
    print("Die Zahl ist negativ.")
…
```

Nach der Eingabe des Doppelpunktes am Ende der Bedingungs-Zeile wird in der nächsten Zeile automatisch eingerückt. Weitere THEN-Anweisungen müssen ev. mit [Tab] eingerückt werden. So können weitere Befehle folgen, die ebenfalls erledigt werden sollen, wenn die Zahl positiv ist.
Gibt es einen ELSE-Zweig, dann wird dieser durch das Schlüsselwort **else** eingeleitet. Auch hier muss ein Doppelpunkt folgen.
Die Befehle des ELSE-Zweiges müssen ebenfalls eingerückt werden.
Kommen dann wieder Befehle, die ungeteilt bearbeitet werden soll, dann wird wieder einmal zurückgerückt. Dieses muss mindestens bis auf die Ebene des IF bzw. des ELSE erfolgen.

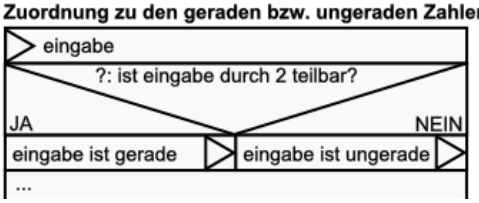
Im nachfolgenden Code-Schnipsel ist die Bedingung anders gestellt. Dadurch tauschen THEN- und ELSE-Zweig.

```
…
# Alternative
if eingabe<0:
    print("Die Zahl ist negativ.")
else:
    print("Die Zahl ist positiv.")
…
```

Welche Variante man nutzt ist mehr Geschmackssache. Üblicherweise beginnt man mit dem Teil (als THEN-Zweig), den man sicher codieren kann.

Betrachten wir noch einen ähnlichen klassischen Fall mit zwei Alternativen. Eine bereitgestellte Zahl (hier: eingabe) soll daraufhin bewertet werden, ob sie gerade oder ungerade ist.

Zuordnung zu den geraden bzw. ungeraden Zahlen:



Eine erste Inspiration bringt uns vielleicht auf die Idee mit dem Test der letzten Ziffer. Handelt es sich um 0, 2, 4, 6 oder 8, dann handelt es sich ja bekanntermaßen um eine gerade Zahl.

So ein Test lässt sich programmieren, aber er ist einfach zu aufwendig. Dazu müsste man die Zahl in eine Zeichenkette wandeln, das letzte Zeichen extrahieren und dann den Ziffern-test durchführen.

Nun könnte man auch verleitet sein, es mal mit der normalen Division zu probieren. Dabei wird man feststellen, dass es x-mal gut geht, aber ab und zu eine fehlerhafte Bewertung auftritt.

Das liegt daran, dass bei einer normalen Division (von Python aus) immer eine Kommazahl (auch Gleitkommazahl genannt) herauskommt. Diese werden vom System i.A. meist in der 7. od. 8. Nachkommastelle gerundet. Und auch, wenn es eigentlich ein endliche rationale Zahl (?,0 bzw. ?,5) werden müsste, entstehen durch die interne Zahlen-Darstellung immer Rundungsfehler.

Diese sind aber kaum vorauszusehen. Weitere Probleme, die praktisch die gleiche Ursache haben, können bei sehr großen Zahle auftauchen, da diese dann in die Exponenten-Schreibung überführt werden. Dieses ist praktisch immer mit Rundungen verbunden.

```
>>>

>>>
```

Aus der Zahlen-Theorie wissen wir, dass wir nur die Teilbarkeit mit 2 prüfen müssen. Besonders einfach geht das mit der ganzzahligen Division. Bleibt ein Rest, dann ist die Zahl ungerade. Geht die Division glatt auf, dann ist die Zahl gerade.

Deshalb bleibt nur die Modulo-Division (ganzzahlige Division). Der Operator ist das Prozent-Zeichen. Als Ergebnis erhält man den Rest der Division. Das testen wir zuerst mal schnell an der Konsole für die Teilbarkeit durch 3 für die Zahlen 4, 5, 6 und 7: Für eine echte Teilbarkeit – wie z.B. bei der 6 – ist der Rest gleich 0. Damit können wir unser Programm prima testen:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
>>>
```

```
…
# Alternative
if eingabe % 2 == 0:
    print("Die Zahl ist gerade.")
else:
    print("Die Zahl ist ungerade.")
…
```

Aufgaben:

1. *Korrigieren Sie das Temperatur-Umrechnungs-Programm um ein Abfangen von Temperaturen, die nicht möglich sind!*
2. *Erstellen Sie ein "super geheimes" Programm, das die Summe und die Differenz zweier einzugebener Zahlen multipliziert! Erweitern Sie das Programm dann um einen Passwortschutz! Nur wer das Passwort richtig eingibt, darf die "super geheime" Berechnung durchführen lassen.*
3. *Erstellen Sie ein Programm, das die Teilbarkeit einer einzugebenen Zahl durch die Teiler 11 bis 20 testet! (Es wird Wert auf eine klare Nutzerführung und –Information gelegt!)*
4. *Erstellen Sie ein Struktogramm und ein Programm, dass für eine einzugebende Zimmer-Temperatur in Grad Celcius ausgibt, in welchem Temperatur-Bereich der Eingabewert liegt!*
   Eine Ausgabe soll anzeigen, ob die Temperatur über 20 °C bzw. gleich/darunter ist! Als zweites soll eine Fein-Differenzierung erfolgen:
   Temperaturen unter 19 °C werden als "zu kalt", über 22 °C als "zu warm" eingestuft. Von 19 bis 20 °C soll "kühl" ausgegeben werden. Für über 20 bis 21 °C gilt die Temperatur als "ok". Im Restbereich ist sie "angenehm".
   Verarbeitung (Bewertung der Eingabe) und Ausgaben sollen abgesetzt hintereinander erfolgen (klassisches EVA-Schema)!
   Nutzer- und Wartungs-Freundlichkeit wird ebenfalls erwartet.
5. *Schreibe eine kleine "Tank-App", die aus dem Tank-Fassungsvermögen, dem aktuellen Tankstand (Eingabe in 10%-Schritten), der Entfernung bis zur nächsten Tankstelle (in km) und dem Durchschnittsverbrauch (in l / 100 km) eine Empfehlung gibt, ob jetzt getankt werden sollte oder ob noch bis zur nächsten Tankstelle genug Benzin im Tank ist!*
   Verbessern Sie die App dahingehend, dass noch 10% Reserve eingeplant sind!
6. *Erstellen Sie ein Programm, dass zu einer erreichten Punktzahl bei einer Arbeit die Bewertung als Note ermittelt! Die mindestens notwendigen Prozentwerte sind: für ein "5" 9%; für eine "4" 36%; für eine "3" 55%; für eine "2" 70% und für eine "1" 85%. Erstellen Sie das Programm mit möglichst wenigen Entscheidungen / Verzweigungen!*
7. *Entwickeln Sie ein Programm, dass für eine einzugebene Zahl ermittelt, ob diese gerade oder ungerade ist!*

für die gehobene Anspruchsebene:

8. *Erstellen Sie ein Programm mit mehreren aufeinanderfolgenden Abschnitten, das für eine einzugebene Zahl die nachfolgenden Bedingungen prüft!*
   a) Zahl ist größer als 20 *b) Zahl ist maximal 23*
   c) Zahl ist kleiner oder größer als 0 *d) Zahl ist kleiner als -273,15*
   e) ist die Wurzel aus der Zahl größer als 10
   (Für jeden Test ist eine vollständige, informative Ausgabe zu realisieren! Beginnen Sie mit der Bedingung, die Ihnen am Leichtesten erscheint!)
9. *Bekommen Sie ein Programm für die Testung auf gerade/ungerade Zahl hin, in dem doch nur ein THEN-Zweig (einer Verzweigung) benutzt wird? (Es gibt zwei grundsätzlich unterschiedliche Lösungen, die praktisch auf dem gleichen Prinzip beruhen. Finden Sie beide?)*

Die Bedingungs-Testung in Python ist sehr einfach gestrickt und lässt dadurch viele Vereinfachungen zu, die aber einen Quelltext u.U. schwieriger lesbar machen. Entweder man nutzt Kommentare oder zwingt sich doch, den vollständigen Code zu notieren.
Als FALSCH (**False**) gilt in Python:

- nummerische NULL-Werte (0; 0L; 0.0; 0.0+0.0j)
- der BOOLEsche Wert: **False** (***Achtung!*** *Schreibung beachten!*)
- leere Zeichenketten
- leere Listen oder Tupel
- leere Dictionary's
- der spezielle (Nichts-)Wert: **None** (***Achtung!*** *Schreibung beachten!*)

Alle anderen Werte werden automatisch als WAHR (True) interpretiert.
Zu Anfang ist das etwas gewöhnungsbedürftig. Aber nach zwei, drei Programmen erscheint das irgendwie urlogisch.

Ausdrücke oder Konstrukte	**Beschreibung**	**(weitere) Beispiel**	**Wahrheits-Wert**
0	per Definition FALSCH (False)		False
0.0	per Definition FALSCH (False)		False
alle anderen Zahlen	somit immer WAHR (True) ist ja schließlich etwas	2 5.2 -3	True True True
10 + 5 - 15	die Berechnung ergibt Null → und die ist per Definition FALSCH		False
21 * 17	ergibt Wert ungleich Null → WAHR		True
3.0 / 0.5	ergibt Wert ungleich Null → WAHR		True
"Text"	enthält Text / etwas → WAHR		True
""	enthält nichts → FALSCH		False
[1, 2, 3, 4]	nicht-leere Liste		True
[ ]	leere Liste		False
[[ ]]	Liste mit einer leeren Liste → also ist es eine nicht-leere Liste		True

In der nachfolgenden Tabelle sind viele logische Operatoren und Ausdrücke zusammengestellt, die allesamt – so oder so ähnlich – als Bedingungen in Verzeigungen dienen können. Gleiches gilt für die später behandelten Schleifen (→ 6.4.2. Schleifen).

Operator	Name	Beschreibung	Beispiel	Wahrheits-Wert
<	kleiner (als)		4 < 6 2 < 1	True False
<=	kleinergleich		5 <= 5 6 <= 5	True False
>=	größergleich		3 >= 2 1 >= 2	True False
>	größer (als)		5 > 4 10 > 50	True False
==	gleich		2+3 == 5 4 == 6 'abc' == 'abc'	True False True
!=	ungleich		12 != 13 2 != 1+1 'abc' != 'abc'	True False False
is	ist / (ist) identisch			True False
not	nicht / (Negation) / NICHT	logisches Nicht; Negation	3 is not 4 not 0	True False True
or	ODER	logisches Oder; Disjunktion	a > 10 or b >= 3	True False
and	UND	logisches Und; Konjugation		True False

Mit logischen Operatoren lassen sich viele mehrstufigen Verzweigungen verkleinern, wenn es wirklich nur wenige Alternativen gibt.
Ein häufig vorkommendes Problem sind Nutzer-Eingaben von Einzel-Buchstaben oder kleinen Texten, die sowohl mit kleinen oder großen Buchstaben geschrieben werden könnten. Nehmen wir als Beispiel die Abfrage eines JA durch die Eingabe entweder von "J" oder "j".

```
…
# Eingabe
eingabe = input("Wollen Sie weiter machen <j,J,n,N>: "))

# Alternative mit mehreren Bedingungen
if eingabe == "J" or eingabe  == "j":
    print("ok! Sie haben es so gewollt.")
else:
    print("Na dann eben nicht!")
…
```

Solche logischen Verknüpfungen lassen sich auch immer umdrehen. Deshalb gilt auch der folgende Programm-Text, der exakt das Gleiche leistet:

```
…
# Eingabe
eingabe = input("Wollen Sie weiter machen <j,J,n,N>: "))

# Alternative mit mehreren Bedingungen
if not (eingabe == "J" or eingabe  == "j"):
    print("Na dann eben nicht!")
else:
    print("ok! Sie haben es so gewollt.")
…
```

Das NOT wandelt das Ergebnis des in Klammern stehenden Ausdrucks in das Gegenteil. Durch das Tauschen von THEN- und ELSE-Zweig kommt wieder das Selbe als Ergebnis heraus.
Es gibt keine wirklich gültigen Regeln, wie ein Programmierer die Bedingungen und die Zweige der Alternative verwendet. Empfohlen wird immer eine typisch intuitive (menschlich logische) Anordnung. Da leere Zweige nicht zugelassen sind, muss also der THEN-Zweig auch mit mindestens einer Anweisung gefüllt werden. Deshalb ist es am Sinnigsten, die Bedingungen auch so zu formulieren, dass zu mindestens der THEN-Zweig benutzt wird.
In ganz seltenen Fällen ist es so – zu mindestens scheint es so – dass der THEN-Zweig nicht gebraucht wird und man unbedingt den ELSE-Zweig programmieren muss. Bei solchen Problemchen kann man sich dadurch helfen, dass man in den nicht benutzten Zweig eine sinnfreie Anweisung schreibt. Python ist zufrieden und wir haben unsere Logik beibehalten können. Diesen Trick kann man auch anwenden, wenn man einen Zweig einer Alternative erst einmal nicht weiter programmieren möchte, aber die Stelle für später schon mal vorsehen möchte. Man sollte solche Stellen dann durch Kommentaren kennzeichnen.
Leider reichen auch nur Kommentare in den Zweigen nicht aus, der Interpreter meckert diese an und erwartet unbedingt eine Anweisung!

```
…
# Eingabe
a=eval(input("Eingabe= "))

# Alternative mit nicht benutztem THEN-Zweig
if a >= 2:
    # nicht benutzter THEN-Zweig
    sinnfrei=1
else:
    print("Zahl erfüllt die Bedingung nicht")
…
```

Die Python-Lösung für **leere Anweisung**en ist das Schlüsselwörtchen **pass**. Damit wird eine Anweisung ausgeführt, die absolut nichts bewirkt, außer vielleicht ein paar Millisekündchen vergehen zu lassen.
Wohl als einzige Programmiersprache lässt Python Ausdrücke, wie die folgenden zu:

```
0 >= anzahl <=100          # zulässige Anzahl von 0 bis 100
not (10 < alter < 67)      # z.B. ermäßigter Eintritt ins Sportstadion (als Kind und Rentner)
a < b == c                 # a muss kleiner als b sein und b gleichgroß wie c
```

Mehrere Verzweigungen können sauber ineinander verschachtelt werden. Dabei dürfen die ELSE-Zweige sich nicht überschneiden und die Einrückungen müssen eingehalten werden.

```
…
# Alternative
if eingabe == 0:
    print("Die Zahl ist Null.")
else:
    if eingabe > 0:
        print("Die Zahl ist positiv.")
    else:
        print("Die Zahl ist negativ.")
…
```

Probieren Sie z.B. mal den folgenden fehlerhaften (!) Code aus:

```
…
# Alternative
if eingabe == 0:
    print("Die Zahl ist Null.")
    if eingabe > 0:
        print("Die Zahl ist positiv.")
else:
    print("Die Zahl ist negativ.")
…
```

!!!: Fehlerhafter Quell-Code!!!

Aufgaben:

1. *Was läuft hier falsch? Analysieren Sie den Quelltext!*
2. *Schreiben Sie die Alternative so um, dass zuerst die negativen Zahlen aussortiert werden!*

6.4.1.2. geschachtelte Alternativen

Eine klassische Einsatz-Variante für Alternativen ist die Unterscheidung von vier Gruppen anhand von zwei Eigenschaften. Im nachfolgenden Beispiel sind das die "Erwachsenen" ab der Altersgrenze 14 Jahre und die Unterscheidung nach dem Geschlecht für eine zu konstruierende Anrede:

```
…
# Eingabe der Personendaten
vorname=input("Geben Sie den Vornamen der Person ein: ")
name=input("Geben Sie den Nachnamen der Person ein: ")
alter=eval(input("Geben Sie das Alter der Person ein:  "))
maennlich=input("Ist die Person männlich <j,J,n,N>: ")
# Definition einer Altersgrenze für die Anrede-Form
altersgrenze=14
# Anrede entscheiden und zusammenstellen
if maennlich == "j" or maennlich == "J":
    if alter >= altersgrenze:
        anrede="Sehr geehrter Herr "+vorname+" "+name
    else:
        anrede="Lieber "+vorname
else:
    if alter >= altersgrenze:
        anrede="Sehr geehrte Frau "+vorname+" "+name
    else:
        anrede="Liebe "+vorname
# Ausgabe
print()
print("Anrede:")
print(anrede)
…
```

Für die Anrede-Konstruktion sind nur das Alter und das Geschlecht zu unterscheiden. Der Name selbst wird dann nur für die Ausgabe gebraucht.
Im Programm-Text wurden die zweiten – inneren / geschachtelten – (Neben-)Verzweigungen dunkler unterlegt. Für ein Testen der ersten (Haupt-)Verzweigung kann man anstelle der Neben-Verzweigung erst einmal eine kleine **print()-**Anweisung setzen.

```
>>>
Geben Sie den Vornamen der Person ein: Monika
Geben Sie den Nachnamen der Person ein: Mustermann
Geben Sie das Alter der Person ein:  29
Ist die Person männlich <j,J,n,N>: n

Anrede:
Sehr geehrte Frau Monika Mustermann

>>>
```

Aufgaben:

1. ***Bei einem Ausverkauf gibt es 20% auf die ausgezeichneten Preise. Weiterhin wird bei einem Umsatz von 100 Euro nochmal 5% Rabatt und bei 200 Euro extra 15 % Rabatt gewährt.***
   Erstellen Sie ein Programm, dass aus der normalen Preissumme den zu zahlenden Betrag ermittelt! Weiterhin soll angezeigt werden, wieviel der Kunde gespart hat und wieviel Mehrwertsteuer im Endpreis enthalten ist. Für alle Waren gilt der normale Steuersatz von 19%.
2. ***Die Anakonda-Bank hat die folgenden Zins-Konditionen:***
   a) bei einem Guthaben werden 1,5% Zinsen p.a. (pro Jahr) dem Guthaben zugeschlagen
   b) Guthaben über 5000 Euro erhalten 2,5 % Zinsen p.a.
   c) bis 1000 Euro Schulden gibt es den Dispokredit mit 5 % Zins p.a.
   d) bei größeren Schulden gilt der übliche Kreditzins von 7,5 % p.a.
   Ein Python-Programm soll für einen einzugebenen letztjährigen Kontostand den aktuellen zurückliefern! (Im Verlaufe des Jahres erfolgten keine Ein- oder Auszahlungen!)
3. ***Dem Programmierer des folgenden Programm's sind diverse Fehler unterlaufen. Finden und korrigieren Sie diese***

```
 1 # Programm zur Interpretation von
 2   Farbkodierungen an Signalleinen
 3   Rettungsdienste / Einsatztauchen
 4   alle 10 m ein Leder-Läppchen
 5   nach je 2m Markierung in:
 6   schwarz, weiss, rot, gelb  #
 7
 8 # Eingabe
 9 leder=eval(input("durchgelaufene Leder-Läppchen: ")
10 letzteFarbe=input("letzte durchgelaufene Farbe: ")
11
12 # Verarbeitung
13 fehler==0
14 if letzteFarbe=="schwarz";
15     laenge=2
16 elif Farbe=="weiss":
17     laenge=4
18 elif letzteFarbe=="rot":
19     laenge=4
20 elif letzteFarbe="grün":
21     leange=8
22 else
23 Fehler=1
24
25 Auswertung
26 if Fehler==1:
27     write("Es ist ein Fehler aufgetreten!
28     if leder > 0:
29         print("mind.",leder*10,"m durchgelaufen")
30 else:
31     gesamt=leder+10*laenge
32     print "es sind","gesamt","m durchgelaufen"
33
```

komplexe und / oder weitere Übungs-Aufgaben zu Alternativen:

1. *Erstellen Sie sich eine dreispaltige Tabelle in Ihrem Hefter! In die erste Spalte kommen die nachfolgenden Ausdrücke! Die zweite Spalte wird mit "Kopf-Computer" und die dritte mit "Python" überschrieben! Überlegen Sie sich dann, welches logisches Ergebnis (True oder False oder kein Wert (weil (syntaktisch) falsch)) bei den einzelnen Ausdrücken herauskommt und tragen Sie das Ergebnis in die Spalte "Kopf-Computer" ein! Überprüfen Sie dann alle Ausdrücke an der Konsole von Python! Die Ergebnisse kommen in die Spalte "Python" der Tabelle. Wie richtig lagen Sie?*

a)	`3 == 3`	b)	`456 <= 289`	c)	`4 == '4'`
d)	`5 == 3+2`	e)	`"Eingabe" == 0`	f)	`"Name" <= "Vorname"`
g)	`8.0 == 8`	h)	`"Hallo!" == "Hallo! "`		
i)	`"Ei" is "Ei"`	j)	`a = 23`	k)	`24 // 8 == 4`
l)	`5 in [2, 3, 5, 7]`	m)	`4 not in [9, 3, 2, 4 ,6, 8, 13, 1, 99]`		
n)	`"Bio" not in ["Astro", Bio, "Chem", "Deu", "Bio, Ma", "Info", SK]`				

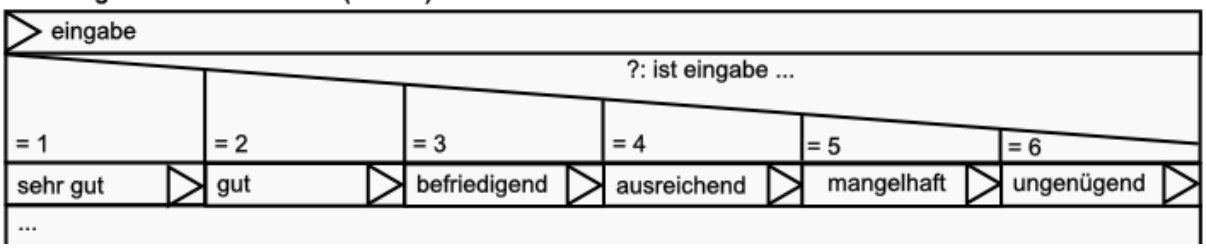
2. *Der pH-Wert zeigt den Charakter einer Lösung an. Dabei sind pH-Werte kleiner als 7 ein Zeichen für saure Lösungen, bei Werten über 7 sprechen wir von basischen Lösungen. Ist der pH genau 7, dann gilt die Lösung als neutral.*
   Erstellen Sie ein Programm, dass aus dem pH-Wert den Charakter der Lösung ermittelt!
3. *Für die Koordinaten eines Punktes (x- und y-Wert) soll ermittelt werden, in welchem Quadranten des kartesischen Koordinatensystems der Punkt einzuzeichnen ist!*
4. *Ein Programm soll für die einzugebende Zimmer-Temperatur in °C ausgeben, ob es zu warm oder zu kalt ist! Als optimaler Wert wurde 21 °C festgelegt.*
5. *Verändern Sie das Programm von 4. so, dass die optimale Temperatur als Variable (Konstante) vorne im Quelltext definiert und auch im weiteren Programm genutzt wird! Speichern Sie das Programm unter einem geänderten Namen ab!*
6. *Verändern Sie das Programm von 4. so, dass der Richtwert durch das Programm abgefragt wird! Der Richtwert darf nicht größer als 25 und nicht kleiner als 15 °C sein!*
7. *Im nachfolgenden Programm sind dem Programmierer diverse Stil-Fehler unterlaufen. Korrigieren Sie diese!*

```
1 a=input()
2 a=int(a)
3 if a>273:
4  if a<373: print("flüssig")
5  else: print("gasförmig")
6 else: print("fest")
7
```

6.4.1.3. Mehrfach-Verzeigungen

Neben den einfachen Verzweigungen kennen viele Programmiersprachen Mehrfach-Verzweigungen. Meist lautet das Schlüsselwort dann SWITCH, CASE oder so ähnlich. Python geht bei den Mehrfach-Verzweigungen einen ganz einfachen Weg – es erweitert einfach die "normale" Verzeigung.
Im nächsten Beispiel sollen eine Schulnote, die als Ziffer eingegeben wird, in die Textform umgesetzt werden.
Das Struktogramm für diese Mehrfach-Verzweigung sieht so aus:

Textausgabe von Schulnoten (Ziffern):

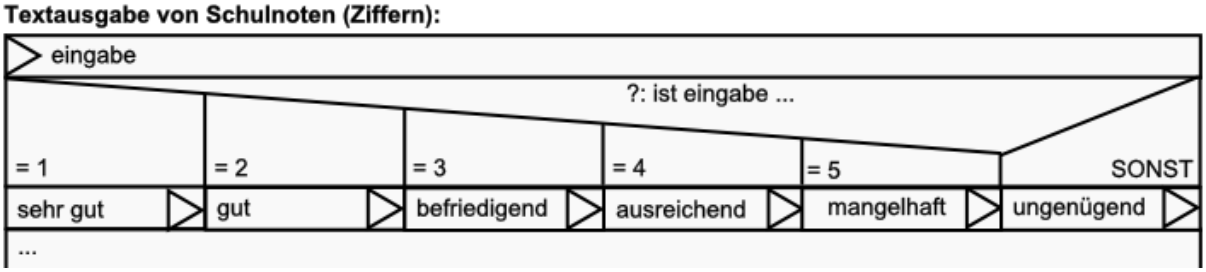


Der Quellcode in Python ist eine erweiterte **if**-Struktur. Vor dem optionalen beendenden **else** können beliebig viele **elif**'s eingefügt werden. Sie stehen für immer jeweils einen Ausgang aus der Mehrfach-Verzweigung.

```
…
# Mehrfach-Alternative
if eingabe == 1:
    print("sehr gut")
elif eingabe == 2:
    print("gut")
elif eingabe == 3:
    print("befriedigend")
elif eingabe == 4:
    print("ausreichend")
elif eingabe == 5:
    print("mangelhaft")
elif eingabe == 6:
    print("ungenügend")
…
```

Im Allgemeinen ist ein Abschluss mit einem **else** besser.
Dann kommt immer etwas bei der Mehrfach-Verzweigung heraus und man kann effektiver nach Fehlern forschen. Das zugehörige Struktogramm würde dann so aussehen:

Textausgabe von Schulnoten (Ziffern):



Und das zugehörige Python-Programm sähe dann so aus:

```
…
# Mehrfach-Alternative
if eingabe == 1:
    print("sehr gut")
elif eingabe == 2:
    print("gut")
elif eingabe == 3:
    print("befriedigend")
elif eingabe == 4:
    print("ausreichend")
elif eingabe == 5:
    print("mangelhaft")
else:
    print("ungenügend")
…
```

```
>>>
Note (als Ziffer): 4
Note in Textform:
ausreichend
>>>
```

Diese Form der Mehrfach-Verzweigung birgt ein großes Risiko. Vergißt man bei komplizierteren Bedingungen z.B. bestimmte Grenzen oder Randbedingungen, dann kann man seinen Programm-Ablauf immer im ELSE-Bereich wiederfinden. Nehmen wir als einfaches Beispiel die Bewertung von Temperaturen (mit Nachkommastellen. Auf den ersten Blick sieht der nochfolgende Quelltext unproblematisch aus, aber der Teufel steckt hier im Detail:

```
…
# Mehrfach-Alternative
if temp < 19.0:
    print("zu kalt")
elif temp > 19 and temp < 20:
    print("kühl")
elif temp > 20 and temp < 22:
    print("angenehm")
else:
    print("zu warm")
…
```

Während die erste Eingabe (hier 19,4) noch ein exaktes Ergebnis liefert, versagt unser Programm bei 19,0 °C.
Das Problem wird deutlich, wenn wir einmal mit 19,0 die Mehrfach-Verzweigung durchgehen:

```
>>>
aktuelle Zimmer-Temperatur [°C]: 19.4
kühl
>>>
aktuelle Zimmer-Temperatur [°C]: 19.0
zu warm
>>>
```

Die Bedingung (<19) in der startenden IF-Anweisung wird mit FALSE beantwortet und somit in der ersten ELIF-Anweisung weiter gemacht. Die Bedingungen treffen einzeln und in der UND-Verknüpfung nicht zu, also wird auch die Auswahl-Mölichkeit übersprungen. Genau geht es der 19,0 in der zweiten ELIF-Anweisung. Was bleibt, ist der ELSE-Zweig. Hier wird die Wertung "zu warm" kreiert. Ähnliches passiert z.B. auch bei der Eingabe von 20,0.
Das Problem sind hier die nicht direkt aneinander anschließenden Bereiche. Wir haben imme kleine Lücken – hier 19 und 20 – die nicht erfasst werden und dann im ELSE-Zweig landen.

Aufgaben:

1. *Berichtigen Sie die Mehrfach-Verzweigung zur Temperatur-Bewertung so, dass keine Lücken mehr auftreten!*
2. *Im einem "anderen" vorgelagerten Programm-Teil wird definiert, wo genau diese Grenzen sein sollen. Der Nutzer kann seine Präferenzen also vorher festlegen. Die Auswertung soll dann die aktuelle Temperatur, die Bereichs-Grenzen und die Bewertung anzeigen! (z.B.:*
   die aktuelle Tempertur 20,8 °C liegt im Bereich von 20,5 bis 22,3 °C und ist somit: angenehm
3. *Trennen Sie sauber Eingabe, Verarbeitung (Bewertung) und Ausgabe! (Innerhalb der Ausgabe (am Ende des Programms) darf keine Verarbeitung mehr erfolgen, sondern wirklich nur noch die Ausgabe der Texte / Daten!*

Ganz mutige und sehr von sich eingenommene Programmierer verzichten auch noch auf den ELSE-Zweig – da kann man ja immer schön mit "Kopieren"-"Einfügen" arbeiten.

```
…
# Mehrfach-Alternative
if temp < 19.0:
    print("zu kalt")
elif temp > 19 and temp < 20:
    print("kühl")
elif temp > 20 and temp < 22:
    print("angenehm")
elif temp > 22:
    print("zu warm")

…
```

Nun versagt unser Programm dann vollends. Im Fall der 19 oder 20 °C wird gar keine Bewertung angezeigt. Solche Fehler mit nur wenigen Test-Daten zu finden, gelingt nur selten. Besser ist der folgende Weg:

```
>>>
aktuelle Zimmer-Temperatur [°C]: 19.4
kühl
>>>
aktuelle Zimmer-Temperatur [°C]: 19.0
>>>
```

Alle Bereiche werden mit IF- bzw. ELIF-Zweigen bearbeitet. Der ELSE-Zweig wird für ein Sammeln der nicht ausgewerteten Daten genutzt – quasi als Fehler-Topf:

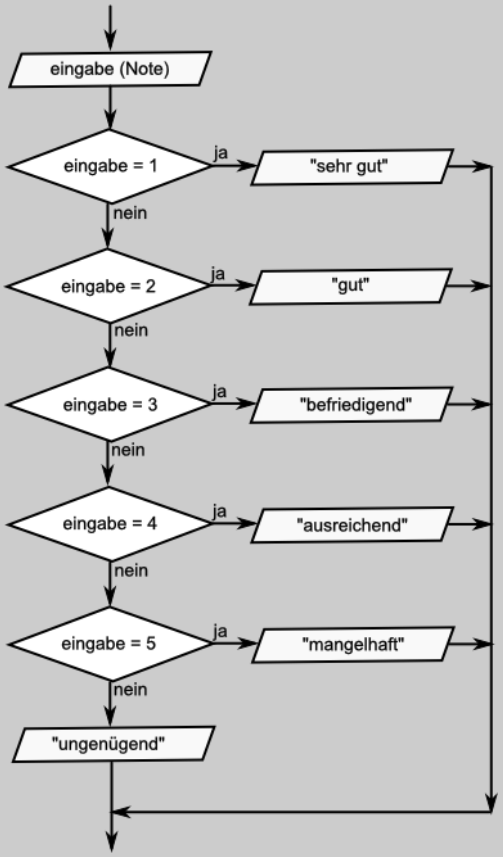
```
…
# Mehrfach-Alternative
if temp < 19.0:
    print("zu kalt")
elif temp > 19 and temp < 20:
    print("kühl")
elif temp > 20 and temp < 22:
    print("angenehm")
elif temp > 22:
    print("angenehm")
else: # nicht ausgewertete Fälle
    print("es ist ein Fehler aufgetreten!")
    print("Melden Sie diesen bitte dem Programmierer!")

…
```

Nun wird bei einem durchrausenden Wert (hier: 20,0) auf einen Fehler hingewiesen. Wenn der Fehler vielleicht auch erst beim Anwender auffällt, die Verarbeitung an sich erzeugt wenigstens keinen Unsinn.

```
>>>
aktuelle Zimmer-Temperatur [°C]: 18.4
zu kalt
>>>
aktuelle Zimmer-Temperatur [°C]: 20.0
es ist ein Fehler aufgetreten!
Melden Sie diesen bitte dem Programmierer!
>>>
```

Exkurs: Mehrfach-Verzeigung – anders dargestellt

Nebenstehend ist ein Algorithmus zur Umsetzung von Noten (in Ziffern) in die zugehörigen Wort-Urteile als Programm-Ablauf-Plan dargestellt. Sachlich entspricht dieses PAP dem letzten Struktogramm.
Insgesamt sieht man bei beiden Darstellungen, dass man bei größeren Algorithmen schnell an die graphischen Grenzen stößt.
Viele (ältere / gestandene) Programmierer schwören auf die guten alten PAP's. In der modernen Programmierung und Algorithmik wird eher auf Struktogramme oder Pseudo-Programm-Text gesetzt.
Ein entscheidender Vorteil der Darstellungen als PAP oder Struktogramm ist auf alle Fälle, dass man quasi mit dem Finger den Ablauf nachvollziehen kann. Das geht bei den PAP's noch besser, als bei den Struktogrammen.
Die Darstellung in Pseudocode spart richtig Platz – ist aber immer schon stark an einer (bestimmten) Programmiersprache angelehnt. Als Pseudosprachen werden dann meist stark vereinfachte Programmiersprachen gewählt, die besonders im akademischen Bereich weit verbreitet sind, wie z.B. Pascal.
Da schon so etwas wie Programmtext vorliegt, ist eine Fehlersuche oder Prüfung des Algorithmus stark von den Programmier-Erfahrungen abhängig und somit nicht unbedingt zielführend.



PAP zur Mehrfachauswahl

Pseudotext einer Mehrfachauswahl:

```
eingabe = INPUT(Notenziffer)
case eingabe of
1: PRINT(sehr gut)
2: PRINT(gut)
3: PRINT(befriedigend)
4: PRINT(ausreichend)
5: PRINT(mangelhaft)
ELSE PRINT(ungenügend)
```

Leider gehört eine CASE-Anweisung nicht zu Python. Wir müssen uns also mit Behelfs-Strukturen begnügen.

Aufgaben:

1. ***Erstellen Sie ein Programm zur eindeutigen Interpretation eines pH-Wertes!***
   (Es gilt: unter 2: sehr sauer; von 2 bis unter 4: sauer; von 4 bis unter 7: schwach sauer; 7 ist neutral; … entsprechend für die basische Seite (es brauchen nur Werte von 0 bis 14 betrachtet werden, bei Werten außerhalb soll ein Hinweis auf "ungewöhnliche Werte" gegegeben werden)***)***
2. ***Wählen Sie eine eigene – mindestens 5-stufige Skala und setzen Sie diese in ein Bewertungs-Programm um!***
3. ***Öffnen Sie sich Ihr gespeichertes Programm von 6.2.1. (Eingabe von zwei Zahlen und eines Operationszeichens) und ergänzen Sie nun die Berechnung und Ausgabe des Ergebnisses der Gleichung!***
4. ***Erstellen Sie ein Programm, dass den BMI für Jungen und Mädchen berechnet und getrennt bewertet!***
5. ***Gesucht ist ein Hilfs-Programm für die Bestandsaufnahme (Biologie, Ökologie), um den Deckungsgrad zu bewerten (z.B. nach Tafelwerk Cornelsen S. 160)!***
6. ***Planen (Struktogramm) und entwickeln Sie ein Programm, dass aus zwei Winkeln und der dazwischen liegenden Seite die restlichen Seiten und den dritten Winkel berechnet. Weiterhin soll das Programm den Umfang und die Fläche ermitteln! Bei der Ausgabe der Daten nach dem EVA-Prinzip (also erst geschlossen am Ende des Programms) sollen auch Hinweise auf besondere Eigenschaften des Dreiecks angezeigt werden (z.B.: rechtwinkliges oder / und gleichseitig usw. usf.)!***
7. ***Erstellen Sie ein Programm zur feineren Interpretation des pOH-Wertes!***
   (Hinweis: Der pOH-Wert ist quasi der Gegenwert zum pH-Wert aus der Sicht der Basen. Er berechnet sich u.a. auch 14 = pH + pOH.)
   über 13: extrem sauer, 13 .. 11 stark sauer; von 11 bis über 9: mäßig sauer; von 9 bis über 7: schwach sauer; 7 ist neutral; von 7 bis über 5 schwach basisch; 5 .. 3 mäßig basisch; von 3 bis über1 stark basisch; unter 1 extrem basisch***)***

komplexere Aufgaben (zu Verzweigungen):

1. *Erweitern Sie das letzte Zensuren-Programm so, dass fehlerhafte Eingaben – also negative Zahlen oder Zahlen größer 6 – mit einem Fehler-Hinweis quittiert werden. Überlegen Sie sich zwei grundsätzlich verschiedene Möglichkeiten!*
2. *Programmieren Sie eine Variante der Noten-Textausgabe, in der nur einfache Verzweigungen (also keine Mehrfach-Verzweigungen) vorkommen! Wieviele if's brauchen Sie?*
3. *Finden Sie die Fehler im nebenstehenden Quelltext! (Die Reihenfolge der Noten (4, 1, 6, …) soll beibehalten werden!)*
4. *Prüfen Sie Ihre Korrekturen in einem einfachen – selbst geschriebenen – Python-Programm!*

```
…
# Mehrfach-Alternative
if eingabe == 4:
    print("ausreichend")
elif eingabe = 1:
    print("sehr gut")
else eingabe == 6:
    print("mangelhaft")
if eingabe <> 3:
    print("ungenügend ")
elif eingabe == 5:
    print("ausreichend ")
else:
    print("ungenügend")
…
```

5. *In einem Programm soll durch Eingabe der Farbe und des Wertes (als Text) einer Spielkarte (des französischen Blattes) die passende Spielkarte aus dem deutschen Skatblatt ermittelt werden!*
6. *Bestimmen Sie für ein einzugebenes Jahr, ob es sich um ein Schaltjahr handelt, oder nicht! Es gelten die folgende Regel:*
   Ein Jahr ist ein Schaltjahr, wenn es ohne Rest durch 400 teilbar ist oder wenn es durch 4, aber nicht durch 100 teilbar ist.
7. *Einer Person soll ein oder mehrere Attribut(e) zugeordnet werden! Dazu gelten die folgenden Rahmen:*
   - *bis 1 Jahr alt: Säugling; Kleinkind bis 4 Jahre; Vorschulkind bis 6; Schulkind bis 12; Jugendlicher bis 18; dann Erwachsener*
   - *volljährig / nicht volljährig*
   - *Rentner ab 67*
8. *Durch ein Programm soll eine Spielkarte aus dem französischen Blatt (Karo, Herz, Pik und Kreuz mit jeweils 7 bis 10, Bube, Dame, König, Ass) über Ja/Nein-Fragen erfragt werden! Planen Sie ein Programm, dass mit möglichst wenig Fragen (für den Nutzer) auskommt! Wieviele if's brauchen Sie?*
9. *Erstellen Sie ein Programm, dass zu einer erreichten Punktzahl bei einer Arbeit die Bewertung als Note ermittelt! Die mindestens notwendigen Prozentwerte sind: für ein "5" 9%; für eine "4" 36%; für eine "3" 55%; für eine "2" 70% und für eine "1" 85%. Erstellen Sie das Programm mit einer IF..ELIF..ELSE-Struktur!*

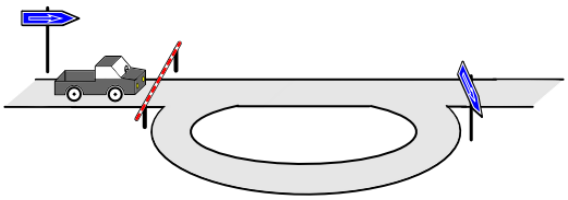
für die gehobene Anspruchsebene:

10. Wie kann man das Programm zu Aufgabe 9 so anlegen, dass es gut wartbar / änderbar für andere Prozentwerte wird?

11. Das Programm von Aufgabe 9 bzw. 10 soll so umgebaut / erweitert werden, dass auch die Punktwertung für die Sekundarstufe II angezeigt wird! Die Fein-Einteilung muss nicht in einer IF..ELIF..ELSE-Struktur erfolgen!

Angaben in % und als Minimum: 1: (96; 90; 85); 2: (80; 75; 70); 3: (65; 60; 55); 4: (50; 45; 36); 5 (27; 18; 9)

6.4.2. Schleifen



In vielen Fällen müssen bestimmte Programm-Abschnitte mehrfach erledigt werden. Die Quelltexte mehrfach hintereinander zu kopieren, wäre eine erste Möglichkeit. Sie empfiehlt sich aber schon deshalb nicht, weil Fehler-Korrekturen extrem aufwändig würden. Häufig weiss der Programmierer auch gar nicht genau, wie oft die Anweisungen wiederholt werden müssen.
Eine Struktur, um Wiederholungen zu realisieren sind die sogenannten Schleifen – oder wie der Schweizer sagt: Schlaufen. Aufgrund bestimmter Bedingungen werden die – in der Schleife liegenden – Anweisungen (Schleifen-Körper) so oft durchlaufen, bis die Arbeit erledigt ist. In der Programmierung unterscheiden wir Schleifen mit vorbestimmten Durchlaufzahlen – die sogenannten Zähl-Schleifen – von den bedingten Schleifen.
Die bedingten Schleifen (Bedingungs-kontrollierten Schleifen) werden nochmals dahingehend unterschieden, wo die Bedingung geprüft wird. Das kann vor dem Durchlauf des Schleifen-Körpers erfolgen. Dann sprechen wir von Kopf-gesteuerten Schleifen. In anderer Literatur werden sie auch vorprüfende Schleifen genannt.
Wird dagegen erst am Ende der Schleife geprüft, dann nennt man die Schleife Fuß-gesteuert oder nachprüfend. Hierbei ist zu beachten, dass der Schleifen-Körper mindestens einmal durchlaufen wird, bevor die Bedingungs-Prüfung am Fuß der Schleife erreicht wird.

6.4.2.1. bedingte Schleifen

Das Konzept der bedingten Ausführung von bestimmten Programm-Teilen (→ 6.4.1. Verzweigungen) kann nun auch auf Schleifen bzw. Wiederholungen angewendet werden. Statt den **if** bei den Verzweigungen verwenden wir nun das Schlüsselwörtchen **while**. Nach dem Durchlauf des eingerückten Programm-Abschnittes kehrt das Programm zur **while**-Stelle zurück und testet erneut, ob ein weiterer Durchlauf notwendig / möglich ist.
Ist die Bedingung nicht erfüllt, dann wird die Schleife nicht durchlaufen. Das Programm setzt dann mit der Bearbeitung der – nach der Schleife – folgenden Anweisungen fort. Das kann natürlich auch schon beim ersten Mal der **while**-Bedingung passieren. Das Struktogramm einer kopfgesteuerten Schleife – so nennt man die while-Schleifen auch – sieht aus, wie ineinander geschachtelte Rechtecke.
Der Schleifen-Körper also der Teil, der innerhalb der Schleife immer wieder abgearbeitet werden soll, kann ein sehr komplexer Blockteil sein.
Da sind Sequenzen, genauso wie Verzweigungen, aber auch neue Schleifen erlaubt. Sie müssen nur sauber ineinander verschachtelt werden. Ein Überlappen ist nicht zulässig!
Für die Entwicklung von Programmen mit mehrfach geschachtelten Schleifen empfiehlt sich zuerst einmal die Top-down-Entwicklungstechnik. Es wird also zuerst die äußertes Schleife programmiert.

Struktogramm: Kopfgesteuerte Schleife

zulässige Schachtelung von Schleifen

In diese können / sollten kleine Ausgaben hinein gebaut werden. Funktioniert die Schleife können die Kontrollausgaben auskommentiert werden und dann die innere Schleife hinzugefügt werden.
Läuft alles, dann können alle Hilfs-Ausgaben gelöscht werden.

unzulässige Schachtelung

Wie sieht eine Schleifen-Struktur in Python aus? Als Beispiel wählen wir hier eine klassische Programmierer-Aufgabe – das Abtesten, ob eine bestimmte Eingabe zulässig ist. Solange das nicht so ist, soll der Nutzer wiederholt zur Eingabe aufgefordert werden.

```
…
# Eingabe mit Gültigkeitstest
eingabe=-1 # Vorgelegung mit falschem Wert,
           # damit man in die Schleife kommt
while eingabe <0 or eingabe > 100:
    eingabe=eval(input("Geben Sie eine Zahl zwischen 0 und 100 ein: "))
…
```

```
>>>
Geben Sie eine Zahl zwischen 0 und 100 ein: 123
Geben Sie eine Zahl zwischen 0 und 100 ein: -5
Geben Sie eine Zahl zwischen 0 und 100 ein: 67
>>>
```

Eine etwas schönere Variante mit einem Fehler-Hinweis könnte z.B. so aussehen:

```
…
# Eingabe mit Gültigkeitstest
eingabe_ok=False       # Vorgelegung mit falschem Wert
while not eingabe_ok:  # ausführlich: eingabe_ok == True
    eingabe=eval(input("Geben Sie eine Zahl zwischen 0 und 100 ein: "))
    if eingabe < 0 or eingabe > 100:
        print("... Bitte den Wertebereich beachten!")
    else:
        eingabe_ok=True
…
```

```
>>>
Geben Sie eine Zahl zwischen 0 und 100 ein: 123
... Bitte den Wertebereich beachten!
Geben Sie eine Zahl zwischen 0 und 100 ein: -5
... Bitte den Wertebereich beachten!
Geben Sie eine Zahl zwischen 0 und 100 ein: 67
>>>
```

Die beiden obigen Programmteile sollte man sich merken und damit alle typischen eingegrenzten Eingaben kontrollieren. Ansosnten droht ev. die Gefahr eines Programm-Absturzes mitten in der Arbeit. Hier sind dann die Ursachen u.U. schwer aus der Fehlermeldung herauszufiltern.

Aufgaben:

1. *Erstellen Sie ein Programm, das solange die Eingabe wiederholt, bis eine Zahl eingegeben wird, die kleiner als 0 oder größer als 100 ist!*
2. *Erweitern Sie das Programm von 1 so, dass die Summe, die Anzahl und der Mittelwert der eingegebenen (richtigen) Zahlen berechnet und angezeigt wird!*
3. *Erstellen Sie ein Programm, dass solange eingegebene Zahlen testet, bis eine 0 eingegeben wird! Dabei sollen die folgenden Tests durchgeführt werden und sachlich korrekte Ausgaben gemacht werden!*
   (Denken Sie sich im Kurs ein Set aus Testzahlen aus, die jeder für seine Programmtests nutzen muss!)
   a) Zahl ist größer als 333
   b) Zahl ist ungerade und durch 3 teilbar
   c) Zahl ist gerade, größer als 28 und durch 7 und 4 teilbar
4. *In einer Schüler-Verwaltungs-Software bekommt jeder Schüler eine fünfstellige ID-Nummer. Diese beginnt niemals mit einer 0. Erstellen Sie ein Programm, dass eine eingegebene Zahl darauf testet, ob sie eine gültige ID ist!*
5. *Mit der Funktion len(zeichenkette) kann man die Länge der Zeichenkette ermitteln. Die Funktion liefert die Anzahl der Zeichen in der Zeichenkette zurück. Erstellen Sie ein einfaches Programm, dass für eine einzugebene Zeichenkette deren Länge ermittelt und ausgibt und anzeigt, ob es sich um eine gültige Eingabe handelt (Zeichenkette besitzt mindestens 3, aber nicht mehr als 25 Zeichen.)*

Aber auch Berechnungen z.B. für Tabellen werden zumeist mit Schleifen aufgebaut. So könnte es z.B. gefordert sein, in einer Tabelle x, x^2 und x^3 für 10 aufeinander folgende Werte zu berechnen und als Tabelle zusammenzustellen.

```
# ========================================
# Programm zur Tabellierung von x-Quadrat
# und x-Kubik
# ----------------------------------
# Autor: Drews
# Version: 0.1 (01.10.2015)
# Freeware
# ========================================
print("Tabellierung von x-Quadrat und x-Kubik")
print("======================================")
print("")
# Eingabe(n)
x_wert=eval(input("Geben Sie den Startwert für x ein: "))
# Ausgabe(n)
print("  x    x²    x³")
# Berechnung / Verarbeitung / Ausgabe
schleifenzaehler=0
while schleifenzaehler < 10:
    print(x_wert, x_wert*x_wert, x_wert*x_wert*x_wert)
    x_wert+=1
    schleifenzaehler+=1
# Warten auf Beenden
input()
```

```
>>>
Tabellierung von x-Quadrat und x-Kubik
======================================

Geben Sie den Startwert für x ein: 5
  x    x²    x³
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
11 121 1331
12 144 1728
13 169 2197
14 196 2744
>>>
```

Die Werte in der "Tabelle" stehen zwar getrennt da, aber so eine Ausgabe entspricht noch nicht wirklich unserem Tabellen-Verständnis. Durch wenige Änderungen und einer Version der **format**-Funktion bekommen wir das aber recht einfach hin:

```
…
# Ausgabe(n)
print("    x    |     x²    |     x³")
print("---------+----------+----------")
# Berechnung / Verarbeitung / Ausgabe
schleifenzaehler=0
while schleifenzaehler < 10:
    print(format(x_wert,"8d"),"|",format(x_wert*x_wert,"8d"),
          "|",format(x_wert*x_wert*x_wert,"8d"))
…
```

```
>>>
Tabellierung von x-Quadrat und x-Kubik
======================================

Geben Sie den Startwert für x ein: 5
    x    |     x²    |     x³
---------+----------+----------
       5 |       25 |      125
       6 |       36 |      216
       7 |       49 |      343
       8 |       64 |      512
       9 |       81 |      729
      10 |      100 |     1000
      11 |      121 |     1331
      12 |      144 |     1728
      13 |      169 |     2197
      14 |      196 |     2744
>>>
```

Was die **format()**-Funktion alles leistet und welche Möglichkeiten zur Formatierung von Ausgaben sie liefert, haben wir uns schon angesehen (→ 6.1. Ausgaben). Hier nur noch mal kurz zur Erinnerung: Die Text-Angabe als 2. Argument in der **format()**-Funktion bewirkt eine Ausgabe eines ganzzahligen Wertes mit insgesamt 8 Ziffernstellen.

Typische Anwendungen für bedingte Schleifen sind Interationen. Da man bei den eigentlich unendlichen Berechnungen irgendwann mal Schluss machen muss und will, braucht man ein passendes Schleifen-Abbruch-Kriterium. Häufig nutzt man die Differenz zum vorlaufenden berechneten Wert. Wenn dieser eine bestimmte Grenze – meist e (Epsilon) genannt – unterschreitet, dann ist man mit der Genauigkeit zu frieden. Genau so verfährt man, wenn sich der

berechnete Wert nicht mehr von seinem Vorgänger unterscheidet. Bei Computersystemen muss man aber beachten, dass eine Gleichheit bei den Werten nicht heißen muss, dass der Wert auch stimmt. Vielfach ist es nur die Genauigkeit des Systems, die uns in Ausführung der weiteren Interationen begrenzt.
Als Beispiel für die Nutzung von Epsoílon als Abbruch-Kriterium nehmen wir die Methode von ARCHIMEDES zur Berechnung von Pi.

Berechnung der Kreiszahl Pi mit der Methode von ACHIMEDES

zugrundeliegende Formel: $x_{n+1} =$

```
import math

epsilon = 1e-20 # Genauigkeit

# Initialisierungen
x = 4
y = 2*math.sqrt(2)
zaehler = 0
# Interationsschleife
while x-y > epsilon:
    x1 = 2*x*y /(x+y)
    y = math.sqrt(x1*y)
    x = x1
    zaehler += 1

print("interierte Kreiszahl  Pi = ",format((x+y)/2,"2.30f"))
print("geforderte Genauigkeit e = ",format(epsilon,"2.30f"))
print("   ---> nach: ",zaehler," Interationen")
print(" zum Vergleich System-Pi = ",format(math.pi,"2.30f"))
```

```
>>>
interierte Kreiszahl  Pi =  3.141592653589792671908753618482
geforderte Genauigkeit e =  0.000000000000000000010000000000
   ---> nach:  26  Interationen
 zum Vergleich System-Pi =  3.141592653589793115997963468544
>>>
```

Bei Schleifen, deren Durchlauf von einer Bedingung abhängig ist, kann es passieren, dass genau die Bedingung immer zutrifft. So etwas passiert z.B. schnell mal bei einer unbedacht angegebenen Bedingung. Man erhält eine Endlosschleife. Diese kann nur durch einen äußeren Eingriff beendet werden.
Trotz alledem gibt es natürlich Aufgaben, die immerzu wiederholt werden sollen. In so einem Fall kann man mit

```
while True:
    SchleifenAnweisungen
```

genauso eine Endlosschleife programmieren. Diese Schleife würde niemals enden, da die Bedingung für die nächste Wiederholung ja immer wahr (=True) ist.
Endlosschleifen kann man durch ein wohlgesetztes **break** in der Schleife beenden. Das **break** würde sich vielleicht aus einer prüfenden Verzeigung ergeben.
Das würde natürlich auch mit eben einer solchen Bedingung im Schleifen-Kopf den gleichen Effekt haben.
Richtig effektiv und auch sinnvoll – wenn auch nicht schön – sind Schleifen, die von mehreren verteilten Bedingungen im Schleifen-Körper abhängig sind. Diese alle in den Schleifen-

Kopf zu platzieren kann unmöglich sein. Dann bieten sich **break**'s an irgendwelchen geeigneten Stellen an.

```
while True:
    SchleifenAnweisungen
    …
    if Bedingung1: break
    …
    if Bedingung2: break
    …
    …
    if Bedingungn: break
    SchleifenAnweisungen
```

Berechnung der Quadratwurzel von x nach der Formel von HERON

zugrundeliegende Formel: $x_{n+1} = \frac{1}{n}\left((n-1)x_n + \frac{a}{x_n^{n-1}}\right)$

```
# Wurzel-Berechnung nach HERON (interativ)
x = eval(input("Aus welcher Zahl soll die Quadratwurzel berechnet werden?: "))
xi = eval(input("Interations-Startwert x0: "))
print()
print("Iteration |   Näherungswert")
print("----------+------------------")
i = 0
xj = xi
while i == 0 or xi != xj: # mind. 1x in Schleife ; Abbruch wenn keine Diff.
    i+=1
    xi = xj
    xj = (xi + x / xi) / 2
    print(format(i,"5d"),'    | ',format(xi,"3.15f"))

print(format(i+1,"5d"),'    | ',format(xj,"3.15f"))
print("fertig")
```

```
>>>
Aus welcher Zahl soll die Quadratwurzel berechnet werden?: 23
Interations-Startwert x0: 5

Iteration | Näherungswert
------------------------------
    1     |  5.000000000000000
    2     |  4.800000000000000
    3     |  4.795833333333333
    4     |  4.795831523313061
    5     |  4.795831523312719
    6     |  4.795831523312719
fertig
>>>
```

Die WHILE-Schleife wird also mindestens 1x betreten, weil ja der Zähler i zu Anfang 0 ist. Später ist dann nur noch die zweite Bedingung entscheidend. Hier wird gepüft, ob der gerade berechnete Nachfolgewert (noch) ungleich dem Vorgängerwert ist. Solange wird weiter interiert.
Aus weiser Voraussicht sollte man aber eine weitere Grenze einziehen. Das könnte die Auswertung der Differenz der beiden Interartionswerte sein. Bei sehr geringem Abstand ist die Berechnung vielleicht schon genau genug für unsere Zwecke. Eine andere Mölichkeit ist es, die Anzahl der Interations-Runden zu beschränken. Wenn z.B. nach 1'000 Interationen noch kein eindeutiges Ergebnis vorliegt, dann wird pro forma abgebrochen, damit der Rechner u.U. nicht ewig rechnet. Die Abbruchzahl ist ein Erfahrungswert und sollte nicht zu niedrig angesetzt werden. Somit änder sich nur die Zeile mit dem WHILE:

```
…
while i == 0 or (i<1000 and xi != xj):
…
```

Berechnung der n-ten Wurzel

Das obige Programm-Muster benutzen wir nun, um die n. Wurzel einer Zahl zu berechnen.

zugrundeliegende Formel: $x_{n+1} = \frac{1}{n}\left((n-1)x_n + \frac{a}{x_n^{n-1}}\right)$

```
# n. Wurzel-Berechnung nach HERON (interativ)
x = eval(input("Aus welcher Zahl soll die n. Wurzel berechnet werden?: "))
n = eval(input("Potenz n der Wurzel: "))
xi = eval(input("Interations-Startwert bzw. Schätzwert x0: "))
print()
print("Iteration | Näherungswert")
print("----------+------------------")
i = 0
xj = xi
while i == 0 or (i<1000 and xi != xj):
    i+=1
    xi = xj
    xj = ((n-1) * xi + x / (xi**(n-1)) ) / n
    print(format(i,"5d"),'    | ',format(xi,"3.15f"))

print(format(i+1,"5d"),'    | ',format(xj,"3.15f"))
print("fertig")
```

Bei anderen Interationen bricht man immer nach einer bestimmten Anzahl von Durchläufen ab. Das wäre dann aber ein klassischer Fall für eine Zählschleife (→ 6.4.2.3. Zähl-Schleifen). Die Berechnung vieler Fraktale basiert auf diesem Prinzip.

Aufgaben:

1. *Verändern Sie das Programm zur Tabellen-Erzeugung für Quadrate und Kubike so, dass statt 10 Zeilen nun 20 Zeilen ausgegeben werden!*
2. *Ändern Sie das Programm zur Tabellen-Erzeugung für Quadrate und Kubike so, dass x nicht in 1er Schritten steigt, sondern immer in 4er Schritten!*
3. *Erstellen Sie ein Programm, dass neben den Doppelten und dem Vierfachen auch die Hälfte in einer Tabelle mit 15 Zeilen zusammenstellt! Der format-Text für Zahlen mit Kommastellen lautet z.B.: "10.3f" für 10 Ziffern-Positionen (insgesamt) mit 3 Nachkommastellen*
4. *In einer 20-zeiligen Tabelle soll ein Programm zu a und seinen Nachfolgern die Wurzel, die Sinus und Tangens-Werte ausgeben! Die Funktion für die Wurzel-Berechnung heißt sqrt(), die für den Sinus sin() und die für den Tangens tan(). Die Funktionen sqrt(), sin() und tan() werden durch die Zeile: from math import * als eine der ersten Zeilen im Programm bereitgestellt (Nutzung eines Moduls).*
5. *Erstellen Sie ein Programm, dass die große Mal-Folge für eine einzugebene Zahl zwischen 1 und 20 – also z.B. für die 2: 11 x 2, 12 x 2, 13 x 2, …, 20 x 2 berechnet und zeilenweise als Gleichungen ausgibt!*
6. *Entwickeln Sie das Nimm-Spiel in Python für zwei menschliche Spieler: Gegeben ist ein Menge Streichhölzer (z.B. 23). Beide Spieler nehmen abwechselnd 1 bis 3 Hölzer weg. Derjenige, der den letzten Streichholz nehmen muss, hat verloren.*
7. *Programmieren Sie das Nimm-Spiel für einen Spieler gegen den Computer! Der Nutzer darf auswählen, wer beginnt. Überlegen Sie sich eine Strategie (für den Computer-Spieler), wie man praktisch ab einer bestimmten Situation nicht mehr verlieren kann!*
8. *Wandeln Sie das letzte Nimm-Spiel so ab, dass sowohl die Maximalzahl entnehmbarer Hölzer als auch die Anfangszahl (mindestens 5 mal größer als die Maximalentnahme) vom menschlichen Spieler gewählt werden kann!*

für die gehobene Anspruchsebene:

9. *Programmieren Sie das Spiel "Groker" für einen Spieler gegen den Computer (der Computer verfolgt die nachfolgende Taktik: (den Quellcode übernehmen Sie so oder mit geänderten Variablennamen in Ihr Programm))*

6.4.2.2. Sammlungs-bedingte Schleifen

ebenfalls Kopf-gesteuert
besondere Form in Python
Sammlungs-bedingte Schleifen beginnen mit dem Schlüsselwörtchen **for**, welches in anderen Sprachen typischerweise für Zählschleifen verwendet wird. In Python muss man hier also umdenken!

wenige andere Programmiersprachen bieten ein ähnliches Konzept

im Schleifen-Kopf wird auf eine Liste (Sammlung) zurückgegriffen, die Elemente-weise abgearbeitet wird
Für unsere Zwecke hier reicht es zu wissen, dass Listen aus Komma-getrennten Elementen in eckige Klammer bestehen. Eine Liste kann einer Variable zugeordnet werden. Weitere Listen-Operationen erklären wir später (→ 9.2. Listen, die I. – einfache Listen, 9.8. Listen, die II. – objektorientierte Listen).

```
# Definition der Listen
faecherliste=["Biologie","Deutsch","Informatik","Mathematik","Sport"]
namensliste=["Arendt","Bauer","Meiser","Lehmann","Meyer","Schulz",
             "Wagner","Zander"]

# zeilenweise Ausgabe der Namensliste
for name in namensliste:
    print(name)
# Warten auf Beenden
input()
```

```
>>>
Arendt
Bauer
Meiser
Lehmann
Meyer
Schulz
Wagner
Zander
>>>
```

Aufgaben:

1. *Ändern Sie das Programm so ab, dass die Fächer zeilenweise ausgegeben werden!*
2. *Lassen Sie das Programm nun Fächer und Namen für sich jeweils zeilenweise ausgeben!*
3. *Überlegen Sie sich, wie Sie die Fächer in einer Zeile hintereinander ausgeben könnten!*

Für Zahlen-Listen kann man auch die runde Klammer **( )** benutzen.

```
# Definition der Liste
zahlenliste= (4,6,7,9,13,102)

# zeilenweise Ausgabe der Zahlenliste mit Quadratzahlen
for zahl in zahlenliste:
    print(zahl," --> ",zahl*zahl)
# Warten auf Beenden
input()
```

Allerdings funktioniert die Nutzung von runden Klammern als Zusammenfassungs-Zeichen auch bei Texten. Genauso sind gemischte Listen möglich.
Offiziell sollen die runden Klammern für sogenannte Tupel – quasi kleine Zusammenfassungen – genutzt werden. Diese sind in vielen Merkmalen den Listen sehr ähnlich.

```
>>>
4  -->  16
6  -->  36
7  -->  49
9  -->  81
4  -->  16
13  -->  169
102  -->  10404
>>>
```

Typische Tupel sind z.B. die Koordinaten eines Punktes im Koordinatensystem. Tupel werden später noch genauer betrachtet (→ 9.1. Tupel).

```
# ========================================
# Programm zur Erstellung einer Schüler-
# Fächer-Tabelle
# ----------------------------------------
# Autor: Drews
# Version: 0.1 (01.10.2015)
# Freeware
# ========================================
# Definition der Listen
faecherliste=["Biologie","Deutsch","Informatik","Mathematik","Sport"]
namensliste=["Arendt","Bauer","Meiser","Lehmann","Meyer","Schulz",
             "Wagner","Zander"]
# Schleife zur Erzeugung eines Tabellen-Kopfes mit mehreren Fächern
# und weiterer Hilfs-Texte
tabellenkopf_zeile="Name        "
zeilen_linie="------------"
leerspalten=""
for fach in faecherliste:
   tabellenkopf_zeile=tabellenkopf_zeile+" | "+format(fach,"12s")
   zeilen_linie=zeilen_linie+"-+-------------"
   leerspalten=leerspalten+" |              "
# Ausgabe des Tabellenkopfes
print(tabellenkopf_zeile)
print(zeilen_linie)
# Erzeugung und Ausgabe des Zeilen-Teils
for name in namensliste:
    print(format(name,"12s")+leerspalten)
# Warten auf Beenden
input()
```

```
>>>
Name         | Biologie     | Deutsch      | Informatik   | Mathematik   | Sport
-------------+--------------+--------------+--------------+--------------+-------------
Arendt       |              |              |              |              |
Bauer        |              |              |              |              |
Meiser       |              |              |              |              |
Lehmann      |              |              |              |              |
Meyer        |              |              |              |              |
Schulz       |              |              |              |              |
Wagner       |              |              |              |              |
Zander       |              |              |              |              |
```

Aufgaben:

1. *Übernehmen Sie den oberen Quelltext in Ihr Python-System!*
2. *Drucken Sie sich den Text einmal aus und nummerieren Sie die Zeilen beginnend bei 1 durch!*
3. *Kommentieren Sie Quelltext zeilenweise aus!*

für die gehobene Anspruchsebene:

4. *Eine Klasse soll in die richtige Zelle der Tabelle eingetragen werden! Dazu liegen die Daten in der Form: eintrag=["Deutsch","Lehmann","10c"] vor.*

für FREAK's:

5. *Es liegt eine Liste von Einträgen für die Lehrer-Fach-Tabelle vor. Alle Einträge sollen richtig eingeordnet werden!*
   eintraege=[["Deutsch","Lehmann","10c"] , ["Biologie","Meyer","7a"] , ...]

Hat man zwei Listen, dann kann man diese auch gemeinsam durchlaufen. Dazu gibt man als Lauf-Variablen für jede Liste eine spezielle Variable Komma-getrennt an und die Listen werden mit der **zip()**-Funktion miteinander verbunden.
Die Durchläufe orientieren sich an der kürzeren Liste.

```
# Definition der Listen
faecherliste=["Biologie","Deutsch","Informatik","Mathematik","Sport"]
lehrerliste=["Arendt","Bauer","Meiser","Lehmann","Meyer"]
# Schleife zur Erzeugung der Lehrer-Fächer-Paare
for fach,lehrer in zip(faecherliste,lehrerliste):
   print(format(lehrer,"12s"),format(fach,"12s"))
```

```
>>>
Arendt       Biologie
Bauer        Deutsch
Meiser       Informatik
Lehmann      Mathematik
Meyer        Sport
```

Aufgaben für die gehobene Anspruchsebene:

1. *In die obige Fächer-Lehrer-Tabelle soll eine passende Klasse an die richtige Stelle eingetragen werden! Die einzutragende Information liegt als kleine Liste ["Deutsch", "Lehmann", "9a"] vor.*
2. *Erweitern Sie das Programm so, dass es mehrere Einträge auswerten und an die richtige Position eintragen kann! Die Informationen liegen als Liste von Listen vor! Z.B.:*
   [["Deutsch", "Bauer", "10c"], ["Deutsch", " Wagner", "11b"], ["Biologie", "Meyer", "7b"]]
3. *Erstellen Sie sich drei Listen für Hauptstädte, Länder und Einwohner (Reihenfolge beachten) für 10 Staaten!*
   a) Geben Sie eine Komma-getrennte Liste der Länder aus!
   b) Zeigen Sie die Liste der Hauptstädte an!
   c) Erstellen Sie eine Tabelle aus Hauptstädten, Ländern und Einwohnern!
   d) Geben Sie für jedes zweite Land die Daten in Form eines Satzes aus!

6.4.2.3. Zähl-Schleifen

Zähler-kontrollierte Schleife

von Programmier-Anfängern besonders gerne benutzte Struktur, da alles sehr gut unter Kontrolle erscheint
später werden dann die WHILE-Schleifen häufiger genutzt, da sie viel mehr Kontrolle und Flexibilität bieten

for *Zählvariable* in range(…):

range(Grenze**)**
erzeugt eine Liste von Elementen von 0 bis Grenze; Grenze selbst ist nicht erhalten

```
# zeilenweise Ausgabe des Schleifenzählers
for i in range(10):
    print(i)
# Warten auf Beenden
input()
```

```
>>>
0
1
2
3
4
5
6
7
8
9
>>>
```

Die Verwendung solcher Schleifen-Variablen, wie i, j, k usw. usf. haben sich unter Programmierern eingebürgert. Solange die Variablen auch nur in der Schleife verwendet werden, ist das auch ok. Braucht man die Werte für andere Zwecke – ev. auch weiter hinter einer Schleife, dann sollte man sprechende Namen benutzen.
Die ungewöhnliche Zählung – beginnend bei 0 – ist in vielen Programmiersprachen üblich. Man gewöhnt sich schnell daran.
Um den echten Schleifen-Durchlauf zu erhalten reicht ein einfaches i+1.

range(Untergrenze,Obergrenze**)**
erzeugt eine Liste von Untergrenze bis Obergrenze; Obergrenze ist ebenfalls nicht mit im Bereich!
die Liste wird dann quasi wie in einer Sammlungs-orientierte Schleife abgearbeitet, die zwische den Grenzen liegenden Werte, werden online erzeugt

Ruft man **range()** mit drei Argumenten auf, dann stehen diese für Untergrenze, Obergrenze und Schrittweite.

range(Untergrenze,Obergrenze,Schrittweite**)**

um Fließkommazahlen in eine Liste zu bekommen, benötigt man eine eigene Funktion; die range()-Funktion liefert hier keine Lösung. Dazu mehr bei der Besprechung von Funktionen (→ 6.5.2. echte Funktionen – Funktionen mit Rückgabewerten).

wird die Laufvariable nicht innerhalb der Schleife gebraucht, dann kann man in Python auch einen Unterstrich (_) quasi als imaginäre Laufvariable benutzen
ist Python-like aber nicht immer schön zu lesen

Aufgaben

1. *Erzeugen Sie eine formatierte Tabelle mit x, dem Quadrat und dem Kubik von x für 20 Zeilen – ausgehend von einem einzugebenen Startwert für x – mittels Zählschleife!*
2. *Erstellen Sie einzelne (kleine) Programme, welche die nachfolgenden Muster in der Anzeige nur mittels Zählschleifen erzeugen!*

a)
```
*
**
***
****
…
```
bis 30 Sterne in der letzten Reihe

b)
```
*

*
**

*
**
***
…
```
bis 10 Sterne in der letzten Reihe

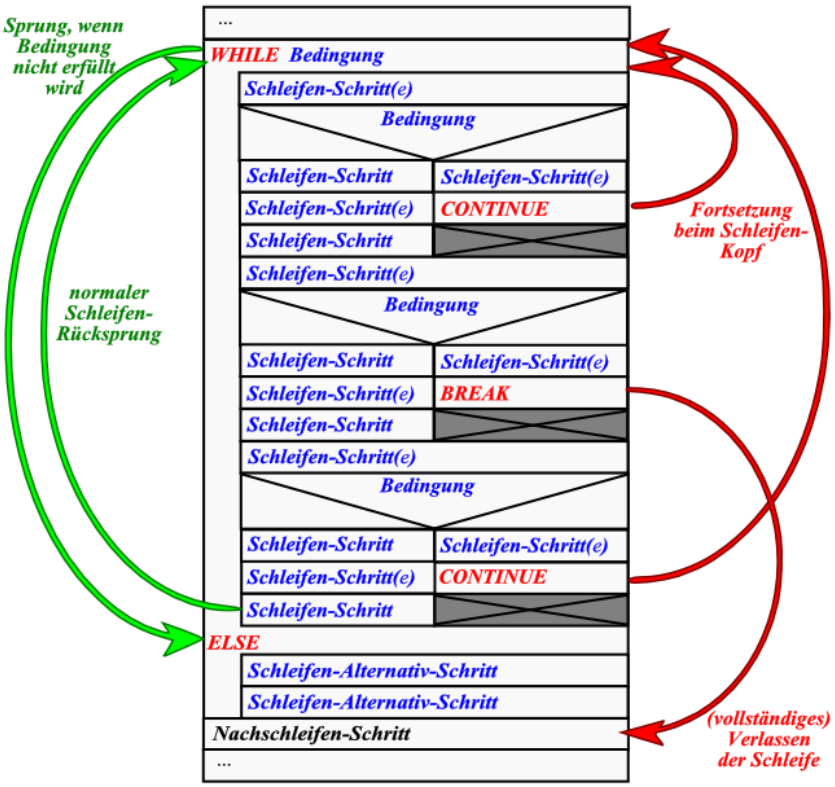
c)
```
*
*#
*#*
*#*#
…
```
bis 20 Zeichen in der letzten Zeile

6.4.2.4. besondere Kontrollstrukturen in Schleifen

Mit einer **else**-Anweisung nach dem Schleifen-Körper
zur Gruppierung einer Anweisungs-Folge, die direkt nach der Schleife abgearbeitet werden soll, wird / kann mit break übersprungen werden
oft gar nicht notwendig, da hinter der Schleife die normalen folgenden Anweisungen folgen
quasi handelt es sich um eine Alternative, die unter bestimmten Bedingungen nach den Schleifendurchlauf abgearbeitet werden sollen

Das Schlüsselwörtchen **continue** zum Abbruch der Anweisungs-Folge im Schleifen-Körper und (Rück-)Sprung zum Schleifen-Kopf, um einen neuen (nächsten, nächstfolgenden) Schleifendurchlauf zu starten

break zum vollständigen Abbruch der Schleife einschließlich des ELSE-Zweiges



Die besprochenen Kontrollstrukturen für Schleifen sollten sparsam und nur mit Bedacht benutzt werden. Sie machen den Quellcode unübersichtlich und schwer verständlich.
Bewährte Schleifen-Konstrukte sollten nur angetastet werden, wenn sie ihre Funktion nicht mehr erfüllen.
Quellcodes werden aber durch sie kompakter und u.U. effektíver

bei Verwendung der besonderen Schleifen-Abbrüche und Verbiegungen sollte man immer gut kommentieren. Viele Programmierer sind saubere Kontrollstrukturen gewohnt und schnell mit außergewöhnlichen Strukturen überfordert (weil ungewohnt).
Außerdem sollte man genau prüfen, mit welchen Werten die in den Schleifen benutzten Variablen nach einer Schleifen-Veränderung herauskommen. Da gibt es schnell böse Überraschungen!

Will man z.B. eine Schleife unendlich oft durchlaufen lassen (z.B. Eingabe-Kontrollen, Tastatur-Abfragen bei laufenden Programmen), dann braucht man meist doch irgendwo einen Notausgang
mit **break** lässt sich das relativ einfach und verständlich realisieren
Nehmen wir eine bedingte Schleifee, die immer wahr ist. Von sich aus wird sie niemals enden.

```
…
while True:
    # auszuführender Code
    …
    # Ende des auszuführenden Codes in der Schleife

# hier kommt die Programm-Abarbeitung niemals!
…
```

D.h. hinter der Schleife kann beliebiger Code oder Unsinn folgen, dieser kann nicht erreicht werden und wird also nie ausgeführt oder interpretiert.
Einen Austieg aus dieser Schleife kann man durch ein break erreichen. Dazu wird im Normalfall irgendwo in der Schleife eine Bedingung (z.B. Tasten-Druck oder das Überschreiten einer Grenze) abgetestet und im Falle des Eintretens mit einem break die Abarbeitung hinter der Schleife fortgesetzt (dann datf da natürlich kein Unsinn mehr stehen!).

```
…
while True:
    # auszuführender Code
    …
    if bedingung:
        break
    …
    # Ende des auszuführenden Codes in der Schleife

# restliches Programm (nach break)
…
```

Man kann natürlich mehrere breaks in die Schleife integrieren. Die Abarbeitung der restlichen Schleife wird immer sofort unterbrochen und hinter der Schleife forgesetzt.
Will man die Schleife ordnungsgemäß am Ende des Schleifen-Körpers verlassen dann kann man den folgenden Code-Rahmen verwenden.

```
…
abbruch=False
while not abbruch:
    # auszuführender Code
    …
    if bedingung:
        abbruch=True
    …
    # Ende des auszuführenden Codes in der Schleife

# restliches Programm
# (nach vollständigem Durchlauf der Schleifenanweisungen)
…
```

Aufgaben:

1. ***Erstellen Sie ein Struktogramm für ein Programm, dass solange einzugebene Zahlen addiert bis ein 0 eingebenen wird!***
2. ***Prüfen Sie das Struktogramm auf Korrektheit, indem Sie die folgenden Zahlen "eingeben"! Legen Sie sich dazu eine Variablen-Tabelle an, die am Ende jedes Schleifendurchlauf die Variablen-Belegung dokumentiert!***
   12, 56, 2, 21, 76, 0, 41
3. ***Realisieren Sie das Programm entsprechend dem Struktogramm!***
4. ***Prüfen Sie mit der obigen Test-Liste und Ihrer Variablen-Tabelle! (Sie können zur Kontrolle am Ende der Schleife auch eine print-Anweisung einbauen! Diese kann dann später auskommentiert werden!)***
5. ***Erstellen Sie ein Programm, dass immer das Quadrat und die Wurzel zu einer eingegeben Zahl ausgibt (Ausgabe in Satzform!)! Die Eingabe soll solange wiederholt werden, bis eine Zahl eingegeben wird, die kleiner als 0 oder größer als 1000 ist!***
6. ***Erstellen Sie ein Programm. dass die Summe der fortlaufenden Zahlen ab einer einzugebenen Zahl berechnet und damit abbricht, wenn das 100fache der eingegebenen Zahl erreicht wird. Wie lautet die letzte aufaddierte Zahl, ohne dass die Grenze überschritten wurde?***

6.4.2.5. Und was ist mit nachprüfenden / Fuß-gesteuerten Schleifen?

Struktogramm: Fuß-gesteuerte Schleife

Für Eingabe-Kontrollen möchte man als Programmierer gerne Fuß-gesteuerte Schleifen nutzen. Sie müssen mindestens einmal durchlaufen werden und dürfen nur verlassen werden, wenn am Ende die Prüfung der Eingabe überstanden wurde.
In Python gibt es keine expliziten Fuß-gesteuerten Schleifen-Konstrukte.
Das wird von vielen Programmiern als Nachteil empfunden. Ändern können wir es aber nicht, also passen wir uns durch kleine Tricksereien einfach an.
Der nachfolgende Quell-Text zeigt eine Möglichkeit, eine nachlaufende Prüfung zu realisieren.

```
…
# pseudo-nachprüfende Schleife
while 1:      # oder: True
    Schleifeninhalt
    # quasi nachlaufende Prüfung
    if Bedingung: break
…
```

Dieses Prinzip kann man beliebig abwandeln. Es bleiben natürlich Kopf-gesteuerte Schleifen, aber gefühlt sind es annehmbare Kompromisse.

Aufgaben:

1. ***Erstellen Sie ein Programm, dass mit einer nachgebildeten Fuß-gesteuerten Schleife die Eingabe testet! Die Eingabe darf nur verlassen werden, wenn ein Großbuchstabe zwischen K und (einschließlich) R eingegeben wurde! Zur Kontrolle soll die Eingabe am Schluss des Programm noch einmal ausgegeben werden.***

diverse Aufgaben zum Thema "Schleifen":

x.

x. Erstellen Sie ein Programm, dass die Anzahl der echten Teiler einer natürlichen Zahl ausgibt!

x. Eine eingegeben Ziffernfolge (liegt im üblichen Format der INPUT-Funktion als Text vor) soll auf die Stellenzahl geprüft werden! Führende Nullen sind vorher zu entfernen!

x. Geben Sie für alle natürlichen Zahlen zwischen 1 und 50 die Anzahl der echten Teiler aus! (OEIS → A000005)

x. Gesucht ist die Zahl - beginnend mit 1 und endend mit 200 - mit den meisten Teilern! Wieviele Teiler sind es?

x.

x. Erstellen Sie ein Programm, dass die monatliche Abzahlung eines Kredites darstellt! Einzugeben sind Darlehens-Betrag (Kredit-Betrag), (monatlichen) Kreditzins und die monatliche Rate. Stellen Sie quasi tabellarisch die Nummer der monatlichen Zahlung, den gezahlten Betrag (Tilgung) und den Rest-Kredit dar!

x. Ein sehr großen Kulturgefäß mit Nährmedium wird am Anfang des Arbeitstages (08:00 Uhr) mit einem Bakterium beimpft. Bakterien teilen sich durchschnittlich alle 20 min. Wieviele Bakterien könnte die Laborantin nach 8 Stunden (16:00 Uhr) im Kulturgefäß vorfinden? Schätzen Sie vorher die Zahl und schreiben Sie diese an die Tafel!

x. Auf dem Bildschirm sollen für eine einzugebene Zahl zwischen 3 und 12 nacheinander die folgenden Muster erzeugt werden! (hier z.B. für 3:)

```
#
##
###
```

```
###
##
#
```

```
  #
 # #
# # #
```

x. Auf dem Bildschirm sollen für eine einzugebene Zahl zwischen 3 und 12 nacheinander die folgenden Muster erzeugt werden! (hier z.B. für 3:)

```
  A
 ---
 A A
-----
A A A
```

```
|\
--
|\|\
----
|\|\|\
```

```
1
2 1
3 2 1
```

x. Die DNA besteht aus 4 Nukleotiden Adenosin (A), Cytosin (C), Guanin (G) und Thymin (T). Für eine Aminosäure eines zu bildenden Eiweißes werden immer 3 Nukleotid (Triplett) benutzt. Lassen Sie ein Programm alle möglichen Triplett-Kombinationen anzeigen und durchzählen!
(Frage nebenbei: Wie viele Aminosäuren könnten damit codiert werden?)
Zusatz:
Zu welchem Ergebnis würde man kommen, wenn statt dem Triplett eine 4er Kombination (Quartett) benutzt würde?

x.

für die gehobene Anspruchsebene:

x. Erstellen Sie ein Programm, dass für einen einzugebenen Zahlen-Bereich (hier z.B.: 8 bis 14) den folgenden Histogramm-ähnlichen Ausdruck erzeugt!
(hinter der Zahl in senkrechten Strichen ein Stern, wenn es sich um eine Primzahl handelt; jede Raute steht für einen Teiler, ein Punkt für Nicht-Teiler; der Stern in Klammern hinter dem Histogramm zeigt an, ob die Teileranzahl selbst eine Primzahl ist)

```
     8 |   |##.#...#
     9 |   |#.#.....#(*)
    10 |   |##..#....#
    11 | * |#.........#(*)
    12 |   |####.#.....#
    13 | * |#..........#(*)
    14 |   |##....#......#

fertig
```

x. Aus der einzugebenen Höhe der Pyramiden (hier z.B.: 5) und einer Buchstabennummer innerhalb des Alphabet's (hier: 20 ; nicht höher als 26 zugelassen!) sollen die folgenden 3 Zeichen-Pyramiden erstellt werden!
(in der letzten Pyramide gilt: für Buchstaben mit ungerader Nummer werden immer die gezählt ungeraden Zeichen ausgegeben (also für C (Buchstabe 3) die 1. und 3. Position), für die geraden Buchstabennummern immer die gezählt geraden Positionen (also für D die 2. und 4.))

```
T
TT
TTT
TTTT
TTTTT

A
B B
C C C
D D D D
E E E E E

A
 B
C C
 D D
E E E

fertig
```

x. Erstellen Sie ein Programm, dass für einen Satz / eine Text-Zeile prüft, ob es sich um ein echtes oder ein einfaches Pangramm handelt! (Echte Pangramme müssen alle Zeichen des Alphabet's genau einmal enthalten. Gemeint sind hier die Buchstaben. Satz-Zeichen werden ignoriert! Einfache Pangramme müssen nur jeden Buchstaben mindestens einmal enthalten.)

Test: "Fix, Schwyz!", quäckt Jürgen blöd vom Paß. → ist echtes u. einf. Pangramm
Prall vom Whisky flog Quax den Jet zu Bruch. → ist einfaches Pangramm

x. Stellen Sie ein Struktogramm oder ein Linien für einen Algorithmus auf, der prüft, ob eine Zeichenkette ein Isogramm ist! Dabei müssen die verwendeten Zeichen immer gleichoft vorkommen!

Test: Otto → Isogramm; ernst → Isogramm; Heizölrückstoßabdämpfung → Isogr.

für absolute Freak's:

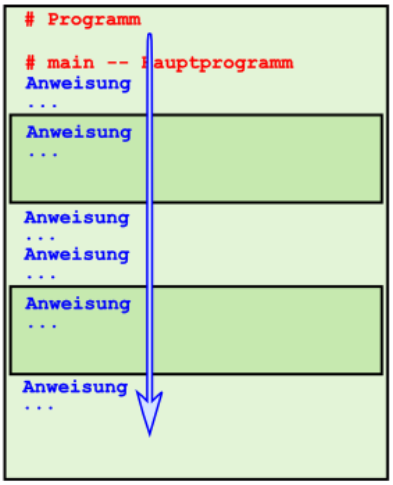
x. Informieren Sie sich, was ein "selbstdokumentierendes Pangramm" ist! Realisieren Sie ein Programm, dass den Sachverhalt an einem String testet!

6.5. Unterprogramme, Funktionen usw. usf.

normaler Programm-Aufbau
Anweisungen bzw. Blöcke praktisch immer in einer mehr oder weniger langen Sequenz
die elementaren Blöcke dürfen dabei ohne weiteres Schleifen oder Verzweigungen sein

letztendlich kommt man immer wieder auf die Grund-Sequenz zurück

manche Sequenz-Abschnitte wiederholen sich. Das ist schon mit einem erhöhten Aufwand verbunden. Entweder die Abschnitte werden noch mal geschrieben oder einfach kopiert.



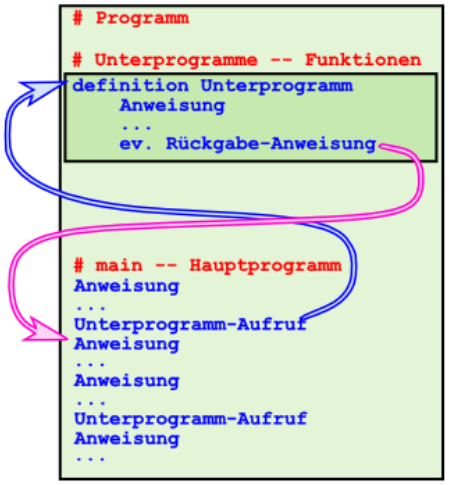
Beim wiederholten Schreiben können – neue / weitere – Fehler auftreten. Beim Kopieren kann man ev. vergessen, das Variablen wieder neu gestartet werden müssen oder – weil sie noch woanders benutzt werden – sie einer Umbenennung bedürfen.
Problematisch ist es auch, wenn sie die Abschnitte als Fehler-behaftet herausstellen. Dann muss man ev. sehr komplexe Änderungen an mehreren Stellen im Programm vornehmen. Das geht meist schief. Irgendwas wird vergessen oder Fehler-behaftet verändert.
Das Finden eines Fehlers gestaltet sich aber relativ einfach, da man die betreffende Stelle gut identifizieren kann.

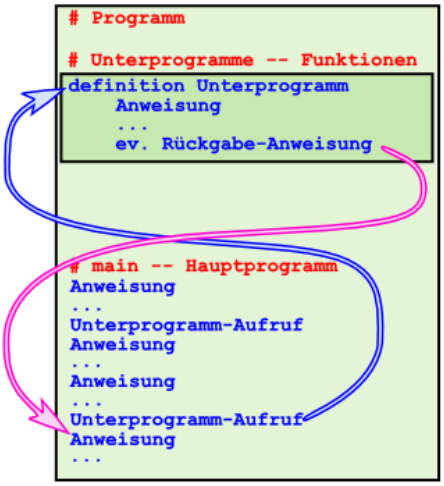
Günstiger wäre / ist es, jeden Programm-Teil nur einmal zu schreiben. Braucht man dann den speziellen Sequenz-Teil, ruft man ihn auf und kehrt danach wieder zur Haupt-Sequenz zurück.
Solche Teil-Sequenzen werden Unter-Programme, Prozeduren und / oder Funktionen genannt. Einige Programmiersprachen unterschieden noch etwas genauer zwischen den verschiedenen Arten. Das ist für uns in Python nicht relevant. Hier gibt es nur Funktionen.
Funktionen (Neben-Sequenzen) werden vor dem eigentlichen Haupt-Programm (meist Main genannt) festgelegt.

Beim Abarbeiten des Programms wird dieser Teil zuerst einmal übersprungen.
Das Interpretieren beginnt bei der Haupt-Sequenz.Trifft der Interpreter auf einen Unterprogramm-Aufruf, so sucht er dessen Definition und führt die enthaltenen Anweisungen aus.

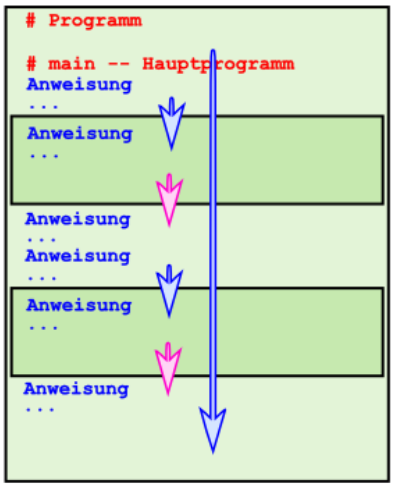
Danach wird mit der nächsten Anweisung im Haupt-Programm fortgesetzt..

Praktisch hat der Interpreter nur einen Umweg gemacht. Sprünge in einem Programm brauchen sehr wenig Rechenzeit.
Auf dem Rücksprung (in Abb. **purpur**) können aus dem Unter-Programm noch sogenannte Rückgabe-Werte mitgegeben werden. Das kann man sich als Ergebnis der Funktion (des Unter-Programm's) vorstellen.
Beim nächsten Unterprogramm-Aufruf passiert das Gleiche wieder. Das Unter-Programm wird angesprungen und abgearbeitet. Dannach wird wieder hinter dem Unter-Programm-Aufruf fortgesetzt.





Denkt man sich alle Teile so, wie sie abgearbeitet wurden direkt hintereinander, dann ergibt sich die gleiche Sequenz, wie bei einer Unter-Programm-freien Sequenz (Abb. rechts).
Vergleichbare Programme mit oder ohne Unter-Programme (mit gleicher Leistung) sind also äquivalent.



6.5.1. Allgemeines zu Funktionen in Python

Funktionen in Python bestehen immer aus einem Namen und einem direkt folgendem rundem Klammer-Paar (**()**). Der Name muss eindeutig sein und folgt den Regeln der Benunnung von Variablen (→). Namen von eingebauten Funktionen dürfen nicht verwendet werden. Eingebaute Funktionen können nicht ersetzt oder wie Programmierer gerne sagen überschrieben werden.
Jede Funktion muss vor dem ersten Benutzen definiert werden. Dazu benutzt man das Schlüsselwort **def**. Die Definition kann im selben Quell-Text erfolgen oder als Import durch eine Bibliothek (→).
Der Anweisungs-Teil wird hinter einem Doppelpunkt (**:**) eingerückt notiert. Die Funktion ist beendet, wenn die Einrückung wieder aufgehoben wird.
I.A. wird empfohlen Funktionen relativ klein zu gestalten. Eine Seite Quell-Text sollte nicht überschritten werden. Komplexe Funktionen dürfen natürlich auch länger sein. Meist kann man aber wieder kleinere Funktionen auslagern.

6.5.2. Funktionen ohne Rückgabewerte

in anderen Programmiersprachen Prozeduren oder Unterprogramme genannt.
Im Wesentlichen geht es um das Einsparen des Eintippens von Quelltexten. Will man 10x an verschiedenen Stellen in einem Programm genau das Gleiche machen, dann müsste man den Quelltext dafür eben 10x an die passende Stelle schreiben oder bestenfalls kopieren. Ist im Code ein Fehler, muss man alle 10 Stellen wiederfinden und den Quelltext einzeln korrigieren. Da sind Fehler vorprogrammiert.
In den meisten Programmiersprachen werden häufig gebrauchte Programm-Abschnitte an einer bestimmte Stelle ausgelagert. Meist ist dies sehr weit vorne im Quell-Text oder in separaten Dateien (Units, Module, Bibliotheken …). Die Funktionen oder Verweise auf andere müssen vor ihrem Aufruf notiert sein. Der Interpreter muss ja schließlich wissen, was er machen soll. Die Funktionen brauchen unbedingt Namen, unter denen man dann später die ausgelagerten Programm-Abschnitte aufrufen kann.
Im Allgemeinen wird man in den Rückgabe-freien Funktionen irgendwo eine Ausgabe programmieren müssen. Damit werden Funktionen aber nur noch eingeschränkt nutzbar. Nicht immer ist auch eine sofortige Ausgabe gewünscht. Viel besser ist es, den berechneten Wert an das aufrufende Programm zurückzugeben (→ 6.5.3. echte Funktionen – Funktionen mit Rückgabewerten). Soll sich das doch um die Ausgabe kümmern.

entsprechen den Prozeduren oder Unter-Programmen anderer Programmiersprachen
Programmteile, die an mehreren Stellen im Programm gebraucht werden, sind in einem extra Abschnitt definiert

Hier steht die Ersetzungs-Funktion im Vordergrund. Der Funktions-Aufruf ist ein Bezeichner für einen Programm-Abschnitt (Unter-Sequenz), die mehrfach in einem Programm gebraucht wird oder eine komplexere Aufgabe erfüllt. Solche komplexeren Aufgaben hat man vielleicht schon mal in einem anderen Programm zusammengestellt und getestet. Nun kopiert man sie einfach in das neue Programm – entweder direkt (wenn nur 1x gebraucht) oder als Unter-Programm für den mehrfachen Gebrauch.

bei Fehlern, Änderungen, Anpassungen usw. ist nur die Korrektur an einer Stelle notwendig

Es gibt auch Argument-freie Funktionen. Für ihre Ausführung sind keine weiteren Informationen aus dem aufrufenden Programm notwendig. In Python kennzeichnet man solche Funktionen durch ein leeres Klammer-Paar.

```
>>> def trennzeile():
      print("------------------")


>>> for i in range(5):
      print(i)
      trennzeile()


0
------------------
1
------------------
2
------------------
3
------------------
4
------------------
>>>
```

die Einhaltung der Anzahl Argumente ist für die Programmierung wichtig, da hier der Übersetzer (Compiler bzw. Interpreter) sofort auf Übereinstimmung prüft
Nichtübereinstimmung – auch bei der Art der übergebenen Daten (Datentypen) wird sofort als Syntax-Fehler gekennzeichnet.

Aufgaben:

1. *Erstellen Sie ein Tabellen-Programm für die Berechnung des großen En-Mal-Eins! Der Nutzer soll eine Anfangszahl (1. Faktor) und einen Multiplikator (2. Faktor zwischen 11 und 20) angeben. Die Tabelle soll nach jedem Wert einen Zwischen-Linie enthalten, die über eine passende Funktion erzeugt wird!*
2. *Ein Programm soll 5 Zahlen multiplizieren! Die Zahlen sollen immer nacheinander eingegeben werden und grundsätzlich zwischen 0 und 100 liegen. Eine nachfolgende Eingabe soll immer mindestens so groß sein, wie die letzte Eingabe! Die Ausgabe eines oder mehrerer Fehler-Texte soll über eine oder mehrere Funktionen erfolgen!*
3. *Erstellen Sie ein Programm, dass nur aus drei Funktionen besteht! Diese sollen Kopf(), Koerper() und Fuss() heißen! Kopf() und Fuss() enthalten nur allegmeine Text-Ausgaben, wie den Programm-Titel und eine kurze Programm-Beschreibung bzw. eine Ende-Hinweis. Das eigentliche Programm soll in Koerper() stecken und dort den Durchnitt aus einzugebenen Noten berechnen! Eingabe-Ende ist eine Null. Bei Noten-Eingaben größer 6 bzw. 8 (für die Gesamtschule) soll ein Fehler-Text erscheinen!*
4. *Erstellen Sie ein Programm, dass die Punkt-Wertungen der Oberstufe verarbeiten kann! Legen Sie den Abbruch-Wert für die Eingabe selbstständig fest!*
5.

6.5.3. echte Funktionen – Funktionen mit Rückgabewerten

klassische Interpretation des Begriff nehmen wir sin x

die Sinus-Funktion benutzt das Argument x (Funktionsargument, x-Wert) zur Berechnung des resultierenden Funktionswertes (y-Wert, abhängige Größe). Dieser kann dann anstelle des Funktions-Ausdrucks eingesetzt werden.

```
>>>
3.0
3.5
>>>
```

In Python – und den meisten Programmiersprachen ist es notwendig, das oder die Argumente in Klammern hinter dem Funktionsnamen aufzuzählen.

klassische Form der Funktion – sie liefert (mindestens) einen Funktionswert zurück

```
…
# Quadrat-Funktion
def quadrat(argument):
    return argument**2

…
```

```
…
# Quadrat-Funktion
def quadrat(argument):
    quadratzahl = argument * argument
return quadratzahl

…
```

Im Abschnitt zu den Zählschleifen (→ 6.4.2.3. Zähl-Schleifen) habe ich darauf hingewiesen, dass es leider nicht möglich ist, sich mit der Funktion **range()** eine Liste mit Gleitkommazahlen zu erstellen. Hier definieren wir uns nun eine Hilfsfunktion **floatrange()**, die genau das kann:

```
…
# range-Funktion für Gleitkommazahlen
def floatrange(start, ende, schrittweite=1.0):
    floatliste=[]
    neuer_wert=float(start)
    while neuer_wert < ende:
        floatliste.append(neuer_wert) # anhängen des letzten Wertes,
                                      # der durch die Bedingung kommt
        # nächsten (ev. möglichen) neuen Wert erstellen
        neuer_wert=neuer_wert+schrittweite
    return floatliste

…
```

```
…
# Typ-unabhängige Additions-Funktion
def summe(argument1, argument2):
    return argument1 + argument2

…
```

die mystery-Funktion:

```
def mystery(x):
    f = [0,4,0,3,2]
    while x > 0:
        x = f[x]
        # print(x,end=' ')
    # print()
    return "fertig"
```

```
>>>
```

Mit welchem Parameter-Wert endet diese Funktion nie? Es ist die 3 – probieren Sie es aus!

Aufgaben:

1. ***Wie könnte man die mystery-Funktion so absichern, dass sie auch bei Parameter-Werten über 4 noch ordnungsgemäß startet? Welche Werte führen dann zu unendliche Schleifen-Arbeit?***
2. ***Erstellen Sie ein Rahmen-Programm, dass die mystery-Funktion für einen einzugebenen Werte-Bereich prüft!***

6.5.4. Funktionen mit Funktionen als Parameter

```
def ausgabe(x):
    print("x = ",x)

def tue(fkt):
    fkt(17)

# Main
tue(ausgabe)
```

```
>>>
x = 17
>>>
```

z.B. bei Maus-Eingaben gebraucht (→ 8.8.10.2. Maus-Eingaben)

6.5.4. Generator-Funktionen – Funktionswerte schrittweise

Manchmal braucht man keine Liste von Werten eines Bereiches (→ range()-Funktion), sondern die Werte sollen immer Schritt-weise zurückgeliefert werden – quasi immer bei jedem Aufruf der nächste gültige Wert. Dazu gibt es Python die Möglichkeit sogenannte Generator-Funktionen zu definieren. Die dazu benötigten Schlüsselwörter von Python hießen **yield** und **next**.

```
…
# range-Generator-Funktion für Gleitkommazahlen
def generatorfloatrange(start, ende, schrittweite=1.0):
    neuer_wert=float(start)
    while neuer_wert < ende:
        yield neuer_wert # zurückliefern des Wertes (quasi: return)
        # nächsten (ev. möglichen) neuen Wert erstellen
        neuer_wert=neuer_wert+schrittweite
    # hier kein return!!!
…
```

Das Benutzen der Generatorfunktion erfolgt in zwei Abschnitten. Zuerst muss er Generator zugeordnet werden. Dazu wird eine Laufvariable mit der Funktion gleichgesetzt. Das entspricht im Prinzip einer Bekanntmachung. Erst wenn jetzt mit **next()** ein Wert abgerufen wird, erzeugt die Generator-Funktion den ersten Funktionswert. Bei jedem weiteren **next()**-Aufruf bekommt man den nächstfolgenden Wert zurückgegeben.

```
…
aktwert=generatorfloatrange(3.0,4.5,0.5)
print(next(aktwert))
print(next(aktwert))
```

```
>>>
3.0
3.5
>>>
```

Ein Problem tritt auf, wenn man einen Wert "zuviel" abruft. Hier kommt es zu einem Laufzeitfehler, der aber abfragbar ist (***StopInteration*** → 8.13. Behandlung von Laufzeitfehlern – Exception's).

```
…
aktwert=generatorfloatrange(3.0,4.5,0.5)
print(next(aktwert))
print(next(aktwert))
print(next(aktwert))
print(next(aktwert))
```

```
>>>
3.0
3.5
4.0
Traceback (most recent call last):
  File "floatrange-funktion.py", line 29, in <module>
    print(next(aktwert))
StopIteration
```

Das muss das aufrufende Programm realisieren. Wird die Generator-Funktion in einer **for**-Schleife verwendet, dann kommt es zu einem regulären Schleifenabbruch (ohne Laufzeitfehler). Für for-Schleifen braucht man aber ganzzahlige Werte.
Um Gleitkommazahlen in einer Schleife zu verwenden, muss man auf while zurückgreifen und dann aber auch das Abbruchkriterium selbst definieren.

```
…

start=3.0
ende=5.5
schritt=0.5
bereichswert=generatorfloatrange(start, ende, schritt)
wert=next(bereichswert)
while wert < ende-schritt*2:
    print(wert)
    wert=next(bereichswert)
```

```
>>>
3.0
3.5
4.0
>>>
```

Aufgaben:

1. *Programmieren und testen Sie eine Generator-Funktion* `zaehlen(bis)` *für das Hochzählen von 0 bis zum Bis-Wert!*
2. *Schreiben und testen Sie eine Generator-Funktion* `countdown(start)` *für das Runterzählen bis 0!*
3.

6.5.5. Interator-Funktionen – Funktionswerte noch wieder anders

Die Rückgabewerte einer Funktion müssen aber nicht immer berechnet werden. Vielfach soll der Wert aus einer Liste (Menge) kommen, deren Werte immer der Reihe nach genutzt werden sollen.
Das folgende Beispiel einer Wochentags-Funktion liefert mit jedem Aufruf den nächsten Wochentags-Namen in abgekürzter Form.
Dazu definieren wir zuerst eine passende Liste und weisen diese dann mit der Standard-Funktion **iter()** einer Laufvariablen (einem Interator) zu.

```
>>> wo_tage=["Mo","Di","Mi","Do","Fr","Sa","So"]
>>> akt_tag=iter(wo_tage)
>>>
```

Die eigentliche Werte-Erzeugung erfolgt mit **next()**. Dabei wird bei jedem Aufruf immer der nächst-folgende Wert zurückgeliefert.

```
>>> next(akt_tag)
'Mo'
>>> next(akt_tag)
'Di'
>>> next(akt_tag)
'Mi'
>>> next(akt_tag)
'Do'
>>> next(akt_tag)
'Fr'
>>> next(akt_tag)
'Sa'
>>> next(akt_tag)
'So'
>>>
```

Das geht solange gut, wie Werte in der Liste vorhanden sind. Beim Versuch nach dem letzten Element noch ein abzurufen, erhalten wir einen StopInteration-Fehler.
Nun müssen bzw. können wir den Interator neu initialisieren und schon kann es wieder von vorne losgehen.

```
>>> next(akt_tag)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    next(akt_tag)
StopIteration
>>> akt_tag=iter(wo_tage)
>>> next(akt_tag)
'Mo'
```

Eine Liste kann von mehreren Interatoren benutzt werden. Jeder Interator zählt eigenständig für sich weiter.

Aufgaben:

1. ***Gegeben ist eine Liste von Farben für ein Etikett. Schreiben Sie ein Programm, dass immer nach Eingabe einer ungeraden Zahl die Farbe wechselt. (Die Eingaben sollen unendlich oft möglich sein. Der Laufzeit-Abbruch am Ende der Liste soll zuerst einmal der Programm-Ausstieg darstellen.)***
   (gelb, rot, blau, weiß, grün)
2. ***Erstellen Sie nun ein Programm, dass Etiketten erstellt, deren Farben sich immer wieder wiederholen! (Als Abbruch soll die Eingabe einer Null dienen.)***
3. ***Nun brauchen wir ein Programm, dass Etiketten mit wechselnder Farbe (siehe Aufgabe 1.) und einer immer wieder neuen Beschriftung erzeugt.***
   (ABC, DEF, GHI, JKL, MNO, PQR, STU, VWX, YZ_)
4. ***Überlegen Sie sich, wieviele verschiedene Etiketten möglich sind! Schreiben Sie ein Test-Programm, dass mindestens 20 mehr Aufrufe der Funktion erzeugt und anzeigt!***

für die gehobene Anspruchebene:

5. ***Gesucht sind sogenannte befreundte Zahlen (auch: amicable numbers,)! Dabei ergibt die Summe der echten Teiler der einen Zahl jeweils die andere. Wie geht die Reihe weiter?***
   (erste Glieder der Reihe: (220,284), (1184,1210), (2620,2924), …)

6.6. Vektoren, Felder und Tabellen

Tabellen sind super Strukturen, um Daten geordnet zu speichern. In Programmiersprachen nennt man die Tabelle üblicherweise Felder (Array's). Es gibt eindimensionale Felder, die Vektoren heißen und mehrdimensionale Felder ohne spezielle Namen.
In Felder werden Daten des gleich Datentyps gespeichert. Der Zugriff erfolgt i.A. über die Zeilen- oder Spalten-Nummern. Man nennt diese Indices. Bei Feldern ist im Vergleich zu Listen keine spätere Erweiterung möglich. Die Größe bleibt so, wie sie einmal definiert wurde.

vektor = [1,2,3,4,5,6]

vektor →	**1**	**2**	**3**	**4**	**5**	**6**
Eintrag-Nr. Index	0	1	2	3	4	5

range für automatische Füllung / Erzeugung einer Liste / eines Vektors

Elementanzahl über Funktion **len()** abrufbar

Zugriff auf Einzel-Elemente über vektor[Elementnummer]

Zugriff auf Bereiche über **:**
: alleine steht für alle Elemente

where um Indizes anhand einer Bedingung auszuwählen

vektor1 →	**1**	**2**	**3**	**4**	**5**	**6**
Eintrag-Nr. Index	0	1	2	3	4	5

vektor2 →	**1**	**4**	**9**	**16**	**25**	**36**
Eintrag-Nr. Index	0	1	2	3	4	5

feld = array([1,2,3,4,5,6],[1,4,9,16,25,36])

Belegung Zeilen-weise mit gleicher Elementanzahl
wenn ungleichviele Elemente in der Dimension, dann bieten sich mehrdimensionale Listen an (also kein **array**-Schlüselwort!)

feld →	0	**1**	**2**	**3**	**4**	**5**	**6**
	1	**1**	**4**	**9**	**16**	**25**	**36**
Eintrag-Nr. Index		0	1	2	3	4	5

Zugriff auf Einzel-Elemente über **feld[**ElementnummerX,ElementnummerY**]** oder **feld[ElementnummerX] [ElementnummerY]**

aus dem **numpy-Modul** kommen:

arange(start, ende, schrittweite)
automatisches Füllen eines Feldes mit Integer- oder Float-Werten

frange(start, ende, schrittweite)
automatisches Füllen eines Feldes mit Float-Werten

linspace(start, ende, schritte)
automatisches Füllen eines Feldes mit Float-Werten

Besonderheit bei numpy: der Teilbereichs-Operator (Slicing-Operator) erzeugt nur eine Sicht (ein view) auf das Original-Array
(anders bei Listen, wo ein neues Listen-Objekt erzeugt wird!)
ändert man die Sicht-Elemente, so ändert man auch die Original-Daten und umgekehrt

feld[start : ende : schrittweite]

für große Array's braucht man dann aber unbedingt die Bibliothek numpy, um effektiv zu arbeiten

```
import numpy as npy
feld2dm0= npy.zeros((zeilen,spalten))

feld2dm1 = npy.ones((zeilen, spalten,ebene))
```

```
feld = { }
zeilen = 4
spalten = 6
for spa in range(zeilen):
        for zei in range(spalten):
                feld[zei,spa] = 0
```

oder als (zweidimensionale) Listen-Konstruktion:

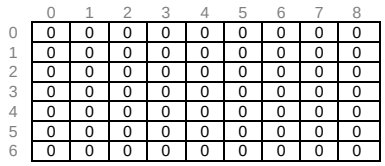
```
feld = [ ]
zeilen = 4
spalten = 6
for spa in range(spalten):
        feld.append(range(zeilen))
        for zei in range(zeilen):
                feld[zei][spa] = 0
```

Kontroll-Ausdruck:
```
print(len(feld), feld)
```

Listen-Konstruktionen von Feldern haben den Vorteil, dass sie variabel sind, also auch erweitert werden können; klassische Felder (Array's) eben nicht
Nachteil ist der relativ hohe Ressourcen-Bedarf

Veranschaulichung einiger Array-Operationen

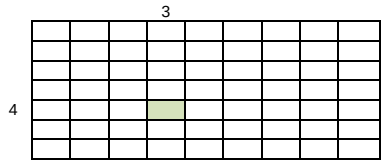
```
import numpy as npy

feld = npy.zeros((7,9))
```



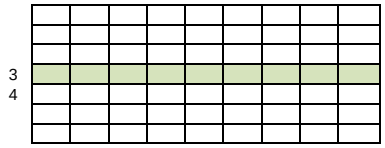

```
wert = feld[4][3]
```

oder

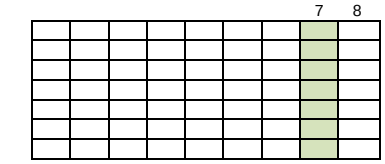
```
wert = feld[4,3]
```



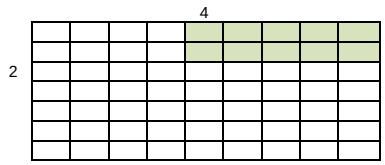

```
sicht = feld[ 3:4, :]
```



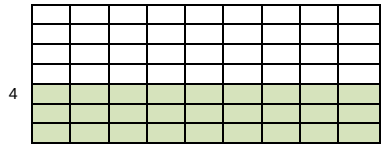

```
sicht = feld[ :, 7:8]
```



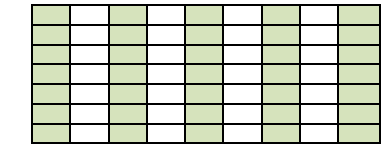

```
sicht = feld[ :2, 4:]
```



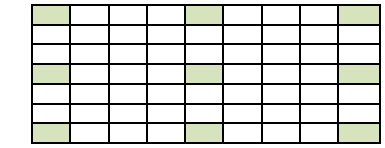

```
sicht = feld[ 4:, :]
```




```
sicht = feld[ ::, ::2]
```



```
sicht = feld[ ::3, ::4]
```



Aufgaben:

1. *Erstellen Sie ein Programm, dass nach Abfrage des Stichproben-Umfanges die Einzelwerte der Stichprobe erfasst und statistisch auswertet! (Das Programm wird auf maximal 100 Werte beschränkt.) Es sollen die folgenden statistischen Maße berechnet werden!:*
   a) arithmetrisches Mittel (arithmetrischer Mittelwert)
   b) Standardabweichung
   c) Minimum
   d) Maximum
   e) maximale Abweichung vom Mittelwert
   f) prozentuale Abweichung vom Mittelwert
   g) Streuung
2. *Erweitern Sie das Programm von Aufgabe 1 um eine Umsortierung in eine aufsteigend geordnete Reihe! Ermitteln Sie nun auch den Median und die Quantillen!*
3. *Das Programm von 1. oder 2. soll noch um eine Häufigkeits-Analyse erweitert werden! Der Nutzer soll dazu die Anzahl der Gruppen vorgeben können (maximal 10) und die untere Grenze soll nach dem Vorschlag vom Programm (Minimum) entweder diesen Wert benutzen oder einen anderen einzugebenen Wert benutzen. Ähnlich ist mit der Spannweite der Gruppen zu verfahren! Die Ausgabe soll aus einer Tabelle (Gruppen-Nr., untere grenze, obere Grenze, Anzahl Werte, prozentualer Anteil) erfolgen!*
4. *Realisieren Sie die Ver- und Entschlüsselung nach dem Four-Square-Verfahren (nach DELASTELLE)!* (→ 8.19.1.x. Four-Square-Verschlüsselung)

6.6.1. Felder mit unterschiedlichen Datentypen

```
# -*- coding: utf-8 -*-

# mehrdimensionale Arrays mit Python 2.2, die für
# ihre Elemente wechselnde Variablentypen zulassen

class varioarray(dict):
    def __init__(self, maxtupel, dummy = ' '):
        dict.__init__(self)
        self.max = maxtupel
        self.dimension = len(self.max)
        self.dummy = dummy

    def test(self, index):
        if len(index) <> self.dimension:
            print 'dimension error'
            return 1 == 2
        for i in range(0, self.dimension):
            if (index[i] > self.max[i]) or (index[i] < 0):
                print 'overflow error'
                return 1 == 2
        return 1 == 1

    def __getitem__(self, index):
        if self.has_key(index):
            return dict.__getitem__(self, index)
        else:
            if self.test(index): return self.dummy

    def __setitem__(self, index, wert):
        if self.test(index):
            dict.__setitem__(self, index, wert)

# Beispiel:
m = varioarray([5, 6, 2], 'None')
"""
das Setzen des Dummys ist nicht zwingend
im constructor ist ' ' voreingestellt
wie bei üblichen Feldinitialisierungen
könnte er auch den Wert Null bekommen
"""
m[1, 2, 1] = '$$$$$' # hier als Zeichenkette
m[1, 6, 1] = -12345  # hier als Integer

print 'Definierte Groesse: ', m.max
print "Dimensionen:", m.dimension
print "Die wirklichen Einträge: ", m
print
s = ''
print 'Beispielzeile [1, x, 1], x von 0 bis 6:'
for i in range(0, 7):
    print m[1, i, 1],
print s

wait = raw_input('enter')
       # kleine Bremse für das Kommandozeilenfenster :-)
```

Q: http://www.way2python.de/

6.7. ein bisschen Statistik

6.7.1. Zufallszahlen

Kommen die Sechsen bei einem Würfel eigentlich immer genauso häufig, wie die anderen Zahlen. Also wenn ich Mensch-ärgere-nicht spiele, dann immer nicht. Glaube ich zu mindestens.
Es ist mal eine schöne Aufgabe die Gültigkeit des Gesetzes von den großen Zahlen (in der Statistik) mit einem echten Würfel-Experiment zu überprüfen. Mit Schüler-Gruppen habe ich das mal machen lassen – und so "überraschend" es für alle war, das Gesetz stimmt. Je häufiger man würfelt, umso genauer tritt die erwartete Häufigkeit von einem Sechstel für die Sechs und natürlich auch für jede andere Zahl auf.
Zum Testen, ob die Schüler auch wirklich würfeln, hatte ich einigen Teams einen besonderen Würfel untergejubelt. Der hatte eine kleine Veränderung. Der Punkt von der Eins war angebohrt und dort eine kleine Madenschraube platziert.
Schnell waren die Gruppen erkannt, die geschummelt hatten!

Aufgabe:

1. ***Stellen Sie eine Hypothese auf, was sich durch die Manipulation verändert!***
2. ***Welches Ergebnis erwarten Sie für ein unmanipuliertes Würfeln von 200 Würfen? Begründen Sie Ihre Vermutung!***
3. ***Würfeln Sie mit einem echten (nicht manipulierten) Würfel 200 mal und erfassen Sie die Würfe in einer Zähltabelle!***
   Sie können die Würfe auch in das Programm "Statistik-String.py" eintragen!
4. ***Fassen Sie die Ergebnisse aller Kursteilnehmer zusammen! Sind die Ergebnisse des Experimentes nun dichter am Erwartungswert? Berechnen Sie dazu die prozentuale Abweichung jedes 200er Experimentes und der Zusammenfassung!***

Nun wollen wir mit Python würfeln. Damit wir eine Zufalls-Funktion zur Verfügung gestellt bekommen müssen wir eine zusätzliche Zeile an den Anfang des Programms schreiben. Dadurch wird ein Modul geladen. Genaueres dazu finden Sie bei → 8.4. Module.
Zum ersten Testen reicht auch die Konsole:

Wenn Sie das nebenstehende Ausprobieren, werden Sie ev. ein anderes Ergebnis bekommen.

```
>>> import random
>>> random.randint(1,6)
4
```

Die Zufalls-Funktion randint() liefert eine Zufallszahl zwischen 1 und 6. Beachten Sie, dass hier die obere Grenze mit eingeschlossen ist!
Mittels einer Schleife lassen wir uns mal 20 "Würfe" anzeigen:

```
import random
for zaehler in range(1,20+1): # +1, weil range oberer Grenze ausschließt
    print(random.randint(1,6), end=' ')
```

```
>>>
6 6 3 6 2 5 1 4 2 5 6 2 1 4 2 1 5 4 1 2
>>>
```

Nun interessiert uns natürlich, ob von allen möglichen Zahlen auch gleichviel gewürfelt werden. Ich verwende zum Merken ein Feld – hie konkret einen eindimensionales – also einen Vektor. Wir brauchen für jede mögliche Augenzahl ein Merkplatz.
Eigentlich würde man jetzt ein Vektor mit der Länge 6 definieren. Der Nachteil ist, dass bei jedem Speichern die Merkposition ausgerechnet werden muss, weil die Felder immer mit dem Index 0 beginnen und wir müssten dann die Würfe mit einer Eins unter dem Index 0 und die Würfe mit einer Zwei unter Index 1 usw. usf. speichern. Das verwirrt schnell und ist eine Fehlerquelle.
Güstiger ist die Speicherung der Würfe mit einer Vier z.B. auch unter Index 4. Der Null-Index kann ja für andere Zwecke benutzt werden, z.B. zum Zählen der Würfe insgesamt. Dann hat man alles schön zusammen gespeichert.
Also definieren wir das Feld mit sieben Positionen so:

```
haeufigkeit=([0,0,0,0,0,0,0])
```

Dabei werden die Werte der einzelnen (sieben) Zellen auf 0 als Startwert gesetzt. Das Feld mit dem Index 0 – also haeufigkeit[0] wird zum Zählen der Würfe genutzt.

```
import random
haeufigkeit=([0,0,0,0,0,0,0])
anzahlwuerfe=1000
while haeufigkeit[0]<anzahlwuerfe:
    haeufigkeit[random.randint(1,6)]+=1
    haeufigkeit[0]+=1
for zaehler in range(0,6+1):
    print(haeufigkeit[zaehler],end=' ')
```

Wird ein bestimmter Wert gewürfelt, so wird der zugehörende Feld-Eintrag über genau diesen Wert als Index gefunden und um eins erhöht (Operator: +=).

Zum Schluss wird das Feld noch schnell ausgedruck.

```
>>>
1000 174 180 160 155 153 178
>>>
```

Wer es bei der Ausgabe auch Feld-orientiert haben möchte kann die letzte Schleife entfernen und die Print()-Anweisung so notieren:

```
…
print(haeufigket)
```

… und wir erhalten tatsächlich ein Vektor.

```
>>>
[1000, 162, 169, 164, 177, 163, 165]
>>>
```

Unter Zuhilfenahme eines zweiten Feldes erfassen wir den Erwartungswert für die Häufigkeit jedes Wurfes und die Abweichung bei jedem einzelnen Wert:

```
import random
haeufigkeit=([0,0,0,0,0,0,0])
anzahlwuerfe=1000
while haeufigkeit[0]<anzahlwuerfe:
    haeufigkeit[random.randint(1,6)]+=1
    haeufigkeit[0]+=1
for zaehler in range(0,6+1):
    print(format(haeufigkeit[wert],"6d"),end=' ')
print()
erwartung=([anzahlwuerfe/6,0,0,0,0,0,0])
print(format(erwartung[0],"5.2f"),end=' ')
for wert in range(1,6+1):
    erwartung[wert]=haeufigkeit[wert]-erwartung[0]
    print(format(erwartung[wert],"6.2f"),end=' ')
```

Die Anzeige wurde **format**-technisch ein bisschen angepasst, damit die Werte ordentlich zueinander stehen.

```
>>>
  1000    156    175    158    186    152    173
166.67 -10.67   8.33  -8.67  19.33 -14.67   6.33
>>>
```

In weiteren Felder lassen sich nun auch andere Häufigkeits-bezogene statistische Kennwerte abspeichern. Da bietet sich z.B. die relative Häufigkeit an:

```
…
print()
rel_haeufigkeit=([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
for wert in range(0,6+1):
    rel_haeufigkeit[wert]=haeufigkeit[wert]/anzahlwuerfe
    print(format(rel_haeufigkeit[wert],"7.3f"),end=' ')
print()
```

```
>>>
   1000     175     188     172     171     144     150
 166.67    8.33   21.33    5.33    4.33  -22.67  -16.67
  1.000   0.175   0.188   0.172   0.171   0.144   0.150
```

Spätestens ab hier müssen wir uns um die Beschriftung kümmern, da nun nicht mehr deutlich wird, was da im Einzelnen berechnet und angezeigt wurde.

Aufgaben:

1. *Verbessern Sie das kleine Würfel-Statistik-Programm so, dass die Anzeigen verständlich werden!*
2. *Erweitern Sie nun das Programm um die prozentuale Abweichung vom Erwartungswert und der prozentualen Abweichung von der erwarteten relativen Häufigkeit! Warum sind die zusammengehörenden Werte immer gleich? (Tipp: Das Prozentzeichen lässt sich gut in der print-Option end unterbringen (end='%'), aber man könnte es auch am Ende der Zeile als Einheit ausgeben!)*
3. *Verändern Sie das Programm nun so, dass man beliebige Würfel (4er, 5er, … bis 10er) einsetzen kann!*
4. *Passen Sie nun das Programm auch noch so an, dass beliebige Wurfzahlen (max. 1'000'000) möglich sind! (Das Überschreiten der Zeilen bei der Wahl vielflächiger Würfel ignorieren wir hier mal!*

für Interessierte:

5. *Wie heißen eigentlich die Körper der ungewöhnlichen "Würfel"?*

für die gehobene Anspruchsebene:

6. *Informieren Sie sich, wie man z.B. bei etwas umfangreicheren Reihen herausbekommen kann, ob die Werte echt erwürfelt wurden oder sich der Nutzer die Werte nur mal so "zufällig" hat einfallen lassen!*

Neben den Häufigkeiten müssen wir in der Statistik auch vielfach die Kennwerte für bestimmte Gruppen von Werten berechnen. Zu den bekanntesten Kennwerten gehören sicher der Mittelwert und die – vielen Nutzern sehr imaginär anmutende - Standardabweichung. Was auch immer ihr Wert aussagen soll?
Ein typisches Beispiel ist die Messung der Masse eines Körpers. Wenn wir ins in der Praxis oft mit einer einzigen Messung zufrieden geben, ist das aus wissenschaftlicher Sicht zu unsicher. Man macht immer viele Messungen und betrachtet dann den Durchschnitt.
Im folgenden Programm werden die Messwerte per Eingabe erfasst und dann sollen nach und nach die statistischen Kennwerte dieser Reihe berechnet werden. Alternativ könnte man natürlich die Werte auch wieder direkt im Programm in ein Feld oder eine Liste schreiben. Vor allem beim Testen ist das wesentlich praktischer.

Für die Auswertung von Experimental-Daten kommen neben den gerade betrachteten eingruppigen Werten auch solche aus zwei zusammenhängenden Reihen in Betracht.

6.8. die Python-Schlüsselwörter im Überblick

```
False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

Scheinbar ist das Wörtchen **access** mit einer Bedeutungen belegt oder früher belegt gewesen. Es sollte deshalb nur mit Bedacht verwendet werden. Vor allem sollte man es nie als Bezeichner etc. nutzen.

False

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

None

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

True

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

and

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

as

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

assert

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

break

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

class

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

continue

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

def

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

del

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

elif

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

else

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

except

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

finally

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

for

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

from

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

global

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

if

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

import

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

in

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

is

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

lambda

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

nonlocal

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

not

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

or

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

pass

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

raise

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

return

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

try

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

while

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

with

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

yield

Syntax	Beschreibung

Beispiel(e)	Kommentar(e)

Links:
https://docs.python.org/3.5/library/ (engl. Beschreibung / Dokumentation der Python-Library's)

Python-Spicker

beliebig oft wiederholbar: {, *wiederholung*} optional / mögliche Ergänzung: [, *option*]
alternativ: *variante* | *variante* **Python-Schlüsselwörter und –symbole**
hilfsausdruck := reguläre Ausdrücke, Befehle, Strukturen

Eingabe:

```
variable = input( "Aufforderungstext" )          # allgemeine / Text-Eingabe
variable = eval(input( "Aufforderungstext" ))    # Zahlen-Eingabe
```

(formatierte) Ausgabe:

ausgabe:=wert | *berechnung* | *"Text"* | *'Text'*

```
print()
print(ausgabe)
print({ausgabe, } format(variable,formattext) {, ausgabe} )
```

Bsp.: *formattext .. '12s' '12d' '12.3f'* *s..String; d..dezimal (Ganzzahl); f..float (Kommazahl);*
… hier mit Platz für 12 Zeichen (und 3 Dezimalstellen)

```
print(ausgabe, {ausgabe, } end=drucksteuerung)
```

Bsp.: *drucksteuerung* *.. '\n' .. Zeilenumbruch;*
.. 'zwischentext' .. druckt Zwischentext ohne Umbruch

```
print("Text mit Platzhalter [%formattext] …" % (variable [, variable]))
```

.. formattext s.a. oben
!! Ausgabe des %-Zeichens in solchen Konstrukten mit %%

```
print("Text mit Platzhalter {[%platz]} [{[%12]}] …".format(variable [,variable]))
```

.. platz ist die Anzahl der reservierten Zeichen

Verzweigung:

```
if bedingung:        # Einleitung und Test/Bedingung
      befehle        # Then-/Dann-/Wahr-Zweig (eingerückt!!! mehrzeilig mögl.)
{elif bedingung:     # zusätzliche(r) untergeordnete(r) Test/Bedingung
      befehle}       # untergeord. Then-/Dann-/Wahr-Zweig
[else:               # optionaler Else-/Sonst-/Falsch/Rest-Zweig
      Befehle]
```

Schleifen:

```
while bedingung:                # while True:    # Endlosschleife
                                      …          # (meist break notwendig)
      befehle
      {continue}        # Sprung zum nächsten Schleifendurchlauf /-anfang
      {befehle
      break}            # Sprung hinter Schleife (noch hinter ELSE)
{else:
      befehle}

for laufvariable in liste | tupel:  # _ als laufvariable, wenn kein Gebrauch in
      befehle                                          Schleife geplant
      [verzweigung : break]          # vorzeitiger Abbruch der Schleife

for laufvariable in range([untere_grenze, ]obere_grenze[, schrittweite]):
      befehle
```

Funktion:

```
def funktionsname(argumente):
        befehle
        [return rückgabewert]
```

Bibliotheken:

Installation über pip (in der Console)

```
python -m pip install --upgrade pip          (Aktualisierung von pip)
pip3 install pygame-……whl                    (Installation des Moduls)
(pip3 install --upgrade pygame-……whl         (Aktualisieren / Überinstal
                                              lieren))
```

Verwendung im Quelltext:

```
import bibliothek
…
bibliothek.funktion(argumente)

from bibliothek import funktion {, funktion}
from bibliothek import *
import bibliothek as lokaler_name
```

z.B.: Würfeln, klassisch

```
import random
dir(random)   #Anzeige Fkt.n
```

```
from random import randint
…
x=randint(1, 6)
```

```
import random
…
x=random.randint(1, 6)
```

```
import random as rdm
…
x=rdm.randint(1, 6)
```

```
from random import *
…
x=randint(1, 6)
```

wichtige Bibliotheken:

math	.. diverse mathematische Funktionen	**sys**	..
re	.. Arbeiten mit regulären Ausdrücken	**turtle**	.. Turtle-Graphik
datetime	.. Zeit- und Datums-Funktionen	**pickle**	..
os	.. Kommunikation mit Betriebssystem		..
shutil	.. Arbeiten mit Dateien und Ordner auf Shell-Ebene		
sqlite3	.. Kommunikation mit einem SQLite3-Server		
	..		

Objekt / Klasse:

```
class klassenname:
  klassenttributname=vorbelegung # übergreifend für alle Objekte/Instanzen

  def methodenname(oberklasse | self, argumente):
    pass                  # gestattet Klassendefinition ohne Implementierung
  def __init__(oberklasse | self, argumente):              # Konstruktor
  def _methodenname_(…):    # anschein-geschützte Methode (protected)
  def __methodenname__(…): # geschützte Methode (private) unsichtbar
    self.attributname=vorbelegung     # Obj.-Attribut, für jede Instanz extra
    self._attributname=vorb.          # anschein-geschütztes Attribut
    self.__attributname=…     # geschütztes Attribut, unsicht (→ get/set !)
```

7. Problem-Lösen mit Python

7.0. Aufgaben versus Probleme

besser wahrscheinlich Aufgaben-Lösen

Problem-Lösen ist eine Stufe komplizierter und geht im Allgemeinen davon aus, das es noch keine Lösung / keinen Algorithmus zur Bearbeitung gibt bzw. dieser nicht sofort offensichtlich ist

meist Umsetzung von Aufgaben-Stellungen / Pflichten-Hefte in Software gemeint

in der Software-Entwicklung wird aber allgemein von einer Problem-Umsetzung gesprochen
das Erledigen von Aufgaben hat so etwas Profanes / Minderanspruchvolles
das Schreiben von Routine-Funktionen ist nicht die Herausforderung, die Software-Entwickler mit dem Image ihres Berufsstandes verbinden
sie brauchen echte Herausforderungen, welche an die Grenzen der Technik oder der Programmiersprache herankommen
es ist quasi ein Kampf Mann (oder Frau) gegen Maschine, in dem man auf sich allein gestellt ist und unbedingt zum großen Helden werden muss
problematisch ist, dass weder die Chef's der Entwicklungs-Abteilungen noch die Nutzer das honorieren, sie können die Komplexität der Programmierung einfach nicht einschätzen
selbst, die im Team mitarbeitenden Programmierer bekommen von der Heldentat nichts mit, weil sie ihren eigenen einsamen Kampf führen

also was macht der unbemerkte Held - er programmiert so, dass kein anderer sein Programm versteht, so ist ihm vielleicht ein verspäteter Ruhm in Aussicht gestellt

ein Mittel dagegen - mit vielen weiteren Vorteilen - ist die Paar-Programmierung (Pair programming, Tandem-Programmierung)
zwei Programmierer arbeiten gemeinsam und gleichzeitig an einem Problem
der eine tippt und der andere kontrolliert gleich mit, der eine ist also aktiv, der andere eher passiv
nach einer aktiven Zeit wechseln die Paar-Mitglieder ihre Rolle
Kombinationen aus Frauen und Männern haben sich i.A. am Besten bewährt, sie gehen unterschiedlich an Probleme heran, hier ergänzen sich diese Vorgehensweisen
da bleibt die Aufmerksamheit länger enthalten, weil sich die Tätigkeiten abwechseln
große Aufmerksamkeit wird auch auf die Verständlichkeit des Code's gelegt
die Paare werden regelmäßig neu zusammengestellt, damit sich keine eingeschworenen Team's bilden

Vorteile der Paar-Programmierung:

- weniger Programm-Fehler
- gegenseitiges Lernen und Lehren
- mehr Freude an der Arbeit (auch bei Routine-Programmier-Aufgaben)
- kleinere / effektivere Programme
- höhere Disziplin bezüglich Absprachen, Team-Regeln, …
- besserer Code
- Arbeitsabläufe werden belastbarer
- geringeres Risiko, das Know how eines Team's zu verlieren, wenn Mitarbeiter Projekte / Firmen / … wechseln
- Paare werden seltener in der Arbeit unterbrochen

- Paar-Programmierung kann auch verteilt / distanziert (z.B. über das Internet) erfolgen

Eine Programmierer-Regel sagt, dass das Finden von Fehlern erst in der Praxis oder bei ersten Tests ungefähr 10x so teuer / aufwändig ist, wie das sofortige Erkennen in der Entwicklungs-Phase
Aber wo es soviele Vorteile gibt, sind die Schattenseiten nicht weit.

Nachteile / Probleme der Paar-Programmierung:

- Paare müssen sich immer wieder aufeinander einstellen, das kostet Arbeitszeit
- Leistungs-Niveau beider Programmierer muss ähnlich sein
- effektiv verlangsamt sich die Programmierung im Vergleich zur parallelen Arbeit beider Programmierer an jeweils anderen Aufgaben
- Urheber-Rechte
- Haftung bei Problemen

Vielfach wird natürlich mit der Entwicklung neuartiger Programm auch informatisches Neuland betreten.

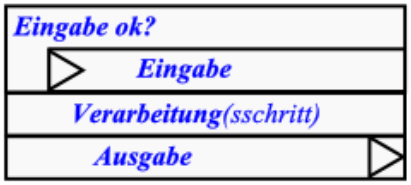
7.0.1. Programm-Entwicklungs-Strategien

Wie fängt man ein komplexeres Projekt sinnig an? Es einfach von vorne bis hinten in einem Ritt zu schreiben, birgt viele Risiken. Was passiert, wenn es vielleicht gar nicht in die Sprache umsetzbar ist? Oder vielleicht möchte der Auftraggeber auch mal Zwischenergebnisse sehen?

In der Praxis haben sich Grund-Techniken für die Programm-Entwicklung herauskristallisiert. Bei der einen Variante – dem **Top-down-Entwurf** – beginnt man mit einem sehr einfachen Programm-Rahmen. Im einfachsten Fall ist es einfach nur der Aufruf eines leeren Hauptprogramms.

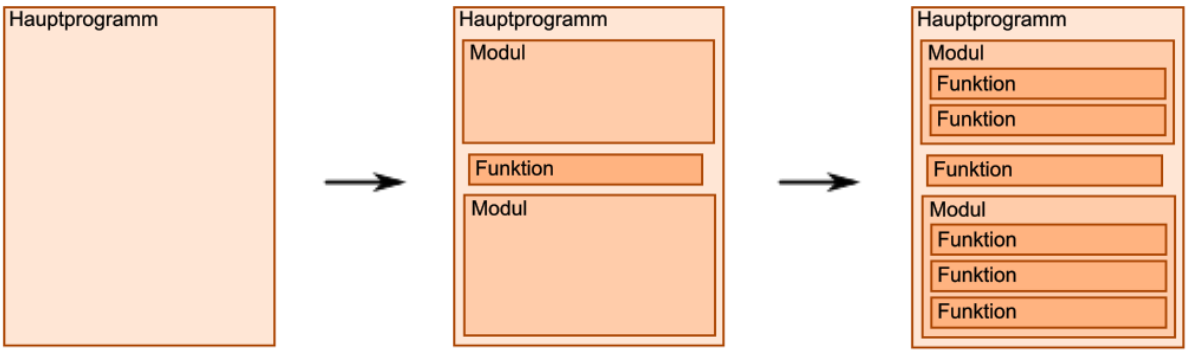
Nach und nach ergänzt man nun einzelne Komponenten. Z.B. könnte man das leere Hauptprogramm in die Teile Eingabe, Verarbeitung und Ausgabe strukturieren.

Im nächsten Schritt erweitert, ergänzt oder verbessert man die einzelnen Komponenten bis man schließlich ein fertiges Produkt erzielt.



Man spricht hier von Deduktion. Die Entwicklung erfolgt quasi von oben nach unten, vom Allgemeinen zum Speziellen.

Der große Vorteil dieser Variante ist, dass man praktisch zu jeder Zeit ein funktionierendes Programm hat, dass nach und nach immer besser / Leistungs-fähiger / Fehler-freier wird.



Erstellen eines Rahmenprogramm's

Erweitern um Modul-Grundgerüste

Ergänzen der Arbeitsfunktionen

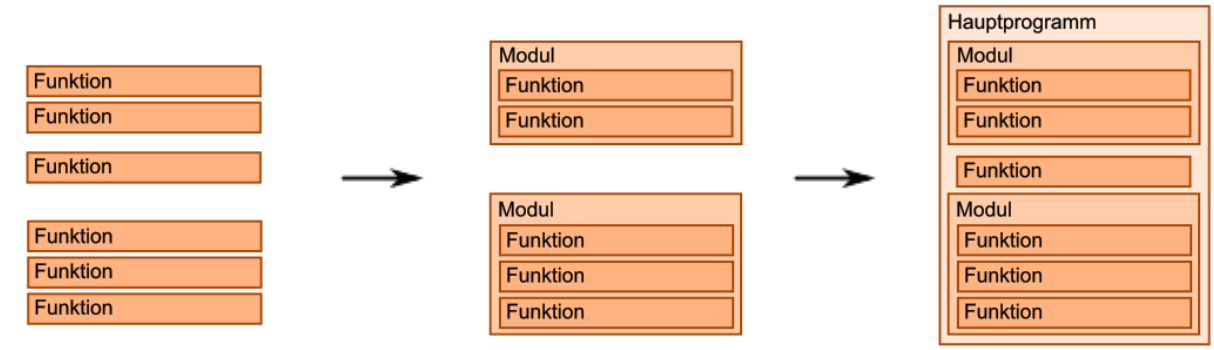
Nachteilig wirkt sich hier aus, dass der komplizierteste Teil – die spezielle Datenverarbeitung – irgendwie fast immer zum Schluss übrig bleibt. Wenn jetzt was nicht läuft, dann hat Huston ein wirkliches Problem. Die möglichen Konsequenzen sind halbfertige Programme, die erst beim Kunden reifen (sogenannte Bananen-Software) oder Verzögerungen beim Zeitablauf.

Sachlich steckt hinter diesem Programmier-Prinzip die Dekomposition. Sie beinhaltet die Zerlegung / Auflösung eines Ganzen in immer kleiner werdende Teile / Segmente. In der Programmierung sind das dann Module, Unterprogramme, Prozeduren oder Funktionen.

Natürlich kann man auch zuerst die spezielle(n) Funktion(en) entwickeln und testen. Schrittweise, werden dann zusätzliche Komponenten hinzugefügt, bis schließlich ein fertiges Programm entstanden ist. Diese Technik nennt man **Bottom-up**.

Es handelt sich hier um eine Induktion, also einer Entwicklung von unten nach oben, vom Speziellen zum Allgemeinen.
Vorteilhaft ist die frühzeitige Fertigstellung der kritischen Programmteile.
Als Nachteil kann sich dabei herausstellen, dass man zwar super Leistungs-fähige Funktionen entwickelt hat, denen aber ein verbindendes Großes-Ganzes fehlt.
Fehlen dann bestimmte Forderungen aus dem Pflichten-Heft, dann sind vielleicht auch sehr aufwändige Nachkorekturen an den Kern-Funktionen notwendig. Das bedeutet dann erneute Tests, Anpassungen usw. usf. Auch bei dieser Projekt-Lösung kann es zu erheblichen Verzögerungen der Fertigstellung kommen.

Die Bottom-up-Technik ist praktisch eine Aggregation. Segmente / Teile / Funktionen / … werden zu einem Großen-Ganzen vereint.

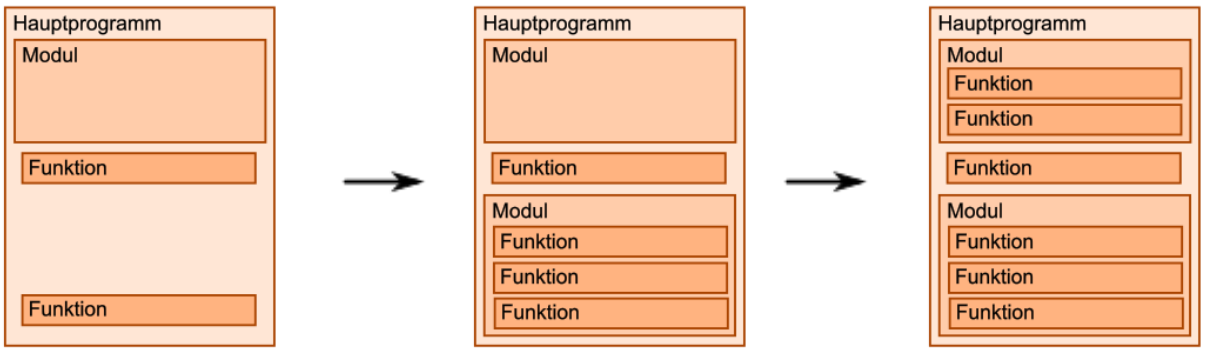


Erstellen der Basis-Funktionen

Zusammenfassen zu Modulen

Zusammenstellen des Hauptprogramm's

Heute werden häufig Top-down- und Bottom-up-Methoden kombiniert. In vielen Software-Schmieden gibt es fertige Sammlungen von Funktionen, die in eine Top-down-Entwicklung nach und nach integriert werden.
Man verlässt sich auf die geprüfte Leistung vorgefertigter Funktionen und hat zu jeder Zeit ein mehr oder weniger gut funktionierendes Programm.



Prototyp mit ersten Funktionen

Erweitern und Modularisieren

Ergänzen fehlender Funktionen

Eine weitere Strategie, die in der letzten Zeit viel von sich reden lassen hat, ist "**Design Thinking**". Darunter versteht man Methoden, Verfahren und Fähigkeiten, sich in Aufgaben-

stellungen und / oder Probleme hineinzufühlen, sie kreativ zu bedenken, Ideen zu kommunizieren sowie produktiv und kollaborativ zusammenzuarbeiten. Vieles wird vorrangig aus der Sicht des Nutzers / Endverbrauchers betrachtet und dieser steht auch im Mittelpunkt. Letztendlich muss dieser mit dem Produkt leben und arbeiten.
In die Entwicklung eines Produkt's oder der Lösung eines Problem's sollen von Anfang an möglichst alle beteiligte Personen-Gruppen einbezogen werden. Team-working, selbstkritisches und kollaboratives Arbeiten sind Kern des Arbeitens. Fehler dürfen gemacht werden, sollen aber möglichst frühzeitig erkannt werden, ohne nach "Schuldigen" zu suchen. Es sollen schnellsens Korrekturen, Verbesserungen und Erweiterungen umgesetzt werden.

Phasen des Design Thinking

• **Phase 1: Situations-Analyse**	Wie ist der aktuelle Stand? Welche Situation ist unbefriedigend? Welches Problem gibt es?
• **Phase 2: Perspektiven entwickeln**	Was wäre das Tollste / Utopischte / …, was man sich als Lösung des Problem's oder der Situation denken könnte? Sei kreativ! (Be creative!) Was gefällt an anderen Lösungen nicht?
• **Ideen generieren**	Was kann mit den gegebenen Mitteln realisert werden? Was soll unbedingt realisiert werden?
• **Konzept-Entwicklung**	Entwickeln erster Prototypen, die einzelne Teil-Probleme lösen, die einen Eindruck von der Lösung geben bzw. die einen ersten Lösungs-Ansatz umsetzen.
• **Testen des Konzept's**	Ausprobieren der Teil-Lösungen sowie des fertigen Produkt's in der Real-Umgebung. Solange die Lösung unbefriedigend oder noch unvollständig oder erweiterbar ist, wird wieder mit Phase 1 gestartet.

Aufgaben:

1. *Vergleichen Sie die Startegien "Top-down", "Bottom-up" sowie die gemischte in einer geeigneten Tabelle!*
2.

7.0.2. Strategien zur Lösung von (echten) Problemen

Zerlegen des Problems in kleinere Aufgaben / Probleme
Analogien zu anderen Problemen suchen (und dann deren Lösungen als Grundlage benutzen)
Versuch und Irrtum (trial and error)
Lernen durch Einsicht

Hilfsmittel / Techniken:

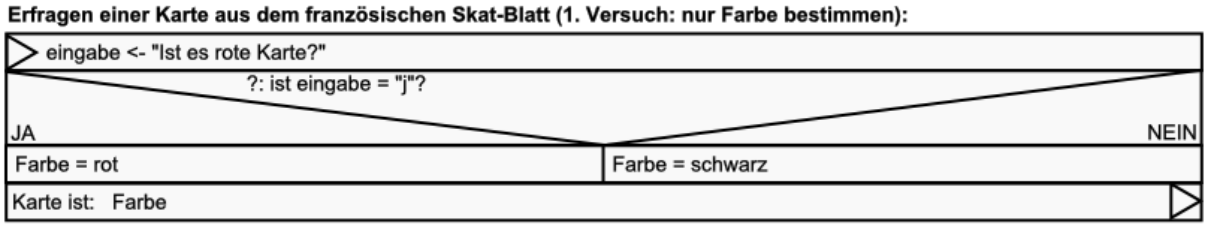
- **Brainstorming**
- **Mind Mapping**
- **Concept Mapping**
- **Kopfstand-Technik**
  Umkehr-Technik
  Flip-Flop-Technik
  1. Aufgabenstellung umdrehen
  2. Lösung für diese Aufgabe suchen
  3. Lösung auf den Kopf stellen
  4. Lösung anpassen / optimieren
- **Negativ-Konferenz**
- **Provokations-Technik**
  Provokation z.B. durch Verallgemeinerung, Pauschalisierung, … als Inspirations-Quelle / zum Verlassen der eingetretenen Denkpfade
- **Superposition**
- **kollektive Notizzettel (collective Notebook, CNB)**
  über einen bestimmten Zeitraum sammeln die Team-Mitglieder ihre Gedanken, Assoziationen, Geistesblitze und Ideen auf Notizzetteln zu notieren → 3 Phasen:
  1. Vorbereitung (Problemstellung formulieren; Teilnehmer auswählen; Notizblöcke bereitstellen)
  2. Durchführung (Notizen machen (spontan und täglich); persönl. Zusammenfassung erstellen)
  3. Auswertung (Zusammenfassungen abgleichen; Notizen durchgehen; Basis-Vorschläge für Lösung heraussuchen / ableiten; Konzept-Erstellung)
- **Pinnwand-Moderation ähnlich: Clustern**
  Sammeln von Ideen- wie beim Brainstorming – allerdings auf Kärtchen / Post-ist; wiederholte Gruppierung der Kärtchen und Gruppen-Benennung; Zusammenfassung des Ergebnisses in möglichst neutraler Form
- **EDISON-Prinzip**
  1. Erfolgs-Chancen erkennen
  2. eingetretene Pfade verlassen
  3. Inspirationen suchen
  4. Spannung erzeugen
  5.Ideen und Erkenntnisse ordnen
  6. Nutzen herausziehen
- **progressive Abstraktion**
  Finden von Zusammenhängen zwischen Problem und erwarteter Lösung; Festlegen von Maßnahmen(-Ebenen) die am Erfolg-versprechendsten erscheinen
- **semantische Intuition**
-

für klassische (sofort lösbare) Aufgaben ("einfache Probleme") bietet sich die folgende Vorgehensweise an:

1. Erfassen des IST-Zustandes (IST-Ananlyse, Ist-Aufnahme, …)
2. Erkennen / Aufzeigen des Unterschiedes / Widerspruchs zum SOLL-Zustand
3. Suchen nach geeigneten Lösungs-Verfahren / Algorithmen
3. Anwendung eines Lösungs-Verfahrens (/ Algorithmus)
4. Prüfen des erreichten Standes bis IST und SOLL gar nicht mehr nur noch im akzeptablen Maß abweichen (ansonsten quasi Zurücksprung zu 1.)

Beispiel für Top-down-Strategie: Erfragung einer Karte aus dem französischen Blatt

1. Prototyp – Test des Verfahrens



Die Umsetzung des Struktogramm wird ganz schematisch erledigt. Für den ersten Versuchs-Prototypen verzichten wir auf fast jeden Schnickschnack. Wir wollen lediglich sehen, ob es funktioniert.

```
eingabe = input("Ist es eine rote Karte? (j)")
if eingabe == "j":
    farbe = "rot"
else:
    farbe = "schwarz"

print("Die Karte ist: ",farbe)
```

Eingabe-Block
Verzweigungs-Block
Ja-Zweig
Nein-Zweig
Ausgabe-Block

```
>>>
```

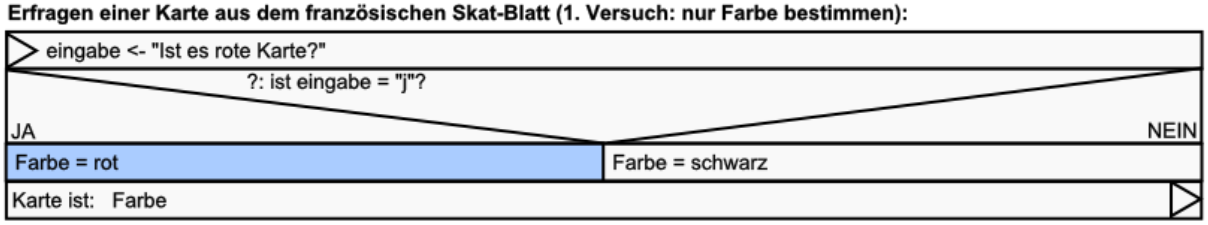
Neben der reinen Funktionalität, sollte man auch gleich die Handhabbarkeit prüfen. Wie kann man die Eingaben für den Nutzer am Einfachsten machen, was macht ein Nutzer intuitiv? Nichts ist nerviger, als eine umständliche Bedienung. Wird der Nutzer z.B. auf deutsch gefragt und muss er dann aber mit "y" für "ja" antworten, da geht das spätestens bei der zweiten Fragen mächtig auf den Docht. Grundsätzlich muss man sich da von seiner egomanischen, selbstbezogenen Zufriedenheit trennen. Es gibt für den Programmierer nur einen "Gott" / eine Werte-Instanz und der sitzt vor dem Computer und versucht das Programm zu bedienen.
Für unsere Ja-Nein-Fragen werden wir nur die Abfrage auf "j" programmieren, das ist verständlich und auch gut zu programmieren. Um vielleicht noch etwas flexibler zu sein, fragen wir auch den Groß-Buchstaben ab. Die notwendige Erweioterung des Programms ist leicht gemacht und wenn wir dann ein Grundgerüst für das Fragestellen und –auswerten haben, dann können wir die restlichen "paar" Fragen mit copy-and-paste dazuprogrammieren.

```
eingabe = input("Ist es eine rote Karte? (j)")
if eingabe == "j" or eingabe == "J":
    farbe = "rot"
else:
    farbe = "schwarz"

print("Die Karte ist: ",farbe)
```

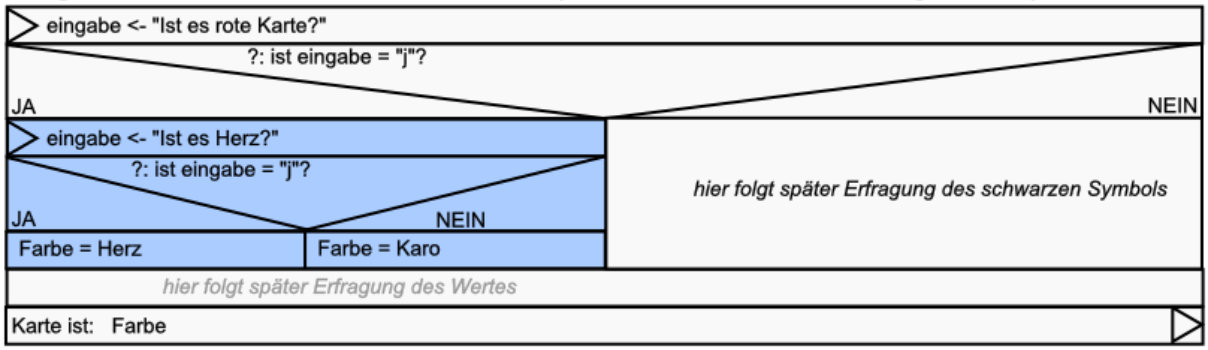
Erweiterung um "J"

2. Schritt – Austausch eines allgemeinen Blockes gegen differenzierte Blöcke

Erfragen einer Karte aus dem französischen Skat-Blatt (1. Versuch: nur Farbe bestimmen):



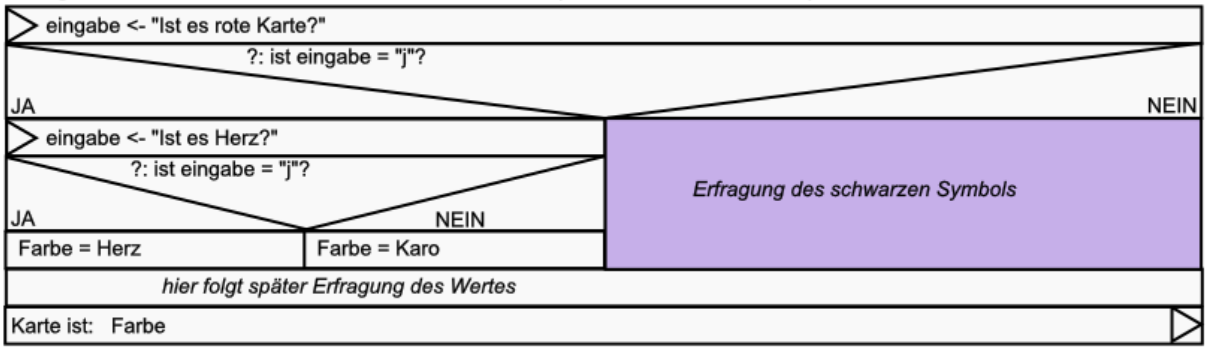
Erfragen einer Karte aus dem französischen Skat-Blatt (1. Teil: Farbe bestimmen; Teillösung rote Karte):



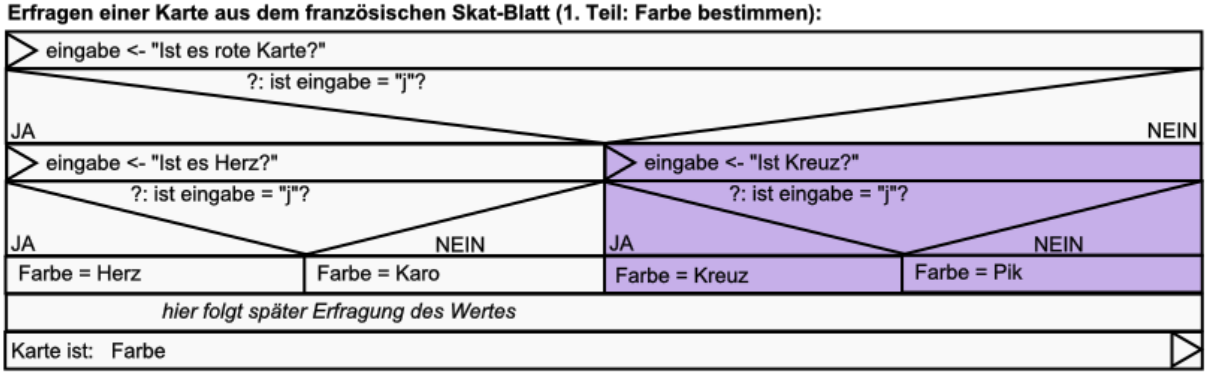

```
>>>
```

3. Schritt – Erweiterung / Vervollständigung

Erfragen einer Karte aus dem französischen Skat-Blatt (1. Teil: Farbe bestimmen):



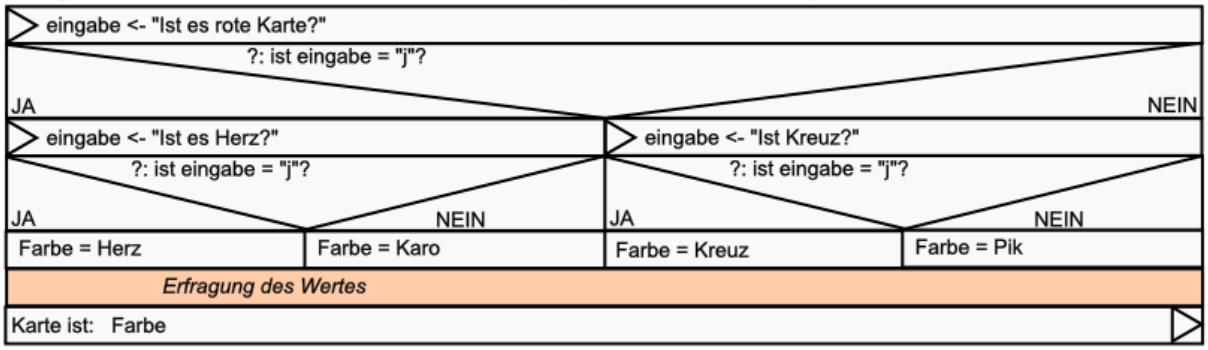
Erfragen einer Karte aus dem französischen Skat-Blatt (1. Teil: Farbe bestimmen):



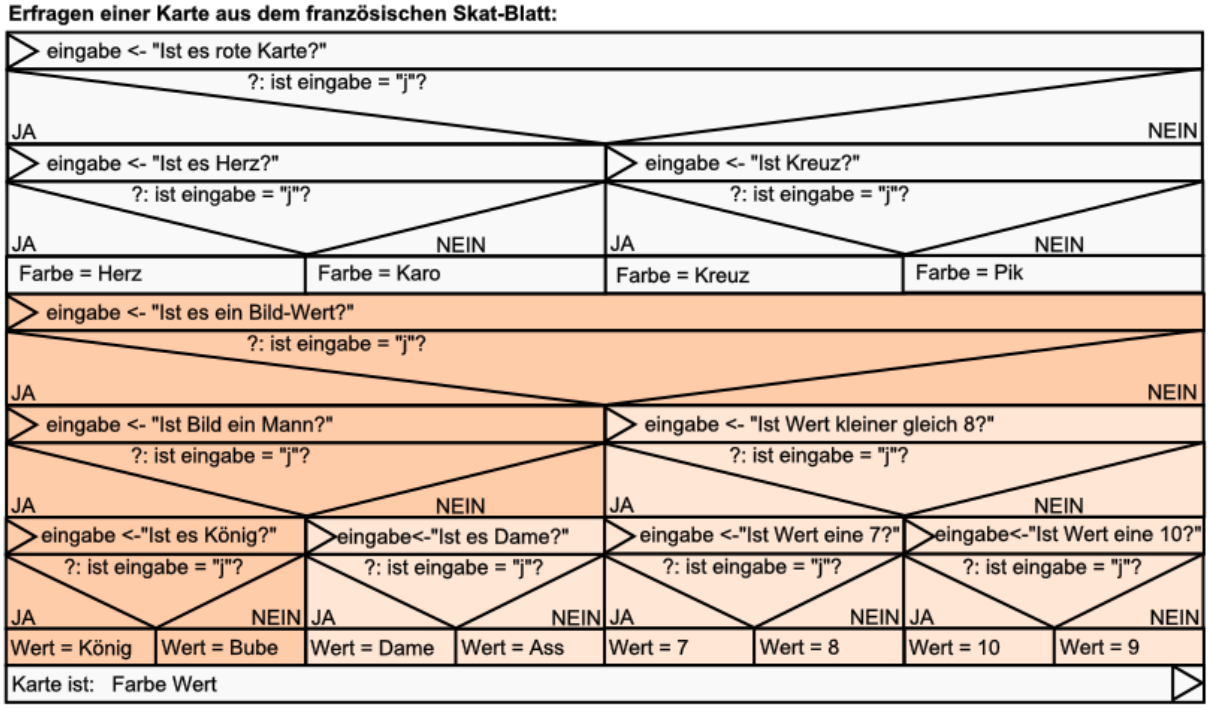

```
>>>
```

4. Schritt – nächster Abschnitt

Erfragen einer Karte aus dem französischen Skat-Blatt (1. Teil: Farbe bestimmen):



Erfragen einer Karte aus dem französischen Skat-Blatt:




```
>>>
```

letzter. Schritt – Verschönerung / Verfeinerung / Benutzerführung optimieren

```
>>>
```

kleine Programm-Beispiele

7.0.3,14 Python am Pi-Day

Am 14. März ist der Pi-Day. An diesem Tag beschäftigen sich Mathematiker und Schüler mit der wohl berühmtesten Konstante der Kreiszahl π. Das Datum ergibt sich aus der amerikanischen Datum-Notation "3/14". Die ganz hart Gesottenen feiern exakt um 01:59:26 Uhr, um Pi bis auf die 7. Nachkommastelle zu ehren.
Auch der 22. Juli wird gelegentlich als Pi-Annäherungstag zelebriert. Hier ergibt sich das Datum aus dem Bruch 22/7, der rund 3,14 – also π ergibt.
Hier seien einige Programme vorgestellt, die Pi auf irgendeine Variante berechnen oder ermitteln.
Vielleicht geht der eine oder andere Quelltext über das gegenwärtige verständnis von Python hinaus, das soll aber bei einem so spannenden Thema nicht das Abbruch-Kriterium sein.

Pi-Berechnung durch Monte Carlo Simulation

```
# pi-monte_carlo.py
# pi-Bestimmung mit der Methode von Monte Carlo
from random import random
print "Monte Carlo Methode zur"
print "Näherung für pi:"
g = input("Gesamtzahl der Tropfen: ")
v = 0
x=0; y=0  # Koordinaten des Punktes P
for i in range(1,g+1):
      x = random()
      y = random()
      if x*x+y*y<= 1:
         v = v + 1
pi_naeh = 4.0*v/g
print g,"Tropfen, davon",v,"Tropfen im Viertelkreis,"
print "pi etwa",pi_naeh
```

Q: http://www.michael-holzapfel.de/progs/python/python_beisp.htm

Pi-Berechnung über Verhältnis der Flächen von äußeren und inneren Vieleck

Interations-Term

$$s_{n+1} = \frac{s_n}{\sqrt{2+\sqrt{4-s_n^2}}}$$

```
# pi-berechn2.py
# pi-Berechnung mit regulären 2n-Ecken
from math import sqrt, pi
n = 6  # Start mit regulärem Sechseck
s = 1  # Seitenlänge des reg. Sechsecks
print "Schrittweise Näherung von pi mit Hilfe eines 2n-Ecks"
for i in range(1,21):
    pi_naeherung = 0.5*n*s
    print pi_naeherung
    s = s/sqrt(2+sqrt(4-s*s))
    n = 2*n  # doppelte Eckenzahl
print "Gute Iteration!"
print "pi =",pi
```

Q: http://www.michael-holzapfel.de/progs/python/python_beisp.htm

Exkurs: besondere Zahlen – Stoff für viele Python-Programme

(einfache) besondere Zahlen

Dreieckszahlen: — **Beispiele / Folge:** 1, 3, 6, 10, 15, …

Dreieckszahlen lassen sich über die Formel $x_i = \frac{i\,(i+1)}{2}$ berechnen (mit *i* dem Rang der Zahl).
Die 10 ist nach PYTHAGORAS eine heilige Zahl, weil sie sich aus der Summe der ersten i Zahlen (also: 1+2+3+4) ergibt. Außerdem lässt sich daraus ein vollkommen gleichseitiges Dreieck legen.

EULERsche Zahl: — **Wert:** 2,718'281'828'…

Die EULERsche Zahl e berechnet sich als Grenzwert $e = \lim \left(1 + \frac{1}{n}\right)^n$ bzw. als unendliche Folge $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ bzw. in der Summen-Schreibweise $e = \sum_{k=0}^{\infty} \frac{1}{k!}$
Die EULERsche Zahl ist eine der bedeutenden Konstanten in der Naturwissenschaft und Mathematik.

goldener Schnitt: — **Beispiele:**

irrationale Zahl
$= \frac{1+\sqrt{5}}{2}$
Der Quotient aus zwei aufeinanderfolgenden FIBONACCHI-Zahlen nähert sich immer mehr dem goldenen Schnitt an.

Kreiszahl π: — **Wert:** 3,141'592'654'…

irrationale Zahl
stellt Verhältnis von Umfang und Durchmesser eines Kreises dar
Verbindung zum goldenen Schnitt 6/5 [2]

narzißtische Zahlen: — **Beispiele:**

Eine narzißtische Zahl mit n Stellen ist gleich groß der Summe der n.-Potenzen ihrer Ziffern.

$153 = 1^3 + 5^3 + 3^3$
$54748 = 5^5 + 4^5 + 7^5 + 4^5 + 8^5$

Vampir-Zahlen: — **Beispiele:**

Vampir-Zahlen haben eine gerade Anzahl von Stellen und lassen sich aus einer beliebigen Multiplikation von Zahlen, die halb soviele Stellen besitzen, wie die Vampir-Zahl selbst hat, keine führende Null beinhalten und insgesamt alle Ziffern der Vampir-Zahl enthalten.

1260 = 21 * 60
1530 = 30 * 51
2187 = 27 * 81

Urantia-Zahlen: — **Beispiele:**

Zahl des heiligen AUGUSTINUS: — **Beispiele:**

Die Zahl des heiligen AUGUSTINUS ist die erste narßistische Zahl. Sie lässt sich auch aus der Summe der Fakultäten von 1 bis 5 berechnen.

153 = 1! + 2! + 3! + 4! + 5!
$153 = 1^3 + 5^3 + 3^3$

???: — **Beispiele:**

über Teilersummen definierte besondere Zahlen

Teilersumme: — **Beispiele:**

die Teiler-Summe σ (einer Zahl) ist die Summe aller ihrer Teiler beginnend bei 1 und abschließend mit der Zahl selbst	$\sigma(12)$ = 1+2+3+4+6+12 = 28
echte Teilersumme: die echte Teiler-Summe σ^* (einer Zahl) ist die Summe aller ihrer Teiler beginnend bei 1 und dabei die Zahl selbst ausschließend	**Beispiele:** $\sigma^*(12)$ = 1+2+3+4+6 = 16
defiziente Zahl: eine Zahl heißt dedizient (oder teiler-arm), wenn die die echte Teilersumme kleiner als die Zahl selbst ist $\sigma^*(n) < n$	**Beispiele:** $\sigma^*(10)$ = 1+2+5 = 8 8 < 10
abundante Zahl: eine Zahl heißt dedizient (oder teiler-reich), wenn die die echte Teilersumme größer als die Zahl selbst ist $\sigma^*(n) > n$	**Beispiele:** $\sigma^*(12)$ = 1+2+3+4+6 = 16 16 > 6
vollkommene Zahl: eine Zahl heißt vollkommen, wenn die die echte Teilersumme die Zahl selbst ist $\sigma^*(n) = n$	**Beispiele:** $\sigma^*(6)$ = 1+2+3 = 6 6 = 6

befreundete Zahlen: Befreundete Zahlen sind zwei unterschiedliche natürliche Zahlen, bei deren die echte Teilersumme genau der anderen Zahl entspricht.	**Beispiele:** 1184 und 1210 5020 und 5564

sonstige (ganz) besondere Zahlen

???:	**Beispiele:**
???:	**Beispiele:**

seltene oder ungewöhnliche Zahlen und Zahlensysteme (in der Schule)

komplexe Zahlen: → in Python intern definiert, dadurch sofort nutzbar	**Beispiele:**
???:	**Beispiele:**

Q:

8. Python für Fortgeschrittene

Nachdem wir die grundlegenden Elemente von Python besprochen haben, gehen wir jetzt mehr in die Detail's. Natürlich gibt es keine echte Grenze zwischen Grundlagen und fortgeschrittener Programmierung. Ab nun schauen wir auch intensiver hinter die Oberfläche und kümmern uns um spezielle Eigenschaften und Möglichkeiten.
Auf den folgende Seiten verzichte ich jetzt auch dann und wann mal auf die farbige Darstellung der Quell-Texte. Wer an dieser Stelle einsteigt, sollte genug Grundkenntnisse besitzen, um mit Python umzugehen. Natürlich können einzelne Quelltexte und Programme auch im Blindflug benutzt und ausprobiert werden. Ob das Sinn macht, muss jeder für sich entscheiden. Aber ein Blindflug wird auch nicht an der Farbigkeit des Quelltextes im Editor scheitern.

Wegen der besonderen Bedeutung von Texten besprechen wir diese hier in einem gesonderten Abschnitt. Im Programmier-Jargon heißen sie Strings (engl. Fäden) oder Zeichenketten.
Einige Programmiersprachen betrachten Zeichenketten auch als eigenständigen / erweiterten Datentyp. Python trennt hier nicht so streng.

8.1. Strings – Zeichenketten

Wie wir schon vilefach gesehen haben, sind Zahlen für sich nicht sehr informativ. Wir brauchen immer Beschreibungen, um die Zahlen in sinnvolle Zusammenhänge zu bringen. Schon die Ausgabe des Namens einer (physikalischen) Größe oder die Nennung einer Einheit sind mit Text-Symbolen verbunden. Erst so macht z.B. eine "21" Sinn. Wenn denn nämlich noch "Temperatur" und die Einheit "°C" dazu angegeben wird, dann verstehen wir die 21 auch im Speziellen.
Heute ist die Verarbeitung von Zeichenketten eine der häufigsten Tätigkeiten / Aufgaben für Programmierer. Viele Daten liegen zuerst einmal als Zeichenketten vor. Bevor man sie für Berechnungen usw. nutzen kann, müssen sie ersteinmal aufgearbeitet werden (→ 8.2. Datentypen und Typumwandlungen).

8.1.1. einzelne Symbole / Zeichen / Charaktere

wir sprechen auch von Charakteren – abgekürzt in vielen Programmiersprachen mit char oder chr
gemeint ist die Repräsentation von Zeichen im ASCII-Zeichensatz oder in den modernen Versionen der Programmiersprachen im Unicode-Zeichensatz

Symbole müssen im Programm-Text entweder in einfache Hockkommata oder Anführungszeichen gesetzt werden
immer nur ein gültiges Zeichen

Beispiele:
'a'
'1'
' '
'#'

Speicherung in Variablen möglich

Umwandlungen von Symbolen und ASCII-Code

ord()
gibt für ein Zeichen / Charakter den ASCII-Code zurück

chr()
wandelt einen ASCII-Code (Ganzzahl!) in ein Symbol / Charakter um

8.1.2. Sequenzen von Zeichen - Zeichenketten / Strings

erste allgemeine und unterschwellige Besprechung schon weiter vorne (u.a. → 6.1. Ausgaben und 6.4.2.2. Sammlungs-bedingte Schleifen)
hier noch einmal mit zusammenfassendem Charakter

im Programmierer-Jargon Strings genannt

Zeichenkette ist eine Symbol-Folge
Im Programm-Text / Listen usw. müssen Strings / Texte entweder in einfache Hockkommata oder Anführungszeichen gesetzt werden
Empfehlung (aber kein Muss!) einzelne Symbole in einfache Hochkommata und Strings in Anführungszeichen

Zeichenketten sind unveränderlich (immutable), einmal definiert sind sie nur durch direktes oder indirektes Kopieren / Manipulieren zu verändern
eine Zeichenänderung über zeichenkette[3] = 's' ist nicht möglich

Symbole / Zeichenketten lassen sich durch Addition (+) verketten / konkatenieren
ein Symbol / eine Zeichenkette läßt sich durch Multiplikation (*) wiederholend verketten / konkatenieren

Vergleich – wie in Python üblich – über == bzw. !=
für die andereren Vergleiche gelten die lexikalischen Ordnungen
ein längerer String ist immer größer

es läßt sich der **in**-Operator verwenden
also prüfen, ob ein Symbol / Teilstring in einem anderen String enthalten ist

len()
Länge der Zeichenkette / Buchstaben-/Zeichen-Anzahl

str()
Umwandlung in einen String

Zugriff auf einzelne Zeichen über den Index
Zählung beginnt mit 0 für das erste Zeichen

zeichen = zeichenkette[Index]

es ist auch der Zugriff auf Zeichenketten-Abschnitte möglich (Slice-Notation)
zeichenkettenabschnitt = zeichenkette**[:3]** liefert die ersten drei (3) Zeichen (Quasi bis zum 3. Zeichen)
zeichenkettenabschnitt = zeichenkette**[3:]** kopiert alle Zeichen ab dem dritten bis zum Zeichenkettenende
zeichenkettenabschnitt = zeichenkette**[4:7]** in der Variable zeichenkettenabschnitt befindet sich die Zeichen von Position 4 bis 6 (also nicht mehr 7)

die Indizes können auch negative Zahlen sein, dann wird von rechts nach links – also quasi vom Ende her – gearbeitet (0 und -0 ist aber die gleiche Position!)
[:-3] liefert die Zeichenkette ohne die letzten drei Zeichen
[-4:] liefert die letzten vier Zeichen der Zeichenkette

Ausgaben mit Platzhaltern

print("Hauptzeichenkette %s Restzeichenkette" % "Zeichenkette")
print("Hauptzeichenkette %s Restzeichenkette" % ZeichenkettenVariable)

print("Hauptzeichenkette %s Restzeichenkette %s weitere Zeichenkette" % (Zeichenkette1, Zeichenkette2))

8.1.1. Objekt-orientierte Nutzung von Strings

Das hört sich irgendwie gefährlich an – Objekt-orientierte Nutzung von Strings – ist aber eigentlich gar nicht sowas Neues. Viele der schon besprochenen / genutzten Module realisieren genau das moderne Objekt-Konzept. Wir werden uns später genauer damit beschäftigen. Also keine Angst – einfach ran an die Bouletten.

Zeichenkette**.strip()**
Zeichenkette**.strip([zeichen]):**:
Entfernt Leerzeichen und Zeilenumbrüche von den Enden des Strings
innere Leerzeichen und Zeilenumbrüche bleiben erhalten

Zeichenkette**.lower()**
Umwandlung in Kleinbuchstaben

Zeichenkette**.upper()**
Umwandlung in Großbuchstaben

Zeichenkette**.append(e)**

Zeichenkette**.extend(l)**

Zeichenkette**.count(e)**

Zeichenkette**.index(e)**

Zeichenkette**.insert(i,e)**

Zeichenkette**.pop(i)**

Zeichenkette**.remove(e)**

Zeichenkette**.reverse()**

Zeichenkette**.sort()**
Zeichenkette**.sort(reverse=True)**

Zeichenkette**.find(e)**
Zeichenkette**.find(e,istart)**
Zeichenkette**.find(e,istart,iende)**

Zeichenkette**.rfind(x,istart,iende)**

Zeichenkette**.rjust()**

Zeichenkette**.ljust()**

Zeichenkette**.replace(ealt,eneu)**

Zeichenkette**.endwith(zeichen,anzahl)**

Zeichenkette**.split()**
Zeichenkette**.split(Trennzeichen)**
split() teilt eine Zeichenkette in Wörter auf. Diese Wörter werden als Liste zurückgegeben. Als Ternnzeichen wird das Leerzeichen benutzt.
Soll ein spezielles Trennzeichen verwendet werden, dann kann dieses bei splitt() als Argument angegeben werden.
Auf diese Art lassen sich z.B. Zeilen aus CSV-Dateien in ihre Elemente Zerlegen, wenn man das gültige Trennzeichen kennt. Da alle Elemente der Liste Texte sind, muss u.U. noch eine Umwandlung in Zahlen – wenn es denn solche sind – erfolgen.

Zeichenkette**.rsplit()**

join(Liste)
erzeugt aus den Elementen der Liste einen verketteten Text
braucht man z.B. Leerzeichen zwischen den Elementen, dann kann man dies so notieren:
" ".join(Liste)
Für die Erzeugung von CSV-Zeilen lässt sich statt dem Leerzeichen natürlich auch ein anderes Trennzeichen verwenden.

8.1.2. besondere Möglichkeiten für Strings in Python

zwei aufeinanderfolgende Literale werden automatisch verknüpft
'Pyt' 'hon' ergibt 'Python'
schöner natürlich mit **+**-Operator: 'Pyt' + 'hon'

gilt nicht für beliebige Kombinationen mit Zeichenketten(-Funktionen)
mit **+**-Operator aber beliebige Zeichenketten-Kombinationen realisierbar

Zahlen müssen ev. vorher mittels **str()**-Funktion in eine Zeichenkette umgewandelt werden

8.2. Datentypen und Typumwandlungen

Irgendwie hat Python fast immer erkannt, mit was für eine Art Daten wir arbeiten. Diese Flexibilität wird von jüngeren Programmierern gelobt und von den älteren / klassischen Programmierern als deutlicher Mangel von Python hervorgehoben.

Grundsätzlich hatten wir es bis hierher mit zwei Datentypen zu tun, die Zahlen und die Texte. Weitere Datentypen sind None als leeres Objekt oder eben "Nichts" und

Bei den Zahlen unterscheiden Informatiker mehrere klassische Zahlen-Arten, die sich zwar an mathematischen Typen orientieren, aber im Wesentlichen unterschiedlich im Prozessor (CPU) verarbeitet werden.
Die einfachste Art Zahlen sind die ordinären bzw. ganzen Zahlen. Sie sind die praktische Darstellung einer Zahl im Dualsystem. Da sie keine Nachkommstellen haben und somit das Komma immer an der rechten Seite haben, spricht man auch von Festkomma-Zahlen. Für das Vorzeichen ist bei einigen (bei Python bei allen) Zahlen-Formaten das höchstwertige Bit reserviert. Diese Art der Zahlen-Darstellung haben wir prinzipiell schon vorgestellt (→ 3.1.2.1. Mathematik für Informatiker – binäres Rechnen). In Python ist das kleinste Festkomma-Zahlenformat der Typ int. Das früher vorhandene und größer als int definierte Typ lon ist in int aufgegangen. Somit gibt es nur noch int, was den Umgang mit Festkommazahlen erleichtert. Die darstellbaren Zahlen sind nicht mehr begrenzt . Für eine Zahl oder eine Variable mit diesem Typ werden also immer viele Bytes vom Haupt-Speicher verbraucht.

Typ	**Beschreibung**	**ev. Grenzen, …**	**Beispiel**	
None	nichts; NULL			
Integer	Ganzzahl	mit führender 0 wird Wert als Oktalzahl, mit führenden 0x als Hexadezimalzahl interpretiert!	x = 3 o = 0127 h = 0x6f4ea	int()
Float	Fließkommazahl Gleitkommazahl		f = 7.85 f1 = 2e6 f2 = -7.25e-3	float()
Complex	komplexe Zahl			complex()
Bool	Wahrheitswert		w = True w = False	bool()
String	Zeichenkette			
List	(veränderbare) Liste (von Elementen) Sequenz			
Tuple	unveränderliche Liste / Sequenz			
Dictionary	Kombinations-Feld Kombinations-Liste Lexikon-Eintrag assoziatives Feld			
Set	(veränderbare) Menge (von Elementen)			
Frozenset	unveränderliche Menge			

Wo sind die Variablen abgespeichert und wieviel Platz (Byte) werden für sie verbraucht?

In Python sind Variablen Referenzen (Verweise / Zeiger) auf bestimmte Speicherzellen. Mittles der id-Anweisung bekommt man die (erste) Speicher-Adresse zurückgeliefert.

Den verbrachten Speicherplatz kann man über die Typ-Bestimmung für die Variable ermitteln. Dazu gibt es die type-Anweisung.

```
>>>
```

eigener Speicher-Verbrauch des Programms:

8.2.1. Zahlen

8.2.1.1. ganze Zahlen

Zahlendarstellung über spezielle Literale

oktale Literale
0**o**724

binäre Literale
0**b**01001

hexadezimale Literale
0**x**A1F31

Umwandlungs-Funktion: **bin()**, **oct()** und **hex()**
arbeiten nur mit int-Zahlen
es handelt sich um Funktionen, die direkt in Python zugänglich sind, ein Modul-Import ist nicht notwendig

8.2.1.2. Fließkommazahlen / Gleitkommzahlen

Zahlen mit Kommastellen nennen die Informatiker auch Fließkomma-Zahlen. Die Zahl besteht dabei immer aus einer vorzeichenbehafteten Mantisse. Die Mantisse hat eine Spanne von 1,0 bis 9,9999… Dazu gehört immer ein Exponent der die zugehörige Zehner-Potenz charakterisiert. Vielleicht kennen Sie die Zahlen-Darstellung als wissenschaftliches Zahlen-Format oder .

$3{,}9745 * 10^{-20}$ → 3.9745e-20

Die Möglichkeit, dass auch nur ein e schon eine – zumindestens theoretisch – mögliche Fließkommazahl ist ($0{,}0 * 10^{0}$), kann in einigen Programmen und / oder Programmiersprachen zu Problemen führen.

In der Mathematik gibt es auch noch eine etwas ungewöhnlich anmutende Zahlen-Menge – die imaginären oder komplexen Zahlen. Sie ergeben sich aus der Lösung des Problems um die Berechnung der Wurzel aus -1. Im Bereich der "normalen" Zahlen (natürliche, ganze, reele, rationale Zahlen) gibt es keine Lösung für die Wurzel aus -1.
Zu diesem Daten-Typ und den zugehörigen Verarbeitungs-Möglichkeiten kommen wir in einem späteren Kapitel genauer. Hier ist erst einmal nur wichtig, dass eine komplexe Zahl in Python vom Typ complex ist und aus zwei Bestandteilen besteht, den reelen und den imaginären Teil. In der Mathematik wird der imginäre Teil mit einem i (imaginäre Einheit) gekennzeichnet, in Python wir dafür das j verwendet.

8.2.1.3. Wahrheitswerte

Zu den Zahlen- oder nummerischen Formaten zählen auch die BOOLEschen Werte für WAHR (TRUE) und FALSCH (FALSE). Sie sind gleichfalls durch 1 bzw. 0 und die Ausdrücke **True** und **False** repräsentiert.
In Python haben alle Werte und Daten-Strukturen einen bestimmten Wahrheitswert. Das ist oft sehr praktisch, kann aber auch schwierig für die Lesbarkeit eines Programm-Codes sein.

Da boolsche Werte zu den nummerischen Datentypen zählen sind neben den typisch boolschen Operatoren auch die arithmetrischen Operatoren zugelassen.
Bei der Verwendung ungewöhnlicher Ausdrücke sollte man ev. eine offensichtlichere Schreibung verwenden oder gut kommentieren.

Operator	Beschreibung / Operation	
a **&** b	bitweise UND	AND
a **|** b	bitweise ODER	OR
a **^** b	bitweise exklusives ODER	XOR
a **>>** n	Bit-Verschiebung um n Stellen nach rechts	entspricht: /2
a **<<** n	Bit-Verschiebung um n Stellen nach links	entspricht: *2
not a	Negation von a	
a **and** b	UND-Verknüpfung	
a **or** b	ODER-Verknüpfung	
a **+** b		
a **-** b		
a ***** b		
a ****** b		
a **==** b	Gleichheit	
a **!=** b	Ungleichheit	

8.2.2. Strings und Co als Datentypen

Die Zeihenketten haben wir schon wegen vieler Besonderheiten und der großen Bedeutung weiter vorne besprochen (→ 8.1. Strings – Zeichenketten).
Hier gehen wir nur noch einmal in Bezug auf Typ-Umwandlungen auf sie ein. Einige Aspekte kommen hier wiederholend vor.

8.2.2.1. einzelne Zeichen

ord()
gibt für ein Zeichen / Charakter den ASCII-Code zurück

chr()
wandelt einen ASCII-Code (Ganzzahl!) in ein Symbol / Charakter um

8.2.2.2. Sequenzen von Zeichen - Zeichenketten

str()
produziert vorrangig für Menschen lesbare Strings

repr()
erstellt Strings, die optimal für den Interpreter sind

float()
wandelt eine Zeichenkette in eine Fließkomma-Zahl um
Achtung! bei Fehler-behafteten Zeichenketten ergibt sich ein Laufzeitfehler
Abfangen von Laufzeitfehlern über Exception's möglich (→ 8.14. Behandlung von Laufzeit-fehlern – Exception's)

int()
wandelt eine Zeichenkette in eine Festkomma-Zahl um
Achtung! bei Fehler-behafteten Zeichenketten ergibt sich ein Laufzeitfehler
Abfangen von Laufzeitfehlern über Exception's möglich (→ 8.14. Behandlung von Laufzeit-fehlern – Exception's)

Aufgaben:

1. ***Schreiben Sie ein Programm, dass einen einzugebenen Text in eine Liste von ASCII-Code's zerlegt und diese Liste dann ausgibt!***
2. ***Erweitern Sie das Programm von der Aufgabe 1 dahingehend, dass die Liste der ASCII-Code's in eine weitere Liste aus Symbolen umgewandelt wird! Lassen Sie diese Liste dann auf dem Bildschirm erscheinen!***
3. ***Im letzten Schritt soll das Programm nochmals um die Rekonstruktion eines String's aus der ASCII-Code-Liste (!!!) erweitert werden!***
4. ***Ein (neues) Programm soll eine einzugebene positive Ganzzahl (z.B.: 134) in "gesprochene" Ziffern-Folgen (z.B.: "eins drei vier") zerlegen!***

für die gehobene Anspruchsebene:

5. ***Erweitern Sie das Programm um die Ausgabe eines Vorzeichens für negative Ganzzahlen!***
6. ***Erstellen Sie ein Programm, dass eine einzugebene Fließkommazahl (z.B.: 183.45) in eine "deutsche", "gesprochene" Ziffernfolge zerlegt! (hier: "eins acht drei Komma vier fünf")!***

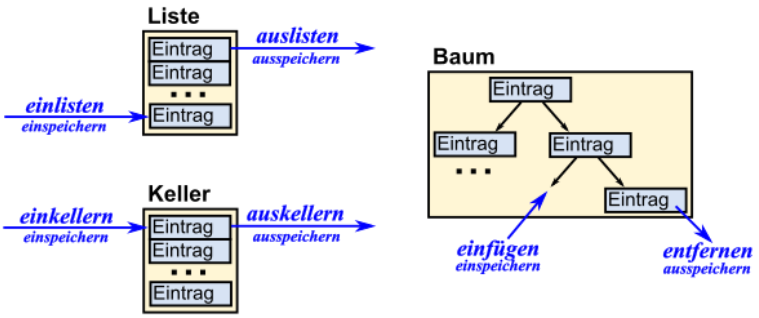
8.2.3. Listen, die I. – einfache Listen

In den vorderen Kapiteln haben wir schon das eine oder andere Mal unterschwellig Listen verwendet, ohne genau auf diese Daten-Struktur einzugehen. Hier sollen nun verschiedene einfache Zugriffs-Möglichkeiten usw. genauer besprochen werden.
Später betrachten wir die Listen auch noch mal aus der Sicht der Objekt-orientierten Programmierung (→ 9.7. Listen, die II. – objektorientierte Listen). Listen sind nämlich auch einfach nur Objekte, für die es dann vorgefertigte Attribute und Methoden gibt. Hier bieten sich dann viele genial-einfache Listen-Nutzungs-Möglichkeiten an, hinter deren Prinzip man aber erst einmal steigen muss. Hier helfen uns die Kenntnisse über die grundsätzliche Art und Weise der Objekt-orientierten Programmierung (→ 8.11. Objekt-orientierten Programmierung).

Definition(en): Datenstruktur
Eine Datenstruktur ist der Informatik eine Vereinbarung zur Organisation und Speicherung von Daten.

wichtige Daten-Strukturen:

- lineare Strukturen:
  - Liste
  - Keller
  - Ring
- verzweigte Strukturen:
  - Baum
  - Netz



ausgewählte (bedeutsame) Datenstrukturen

Speicher-Typ: **FIFO** - First In - First Out, selten auch FCFS (für: First come, first served.)

Daten, die zuerst in die Struktur aufgenommen / gespeichert werden, werden auch zuerst wieder aus ihr entfernt / entnommen

Wer zuerst kommt, mahlt zuerst. First come, first served.

bekannteste und auch häufigste Struktur in informatischen Systemen ist die (Warte-)Schlange
hier auch vielfach Queue oder Pipe genannt

alternativer Speicher-Typ für lineare Daten-Strukturen ist LIFO für "Last IN - First out"
z.B. bei Keller (Stapel-Speicher, Stack) realisiert
im englischen auch LCFS für "Last come - first served."

LIFO und FIFO sind ausschließlich Zeit-bedingt. Prioritäten spielen keine Rolle. Dafür werden dann spezielle Versionen dieser Speicher-Typen realisert.

8.2.3.0. theortische Vorbetrachtungen

8.2.3.0. 1. Listen – eine Form der Datensammlung

Wahrscheinlich sind Listen wohl die älteste Form der strukturierten Daten-Sammlung. Es gibt Höhlenmalerein, in denen die Jagd-Erfolge aufgelistet sind.
Abei auch in unserem heutigen Leben spielen Listen immer noch eine große Rolle. Vielzitierte Beispiele sind:

- Einkaufs-Listen
- Stück-Listen (z.B. am Anfang von Bau-Anleitungen (→ LEGO ®-Bausätze, IKEA ®-Möbel, …)
- Inventar-Listen
- Arbeits-Aufträge (To do-Listen)
- Check-Listen (für Flugzeugstart‘s od.ä.; Reinigungs-Arbeiten, …)
- Vokabel-Listen
- Anrufer-Liste im Telefon
- Mail-Box
- empfangene oder gesendete eMail’s
- Schüler-Liste im Klassenbuch
- Strafakte
- Adressliste / Telefon-Liste
- Messwerte
- Playlist im MP3-Player / Smartphone
- Dateien in einem Ordner
- Inhalts-Verzeichnis
- Handels-Register
- Warte-Liste
- Chroniken
- Wähler-Verzeichnis
- Speisekarte / Menü-Liste
- Robinson-Liste
- Mitglieder-Liste eines Vereins
- Lieferschein
-

Neben den einzelnen Elementen interessieren in Listen oft auch die Anzahl der Einträge. Quantitative Aspekte spielen bei Listen also eine wichtige Rolle. Listen haben einen Inventar-Charakter bezogen auf Objekte, die unter einem Aspekt zusammengestellt wurden.

Definition(en): Listen
Eine Liste ist eine Sammlung von thematisch zusammengehördnen Informationen / Daten in einer sich ständig wiederholenden Form.

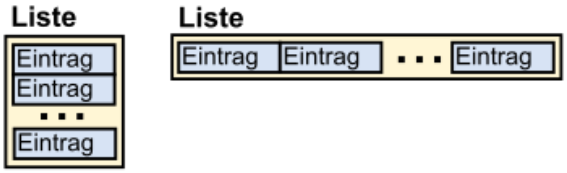
Selbst soetwas wie Spick-Zettel könnte man in die Kategorie Listen einordnen.
Viele der oben erwähnten Anwendungen von Listen gibt es sicher auch schon für unsere modernen Smartphone's als App. Hier ist dann der Bezug zur informatischen Datenstruktur Liste meist besonders leicht herzustellen. Da hier die Daten (z.B. eMail's, Kontakte usw. usf.) als Listen bereitgestellt werden.

Aufgaben:

1. ***Wählen Sie eine Gruppe von 10 App's auf Ihrem Smartphone aus (z.B. die auf einer Seite zusammenstehen) und notieren Sie deren Namen und Zugehörigkeit zu einem App-Typ! Prüfen Sie nun, ob und welche Daten in einer oder mehrerer Listen verwaltet werden!***
2. ***Was sind eigentlich Blacklist, Whitelist und Graylist?***
3.

Listen sind entweder leer oder bestehen aus einem (Kopf-)Element und einem Verweis auf eine (Rest-)Liste
die Rest-Liste kann natürlich auch leer sein

8.2.3.0.2. Daten-Struktur: Liste



Listen sind in Python wohl die Daten-Struktur. Ob wir sie dabei als vertikal oder horizontal angeordnet betrachten, ist für uns völlig egal. Meist wird aber eher eine horizontale betrachtung vorgezogen.
Vielfach sind die Elemente / Einträge in einer Liste gleichartig.
In Python muss dies nicht so sein. Man kann in einer Liste Obejekte ganz unterschiedlicher Typen sammeln. Das können auch wieder Listen sein. Gerade dies macht Listen in Python so unwahrscheinlich flexibel.
Im Vordergrund stehen die folgenden Gründe für den Einsatz von Listen:

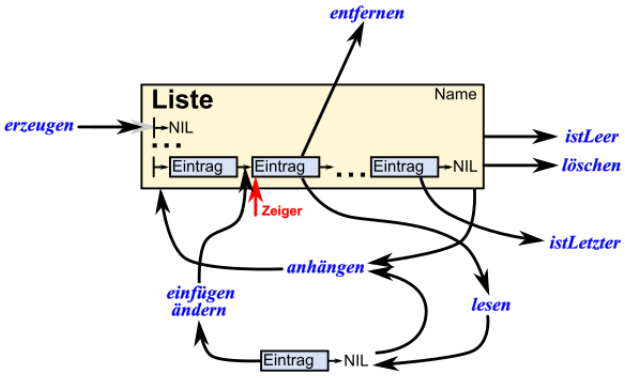
- gleichartige zu bearbeitende Elemente
- einfache Aufzählung von Objekten
- unbestimmte Anzahl von Elementen, Anzahl ist mehr oder weniger stark veränderlich
- Reihen-Folge der Listen-Einträge kann, muss aber keine Rolle spielen

Definition(en): Liste
Im informatischen Sinn versteht man unter einer Liste eine Sammlung von (gleichartig zu bearbeiteten) Elementen in einer Aneinander-Reihung.
Eine Liste ist eine dynamische Datenstruktur von Elementen / Objekten in einer verketteten Form.
Eine Liste ist eine lineare Datenstruktur,

Einlisten – Eintragen eines Elementes / Listen-Eintrag's in eine Liste
Auslisten – Entfernen eines Elementes / Listen-Eintrag's aus einer Liste

Größe der Liste wird maßgeblich vom verfügbaren / bereitgestellten / (dafür) reservierten Speicher bestimmt
die reine Anzahl an Einträgen kann sehr groß werden

Attribute von Listen:
Name
(aktuelle Zeiger-Position)



Operationen / Methoden zu / auf Listen:
initialisieren (init) → erstellen einer Liste
?leer / istLeer (isEmpty) → ist die Liste leer?
einspeichern / anhängen (append) → anhängen eines Element's an eine bestehende Liste

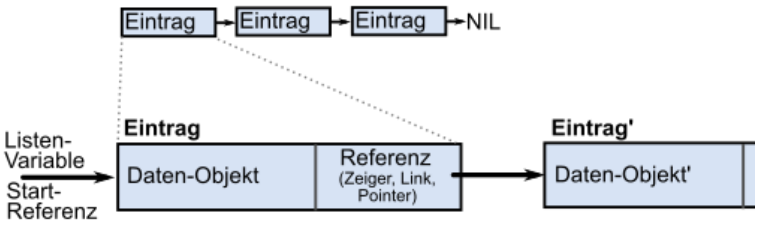
lesen / ausspeichern (read) → Lesen eines Eintrags, ohne ihn zu löschen
geheZumAnfang / (toFirst) → gehe zum ersten bzw. Kopf-Element
tauscheElement / (remove) → ersetze das aktuelle / ausgewählte Listen-Element durch ein anderes
einfügen / (insert) → ein Element an der aktuellen Stelle (Zeiger-Position) einfügen und den Rest der Liste anhängen
vorne einfügen / ()
entfernen / löschen () → entfernen / löschen des aktuellen Element's (Rest-Liste wird an den Vorgänger angehängt)

istEintrag / suche (search /) → ist ein bestimmter Eintrag in der Liste vorhanden?
finde / gibPosition (find / getPosition) → gibt die Listen-Position eines Eintrag's an / zurück
anzahl / länge (length) → gibt die Anzahl an Einträgen in der Liste zurück

Kopf-bezogenes Arbeiten:
insert bzw. push zum Einspeichern (neue Liste ::= neuer Eintrag, alte Liste)
pop(0) zum Ausspeichern (liefert das Kopf-Element zurück und entfernt es vom Kopf der Liste)
Liste wird als Konstrukt aus Kopf und Rest verstanden

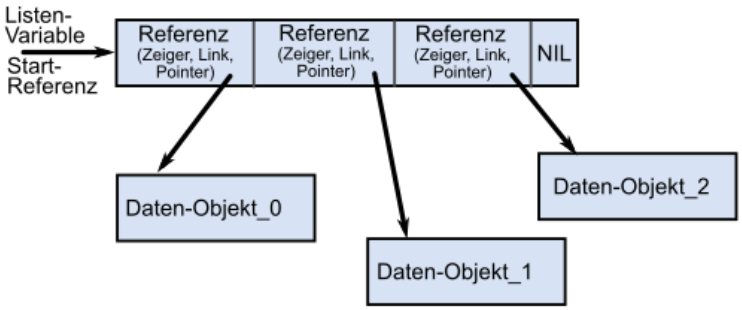
Schwanz- / Ende-bezogenes Arbeiten:
append anhängen eines Eintrags an die Liste (einspeichern)
pop() zum Ausspeichern (liefert das letzte Element zurück und entfernt es vom Ende der Liste) (!!! Fehler bei leerer Liste)
eine Liste wird als Konstrukt aus einer (kleineren) Liste und einem (letzten) Element - dem Ende - verstanden

Vielfach sind Listen als zweigeteilte Objekte in Programmiersprachen angelegt. Neben dem eigentlichen Daten-Objekt enthält ein Eintrag auch noch eine Referenz auf den nächsten Eintrag. Eine Referenz ist praktisch eine Adresse im Hauptspeicher.



Beim letzten Eintrag wird die Referenz auf NIL gesetzt. NIL steht dabei für "Not in List" bzw. "Not in line". Es repräsentiert einen Null-Wert.

In Python wurde eine abweichende Organisations-Form gewählt. Die Liste besteht nur aus Referenzen. Diese verweisen auf irgendwo gespeicherte Daten-Objekte.
Ganz genau informatisch betrachtet sind Listen in Python also Felder (Array's) von Zeigern (Pointern).
Der NIL-Wert kann auch als Speicher-Adresse verstanden werden.



In dieser ist praktisch - wie in einem Mülleimer- ganz viel Platz.

in Python haben wir keinen Zugriff auf den Link-Teil eines Eintrages

lediglich das soundsovielte Element einer Liste kann angesteuert werden, was dem temporären Setzen des Zeiger's (auf ebendiesen Eintrag) entspricht

Listen lassen sich mithilfe von verschiedenen Schleifen durchlaufen
z.B. mit Zählschleife bei bekannter Anzahl von Einträgen (Index-Zugriffe)
z.B. mit einer bedingten Schleife und einer Erkennung des letzten Eintrag's (→ NIL)
z.B. mit Sammlungs-bedingten Schleifen / (verdeckten) Interatoren

praktische Nutzungen / Abwandlungen in …:

- Warteschlange
  - Drucker-Warteschlange
  - Übertragungs-Puffer (z.B. TCP/IP; Tastatur-Eingaben)
  -
-

Operationen auf Listen (Attribut-Schreibweise)

Bezeichnung	Operation	Resultat
	Liste **= []**	erzeugen einer leeren Liste
	Liste**[i] =** Wert	i. Eintrag in der Liste (er)setzen / einspeichern
	Liste**[i : j] =** Teilliste	in der Liste wird der Abschnitt von Eintrag i bis j durch die angegebene Teilliste ersetzt
	del Liste**[i : j]**	löst in der Liste den Abschnitt von Eintrag i bis j (→ Verkürzung der Liste) äquivalent zu: Liste[i : j] = []
	Liste**.append(**Wert**)**	hängt Wert als neuen Eintrag an die Liste an äquivalent zu: Liste[len(Liste) : len(Liste)] = [Wert]
	Liste**.extend(**Wert**)**	äquivalent zu: Liste[len(Liste) : len(Liste)] = Wert
	Liste**.count(**Wert**)**	liefert die Anzahl der Einträge zurück, die Wert entsprechen return: Anzahl der i mit Liste[i] == Wert
	Liste**.index(**Wert**)**	liefert den ersten Index zurück, bei dem der Eintrag dem Wert entspricht return: Erstes i mit Liste[i] == Wert
	Liste**.insert(i,** Wert**)**	fügt einen neuen Eintrag mit dem Wert an Position i ein äquivalent zu: Liste[i : i] = Wert , wenn i >= 0
	Liste**.remove(**Wert**)**	entfernt das erste Auftreten eines Wertes ohne Kenntnis des Index äquivalent zu: del Liste[Liste.index(Wert)]
	Liste**.peek()**	liest den obersten Wert der Liste äquivalent zu: Liste[0]
	Liste**.pop()**	liest und entfernt das oberste Objekt aus der Liste äquivalent zu: del Liste[0]
	Liste**.reverse()**	Liste wird intern (in place) umgedreht

		(originale Reihenfolge durch erneutes reverse() wiederherstellbar)
	Liste**.sort()**	Liste wird intern (in place) sortiert (originale Reihenfolge der Einträge geht verloren!)
	Liste**.sort(**VergleichsFunktion**)**	return -1, 0, +1 ; wenn x<y, x=y, x>y
	Liste**.( )**	

Operationen auf Listen (Präfix-Schreibweise)

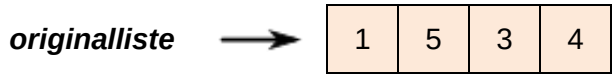
Bezeichnung	Operation	Resultat
	map(Funktion, Liste**)**	neue (temporäre) Liste: [Funktion(Liste[0], Funktion(Liste[1], …, Funktion(Liste[n]]
	filter(Bedingung, Liste**)**	neue (temporäre) Liste mit allen Einträgen aus der Liste, die die Bedingung erfüllen
	Liste1 **+** Liste2	neue (temporäre) Liste mit allen Einträgen aus Liste 1 und 2
	reduce(Funktion, Liste**)**	
	reduce(Funktion, Liste, init**)**	
	zip(Liste1, Liste2**)**	erzeugen einer (temporären) Paar-Liste mit zusammengehörenden Tupeln aus der Einträgen der beiden Listen [[Liste1[0], Liste2[0], [Liste1[1], Liste2[1], …, [Liste1[n], Liste2[n]]

8.2.3.1. Definition und Zuweisung von Listen in Python

Definieren wir uns zuerst einmal eine Liste mit dem Namen ***originalliste***. Sie soll bei den nächsten Versuchen immer wieder die Ausgangsbasis sein.

```
>>> originalliste=[1,5,3,4]
>>> print(originalliste)
[1, 5, 3, 4]
```

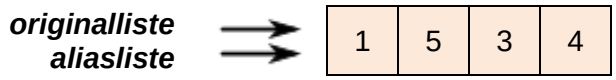
Der Name originalliste ist im Prinzip ein Verweis (bzw. ein Zeiger) auf die Speicher-Zellen, wo sich die Daten befinden.



Die nachfolgende Shell-Eingabe liest sich so, als würde man eine neue Liste – sozusagen eine Kopie vom Original erstellen.

```
>>> aliasliste=originalliste
>>> print(aliasliste)
[1, 5, 3, 4]
>>>
```

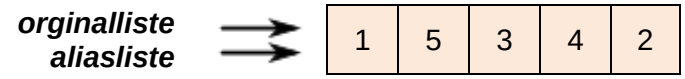
Praktisch wird aber nur ein zweiter Verweis auf die Originalliste gelegt.
Das merken wir spätestens, wenn wir eine der Listen verändern.



Einen Nebenverweis nennt man einen Alias bzw. einen Aliasnamen.

```
>>> aliasliste=aliasliste+[2]
>>> print(aliasliste)
[1, 5, 3, 4, 2]
>>> print(originalliste)
[1, 5, 3, 4, 2]
>>>
```

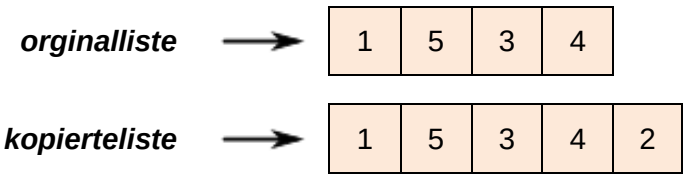
Beide Listen sind länger geworden. Ganz exakt müsste man eigentlich sagen, die (eine) Liste ist länger geworden



Das Aliasieren erzeugt nur eine sogenannte **flache Kopie** der Liste. Eine **tiefe Kopie** – wir würden wohl eher **echte Kopie** sagen – erhält man z.B. durch das Slicing (→ 8.2.3.5. Listen-Abschnitte (Slicing)).

```
>>> kopierteliste=originalliste[:]
>>> kopierteliste=kopierteliste+[7]
>>> print(originalliste)
[1, 5, 3, 4]
>>> print(kopierteliste)
[1, 5, 3, 4, 7]
>>>
```

Die Original-Liste bleibt bei Operationen auf die kopierte Liste unverändert.
Das Löschen der Original-Liste hat auch keine Auswirkungen, da beide Listen völlig eigenständig sind.



So zumindestens lautet die Theorie bzw. lauten die Aussagen vieler Buchautoren und der Python-Guru's aus dem Internet.
In meinen Test's mit einem Python-System (V. 3.4.3; 32 bit) sah das alles anders aus. Hier wird ganz offensichtlich eine echte (**tiefe**) **Kopie** der Liste erzeugt und auch durchgehend verwaltet. Beide Listen (originalliste und aliasliste) lassen sich unabhängig manipulieren und löschen.
Die zwei nachfolgenden Shell-Dialoge zeigen dieses ganz klar.

```
>>> originalliste=[1,5,3,4]
>>> print(originalliste)
[1, 5, 3, 4]
>>> aliasliste=originalliste
>>> print(originalliste)
[1, 5, 3, 4]
>>> print(aliasliste)
[1, 5, 3, 4]
>>> aliasliste=aliasliste+[3]
>>> print(originalliste)
[1, 5, 3, 4]
>>> print(aliasliste)
[1, 5, 3, 4, 3]
>>>
```

Erweiterung der Aliasliste

… und es wurde auch nur die Aliasliste erweitert

```
>>> originalliste=[1,5,3,4]
>>> print(originalliste)
[1, 5, 3, 4]
>>> aliasliste=originalliste
>>> print(aliasliste)
[1, 5, 3, 4]
>>> originalliste=originalliste+[3]
>>> print(originalliste)
[1, 5, 3, 4, 3]
>>> print(aliasliste)
[1, 5, 3, 4]
>>> del originalliste[2]
>>> print(originalliste)
[1, 5, 4, 3]
>>> print(aliasliste)
[1, 5, 3, 4]
>>> del originalliste
>>> print(originalliste)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(originalliste)
NameError: name 'originalliste' is not defined
>>> print(aliasliste)
[1, 5, 3, 4]
>>>
```

Erweiterung der Originalliste ...
… und auch nur diese wurde erweitert

Löschen des 3. Wertes in der Originalliste …
was offensichtlich klappt
… aber die Aliasliste davon unberührt lässt.
… genau so wie das Löschen der Originalliste

… die wirklich weg ist …

… aber die Aliasliste noch völlig im "Original"-Zustand vorhanden ist

Erst einmal bleibt uns jeweils wohl nur der Test am eigenem System!

Kehren wir zur offiziellen Version zurück und nehmen mein System als Ausrutscher!

Wie bekommen wir nun aber eine echte / eigenständige Kopie einer Liste? Eine kleine Variante haben wir oben schon gezeigt. Eine weitere Möglichkeit sind fertige Objekt-orientierte Funktionen zu Listen. Diese besprechen wir etwas später (→ 9.8. Listen, die II. – objektorientierte Listen). Eine Möglichkeit werden wir gleich bei der Listen-Bearbeitung, eine weitere nochmals genauer bei den Listen-Abschnitten (Slicing), besprechen. Python bietet mehrere (gut und weniger gut leserliche) Möglichkeiten – der Programmierer hat hier die freie Wahl. Man nennt dies auch Klonen von Listen.

8.2.3.2. Listen-Operationen (Built-in-Operatoren)

Für Listen funktionieren einige "Rechen"-Operationen im intuitiven Sinne. So ergibt sich bei Verwendung des Multiplikations-Zeichens ***** eine x-mal erweiterte / verlängerte Liste.

```
>>> liste=[2]
>>> liste=liste*5
>>> print(liste)
[2, 2, 2, 2, 2]
>>>
```

Mit **+** lassen sich Listen addieren, aneinander anhängen bzw. verbinden.

```
>>> liste=liste + liste
>>> print(liste)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
>>>
```

Nicht wirklich eine Rechen-Operation steckt hinter dem **in**-Operator. Mit ihm können wir schnell testen, ob Etwas ein Element einer Liste ist. Als Ergebnis erhält man entweder **1** – für ist ***in*** der Liste – oder eben **0** (nicht ***in*** der Liste).
Der in-Operator lässt sich mit **not** in Verbindung nutzen

```
>>> liste=['gelb','grün','rot','weiß','gelb','braun','blau']
>>> print(liste)
['gelb', 'grün', 'rot', 'weiß', 'gelb', 'braun', 'blau']
>>> 'grün' in liste
True
>>> 'schwarz' in liste
False
>>> 2 in liste
False
>>>
```

Der in-Operator lässt sich mit **not** in Verbindung nutzen

```
>>> 'blau' not in liste
False
>>> 3 not in liste
True
>>>
```

Sehr praktisch sind die Funktionen **min()** und **max()**. Mit ihnen kann man ohne (eigenen) Programmieraufwand das kleinste bzw. größte Listenelement heraussuchen. Bei Texten gilt die alphabetische Ordnung, bei Zahlen die normale Rangfolge.

```
>>> liste=['rot', 'grün', 'gelb']
>>> print(min(liste1))
gelb
>>> print(max(liste1))
rot
>>>
```

Bei gemischten Listen gibt es eine Typ-Fehlermeldung.
An dieser Stelle hat die nachfolgende Funktion eigentlich keine Bedeutung. Ich erwähne sie hier wegen der Vollständigkeit und einem vielleicht benötigtem Hinweis. Mit der Funktion list() lassen sich Tupel (→) in Listen umwandeln. Das spielt immer dann eine Rolle, wenn die Daten in Tupeln vorliegen und bearbeitet werden sollen, was bei Tupeln eigentlich nicht geht.

```
>>> tupel1=(23,'gelb',['rot',2])
>>> print(tupel1)
(23, 'gelb', ['rot', 2])
>>> tupel1[0]=24
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    tupel1[0]=24
TypeError: 'tuple' object does not support item assignment
>>> liste1=list(tupel1)
>>> print(liste1)
[23, 'gelb', ['rot', 2]]
>>> liste1[0]=24
>>> print(liste1)
[24, 'gelb', ['rot', 2]]
>>>
```

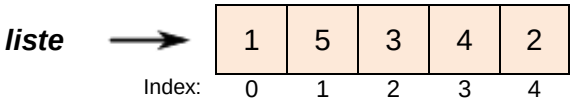
cmp(liste1, liste2**)**
vergleicht Listen (wahrscheinlich nur bis Python2)

Anhängen eines Elemntes bzw. einer Liste an eine andere ist auch über den Plus-Operator (**+**) möglich. Für interierende Aufgaben ist auch **+=** möglich. Bei der Verwendung sollte man aber beachten, dass diese Opratoren über 1'000x langsamer sind als z.B. die append()-Funktion.

8.2.3.3. Listen-Indexierung

Auf die einzelnen Elemente einer Liste kann man natürlich auch direkt zugreifen. Dazu muss man aber wissen, dass die Liste mit mit 0 beginnend gezählt wird. Die Zählung wird deshalb auch Indexierung genannt und der Zugriffswert Index. Das erste Listen-Element hat also den Index 0.

Der Zugriff wird in Python denkbar einfach ermöglicht. Wir müssen den Index nur an den Listennamen in eckigen Klammern angeben. Der Index kann beliebig berechnet werden.



Wichtig ist dabei nur, dass das Ergebnis immer ganzzahlig sein muss.

```
>>> liste=[1,5,3,4,2]
>>> print(liste)
[1, 5, 3, 4, 2]
>>> print(liste[2])
3
>>> print(liste[2*2-1])
4
>>> print(liste[5/2])
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    print(liste[5/2])
TypeError: list indices must be integers, not float
>>>
```

Aber das erscheint wohl logisch, denn z.B. ein 2,5tes Element gibt es eben nicht. Der Zugriff auf einen fehlerhaften oder zu großen Index ergibt eine Fehler-Meldung:

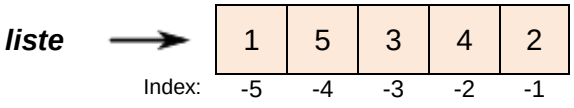
```
>>> print(liste[13])
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print(liste[13])
IndexError: list index out of range
>>>
```

Um hier fehlerfrei agieren zu können müssen wir natürlich wissen, wieviele Elemente in der Liste sind. Dafür gibt es die Funktion **len()**, die genau diese Aufgabe erfüllt.

```
>>> len(liste)
5
>>>
```

Wird ein negativer Index angegeben, dann wird in der Liste zurückgezählt:

Natürlich darf auch hier der Index-Wert nicht größer (gemeint ist natürlich: kleiner) als die Listenlänge werden.



Das Löschen einer ganzen Liste oder eines Elementes aus der Liste ist mit **del** möglich.

```
>>> originalliste=[1,5,3,4]
>>> del originalliste[2]
>>> print(originalliste)
[1, 5, 4]
>>>
```

where um Indizes anhand einer Bedingung auszuwählen

```
>>> i=0
>>> while i<len(farbenliste):
      print(farbenliste[i])
      i+=1


gelb
grün
rot
blau
braun
>>>
```

??? feld=array([[0,1,2],[3,4,5]])
??? feld[where(feld > 3)] where(feld > 3)

An die reine Indexierung schließt das sogenannte Slicing von Listen (→ 8.2.3.5. Listen-Abschnitte (Slicing)) an.

Zuweisen von Werten aus einer Liste auf Positionen, die in einer anderen Liste verwaltet werden

```
liste=[1,2,3,4,5,6,7,8,9,10]
indizies=[3,4,6]
neueWerte=[100,200,300]
```

Interieren über die Tupel (Paare) von Index und Wert

```
for i in zip(indizies, neueWerte):
    liste[i[0]]=i[1]

print(liste)
```

2. Möglichkeit
mit der eingebauten Funktion (Build In Funktion) enumerate über Listen interieren

```
for index,wert in enumerate(indizies):
    liste[wert]=neueWerte[index]

print(liste)
```

3. Möglichkeit
die Liste in ein (NumPy-)Array umwandeln und dann eine Zuweisung af der Basis der Indizies durchführen

```
import numpy as np

feld=np.array(liste)
feld[indizies]=neueWert
liste=list(feld)

print(liste)
```

zur Nutzung der Bibliothek NumPy gibt es weitere Informationen in einem gesonderten Kapitel (→ 8.14.3. Python numerisch, Python für Big Data)

8.2.3.4. Listen-Bearbeitung

Das Gute zuerst, Listen lassen sich bearbeiten. Das überrascht vielleicht etwas, nachdem wir bei den Strings (→ 8.2. Strings – Zeichenketten) und den Feldern (→ 6.6. Vektoren, Felder und Tabellen) dahingehend enttäuscht worden sind. Listen sind konzeptionell einfach die priviligierten Daten-Objekt in Python.

An dieser Stelle sei kurz auf eine den Listen sehr ähnliche Daten-Struktur hingewiesen – die Tupel. In Abschnitt 9.1. Tupel werden sie noch genauer besprochen. Prinzipiell ähneln sie den Listen, man kann fast alle Operationen usw. auf sie anwenden. Allerdings sind Tupel feststehende Objekte. Nach ihrer Erzeugung lassen sie sich nicht mehr ändern!

Am Einfachsten lassen sich Listen mittels Schleifen bearbeiten. Dabei bieten sich die **for**-Schleifen sofort an.

Wollen wir die Listenelemente einzeln ausgeben, dann reicht schon der folgende – leicht verständliche – Konstrukt:

```
>>> farbenliste=['gelb','grün','rot','blau','braun']
>>> for farbe in farbenliste:
      print(farbe)


gelb
grün
rot
blau
braun
>>>
```

Es geht natürlich genauso mit einer **while**-Schleife, aber hier müssen wir die Indizierung selbst verwalten.
Für die gleiche Beispiel-Aufgabe – also die Element-weise einer Liste – würde der Quelltext dann z.B. so aussehen:

```
>>> i=0
>>> while i<len(farbenliste):
      print(farbenliste[i])
      i+=1


gelb
grün
rot
blau
braun
>>>
```

Natürlich hätte man hier auch z.B. die Lauf-Variable farbe benutzen können, wie in der for-Schleife z.B. ein einfaches i.
Insgesamt ist der Programmier-Aufwand etwas größer. Welchen Schleifen-Konstrukt man dann in seinem Programm später verwendet, entscheidet sich wahrscheinlich neben der Vorliebe auch aus praktischen Abwägungen.
Um nun eine Liste zu klonen – also eine echte Kopie zu erzeugen – bietet sich der folgende Algorithmus an:

```
>>> neueListe=[]
>>> for f in farbenliste:
      neueListe=neueListe+[f]


>>> print(neueListe)
['gelb', 'grün', 'rot', 'blau', 'braun']
>>>
```

Es wird praktisch jedes einzelne Element aus der Originalliste ausgelesen und in die Kopie geschrieben.
Nachteilig ist es in vielen Programmiersprachen, wenn man lange Listen oder Felder (→) mittels Schleifen-Konstrukt durchlaufen muss, um z.B. eine Summe zu berechnen oder das Minimum herauszusuchen. Da bietet uns Python die map-Funktion, um beliebige mathematische Funktionen auf die Elemente einer Liste anzuwenden.

map(funktion, liste**)**

Die Funktion kann eine von Python vordefinierte oder eine selbst-definierte Funktion sein. Im map-Funktions-Aufruf wird die Funktion nur mit ihrem Namen (also ohne Klammern und Argumenten) notiert. Die Liste muss eine Sequenz sein, auf deren Elemente die Funktion angewendet werden kann.
Definieren wir zuerst eine Funktion, die den übergebenen Wert mit 10 multipliziert und dann noch inkrementiert. Die Operationen sind mit Absicht in einzelnen Schritten aufgeschrieben worden. Natürlich können sie in einer Zeile oder gar hinter **return** zusammengefasst werden.

```
>>> def mult10add1(x):
        x*=10
        x+=1
        return x
>>>
```

Die klassische Struktur einer Element-weisen Anwendung einer Funktion kann z.B. wie nebenstehend aussehen.
Wir definieren eine nutzbare Funktion. Alternativ kann auch jede interne Funktion genutzt werden.
In einer Schleife wird diese Funktion dann Element-weise angewendet und das Ergebnis an die ursprüngliche Listen-Position gespeichert.

```
def mult10add1(x):
    x*=10
    x+=1
    return x
…
liste=[2,3,5,8,11]
for i in range(len(liste)):
    liste[i]=mult10add1(liste[i])
print(liste)
```

```
[21, 31, 51, 81, 111]
```

```
liste = map(mult10add1, liste)
print(liste)
```

Der **map**-Operator ergibt im Allgemeinen kompaktere Quell-Code's. Auch die Ausführung ist effektiver. Aus meiner Sicht sind die Quell-Texte aber nicht gut lesbar, da geschachtelte Strukturen entstehen, die immer wieder mit zurück-denken oder eine Ebene höher-denken zu tun haben. Map-Konstrukte sollten immer gut kommentiert werden! Sie sind etwas für fortgeschrittene Programmierer / Programmier-Team's. Für kompexere Funktionen können vor allem einzeilig notierte map-Funktionen praktisch unlesbar werden und sind deshalb unbedingt zu vermeiden.

List-Comprehension

In der letzten Zeit werden auch sogenannte List-Comprehension's immer mehr in Programmiersprachen umgesetzt. Das gilt auch für Python.
Comprehension's kann man als Bedeutungs- oder Anwendungs-Objekte oder -Strukturen verstehen. Ziel ist eine bessere Lesbarkeit von Quelltexten, verbunden mit einer hohen Kompaktheit des Quelltextes.
Soll z.B. aus einer Daten-Liste eine Liste mit Quadraten erzeugt werden, dann würde man eine Sammlungs-orientierte FOR-Schleife benutzen. Dies ist im oberen Quellcode-Block notiert.
Eine Comprehension-Struktur könnte z.B. so aussehen, wie es im 2. Block zu sehen ist.
Die Struktur besteht aus der Listen-Beginn-Klammer ([) gefolgt von der Operation (auch Expression genannt) und der FOR-Schleife. Die Expression ist in Beispiel die Multiplikation von Wert mit sich selbst. Das Ende wird durch die schließende Listen-Klammer (])gekennzeichnet.

```
daten=[1,2,3,4,5]


quadrate=[]
for wert in daten:
    quadrate.append(wert*wert)
```

```
quadrate=[wert*wert for wert in daten]
```

```
quadrate=[wert*wert for wert in daten ↵
          if wert > 0]
```

```
quadrate=[wert*wert if wert > 0 else 0 ↵
          for wert in daten]
```

Die Ausführung der Expression / Operation lässt sich noch durch Bedingungen steuern. Eine ausschließliche IF-Bedingung (einfache Bedingung) wird hinter den Schleifen-Konstrukt geschrieben. Der 3. Quellcode-Block zeigt ein Beispiel, dass hier zum gleichen Ergebnis führt, wie die beiden oberen Quellcode-Abschnitte.
Benötigt man auch einen ELSE-Zweig, dann folgt der angepasste IF-ELSE-Konstrukt direkt hinter der Expression und noch vor dem Schleifen-Teil.

Mit dem List Comprehension lassen sich auch Listen filtern.

```
[wert for wert in liste if wert>20]
```

```
[wert for wert in liste if (wert>20) and (wert<100)]
```

```
[True if wert==1 else False for wert in daten]
```

8.2.3.5. Listen-Abschnitte (Slicing)

Der Zugriff auf Abschnitte einer Liste wird durch die sogenannte Slice-Notation (Doppelpunkt-Notation) erheblich erleichtert. Für Python-Einsteiger oder Umsteiger aus anderen "normalen" Programmiersprachen werden diese Konstrukte aber erst einmal schwer zu verstehen sein. Zur Verdeutlichung nehmen wir eine etwas längere Liste mit etwas größeren Zahlen:

liste →	111	222	333	444	555	666	777	888	999
Index:	0	1	2	3	4	5	6	7	8
Item:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Benötigt man nur die Liste bis zu einem bestimmten Index, dann wird der Abschnitt mit [: endeindex] notiert.

```
>>> liste=[111,222,333,444,555,666,777,888,999]
>>> sliceliste=liste[:3]
>>> print(sliceliste)
[111, 222, 333]
>>>
```

Es sind mit **[:3]** also die ersten 3 Listen-Elemente (Items) gemeint, der Doppelpunkt steht also hier für "bis" zum dritten (3.) Element. Da die Indizierung bei Null startet sind es also die Index-Elemente 0, 1 und 2.

	111	222	333	444	555	666	777	888	999
Index:	0	1	2	**3**	4	5	6	7	8
Item:	**1**	**2**	**3**	4	5	6	7	8	9

Wird dagegen nur ein Index vor dem Doppelpunkt (Slice) eingegeben, dann meint man den Abschnitt nach diesem indizierten Element bis zum Ende der Liste.

```
>>> sliceliste=liste[5:]
>>> print(sliceliste)
[666, 777, 888, 999]
>>>
```

Mittels **[5:]** wird also es sind die Listen-Elemente nach Item 5 (bzw. ab Index = 5) bearbeitet.

liste →	111	222	333	444	555	666	777	888	999
Index:	0	1	2	3	4	**5**	6	7	8
Item:	1	2	3	4	**5**	6	7	8	9

Natürlich dürfen zur Auswahl eines Mittelstücks aus einer Liste auch vordere und hintere Grenze angegeben werden.

```
>>> print(liste[2:7])
[333, 444, 555, 666, 777]
>>>
```

Mit **[2:7]** meint man dann den Abschnitt ab Index = 2 bis an den 7. Index ran.
es sind die Elemente nach dem 3. (also ab dem zweiten (2.)) bis zum 5. gemeint

liste →	111	222	333	444	555	666	777	888	999
Index:	0	1	**2**	3	4	5	6	**7**	8
Item:	1	**2**	3	4	5	6	**7**	8	9

Somit gilt also allgemein: abschnitt = liste[anfangsindex : endeindex].
Interessanterweise funktioniert das Slicen auch zum Einfügen eines Listen-Abschnitts:
Die Notierung wäre also: liste[anfangsindex : endeindex] = abschnitt

```
>>> print(sliceliste)
[666, 777, 888, 999]
>>> liste[6:7]=sliceliste
>>> print(liste)
[111, 222, 333, 444, 555, 666, 666, 777, 888, 999, 888, 999]
>>>
```

Die eingeslicte Liste wird zuerst noch einmal angezeigt (geprintet) und ist dann in der Ergebnis-Liste farblich unterlegt.

8.2.3.6. Listen-Erzeugung – fast automatisch

mit **range()** werden Listen automatisch erzeugt

range(ende**)**
erzeugt Liste von 0 bis (ausschließlich) ende

range(anfang, ende**)**
erzeugt Liste von anfang bis (ausschließlich) ende

range(anfang, ende, schrittweite**)**
erzeugt Liste von anfang bis (ausschließlich) ende mit der schrittweite (also: anfang + n * schrittweite)

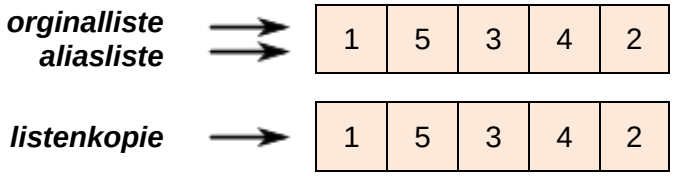
Listen können auch wieder Listen enthalten (Verschachtelung, Nesting)
so lassen sich Matrizen darstellen und bearbeiten. da das aber eher für mahematisch Fortgeschrittene interessant wird, folgen dazu in Kapitel → 8.13.2. Matrizen (Matrixes) mehr Informationen

Zugriff für aneinander-gereihte Index-Operatoren
Hierbei ist besonders auf die Gültigkeit der Indexes zu achten

Eine andere Möglichkeit zum Listen-Klonen (Kopieren einer Liste) ergibt sich aus der Slice-Notierung. Die neue Liste soll den Namen ***listenkopie*** bekommen.

```
>>> listenkopie=originalliste[:]
>>> print(listenkopie)
[1, 5, 3, 4, 2]
>>>
```

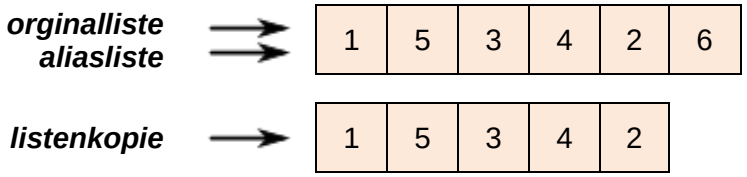
Der Doppelpunkt in der Slice-Notierung zieht eine Grenze. Da aber weder davor eine Anzahl Elemente, noch danach eine Anzahl angegeben wurde, handelt es sich um die gesamte Liste.



Zum Überprüfen, dass es sich bei der Kopie wirklich um einen neue / eigenständige Liste handelt, verändern wir sie durch Hinzufügen eines weiteren Elementes:

```
>>> aliasliste=aliasliste+[6]
>>> print(aliasliste)
[1, 5, 3, 4, 2, 6]
>>> print(listenkopie)
[1, 5, 3, 4, 2]
>>>
```

Die Situation im Speicher kann man sich etwa so vorstellen:



.

8.2.3.7. Listen - extravagant

Der Umgang mit Listen hält noch weitere Besonderheiten / Überraschungen bereit.

erweitertes Listen-Generieren

liste = [x for x in range(20) if x % 2]
erzeugt eine Liste der ungeraden Zahlen (von 0) bis an 20 ran

liste = [(x,y) for x in range(10) if not x % 3 for y in range(6) if y % 2]
erzeugt eine Liste aus Tupel, bei denen x die durch 3 teilbaren Zahlen (von 0) bis an 10 ran und y die ungeraden Zahlen (von 0) bis an 6 ran verwendet werden

gemeinsame Elemente zweier Liste in eine neue Liste
liste1=[1,2,3,4]
liste2=[3,4,5,6]
liste=[i for i in liste1 if i in list2]

aus zwei Datenlisten eine Liste aus Tupeln zusammenstellen
liste1=[1,2,3,4]
liste2=['a','b','c','d']
tupelliste=[(i,j) for i in liste1 for j in liste2]

erweitertes Slicing

text = 'abcdefg'
print(text[1:6:2]) ➔ 'bdf'
vom ersten bis zum sechsten Element (Achtung es geht immer noch bei 0 los!) jedes zweite Element

print(text[::-1]) ➔ 'gfedcba'

print(range(10)[::2]) ➔ [0, 2, 4, 6, 8] (entspricht: range(0,10,2) (besser verständlich)
print(range(10)[::-1]) ➔ [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (entspricht: range(9,-1,-1))

Aufgaben:

x. Die chemischen Elemente lassen sich in verschiedene Gruppen einteilen. Dazu gehören die Hauptgruppen und die Metalle und Nichtmetalle. Schreiben Sie ein Programm, dass ein einzugebenes chemisches Symbol – z.B. Na – einer Hauptgruppe (mit Namen) zuordnet!
Erweitern Sie das Programm dann um die Zuordnung in die Perioden sowie zu den Metallen, Nichtmetallen bzw. Halbmetallen!
Zusatz:
Unterscheiden Sie die Elemente auch nach ihrem Aggregatzustand! Versuchen Sie möglichst kleine Listen (Datenbasen) zu verwenden!

Gruppe	Vertreter
I. Hauptgruppe, Alkalimetalle	H, Li, Na, K, Rb, Cs, Fr
II. Hauptgruppe, Erdalkalimetalle	Be, Mg, Ca, Sr, Ba, Ra
III. Hauptgruppe, Bor-Gruppe, Erdmetalle	B, Al, Ga, In, Tl
IV. Hauptgruppe, Cohlenstoff-(Silicium-)Gruppe, Tetrele	C, Si, Ge, Sn, Pb
V. Hauptgruppe, Stickstoff-(Phosphor-)Gruppe, Pnictogene	N, P, As, Sb, Bi
VI. Hauptgruppe, Chalkogene, Erzbildner, Sauerstoff-Gruppe	O, S, Se, Te, Po
VII. Hauptgruppe, Halogene, Fluor-Gruppe, Salzbildner	F, Cl, Br, I, At
VIII. Hauptgruppe, Edelgase, Helium-Gruppe	He, Ne, Ar, Kr, Xe, Rn

kleine Zusammenfassung (Indexierung / Slicing)

allgemeine Struktur	**Funktion / Leistung** ***Umschreibung***	**Beispiel-Liste: liste=[1,2,3,4,5,10,11,12,13,14,15]**	
		Beispiel	**Ergebnis**
liste[i]	liefert den Eintrag von Index ***i*** **(Achtung! Zählung / Index beginnt bei 0)** *Welcher Wert steht an Index-Position i?*	liste[2] liste[0] liste[6]	3 1 11
liste[-1]	liefert das letzte Elem. zurück *Welcher Wert steht an der letzten Index-Position?* *Welcher Wert steht an 1. von hinten gezählten Index-Position?* **(Achtung! Zählung / Index beginnt hier mit -1, da es ein -0 natürlich nicht gibt)**	liste[-1]	15
liste[-i]	liefert den ***i***-ten Eintrag ***von hinten*** *Welcher Wert steht an der zurück gezählten Index-Position i?*	liste[-3] liste[-10]	13 2
liste[:]	liefert alle Elemente (ohne Begrenzungen) einer Liste als neue Liste *Welche Werte stehen in der Liste (ohne Begrenzungen durch Indizies)?*	liste[:]	[1,2,3,4,5,10,11,12,13,14,15]
liste[nach:]	liefert die Elemente ***nach*** Position bzw. ab Index ***nach*** als neue Liste *Welche Werte folgen ab der Index-Position?*	liste[2:] liste[6:] liste[0:]	[3,4,5,10,11,12,13,14,15] [12,13,14,15] [1,2,3,4,5,10,11,12,13,14,15]
liste[:bis]	liefert die Elemente ***bis*** Position bzw. kleiner dem Index ***bis*** als neue Liste *Welche Werte stehen vor der Index-Position?*	liste[:2] liste[:7] liste[:-1]	[1,2] [1,2,3,4,5,10,11,12] [1,2,3,4,5,10,11,12,13,14]
liste[nach:bis]	liefert als neue Liste die Elemente ***nach*** Position bzw. ab Index ***nach*** und ***bis*** Position bzw. kleiner dem Index ***bis*** *Welcher Wert stehen zwischen den Index-Positionen?*	liste[1:4] liste[4:8] list[3:-1]	[2,3,4] [5,10,11,12,13] [4,5,10,11,12,13,14]
liste[::sprung]	liefert eine neue Liste der Einträge einer originalen Liste, die nacheinander mit einem ***sprung*** erreicht werden (beginnend mit Index 0) *Welcher Wert steht an Index-Position i?*	liste[::3] liste[::-2] liste[::0]	[1,4,12,15] [15,13,11,4,2] → Fehler
liste[nach::]	wie liste[nach:] bzw. liste[nach::sprung]	liste[2::]	[3,4,5,10,11,12,13,14,15]
liste[nach::sprung]	liefert die Einträge, die ***nach*** dann nacheinander mit einem ***sprung*** erreicht werden *Welche Werte folgen ab der Index-Position in bestimmten Schritten?*	liste[4::2] liste[2::4] liste[0::1]	[5,12,14] [3,12] [1,2,3,4,5,10,11,12,13,14,15]

allgemeine Struktur	Funktion / Leistung *Umschreibung*	Beispiel-Liste: listeA=[1,2,3,4,5,10,11,12,13,14,15]	
		Beispiel	**Ergebnis**
listeA[nach:]= listeB[von:bis]	ersetzt in der A-Liste die Elemente ab ***nach*** durch die Elemente aus der B-Liste (hier ein Ausschnitt)	liste[2:]= liste[2:7] liste[4:]=	
listeA[:bis]= listeB[von:bis]	ersetzt in der A-Liste die Elemente vor ***bis*** durch die Elemente aus der B-Liste (hier ein Ausschnitt)	liste[:2]	
listeA[vor:nach]= listeB[von:bis]	fügt in die A-Liste in die Position zwischen ***vor*** und ***nach*** ersetzend die B-Liste (hier ein Ausschnitt) ein	liste[3:4]= liste[2:7] liste[4:7]= liste	[1,2,3,3,4,5,10,11,12,5,10, 11,12,13,14,15]

Aufgaben:

1. *Erstellen Sie sich eine Liste mit den ersten 9 Buchstaben (als Zeichen) unter dem Namen liste!*
2. *Überlegen Sie sich (ohne Python!)) für alle Beispiele aus der Zusammenfassungs-Tabelle oben die Ergebnisse!*
3. *Prüfen Sie nun mit Python!*
4. *Erstellen Sie sich eine Liste aller Groß-Buchstaben als Liste von Zeichen mit dem Namen buchstaben!*
5. *Geben Sie für die nachfolgenden Problemstellungen eine Listen-Formulierung an!*
   a)
6. *Prüfen Sie nun mit Python!*

8.2.3.8. Ringe – geschlossene Listen

Ringe sind in Python als solche nicht vorgesehen. Da man aber auch mit negativen Indizies arbeiten kann, hat man es praktisch bei Listen mit Ringen zu tun. Die Ring-Größe wird durch die Länge der Liste bestimmt. Die jeweilige Schreib- oder Lese-Position wird durch den aktuellen Index der innerhalb der Listen-Länge inkrementiert oder dekrementiert werden kann. Alternativ bietet sich auch eine Überwachung über die Modulo-Operation an.

Aufgaben:

1. *Erstellen Sie sich ein Programm, dass in einer einzugebenen Liste aus Elementen jeweils die Liste und die aktuelle Index-Postion (Zeiger) anzeigt (z.B. als "I" unter der Liste, s.a. folgendes Beispiel)!*

```
Liste/Ring       :    1    2    3    4    5    6    7    8
akt. Zeiger(=  2):              I
```

2. *Erweitern Sie nun das Programm von 1. so, dass der Nutzer (in einer Schleife) angeben kann, um wieviele Positionen sich der Zeiger verschieben soll! Die Anzeige soll wieder die Liste und die aktuelle Zeiger-Postionen sein!*
3.

8.2.3. Dictionarys - Wörterbücher

Wenn man es ganz genau nimmt, dann sind Dictionarys eigentlich eher Vokabel-Listen oder Daten-Paare. Ein Daten-Paar besteht immer aus einem "Schlüssel" – also dem beschreibenden Begriff – und einem zugeordneten Daten-Element. Als Daten dürfen die verschiedenen schon besprochenen Datentypen fungieren, sowie deren Verknüpfungen in Tupel (→), Mengen (→) und Vektoren (→).
Wir sehen hier schon, dass Dictonary's auch so einiges mit Listen gemeinsam haben. In der Informatik werden die Dictonary's aber eher als sehr lockere Liste besser Sammlung von Daten-Paaren gesehen.
Dictonary's werden einfach els eine spezielle Datenstruktur definiert. Wenn man etwas mit einfachen Listen machen will (→ 8.2.3. Listen, die I. – einfache Listen), dann nutzt man auch die Datenstruktur Liste. Stehen díe Daten-Paare im Vordergrund und die Auflistung ist zweitrangig, dann sind Dictonary's eine mögliche Wahl.

Definition(en): Dictonary
Im informatischen Sinn versteht man unter der Datenstruktur Dictonary eine Listen-artige Sammlung von Daten-Paaren.

8.4. Interation oder Rekursion? – das ist hier die Frage!

Die Frage, die wir uns hier stellen müssen, ist die nach dem besten Vorgehen beim Lösen eines Problems. Eine Variante wäre es ein Problem zuerst einmal auf ein oder mehrere einfachere Probleme zurückzuführen. Das macht man solange, bis es kein einfacheres Problem mehr gibt oder die Lösung offensichtlich ist. Auf dem Rückweg zum ehemaligen aufrufenden (großen) Problem ergänzt man die primitive Lösung immer ein Stück weiter.
Glaubt man der Literatur, dann ist dieses Lösungs-Verfahren, welches von Menschen und Programmierern (auch das sollen Menschen sein?!), am häufigsten / vorrangig genutzt wird.
Die andere Variante ist das gleichartige Wiederholen einer bekannten / einfachen Lösung bzw. einer Teil-Tätigkeit, bis die Aufgabe gelöst ist.
Meiner Meinung nutzen Menschen eher diese Methode. Bei Personen, die Programmieren lernten ist es ebenfalls die zuerst gewählte Lösungs-Strategie.
Praktisch ist es wohl eine nicht-entscheidbare Frage – wie die, was denn nun zuerst da war, das Huhn oder das Ei. Zum Einen lassen sich Probleme fast immer mit beiden Strategien lösen. Dabei ist meist die eine Strategie eleganter / effektiver / cleverer / schöner / …, aber das steht nicht Disposition.
Zum Anderen gibt es sie nicht – die universell beste Strategie, sonst könnten wir sie ja einfach ansagen / lehren / predigen. Vielfach hängt das beste Vorgehen von den Rahmen-Bedingungen ab, die zur Verfügung stehen. Im Computer-Bereich sind dies z.B. Speicherplatz oder die Rechen-Zeit.
Meist geht es bei Interation und Rekursion auch begrifflich etwas hin und her. Den die Interation oder die Rekursion gibt es nicht. Es sind verallgemeinerte Strategien.
Praktisch müsste man zwischen der interativen und / oder rekursiven Defintion einer Funktion und der programmiertechnischen Implementierung unterscheiden.
i.A. lassen sich die – wie auch immer definierten – Funktionen auf beide Arten implementieren; allerdings gibt es ohne weiteres Programmier-System, die bestimmte Strategien bevorzugen bzw. manche andere gar ausschließen.
Vielfach entscheidet der Programmierer, was günstiger ist.
Da beide Umsetzungen Vor- und Nachteile haben, müssen die System-Bedingungen aber mit beachtet werden

Die Suchmaschine google zeigt nach der Eingabe eines Suchbegriffes gleich unter der Trefferzahl und er Bearbeitungszeit oft auch ein "Meinst du: *XYZ*". Dabei werden vorrangig kleine Schreibfehler "korrigiert" oder alternative Begriffe angeboten. Sucht man nun auf der deutschen google-Seite nach Rekursion, dann ist das Ergebnis schon etwas überraschend. Auch auf der englischsprachigen Seite passiert mit dem Begriff "recursion" das Gleiche. Ist google hier ein Fehler unterlaufen? Wie unterscheiden sich die – und weitere alternative - Antwortseiten?

8.4.1. Interation

Denken wir z.B. an die Aufgabe eine 10 Kisten mit Wasserflaschen in der dritten Stock zu transportieren. Für einen echten Body-Builder kein Problem. Er weiss bloß nicht, was er in die andere Hand nehmen soll (;-).
Jeder würde diese Aufgabe sicher dadurch lösen, dass er kleinere Mengen (wahrscheinlich immer 2 Kästen) nach oben bringt. Die komplizierte (schwer zu lösende) Aufgabe wurde in mehrere gleiche Teil-Aufgaben zerlegt.

Teilaufgabe
Teilaufgabe
Teilaufgabe
Teilaufgabe
Teilaufgabe

Ein solches Problem-Lösen nennen wir **interatives Vorgehen**.
Aus informatischer Sicht ist das die Wiederholung strukturgleicher Blöcke mit Teilaufgaben. Wenn wir irgendwelche Dinge – z.B. eine Ausgabe x-mal wiederholen wollten, dann haben wir das in einer Schleife erledigt. Das ist eine klassische interative Lösung. Wir hätten auch ein Programm schreiben können, dass zumindestens für eine bekannte Anzahl von Wiederholungen, genau die gleiche Ausgabe in einem Stück erzeugt hätte. Da würden wir uns entweder die Finger wund tippen oder x-mal die Copy-und-Paste-Strategie anwenden müssen.
Alle Schleifen stellen typische Interationen dar. Der Wortstamm kommt auch vom latainischen *interare* für wiederholen.

Definition(en): Interation
Unter Interation versteht man das mehrfache (abzählbare / gezählte) Wiederholen einer Aktion / Handlung / Anweisung.
Interation ist die Anwendung immer gleicher Prozesse auf bereits gewonnene Zwischen-Ergebnisse.

Vorteile einer / der Interation

- **weniger Speicher-Bedarf**
- **intuitiv verständlich**
- **im direkten Vergleich meist schneller** meist sogar deutlich schneller
-

Nachteile einer / der Interation

- **kompliziertere Umsetzung**
- **längere Programmtexte**
-

8.4.1.1. typische Interations-Anwendungen

Eigentlich könnte ich mir diesen Abschnitt sparen, da die bisher besprochenen Wiederholungen fast ausnahmslos Interationen waren.
Da aber Summen und Produkte und vor allem deren Entwicklung in Schleifen zu den klassischen Programmier-Aufgaben gehören, seien sie hier noch mal aufgeführt, wiederholt und zum systematischen Verständnis dargestellt.
Wem die Summen- und Produkt-Bildung schon zur Nase raushängt und die Schwierigkeit damit nicht verstehen kann, der sollte gleich zu den Rekursionen (→ 8.4.2. Rekursion) übergehen. Da erwartet ihn vielleicht Neueres und Spannendes.

8.4.1.1.1. Summen-Bildung

```
def summe(endzahl):
    sum=0
    for i in range(1,endzahl+1):
        sum=sum+i
    return sum

# main
endzahl=eval(input("Bis zu welcher Zahl soll summiert werden?: "))
print("Die Summe lautet: ",summe(endzahl))
```

Wie sieht die Speicher-Belegung zum Zeitpunkt des Eintritts in die Zählschleife aus?
Ein Speicherzelle "endzahl" wurde mit der Eingabezeile angelegt und mit der Nutzer-Eingabe (hier: 10) gefüllt.
Beim Aufruf der Funktion summe wird nun eine Kopie dieser Speicherzelle angelegt, die aber nur innerhalb der summe-Funktion gültig ist. Gleiches gilt für die anderen Variablen.

summe()	i	1
summe()	sum	0
summe()	endzahl	10
	endzahl	10
	Name	***Speicher***

Man kann die Unabhängigkeit von endzahl gut testen, indem man z.B. innerhalb der Funktion die endzahl (vielleicht direkt vor dem return) ändert. Eine Ausgabe von endzahl im Hauptprogramm liefert die eingegebene Zahl. Mit dem return werden alle Variablen der Funktion summe gelöscht.
Auch davon kann man sich durch eine versuchte Ausgabe der summe-Funktions-Variablen im Hauptprogramm überzeugen. Es gibt eine Fehlermeldung.
Beim ersten Schleifen-Durchlauf ist i gleich 1 und wird in der Summierungszeile zuerst einmal (rechte Seite des Terms) auf den (alten) Inhalt von sum aufaddiert. Das Berechnungs-Ergebnis wird dann in der Speicherzelle sum (quasi als neue Belegung) gespeichert.

summe()	i	1
summe()	sum	1
summe()	endzahl	10
	endzahl	10
	Name	***Speicher***

Eigentlich würden wir in Python die Aufsummierung ja eher so schreiben: sum+=i. Das macht den Ablauf der inneren Speicher-Abläufe aber nicht nachvollziehbar.

summe()	i	1

summe()	sum	1
summe()	endzahl	10
	endzahl	10
	Name	***Speicher***

Mit dem Erreichen der letzten Schleifen-Anweisung (hier haben wir ja nur eine) wird i um Eins erhöht und geprüft, ob die Schleife ein nächstes Mal durchlaufen werden muss (i ist jetzt noch kleiner als endzahl+1).
Am Ende aller Schleifendurchläufe ist sum mit 55 belegt. Dieser Wert wird nun an die print-Anweisung übergeben. Natürlich hätte man auch eine andere Variable zur Übernahme des Funktionswertes nutzen können.

summe()	i	11
summe()	sum	55
summe()	endzahl	10
	endzahl	10
	Name	***Speicher***

Die gesamte Variablen-Struktur der Funktion wird nach dem return gelöscht und ist nicht wieder erreichbar. Nur bei speziellen Generator- Funktionen (→ 6.5.3. Generator-Funktionen – Funktionswerte schrittweise) bleibt die Variablen-Struktur für einen erneutetn Funktionsaufruf erhalten.

8.4.1.1.2. Produkt-Bildung

Die algorithmischen Änderungen zur Summe-Funktion sind minimal. Natürlich sollten die Bezeichner usw. angepasst werden. Aber für ein schnelles Test-Programm würde es auch ohne gehen.

```
def produkt(endzahl):
    prod=1
    for i in range(1,endzahl+1):
        prod=prod*i
    return prod

# main
endzahl=eval(input("Bis zu welcher Zahl soll multipliziert werden?: "))
ergebnis= produkt(endzahl)
print("Das Produkt lautet: ",ergebnis)
```

Aufgaben:

1. ***Erstellen Sie ein Speicher-Schema für das Produkt-Programm!***
2. ***Überlegen Sie sich, was passieren würde, wenn man innerhalb der Schleife ergebnis immer auf 13 setzt! Diskutieren Sie Ihre Voraussage mit anderen Kursteilnehmern! Probieren Sie es dann aus!***
3. ***Schreiben Sie eine Summe- und eine Produkt-Funktion in einem Programm, welche immer die Zahlen von einer Start- bis zu einer Endzahl (über Eingaben festzulegen) verarbeiten!***

8.4.2. Rekursion

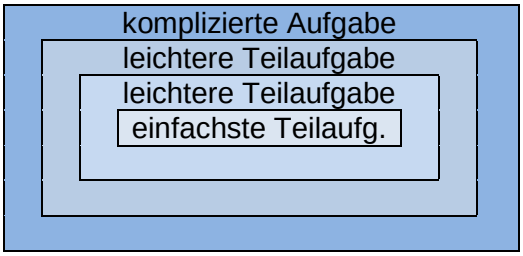
Kommen wir noch mal auf unser 10-Wasser-Kisten-Beispiel zurück. Um sie in den drutten Stock zu bekommen, können wir selbst mit jeweils 2 Kisten fünfmal Treppen steigen und die Kisten hochschleppen.
Eine andere Strategie wäre es, die Aufgabe einfach zu zerlegen. Ich übergebe das Kisten-Problem an den nächsten Party-Gast / Wassertrinker, indem ich ihn für den Transport von 8 Kisten verantwortlich mache. Ich selbst nehme 2 Kisten und bringe sie hoch. Der andere hat ein deutlich einfacheres Problem, als ich vorher mit 10 Kisten. Der Zweite kann nun genauso vorgehen. Sich einen "Dummen" suchen, der 6 Kisten als Auftrag bekommt und er selbst auch 2 Kisten nach oben transportiert. Der "Dumme" wird so weiterverfahren. Wenn es dann irgendwann nur noch 4 Kisten sind, übergibt der vorletzte Transporteur die (leichteste / letzte) Aufgabe an den letzten Party-Gast / Wassertrinker. Jeder der beiden löst nun seine Transport-Aufgabe und bringt jeweils 2 Kisten nach oben. In der Wohnung wird dann alles wieder zu einem 10-Kisten-Stapel zusammengesetzt.

in der Informatik versteht man darunter die Rückführung einer schwierigeren / aufwändigeren / komplizierteren / allgemeinen Aufgabe in eine leichtere / weniger aufwändigen / einfacheren / speziellen.
vom lat.: recurrere (zurücklaufen, zurückkehren)

besonders gern benutzt und besonders eindrucksvoll sind Rekursionen in der Grapfik-Programmierung
nutzt man dann noch die Turtle-Graphik (→ 8.8. Turtle-Graphik – ein Bild sagt mehr als tausend Worte), dann kann man Rekursion praktisch erleben (→ 8.8.6. Rekursion)

– theoretisch unendlich oft – in sich selbst geschachtelte Schleife
wobei hier nicht die Schleife das Struktur-Objekt ist sondern eine sich selbst-aufrufende Funktion

Ein Problem haben wir allerdings. Man braucht immer eine leichteste / letzte Teilaufgabe. Diese nennen wir Rekursions-Abbruch oder aus der anderen Richtung betrachtet Rekursions-Anfang. Die anderen – delegierenden / vereinfachenden – Schritte werden Rekursions-Schritt genannt.



In der Mathematik ist die Rekursion ein gängiges Mittel zur Definition von Funktionen

z.B.: Bildung einer Summe

sum(0) = 0 *Rekursions-Anfang*

jede andere Summe lässt sich dann so berechnen:

sum(n) = sum(n-1) + n *Rekursions-Schritt*

die gesamte Definition lautet dann

$$\text{sum}(n) = \begin{cases} 0, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ \text{sum}(n-1) + n, & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

Auch wenn es ein bisschen wie eine interative Lösung aussieht, hier ist der entscheidende Unterschied, dass die Funktion sich selbst wiederaufruft. Bei der Interation wird nur wiederholt.

damit ein Problem rekursiv zu lösen geht, muss es die folgenden Bedingungen erfüllen:

- das Problem muss sich in eine einfachere Variante von sich selbst zerlegen lassen
- bei der Zerlegung in eine einfachere Variante muss irgendwann eine Variante erreicht werden, die sich ohne weitere Zerlegung lösen lässt
- wenn die Teilprobleme gelöst sind, dann müssen sich die Teil-Lösungen zu einer Lösung des Ausgangs-Problems zusammensetzen lassen

Definition(en): Rekursion
Unter Rekursion versteht man das nicht voraussehbare Wiederholen einer Aktion / Funktion durch Aufruf von sich selbst.
Rekursion ist das Problemlösungs-Konzept, bei dem eine (komplexe) Aufgabe in (kleinere, leichter lösbare) Teil-Aufgaben (der gleichen Klassen) zerlegt wird, diese gelöst werden und dann zur Gesamt-Lösung zusammengesetzt werden. Rekusionen bedürfen einer trivialen Teil-Aufgaben-Lösung, ab der eine weitere Aufgaben-Zerlegung nicht mehr durchgeführt werden kann.

Vorteile einer / der Rekursion

- **relativ einfache Defintion**
- **dem menschlichen Denken ähnlich**
- **Korrekheit ist i.A. leichter zu prüfen**
- **kürzere Formulierung**
- **kürze Implementierungen**
- **spart Variablen**
- **(i.A.) sehr effektiv**

Ob rekursives Arbeiten wirklich dem menschlichen Denken sehr nahe kommt, wage ich zu bezweifeln. Meine Erfahrungen sagen eher, dass rekursive Prinzipien / Funktionen zumindestens sehr einfach erscheinen, beim Umsetzen in ein Programm wird es deutlich schwieriger und problematisch wird es, wenn selbst neuartige Sachverhalte / Probleme rekursiv gelöst werden sollen
Meist erscheint dann irgendwie die interative Lösung logischer oder eingängiger. Kommt man später auf eine rekursive Lösung, ist sie zwar meist deutlich eleganter, aber auch schwerer zu verstehen und zu warten.

Nachteile einer / der Rekursion

- **unübersichtlicher Programmablauf**
- **schlechtes Laufzeit-Verhalten** — meist deutlich langsamer
- **größerer Speicher-Bedarf** — z.B. für Rücksprung-Adressen von noch nicht gelösten übergeordneten Funktions-Aufrufen
  (großer Overhead von Funktions-Aufrufen)

•

einige Programmiersprachen kennen nur Rekursionen, bei ihnen fehlen andere Wiederholungs-Strukturen (z.B. Scheme)
Computer arbeiten intern aber immer interativ, aber das ist nicht unsere Ebene

Wir unterscheiden direkte und indirekte Rekursion. Die direkte ist dadurch gekennzeichnet, dass die Funktion sich immer wieder selbst aufruft. Bei der indirekten Rekursion rufen sich mehrere Funktionen gegenseitig auf. Sind es z.B. zwei, dann ruft Funktion1 die Funktion2 auf und diese dann wieder Funktion1.

8.4.2.1. Rekursions-Beispiele: Summen- und Produkt-Bildung

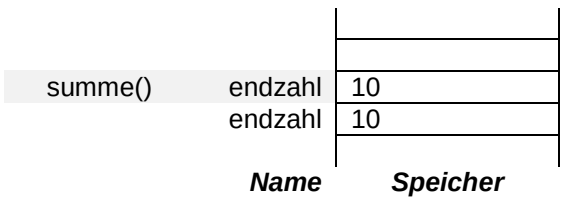
```
def summe(endzahl):
    sum=0
    for i in range(1,endzahl+1):
        sum=sum+i
    return sum

# main
endzahl=eval(input("Bis zu welcher Zahl soll summiert werden?: "))
print("Die Summe lautet: ",summe(endzahl))
```

Betrachten wir hier auch die beiden Funktionen (summe und produkt), die oben bei den Interationen nochmals besprochen worden.
Über die Rekursion beschreiben wir die Funktion summe wie oben besprochen:

$$summe(n) = \begin{cases} 0, & \text{falls } n = 0 \quad \textit{Rekursions-Anfang} \\ summe(n-1) + n, & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

```
def summe(endzahl):
    if endzahl==0:
       return 0
    else:
       return summe(endzahl-1)+endzahl

# main
endzahl=eval(input("Bis zu welcher Zahl soll summiert werden?: "))
print("Die Summe lautet: ",summe(endzahl))
```

Typisch ist die umgekehrte Abarbeitung zur kleinsten Zahl / zum Abbruch-Kriterium hin.
Die Speicher-Belegung ist aber letztendlich deutlich verschieden.
Beim Aufruf der Funktion summe wird wieder eine Kopie von endzahl angelegt. Nun wird die Verzeigung passiert und bevor irgenwas getan



wird, wird die Funktion schon wieder verlassen allerdings mit einem erneuten Aufruf von summe. Das Argument wurde aber um Eins verringert.
Zur Kennzeichnung eines untergeordneten Aufrufs verwende ich unterschiedlich dunkle Grautöne.

summe()	endzahl	9
summe()	endzahl	10
	endzahl	10

Dieser Vorgang wiederholt sich jetzt einige Male bis der Aufruf mit dem Argument 0 (für die endzahl) erfolgt.
Es erfolgt ein Return mit 0 und nun wird der Speicherstapel abgebaut, indem der Rückgabe-Wert des untergeordneten Funktions-Aufrufs mit der – auf der jeweiligen Ebene gültigen – endzahl addiert wird.
Letztendlich kommen wir so zum 1. Funktionsaufruf zurück und der gibt nun das Ergebnis (aus der Berechnung summe(9)+endwert) an den aufrufenden Programmschritt zurück (hier die Ausgabe).

summe()	**endzahl**	0
summe()	endzahl	1
		…
summe()	endzahl	7
summe()	endzahl	8
summe()	endzahl	9
summe()	endzahl	10
	endzahl	10

Schon bei nur 10 Rekursionen wird also deutlich mehr Speicher gebraucht, als in der interativen Version.
Typische Rekursionen haben meist eine deutlich größere Rekursionstiefen und häufig auch noch interne Variablen. Auch diese benötigen Platz im sogenannten Kellerspeicher, LIFO-Speicher oder Stack. Der zuletzt gespeicherte Inhalt wird zuerst wieder herausgeholt (Last In First Out). Anders herum kann man sich das Speicher-Prinzip auch als Stapel (engl.: stack) verstellen. Man muss Neues oben auflegen und auch von oben der Stapel wieder abbauen.
Die informatische Datenstruktur "Keller" wird später nochmals ausführlich (Objekt-orientiert) besprochen (→ 9.8. Keller).

Aufgaben:

1. *Erstellen Sie die Definition für ein Produkt!*
2. *Erstellen Sie ein Programm mit einer rekursiven Produkt-Funktion!*
3. *Zeigen Sie an einem Speicher-Schema, welche Variablen wann angelegt werden und welche Werte sie beinhalten!*

8.4.2.2. weitere typische Anwendungen für Rekursionen

8.4.2.2.1. Überführung einer Dezimal-Zahl in eine Dual-Zahl

```
def dualzahl(dezimalzahl):
    ganzzahlteiler=dezimalzahl/2
    rest=dezimalzahl%2
    if rest==0:
        stellensymbol="0"
    else:
        stellensymbol="1"
    if ganzzahlteiler==0
        return stellensymbol
    else:
        return dualzahl(ganzzahlrest) + stellensymbol
```

8.4.2.2.2. die Fakultät

faktorielle Funktion
für die Wahrscheinlichkeitsrechnung / Stochastik häufig gebraucht
in der Mathematik durch das Ausrufe-Zeichen nach der Zahl ausgedrückt:

6! = 1 * 2 * 3 * 4 * 5 * 6 = 720

oder eben allgemein:

n! = 1 * … * (n-1) * n = $\prod_{i=1}^{n} i$

die meisten Programmierer würden wohl auch eher interativ an die Implementierung herangegen (→ 8.4.1.1.2. Produkt-Bildung)
hier schauen wir uns aber auch mal die rekursive Lösung an:

$$\text{fakultät}(n) = \begin{cases} 1, & \text{falls } n = 1 \quad \textit{Rekursions-Anfang} \\ \text{fakultät}(n-1) + n, & \text{sonst} \quad \textit{Rekursions-Schritt} \end{cases}$$

```
def fakultaet(x):
    if x==1: return 1
    else:
        return fakultaet(x-1)*x
```

8.4.2.2.3. die FIBONACCHI-Folge

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$$fib(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Rekursions-Anfang

Rekursions-Schritt

Exkurs: FIBONACCHI ohne die Vorglieder?

Das Berechnen eines bestimmten Gliedes der FIBONACCHI-Folge ist durch Rekursion und Interation möglich. Beide Lösungswege – also die interative bzw. die rekursive – haben durch die vielen Wiederholungen bzw. Funktionsaufrufe einen recht großen Rechenaufwand. Schließlich müssen alle Vorglieder berechnet werden, um dann die letzten beiden Vorglieder zum Ergebnis zu addieren.
Besonders für höhergliedrige Werte in der Folge ist der Rechenaufwand dann enorm.
Der französische Mathematiker J.-Ph.-M. BINET schlug (1843) eine andere Funktion zur Berechnung der Einzelglieder vor:

$$fib(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Einen solchen Lösungsweg nennen wir **explizit**. Explizite Lösungen sind meist extrem schnell – vor allem im Vergleich zu den anderen beiden Lösungs-Strategien. Der Aufwand für die Implementierung liegt im Bereich der Interation – also etwas aufwändiger, als für eine Rekursion.
Explizte Lösungen von Problemen, die im "normalen" Leben einen großen Rechnenaufwand haben stellen z.B. häufig Sicherheits-Probleme dar. Wenn z.B. eine Sicherheits-Lösung darauf aufbaut, dass sie erst mit einem riesigen Rechenaufwand geknackt werden kann, und es exisiert auf einmal eine explizite Lösung, dann stürzt das Sicherheits-Konzept in sich zusammen.

Aufgaben:

1. ***Programmieren Sie die nachfolgende "Super"-FIBONACCHI-Folge als rekursive Funktion mit kleinem Rahmen-Programm zur Anzeige mehrerer Folge-Glieder!***

$$sfib(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ 2, & \text{falls } n = 2 \\ sfib(n-1) + sfib(n-2) + sfib(n-3), & \text{sonst} \end{cases}$$

Rekursions-Anfang

Rekursions-Schritt

zur Kontrolle: erwartete erste Glieder der Folge:
0, 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, …

Aufgabe (für Fortgeschrittene):

2. Erstellen Sie ein Programm, dass für die ersten 20 Glieder der FIBONACCHI-Folge die Werte jeweils klassisch interativ und rekursiv und dann noch einmal mit der BINET-Funktion berechnet. Prüfen Sie, ob es Differenzen gibt (Anzeigen lassen!)!

8.4.2.2.4. das ggT – der Größte gemeinsame Teiler

Natürlich müsste es der ggT (GGT; eng.: gcd (greatest common divisor)) heißen, aber wer spricht schon so?

beim ggT mehrerer (mehr als 2) Zahlen muss allerdings auf die Primfaktoren-Zerlegung zurückgegriffen werden

bei zwei Zahlen

$$\begin{aligned} 10584 &= \mathbf{2^3} * \mathbf{3^3} * 7^2 \\ 40500 &= \mathbf{2^2} * \mathbf{3^4} * 5^3 \\ \\ ggT: &\quad \mathbf{2^2} * \mathbf{3^3} = 108 \end{aligned}$$

wird für mehr Zahlen

$$\begin{aligned} 1400 &= \mathbf{2^3} * \mathbf{5^2} * 7^2 \\ 283500 &= \mathbf{2^2} * 3^4 * \mathbf{5^3} * 7^1 \\ 20250 &= \mathbf{2^1} * 3^4 * \mathbf{5^3} \\ \\ ggT: &\quad \mathbf{2^1} * \mathbf{5^2} = 50 \end{aligned}$$

Primfaktoren-Zerlegung sehr rechen-aufwändig

EUKLIDischer und STEINscher Algoritmus

Grundidee von EUKLID und dann durch STEIN verbessert

40500	:	**10584**	=	3	Rest:	**8748**
10584	:	**8748**	=	1	Rest:	1836
8748	:	1836	=	4	Rest:	1404
1836	:	1404	=	1	Rest:	432
1404	:	432	=	3	Rest:	108
432	:	**108**	=	4	Rest:	**0**

ggT(x, y, z) = ggT(ggT(x, y), z) = ggT(x, ggT(y, z))

8.4.2.2.5. Erkennung von Palindromen

rekursiv:

```
def ist_palindrom(zeichenkette):
    if len(zeichenkette)<=1:
        return 1
    if zeichenkette[0]!=zeichenkette[-1]:
        return 0
    return ist_palindrom(zeichenkette(s[1:-1])
```

interativ:

```
def ist_palindrom(zeichenkette):
    links=0
    while links<rechts:
        if zeichenkette[links]!=zeichenkette[rechts]:
            return 0
        links+=1
        rechts-=1
    return 1
```

mit speziellen Python-Funktionen für Strings und Listen:

```
def ist_palindrom(zeichenkette):
    buchstabenliste=list(zeichenkette)
    buchstabenliste.reverse()
    return ("".join(1))
```

ist bei Zeitvergleichen die schnelleste Variante, weil die Listen- und String-Funktionen in Maschinensprache realisiert sind

8.4.2.2.x. weitere klassische Rekursions-Probleme

Türme von Hanoi

rekursive Zerlegung des aktuellen Turm in die größte / untereste Scheibe und einen kleineren (Rest-)Turm

ACKERMANN-Funktion

1926 von Wilhelm ACKERMANN beschrieben
wird zur Austestung von Speicher- und Computer-Modellen benutzt, da die Funktion extrem schnell wächst

$$
\begin{aligned}
ack(a, b, 0) &= a + b \\
ack(a, 0, n+1) &= ack2(a, n) \\
ack(a, b+1, n+1) &= ack(a, ack(a, b, n+1), n)
\end{aligned}
$$

$$
ack2(a, n) = \begin{cases} 0, & \text{wenn } n=0 \\ 1, & \text{wenn } n=1 \\ a, & \text{wenn } n>1 \end{cases}
$$

durch PÈTER 1935 etwas einfacher definiert:

$$
\begin{aligned}
ack(0, m) &= m+1 \\
ack(n+1, 0) &= ack(n, 1) \\
ack(n+1, m+1) &= ack(n, ack(n+1, m))
\end{aligned}
$$

rekursiv:

```
def ackermann(n, m):
    if n==0:
        return m+1
    elseif  m==0:
        return ackermann(n-1, 1)
    else:
        return ackermann(n-1, ackermann(n, m-1)
```

teilweise interativ:

```
def ackermann(n, m):
    while n!=0:
       if m==0:
           m=1
       else:
           m=ackermann(n, m-1)
       n+=1
    return m+1
```

Quicksort

Beim Quicksort-Verfahren wird eine Liste von Zahlen od.ä. Objekten dadurch sortiert, das die originale Liste in immer kleiner werdende Liste aufgeteilt wird. Dabei wird einfach nur nach Größe in die eine oder andere Liste eingeordnet. Als Entscheidungs-Element (Grenzwert) wird ein zufälliger Wert oder z.B. einfach das erste Element der Liste benutzt. Das Entscheidungs-Element wird auch Pivot-Element genannt. Das Wörtchen *pivot* bezeichnet im Französischen den Dreh- und Angel-Punkt.
Optimalerweise sollten die Teil-Listen immer die halben Listen der Vorgänger-Liste sein, dann sortiert dieses Verfahren sehr schnell.
Genaueres später bei der Besprechung verschiedener Sortier-Algorithmen (→ 8.15. Sortieren – eine Wissenschaft für sich).

Mergesort

Eine ähnliche Strategie verfolgt der Sortier-Algorithmus Mergesort. Auch hier wird in kleine(re) Listen zerlegt, die dann für sich sortiert werden. Am Schluss werden die sortierten Teil-Listen durch Mischen (*merge* = engl.: verschmelzen) vereint.
Mergesort folgt dem Teile-und-herrsche-Prinzip (divide and conquer), welches erstmals von J. VON NEUMANN (1945) beschrieben wurde und praktisch auch in seinen Rotor-Maschinen zum Knacken des Enigma-Code's verwendet wurde.
Genaueres später bei der Besprechung verschiedener Sortier-Algorithmen (→ 8.15. Sortieren – eine Wissenschaft für sich).

Potenzierung von Zahlen

interativ

```
def potenz(basis, exponent):
    pot=1
    for i in range(exponent+1):
        pot*=basis
    return pot
```

rekursiv

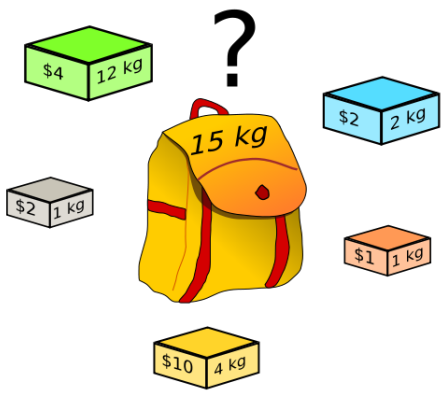
```
def potenz(basis, exponent):
    if exponent==0:
        return 1
    else:
        return basis*potenz(basis, exponent-1)
```

Rucksack-Problem

eng.: knapsack problem
Optmirungs-Problem aus der Kombinatorik

gegeben ist eine Menge von Objekten, die einen Nutzwert und ein Gewicht (Kostenfaktor) besitzen
gesucht ist eine Teilmenge, deren Gewicht eine bestimmte Grenze nicht überschreitet und der Nutzen aber maximiert sein soll

gehört zu den klassischen NP-vollständigen Problemen (Richard KARP (1972))



Veranschalichung des Rucksack-Problems
Q: de.wikipedia.org (Dake)

Zahlen-Beispiel von http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo15.php

Objekt	1	2	3	4	5	6	7	8		
Gewicht	153	54	191	66	239	137	148	249		
Profit	232	73	201	50	141	79	48	38		
Profit-Dichte	1,52	1,35	1,05	0,76	0,59	0,58	0,32	0,15		

Gewichts-Schranke soll z.B. bei 645 liegen

1. intuitiver Lösungs-Ansatz:
nehme die Objekte mit der höchsten Profit-Dichte
also → 1, 2, 3, 4 → Gewicht=464 → Profit=556

2. Lösung, wie 1. und Auffüllen mit weiteren passenden Objekten (entsprechend der Rangfolge)
also → 1, 2, 3, 4, 6 → Gewicht=601 → Profit=647

→ aber nicht optimal! ?????
es muss jede Kombination ausprobiert werden!

2^n Möglichkeiten (einpacken oder nichteinpacken / 1 oder 0)
Problem ist hier die expotentielle Steigerung des Rechen-Aufwandes
es gibt scheinbar mehrere Lösungen !?

besser ist der Algorithmus von NEMHAUSER und ULLMANN (1969)
basiert auf PARETO-Prinzip

Alpha-Beta-Suche für Spielzüge bei Brettspielen (Computer-Stategie)

Volumen-Berechnung einer n-dimensionalen Hyperkugel

Suche in einem Baum

Weg aus einem Labyrint

Rechte-Hand-Regel

geht natürlich auch als Linke-Hand-Regel

Permutationen

```
def permutation():

    return
```

effektive Speicherung von Daten (z.B. Bilder)

Wollten wir das nebenstehende Bit-Muster / Bild über eine Liste abspeichern, dann würde diese mit 64 Elementen doch recht lang werden. Nehmen wir an, es geht oben links los und es wird Zeilen-weise gearbeitet, dann ergibt sich die folgende Liste:

1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	1	1
1	1	1	1	0	0	1	1
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	1	1	0	0	0	0

muster=[1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1,
1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0,
0, 0, 1, 1, 1, 1, 0, 0, 0, 0]

Das Bild / Daten-Muster wird zuerst als das größtmögliche Quadrat (gleicher Elemente) betrachtet. Wäre es z.B. homogen nur mit Einsen gefüllt, dann würde sich als Muster die Liste **muster=[1]** ergeben. Statt der 64 Speicher-Elemente hätten wir es nur noch mit einem zu tun. Wir bräuchten also grob 1/64 des Speicherbedarfs – wenn das keine Kompression ist?!
Das Muster ist aber heterogen, also müssen wir es verkleinern.
Die verkleinerten Quadrate sind umrandet hervorgehoben. Für jedes der kleineren Quadrate müssen wir nun in unserer muster-Liste ein Listen-Element verwenden. Da es mehr als eins ist, wird klar, dass das gesamte 8x8-Quadrat strukturiert ist. Die Grundstruktur sieht dann so aus:

muster=[, , ,]

Wären jetzt die kleineren (4x4)-Quadrate einheitlich gefärbt (mit 1 oder 0 belegt), dann würden wir mit 4 Listen-Elementen hinkommen, was immer noch einer Effektivität der Kompression von 4/64 entsprechen würde. Aber leider ist es im Beispiel nicht so, also müssen wir genauer weiter differenzieren.
Das oberste linke Quadrat ist vollständig mit Einsen gefüllt, also speichern wir uns in die Teil-Liste eine Eins:

muster=[1, , ,]

Nun wechseln wir zum rechten oberen 4x4-Quadrat. Es ist nicht homogen und muss deshalb wieder unterteilt werden. Es entsteht also eine Liste (**rot gekennzeichnet**) in der Liste (Das Listen-Element ist selbst wieder eine Liste).

muster=[1, **[, , ,]**, ,]

Die sich ergebenden 2x2-Quadrate sind homogen, also kann die Muster-Liste nun so geschrieben werden:

muster=[1, **[0, 0, 0, 1]**, ,]

Das untere, linke Quadrat ist nicht homogen, also muss es zerlegt werden. Auch das oberste linke 2x2-Quadrat ist nicht homogen, also muss es als Bit-Muster in die Liste geschrieben werden.

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], , ,],]

Das zweite 2x2-Quadrat ist homogen mit Nullen belegt, also speichern wir ein 0 in die Liste.

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, ,],]

Bei den unterern beiden 2x2-Quadraten verfahren wir in der gleichen Weise und erhalten dann:

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, [1, 0, 1, 1], 1],]

Bleibt das letzte (untere, rechte) 4x4-Quadrat. Es ist homogen, so dass die Liste nur die darin enthaltene Null repräsentieren muss:

muster=[1, [0, 0, 0, 1], [[0, 1, 1, 0], 0, [1, 0, 1, 1], 1], 0]

Im vergleich zur obigen Voll-Liste kommen wir nun mit 26 Speicher-Elementen aus. Das bedeutet eine Verbesserung fast um den Faktor 2,5 (grob: 16/64).
Die Kompressionsraten sind sehr theoretisch berechnet. Es muss beachtet werden, dass noch Strukturierungs-Elemente (zur Unterscheidung von über- und unter-geordneten Listen) mit abgespeichert werden müssen.
So ähnlich – wie hier besprochen – laufen z.B. Kompressions-Verfahren, wie das JPEG oder MP4 ab. Neben dem Vergleich der Bild-Elemente werden auch noch die vorlaufenden Bilder mit verglichen. Dabei nutzt man den Effekt aus, dass sich in einer Bildfolge meist nur wenige – isolierte – Teile verändern.

Aufgaben:

1. *Übernehmen Sie das Muster und Muster-Liste! Kennzeichen Sie durch unterschiedlich farbige Umrandungen im Muster und durch entsprechend farbige Klammern, welche Bitmuster zu welchen Listen-Elementen gehören!*

Eine Rekursion bietet sich immer dann an, wenn das Problem / die Funktion schrittweise auf ein kleineres / leichteres Problem // eine einfachere Funktion reduziert werden kann.

McCARTHYs "91-Funktion"

$$f(n) = \begin{cases} n-10 & \text{falls } n > 100 \\ f(f(n+11)) & \text{sonst} \end{cases}$$

PELL-Folge

$$P(n) = \begin{cases} 0, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ 1, & \text{falls } n = 1 & \textit{Rekursions-Anfang} \\ 2P(n-1) + P(n-2) & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 0, 1, 2, 5, 12, 29, 70, 169, 408, …

die ersten beiden Elemente sind mit 0 und 1 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem verdoppelten Vorgänger und dem (einfachen) Vorvorgänger

PELL-Folge 2. Art

$$Q(n) = \begin{cases} 2, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ 2, & \text{falls } n = 1 & \textit{Rekursions-Anfang} \\ 2Q(n-1)+Q(n-2) & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 2, 2, 6, 14, 34, 82, 198, 478, 1154, …

die ersten beiden Elemente sind mit 2 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem verdoppelten Vorgänger und dem (einfachen) Vorvorgänger

LUCAS-Folge(n)

$$L(n) = \begin{cases} x, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ y, & \text{falls } n = 1 & \textit{Rekursions-Anfang} \\ L(n-1)+L(n-2) & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: immer abhängig von x und y
bei x=2 und y=1 → 2, 1, 3, 4, 7, 11, 18, 29, 47, …

die ersten beiden Elemente sind mit x und y definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem Vorgänger und dem Vorvorgänger

JACOBSTHAL-Folge

$$J(n) = \begin{cases} 0, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ 1, & \text{falls } n = 1 & \textit{Rekursions-Anfang} \\ J(n-1)+2J(n-2) & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683, …

die ersten beiden Elemente sind mit 0 und 1 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem Vorgänger und dem verdoppelten Vorvorgänger

RECAMÀNs-Folge
(OEIS → A005132)

$$R(n) = \begin{cases} 0, & \text{falls } n = 0 & \textit{Rekursions-Anfang} \\ R(n-1)-n & \text{falls } R(n) >= 0 \text{ und nicht in Sequenze} & \textit{Rekursions-Schritt} \\ R(n-1)+n & \text{sonst} & \textit{Rekursions-Schritt} \end{cases}$$

erste Elemente: 0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, 22, 10, 23, 9, 24, 8, 25, 43, …

die ersten beiden Elemente sind mit 0 und 1 definiert
die nachfolgenden Elemente ergeben sich als Summe aus dem Vorgänger und dem verdoppelten Vorvorgänger

interessante Links:
https://oeis.org/wiki/Welcome (On-Line Encyklopedia of Integer Sequences ® OEIS ®)
https://oeis.org/A?????? (Informationen zur Folge mit der Nummer ??????)

Aufgaben für die gehobene Anspruchsebene:

1. *Informieren Sie sich zur Biographie von N.J.A. SLOANE!*
2. *Was verbirgt sich hinter der Folge A000108?*

NUR!!! zum Üben: die DREWS-Folgen

Nicht wundern, natürlich gibt es diese Folge (wahrschein) nicht wirklich – und wenn, dann unter einem anderen Namen! Sie sind praktisch abgewandelte FIBONACCHI-Folgen. Bei der ersten Folge wird immer eine 1 dazuzählt. Also typisch DREWS – immer noch Einen drauf setzen. Die Folgen haben keinen tieferen Zweck, außer dem Programmieren zu dienen.

0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, …

$$dre(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ dre(n-1) + dre(n-2) + 1, & \text{sonst} \end{cases}$$

Rekursions-Anfang

Rekursions-Schritt

Die zweite Folge ist etwas komplexer. Hier unterscheidet sich das Zuzählen danach, ob die Gliednummer gerade oder ungerade ist.

0, 1, 2, 5, 8, 15, 24, 41, 66, 109, 176, 287, …

$$dre2(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 2, & \text{falls } n = 1 \\ dre2(n-1) + dre2(n-2) + 1, & \text{sonst, falls n gerade} \\ dre2(n-1) + dre2(n-2) + 2 & \text{sonst, falls n ungerade} \end{cases}$$

Rekursions-Anfang

Rekursions-Schritte

Aufgaben:

1. *Programmieren Sie die 1. DREWS-Folge als rekursive Funktion mit einem kleinen Rahmen-Programm!*
2. *Erstellen Sie ein Programm, mit dem die DREWS-Folge sowohl rekursiv als auch interativ berechnet wird!*
3. *Entwickeln Sie nun ein Programm, das die DREWS2-Folge rekursiv berechnet!*

für die gehobene Anspruchsebene:

4. *Erstellen Sie ein Programm, mit dem die DREWS2-Folge sowohl rekursiv als auch interativ berechnet wird!*
5. *Vergleichen Sie den Implementier-Aufwand für die Berechnung der DREWS2-Folge beim interativen und rekursiven Vorgehen!*

8.4.2.2. direkte Gegenüberstellung von interativen und rekursiven Algorithmen

In der Literatur und in der täglichen Programmierarbeit finden sich bzw. entstehen die unterschiedlichsten Umsetzungen von bestimmten Problem. Einige sind hier gesammelt und jeder programmierer wird nach und nach den einen oder anderen programm-Text hinzutuen können. Ob die einzelnen Lösungen immer optimal (gut lesbar, schnell, wenig Speicherbedarf, …) sind, wird hier nicht bewertet. Sollten Algorithmen entscheidend für ein Programm sein, dann müssen spezielle Test (Abfrage Speicherbedarf, Zeitmessungen, …) erfolgen. Auf einige Möglichkeiten gehen wir noch ein.
In einigen Algorithmen sind **blaue** **print**-Anweisungen eingebaut. Diese dienen als optionale Ausgabe, um das Arbeiten des Algorithmus zu verfolgen. Für echte Anwendungen sollten sie dann raus genommen werden. Ev. lassen sich weitere sinnvolle Ausgaben erzeugen, z.B. um die Anzahl der Schleifendurchläufe bzw. die Rekursion-Aufrufe zu zählen. Dafür müssen dann aber eigene Variablen und Zähl-Anweisungen eingebaut werden.
Bei einigen ausgewählen Algorithmen-Umsetzungen notieren wir das in **rot**er Farbe. Auch diese Quelltext-Teile sollten vor dem echten Einsatz entfernt werden.

8.4.2.2.1. GGT – größter gemeinsamer Teiler

Lösung	interativ	rekursiv

1	`def ggt(a,b):`
	`  i = 0`
	`  while b > 0:`
	`  i += 1`
	`  print("Durchlauf: ",i)`
	`  print("a= ",a,"b= ",b)`
	`  r = a % b`
	`  a = b`
	`  b = r`
	`  return a`
		`def ggt(a,b,i):`
		`  i += 1`
		`  print("Aufruf: ",i)`
		`  print("a= ",a,"b= ",b)`
		`  if b == 0:`
		`  return a`
		`  return ggt(b, a % b,i)`
		
		`# Aufruf der Funktion:`
		`i = 0`
		`ggt(a,b,i)`

2	`def ggt(a,b):`
	`  while b != 0:`
	`  a,b = b, a % b`
	`  return a`
		`def ggt(a,b):`
		`  return if a == b:`
		`  a`
		`  elif a > b:`
		`  ggt(a-b,b)`
		`  else:`
		`  ggt(a, b-a)`

3	`def ggt(a,b):`
	`  while a != b:`
	`  if a > b:`
	`  a=a-b`
	`  else:`
	`  b=b-a`
	`  return a`	

		`def ggt(a,b):` `  return if a > b:`

8.4.2.2.2. Palindrom-Prüfung

Lösung	interativ	rekursiv

1	`def ist_palim(s):`
	`  links = 0`
	`  rechts = len(s)-1`
	`  while links < rechts:`
	`  if s[links] != s[rechts]:`
	`  return 0`
	`  links += 1`
	`  rechts -= 1`
	`  return 1`
		`def ist_palim(s):`
		`  if len(s) <= 1:`
		`  return 1`
		`  if s[0] != s[-1]:`
		`  return 0`
		`  return ist_palim(s[1:-1])`

2		

außer Konkur-renz	`def ist_palim(s):`
	`  liste = list[s]`
	`  liste.reverse()`
	`  return ("".join(liste))`	

8.4.2.2.2. Potenz-Prüfung

ist p eine ganzzahlige Potenz von x

Lösung	interativ	rekursiv

1	`def istpotenz(p,x):`
	`  return if p == 1 or p == x`
	`  or p % x != 0:`
	`  p == 1 or p == x`
	`  else:`
	`  istpotenz(p/x,x)`
		`def istpotenz(p,x):`
		`  return if p == 1 or p == x:`
		`  True`
		`  elif p % x !=0:`
		`  False`
		`  else:`
		`  istpotenz(p/x,x)`

2	`def istpotenz(p,x):`
	`  while p != 1 and p != x`
	`  and p % x == 0:`
	`  p = p / x`
	`  return p == 1 or p == x`	

8.4.3. komplexe Programmier-Aufgaben:

Wählen Sie eine geeignete oder Ihre präferierte Programmiersprache zur Lösung der nachfolgenden Aufgaben aus!
Überlegen Sie sich bzw. vergleichen mit anderen, ob die von Ihnen präferierte programmiersprache gut geeignet ist das gewählte Problem zu lösen!
Zahlen-Eigenschaften nach: www.zahlen.mathematic.de

Aufgaben:

1. ***Berechnen Sie die Summe und das Produkt einer Reihe von einzugebener Zahlen sowie Summe und Produkt der reziproken Werte!***
2. ***Erstellen Sie ein Programm, dass im Zahlen-Raum bis zur einer einzugebenen (größeren) natürlichen Zahl, die Kombination von drei aufeinanderfolgenden Primzahlen findet, deren Produkt möglichst dicht an der Zahlen-Grenze liegt!***
3. ***Prüfen Sie ob eine als Zeichen-String vorgegebene Zahl (ohne Leer- und Vorzeichen bzw. Nachkommastellen) im auszuwählenden Zahlensystem gültig ist! (Die Ziffern werden als ASCII-Zeichen notiert. Gültige und unterscheidbare Zeichen sind: 0 .. 1 A .. Z a .. z → das sollte auch bis zum Sexagesimal-System reichen! Doppeldeutung A = a muss nicht beachtet werden!)***
4. ***Lassen Sie durch eine Erweiterung des Programms von 3. prüfen, ob es sich bei der eingegeben Zahl um eine normale Zahl handelt! Normale Zahlen enthalten alle Ziffern ihres Alphabetes mit der gleichen Häufigkeit.***
5. ***Erstellen Sie das Programm "Zahlen-Charakterisierer"! Das Programm soll eine einzugebene ganze Zahl (ev. zuerst nur für natürliche Zahlen) Charakter-Eigenschaften prüfen bzw. bestimmen und ausgeben, ob die Zahl die Eigenschaft hat oder nicht bzw. den berechneten Wert. Das Programm sollte später um weitere Zahlen-Eigenschaften ergänzt werden können und passend kommentiert sein! Auf die (spätere) Nutzbarkeit von Unterprogrammen ist zu achten! Wählen Sie sich mindestens 12 Eigenschaften aus! Die Reihenfolge kann frei geändert werden!***
   - ***a) männliche Zahl (Zahl ist ungerade und größer als 1)***
   - ***b) Quersumme (ist die Summe der einzelnen Ziffern der Zahl (ohne deren Potenzwert))***
   - ***c) titanische Zahl (ist eine Primzahl mit mindestens 1000 Stellen***
   - ***d) weibliche Zahl (Zahl ist eine positive gerade Zahl)***
   - ***e) Totient od. Indikator (ist die Anzahl der Primzahlen, die kleiner als die (gegebene) Zahl ist)***
   - ***f) zusammengesetzte (od. zerlegbare od. teilbare) Zahl (ist eine Zahl, die mehr als zwei positive Teiler hat ODER eine gerade Zahl, die größer als 1 ist)***
   - ***g) abundante Zahl (wenn echte Teilersumme (Summe aller Teiler (ohne Rest), außer die Zahl selbst) größer als die Zahl selbst ist)***
   - ***h) arme od. defizierte od. mangelhafte Zahl (wenn echte Teilersumme kleiner als das doppelte der Zahl ist)***

i) vollkommene od. perfekte Zahl (wenn die echte Teilersumme gleich der Zahl selbst ist)

j) Sophie-GERMAIN-Primzahl (sind Primzahlen, bei denen der Term 2 p +1 wieder eine Primzahl ist)

k) reiche od. überschießende od. übervollständige Zahl (wenn die echte Teilersumme größer als das Doppelte der Zahl selbst ist)

l) SMITH-Zahl (wenn die Quersumme der Zahl gleich der Quersummen ihrer Primfaktoren ist; außer Primzahlen!)

m) erhabene Zahl (wenn Zahl und deren echte Teilersumme vollkommene Zahlen sind)

n) palindrome Zahl (wenn die Zahl und die umgedrehte Ziffernfolge gleich (groß) sind)

o) palindrome Primzahl (wenn Zahl eine Primzahl ist und die Zahl und deren umgedrehte Ziffernfolge gleich sind)

p) SIERPINSKI-Zahl (ist eine ungerade, natürliche Zahl n, bei der der Term $n\ 2^x + 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

q) RIESEL-Zahl (sind ungerade, natürliche Zahlen, bei denen der Term $n\ 2^x - 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

r) strobogrammatische Zahl (ist eine Zahl, die um 180° gedreht wieder die gleiche Zahl ergibt (hier gelten 1, 2 mit 5, 6 mit 9, 8 und 0 als drehbare oder strobogrammatische Ziffern))

s) strobogrammatische Primzahl (ist eine Primzahl, die auch strobogrammatisch ist)

6. Gesucht wird ein modulares Programm, dass für zwei natürliche Zahlen prüft, ob es sich um ein Paar mit den folgenden Eigenschaften handelt!

a) befreundete Zahlen (wenn die echten Teilersummen beider Zahlen gleich sind))

b) Primzahlen-Zwilling (wenn zwei aufeinanderfolgende Primzahlen eine Differenz von 2 aufweisen)

c) Teiler-fremde (od. inkommensurable) Zahlen (ganze Zahlen, die außer -1 und 1 keine gemeinsamen Teiler besitzen)

7. Gesucht wird ein modulares Programm, dass für drei natürliche Zahlen prüft, ob es sich um ein Tripel mit den folgenden Eigenschaften handelt!

a) pythagoreische Zahlen (Tripel erfüllt die diophantische Gleichung 2. Grades ($a^2 + b^2 = c^2$))

b) Primzahlen-Drilling (wenn drei aufeinanderfolgende Zahlen die Reihe p, p+2, p+6 bilden ODER wenn innerhalb einer Dekade (also 10 aufeinanderfolgenden Zahlen) drei Primzahlen vorkommen)

8.

8.5. Umgang mit Dateien

8.5.0. Dateien und Ordner

Text-Dateien
relativ leicht zu erzeugen, immer selbst durch Programmierer möglich, meist aber Module zum effektiveren Umgang verfügbar, sowohl vom Computer, als auch von Menschen lesbar
relativ Fehler-tolerant

praktisch ein Umleiten der Bildschirmausgabe in eine Datei

Binär-Dateien
Daten sind sehr kompakt gespeichert, praktisch nur noch Maschinen-lesbar
Fehler können für völlige Unlesbarkeit der Daten sorgen

Arbeiten mit Dateien bestehen immer aus drei Abschnitten, die unbedingt eingehalten bzw. erledigt werden müssen

- **Eröffnung, Initialisierung** — Festlegen der Datei über den Dateinamen und den Datentyp
  Festlegung der Zugriffsart auf Datei
- **eigentliches Schreiben bzw. Lesen** — eben genau das
  (praktisch könnte dieser Abschnitt auch entfallen, aber wozu dann der andere – unbedingt notwendige! - Aufwand)
- **Datei-Freigabe** — Beenden des Zugriffs auf die Datei, damit wird die Datei für andere Programme, Programmteile etc. benutzbar
  an dieser Stelle erfolgt vielfach erst das physikalische Schreiben

8.5.1. Dateien lesen

8.5.1.1. Lesen von Text-Dateien

Dateivariable = **open(**Dateiname,"r"**)**

Zeilenlistenvariable = Dateivariable.readlines()

Zeilenvariable = Dateivariable.readline()

Dateivariable**.close()**

Dateivariable**.seek(**Position**)**

Will man den gesamten Datei-Inhalt in einen String schreiben, dann lässt sich das folgendermaßen realisieren:

```
Dateivariable = open(Dateiname,"r")
Inhalt = Dateivariable.read()
print("Datei-Typ: ",type(Inhalt)
print("Datei-Inhalt:")
print(Inhalt)
Dateivariable.close()
```

8.5.1.1.1. Lesen von CSV- bzw. strukturierten TXT-Dateien

Einlesen einer Text-Datei und Speichern als CSV

```
zielDatei = open("daten.CSV", "w")

for zeile in open("Text.TXT"):
    zeile = zeile.strip()
    if zeilestartswith("#"):
        continue
    elemente = zeile.split()
    print(elemente)
print(";".join(elemente), file = zielDatei)
```

8.3.1.1.2. Lesen von XML-Dateien

spezielle Module verfügbar

8.1.1.1.3. Lesen von JSON-Dateien

spezielle Module zum decodieren verfügbar

```
import sys, json

dateinmae = "test.JSON"

print(json.dumps(json.load(dateinmae)), indent =2))
```

8.5.1.2. Lesen von Binär-Dateien

8.5.2. Dateien schreiben

8.5.2.1. Schreiben von Text-Dateien

8.5.2.1.1. Schreiben einer neuen Datei

Dateivariable = **open(**Dateiname,"w"**)** zum Neuschreiben

Dateivariable**.write(**"\n"+Zeile | "\n"+Zeilenvariable**)**
die write-Funktion liefert übrigens die Anzahl der geschriebenen Zeichen wieder zurück

alternativ auch Umleitung der Bildschirmausgabe möglich
print >> Dateivariable, Text | Textvariable
etwas einfacher, weil Ausgaben immer in String- / Text-Format umgewandelt werden

Dateivariable**.close()**
hier extrem wichtig, weil erst jetzt das echte Speichern erfolgt!

mit writelines(StringListe) kann eine Liste von Strings in einen Text-Datei geschrieben werden

8.5.2.1.2. anhängendes Schreiben

zum Anhängen weiterer Daten an eine existierende Datei
Dateivariable = **open(**Dateiname,"a"**)**
Dateivariable**.write(**Zeile+"\n" | Zeilenvariable+"\n"**)**

letzte Zeile ohne "\n"
Dateivariable**.write(**Zeile | Zeilenvariable**)**

Dateivariable**.close()**
nicht vergessen!

8.5.2.1.3. Schreiben von CSV- bzw. strukturierten TXT-Dateien

8.5.2.1.4. Schreiben von XML-Dateien

8.5.2.1.5. Schreiben von JSON-Dateien

8.5.2.2. Schreiben von Binär-Dateien

8.5.3. gepickelte Dateien – Dateien mit gemischten Daten

8.5.3.1. Schreiben von Dateien mit gemischten Daten

8.5.3.2. Lesen von Dateien mit gemischten Daten

8.6. Module

= Bibliothek
Sammlung vorgefertigter Programm-Teile (meist Funktionen)
praktisch Objekte (→ Objekt-orientierte Programmierung)

Probleme dann möglich, wenn im aktuellen Programm-Ordner schon eine Datei mit dem Namen des Moduls vorhanden ist, dann muss mit Fehler-Meldungen gerechnet werden
genau wenn man versucht seine eigene Datei mit dem Namen eines Moduls abzuspeichern, das geht zwar, aber der Aufruf der Module / Modul-Funktionen geht nicht → Fehler-Meldungen

vollständiger Import eines Moduls

```
import modul
wert = modul.funktion(10)
print(modul.funktion(20))
```

Funktionen müssen mit vorgesetztem Modul-Namen aufgerufen werden
Vorteile:
man kann eigene Funktionen und (globale) Variablen mit dem gleichen Namen im Programm händeln

Nachteile:
lästiges Mitschreiben des Modul-Namens

Import einzelner Funktionen eines Moduls

```
from modul import funktion
wert = funktion(10)
print(funktion(20))
```

Vorteile:
Funktion kann ohne Modul-Namen aufgerufen werden

Nachteile:
Aufruf **from … import** für jede einzelne Funktion oder für Gruppen notwendig

vollständiger Import eines Moduls als integraler Programmteil

```
from modul import *
wert = funktion(10)
print(funktion(20))
```

Vorteile:
keine selektiven Importe mehr

Nachteile:
es werden viele unnötige Funktionen importiert

Modul-Import mit Vergabe eines internen Namens

```
import modul as mo
wert = mo.funktion(10)
print(mo.funktion(20))
```

Vorteile:
keine selektiven Importe mehr
kürzere Modulschreibung möglich
Module mit gleichen internen Funktionen / Attributen lassen sich sauber trennen

Nachteile:
es werden viele unnötige Funktionen importiert

Anzeige der verfügbaren Funktionen und (globalen) Variablen

```
import math
print(dir(math))
```

```
>>>
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

8.6.1. "built-in"-Funktionen

Funktionen, die schon direkt im klassischen Python verfügbar sind
häufig gebraucht; weiterhin sollen sie schnell und Fehler-frei bzw. Fehler-unanfällig sein
in vielen anderen Programmiersprachen gehören sie gleich zum Befehls-Umfang dazu
in Python extra Module; dadurch etwas langsamer aber auch veränderlich / überschreibbar, wenn's denn wirklich notwendig ist

z.B. **max()**, **min()**, **abs()**, **type()**

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

8.6.2. wichtige interne Module

??? random, math

8.6.2.x. Verschiedenes zum Modul: sys

8.6.2.x. Verschiedenes zum Modul: time

clock()
liefert einen Programm-internen Zeitstempel (Programm-Laufzeit) zurück
für Laufzeit-Messungen vergleicht man einfach die Zeitstempel vor und nach dem zu prüfenden Programm-Teil / Algorithmus / Funktions-Aufruf

```
from time import *

…

t0 = clock()
# hier steht dann der zu testende Quelltext
t1 = clock()

print("Laufzeit: ",t1-t0,"s")
```

```
>>>
```

time()
liefert Zeitstempel als Fließkommazahl
gut für genauere Zeit-Differenzen auch im ms-Bereich geeignet

```
import time

…

t0 = time.time()
# hier steht dann der zu testende Quelltext
t1 = time.time()

print("Laufzeit: ",t1-t0,"s")
```

```
>>>
```

ctime()
Zeitstempel wird als Text ausgegeben (mit Datum und Uhrzeit)

8.6.2.x. Verschiedenes zum Modul: os

Ermitteln des aktuell verfügbaren / freien Speichers

für Linux-basierte Systeme (über Shell-Aufruf)

```
import os

…

speicher = os.popen('df -m | grep rootfs | cut -c33-42').readline()
print("freier Speicher: ", int(speicher)," Byte")
```

```
>>>
```

die intern verfügbaren Module der Standard-Python-Installation (V. 3.5.0.)

```
AutoComplete        _pickle             enum                pydoc_data
AutoCompleteWindow  _pyio               errno               pyexpat
AutoExpand          _random             faulthandler        pygame
Bindings            _sha1               filecmp             queue
CallTipWindow       _sha256             fileinput           quopri
CallTips            _sha512             fnmatch             random
ClassBrowser        _sitebuiltins       formatter           re
CodeContext         _socket             fractions           reprlib
ColorDelegator      _sqlite3            ftplib              rlcompleter
Debugger            _sre                functools           rpc
Delegator           _ssl                gc                  run
EditorWindow        _stat               genericpath         runpy
FileList            _string             getopt              sched
FormatParagraph     _strptime           getpass             select
GrepDialog          _struct             gettext             selectors
HyperParser         _symtable           glob                setuptools
IOBinding           _testbuffer         gzip                shelve
IdleHistory         _testcapi           hashlib             shlex
MultiCall           _testimportmultiple heapq               shutil
MultiStatusBar      _thread             hmac                signal
ObjectBrowser       _threading_local    html                site
OutputWindow        _tkinter            http                smtpd
ParenMatch          _tracemalloc        idle                smtplib
PathBrowser         _warnings           idle_test           sndhdr
Percolator          _weakref            idlelib             socket
PyParse             _weakrefset         idlever             socketserver
PyShell             _winapi             imaplib             sqlite3
RemoteDebugger      abc                 imghdr              sre_compile
RemoteObjectBrowser aboutDialog         imp                 sre_constants
ReplaceDialog       aifc                importlib           sre_parse
RstripExtension     antigravity         inspect             ssl
ScriptBinding       argparse            io                  stat
ScrolledList        array               ipaddress           statistics
SearchDialog        ast                 itertools           string
SearchDialogBase    asynchat            json                stringprep
SearchEngine        asyncio             keybindingDialog    struct
StackViewer         asyncore            keyword             subprocess
ToolTip             atexit              lib2to3             sunau
TreeWidget          audioop             linecache           symbol
UndoDelegator       base64              locale              symtable
WidgetRedirector    bdb                 logging             sys
WindowList          binascii            lzma                sysconfig
ZoomHeight          binhex              macosxSupport       tabbedpages
__future__          bisect              macpath             tabnanny
__main__            builtins            macurl2path         tarfile
_ast                bz2                 mailbox             telnetlib
_bisect             cProfile            mailcap             tempfile
_bootlocale         calendar            marshal             test
_bz2                cgi                 math                textView
_codecs             cgitb               mimetypes           textwrap
_codecs_cn          chunk               mmap                this
_codecs_hk          cmath               modulefinder        threading
_codecs_iso2022     cmd                 msilib              time
_codecs_jp          code                msvcrt              timeit
_codecs_kr          codecs              multiprocessing     tkinter
_codecs_tw          codeop              netrc               token
_collections        collections         nntplib             tokenize
_collections_abc    colorsys            nt                  trace
_compat_pickle      compileall          ntpath              traceback
_csv                concurrent          nturl2path          tracemalloc
_ctypes             configDialog        numbers             tty
_ctypes_test        configHandler       opcode              turtle
_datetime           configHelpSourceEdit operator            turtledemo
_decimal            configSectionNameDialog optparse            types
_dummy_thread       configparser        os                  unicodedata
_elementtree        contextlib          parser              unittest
_functools          copy                pathlib             urllib
_hashlib            copyreg             pdb                 uu
```

```
_heapq              crypt               pickle              uuid
_imp                csv                 pickletools         venv
_io                 ctypes              pip                 warnings
_json               curses              pipes               wave
_locale             datetime            pkg_resources       weakref
_lsprof             dbm                 pkgutil             webbrowser
_lzma               decimal             platform            winreg
_markerlib          difflib             plistlib            winsound
_markupbase         dis                 poplib              wsgiref
_md5                distutils           posixpath           xdrlib
_msi                doctest             pprint              xml
_multibytecodec     dummy_threading     profile             xmlrpc
_multiprocessing    dynOptionMenuWidget pstats              xxsubtype
_opcode             easy_install        pty                 zipfile
_operator           email               py_compile          zipimport
_osx_support        encodings           pyclbr              zlib
_overlapped         ensurepip           pydoc
```

8.6.3. externe Module installieren und nutzen

8.6.3.x. Package-Installer PIP

Für das Installieren von Paketen (Bibliotheken, Modulen, …) gibt es Python das Tool PIP.
Es wird in der Eingabeaufforderung / Konsole bedient. Soll in ein aktuelles System ein zusätzliches Paket installiert werden, dann wechselt man in der Konsole ins Installations-Verzeichnis von Python.
Eine gute Hilfe für die nicht mehr DOS-mächtigen Konsolen-Benutzer ist ein verstecktes feature von Windows. Im Windows-Explorer klickt man bei gedrückter [⇧]-Taste auf die rechte Maus-Taste. Im nun angezeigten Kontext-Menü findet man auch den Menü-Punkt: "Eingabeaufforderung hier öffnen".
Sollte dieser Menü-Punkt nicht erscheinen, dann geht auch die folgende Schrittfolge:

1. Öffnen des Windows-Explorer's und Auswählen des übergeordneten Ordner's (bezogen auf den Ziel-Ordner)
2. Öffnen der Konsole
3. Eintippen des Befehls: cd
4. Ziehen des Ziel-Ordner-Symbols aus dem Windows-Explorer in die Konsole

→ der vollständige Pfad steht jetzt hinter dem cd-Befehl und kann ausgeführt werden

Ev. muss noch ein Laufwerks-Wechsel mittels Laufwerk-Buchstabe und angehängtem Doppelpunkt gemacht werden. In der Konsole wird dann mit:

```
pip install Paketname
```

das angegebene Paket installiert. Es werden diverse Verlaufs-Informationen angezeigt und beim ordnungs-gemäßen Durchlauf auch eine Erfolgs-Bestätigung.
Zum Deinstallieren verwendet man:

```
pip uninstall Paketname
```

Das ist aber eigentlich in speziellen Situationen notwendig.
Läuft das Python-System schon länger, dann sollte man das PIP-Programm zuerst einmal selbst updaten:

```
pip install –upgrade pip
```

Das ist auch eine Option, falls eine Paket-Installation nicht klappt. Oft geht es dann mit der aktuellen PIP-Version.
Die Anzeige der installierten Pakete erfolgt mit:

```
pip list
```

Sollen (ver)alte(ter) Paket angezeigt werden und die zugehörigen neuerenVersionen, dann hilft das Kommando:

```
pip list --outdated
```

Vielleicht ist der genaue Paketname nicht bekannt, oder die Versionen überschlagen sich, dann ist eine Suche nach bestimmten Paketen mit:

```
pip search "Anfrage"
```

möglich.

8.7. Graphik

Auf der Kommando-Ebene und in der Grund-Version bietet Python keinen Zugriff auf die graphischen Fähigkeiten von Windows oder vergleichbaren Betriebssystemen und / oder ihren Benutzer-Oberflächen.
Für einfache Zwecke kann man sich mit Pseudo-Graphiken behelfen. Dabei werden die verschiedensten ASCII-Symbole benutzt. Vor allem im erweiterten ASCII-Code ab Symbol 128 sind diverse Zeichen für Pseudographiken enthalten. Damit lassen sogar grobe Diagramme zeichnen.

Um die graphischen Möglichkeiten moderner betriebssysteme zu nutzen, bedarf es aber der Einbindung von geeigneten Modulen.
An vorderster Front ist hier die Turtle-Graphik mit dem Modul turtle (→ 8.8. Turtle-Graphik – ein Bild sagt mehr als tausend Worte)zu nennen. Sie geht auf

zurück.
Heute ist die Turtle-Graphik vollständig Pixel-orientiert. Die Schildkröte als Zeichen-Stift soll nur einer besseren Orientierung und der Verständlichkeit / Nachvollziehbarkeit dienen.

Andere graphische Systeme ermöglichen den Zugriff oder die Nutzung von typischen Bedien-Elementen aus den Betriebssystemen. Das sind zum Einen die Einzel-Bedienelemente, wie Text-Felder (Edit-Feld), Beschriftungen (Label), Optionen (), Auswahl-Listen (List- oder Combo-Boxen usw. usf. Zum anderen werden fertige Dialoge bereitgestellt. Hierzu zählen kleine Meldungs-Fenster (Message-Dialog) aber auch Datei-Speicher- oder -Öffnen-Dialoge.
Dies ermöglicht es dem Programmierer, sich auf den Kern seines Programm's zu konzentrieren und die sich wiederholenden, klassischen Aufgaben der Kompetenz von System-Profi's zu überlassen.
Die Module stellen dabei vorrangig Schnittstellen und Übersetzungen zwischen Python und den Graphik-Systemen der Betriebssysteme zur Verfügung.
Zu den bekanntesten Modulen, die Bedien-Elemente, Fenster und Dialoge bereitstellen, gehört Tkinter (→ 8.12. GUI-Programme mit Tkinter).

8.8. Turtle-Graphik – ein Bild sagt mehr als tausend Worte

didaktische Entwicklungs-Oberfläche (Interface-Builder)
schöne Bilder und coole Abläufe, Programmierung soll da so nebenbei mit aufgenommen werden
aber genau das ist die Gefahr aus meiner Sicht, das Nebenbei wird häufig zu wenig aufgenommen und von einem Jahr Programmierung bleiben nur wenige Monate Ahnungs-Effekte - an die tolle Oberfläche kann sich aber jeder noch erinnern
besonders gut für einen sehr frühen Kontakt mit Programmierung

wir werden hier die schon getätigten Schritte dieses Skriptes zur Einführung in die Programmierung quasi auf graphischem Niveau wiederholen

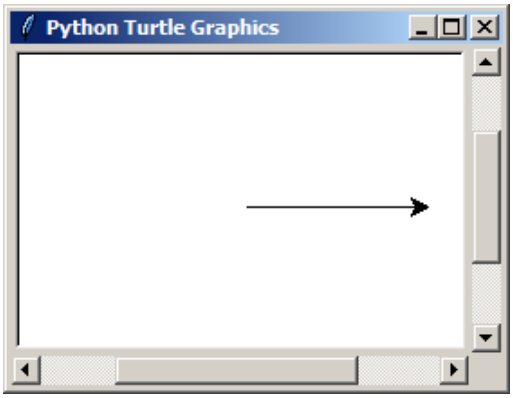
8.8.1. Turtle auf der Shell

Die Turtle-Graphik ist ein internes Modul, was mit der Installation von Python schon mit eingerichtet wurde. Um die Turtle-Graphik nutzen zu können, müssen wir unsere aktuelle Python-Version etwas pushen.

```
>>> import turtle
>>> turtle.forward(0)
>>> turtle.forward(100)
>>>
```

Durch die Eingabe von **`import turtle`** werden die Befehle und Funktionen der Turtle-Graphik geladen und verfügbar gemacht.
Eine alternative Bibliothek ist **gturtle**.

In den meisten Fällen passiert gar nichts, weder gibt es eine Meldung in der Shell, noch sehen wir eine Schildkröte.
Gibt man als nächsten Befehl z.B. **`turtle.forward(0)`** ein, dann erscheint ein Graphik-Fenster, in dessen Mitte eine Pfeilspitze zu sehen ist. Die Pfeilspitze ist unsere Schildkröte. Sie zu bewegen und sie dazu anzuregen Spuren zu hinterlassen, dass ist der Hintergedanke bei der Turtle-Graphik.

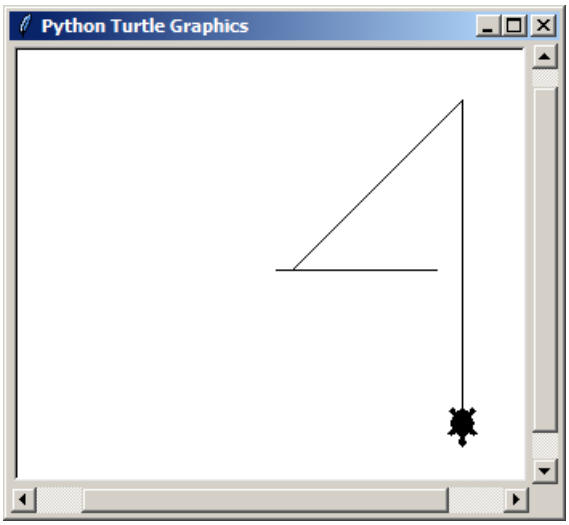


Mit **`turtle.forward(100)`** bewegen wir die Schildkröte 100 Pixel (auf dem Bildschirm) vorwärts. Üblicherweise werden die Pixel als Schritte interpretiert.
Der zurückgelegte Weg wird als Linie (gesetzte Pixel) sichtbar. Rückwärtes geht's mit **`turtle.backward(`***`schritte`***`)`**. Die zurückgelegten Wege sind dann vollständig oder teilweise deckungsgleich. Es ist aber nur eine Linie zu sehen.

```
>>> turtle.backward(90)
>>> turtle.left(45)
>>> turtle.forward(150)
>>> turtle.right(135)
>>> turtle.forward(200)
>>> turtle.shape("turtle")
```

Richtungs-Änderungen lassen sich mit **`turtle.left()`** und **`turtle.right()`** erreichen. Als Argument müssen wir den (Dreh-)Winkel (in Grad) übergeben. Wem das Dreieck als Turtle zu abstrakt ist und lieber eine echte Schildkröte wandern sehen möchte, der kann ja mal **`turtle.shape("turtle")`** ausprobieren.

Den ursprüngliche Dreiecks-Zeiger erhält man über den Text **`"classic"`**. Weitere Formen sind **`"arrow"`**, **`"circle"`**, **`"square"`** und **`"triangle"`**.



Ein Neustart der Turtle-Graphik – quasi das Löschen des Ausgabe-Bildschirm ("Vergessen der alten Wege") erreicht man mit **`turtle.reset()`**.
Beim Beenden des Shell-Fensters wird auch die Turtle-Graphik geschlossen. Will man dieses Fenster weiterhin sehen, gibt man in der Shell den Befehl **`turtle.exitonclick()`** ein.
Erwähnt sei hier auch noch der Befehl **`turtle.undo()`**, der den jeweils letzten Befehl rückgängig macht.
Hat man etwas Geduld bei der Eingabe der turtle-Befehle, das erscheint nach dem Eintippen des Punktes eine Code-Ergänzungs-Auswahlbox.
Mit der Tab-Taste wird die ausgewählte Funktion übernommen.
Vorher kann man mit der Maus oder mitttels der Eingabe weiterer Buchstaben die geeignete Funktion heraussuchen.

Aufgaben:

1. ***Aktivieren Sie die Turtle-Graphik und lassen Sie sich das Graphik-Fenster mit der Schildkröte in Lauerstellung anzeigen!***
2. ***Bewegen Sie die Schildkröte so, dass ein Summen-Zeichen (∑) auf dem Bildschirm zu sehen ist!***

Kommen wir nun schon im Vorfeld der echten Programmierung zu den Befehlen, die besonders von Mädchen / weiblichen Programmierern als erstes erfragt werden. Das ist ein Befehl, der die Farbe der Schildkröte und damit auch ihre Spur-Farbe ändern kann.
In einer ersten Möglichkeit legt man die Farbe mittels eines Color-Strings fest.
Das sind vordefinierte Farben mit den üblichen englisch-sprachigen Bezeichnungen.

Die zweite Variante der Farb-Festlegung ist weitaus Leistungs-fähiger, aber auch aufwändiger und komplizierter in der Umsetzung.

Bei dieser Variante wird die Farbe als RGB-Tupel übergeben. Es können nun die Zahlen für die Rot-, Grün- und Blau-Anteile von 0 bis 255 verändert werden.

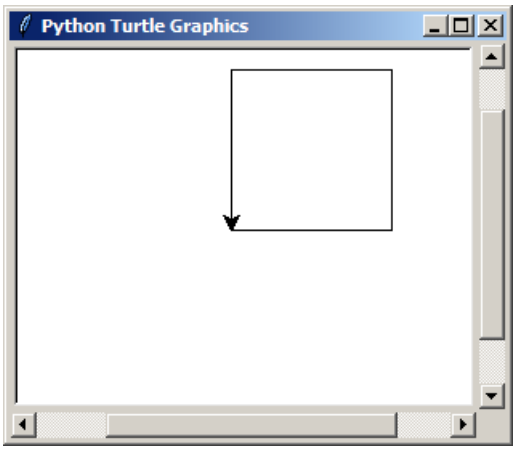
Verwendet man die Funktion turtle.color() ohne Argumente, dann wird die aktuelle Farbeinstellung als Color-String oder Hexadezimal-Code zurückgegeben.

Der Hintergrund des Graphik-Fenster lässt sich mit der Funktion turtle.bgcolor(farbcode) einstellen.
Ein geschlossener Polygonzug kann auch in der Fläche eingefärbt werden.

8.8.2. Turtle-Programme und Sequenzen

Das Erstellen von Programmen unterscheidet sich kaum von der Eingabe auf der Konsole (Shell) bzw. von der bei anderen Programmen. Das forward(0) zum Aktivieren / Anzeigen des Graphik-Fenster kann ausbleiben. Python zeigt das Fenster mit den ersten Programmschritten sofort an.
Wichtiger Hinweis auf einen Fehler-Klassiker bei der Turtle-Programmierung. Schnell vergibt man den Dateinamen für den selbstgeschriebenen Quelltext mit turtle.py. Danach funktioniert die Turtle-Programmierung nicht mehr, weil das Turtle-Modul quasi durch das eigene Programm ersetzt wurde. Man importiert das eigene Programm als Modul, was keinen Sinn macht und auch noch rekursiv ins Nirvana führt.
Unsere erste Aufgabe soll das Zeichnen eine Quadrates sein. Dazu brauchen wir die vier gleichen Seiten einer bestimmten Länge und am Ende der Seite immer eine Drehung um 90°.

```
# Zeichen eines Quadrates
import turtle
laenge=100
turtle.forward(laenge)
turtle.left(90)
turtle.forward(laenge)
turtle.left(90)
turtle.forward(laenge)
turtle.left(90)
turtle.forward(laenge)
```



Das Ergebnis überrascht wenig.
Aber das einfache Aneinanderreihen von Turtle-Anweisungen nervt jetzt schon ein bisschen. Viel tippen und wenig Leistung des Programms. Wir wissen ja schon, dass man mit Schleifen effektiver arbeiten kann.
Eigentlich würden jetzt zuerst die Verzweigung folgen, aber Graphik-Programmierung lebt mehr von Wiederholungen als von Alternativen. Die sind aber gleich danach dran (→ 8.8.4. Verzweigungen).

Aufgaben:

1. ***Erstellen Sie ein Programm, in dem die Schildkröte ein Rechteck mit den Kantenlängen 250 und 320 zeichnet!***
2. ***Erstellen Sie ein Programm, das ein Quadrat mit einer Kantenlänge von 200 zeichnet!***
3. ***Lassen Sie die Schildkröte das Quadrat so zeichnen, dass es um 45° gedreht ist!***
4. ***Erstellen Sie ein Programm mit der (ungeprüften) Eingabe eines Winkels (am Zeichen-Ursprung) für ein rechtwinkliges Dreieck! Die erste Kantenlänge soll 200 betragen! Lassen Sie alle anderen Winkel und Kanten berechnen! (Die Quadrat-Wurzel-Funktion sqrt() ist in der math-Bibliothek verfügbar.)***

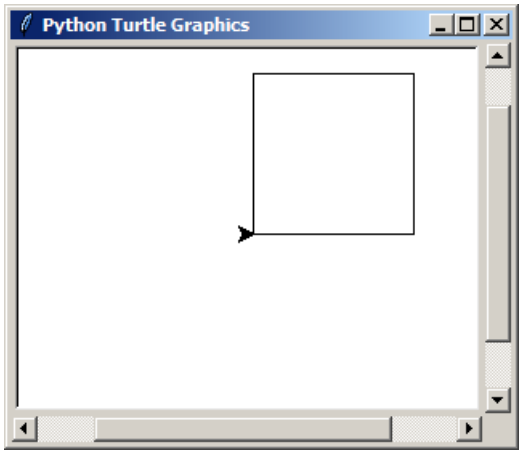
für die gehobene Anspruchsebene:

5. ***Gesucht ist das Programm, mit dem ein Rhombus mit einer Seitenlänge von 180 und den Winkeln 60 und 120 Grad gezeichnet wird! In der Erweiterung soll der gleicher Rhombus um 45° gedreht zusätzlich dargestellt werden!***

8.8.3. Schleifen

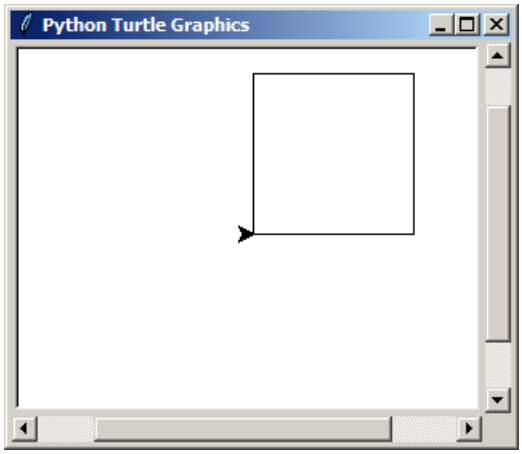
Also programmieren wir unser Quadrat über eine Schleife. Der Einfachheit halber nehmen wir auch noch eine abschließende Drehung in das Programm auf. Die Schildkröte wird somit auf die Ausgangs-Position und –Richtung zurückgesetzt.

```
# Zeichen eines Quadrates
import turtle
laenge=100
for i in range(4):
    turtle.forward(laenge)
    turtle.left(90)
```



Natürlich lässt sich das Gleiche auch mit einer while-Schleife erreichen. Was man praktisch wählt ist auch ein bisschen Geschmackssache. Im Falle von klaren Zähl-Vorgängen ist die Nutzung von for-Schleifen aber logisch verständlicher.
Mich persönlich schreckt auch immer der zusätzliche Aufwand (Vorbelegung der Laufvariable, Korrektur der Laufvariable) ab. Alles Fehler-Quellen, die sich durch eine "schöne" Zählschleife in Grenzen halten lassen.

```
# Zeichen eines Quadrates
import turtle
laenge=100
anzahl=0
while anzahl<4:
    turtle.forward(laenge)
    turtle.left(90)
    anzahl+=1
```



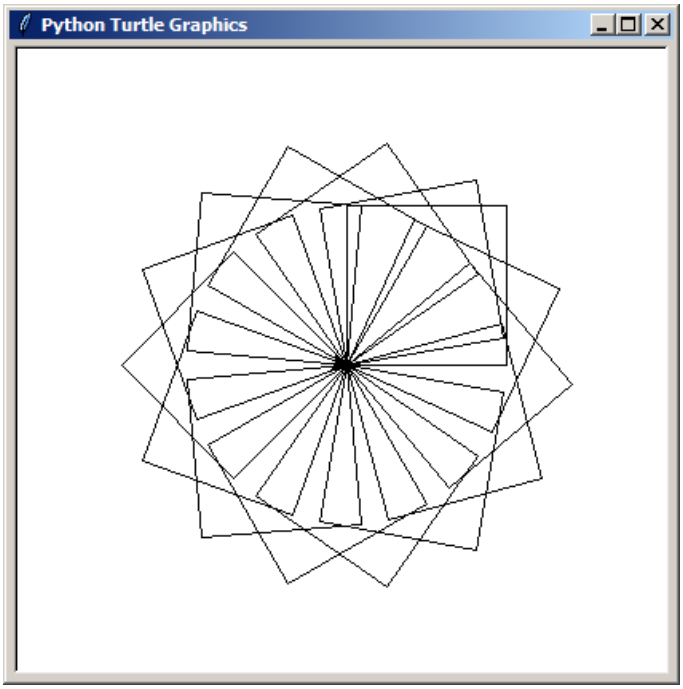
Beim Erzeugen von Mustern oder Wiederholungen, deren Anzahlen sich schwer abschätzen lassen, sind while-Schleifen dann natürlich die bessere Wahl.
Zur Demonstration nehmen wir hier mal das Zeichnen eines Musters aus Quadraten, die immer leicht verdreht zueinander solange gezeichnet werden sollen, bis die Umkreisung vollständig ist.

```
# Zeichen eines Musters aus Quadraten
import turtle
laenge=100
drehwinkel=25
gesamtwinkel=0
while gesamtwinkel<360:
    for i in range(4):
        turtle.forward(laenge)
        turtle.left(90)
    turtle.right(drehwinkel)
    gesamtwinkel+=drehwinkel
```

Zeichnen begrenzen; Schluß nach mind. 360°
einegentliches Zeichnen des Quadrates

Drehung auf neue Anfangsricht
(Gesamt-)Drehwinkel verfolgen / korrigieren

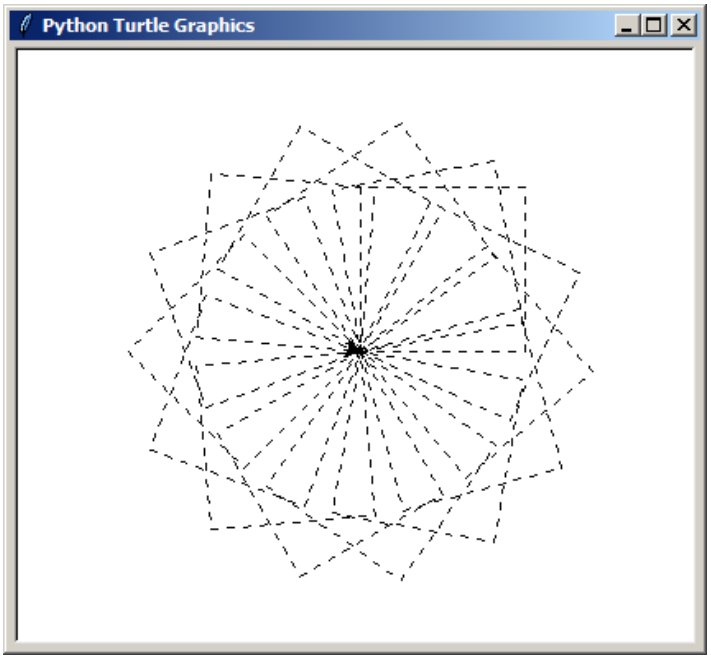
Schleifen-Konstrukte können wir auch benutzen, um z.B. gestrichelte Linien zu erzeugen.
Dazu müssen wir natürlich wissen, wie man die Schildkröte ohne Spur bewegt. Mit der Anweisung **`turtle.penup()`** wird der Zeichenstift (für die Spur) abgehoben und mit **`turtle.pendown()`** wieder aufgesetzt.
Will man nun ein Muster mit getsrichelten Linien zeichnen, dann kommt noch eine dritte Schleife hinzu, die quasi die alte Linienführung (turtle.forward(laenge) durch eine zusätzliche Muster-Erzeugung ersetzt.
Hier würde ich intuitiv lieber eine while-Schleife nehmen. Vor allem, weil die Länge ja als intern Veränderlich im Programm steht. Da will man auf alle Eventualitäten vorbereitet sein.



Um Ihnen nicht den Spaß am Programmieren und Experimentieren zu nehmen stelle ich hier nur die Erzeugung einer einzelnen gestrichelten Linie vor. Das Zusammenstellen und Einarbeiten in eigene Programme überlasse ich Ihnen.

```
# Strichel- und Nichtstrichel-Länge müssen zusammen
# einen Teiler von Länge bilden!
strichellaenge=5
nichtstrichellaenge=5
…
gesamtlaenge=0
while gesamtlaenge<laenge:
     turtle.forward(strichellaenge)
     turtle.penup()
     turtle.forward(nichtstrichellaenge)
     turtle.pendown()
     gesamtlaenge+=(strichellaenge+nichtstrichellaenge)
```

Ein Muster aus Quadraten mit Strichelmustern könnte dann so aussehen.



Aufgaben:

1. ***Erstellen Sie ein Programm, dass ein gleichseitiges Dreieck mit der Kantenlänge 75 erzeugt!***
2. ***Erweitern Sie das Dreiecks-Programm nun um die Drehung um 5° solange bis die Schildkröte mindestens einmal um sich selbst gekreist ist!***
3. ***Lassen Sie ein Sechseck mit einer Kantenlänge von 125 zeichnen!***
4. ***Erstellen Sie ein Muster aus um sich kreisenden Sechsecken, die jeweils immer um 25° zueinander verdreht sind! Das Muster-Zeichnen soll erst dann beendet werden, wenn die Schildkröte wieder exakt die Ausgangsrichtung besitzt!***
5. ***Verändern Sie das Programm von 3. so, dass nur while-Schleifen zur Anwendung kommen!***

für die gehobene Anspruchsebene:

6. ***Erstellen Sie ein Muster, wie oben, aus selbstgewählten n-Ecken! Die Linien sollen getrichelt ausgeführt werden!***
7. ***Erstellen Sie Programm, dass ein Muster aus Quadraten erstellt, bei dem der Nutzer vorher eingeben darf, wie lang die Strichel- und die Nichtstrichellinien sein sollen! (Die Begrenzung auf Teilbarkeit soll und muss hier aufgehoben und umschifft werden!)***

8.8.4. Verzweigungen

übliche Verzweigungen
z.B. Auswertung von Eingaben

Verzweigungen basierend auf Turtle-Eigenschaften

Beispiel Abfrage, ob der Stift zeichnet oder nicht (unten oder oben ist)
isdown() liefert entsprechend True oder False zurück

oder Abfrage der X- bzw. Y-Koordinaten, um z.B. Überschreitungen von Grenzen auszuwerten
Auswertung über Vergleiche

weitere auswertbare Eigenschaften der Schildkröte findet man in der Zusammenstellung der Turtle-Funktionen (→ Anweisungen, Funktionen und Methoden des Turtle-Graphik-Moduls)

Aufgaben:

1. ***Erstellen Sie ein Programm, das eine getrichelte Linie zeichnet! Der Nutzer soll vorher als Eingaben die Gesamt-Länge und die Strichel-Länge eingeben! (Die Prüfung der Exaktheit der Daten soll im Programm erfolgen und ev. Fehlerhinweise ausgegeben werden! Nur dann zeichnen, wenn die Werte ok sind.))***
2.

für die gehobene Anspruchsebene:

3. ***Statt, wie in Aufgabe 1 soll eine Strich-Punkt-Linie gezeichnet werden! Der Punkt wird hier mit einer Kantenlänge von 2 festgelegt! Die Striche müssen mindestens doppelt so lang sein!***

8.8.5. Funktionen

Nutzung von Python-eigenen Funktionen bzw. Funktionen aus importierten Modulen kein Problem
Nutzung, wie üblich mit vorgestelltem Modul-Namen oder dem speziellen Importnamen

Zusammenfassung von Turtle-Anweisungen

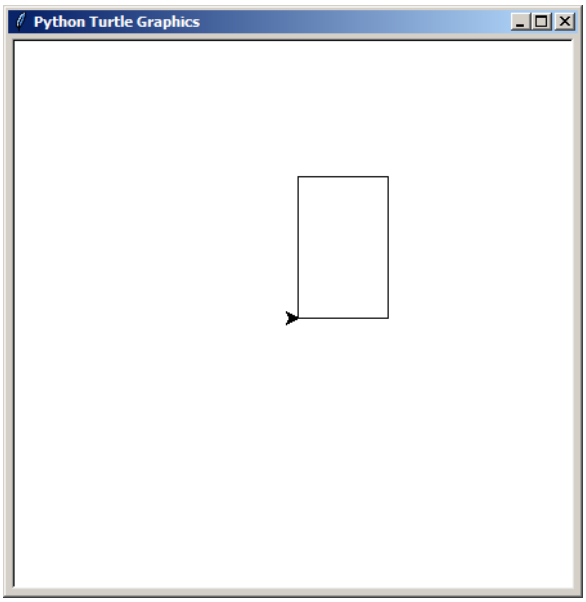
Beschreibung von Objekten z.B. ein Quadrat, n-Eck, …

übliche def-Struktur

ein Rückgabewert wird meist nicht gebraucht, kann aber – wie üblich – zurückgegeben werden

z.B. um Fehlerwerte oder Berechnungs-Ergebnisse dem übergeordneten Programm bzw. der übergeordneten Funktion mitzuteilen

```
from turtle import *

def rechteck(a,b):
    for _ in range(2):
        forward(a)
        left(90)
        forward(b)
        left(90)

rechteck(70,110)
```



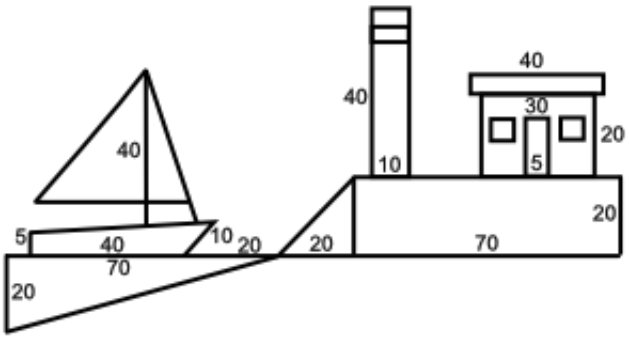
geschlossene farbige Polygonzüge

```
from turtle import *
from random import *

Farbliste=[]

def farbQuadrat(x,y,l,farbe):
    up()
    goto(x,y)
    down()
    begin_fill()
    fillcolor(farbe)
    for _ in range(4):
        forward(l)
        left(90)
    end()


for farbe in Farbliste:
    x=randint(1,400)
    y=randint(1,400)
    l=randint(1,100)
    farbQuadrat(x,y,l,farbe)
```

Aufgaben:

1. ***Lassen Sie das Haus vom Nikolaus über eine Funktion zeichnen (die Seitenlänge soll 100 betragen! Die Schildkröte soll am Schluß wieder in der Start- bzw. Ausgangs-Ausrichtung stehen! (Überlegen Sie sich, ob es Sinn macht, mit Parametern zu arbeiten!)***
2. ***Lassen Sie sich mit Ihrer Nikolaus-Haus-Funktion 5 Nikolaus-Häuser direkt nebeneinander zeichnen!***
3. ***Ändern Sie das Programm von Aufgabe 2 nun so, dass die Häuser statt des üblichen rechtwickligen Dach's nun ein gleichseitiges bekommen!***
4. ***Verändern Sie das Progamm von Aufgabe 3 nun so, dass beliebige Seitenlängen eingegeben werden können!***
5. ***Erstellen Sie ein Programm, dass mit Hilfe einer Funktion gleichSeitigesDreieck(laenge) eine Reihe von 12 Dreiecken direkt nebeneinander zeichnet!***
6. ***Machen Sie aus dem Programm von Aufgabe 5 ein neues, dass eine Reihe von 7 Dreiecken zeichnet, die immer abwechselnd nach oben und unten zeigen! Es darf nur eine Funktion gleichSeitigesDreieck(...) verwendet werden!***
7. ***Lassen Sie eine neue Programm-Version von Aufgabe 6 eine Dreiecks-Reihe zeichnen, die aus 5 Basis-Dreiecken besteht! Die Dreiecks-Spitzen sollen miteinander verbunden sein, so dass eine "Stahl-Brücke" im EIFEL-Stil entsteht! (Es darf nur die neue gleichSeitigesDreieck(...) verwendet werden!)***
8. ***Zeichnen Sie ein Sechseck aus gleichseitigen Dreiecken!***

komplexe Aufgaben:



1. *Erstellen Sie ein Programm, das die abgebildete Szene zeichnet! (Gehen Sie schrittweise vor! Verwenden Sie sinnvolle Funktionen!)*
2. *Denken Sie sich ein eigenes Muster oder eine Szene aus! Skizieren Sie diese auf ein Blatt Papier (z.B. kleinkariert) und legen Sie die Maße fest!*
   Lassen Sie die Szene vom Kursleiter abzeichnen! Setzen Sie die Szene in ein Turtle-Programm um! (Wenn das Programm funktioniert, können Sie den Mal-Effekt durch eine delay()-Funktion zwischen den Programmteilen noch verstärken!)
3. *Erstellen Sie eine Funktion wellenOrnament(), die nebenstehendes Muster erzeugt!*

4. *Gesucht ist ein Programm, dass die Turtle-Zeichenfläche mit diesem Muster umgibt!*

x.

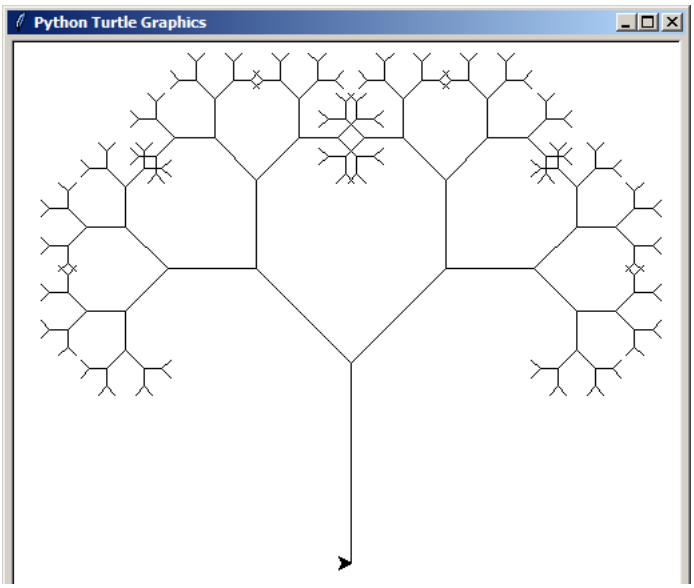
für die gehobene Anspruchsebene:

x.

8.8.6. Rekursion

```
#rekursives Zeichen eines Baumes mit Turtle
import turtle

def baum(laenge,tiefe):
    if tiefe>=0:
        turtle.forward(laenge)
        turtle.left(45)
        zweiglaenge=laenge/1.5
        baum(zweiglaenge,tiefe-1)
        turtle.right(90)
        baum(zweiglaenge,tiefe-1)
        turtle.left(45)
        turtle.backward(laenge)

### main
# Eingaben
print("Zeichnen eines selbstähnlichen Baums")
laenge=eval(input("Stamm-Länge: "))
tiefe=eval(input("Verzweigungstiefe: "))
# Beginn des Zeichnens
turtle.left(90) # Drehen zum Zeichnen nach oben
baum(laenge,tiefe)
turtle.right(90) # wieder auf die ursprüngliche Richtung drehen
```

Das Ergebnis und vor allem die Arbeit der Schildkröte beim Erstellen der Graphik ist richtig cool.



Aufgaben:

1. *Erklären Sie, warum die Schildkröte auf dem Rückweg immer die richtige Weglänge weiss!*

8.8.7. Eingaben mit der Maus

onclick(*auszuführende_Funktion*)

```
from turtle import *

def anzeige(x,y):
    print(Mausklick auf Position: ",x,",",y)

onclick(anzeige)
```

Programm nach einem Durchlauf beendet.

mit der Funktion **mainloop()** wird eine unendliche Kontrollschleife (für Eingaben / Ausgaben) erzeugt, die praktisch während der gesamten Programm-Laufzeit aktiv ist

```
from turtle import *

def anzeige(x,y):
    print(Mausklick auf Position: ",x,",",y)

onclick(anzeige)

mainloop()
```

Nutzung nun z.B. um die Schildkröte an die Klick-Position zu bewegen

```
from turtle import *

def anzeige(x,y):
    goto(x,y)

onclick(anzeige)

mainloop()
```

so erhalten wir ein kleines Zeichen-Programm

8.8.8. Und wie geht es weiter?

Die Objekt-orientierte Turtle-Programmierung folgt hinter der theoretischen Vorstellung und ersten praktischen Übungen zur Objekt-orientierten Programmierung ganz allgemein.

Windrad aus Rechtecken

```
# windrad.py
from turtle import *

def rechteck(seite): # Prozedur rechteck wird definiert
    for i in [1,2]:
        forward(seite); left(90)
        forward(seite/4); left(90)

tracer(0) # maximale Zeichengeschwindigkeit
width(2) # Zeichenstiftbreite
for i in range(1,9):
    rechteck(100)
    left(45)
```

Parkettierung (mit Rhomben)

```
# parkett.py
from turtle import *
def raute(laenge, winkel, strich_dicke, col):
    width(strich_dicke)
    color(col)
    for i in range (1,3):
       forward (laenge); left(winkel)
       forward (laenge); left(180-winkel)
tracer(0)
anzahl_reihe = 10
up(); backward(280); left(90); forward(220); right(90); down()
for i in range(1,15):
  for j in range(1,anzahl_reihe+1):
    raute(50, 45, 1, 'darkgreen')
    up(); forward(50); down()
  up(); backward(anzahl_reihe*50); right(90); forward(35); left(90); down()
```

Wald aus Bäumen

```
# baeume.py
# Wald mit Bäumen
from turtle import *
from random import random

def baum(a):
   for i in range(1,3):
       color("brown")
       fill(1)
       forward(a); left(90)
       forward(2*a); left(90)
       fill(0)
   up(); left(90); forward(2*a); left(90); forward(a); left(180); down()
   color("darkgreen")
   fill(1)
   forward(3*a); left(110)
   forward(4.39*a); left(140); forward(4.39*a); left(110)
   fill(0)
   up(); forward(a); right(90); forward(2*a); left(90); down()

tracer(0)      # max. Zeichengeschwindigkeit
up(); backward(220); left(90); forward(220); right(90); down()
# Turtle im Windwows-Fenster nach links oben setzen
a=8 # Breite Baumstamm
anzahl_x = 12  # Anzahl der Bäume in x-Richtung
anzahl_y = 8   # Anzahl der Baumreihen in y-Richtung
for j in range(1,anzahl_y+1):
  for i in range(1,anzahl_x+1):
    baum(a); up(); forward((0.5+random())*5*a); down()
  up(); backward(anzahl_x*5*a); right(90); forward(8*a); left(90); down()
```

Q: http://www.michael-holzapfel.de/progs/python/python_beisp.htm

Zeichnen eines Strauches

```
# strauch.py
# Strauch
from turtle import *
import time

def strauch(a, n):
# Die Prozedur strauch ruft sich rekursiv selber auf!
  if n>0:
     forward(a);left(30); forward(a);
     strauch(a/2,n-1); backward(a); right(30); forward(a);
     right(30); forward(a/2);
     strauch(a/2,n-1); backward(a/2); left(30); forward(a);
     strauch(a/2,n-1); backward(3*a)

tracer(0)
a=30               # Länge a
color("darkgreen")
width(2)           # Strichdicke
left(90)
strauch(a,5)       # Aufruf der Prozedur strauch
time.sleep(4)      # Programm hält 4 Sekunden an
exit()
```

Q: http://www.michael-holzapfel.de/progs/python/python_beisp.htm

Baum mit Früchten

```
# orangenbaum.py
from turtle import *

def baum(s,t):
    if t>1:
       color("brown")
       for i in range(1,3):
           fill(1)
           forward(s); left(90)
           forward(3*t); left(90)
           fill(0)
       forward(s)
       left(30)
       baum(s*0.6, t-1)
       right(55)
       baum(s*0.65, t-1)
       left(25)
       backward(s)
    elif t>=0:
       color("darkgreen")
       fill(1)
       forward(4*s); circle(2*s); backward(4*s)
       fill(0)
       color("red")
       fill(1)
       forward(3*s); circle(5); backward(3*s)
       fill(0);
       forward(s)
       left(50);
       color("brown")
       baum(s*0.6, t-1)
       right(85)
       baum(s*0.65, t-1)
       left(35)
       backward(s)

setup (width=400, height=400, startx=0, starty=0)
# Fenstergröße
title(" Orangenbaum")
# Fenstertitel

tracer(0)
left(90)
width(1)
up()
backward(150)
down()
baum(100,6)
```

Q: http://www.michael-holzapfel.de/progs/python/python_beisp.htm

Python-Stern

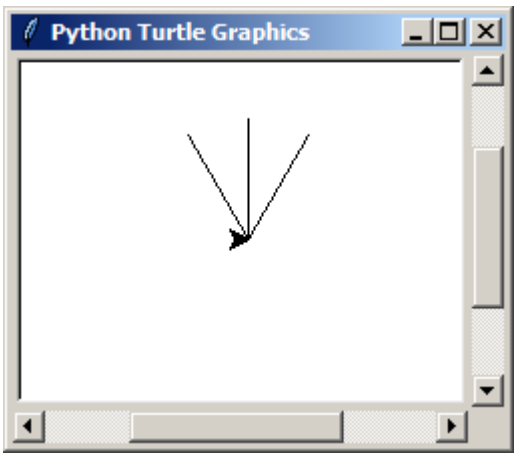
```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

8.8.9. Turteln bis zu Umfallen - rekursive Probleme schrittweise Lösen

Ein rekursives Graphik-Problem zu lösen ist nicht immer so trivial, wie es die Definition einer Rekursion suggeriert.
In Anlehnung an den gerade gezeigten Baum wollen wir nun einen Strauch rekursiv zeichnen. Er soll keinen Stamm haben und gleich mit Zweigen anfangen und statt der dichtomen (zwei-spaltigen) Teilung noch einen mittleren Zweig enthalten (also trichotom geteilt sein).
Wir fangen bei ganz einfachen Versionen an und arbeiten uns dann zu einem vollständigen Programm vor.
Im ersten Schritt erstellen wir eine einfache Funktion (noch ohne rekursive Elemente) für einen rudimentären Strauch und ein einfaches Haupt-Programm. Selbst auf Eingaben verzichten wir erst einmal um die Zeichnung ohne viel Schnick-Schnack auf den Bildschirm zu bekommen.

```
import turtle

def strauch(laenge,tiefe):
    turtle.left(30)
    turtle.forward(laenge)
    turtle.backward(laenge)
    turtle.right(30)
    turtle.forward(laenge)
    turtle.backward(laenge)
    turtle.right(30)
    turtle.forward(laenge)
    turtle.backward(laenge)
    turtle.left(30)


# Main
laenge=60
tiefe=4
turtle.left(90)  # Turtle aufrecht drehen
strauch(laenge,tiefe)
turtle.right(90)   # Turtle in die Ausgangslage zurückdrehen
```

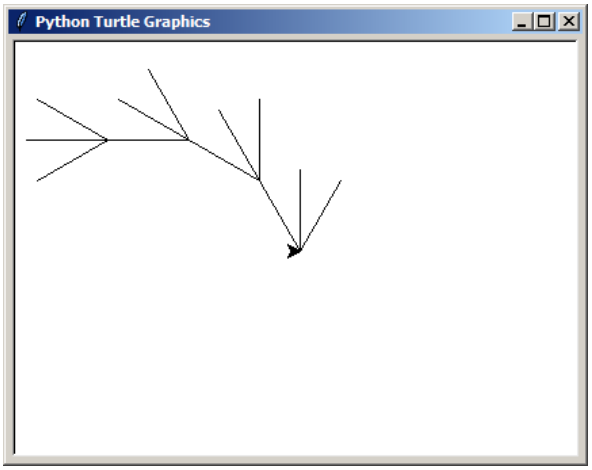
Nachdem der rudimentäre Strauch steht, können wir uns nun die Positionen heraussuchen, wo ein rekursiver Aufruf erfolgen soll. Das muss immer jeweils am Ende der 3 Zweige erfolgen. Die Länge belassen wir zuerst einmal so, wie in der ersten Rekursions-Ebene. Verkürzungen organisieren wir als Nächstes.
Was man natürlich nicht vergessen darf, ist der Rekursions-Abbruch, ansonsten zeichnet das Sysem ziemlich lange.



Wer will kann auch ersteinmal nur den ersten (linken) Zweig programmieren, damit Fehler noch gut sichtbar sind.

```
import turtle

def strauch(laenge,tiefe):
    if tiefe>0:
        # linker Zweig
        turtle.left(30)
        turtle.forward(laenge)
        stauch(laenge,tiefe-1)
        turtle.backward(laenge)
        # mittlerer Zweig
        turtle.right(30)
        turtle.forward(laenge)
        turtle.backward(laenge)
        turtle.right(30)
        # rechter Zweig
        turtle.forward(laenge)
        turtle.backward(laenge)
        turtle.left(30)


# Main
laenge=5
tiefe=0
turtle.left(90)   # Turtle aufrecht drehen
strauch(laenge,tiefe)
turtle.right(90)   # Turtle in die Ausgangslage zurückdrehen
```

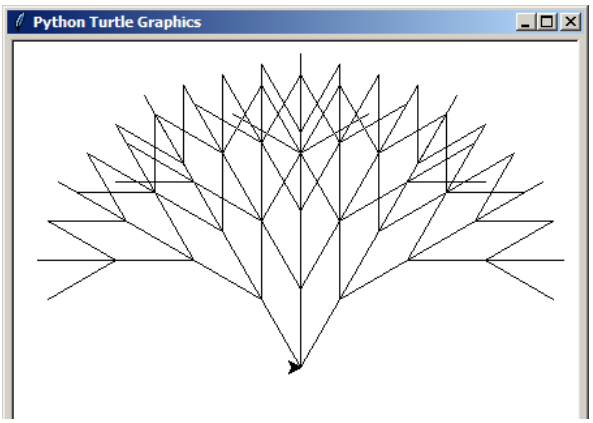
Das Graphik-Fenster zeigt saubere Linien und die Schildkröte ist auch wieder an ihrem Start-Platz angekommen. Soweit scheint die Programmierung und die Rekursion zu funktionieren.




```
import turtle

def strauch(laenge,tiefe):
    if tiefe>0:
        # linker Zweig
        turtle.left(30)
        turtle.forward(laenge)
        strauch(laenge,tiefe-1)
        turtle.backward(laenge)
        # mittlerer Zweig
        turtle.right(30)
        turtle.forward(laenge)
        strauch(laenge,tiefe-1)
        turtle.backward(laenge)
        turtle.right(30)
        # rechter Zweig
        turtle.forward(laenge)
```

```
        strauch(laenge,tiefe-1)
        turtle.backward(laenge)
        turtle.left(30)


# Main
laenge=5
tiefe=0
turtle.left(90)   # Turtle aufrecht drehen
strauch(laenge,tiefe)
turtle.right(90)   # Turtle in die Ausgangslage zurückdrehen
```

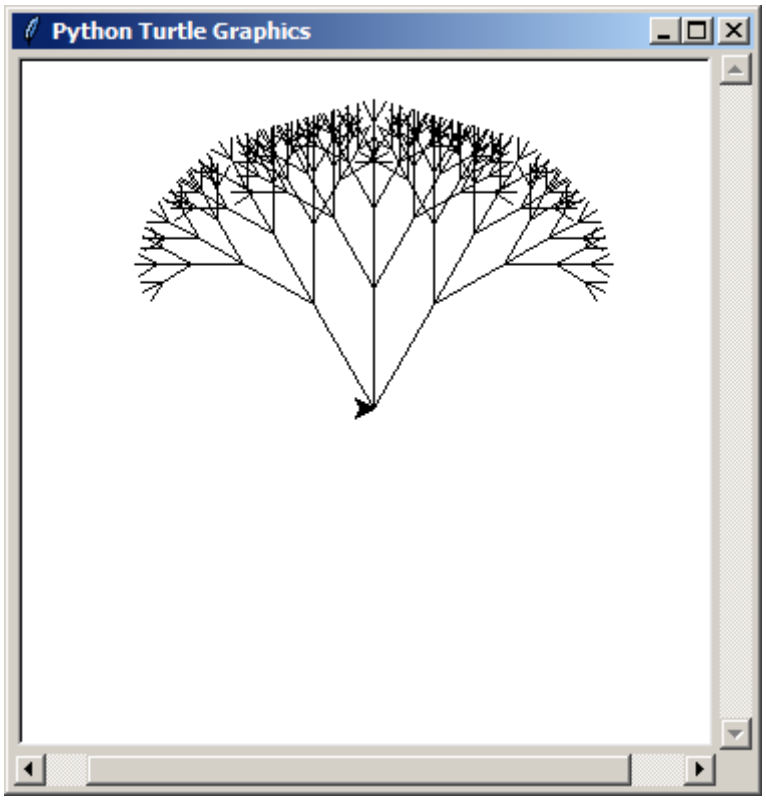
Wenn die gesamte Funktion durchdefiniert ist und es funktioniert, dann kann man noch das Rahmen-Programm anpassen. Dazu gehören sicher ordentlich abgesicherte Eingaben und die angepasste (- kürzere -) Zweiglänge für die untergeordneten Zweige.




```
import turtle

def strauch(laenge,tiefe):
    if tiefe>0:
        # linker Zweig
        turtle.left(30)
        turtle.forward(laenge)
        zweiglaenge=int(laenge/1.5)
        strauch(zweiglaenge,tiefe-1)
        turtle.backward(laenge)
        # mittlerer Zweig
        turtle.right(30)
        turtle.forward(laenge)
        strauch(zweiglaenge,tiefe-1)
        turtle.backward(laenge)
        turtle.right(30)
        # rechter Zweig
        turtle.forward(laenge)
        strauch(zweiglaenge,tiefe-1)
        turtle.backward(laenge)
        turtle.left(30)


# Main
laenge=5
tiefe=0
turtle.delay(0)  # Turtle beschleunigen
while not(laenge>=10 and laenge<=200):
    laenge=eval(input("Zweiglänge [10 .. 200]: "))
if laenge>=10 and laenge<200:
    while not(tiefe>=1 and tiefe<=100):
        tiefe=eval(input("Verzweigungen [1 .. 10]: "))
if tiefe>=1 and tiefe<=10:
    turtle.left(90)   # Turtle aufrecht drehen
    strauch(laenge,tiefe)
    turtle.right(90)   # Turtle in die Ausgangslage zurückdrehen
```

```
>>>
Zweiglänge [10 .. 200]: 60
Verzweigungen [1 .. 10]: 5
```

Ob man das Ergebnis nun als Strauch durchgehen lässt oder es eher einem Blütenstand eines Doldenblüten-Gewächs entspricht, bleibt der Phantasie des Betrachters überlassen.



Aufgaben:

1. *Ändern Sie die Funktion* strauch *so ab, dass sie mit einem vom Haupt-Programm vorgegebenen Winkel zwischen den Zweigen arbeiten kann!*

für die gehobene Anspruchsebene:

2. *Ändern Sie das Programm so ab, dass die Zweige einwenig zufällig differieren! Geht das überhaupt, oder muss der Baum / Strauch imme symmetrisch sein? Überlegen Sie sich eine begründete Anwort vor dem Lösen des Problems!*

Lösen von weiteren graphischen Aufgaben und Problemen

am Ende des Abschnittes zur Turtle-Graphik gibt es eine kleine Zusammenstellung der verschiedenen Turtle-Befehle

Zeichnen eines Labyrint's (Aaron Bies)

```
import turtle, random, math

# Zeichnet zwar kein Quadrat,
# dafür aber ein perfektes Labyrinth.
#
# Der Algorithmus ist rekusiv, also
# kann es sein, dass es nach ein paar
# Sekunden abstürtzt. Einfach Skript
# neustarten, wenn das passiert.
#
# Wenn jemand weiss, warum kein Quadrat
# rauskommt, schreibt mir bitte.

turtle.delay(2)

size = 42
scale = round(380/size)
visited = []

def generate(x=0, y=0, lor=0):
    visited.append((x,y))

    order = list(range(4))
    random.shuffle(order)

    for direct in order:
        newX            =            max(-size/2,            min(size/2,
x+round(math.cos(direct*math.pi/2))))
        newY            =            max(-size/2,            min(size/2,
y+round(math.sin(direct*math.pi/2))))

        if (newX,newY) not in visited:
            turtle.goto(newX*scale, newY*scale)
            generate(newX, newY, lor+1)
            turtle.goto(x*scale, y*scale)

generate()
turtle.hideturtle()
```

Aufgaben: (relativ einfach)

- *x. Erstellen Sie das Haus vom Nikolaus nach der klassichen Regel, das keine Linie doppelt gezogen werden darf! (Als besondere intellektuelle Herausforderung suchen wir noch das nebenstehende Haus vom Weihnachtsmann.)*
- *x. Zeichnen Sie eine Strichel-Linie, bei der die Strichel-Linien immer ein kleines Stück länger werden! (es reichen 10 Strichel!)*
- *x. Erstellen Sie ein Programm, dass 10 ineinander geschachtelte Quadrate (mit dem gleichen Startpunkt) zeichnet! Das erste Quadrat soll eine Kantenlänge von 20 Pixeln haben, die nachfolgenden sollen immer um 10 Pixel verlängert werden!*
- *x. Erstellen Sie ein "Spielfeld" für ein Tic-Tac-Toe-ähnliches Spiel aus 9 einzelnen Quadraten, die sich in den betreffenden Kanten berühren, aber nicht überschneiden! Zur Demonstration, dass es sich auch wirklich um einzelne Quadrate handelt, können Sie z.B. eine Diagonale mit einer Farbe ausfüllen.*
- *x. Schreiben Sie ein Programm, bei dem 15 Rechtecke (Start-Seitenlängen 30 und 50 Pizel) mit einer Seiten-Verlängerung um 10 Pixel ineinander (eigentlich ja auseinander) schachtelt!*
- *x. Schreiben Sie eine Funktion* linie_ohne_bewegung(länge)*, bei der eine Linie der mit der gewünschten Länge gezeichnet wird, die Schildkröte aber wieder zum Ausgangspunkt zurückkehrt!*
- *x. Zeichnen Sie mit Hilfe der Funktion von Aufgabe () einen Strahlenkranz mit dem Radius 150 Pixel!*

Aufgaben: (schon schwerer)

- *x.*
- *x. Erstellen Sie ein Programm, dass aus Quadraten ein Dreh-Muster erstellt, wobei immer abwechselnd links und rechts gezeichnet wird! (quasi ein Flügel-Effekt)*
- *x. Gesucht wird ein Programm, bei dem 12 ineinander geschachtelte Dreiecke gezeichnet werden, bei denen sich keine Kanten berühren und die Dreicke zueinander immer 20 Pixel Abstand haben! Das äußerste Dreieck soll eine Kantenlänge von 400 Pixeln haben!*
- *x. Erstellen Sie ein Muster aus 7 (gleichseitigen) Sechsecken, die zu einem Wabenmuster angeordnet sind! Das Zeichnen eines Sechseckes ist als Funktion zu realisieren!*

Aufgaben: (schwer)

x. Gesucht wird die Simulation einer Schildkröte, die sich in einem Kasten (Kantenlänge 300 x 500) bewegt!

x. Erstellen Sie ein Programm für eine Teilchen-Simulation (Kugel) in einem Gefäß (Kasten 400 x 200)! Die Teilchen-Bewegung erfolgt Zufalls-gesteuert (BROWNsche Molekularbewegung). An den Wandungen wird das Teilchen nach den Gesetzen der Physik zurückgeworfen.

Aufgaben für die gehobene Anspruchsebene: (richtig schwer)

x.

Anweisungen, Funktionen und Methoden des Turtle-Graphik-Moduls

allgemeine Hinweise / Bemerkungen

X- und Y-Positionen können Integer oder Float-Werte sein; als Rückgabe-Werte der Funktionen sind es immer Float-Werte

None bedeutet, dass der Wert / das Argument i.A. weggelassen werden kann

zulässige Farbwerte sind: ""; "yellow"; "red"; "brown"; "green", "violet", "blue", "", "" oder z.B.: für weiss: '#ffffff' od. 255 bzw. schwarz: '#000000' od. 0;

Schildkröten-Anweisung turtle. …	**Beschreibung / Leistung / Funktion**	**Beispiele / Hinweise**
addshape(*formname, objektname*)	registriert ein definiertes Objekt und macht es unter *objektname* (als Turtle) benutzbar	*formname* kann auch eine GIF-Datei sein
addcomponent(*komponente, farbstring, hintergrundfarbe*)	fügt zu einem Objekt eine komponente mit Vordergrund- und Hintergrund-Farbe hinzu	Objekt muss vom Typ "compound" sein (→ Shape)
back(*länge*)	→ backward	
backward(*länge*)	bewegt Turtle um *länge* Pixel rückwärts	
begin_fill()	Start- / Initialisierungs-Aufruf für nachfolgendes Füllen	beenden mit → end_fill(); Farbe setzen mit → fillcolor()
begin_poly()	Start- / Initialisierungs-Aufruf für nachfolgendes Polygon-Zeichnen	aktuelle Position ist erster Polygon-Punkt
bgcolor	liefert die Hintergrundfarbe des zugrundeliegenden screen-Objektes zurück	es kann auch screen.bgcolor() genutzt werden
bgpic(bildname) bgpic()	setzt den Bildschirmname mit *bildname* oder gibt ihn zurück	es kann auch screen.bgpic() genutzt werden
bk(*länge*)	→ backward	
bye()	schließt das Zeichenfenster	
circle(*radius*) circle(*radius, sektor*) circle(*radius, sektor, schritte*)	zeichnet einen Kreis mit dem angegebenem radius; *sektor* ist der gezeichnete Kreisbogen in rad; *schritte* legt die Anzahl Polygone fest, aus der der Kreis gezeichnet werden soll	*sektor* und *schritte* können auch None sein
clear	→ clearscreen	
clearscreen	löscht die aktuelle Turtle-Zeichnung	die Turtle wird nicht bewegt od. ihre Parameter verändert!
clearstamp(*stempel_ID*)	löscht den durch *stempel_ID* (stamp-ID) gekennzeichneten Turtle-Stempel	
clearstamps() clearstamps(*anzahl*) clearstamps(*-anzahl*)	löscht die aktuelle Liste der stamp-ID's bzw. die durch anzahl bestimmten ersten bzw. letzten Turtle-Stempel	
clone()	erstellt bzw. liefert einen Klon der Turtle an der aktuellen Position	
color() color(*farbstring, hintergrundfarbe*) color(*farbstring*) color(*RGBtripel*) color(*RGBtupel, RGBtupel*)	liefert Tupel aus Vorder- und Hintergrundfarbe zurück bzw. setzt Vordergrund- und ev. auch Hintergrund-Farbe	*hintergrundfarbe* ist ein *farbstring*
colormode(*colormodus*)	setzt den *colormodus* (Wert kann 1 od. 255 sein)	begrenzt die Spanne für die RGB-Anteile
degrees(*gradzahl*)	legt die *gradzahl* / Schritte für einen Vollkreis fest	Standard sind 360°
delay(*verzoegerung*) delay()	setzt die *verzoegerung* in ms oder gibt sie zurück	

Schildkröten-Anweisung turtle. …	Beschreibung / Leistung / Funktion	Beispiele / Hinweise
distance(koordinatenpaar) distance(x_position, y_position)	liefert die Entfernung zu einem anvisierten Punkt zurück	
done()	macht letzte Anweisung rückgängig	
dot(*groesse, farbe*) dot() dot(*groesse*)	zeichnet einen Punkt mit den Parametern *groesse* und *f*arbe an der aktuellen Position	*farbe* ist ein Farbstring oder ein RGBtripel *groesse* kann auch None sein
down()	→ pendown	
end_fill()	End- / Destruktions-Aufruf für Füll-Vorgang	starten mit → begin_fill(); Farbe setzen mit → fillcolor()
end_poly	End- / Destruktions-Aufruf für Polygon-Zeichen-Vorgang	aktuelle Position ist der letzte Punkt des Polygons und wird mit dem ersten verbunden (→ begin_poly)
exitonclick()	bindet die bye-Methode an einen Mausclick auf / in das Zeichenfenster	
fd(*länge*)	→ forward	
fillcolor() fillcolor(*farbstring*) fillcolor(*RGBtupel*) fillcolor(*rotwert, gruenwert, blauwert*)	gibt aktuelle Farbwerte (als *RGBtupel*) zurück oder setzt die Werte	*farbstring* kann sein: Farbnamen → s.a. oben oder ein: *RGBhexadezimalcode* (Start und Ende des Füllens mit → begin_fill() bzw. end_fill())
filling()	gibt True oder False zurück jenachdem, ob der Füllmodus eingschaltet oder nicht-eingeschaltet ist	
forward(*länge*)	bewegt Turtle um *länge* Pixel vorwärts	
get_poly()	liefert das letzte gezeichnete Polygon zurück	
get_shapepoly()	liefert das Polygon der aktuellen Turtle-Form als Koordinaten-Tupel zurück	
getcanvas()	liefert Zeichenfläche als Objekt zurück	Objekt ist ein Tkinter-Objekt und kann damit weiter verwendet werden
getpen()	liefert das Turtle-Objekt sich selbst (zeigt Speicher-Adresse des Objektes)	
getscreen()	liefert die Zeichen-Fläche als Objekt zurück (zeigt Speicher-Adresse des Objektes)	einzelne Attribute lassen sich dann ändern
getshapes()	liefert eine Liste mit den Namen der möglichen Turtle-Formen zurück	
getturtle()	liefert das Turtle-Objekt sich selbst (zeigt Speicher-Adresse des Objektes)	
goto(*position*)	→ setposition	
heading		
hideturtle()	versteckt die Turtle samt Spur (z.B. bei komplexen Zeichenvorgängen)	→ isvisible
home()	setzt Turtle wieder auf die Start- / Ausgangs-Position	entspricht: turtle.setpos(0,0)
ht()	→ hideturtle	
isdown()	gibt True oder False zurück jenachdem, ob der Stift zeichnet oder nicht-zeichnet	
isvisible		

Schildkröten-Anweisung turtle. …	Beschreibung / Leistung / Funktion	Beispiele / Hinweise
left(*winkel*)	dreht Turtle um *winkel* nach links	
listen(*dummy_x_position, dummy_y_position*)	setzt den Focus auf die Zeichenfläche; die Dummy-Argumente werden für die onclick-Methode genutzt	
lt(*winkel*)	→ left	
mainloop()	startet die Ereignis-Abfrage-Schleife des übergeordneten Objektes (screen von Tkinter)	muss die letzte Anweisung in einem Turtle-Programm sein (im Script-Modus nicht notwendig)
mode(modus) mode()	setzt den modus für die Turtle-Graphik oder liefert ihn zurück modus kann sein: "logo", "world", "standard"	"standard": Ausrichtung Ost, Drehung entgegen Uhrzeiger; "logo": Ausrichtung Nord, Drehung mit Uhrzeiger
numinput(*titel, text, vorgabe, minimum, maximum*)	erzeugt ein Popup-fenster mit dem / einem *titel* und der Eingabe-Aufforderung *text*; optional können eine *vorgabe*, das *minimum* und *maximum* für die einzugebene Zahl angegeben werden	
onclick(*funktion*)	Aufruf einer Argument-losen *funktion* beim Klicken mit der *linken Maustaste*	
onclick(*funktion, maustaste, add*)	Aufruf einer zwei-argumentigen *funktion* (Click-Position) mit einer *maustaste*;	*maustaste* ist normalerweise: 1 .. linke Maustaste 2 .. 3 ..
ondrag		
onkey(*funktion, taste*)	Aufruf einer Argument-losen *funktion* beim Loslassen einer *taste*	*taste* kann auch ein Tasten-Sybol-String z.B. "space" sein
onkeypress	Aufruf einer Argument-losen *funktion* beim Drücken einer *taste*	*taste* kann auch ein Tasten-Sybol-String z.B. "space" sein
onkeyrelease(*funktion, taste*)	Aufruf einer Argument-losen *funktion* beim Loslassen einer *taste*	*taste* kann auch ein Tasten-Sybol-String z.B. "space" sein
onscreenclick		
ontimer(*funktion, zeit*)	Aufruf einer Argument-losen *funktion*, nach einer bestimmten *zeit* in ms	
pd	→ pendown	
pen(*kategorie*)	liefert Informationen zu bestimmten Kategorien über die Turtle zurück: kategorie: "shown"; "pendown"; "pencolor"; "fillcolor"; "pensize"; "speed"; "resizemode"; stretchfactor"; "outline"; "tilt"	die Rückgabewerte sind entweder True / False oder die üblichen Über- bzw. Rückgabe-Werte / -Typen der Kategorie)
Pen		
pencolor() pencolor(*farbstring*) pencolor(*RGBtupel*) pencolor(*rotwert, gruenwert, blauwert*)	liefert aktuellen Farbwert als *RGBtupel* zurück bzw. setzt die Farbwerte	*farbstring* kann sein: Farbnamen → s.a. oben oder ein: *RGBhexadezimalcode*
pendown()	senkt den Stift zum Zeichnen ab (→ Turtle-Spur)	

Schildkröten-Anweisung turtle. …	Beschreibung / Leistung / Funktion	Beispiele / Hinweise
pensize(*dicke*) pensize()	bestimmt die *dicke* der Spur bzw. liefert die Dicke der Spur zurück	*dicke* kann auch None sein
penup	hebt den Stift ab (→ keine Turtle-Spur)	
pos		
position()	liefert die aktuelle Turtle-Position als Tupel zurück	turtlePos=turtle.pos()
pu	→ penup	
radians(*gradmass*)	setzt die Messeinheit (*gradmass*) für (die nächste Aktion(en)) auf rad fest	???
RawPen		
RawTurtle		
register_shape(*formname, objektname*)	registriert ein definiertes Objekt und macht es unter *objektname* (als Turtle) benutzbar	*formname* kann auch eine GIF-Datei sein
reset()	→ resetscreen	
resetscreen	löscht aktuelle Turtle-Zeichnung und setzt alle Turtle-Parameter wieder auf die Ausgangswerte	Turtle ist wieder auf Ausgangsposition mit allen Standardwerten
resizemode(*modus*)	*modus* kann sein: "auto", "user", "noresize"	
right(*winkel*)	dreht Turtle um *winkel* nach rechts	
rt	→ right	
Screen	Tkinter-Objekt:	
ScrolledCanvas	Tkinter-Objekt:	
screensize(*bildschirmweite, bildschirmhoehe, farbstring*)	setzt die Zeichenfläche (Turtle-Bildschirm / übergeordnetes screen-Objekt) auf eine bestimme Weite (x-Ausdehnung), Höhe (y-Ausdehnung) und Hintergrundfarbe	*farbstring* kann sein: Farbnamen → s.a. oben oder ein: *RGBhexadezimalcode* es kann auch screen.screensize() genutzt werden
seth	→ setheading	
setheading(*winkel*)	legt Orientierungs-Richtung *(*als *winkel) für die Turtle fest	im logo-Modus ist Norden bei 0°; sonst ist 0° Richtung Osten
setpos(*position*)	→ setposition	
setpostion(*position*)	setzt Turtle auf die *position* *position* ist ein Vec2D oder ein Koordinaten-Paar	setpos(20, 50) setpos((20,50))
settilangle(*winkel*)		die Ausrichtung der Turtle wird nicht geändert
setundobuffer(*schritte*)	setzt oder deaktiviert den Rückschritt-Speicher (Keller-Speicher) auf *schritte*	*schritte* kann None sein
setup(*x_pixel, y_pixel, x_start, y_start*) setup(*x_anteil, y_anteil*)	gibt die Ausdehnung des Zeichenfensters in Pixeln und die Start-Position vor; bei Angabe einer Float-Zahl wird der Anteil am Gesamtbildschirm gewählt	Standard sind 50% = 0.5 des Gesamt-Bildschirms
setworldcoordinates(*x_linksoben, y_linksoben, x_rechtsunten, y_rechtsunten*)	setzt die Zeichenflächen-Koordinaten (Runter-Diagonale)	

Schildkröten-Anweisung turtle. …	Beschreibung / Leistung / Funktion	Beispiele / Hinweise
setx(*x_wert*)	setzt *x_wert* der Turtle-Position (Horizontal-Position)	
sety(*y_wert*)	setzt *y_wert* der Turtle-Position (Vertikal-Position)	
shape(*form*)	legt das Aussehen der Turtle fest; *form* kann sein: "arrow", "turtle", "circle", "square", "triangle", "classic"	*form* kann auch None sein
Shape	Tkinter-Objekt	
shapesize() shapesize(*x_faktor, y_faktor, umriss_staerke*)	liefert die aktuelle Vegrößerungsfaktoren und die Umriss-Stärke zurück oder setzt sie (im → resizemodus = "user")	*x_faktor, y_faktor* und *umriss_staerke* sind positive Werte od. None
shapetransform(*t11, t12, t21, 22*)	transformiert die Matrix der Turtle-Form	*t11, t12, t21, 22* können auch None sein
shearfactor()	gibt oder setzt	die Ausrichtung der Turtle wird nicht geändert
showturtle()	zeigt die (unsichtbare) Turtle bzw. deren Spur an (seit letztem Unsichtbarmachen)	→ isvisible
speed() speed(*geschwindigkeit*) speed(*tempostring*)	bestimmt die *geschwindigkeit* des Turtle's; Werte von 0 .. 10 werden ausgewertet, andernfalls wird 0 gesetzt; 0 .. ohne Animation	*tempostring*: "fastest" .. 0; "fast" .. 10; "normal" .. 6; "slow" .. 3; "slowest" .. 1
st()	→ showturtle	
stamp()	hinterlässt eine Turtle-Abdruck an der aktuellen Position und liefert eine stamp_ID zurück	s.a. clearstamp
Terminator		
textinput(*titel, text*)	erzeugt ein Popup-Fenster mit dem / einem *titel* und der Eingabe-Aufforderung *text*	
tilt(*winkel*)		die Ausrichtung der Turtle wird nicht geändert
tiltangle(*winkel*)		die Ausrichtung der Turtle wird nicht geändert
title(*titel*)	setzt den *titel* des Zeichenfensters	
towards(x-position, y_position) towards(koordinatenpaar)		
tracer(*anzahl, verzoegerung*) tracer(*schalter*)	*schalter* legt fest, ob die Zeichnung mit voller Geschwindigkeit (1) ohne sichtbare Turtle oder verzögert (0) mit sichtbarer Turtle erstellt werden soll	→ delay
Turtle	liefert ein neues Turtle-Objekt (Konstruktor)	
turtles()	liefert eine Liste der Turtle's vom Bildschirm zurück	
TurtleScreen	Tkinter-Objekt:	
turtlesize		

Schildkröten-Anweisung turtle. …	Beschreibung / Leistung / Funktion	Beispiele / Hinweise
undo() undo(*anzahl*)	macht die letzte bzw. die durch *anzahl* bestimmte Menge an Turtle-Aktionen rückgängig	
undobufferentries()	liefert die Anzahl der Einträge im Rückschritt-Speicher zurück	
up	→ penup	
update	zeigt den aktuellen Bildschrim z.B. während eines tracer-Modus	→ tracer
Vec2D	Tkinter-Objekt:	
width(*dicke*) width()	bestimmt die *dicke* der Spur bzw. liefert die Dicke der Spur zurück	*dicke* kann auch None sein
window_height	gibt die Fenster-Höhe (y-Ausdehung) des Zeichenfensters zurück	
window_widht	gibt die Fenster-Breite (x-Ausdehung) des Zeichenfensters zurück	
write(*schreibobjekt, bewegt, ausrichtung, schrift*)	schreibt ein *schreibobjekt* mit der *ausrichtung* ("left"; "center", "right") und der *schrift* (*schriftname, schriftgroesse, schrifttyp*) an der aktuellen Position	Bsp. für *schrift*: "Arial, 11, "normal" *bewegt* besagt, ob die Schreib-Position auf Ausgang oder Ende des Schreib-Objektes gesetzt werden soll
write_docstringdict		
xcor()	liefert die x-Koordinate der Turtle zurück	
ycor()	liefert die y-Koordinate der Turtle zurück	

u.a. Q: https://docs.python.org/3.5/library/turtle.html (hier Dokumentation nach Kategorien!)

Default- / Vorgabe-Einstellungen für Turtle-Graphik (in turtle.cfg gespeichert)

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Es existieren verschiedene Demo-Programme / -Skripts zur Turtle-Graphik. Diese können mit:

```
python –m turtledemo
```

entpackt werden.

8.8.10. Animationen mittels turtle-Grafik

Bisher war die Dynamik unserer Zeichnungen auf die Erstellung beschränkt. Nun wollen wir uns an echte Animationen machen.

Nehmen wir als Beispiel einen Fisch. Dies könnte ein Skalar sein, der durch sein rhomische Seiten-Ansicht ein Hinkucker in jedem Aquarium ist. Wir nutzen zuerst einmal nur wenige Linien zur Veranschaulichung. Später können wir dann noch ein paar Details ergänzen.

Die Grundform könnte ein Quadrat und ein Dreieck sein. Eine andere Lösung basiert auf drei rechten Winkeln.

Da es später große und kleine Fische geben soll, definieren wir eine Funktion fisch() mit möglichen Eigenschaften. Für uns wäre das wohl die Seitenlänge und die Farbe. Auch eine Schwimm-Richtung wäre wohl angebracht.

Mein Fisch soll aus den Winkeln gebaut sein. Für diese Winkel entwickle ich zuerst auch eine Unter-Funktion. Neben Größe und Farbe interessiert uns sicher auch die Richtung.

Auch für die Winkel definieren wir Richtungen. Als Orientierung verwende ich hier die Himmels-Richtungen.

Die Winkel-Funktion soll möglichst effektiv ablaufen, da sie ja sehr häufig benutzt wird.

In welcher Zusammenstellung wir aus Winkeln einen "Fisch" machen, ist uns überlassen. Ich wähle hier ein Quadrat aus zwei Winkeln und die Schwanzflosse als ein Winkel. Da kann ich dann später vielleicht auch Fisch mit andersfarbiger Flosse erstellen.

```
from turtle import *

def winkel(laenge, richtung):
    # Richtung: von 0 bis 3 für N, O, S u. W
    # am Ende hat Turtle wieder Start-Pos. u. –Richtg.
    case richtung on:
        1:
    else: # für 0
    end
    forward(laenge)
    left(90)
    forward(laenge)
    backward(laenge)
    right(90)
    backward(laenge)
    case richtung on:

    end

# MAIN (Test-Programm)
delay(0)
winkel(100,0)
winkel(100,1)
winkel(100,2)
winkel(100,3)
```

Aufgaben:

1. ***Realisieren Sie die Winkel-Funktion!***
2. ***Testen Sie die Geschwindigkeit der Winkel-Funktion, indem Sie sie z.B. 10'000x aufrufen und dabei die benötigte Zeit messen!***
3. ***Varieren Sie die Erstellungs-Möglichkeiten für einen Winkel! Testen Sie die Leistungsfähigkeit Ihrer Varianten! Wählen Sie die schnellste Variante aus!***

für die gehobene Anspruchsebene:

4. ***Testen Sie die Leistungsfähigkeit anderer Erstellungs-Möglichkeiten für einen Fisch! Welche Variante ist warum die Günstigste?***

Meine Variante ist aus mehreren Gründen nicht sehr effektiv. Sicher haben Sie eine schnellere Variante für das Zeichnen gefunden. Diese sollten Sie nun weiter für das Erstellen eines Fisches verwenden.

```
def fisch(laenge, farbe, richtung):
    # Richtung: -1 od. 1 für links od. rechts
    pencolor(farbe)
```

An dieser Stelle darf man auch Mal hinterfragen, ob es günstig ist, bei der Winkel-Funktion immer an die Start-Position zurückzukehren oder ob man besser fährt, wenn man am Zeichen-Endpunkt stehen bleiben (und die Richtung vielleicht zurückmeldet)?
Egal, wie nun unser Fisch gezeichnet wird, jetzt kann er schon mal bewegt werden.
Für einen ersten Test nehmen wir eine Zählschleife mit

```

```

```
def vorwaerts(schritte, richtung, flaenge, ffarbe, frichtung):
    # zuerst alten Fisch löschen
    fisch(flaenge, 0, frichtung)
    # Fisch an neuer Position zeichnen
    penup()
    apos=pos().x
    if richtung == 1:
        apos+=schritte
    else:
        apos-=schritte
    goto(apos, pos().y)
    pendown()
    fisch(flaenge, ffarbe, frichtung)
```

```
# MAIN
pause=100
```

```
richtung=100
# Fisch zeichnen (Start-Situation)
laenge=50
farbe=3
richtung=1
fisch(laenge, farbe, richtung)
while true:
    delay(pause)
```

Erstellen eigener Figuren (Sprite's)

register_shape(Name, Punktliste)

Beispiel: "Schiff"
register_shape('schiff', ((0,10),(5,0),(15,0),(20,10)))

setzen der (Vordergrund-)Farbe mit:
color(Farbname) 'red', 'blue', 'black', 'green', …

für mehrere Objekte:
t1 = Turtle() erzeugt ein neues turtle-Objekt mit dem Namen t1

Zugriff über Variablennamen und dann mittels Punkt abgetrennt die zugehörigen Attribute und Methoden, z.B.:

t1.shape('schiff')
t1.goto(posx,posy)

```
from turtle import *

register_shape('schiff',((0,10),(5,0),(15,0),(20,10)))

# 1. Schiff
t1=Turtle()
t1.shape('schiff')
t1.left(90)
t1.up()
t1.color('red')
t1.goto(0,50)

# 1. Schiff
t2=Turtle()
t2.shape('schiff')
t2.left(90)
t2.up()
t2.color('green')
t2.goto(0,50)

#
for x in range(100):
```

```
    t1.goto(100-x,50)
    t2.goto(x,-50)
```

Eine Turtle-ähnliche Bibliothek ist **frog** (→ http://www.viktorianer.de/info/prog-frog.html), die sich eher spielerisch an das Problem der Graphik-Programmierung macht und deshalb mehr für jüngere Python-Programmierer geeignet ist

8.9. Musik mit python-sonic

extrenes Modul

aktuelle Version unter:
https://github.com/gkvoelkl/python-sonic

sehr gut mit dem Raspberry Pi realisierbar, hier ist die Version "sonic pi" schon in mehreren Betriebssystem-Distributionen vorinstalliert

Kopfhörer können direkt angeschlossen werden

einfacher Einstieg mit einfachen Ergebnissen möglich (wird in diesem Skript auch den wesentlichen Teil der Besprechung ausmachen)

relativ komplexes System
sehr Leistungs-fähig
für Anfänger und nicht so Noten-affinen Nutzern schnell zu / sehr kompliziert
Fehler-Findung recht schwierig (Welche Note ist wann und wie falsch?)

8.10. das Modul "pygame"

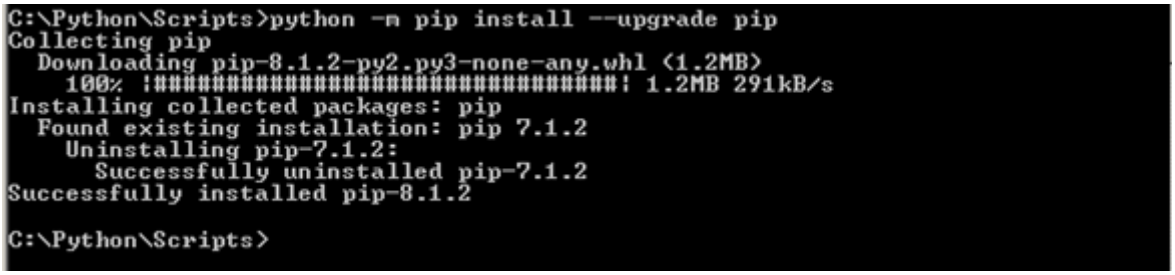
Da Python nicht direkt auf die Hardware zugreifen kann, dieses aber eigentlich für viele Programme und besonders schnelle Spiele usw. notwendig ist, bietet das Modul pygame indirekte Zugriffe und Funktionen an. So bleiben wir bei Programmieren auf Python-Ebene und die Programme können diverse Funktionen moderner Multimedia-Hardware nutzen. Pygame sorgt auch dafür, dass es uns völlig egal ist, welche konkrete Graphik- oder Sound-Karte (oder entsprechende onboard-Version) auf unserem Rechner installiert ist. Da ist Sache von pygame und braucht uns nicht zu kümmern.

8.10.0. Quellen und Installation

Als Download-Quellen sind einmal die offizielle pygame-Seite im Internet (www.pygame.org) und eine inoffizielle – aber scheinbar bestens gepflegte – Quelle für alle möglichen Module zu Python (http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame) zu nennen.
Eigentlich sollte es mit den Installations-Dateien (msi-Dateien für Windows-Systeme) eine Installtion gelingen. Bei mir funktionierten das nur mit einem Python-2-System.
Bei der letzteren Quelle sind sogenannte whl-Dateien zu downloaden, die mit dem internen pip-Progamm zu installieren sind. Das war bei mir der einzige funktionierende Weg.
Je nach Betriebssystem-Type (32- oder 64bit) und Python-Version findet man dort die passende whl-Datei.
Am einfachsten ist es, diese gleich in den Scripts-Ordner der lokalen Python-Installation zu downloaden. Wenn die Datei vom Browser woanders hin gespeichert wird, dann kopiert man sie einfach in das Verzeichnis Scripts.
Nun brauchen wir eine Eingabe-Aufforderung (Konsole) in dem Ordner. Dazu geht man im Windows-Explorer (od. Arbeitsplatz) zum Python- und dann in den Scripts- Ordner.
Mittels Hochstell-Taste (Shift) und rechter Maustaste kann man sich eine Eingabe-Aufforderung öffnen.
Nun muss man die Befehle sehr exakt eintippen, da sonst Fehlermeldungen folgen oder gar nichts passiert.
Zuerst aktualisieren wir das pip-Programm:

```
python –m pip install --upgrade pip
```

Das Ergebnis sollte dann in etwa so aussehen. Die Versionen werden sich sicher schon wieder unterscheiden.



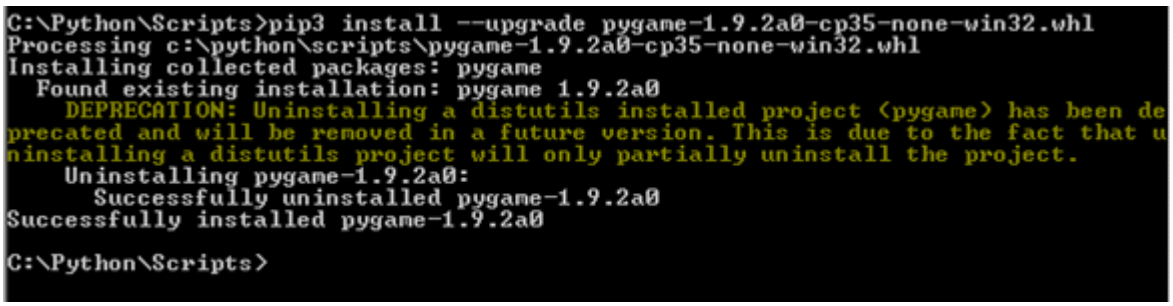
Nun können wir die passende whl-Datei installieren:

```
pip3 install pygame-……whl
```

Statt der Punkte müssen da die exakten Angaben der whl-Datei verwendet werden. Man kann sich einen Dateiname im Explorer kopieren und dann über die rechte Maustaste in der Konsole einfügen.
Kommen Fehler-Meldungen, dann bleibt nur ein erneuter Versuch. Allerdings ist jetzt (meist) ein upgrade nowendig.

```
pip3 install --upgrade pygame-……whl
```

Wenn alles geklappt hat, dann sollte eine Bestätigungs-Meldung in der Konsole erscheinen.



Jetzt steht ersten Tests nichts mehr im Weg. Ein Neustart des Rechners ist zu empfehlen, damit die integrierten DLLs ordnungsgemäß geladen werden.

Bei aktuellen Python-Installationen kann man es zuerst einmal mit:

```
pip3 install pygame
```

probieren.

8.10.1. Ausprobieren / Testen / Grundlagen

Bevor wir uns nun auf die Möglichkeiten von pygame einlassen, testen wir erst einmal die Installation. Das nochfolgende Programm erzeugt nur ein schwarzes Fenster und warten auf das reguläre Schließen über den zugehörigen Fenster-Knopf.

```
import pygame

pygame.init()
screen=pygame.display.set_mode([640,480])
aktiv=True
while aktiv:
    for event in pygame.event.get():
        if event.type==pygame.QUIT:
            aktiv=False
pygame.quit()
```

Sollte das programm nicht funktionieren, sollte man nochmals pygame installieren / upgraden. Bleibt dieses erfolglos, dann bleibt nur die große Suche nach Hilfe im Internet oder ein Überspringen dieses Kapitels.
Nach meinen ersten anfänglichen Schwierigkeiten mit pygame fand ich das unten folgende Test-Programm für die pygame-Schnittstellen von der Seite www.spieleprogrammierer.de. Auf dieser steht auch ein online-Tutorial zur Verfügung.

```
# Pygame-Modul importieren.
import pygame

# Überprüfen, ob die optionalen Text- und Sound-Module geladen werden konnten.
if not pygame.font: print('Fehler pygame.font Modul konnte nicht geladen werden!')
if not pygame.mixer: print('Fehler pygame.mixer Modul konnte nicht geladen wer-
den!')

def main():
    # Initialisieren aller Pygame-Module und
    # Fenster erstellen (wir bekommen eine Surface, die den Bildschirm repräsen-
tiert).
    pygame.init()
    screen = pygame.display.set_mode((800, 600))

    # Titel des Fensters setzen, Mauszeiger nicht verstecken und Tastendrücke wie-
derholt senden.
    pygame.display.set_caption("Pygame-Tutorial: Grundlagen")
    pygame.mouse.set_visible(1)
    pygame.key.set_repeat(1, 30)

    # Clock-Objekt erstellen, das wir benötigen, um die Framerate zu begrenzen.
    clock = pygame.time.Clock()

    # Die Schleife, und damit unser Spiel, läuft solange running == True.
    running = True
    while running:
        # Framerate auf 30 Frames pro Sekunde beschränken.
        # Pygame wartet, falls das Programm schneller läuft.
        clock.tick(30)

        # screen-Surface mit Schwarz (RGB = 0, 0, 0) füllen.
        screen.fill((0, 0, 0))

        # Alle aufgelaufenen Events holen und abarbeiten.
        for event in pygame.event.get():
            # Spiel beenden, wenn wir ein QUIT-Event finden.
            if event.type == pygame.QUIT:
                running = False

            # Wir interessieren uns auch für "Taste gedrückt"-Events.
            if event.type == pygame.KEYDOWN:
                # Wenn Escape gedrückt wird, posten wir ein QUIT-Event in Pygames
Event-Warteschlange.
                if event.key == pygame.K_ESCAPE:
                    pygame.event.post(pygame.event.Event(pygame.QUIT))

        # Inhalt von screen anzeigen.
        pygame.display.flip()


# Überprüfen, ob dieses Modul als Programm läuft und nicht in einem anderen Modul
importiert wird.
if __name__ == '__main__':
    # Unsere Main-Funktion aufrufen.
    main()
```

Q: www.spieleprogrammierer.de

Links:
http://www.spieleprogrammierer.de
www.pygame.org (die offizielle pygame-Seite; offizielle Installations-Dateien für alle Betriebssysteme)
http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame (whl-Dateien für die Installation über pip)

8.10.1. Sound mit pygame

Sound-Dateien erzeugen (bzw. aufnehmen) und abspielen
Python bzw. das Modul pygame kann mit folgenden Sound-Dateien arbeiten:

- **WAV** Wave-Dateien
-
- **MP3**
- **WMA** WindowsMedia
- **OGG** Ogg Vorbis-Dateien komprimierte Sound-Dateien mit sehr guter Dynamik und geringen Konvertierungs-Verlusten

Sound erzeugen über einen einfachen integrierten Synthesizer

8.10.1.1. Sound-Dateien abspielen

```
# Initialisierung der Soundverarbeitung
import pygame
pygame.init()
pygame.mixer.init()
…
# Auswahl der Datei
dateiname="musik.wav"

# eigentliches Abspielen -- Variante 1
soundobjekt1=pygame.mixer.Sound(dateiname)
soundobjekt1.play()
…
# erneutes Abspielen -- Variante 1
soundobjekt1.play()
…

…
# eigentliches Abspielen -- Variante 2
pygame.mixer.music.load(dateiname)
pygame.mixer.music.play()
…
# erneutes Abspielen -- Variante 2
pygame.mixer.music.load(dateiname)
pygame.mixer.music.play()
…
```

```
# Initialisierung der Soundverarbeitung
import pygame
pygame.init()
pygame.mixer.init()
…
# Auswahl der Datei
dateiname="musik.wav"

# eigentliches Abspielen -- Variante 1
soundobjekt1=pygame.mixer.Sound(dateiname)
soundobjekt1.play()
…
# erneutes Abspielen -- Variante 1
soundobjekt1.play()
…

…
# eigentliches Abspielen -- Variante 2
pygame.mixer.music.load(dateiname)
pygame.mixer.music.play()
…
# erneutes Abspielen -- Variante 2
pygame.mixer.music.load(dateiname)
pygame.mixer.music.play()
…
```

8.10.1.2. Sound-Dateien erzeugen / aufnehmen

8.10.1.3. Musik aus dem Synthesizer

8.10.2. Grafik mit pygame

Haupt-Programm (Starter) – oft main.py genannt:

```
import pygane, sys, ObjektModul

# Definitionen / Konstanten
fensterBreite=800
fensterHoehe=600

# Initialisierungen
pygane.init()
```

```
fenster=pygame.display.set_mode(fensterBreite,fensterHoehe)
spites=pygame.sprite.Group()
anzeigeObjekt= …      # aus ObjektModul
sprites.add(anzeigeObjekt)
zeit=pygame.time.Clock()

# Hauptschleife
while True:
    for event in pygame.event.get():
        if event.type==pygame.QUIT:
            pygame.quit()
            sys.exit()
        # ev. noch andere Events / Eingaben abfragen

        fenster.fill((255,255,255))
        sprites.aktualisieren()
        sprites.draw(fenster)

        pygame.display.flip()
        zeit.tick(30)
```

Ein dazu gehörendes Objekt-Modul könnte dann so aussehen:

```
class AktionObjekt(pygame.sprite.Sprite) # erbt von pygame…

    # Initialisierung
    def __init__(self, fensterBreite, fensterHoehe):
        super().__init__()
        self.fensterB=fensterBreite
        self.fensterH=fensterHoehe
        self.bild=pygame.image.load("???.png")
        self.bildFlaeche=self.image.rect()
        self.rect.center=(self.fensterB/2,self.fensterH/2)

    # Funktionen (Bewegungen, …)
    def aktualisieren(self):
        eingabeTaste=pygame.key.get_pressed()

        # Eingabe-Auswertung und Aktionen, Funktionen, …
        if eingabeTaste[pygame.K_RIGHT]:
            self.rect.x+=10
        if eingabeTaste[pygame.K_LEFT]:
            self.rect.x-=10
        if eingabeTaste[pygame.K_UP]:
            self.rect.y+=10
        if eingabeTaste[pygame.K_DOWN]:
            self.rect.y-=10

        # Ojekt einfangen
        self.rect.clamp_fp(pygame.Rect(0,0,self.fensterB,self.fensterH))
```

Dieses Modul muss dann natürlich im Haupt-Programm (statt: ***ObjektModul***) importiert werden. Bei mehreren unterschiedlichen Objekten – die andere Funktionen usw. gebrauchen – sind auch mehrere Objekt-Module notwendig.

```
…
```

8.11. Objekt-orientierte Programmierung

Die **Objekt-orientierte** Programmierung ist neben der **imperativen** und der **deklarativen** ein weiteres **Programmier-Paradigma**. Paradigmen beschreiben grundsätzliche Denk- und Arbeitsweisen. In der Informatik verstehen wir darunter vor allem den Programmier-Stil. Insgesamt haben sich schon viele verschiedene Programmier-Paradigmen herauskristallisiert. Praktisch kann man mit jeder Herangehensweise ein Programm für ein spezielles Problem erstellen. Dabei sind Aufwand und die Qualität des Programm's oft sehr unterschiedlich. Ziel ist aber immer ein Fehler-freies, effektives, gut lesbares, Redundanz-freies, modulares und Nebenwirkungs-freies Programm zu entwickeln. Gute Programmierer wählen immer eine spezielle Programmiersprache – die meist unterschiedlichen Paradigmen zugehören – um ein Problem zu lösen.
Viele Programmier-Sprachen lassen sich mit mehreren Herangehensweisen benutzen. So können wir Python imperativ und Objekt-orientiert verwenden.
In der letzten Jahren sind viele völlig neuartige Paradigmen entwickelt worden. Hier seien einige kurz genannt:

neuartige Programmier-Paradigmen

- **Komponenten-orientierte Programmierung**
- **Agenten-orientierte Programmierung**
- **Aspekt-orientierte Programmierung**
- **generative Programmierung**
- **generische Programmierung**
- **Subjekt-orientierte Programmierung**
- **Datenstrom-orientierte Programmierung**
- **Graphen-ersetzende Programmierung**
- **konkatenative Programmierung**
- **multi-paradigmatische Programmierung**
-

Bei der
beim Objekt-orientiuerten Ansatz geht es darum Eigenschaften (Attribute) und Prozeduren (Methoden), die zu einem Ding (Objekt) zugehören gemeinsam zu verwalten und zu programmieren
bei Modulen ist die Zusammenfassung eher inhaltlich gemein z.B. mathematische Funktionen oder Funktionen zur Zeit (Berechnungen, Umrechnungen, …)

hier nur grobe Verwendung und Nutzung

HelloWorld
mit 2 Texten Begrüßung und Verabschiedung

Bevor wir uns in die Objektwelt von Python begeben, wollen wir erst einmal klären mit was wir es hier zu tun haben.

Objekte sind in der Realität irgendwelche konkreten Dinge, wie z.B. der Mitschüler Friedrich, der Lehrer Müller, der Porsche von Frau Geizig oder die gelbe Blume auf dem Küchentisch.
In der Informatik werden ebenfalls Objekte der Realität modelliert. Wir schaffen uns also ein informatisches Objekt, um Informationen rund um das Real-Objekt herum elektronisch / informatisch bearbeiten zu können.
Praktisch jedes Objekt kann man einer oder mehrerer Gruppen zuordnen. In der Informatik heißen die Gruppen Klassen. Objekte mit gleichen gemeinsamen Merkmalen werden in einer Klasse bearbeitet. Im Prinzip kann man zu einer Klasse immer spezielle Mengen / Arten von Informationen verarbeiten. Gerade deshalb lohnt es sich nicht für jedes Objekt die einzelnen Informations-verarbeitenden Vorgänge einzeln zu programmieren, sondern es ist effektiver, sie irgendwie gemeinsam zu erstellen. Eine Möglichkeit so etwas zu machen, haben wir mit den Funktionen kennengelernt. Sie liefern aufgrund bestimmter übergebener Argumente einen oder mehrere Werte (Ergebnisse) zurück, egal für welche konkreten Variable oder Wert dies erfolgt.

Definition(en): Objekt (allg.)
Objekte sind Dinge der Realität, die miteinander Informationen austauschen (kommunizieren).

Definition(en): Objekt (informatisch)
Informatische Objekte sind Abbildungen real-existierender Dinge in einem Computer-Modell.

Wenn wir ein Objekt einer Klasse zugeordnet haben oder aus ihr heraus entwickeln, dann nennen wir das Objekt eine Instanz (der Klasse).
Jedes Objekt hat eine bestimmte Menge von Eigenschaften, wie z.B. einen Namen oder eine Farbe usw. usf. In Klassen werden die gemeinsamen Arten von Eigenschaften Attribute genannt. Bei der Notierung hat sich punktierte Schreibung durchgesetzt. So wird der Name von dem Lehrer-Objekt **Müller** über die Notierung **Müller.Name** erreicht.
Da alle Instanzen z.B. der Lehrer-Objekte – also der Klasse Lehrer – über einen Namen verfügen, wird der Klasse das Attribut Name zugeordnet. Geschrieben wird dann **Lehrer.Name**.
Neben den charakterisierenden Eigenschaften eines Objektes (eben die Attribute) brauchen wir noch Verfahren z.B. zum Abfragen oder Ändern des **Lehrer.Name**'s. Die Verfahren werden Methoden genannt. Praktisch gehört fast immer zu jedem Attribut einer Klasse eine setzende und eine abfragende Methode. Die beiden heißen fast immer SET und GET.
Die Notierung würde dann **Lehrer.Name.set(…)** bzw. **Lehrer.Name.get(…)** lauten. GET und SET sind also praktisch Funktionen.

Definition(en): Klassen
Klassen sind die allgemeinen Beschreibungen / Bildungs-Vorlagen für (informatische) Objekte.

Eine Klasse ist ein Bildungsschema für ähnliche / vergleichbare Objekte.

Definition(en): Attribute

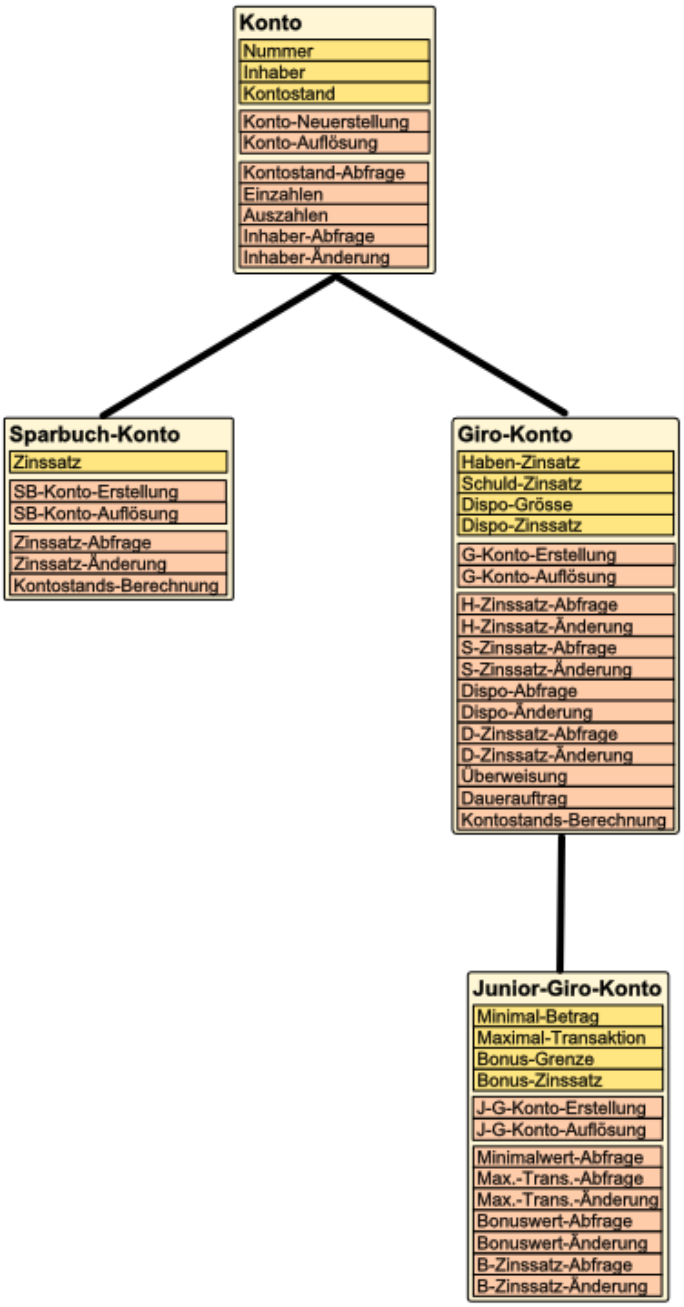
Attribute sind die individuellen Eigenschaften von Objekten.

Definition(en): Methoden

Methoden sind die Funktionen / Arbeitsmöglichkeiten / … von Objekten.

Zum besseren Verständnis und für eine genauere Übersicht werden Objekte und Klassen in verschiedenen Schemata dargestellt.

Wie das Objekt intern funktioniert, also wie es z.B. den Namen abspeichert oder beim Konto den aktuellen Konto-Stand berechnet bleibt für die Umgebung uninteressant und (meist) auch unsichtbar.



beim Objekt-orientierten Ansatz geht es darum Eigenschaften (Attribute) und Prozeduren (Methoden), die zu einem Ding (Objekt) gehören gemeinsam zu verwalten und zu programmieren
bei Modulen ist die Zusammenfassung eher inhaltlich gemein z.B. mathematische Funktionen oder Funktionen zur Zeit (Berechnungen, Umrechnungen, …)

Instanz-Variablen sind die Attribute eines Objektes

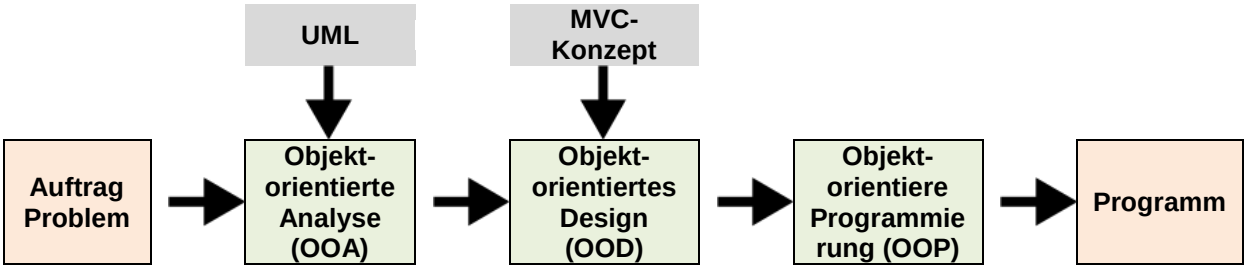
Objekt-orientierte Programmierung (OOP)

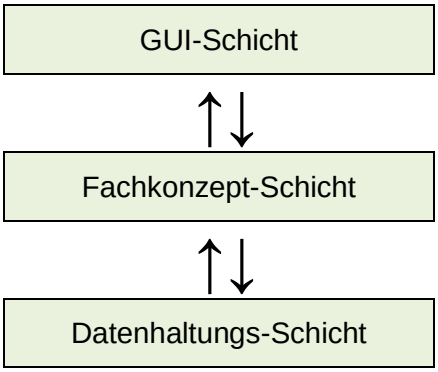
Konzepte der OOP

- **Abstraktion**

- **(Daten-)Kapselung**
- **Feedback**
- **Persistenz**
- **Polymorphie**
- **Vererbung**

Definition(en): Objekt-orientierte Programmierung (OOP)





Sicht des Nutzers auf die Daten (Daten-Eingabe und –Anzeige; Bedienung)

Datenverarbeitung; Fach-Modell;

Daten-Speicherung z.B in einer Datenbank

Design pattern – Entwurfsmuster

Analyse-Situation	Lösung / UML-Diagramm	
Gemeinsame Attribute und Methoden	**abstrakte Oberklasse**	
mehrere Klassen haben einige gemeinsame Attribute in der Oberklasse klassische Vererbungs-Situation der **Verallgemeinerung**		
spezielle Unterklasse	**konkrete Oberklasse**	
zu einer (konkreten) Klasse kommt eine Unterklasse mit		
Daten einer Beziehung festhalten **Koordination von Objekten**	**Assoziation über eine vermittelnde Klasse**	
Container / Kollektion und ihr Inhalt	**Aggregation in einer Sammlung**	
Beschreibung **Registrierung von Ereignissen**	**Aggregation Beschreibungen**	
Attribut-Werte weitergeben	**Aggregation zur Zuordnung gleicher Attribut-Werte**	

nach Q: http://www.oszhdl.be.schule.de/gymnasium/faecher/informatik/ooa-ood/designpattern.htm

8.11.x. Objekt-orientierte Programmierung mittels Turtle-Grafik

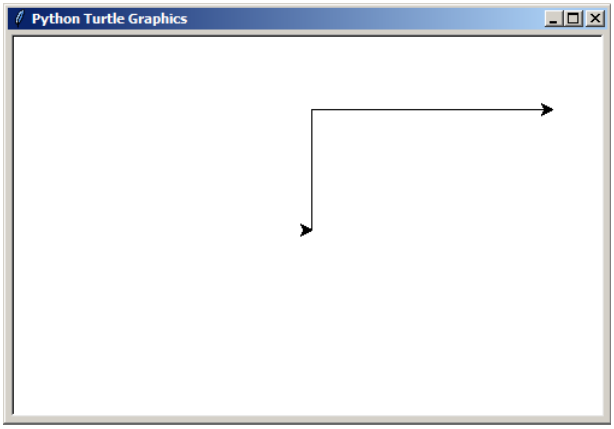
Das Modul turtle ermöglicht auch die Objekt-orientierte Programmierung von zeichnenden Schildkröten. Der Umgang mit Objekten gehört heute für eigentlich alle Module zum Standard. Moderne Programme für grafische Bedienoberflächen – wie Windows, iOS, Android oder Linux-KDE bzw. Linux-gnome – lassen sich anders gar nicht mehr benutzen. In unseren ersten Turtle-Übungen haben wir nur unterschwellig mit einem Turtle-Objekt gearbeitet (→ 8.8. Turtle-Graphik – ein Bild sagt mehr als tausend Worte). Es wurde gleich für uns automatisch angelegt und wir konnten es benutzen, ohne dass wir uns um irgendwelche "Objekte" einen Kopf machen mussten. Das war zu diesem Zeitpunkt auch sinnvoll, da wir uns einfach (und unkompliziert) die grafische Seite der Programmierung ansehen wolten.
Nun gehen wir aber einen deutlichen Schritt in Richtung moderne Programmierung.
Übrigens werden Sie merken, dass die meisten Programme aus dem Internet auch immer Objekt-orientiert programmiert sind.
Wir wollen zuerst einmal zwei Schildkröten auf dem Bildschirm erzeugen. Diese sollen dann eigenständig Bewegungen ausführen.
Zuerst brauchen wir – wie üblich – das Modul turtle, welches wir hier importieren.

```
from turtle import *

t1=Turtle()
t2=Turtle()
```

Nun erstellen wir uns zwei (unabhängige) Turtle-Objekte. Üblich ist die Notierung der Klassen-Namen mit einem Großbuchstaben beginnend.
Lassen wir das Programm an dieser Stelle laufen, sehen wir allerdings nur eine Schildkröte. Das liegt daran, dass beim Erstellen die gleichen Vorgaben benutzt wurden. Beide Schildkröten haben die gleiche Farbe und liegen an der gleichen Position. Der Konstruktor – der aus der Klassen-Vorgabe ein konkretes Objekt erzeugt hat, hatte nur diese Vorgaben.
Um nun zu zeigen, dass wir es mit zwei eigenständigen Objekten zu tun haben, können wir die eine Schildkröte mit unseren typischen Funktionen bewegen.Die Klasse Turtle wird also zweimal bemüht und die beiden Objekte t1 und t2 zu erstellen.
Allerdings müssen wir jetzt immer sagen, welche Schildkröte wir bewegen wollen. In der Objekt-orientierten Programmierung wird dazu üblicherweise die Punkt-Notierung genutzt. Wir geben zuerst das benutzte Objekt an und dann hinter einem Punkt die auszuführende Aktion.

```
from turtle import *

t1=Turtle()
t2=Turtle()

t1.goto(0,100)
t1.forward(200)
```

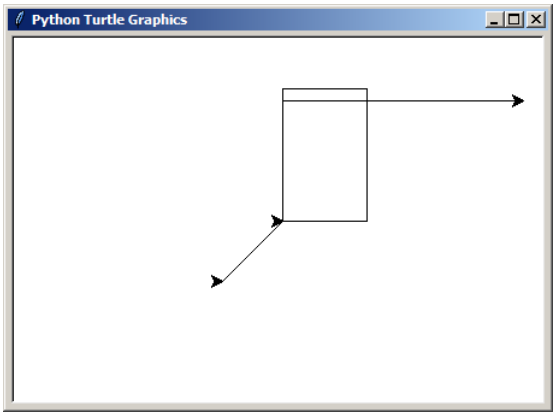
Zuerst lassen wir die Schildkröte t1 an eine andere Position wandern.
Danach bewegen wir sie ein Stück vorwärts. Die Aktionen, Funktionen usw. werden Methoden genannt. Es ergibt sich also die allgemeine Notierungs-Vorschrift **objekt.methode**. Man sieht sehr schön, dass eine Schildkröte am Koordinaten-Ursprung zurückbleibt.



Beide Schildkröten haben jetzt unterschiedliche Positionen. Diese Objekt-Eigenschaften werden unabhängig für jedes Objekt verwaltet und heißen Attribute. Jede Schildkröte hat zwar die gleichen Attribute, aber unterschiedliche gespeicherte Attribut-Werte.

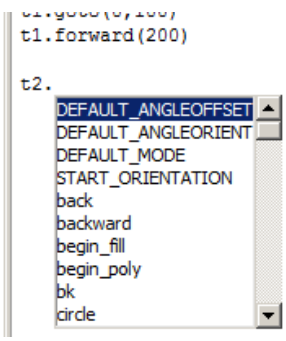
Die "zurückgebliebene" Schildkröte wollen wir nun auch testweise bewegen. Lassen wir sie z.B. ein Rechteck mit einer unserer getesteten Funktionen (→ 8.8.5. Funktionen) zeichnen.
Zum deutlichen Abgrenzen setzen wir sie auch noch auf einen anderen Startplatz.
Fraglich ist ja eigentlich schon, welche Schildkröte bewegt wird. Ist es die letzte angesprochene?
Die Überraschung ist perfekt, mit einem Mal haben wir drei Schildkröten.
Die Schildkröte t2 wurde zwar schräg nach unten bewegt, aber das Rechteck ging nicht von dieser Position und nicht von dieser Schildkröte aus.
Ganz offensichtlich fuktioniert zwar unsere Rechteck-Funktion, aber nicht Objekt-bezogen.
Um einen Objekt-Bezug hinzubekommen, müssen unsere Funktionen speziell definiert werden. Als ersten Parameter übergeben wir einen Objekt-Bezug (**rot** hervorgehoben). Der wird üblicherweise self genannt. Mit diesem self-Objekt (**blau** hervorgehoben) werden dann alle Aufgaben innerhalb der Funktion – besser jetzt Methode genannt – erledigt.
Nun befriedigt uns das Ergebnis auch hinsicht des erwarteten Ergebnisses.

```
from turtle import *

def rechteck(a,b):
    for _ in range(2):
        forward(a)
        left(90)
        forward(b)
        left(90)

t1=Turtle()
t2=Turtle()

t1.goto(0,100)
t1.forward(200)


t2.goto(-50,-50)
t2.rechteck(70,110)
```

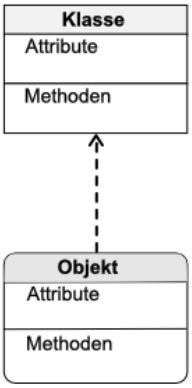



```
...

def rechteck(self,a,b):
    for _ in range(2):
        self.forward(a)
        self.left(90)
        self.forward(b)
        self.left(90)

...
```

Tippt man langsam genug – bzw. wartet man einen kleinen Augenblick nach der Eingabe des Punktes, dann zeigt uns IDLE aufeinmal ein kleines Auswahlwahl-Fensterchen an. Hier können wir die verfügbaren Methoden



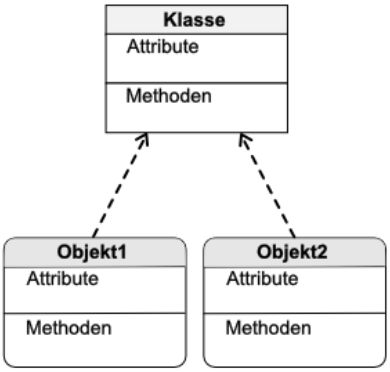
Informatisch betrachtet haben wir es bei der Objekt-orientierten Programmierung ja immer mit Klassen und Objekten zu tun. Mit dem Modul turtle bekommen wir eine Klasse Turtle zur Verfügung gestellt, die alle Attribute (Eigenschaften) und Methoden (Funktionen) enthält.



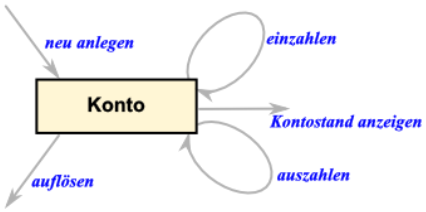
Von der Klasse Turtle können wir uns nun beliebig viele Schildkröten zum Zeichnen ableiten. Jede Schildkröte hat nun quasi einen Namen, da beim Ableiten eines Turtle-Objektes immer ein Variablen-Name angegeben werden musste. Über diesen Namen ist das Objekt im folgenden Programm ansprechbar. Der Varibalen- bzw. Objekt-Name steht immer vor dem Punkt in einer Objekt-Anweisung.

Jedes Objekt bekommt einen Satz eigener Attribute. Das sind z.B. die x- und y-Position oder auch die Farbe. Schließlich soll sich ja auch gjedes Objekt frei auf der Zeichenfläche bewegen können.

Die abgeleiteten turtle-Objekte haben Zugriff auf den gesamten Methoden-Satz. Jedes Turtle-Objekt kann sich unabhängig von den anderen vorwärts oder rückwärts bewegen oder auch drehen. Die für das jeweilige Objekte gewünschte Methode wird immer hinter dem Punkt der Objekt-Anweisung notiert.



8.11.x. Klassen – selbst erstellen



am Besten vorher Klassen-Diagramme erstellen bzw. vorhandene benutzen

geeignete Form UML-Diagramme

stellen standardisiert Klassen mit ihren Attributen und Methoden sowie die Beziehungen zu anderen Klassen dar

nebenstehend für eine minimalistische Konto-Klasse

	Konto	***Klassenname***
-	Kontostand	***Attribute***
+C	neu	***Methoden***
+!	zeigeKontostand(Kontostand)	
+!	einzahlen(Betrag)	
+!	auszahlen(Betrag)	
+D	löschen	

verbesserte bzw. erweiterte Konto-Klasse

	Konto	***Klassenname***
+	Kontenzähler	***Attribute***
-	Kontostand	
+	Dispo	
-	Inhaber	
-	Berechtige	
+C	neu	***Methoden***
+!	zeigeKontostand(Kontostand)	
+!	zeigeInhaber(Inhaber)	
+!	ändereInhaber(neuerInhaber)	
+!	berechtigt(Akteur)	
+!	einzahlen(Betrag)	
+!	auszahlen(Betrag)	
+D	löschen	

8.11.x.1. Erstellen einer Klasse

Erzeugen einer Instanz / eines Objektes im Programm dann mit

instanzname=klassenname(attribute)

mit **pass** wird für die Phase der Programm-Entwicklung temporär die Notwendigkeit von inhaltlichen Quellcode ausgeschaltet
das gilt für Klassen und Methoden

Normalerweise sind Attribute und Methoden einer Klassen von außen sichtbar bzw. nutzbar. Zugrif immer über den Namen der Instanz und dem Punkt-getrennten entsprechenden Attribut bzw. dem Methodennamen

Unsichtbare Klassen-Bestandteile werden mit **private** deklariert.
Gibt man direkt vor dem Namen einen Unterstrich an, dann ist der Klassenbestandteil als protected gekennzeichnet. Dieses gilt nur für Programmierer (als Konvention / Empfehlung), dem Interpreter ist dies egal. Für ihn sind die Attribute und Methoden immer (über die Instanz) sichtbar

Konto-Beispiel (Schritt 1)

```
class Konto:
    pass
```

8.11.x.1.1. der Konstruktor

```
class Klassenname():
    def __init__(self):
      pass
```

hat eine Klasse keinen Konstruktor, dann wird der Konstruktor der Oberklasse aufgerufen
man kann aber auch den Konstruktor der Oberklasse explizit aufrufen und ausgewählte Eigenschaften etc. verändern / überschreiben

```
class Klassenname(Objekt):
    def __init__(self):
        Objekt.__init__(self)
```

der Aufruf – und damit das Erzeugen eines (konkreten) Objektes erfolgt dann im Programm mit:

objektname = Klassenname()

Wenn man sich schon auf der Ebene des Konstruktors über die unbedingt zu definierenden (initialen) Variablen und ev. auch Werte schon klar ist, dann kann man diese gleich mit in die Vordefinition des Konstruktors einfließen lassen.
Grundlage dafür könnte z.B. ein gut ausgearbeitetes UML-Diagramm sein.

Konto-Beispiel (Schritt 2)

```
class Konto:
    def __init__(self,inhaber,betrag,autorisiert=["Banker"]):
        pass
```

Diese Vordefinition – immer gut am pass zu erkennen, muss dann später unbedingt mit Leben – sprich: Programm-Text – ausgefüllt werden.
Dazu müssen wir aber erst mal klären, wo diese Werte hin gehören.

8.11.x.2. Attribute einer Klasse

es wird zwischen **Klassen-Attribute**n und **Instanz-Attribute**n unterschieden
Klassen-Attribute werden für eine Klasse nur einmal angelegt und gelten für die Klasse allgemein bzw. für alle Instanzen gemeinsam
eine typische Verwendung ist das Zählen der Instanzen, die gerade von einer Klasse erzeugt / gehändelt werden
Klassenattribute folgen direkt im class-Block einschließlich der Zuweisung eines Initial-Wertes

Instanz-Attribute sind nur innerhalb der Instanz gültig,
ein Attribut der einen Instanz eines Klassenobjektes ist unabhängig vom gleichnamigen Attribut einer anderen Instanz der selben Klasse

public-Attribute sind von außen über die Punkt-Schreibung benutzbar

private-Attribute sind nur innerhalb einer Instanz benutzbar / sichtbar; ein Zugriff von Außen muss über darauf abgestimmte Methoden (z.B. GET- und SET-Funktionen) realisiert werden
in Python wird ein Attribut **private**, wenn man vor den Namen zwei Unterstriche schreibt

Attribute mit nur einem Unterstrich sind **protected**, sie sind praktisch public, aber Programmierer sollten nicht auf diese Zugreifen; die protected-Klassifizierung ist also nur eine Empfehlung, kein echtes Statement

ev. als Beispiel dummer und schlauer Melder aus "Python für Kids" S. 387 ff.

Schreibung der Variable	Status-Bezeichnung	Bedeutung / Eigenschaften
name	public	von außen sichtbar (lesbar und schreibar) innerhalb der Klasse lesbar und schreibbar
_name	protected	von außen sicht (lesbar und schreibar), ABER Zugriff durch Programmierer der Klasse nicht gewollt innerhalb der Klasse lesbar und schreibbar
__name	private	von außen nicht sichtbar (nicht lesbar und nicht schreibbar) innerhalb der Klasse lesbar und schreibbar

ev. Objekt-Hierarchie BlackBox mit verschiedenen inneren (nach außen) unsichtbaren Funktionen / Reaktionen
ähnlich Skalierungs-Objekt / -Klasse aus "Raspberry Pi programmieren" S.85 ff.

Konto-Beispiel (Schritt 2)

```
class Konto:
    kontenzaehler=0

    def __init__(self,inhaber,betrag,autorisiert=["Banker"]):
        self.__inhaber=inhaber
        self.__kontostand=betrag
        self.__autorisiert=autorisiert
        Konto.kontenzaehler+=1
        self.__kontonummer=Konto.kontenzaehler
```

8.11.x.3. Methoden einer Klasse

Methoden sind nichts anderes als Funktionen innerhalb einer Klasse. Sie sind auch nur innerhalb der Klasse sichtbar und nutzbar.
Will man eine Klassen-Methode nutzen, dann geht das immer nur über die erzeugte Instanz (das erstellte Objekt).
Ein Zugriff von außen kann verhindert werden, wenn die Methode als **privat** deklariert wurde. Eine private Methode kann nur innerhalb einer Klasse benutzt werden.
Alle Methoden müssen mindestens und damit als ersten Parameter eine Referenz auf sich selbst als aufrufendes Objekt enthalten.

def methode(self{, parameter})

def methode(self{, parameter=wert})

Konto-Beispiel (Schritt 3)

```
class Konto:
    kontenzaehler=0
    kontennummer=0

    def __init__(self,inhaber,betrag,autorisiert=["Banker"]):
        self.__inhaber=inhaber
        self.__kontostand=betrag
        self.__autorisiert=autorisiert+inhaber
        Konto.kontenzaehler+=1
        Konto.kontennummer+=1
        self.__kontonummer=Konto.kontennummer

    def abfragen(self):
        pass

    def einzahlen(self):
        pass

    def auszahlen(self):
        pass
```

Nun werden die einzelnen Methoden durch Quelltext untersetzt. Ev. sollten auch noch durch Fluß- oder UML-Diagramme die notwendigen Parameter geklärt werden. Einmal programmierte Funktionen sollten dann später auf der Ebene der Parameter-Listen nicht mehr geändert werden. Bei kleinen Projekten überschaut man die verschiedenen Stellen mit den zu ändernden Parameter-Listen in Methoden-Aufrufen noch, aber bei großen Projekten wird schnell mal eins übersehen.

```
…
    def abfragen(self):
        print("===> Kontostand: ",self.__kontostand
…
```

Hier ist quasi auch eine wesentliche Entscheidung über das Ausgabe-Prinzip gefallen. Wir geben in diesem Beispiel immer gleich in den Methoden aus. Praktisch kann das auch im

Haupt-Programm erledigt werden. Dann hätte die abfragen-Methode z.B. so asussehen können:

```
…
    def abfragen(self):
        return self.__kontostand
…
```

Konto-Beispiel (Schritt 4)

```
…
    def einzahlen(self,betrag):
        if betrag > 0:
            self.__kontostand+=betrag
            print("===> Kontostand: ",self.__kontostand)
        else
            print("===> Fehler! (Kein gültiger Betrag!)")
…
```

Konsequenterweise sollten wir hier auch gleich unsere abfragen-Methode benutzen, statt die Ausgabe hier wieder zu organisieren.

```
…
    def einzahlen(self,betrag):
        if betrag > 0:
            self.__kontostand+=betrag
            self.abfragen
        else
            print("===> Fehler! (Kein gültiger Betrag!)")
…
```

```
…
    def auszahlen(self,betrag,initiator):
        if initiator in self.__autorisiert:
            if betrag <= self.__kontostand:
            self.__kontostand-=betrag
            print("===> AUSZAHLUNG: ",betrag)
        else
            print("===> Fehler! (Kein gültiger Betrag!)")
        print("===> Kontostand: ",self.__kontostand)
…
```

8.11.x.4. Speicher-Bereinigung

8.11.x.4.1. der Destruktor

def __del__()

in den Rumpf werden die Anweisungen geschrieben, die vor / beim Löschen des Objektes (der Instanz / Referenz) erledigt werden sollen / müssen

del(instanz**)**

Konto-Beispiel (Schritt 5)

```
…
    def __del__(self):
        if betrag > 0:
            self.auszahlen(self)
        else
            print("===> Fehler! (Kontenausgleich notwendig!)")
…
```

Aufgaben:

1. *Verbessern Sie das Konten-Beispiel so, dass mehrfache (gleiche) Fehler-Meldungen in eine eigene Methode ausgelagert werden!*
2. *Erweitern Sie das Konto-Programm so, dass alle Fehler-Meldungen über eine Methode ausgegeben werden!*
3. *Überlegen Sie sich, wie man alle Fehler-Meldungen in einer (neuen) Methode unterbringen könnte! Die Methode soll immer den gerade passenden Fehler ausgeben!*
4. *Regionaliseren Sie das Programm für den englischen Sprachraum!*
5. *Erstellen Sie eine Klasse "Auto" unter Beachtung der folgenden Vorgaben und geforderten Methoden! Testen Sie alle Methoden in einem kleinen Test-Programm! Nach einem erfolgreichen Test können einzelne Methoden aus dem Test-Programm auskommentiert werden – müssen aber wieder nutzbar gemacht werden können!*
   Ein Auto hat die Merkmale Kennzeichen, Verbrauch (gemeint pro 100 km), TachoStand, TankMaxVol (maximales Tank-Volumen) sowie TankAktVol (aktuelles Tank-Volumen).
   Beim Erstellen eines Auto's gehen wir davon aus, dass es ungefahren und unbetankt ist.
   Die Klassen-Definition soll alle (sinnvollen) Geter und Seter (Gib- und Setz-Methoden) enthalten.
   Als spezielle Möglichkeiten sollen einem späteren Hauptprogramm die folgenden Funktionen zur Verfügung stehen: StatusAnzeige() (als Nutzerfreundliche Anzeige aller Attribute in einer zweizeiligen Ausgabe), tanken(volumen) und fahren(kilometer).
   Für die Maximal-Bewertung werden auch Fehler-Meldungen in den Methoden erwartet, z.B. wenn versucht wird, zuviel Treibstoff einzufüllen usw. usf.! Eine erste Klassen-Definition kann auch noch mit z.B. negativen Tank-Volumen usw. arbeiten! Das Haupt-Programm liefert kontrollierte Werte für Eingaben / Parameter!
   Extra-Bewertung: die Tanken-Methode so gestalten, dass nicht gebrauchter Treibstoff an das Hauptprogramm zurückgegeben wird! Die Anzeige soll über das Haupt- bzw. Test-Programm erfolgen!

für die gehobene Anspruchsebene:

6. *Regionaliseren Sie Ihr Konto-Programm für den französischen oder spanischen oder … Sprachraum (mit lateinischen Buchstaben)! (Lassen Sie sich ev. von einem anderen Kursteilnehmer die Texte und / oder Stichworte) geben!*

Projekt-Aufgaben:

1. ***Erstellen Sie ein Menü-gesteuertes Programm nach unten aufgezeigten Beispiel mit einer eigenen / geänderten Klassen-Konstruktion!***
2. ***Gebraucht wird eine praktisch nutzbare Klassen-Definition (einschließlich Attributen, Methoden etc.) für ein Adressbuch! Im Adressbuch sollen Name, Vorname, Geburtsdatum, eMail und Telefonnummer gespeichert werden. Weiterhin soll die Klasse die Anzahl der Kontakte ausgeben können und eine Anzeige machen, wenn de Kontakt heute Geburtstag hat!***
3. ***Erstellen Sie ein kleines Testprogramm, um die Klasse mit ausgedachten Daten auszuprobieren!***

für die gehobene Anspruchsebene:

4. ***Erweitern Sie das Adressbuch um eine Vorwarnung, wenn jemand morgen bzw. übermorgen Geburtstag hat!***
5. ***Erstellen Sie ein Menü-gesteuertes Haupt-Programm für die Adressbuch-Klasse!***

bank.py

```
class Bankkonto:
   """Einfache Bankkonto-Klasse"""

   def __init__(self,startbetrag):
      """Konstruktor: erzeugt Bankkonto"""
      self.kontostand = startbetrag

   def einzahlung(self, betrag):
      self.kontostand = self.kontostand + betrag

   def auszahlung(self, betrag):
      self.kontostand = self.kontostand - betrag

   def anzeigen(self):
      print self.kontostand

# ausprobieren
konto1 = Bankkonto(100)
konto1.anzeigen()
konto1.einzahlung(200)
konto1.anzeigen()
konto1.auszahlung(125)
konto1.anzeigen()
print konto1.__doc__
```

Q: http://www.wspiegel.de/pykurs/kurs_index.htm

Beispiel 1

```
"""
direkter Dialog zwischen zwei Instanzen unterschiedlicher Klassen über deren Methoden
Nur client ist hier wirklich aktiv, server reagiert nur
Die melde-funktion von server dient hier nur zur Kontrolle
"""

class server:
  def __init__(self, wert=0):
    self.data = wert
def neu(self, wert):
  self.data = wert
def melde(self):
  return self.data

"""
server macht hier das Einfachste vom Einfachen
er merkt sich nur eine Zahl
"""

class client:
  def __init__(self, wert):
    self.wert = wert
  def setze(self):
    s.neu(self.wert)
```

```
  def frage(self):
    print s.melde()

s = server()
print s.melde()    # server in Grundstellung (wert = default)
c1 = client(123)

# nun sind zwei Instanzen geschaffen, die miteinander reden können

c1.setze()
# jetzt hat c1 wert von server neu gesetzt

s.melde()
```

Beispiel 2

```
"""
direkter Dialog zwischen Instanzen unterschiedlicher Klassen über deren Methoden
Auch hier ist nur client wirklich aktiv, server reagiert nur
Die melde-funktion von server dient nur zur Kontrolle
"""

class server:
  def __init__(self):
    self.data = []
  def hinzu(self, wert):
    self.data.append(wert)
  def melde(self):
    return self.data

"""
server macht hier schon mehr, als im Beispiel 1, er merkt sich die Zahlenwerte aller ange-
schlossenen client
aus Vereinfachungsgründen können diese Zahlenwerte durch client nach der Erstmeldung
nicht nochmal geändert werden
"""

class client:
  def __init__(self, wert):
    self.wert = wert
    s.hinzu(wert)
  def melde(self):
    return self.wert

s = server()
print "Wertesammlung in server vorher: ", s.melde()    # server - Liste noch leer


clients = []
for i in (6, 5, 100, 19, 27):
  c = client(i)
```

```
  clients.append(c)

print "Wertesammlung in server nachher:", s.melde()

"""
wo sind eigentlich die 5 clients geblieben, die wir in der for - Schleife erzeugt haben? Sie
liegen als Feld von Adresszeigern in der Liste clients und könnten dort jederzeit weiterver-
wendet werden. Das ginge dann so:
"""

for i in range(0, 5):
  print str(i+1) + ". client hat den Wert:", clients[i].melde()

"""
Und eben dies könnte doch auch die server - Klasse verwalten!
"""
```

Q: http://www.way2python.de/

Beispiel 3

```
"""
direkter Dialog zwischen Instanzen unterschiedlicher Klassen über deren Methoden
Hier werden sowohl client als auch server aktiv
die clients hinterlegen im Server ihre Adresse und werden durch server auf ihren Zustand
befragt.
"""

class server:
  def __init__(self):
    self.data = []
  def hinzu(self, adr):
    self.data.append(adr)
  def abfrage(self):
    werte = []
    for i in self.data:
      werte.append(i.melde())
    return werte

"""
server macht hier noch mehr, als im Beispiel 2, er merkt sich die Adressen aller angeschlos-
senen client
Zur Abfrage holt er sich die aktuellen Werte der clients, wenn es soweit ist.
Hier können die Zahlenwerte der clients nach der Erstmeldung jederzeit geändert werden
"""

class client:
  def __init__(self, wert):
    self.wert = wert
    s.hinzu(self)
    # jetzt ist es passiert, hier geht die client-adr in die Liste und nicht der Wert
  def melde(self):
```

```
    return self.wert
  def setzneu(self, wert):
    self.wert = wert

s = server()
print "Wertesammlung in server vorher: ", s.abfrage() # server - Liste noch leer

clients = []
for i in (6, 5, 100, 19, 27):
  c = client(i)
  clients.append(c)

print "Wertesammlung in server nachher:", s.abfrage()

clients[3].setzneu(4000)
print "Wertesammlung in server, nach einer Änderung:", s.abfrage()
```

Q: http://www.way2python.de/

```
# -----------------------------------------------------
# Dateiname: konto.py
# Modul mit Implementierung der Klasse Konto. Sie wird von
# der Klasse Geld abgeleitet und modelliert ein Bankkonto
#
# Objektorientierte Programmierung mit Python
# Kap. 10
# Michael Weigend 20.9.2009
#-----------------------------------------------------
import time
from geld2 import Geld
class Konto(Geld):
    """ Spezialisierung der Klasse Geld zur Verwaltung eines Kontos

        Öffentliche Attribute:
           geerbt: waehrung, betrag, wechselkurs

        Öffentliche Methoden und Überladungen:
           geerbt: __add__(), __cmp__(), getEuro()
           ueberschrieben: __str__()
           Erweiterungen:
             einzahlen(), auszahlen(), druckeKontoauszug()
    """
    def __init__(self, waehrung, inhaber):
        Geld.__init__(self,waehrung, 0)              #1
        self.__inhaber = inhaber                     #2
        self.__kontoauszug = [str(self)]             #3

    def einzahlen(self,waehrung, betrag):            #4
        einzahlung = Geld(waehrung,betrag)
        self.betrag =(self+einzahlung).betrag        #5
        eintrag = time.asctime()+ ' ' + str(einzahlung)+ \
                ' neuer Kontostand: ' + self.waehrung + \
                format (self.betrag, '.2f')
        self.__kontoauszug += [eintrag]              #6

    def auszahlen(self, waehrung, betrag):
        self.einzahlen(waehrung,-betrag)

    def druckeKontoauszug(self):                     #7
        for i in self.__kontoauszug:
            print(i)
```

```
        self.__kontoauszug = [str(self)]

    def __str__(self):                                  #8
        return 'Konto von ' + self.__inhaber + \
                ':\nKontostand am ' + \
                time.asctime()+ ': '+ self.waehrung + ' ' +\
                format (self.betrag, '.2f')
```

```
# ------------------------------------------------
# Dateiname: geld2.py
# Klasse Geld mit Überladung der Operatoren +, <, >, ==
# Objektorientierte Programmierung mit Python
# Kap. 10
# Michael Weigend 20.9.2009
#-------------------------------------------------
class Geld(object):
    wechselkurs={'USD':0.84998,
                 'GBP':1.39480,
                 'EUR':1.0,
                 'JPY':0.007168}

    def _berechneEuro(self):                            #1
        return self.betrag*self.wechselkurs[self.waehrung]

    def __init__(self, waehrung, betrag):
        self.waehrung=waehrung
        self.betrag=float(betrag)

    def __add__ (self, geld):                           #2
        a = self._berechneEuro()
        b = geld._berechneEuro()
        faktor=1.0/self.wechselkurs[self.waehrung]
        summe = Geld (self.waehrung, (a+b)*faktor )
        return summe

    def __lt__(self, other):
        a = self.getEuro ()
        b = other.getEuro ()
        return a < b

    def __le__(self, other):
        a = self.getEuro ()
        b = other.getEuro ()
        return a <= b

    def __eq__(self, other):
        a = self.getEuro ()
        b = other.getEuro ()
        return a == b

    def __str__(self):
        return self.waehrung + ' ' + format(self.betrag, '.2f')
```

8.11.x.6. eine "Auto"-Klasse

```
class Auto:

# Konstruktor
    def __init__(self, ):
        self.Name=name
        self.Kennzeichen=kennzeichen
        self.TankVolumen=tankvolumen
        self.Verbrauch=
…
```

8.11.x.6.1. Erweiterung der "Auto"-Klasse um LKW's

8.11.x.7. eine "Personen"-Klasse

Für eine Personen-Datenbank wird eine Klasse "Person" gebraucht. Diese soll dann später in einer Verwaltung für Familien-Betreuung genutzt werden.
Wir gehen dieses Mal in etwas größeren Schritten vor und erläutern die einzelnen Schritte nicht mehr zu ausführlich.
Zuerst erstellen wir uns die klassische Struktur aus Klassen-Defintion, Konstruktor und einem einfachen Test-Programm. Das Test-Programm wir immer nur schnell erweitert, um die neuen Attribute und Methoden unserer neuen Klasse gleich testen zu können. Sinn muss dieser Test-Teil nicht unbedingt ergeben.

```
class Person:

# Konstruktor
    def __init__(self, name, vorname):
        self.Name=name
        self.Vorname=vorname


#Test-MAIN
P1=Person("Muster","Oleg")
P2=Person("Schulz","Franka")
P3=Person("Bauer","Kim")
```

Beim Laufen-Lassen unseres kleinen Programm erhalten wir keine Fehler-Meldung, aber auch keine Anzeige.
Unsere nächste Aufgabe soll also eine Anzeige der gespeichert Personen-Daten sein. Die Klassen-Definition wir entsprechend um die Methode zeigePersonDaten() erweitert. Den Test der Methode hängen wir dann auch gleich im Test-Teil an.

```
…
    def zeigePersonDaten(self):
        print("Name: ",self.Name,"   Vorname: ",self.Vorname)


#Test-MAIN
…
P3.zeigePersonDaten()
P1.zeigePersonDaten()
```

Jetzt erzeugen wir bei Ausprobieren auch tatsächlich Ausgaben auf dem Bildschirm.

8.11.x.7.1. Erweiterung der "Personen"-Klasse auf eine Familie

8.11.x.7. eine "Nachrichten"-Klasse

Klasse Nachrichten mit den Attributen Sender, Empfänger und Text (ev. ++)
Methoden senden, antworten und weiterleiten

8.11.x.y. eine Graphik-Beispiel-Klasse

→ http://www.b.shuttle.de/b/humboldt-os/python/kapitel4/index.html

```
# Die Grafik-Klassen in   graph.py

#! /usr/bin/python

import Tkinter
from Tkconstants import *
import Canvas

class Image:
   """
   /* Bildklasse
   """
   def __init__(self,Name):
      """
      /* Name: string : Dateiname des Bildes
      """
      self.Bild=Tkinter.PhotoImage(file=Name)
      self.Breite=self.Bild.width()
      self.Hoehe=self.Bild.height()

   def get_Bild(self):
      """
      /* liefert das Bildobjekt für 'image' in Canvas.ImageItem
      /* (Darstellung des Bildes auf einer Zeichenfläche)
      """
      return self.Bild

   def get_Breite(self):
      """
      /* liefert die Bildbreite in Pixeln
      """
      return self.Breite

   def get_Hoehe(self):
      """
      /* liefert die Bildhoehe in Pixeln
      """
      return self.Hoehe

class TColor:
   """
   /* erste primitive Version mit nur wenigen Farben
   /* Die Farben können über die deutschen Namen oder über
   /* Zahlen abgerufen werden
   /* Hinweis:   "#fff" entspricht "weiss",
   /*      statt "#xxx" kann auch "red", "green"  usw. benutzt werden
   /* andere Lösungen mit true-colors sind denkbar
   """
   def __init__(self):
      """
      /* transparent, schwarz, blau, gruen, tuerkis, rot, gelb, grau, weiss
      """
      self.Fnamen = { \
                     0:"transparent", \
                     1:"schwarz" , \
                     2:"blau"    , \
                     3:"gruen"   , \
                     4:"tuerkis" , \
                     5:"rot"     , \
                     6:"gelb"    , \
                     7:"grau"    , \
                     8:"weiss"   \
                  }

      self.Farbe={ \
                     "transparent":"", \
                     "schwarz"    :"#000", \
                     "blau"       :"#00f", \
                     "gruen"      :"#0f0", \
                     "tuerkis"    :"#0ee", \
                     "rot"        :"#f00", \
                     "gelb"       :"#ff0", \
                     "grau"       :"#ccc", \
```

```
                    "weiss"      :"#fff" \
                }

    def getColor(self,nr):
        """
        /* nr : int : 0 .. 8  für die oben angegebenen Farben
        /* liefert die Farbdarstellung für X
        """
        return self.Farbe[self.getFarbnamen(nr)]

    def getFarbnamen(self,nr):
        """
        /* nr : int : 0 .. 8  für die oben angegebenen Farben
        /* liefert den (deutschen) Bezeichner der Farbnummer (s.o.)
        """
        return self.Fnamen[nr]

    def getFarbe(self,wort):
        """
        /* wort : string :  ein Element aus den oben angegebenen Farben
        /* liefert die Farbdarstellung fuer X
        """
        return self.Farbe[wort]

class TFigur:
    """
    /* interne Hinweise:
    /* ZF ist Referenz auf Zeichenfläche, wird später gesetzt
    /* grafObj ist das aktuelle Grafikobjekt
    """
    def __init__(self):
        """
        /* Alle Grafik-Klassen erben von TFigur. TFigur wird beschrieben durch
        /* folgende Attribute:
        /* X1,Y1 (linke obere Ecke)
        /* X2,Y2 (rechte untere Ecke)
        /* Farbe
        /* Fuellfarbe
        """
        self.X1=20
        self.Y1=20
        self.X2=100
        self.Y2=100
        self.Farben=TColor()
        self.Farbe=self.Farben.getColor(0)
        self.Fuellfarbe=self.Farben.getColor(0)

    def setPos(self,ax1,ay1,ax2,ay2):
        """
        /* ax1,ay1 : int :(linke obere Ecke)
        /* ax2,ay2 : int :(rechte untere Ecke)
        """
        self.X1=ax1
        self.Y1=ay1
        self.X2=ax2
        self.Y2=ay2

    def getXPos(self):
        """
        /* liefert x-Wert der Position der linken oberen Ecke
        """
        return self.X1

    def getYPos(self):
        """
        /* liefert y-Wert der Position der linken oberen Ecke
        """
        return self.Y1

    def setFarbe(self,F):
        """
        /* F : string : deutscher Bezeichner (s.o.)
        """
        self.Farbe=self.Farben.getFarbe(F)

    def getFarbe(self):
        """
        /* gibt akt. Farbe zurück : string : Farbrepr. für X
        """
        return self.Farbe
```

```
   def setFuellfarbe(self,F):
      """
      /* F : string : deutscher Bezeichner (s.o.)
      """
      self.Fuellfarbe=self.Farben.getFarbe(F)

   def getFuellfarbe(self):
      """
      /* gibt akt. Füllfarbe zurück : string : Farbrepr. für X
      """
      return self.Fuellfarbe

   def pos_versetzen_um(self,dx,dy):
      """
      /* versetzt die Position des heweiligen Grafikobjektes um dx und dy
      """
      self.X1=self.X1+dx
      self.X2=self.X2+dx
      self.Y1=self.Y1+dy
      self.Y2=self.Y2+dy

   def zeigen(self):
      """
      /* zeigt das Grafikobjekt auf dem Schirm an
      """
      pass

   def loeschen(self):
      """
      /* löscht das Grafikobjekt auf dem Schirm
      """
      self.grafObj.move(1000,1000)

   def entfernen(self):
      """
      /* entfernt das Grafikobjekt aus dem Speicher
      """
      self.grafObj.delete()


class TLinie(TFigur):
   """
   /* Klasse Linie
   """
   def __init__(self):
      TFigur.__init__(self)
      x = self.getFarbe()
      self.grafObj=Canvas.Line(TFigur.ZF,(self.X1, self.Y1),(self.X2, self.Y2)\
                                                          , {"fill": x})

   def zeigen(self):
      self.grafObj.config(fill=self.getFarbe())
      self.grafObj.coords(((self.X1,self.Y1),(self.X2,self.Y2)))

class TEllipse(TFigur):
   """
   /* Klasse Ellipse
   """
   def __init__(self):
      TFigur.__init__(self)
      x = self.getFarbe()
      y = self.getFuellfarbe()
      self.grafObj=Canvas.Oval(TFigur.ZF, (self.X1, self.Y1), \
                              (self.X2, self.Y2), {"outline": x, "fill": y})

   def zeigen(self):
      self.grafObj.config(fill=self.getFuellfarbe(),outline=self.getFarbe())
      self.grafObj.coords(((self.X1,self.Y1),(self.X2,self.Y2)))

class TKreis(TFigur):
   """
   /* Klasse Kreis
   """
   def __init__(self):
      """
      /* zus. Attribute sind hier: Radus, x-Mittelpunkt, y-Mittelpunkt
      """
      TFigur.__init__(self)
      self.R=0
```

```
      self.Mx=0
      self.My=0
      x = self.getFarbe()
      y = self.getFuellfarbe()
      self.grafObj=Canvas.Oval(TFigur.ZF, (self.X1, self.Y1), \
                               (self.X2, self.Y2), {"outline": x, "fill": y})

   def __berechne_Standard(self):
      self.X1=self.Mx-self.R
      self.X2=self.Mx+self.R
      self.Y1=self.My-self.R
      self.Y2=self.My+self.R

   def setRadius(self,r):
      """
      /* r : int : Radius
      """
      self.R=r
      self.__berechne_Standard()

   def getRadius(self):
      """
      /* liefert aktuelle Radiuslänge
      """
      return self.R

   def setMPos(self,ax,ay):
      """
      /* ax, ay : int
      /* setzt Mittelpunktskoordinaten
      """
      self.Mx=ax
      self.My=ay
      self.__berechne_Standard()

   def zeigen(self):
      self.grafObj.config(fill=self.getFuellfarbe(),outline=self.getFarbe())
      self.grafObj.coords(((self.X1,self.Y1),(self.X2,self.Y2)))


class TRechteck(TFigur):
   """
   /* Klasse Rechteck
   """
   def __init__(self):
      TFigur.__init__(self)
      x = self.getFarbe()
      y = self.getFuellfarbe()
      self.grafObj=Canvas.Rectangle(TFigur.ZF, (self.X1, self.Y1), \
                               (self.X2, self.Y2), {"outline": x, "fill": y})

   def zeigen(self):
      self.grafObj.config(fill=self.getFuellfarbe(),outline=self.getFarbe())
      self.grafObj.coords(((self.X1,self.Y1),(self.X2,self.Y2)))


class TText(TFigur):
   """
   /* Klasse Text zur Beschriftung der Zeichenfläche
   """
   def __init__(self):
      """
      /* Attribute sind
      /* Text : string
      /* Schriftart : String  (X-Fonts-Bezeichner)
      /* Zeichen-Hoehe : int : default = 10
      """
      TFigur.__init__(self)
      self.Text=""
      self.Schriftart="*"
      self.Hoehe=10

   def setPos(self,ax,ay):
      """
      /* ax, ay : int : Position des ersten Zeichens
      """
      self.X1=ax
      self.Y1=ay

   def setText(self,Text):
```

```
      """
      /* Text : string : auszugebender Text
      """
      self.Text=Text

   def setFont(self,Art="*",Grad=10):
      """
      /* Art : string : Font-Name
      /* Grad : int : Zeichengröße
      """
      self.Schriftart=Art
      self.Hoehe=Grad

   def zeigen(self):
      self.grafObj=Canvas.CanvasText(TFigur.ZF, self.X1, self.Y1, \
             anchor="w", fill=self.Farbe, font=(self.Schriftart, self.Hoehe))
      self.grafObj.insert(0,self.Text)


class TZeichenblatt:
   """
   /* Zeichenblatt entspricht Canvas. Mit Init wird ein Bild unterlegt
   /* Zeichenblatt vom Typ TZeichenplatt wird erzeugt und steht zur
   /* Verfügung.
   """
   def __init__(self):
      pass

   def Init(self,Name):
      """
      /* Init hinterlegt das Bild
      /* Name : string : Dateiname (gif)
      """
      self.oWindow=Tkinter.Tk()
      self.oWindow.title("Zeichenfläche - nach S. Spolwig    -----   Kokavecz")
      self.oBild=Image(Name)
      self.X1=0
      self.Y1=0
      self.X2=self.oBild.get_Breite()
      self.Y2=self.oBild.get_Hoehe()
      Geometrie=str(self.X2)+"x"+str(self.Y2)+"+0+0"
      self.oWindow.geometry(Geometrie)
      self.oEbene=Tkinter.Canvas(self.oWindow,relief=SUNKEN, bd=5, \
                                          width=self.X2, height=self.Y2)
      bild=Canvas.ImageItem(self.oEbene,(0,0),anchor="nw", \
                                             image=self.oBild.get_Bild())
      self.oEbene.pack()
      TFigur.ZF=self.oEbene


   def get_Breite(self):
      """
      /* liefert die Bildbreite in Pixeln
      """
      return self.oBild.get_Breite()

   def get_Hoehe(self):
      """
      /* liefert die Bildhoehe in Pixeln
      """
      return self.oBild.get_Hoehe()


   def refresh(self):
      TFigur.ZF.update()

# sollte oZeichenblatt oder mein_Zeichenblatt heißen:
Zeichenblatt = TZeichenblatt()
```

Klassen-Dokumentation → http://www.b.shuttle.de/b/humboldt-os/python/kapitel4/grafik.py.html

8.11.x.2. Polymophismus und Vererbung

class erbendeKlasse(vererbendeKlassenListe):

z.B. an erweiterter Konten-Klasse jetzt auch mit Zinsen und Schulden

Aufgaben:

1. *Erweitern Sie das Konten-Programm so, dass für jede Minute der Zins für einen Monat angesetzt wird!*

alle Attribute und Methoden der in der vererbenden Klassenliste aufgeführten Klassen sind nun auch in der erbenden Klasse verfügbar (diese können hier aber auch überschrieben werden!)
die originalen Methoden werden über **vererbendeKlasse.Methode** und die überschriebenen über **erbendeKlasse.Methode**
Aufrufe ohne Klassen-Angabe verbleiben erst einmal in der aktuellen Klasse

Beispiel: Bücher-Klasse

```
class Buch:
    BuecherZahl=0

    def __init__(self,titel,autor,verlag,isbn,preis):
        self.Titel=titel
        self.Autor=autor
        self.Verlag=verlag
        self.ISBN=isbn
        self.Preis=preis
        Buch.BuecherZahl+=1

    def zeigeBuchInfo(self):
        print("Buch-Info:")
        print("Autor: ",self.Autor,"   Titel: ",self.Titel)
        print("Verlag: ",self.Verlag,"   ISBN: ",self.ISBN)

    def pruefeISBN(self):
        pass

    def __del__(self):
        Buch.BuecherZahl-=1

# Main
buch1=Buch("Das Leben der Z","Silp","Universal","1234567890X",24)
buch1.zeigeBuchInfo()
print("Preis= ",buch1.Preis, "Euro")
print("--> akt. Buchbestand: ",Buch.BuecherZahl)
print()
print()
print("Entfernen Buch1")
del(buch1)
print("--> akt. Buchbestand: ",Buch.BuecherZahl)
print("Ende")
```

```
>>>
Buch-Info:
Autor:  Silp    Titel:  Das Leben der Z
Verlag:  Universal    ISBN:  1234567890X
Preis=  24 Euro
--> akt. Buchbestand:  1


Entfernen Buch1
--> akt. Buchbestand:  0
Ende
>>>
```

nun braucht man z.B. für Fachbücher neben den üblichen Angaben vielleicht auch noch Informationen zu Fachgebieten und Themen oder Stich
Natürlich möchten – wir faulen Programmierer – nicht wieder alles neu programmieren. Wir haben ja schon eine super programmierte und getestete Klasse für normale Bücher.
Auf der Basis dieser Bücher-Klasse erstellen wir nun die Fachbuch-Klasse.

```
…
       Buch.BuecherZahl-=1
```

```
class FachBuch(Buch):
    BuecherZahl=0

    def
__init__(self,titel,autor,verlag,isbn,preis,fachbereich,stichwort):
        # Buch.__init__(self,titel,autor,verlag,isbn,preis)
        super().__init__(titel,autor,verlag,isbn,preis)
        self.Fachbereich=fachbereich
        self.Stichwort=stichwort
        FachBuch.BuecherZahl+=1

    def zeigeBuchInfo(self):
        print("Fach-",end='')
        # Buch.zeigeBuchInfo(self)
        super().zeigeBuchInfo()
        print("Fachbereich: ",self.Fachbereich,"   Stichwort:
",self.Stichwort)

    def __del__(self):
        FachBuch.BuecherZahl-=1
        Buch.BuecherZahl-=1
```

```
# Main
buch1=Buch("Das Leben der Z","Silp","Universal","1234567890X",24)
buch1.zeigeBuchInfo()
print("Preis= ",buch1.Preis, "Euro")
print("--> akt. Buchbestand: ",Buch.BuecherZahl,"   davon: ",
      FachBuch.BuecherZahl," Fachbücher")
print()
print()
buch2=FachBuch("LB Informatik","Meier","Fachbuchverlag",
           "1234567890X",30,"Progammierung","Python")
buch2.zeigeBuchInfo()
print("Preis= ",buch2.Preis, "Euro")
print("--> akt. Buchbestand: ",Buch.BuecherZahl,"   davon:
",FachBuch.BuecherZahl," Fachbücher")
print()
print()
print("Entfernen Buch2 (Fachbuch)")
del(buch2)
print("--> akt. Buchbestand: ",Buch.BuecherZahl,"   davon:
",FachBuch.BuecherZahl," Fachbücher")
print()
print("Entfernen Buch1")
…
```

```
>>>
Buch-Info:
Autor:  Silp    Titel:  Das Leben der Z
Verlag.  Universal    ISBN:  1234567890X
Preis=  24 Euro
--> akt. Buchbestand:  1    davon:  0  Fachbücher


Fach-Buch-Info:
Autor:  Meier    Titel:  LB Informatik
Verlag.  Fachbuchverlag    ISBN:  1234567890X
Fachbereich:  Progammierung    Stichwort:  Python
Preis=  30 Euro
--> akt. Buchbestand:  2    davon:  1  Fachbücher


Entfernen Buch2 (Fachbuch)
--> akt. Buchbestand:  1    davon:  0  Fachbücher

Entfernen Buch1
--> akt. Buchbestand:  0    davon:  0  Fachbücher
Ende
>>>
```

Beim genauen Betrachten des Quelltextes kann man einige Spezialitäten erkennen. Zum Ersten kann man innerhalb jeder Objekt-Ebene gleich namige Attribute / Variablen-Namen nutzen (hier: BuecherZahl). Sie beziehen sich immer auf die jeweilige Ebene, die als Objektnamens-Teil (vor dem Punkt) mit angegeben werden muss.
Dann können wir mit dem allgemeinen Namen **super** für die übergeordnete Klasse zurückgreifen. Das ist z.B. dann praktisch, wenn sich solche Namen öfter ändern oder der Quelltext mehrfach genutzt werden soll. **Super** ist somit strukturell dem **self** äquivalent.
In Python lässt sich auch über die Methoden hinweg z.B. eine spezielle Ausgabe realisieren. Im Fall eines Fachbuches wird nur das Wörtchen "Fach-" zur Anzeige (zeigeBuchInfo()) gebracht und dann direkt die Anzeige-Methode von Buch aufgerufen Die Anzeige-Methode von Fachbuch ergänzt dann noch die speziellen Attribute von Fachbuch.
Interessant ist auch, dass die Methoden den Objekten entsprechend ihrer Klasse zugeordnet werden. In beiden Bücherklassen gibt es die gleichlautende Methode zeigeBuchInfo(). Beim Aufruf von einem Fachbuch aus wird zuerst die Fachbuch-Methode genutzt. Diese verwendet dann – in unserem Beispiel – die gleichnamige Methode aus der Buch-Klasse.
Gibt es keine zeigeBuchInfo()-Methode in der Fachbuch-Klasse, dann wird dies aus der übergeordneten Klasse genutzt. Natürlich fehlen dann die zusätzlichen Fachbuch-Informationen.

8.11.x.y. Tips und Tricks zu Objekt-orientierten Programmen / Klassen-Definitionen

um eine Klassen-Defintion mit einem kleinen Test-Programm auch als Modul benutzen zu können, gibt es den folgenden Konstrukt, der nur dann den THEN-Zweig ausführt, wenn der Quelltext als Haupt-Programm (MAIN) ausgeführt wird. Ist der Quelltext ein Modul wird dieser Zweig nicht ausgeführt.

```
in __name__ == "__main__":
    # hier stehen die Anweisungen für die Nutzung als (Haupt-)Programm
```

8.11.x. OOP-Programmbeispiele

```
# -*- coding: utf8 -*-

  # Klasse zur Verwaltung von Personen
  class Person(object):
    # Konstruktor/Initialisierer
    def __init__(self, alter, groesse, name = None):
      self.alter = alter
      self.groesse = groesse
      self.name = name

    # String-Repräsentation einer Person erstellen
    def __repr__(self):
      return repr((self.alter, self.groesse, self.name))

    # einfache String-Repräsentation einer Person erstellen
    def __str__(self):
      return '%s/%s/%s' % (self.alter, self.groesse, self.name)

    # Person altern lassen, also Alter um n Jahre erhöhen
    def altern(self, n = 1):
      self.alter += n

    # mittels eines Dekorators eine Property mytuple erzeugen
    @property
    def mytuple(self):
      # das ist der Getter; den Namen lassen wir hier aus
      return self.alter, self.groesse
    # alternativ:
    #   def mytuple(self): return self.alter, self.groesse
    #   mytuple = property(mytuple)

    # einen Setter für die Property definieren
    @mytuple.setter
    def mytuple(self, t):
      if t[0] > 10 and t[1] > 150:
        self.alter, self.groesse = t[:2]
        if len(t) > 2:
          # wenn t einen Namen enthält, dann diesen auch setzen
          self.name = t[2]

  # Personenliste erstellen
  personen = [Person(39, 172, 'ABC'), Person(88, 165), Person(15, 181),
Person(88, 175)]

  # Ausgabe der Personenliste
  print personen
  print '==='

  # Iteration über der Personenliste und Ausgabe der einzelnen Personen
  for pers in personen:
    print pers, '==>', repr(pers)
  print '==='

  # alle Personen altern lassen
  for pers in personen:
    pers.altern(3)

  # nochmal ausgeben
```

```
print 'nach dem Altern'
print personen
print '==='

# nochmal altern lassen, diesmal funktional
map(lambda x: x.altern(3), personen)
print 'nach dem 2. Altern'
print personen
print '==='

# nochmal funktional altern lassen, diesmal mit Vorzugswert n
map(Person.altern, personen)
print 'nach dem 3. Altern'
print personen
print '==='

# Ausgabe der sortierten Personenliste
print sorted(personen, key = lambda pers: (pers.alter, pers.groesse))
print '==='

# dito mit benannter Funktion statt einer anonymen lambda-Funktion
def pers_key(pers):
  return pers.alter, pers.groesse

print sorted(personen, key = pers_key)

# Attribute sind public, man kann von außen zugreifen
print personen[0].alter
print personen[0].groesse
print personen[0].name

p = personen[0]
print p.alter + p.groesse

# Nutzung der Property mit Getter
print p.mytuple
p.alter += 100
print p.mytuple

# Nutzung des Setters der Property
p.mytuple = 1, 2 # wird vom Setter stillschweigend ignoriert
print repr(p.mytuple)

p.mytuple = 11, 155
print repr(p.mytuple)

# nochmal, aber mit Name
p.mytuple = 11, 155, 'Pumuckl'
print repr(p.mytuple)

for p in personen:
  print p

print 'maximales Element einer Personen-Liste bestimmen'
print max((person.alter, person.groesse) for person in personen)

# alternativ nutzbar wären
print max(map(lambda elem: (elem.alter, elem.groesse), personen))
print max(personen, key = lambda elem: (elem.alter, elem.groesse))

print

# Größe von außen ändern
personen[1].groesse += 5
```

```
for p in personen:
  print p

print

# neues Attribut setzen
personen[1].name2 = 'XYZ'
for p in personen:
  print p # __str__() wird für die String-Darstellung gerufen

print

# hier sieht man das neue Attribut
for p in personen:
  print vars(p)

print

print 'Maximum der Property mytuple'
print max(person.mytuple for person in personen)

# das Tupel der letzten Person der Liste ändern
personen[-1].mytuple = 110, 190

# Maximum erneut ausgeben
print 'Maximum der Property mytuple nach Zuweisung'
print max(person.mytuple for person in personen)

# nochmal alle Attribute mit vars()
print
for p in personen:
  print vars(p)
```

Q: https://www-user.tu-chemnitz.de/~hot/PYTHON/

8.12. GUI-Programme mit Tkinter

Tkinter ist nicht etwa eine Fortsetzung ode Erweiterung der Turtle-Graphik. Nein, es ist genau anders herum – die Turtle-Graphik basiert auf dem mächtigen Graphik-Modul Tkinter.
Aber die Turtle-Graphik ist einfach der bessere Einstieg in die graphische Programmierung. Es macht richtig Laune, der Schildkröte zuzusehen.
Echte graphische Aufgaben löst man dann eher mit Tkinter.
Tkinter ist die Python-Schnittstelle zur Graphik-GUI "Tcl/Tk" (GUI … Graphical User Interface)

Tk ist das GUI-Erweiterung für Tcl
Tcl ist eine Scriptsprache die 1991 von John OUSTERHOUT entwickelt wurde
zu Tk und Tcl gobt es Schnittstellen für die verschiedensten Programmiersprachen (z.B. Perl, Ruby, Common LISP, Ada, R , …)

Tk stellte Widgets (Steuerelemente, Bedienelemente) für die Erstellung / Zusammenstellung und Funktionalisierung von graphischen Programmen zur Verfügung

Für die in graphischen Oberflächen weniger bewanderten folgt hier eine Vorstellung / Zusammenstellung von Objekten, die eben Tk – wie andere Programmier-Systeme eben auch – bereitstellt.

Tk-Widgets

- **button** Schaltfläche
- **canvas** Graphik-Fläche
- **checkbutton** Options-Feld
- **combobox** Auswahl-Box
- **entry** Eingabefeld
- **frame** Fenster-Bereich; Container für andere Objekte
- **label** Beschriftung(sfeld)
- **labelframe**
- **listbox** Listen-Feld
- **menu** Menü
- **menubutton** Menü-Eintrag
- **message** Text-Feld
- **notebook**
- **panedwindow**
- **progressbar**
- **radiobutton** Auswahl-Feld
- **scale** Gleiter
- **scrollbar** (Bild-)Laufleiste
- **separator** Fenster-Teiler
- **sizegrip**
- **spinbox**
- **text** Text-Feld
- **treeview**
- **tk_optionMenu**

Für die Integration in das Windows-System werden die klassischen System-Fenster bereitgestellt. Dazu gehören:

Tk-System-Fenster:

- **tk_chooseColor**
- **tk_chooseDirectory**
- **tk_dialog**
- **tk_getOpenFile**
- **tk_getSaveFile**
- **tk_messageBox**
- **tk_popup**
- **toplevel**

Geometrie-Manager organisieren die Anordnung der Bedien-Elemente im Fenster / auf der Fensterfläche

Tk-Geometrie-Manager

- **pack** einfache Geometrie, Objekte werden vorrangig untereinander angeordnet
- **grid** Gitter- bzw. Tabellen-orientierte Geometrie; Objekte werden an Gitter-Plätzen innerhalb des Fensters angeordnet (Spalten: 0 bis x; Zeilen: 1 .. y)
- **place** genaue (absolute) oder relative Platzierung der Objekte

Mit Tkinter kann man auch auf der Konsolen-Ebene arbeiten. Bei manchen Aktionen ist sogar sehr sinnvoll. In der Praxis sind abe eher nachnutzbare Programme interessant. Deshalb werden wir hier fast ausschließlich zusammenhängende Quell-Texte schreiben und diese dann ausprobieren.

8.12.1. … und der erste Programmierer sprach: "Hallo Welt!"

Wir gehen mal ganz klassisch vor. Also …

Aufgabe:
Geben Sie das folgende Programm ein! Die vielen Leerzeilen sind nicht wirklich notwendig, sie dienen nur der Strukturierung für die Erläuterungen rechts neben dem Quelltext.

```
from tkinter import *

fenster=Tk()




elem=Label(fenster,text="Hallo Welt!")
elem.pack()


fenster.mainloop()
```

import des Tkinter-Moduls

Erzeugen eines Root-Objektes namens **fenster** vom Typ Tk

Erzeugen eines untergeordneten Label-Objektes namens **elem** und einem Text

mit der Pack-Methode wird das untergeodnete Objekt integriert

realisiert die Anzeige der Objekte und erzeugt die Ereignis-Abfrage-Schleife

Die etwas ungewöhnliche Notierung ist ein typisches Beiwerk der sogenannten Objektorientierten Programmierung. Das irgnorieren wir hier einfach und zwingen uns zu dieser Schreibweise. Später werden wir sie verstehen, hier ist sie erst mal als gegeben / notwendig zu akzeptieren.
Das Ergebnis sieht natürlich richtig Windows-like aus, aber provoziert sofort die Fragen:
Geht der Text auch in farbig?
Kann man den Text größer oder in einer anderen Schriftart darstellen?

Bevor aber dazu kommen, schnell ein paar Hinweise zur Notierung der Import-Anweisung und er sich daraus ergebenen Notierung im weiteren Quell-Text.
Das gleiche "Hallo Welt!-Programm kann auch mit der Import-Zeile:

```
import tkinter as tk
```

beginnen. Das Programm würde dann so aussehen:

```
import tkinter as tk

fenster=tk.Tk()

elem=tk.Label(fenster, text="Hallo Welt!")
elem.pack()

fenster.mainloop()
```

Diese Notierung wird man gegegebenfalls auch in verschiedenen Beispielen aus Büchern oder dem Internet finden. Man muss allerdings jetzt vor jedem benutztem Objekt noch die Herkunft von tk (als solches haben wir das Modul Tkinter ja jetzt importiert) mit angeben. Das bedeutet nur deutlich mehr Schreibaufwand. Nur bei Kombinationen und Überschneidungen mit anderen Objekten / Modulen ist diese Schreibung sinnig.
Ebenfalls funktioniert die vereinfachte Import-Anweisung:

import tkinter

Kommen wir zu Gestaltung / Formatierung eines Label zurück. Natürlich können wir Farben und Schriften verändern und es auch nicht wirklich schwer. Wegen der ungewöhnlich Text-aufwändigen Notierung, sollten wir uns gleich an eine übersichtlichere Strukturierung des Quell-Textes gewöhnen. Hier sit es allerdings nicht so, dass diese – wie bei Verzweigungen Schleifen und Funktionen – notwendig ist.

```
from tkinter import *

fenster=Tk()

elem=Label(fenster,
           text="Hallo Welt!",
           fg="blue",
           bg="yellow",
           font="Times 12 bold"
           ).pack()

fenster.mainloop()
```

auch möglich:
font=('Times','12','bold')

Schaut man sich nebenstehendes Resultat des obigen Quelltextes an, dann werden die einzelnen Eigenschaften(-Kürzel) schnell klar.
Typische Schriftarten sind **"Courier", "Arial", "Comic Sans MS", "Verdana", "System", "Fixedsys", "MS Sans Serif", "Symbol", "Helvetica"** und **"ansi"**.
Die anderen Schriftstil-Bezeichnungen sind **"normal"**, **"italic"** für kursiv, **"roman"** für ???, **"underline"** für unterstrichen und **"overstrike"** für durchgestrichen.
Einige Worte noch zu der seltsamen **mainloop()**-Funktion am Ende der meisten hier gezeigten Programme. Mainloop startet die Ereignis-Abfrageschleife für das gestartete Programm. Irgenwie soll es ja auf Maus-Klicks oder Tastatur-Befehle reagieren. Genau das realisiert die mainloop()-Funktion. Die **mainloop()**-Funktion wird mit dem Zerstören des (Haupt-)Programm-Fensters durch das destroy-Kommando beendet.
Tkinter-Programme ohne mainloop lassen sich nur in bzw. mit IDLE benutzen. Dort übernimmt der Interpreter die Ereignis-Verarbeitung bzw. übergibt sie zeitweilig an das gerade benutzte Programm.
Programme, die eigenständig funktionieren sollen – also auch direkt aus dem Arbeitsplatz oder dem Windows-Explorer heraus gestartet und benutzt werden sollen – müssen am Schluss die **mainloop()**-Funktion enthalten. Sie können und sollten dann auch als **pyw**-Datei gespeichert werden. In diesem Dateityp lassen sich die Programme direkt unter Windows etc. ausführen

8.12.2. Nutzung verschiedener Bedienelemente

Die Vielfalt der verfügbaren Bedienelemente ist in Windows und in Tk recht groß. Viele sind für eine moderne Interaktion mit den Programmen toll, aber nur wenige Elemente sind für rein funktionelle Programme wirklich notwendig. Diese werden wir hier vorstellen.
Wer mit diesen klar kommt, kann sich dann in die höheren Sphären der GUI-Programmierung begeben.

8.12.2.1. Button's - Schaltflächen

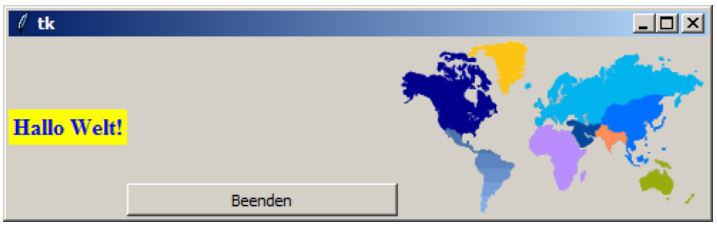
Unser erstes "Hello Welt!"-Programm ließ sich nur über die Fenster-Schaltflächen schließen. Für einfache Programme ist das auch ok, aber wir wollen ja später doch ein bisschen professioneller arbeiten. Also müssen Schaltflächen in die Programme rein.
Beginnen wir mit einem Beenden-Button, der natürlich auch die passende Funktion verpasst bekommen soll.

```
from tkinter import *

fenster=Tk()

elemLabel=Label(fenster,
                   text="Hallo Welt!",
                   fg="blue",
                   bg="yellow",
                   font="Times 12 bold"
            ).pack()
elemButton=Button(fenster,
                  text='Beenden',
                  width=30,
                  command=fenster.destroy
            ).pack()
fenster.mainloop()
```

Die Fenster-Elemente hätte man alle auch nur mit elem bezeichnen können. Später wollen wir aber doch mal das eine oder andere Detail eines Bedien-Elementes ändern. Deshalb bekommt jedes Objekt einen eigenen Namen. Ein einfaches Durchzählen ist natürlich auch möglich.
Nun soll noch ein kleines Bildchen – ein Icon – mit angezeigt werden. Zuerst einmal soll das Bildchen rechts nebendem "Hallo Welt!"-Text erscheinen. Dazu muss der verfügbare Raum verteilt werden. Es gibt ein Label-Objekt links und ein weiteres Label-Objekt – mit der Bildchen – rechts. Die Bildchen müssen als (nicht-animierte) GIF-Datei vorliegen (alternativ gehen auch PGM- bzw. PPM-Dateien).

```
from tkinter import *

fenster=Tk()

elemLabel1=Label(fenster,
                   text="Hallo Welt!",
                   fg="blue",
                   bg="yellow",
                   font="Times 12 bold"
            ).pack(side="left")

bildchen=PhotoImage(file="erde50.gif")
elemLabel2=Label(fenster,
                   image=bildchen
            ).pack(side="right")

elemButton=Button(fenster,
                  text='Beenden',
                  width=30,
                  command=fenster.destroy
            ).pack(side="bottom")

fenster.mainloop()
```

Wer ein bisschen mit den side-Parametern in der Pack-Methode rumspielt, wird schnell merken, dass das Positionieren der Widgets (Bedienelemente) nicht so ganz ohne ist.



Ein Label kann nun auch dafür benutzt werden, um Daten auszugeben. Für einen ersten Versuch soll im Hintergrund ein Zähler laufen, der nach einer Sekunde den Text des Labels ändert. Als Text wird der aktuelle Zähler-Stand genutzt.

```
from tkinter import *

fenster=Tk()

zaehler=0
def zaehlerLabel(label):
  def zaehlen():
    global zaehler
    zaehler+=1
    label.config(text=str(zaehler))
    label.after(1000, zaehlen)
  zaehlen()


fenster.title("Zähler")

elemLabel=Label(fenster, font="Times 20 bold")
elemLabel.pack()

zaehlerLabel(elemLabel)

elemButton=Button(fenster,
                  text='Beenden',
                  width=20,
                  command=fenster.destroy)
elemButton.pack()

fenster.mainloop()
```

Zähl-Funktion

hier wird der Label-Text neu festgelegt

Beschriftung / Titel des Fensters

Label, der für die Anzeige des Zählers genutzt werden soll

starten der Zähler-Funktion

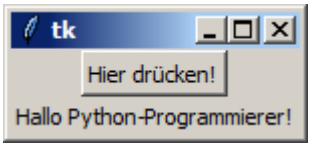
Nach dem Programm-Start beginnt der Zähler zu zählen und nach jeweils 1000 ms (= 1 s) ändert die Funktion den Text des Labels mit dem neuen Zähler-Wert.

8.12.2.1.1. eine eigene Button-Aktion erstellen

Wollen wir nun einem Button eine eigene Aktion programmieren. Dabei müssen zwei Dinge erledigt werden. Einmal müssen wir eine Aktion als Funktion definieren und zum Anderen muss diese Aktions-Funktion der Betätigung der Schaltfläche zugeordnet werden.
Letzteres passiert in den Optionen des erstellten Button als command-Attribut. Der Name der Funktion sollte eindeutig sein, damit später bei vielen Aktionen eine klare Zuordnung möglich ist.
Die Funktion selbst wird ganz klassisch angelegt und in unserem Fall einfach mit einem neu anzulegenen Label versehen.

```
from tkinter import *

def button_aktion():
    elem=Label(fenster,text="Hallo Python-Programmierer!")
    elem.pack()


fenster=Tk()

elem=Button(fenster,
            text="Hier drücken!",
            command=button_aktion)
elem.pack()

fenster.mainloop()
```

Beim Starten des Programm bekommen wir zuerst das obere Fenster angezeigt. Sobald die Schaltfläche gedrückt wird, erscheint der untere Text.



Rücksetzen des Zählers über einen weiteren Button

8.12.2.1.2. Button gestalten / formatieren

bitmap="questhead"

bitmap="error"

bitmap="warning"

bitmap="error"

bitmap="hourglass"

Neben reinem Text lassen sich Button auch mit verschiedensten anderen Details darstellen.
Ersetzt man z.B. das Text-Argument durch ein Bitmap-Argument, dann erscheinen statt dem Text je nach Option verschiedenen kleine Icon's.
Nebenstehend sind die wichtigsten Beispiele aufgezeigt. Solche kleinen Schaltflächen eignen sich z.B. zum Aufrufen von kleinen Hilfestellungen usw.

```
from tkinter import *

def button_aktion():
    elem=Label(fenster,text="Super gemacht!")
    elem.pack()

fenster=Tk()

elem=Button(fenster,
            text="Drücke jetzt!",
            bitmap="warning",
            command=button_aktion)
elem.pack()

fenster.mainloop()
```

Text und Bild lassen sich aber nicht direkt nebeneinander in einer Schaltfläche anzeigen. Die Bild-Option hat Vorrang und der Text wird ignoriert.

8.12.2.2. Nachrichten-Felder / Text-Felder

Label sind doch recht beschränkte Objekte. Für Beschriftungen sind sie völlig ausreichend. Längere Texte oder Nachrichten lassen sich damit nur sehr aufwändig darstellen. Ein etwas flexibleres Widget ist die message.
Mit ihr lassen sich längere, mehrzeilige mit zusätzlichen Hervorhebungen realisieren.

ausgewählte Optionen für Message-Widgets

- **background bg** Hintergrundfarbe; wenn keine angegeben wird, dann wird die Systemeinstellung genutzt
- **font** Schrift-Art, -Größe und -Stil; wenn nichts angegeben wird, dann wird die Systemeinstellung genutzt
- **foreground fg** Vordergrundfarbe / Textfarbe; wenn keine angegeben wird, dann wird die Systemeinstellung genutzt
- **text** anzuzeigender Text
- **textvariable** ist eine spezielle Textvariable, die im Hintergrund geändert werden kann → die Anzeige ändert sich entsprechend
- **takefocus** wird dieser auf True gesetzt, dann erhält die Message-Box den Eingabe- / Bedien-Focus; normalerweise ist der Wert: False

Weitere Optionen sind **anchor, aspect, borderwidth, cursor, highlightbackground, highlightcolor, highlightthickness, justify, padx, pady, relief** und **width**. Wenn für spezielle Gestaltungen von Message-Boxen Bedarf besteht, dann kann man sich in den üblichen Quellen informieren. Für Standard-Anwendungen sind sie nicht notwendig.

8.12.2.3. Eingabe-Felder / Eingabezeilen

Widmen wir uns nun der Eingabe von Texten, Zahlen usw. Dafür sind primär die Entry-Objekte gedacht. Ein erstes Programm soll die gewaltige Aufgabe lösen, aus eingegebenen Vor- und Nachnamen eine ordentliche Begrüßung zu erzeugen!

```
from Tkinter import *          ## Tkinter importieren
root=Tk()                      ## Wurzelfenster!
eingabe = Entry(root)          ## Eingabezeile erzeugen
eingabe.pack()                 ## und anzeigen
```

.get()
gibt Eingabezeile als Text (!) zurück

.delete(position)
löscht Zeichen an der Position (Zählung beginnt bei 0)

.insert(END, text)

```
e = Entry(master)
e.pack()

e.delete(0, END)
e.insert(0, "a default value")
```

```
s = e.get()
```

```
v = StringVar()
e = Entry(master, textvariable=v)
e.pack()

v.set("a default value")
s = v.get()
```

```
from Tkinter import *

master = Tk()

e = Entry(master)
e.pack()

e.focus_set()

def callback():
    print e.get()

b = Button(master, text="get", width=10, command=callback)
b.pack()

mainloop()
e = Entry(master, width=50)
e.pack()

text = e.get()
def makeentry(parent, caption, width=None, **options):
    Label(parent, text=caption).pack(side=LEFT)
    entry = Entry(parent, **options)
    if width:
        entry.config(width=width)
    entry.pack(side=LEFT)
    return entry

user = makeentry(parent, "User name:", 10)
password = makeentry(parent, "Password:", 10, show="*")
content = StringVar()
entry = Entry(parent, text=caption, textvariable=content)

text = content.get()
content.set(text)
```

mehr: http://www.wspiegel.de/tkinter/tkinter02.htm

8.12.2.4. Nachrichten-Boxen

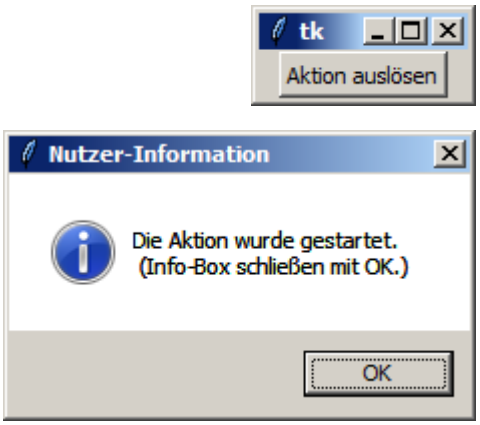
Vielfach sollen kleine Informationen, Fehlerhinweise usw. usf. auf dem Bildschirm gebracht werden. Diese Art der Nutzer-Information wird Message-Box genannt und
Windows – und die meisten anderen graphischen Betriebssysteme – kennen mehrere Arten von Message-Boxen. Diese unterscheiden sich praktisch vor allem hinsichtlich der eingeblendeten Icon's und / oder der Farbgebung.
Klassisch unterscheidet man zwischen "Fehler"-, "Warnung"-, "Frage"- und "Information"-s-Box.
Von der Ebene des Programmierers aus sind alle gleich.
Die Message-Boxen von Python sind etwas breiter aufgestellt. Zumindestens scheint es so. Die Abfrage- / Frage-Box kann mit unterschiedlichen Schaltflächen (Button's) versehen werden. Aus den verschiedenen Varianten ergeben sich unterschiedliche Box-Typen:

Tk-Message-Boxen

- **showinfo** Message-Box mit einer Info-Blase als Icon
- **showwarning** Message-Box mit Warn-Zeichen
- **showerror** Message-Box mit Fehler-Icon
- **askyesno** Message-Box mit Frage-Zeichen als Icon und den Schaltflächen [Ja] und [Nein]
- **askokcancel** Message-Box mit Frage-Zeichen als Icon und den Schaltflächen [OK] und [Abbrechen]
- **askretrycancel** Message-Box mit Frage-Zeichen als Icon und den Schaltflächen [Fortsetzen] und [Abbrechen]

```
from tkinter import *

def antwort():
    titel="Nutzer-Information"
    nachricht="Die Aktion wurde gestartet.\n
               (Info-Box schließen mit OK.)"
    messagebox.showinfo(titel,nachricht)


fenster=Tk()

button1=Button(fenster,
                text="Aktion auslösen",
                command=antwort)
button1.pack()

fenster.mainloop()
```

die Texte für Titel und Nachricht können natürlich auch direkt in den Aufruf der MessageBox notiert werden
\n steht für einen Zeilenumbruch



Wenn bei Ihnen die Message-Box-Fenster anders aussehen, dann ist das dem benutzten Betriebssystem geschuldet. Diese Message-Boxen werden nämlich direkt von Windows (oder dem jeweils benutzten Betriebssystem) direkt zur Verfügung gestellt.

Beide Fenster übergeben sich immer gegenseitig den Focus.
Erst ein "Schließen" des Haupt-Fensters ("tk") beendet das Wechselspiel.

Ein Nachteil der Message-Boxen ist sicher, dass sie immer gleichartig aussehen. Dafür kann man Informationen, Warnungen usw. usf. schnell und effektiv programmieren.
Einfache Message-Boxen (klassischerweise Info-Boxen) lassen sich temporär in Programme integrieren, um sich Zwischenwerten usw. anzeigen zu lassen. Dann muss man nicht jedes Mal das Layout des Programm-Fenster bemühen.

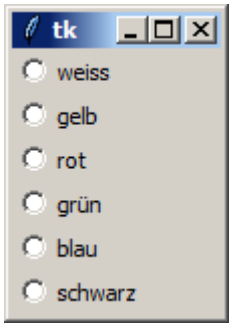
8.12.2.5. Checkbutton-Wdget's – Options-Felder

```
from Tkinter import *

master = Tk()

var = IntVar()

c = Checkbutton(master, text="Expand", variable=var)
c.pack()

mainloop()
```

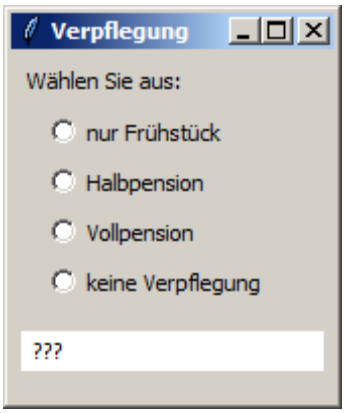
```
var = StringVar()
    c = Checkbutton(
        master, text="Color image", variable=var,
        onvalue="RGB", offvalue="L"
        )


v = IntVar()
    c = Checkbutton(master, text="Don't show this again", variable=v)
    c.var = v
```

8.12.2.6. Radiobutton-Widget – Options-Auswahl

```
from tkinter import *

fenster=Tk()

auswahl=IntVar()

Radiobutton(fenster, text="weiss",
            variable=auswahl, value=1).pack(anchor=W)
Radiobutton(fenster, text="gelb",
            variable=auswahl, value=2).pack(anchor=W)
Radiobutton(fenster, text="rot",
            variable=auswahl, value=3).pack(anchor=W)
Radiobutton(fenster, text="grün",
            variable=auswahl, value=4).pack(anchor=W)
Radiobutton(fenster, text="blau",
            variable=auswahl, value=5).pack(anchor=W)
Radiobutton(fenster, text="schwarz",
            variable=auswahl, value=6).pack(anchor=W)

mainloop()
```



```
from tkinter import *

# Ereignisverarbeitung
def gewaehlt():
    if wahl.get()==1:
        anzeige="Sie wählen nur Frühstück"
    elif wahl.get()==2:
        anzeige="Sie wählen Halbpension"
    elif wahl.get()==3:
        anzeige="Sie wählen Vollpension"
    elif wahl.get()==4:
        anzeige="Sie wählen keine Verpflegung"
    else:
        anzeige="Sie haben noch nicht gewählt"
    ergebnis.config(text=anzeige)

# Erzeugung des Fensters
tkFenster = Tk()
tkFenster.title('Verpflegung')
tkFenster.geometry('160x175')
# Aufforderungslabel
aufforderung=Label(master=tkFenster, text="Wählen Sie aus:", anchor='w')
aufforderung.place(x=5, y=5, width=140, height=20)
# Kontrollvariable
wahl=IntVar()
# Radiobutton Optionsauswahl
rb1=Radiobutton(master=tkFenster, anchor='w', text='nur Frühstück',
                value=1, variable=wahl, command=gewaehlt)
rb1.place(x=15, y=30, width=140, height=20)
rb2=Radiobutton(master=tkFenster, anchor='w', text='Halbpension',
                value=2, variable=wahl, command=gewaehlt)
rb2.place(x=15, y=55, width=140, height=20)
rb3=Radiobutton(master=tkFenster, anchor='w', text='Vollpension',
                value=3, variable=wahl, command=gewaehlt)
rb3.place(x=15, y=80, width=140, height=20)
rb4=Radiobutton(master=tkFenster, anchor='w', text='keine Verpflegung',
                value=4, variable=wahl, command=gewaehlt)
rb4.place(x=15, y=105, width=140, height=20)
# ev. Vorauswahl
# radiobutton3.select()
# Ergebnislabel
ergebnis=Label(master=tkFenster, bg='white', anchor='w', text=" ??? ")
ergebnis.place(x=5, y=140, width=150, height=20)
# Aktivierung des Fensters
tkFenster.mainloop()
```



Erzeugen einer Radiobutton-Auswahl aus einer Liste (von Tupeln) heraus:

```
MODES = [
        ("Monochrome", "1"),
        ("Grayscale", "L"),
        ("True color", "RGB"),
        ("Color separation", "CMYK"),
    ]

    v = StringVar()
    v.set("L") # initialize

    for text, mode in MODES:
        b = Radiobutton(master, text=text,
                        variable=v, value=mode)
        b.pack(anchor=W)
```

8.12.2.7. Text-Fenster / Text-Widget

```
from Tkinter import *             ## Tkinter importieren
root=Tk()                         ## Wurzelfenster!
textfenster = Text(root)          ## Ein Textfenster erzeugen
textfenster.pack()                ## und anzeigen
```

Zeilen von 1 bis y durchgezählt; Spalten von 0 bis x

.get('anfangzeile.anfangspalte','endezeile.endespalte')
.get('anfangzeile.anfangspalte','endezeile.end')
.get('anfangzeile.anfangspalte',END)

.insert(END, text)
.insert('einfügezeile.einfügespalte', text)
.insert('einfügezeile.end', text)

.delete('anfangzeile.anfangspalte','endezeile.endespalte')
.delete ('anfangzeile.anfangspalte','endezeile.end')
.delete ('anfangzeile.anfangspalte',END)

.see(indexzeile)
.see(END)
scrollt den Text, bis angegebene Zeile sichtbar ist

.yview(indexzeile)
.yview(END)
scrollt den Text, bis angegebene Zeile sichtbar ist
Index wandert nach oben (???)

.mark(markierungsname, 'zeile.spalte')
erstellt eine Markierung an der Position

.index(markierungsname)
gibt den Index einer Markierung zurück

.names()
liefert die Namen der verfügbaren Markierungen zurück

.search(suchtext, 'endezeile.endespalte')
.search(suchtext, 'endezeile.end')
.search(suchtext, END)

.tag_add(text, 'anfangzeile.anfangspalte','endezeile.end')
.tag_config(text, foreground=farbe)
.tag_names()

```
from Tkinter import *
root = Tk()
textfenster = Text(root)
textfenster.pack()
eingabe = Entry(root,width=60)
eingabe.pack(side=LEFT)
def hole():
     textfenster.insert(END, '\n' + eingabe.get())

but = Button(root,text='Hole', command = hole)
but.pack(side = LEFT)
root.mainloop()
```

weitere Teile für ein Chat-Programm (zusätzlich zu obigen Quelltext, bzw. Änderungen)

```
…
root = Tk()
def ende():
    root.destroy()

root.title('Chatten mit Python')
…

…
textfenster = ScrolledText(root,width=90)
textfenster.pack()
…
```

mehr: http://www.wspiegel.de/tkinter/tkinter02.htm

8.12.2.8. Frames – Group-Box's – Gruppen-Boxen

Frame-Widget
besser Container genannt,
beinhalten andere Bedien-Elemente, können so gruppiert angeordnet oder z.B. ein- und ausgeschaltet werden

```
from Tkinter import *

master = Tk()

Label(text="one").pack()

separator = Frame(height=2, bd=1, relief=SUNKEN)
separator.pack(fill=X, padx=5, pady=5)

Label(text="two").pack()

mainloop()
```

```
frame = Frame(width=768, height=576, bg="", colormap="new")
frame.pack()

video.attach_window(frame.window_id())
```

8.12.2.9. Menüs / Menu-Widget

```
from Tkinter import *

def callback():
    print "called the callback!"

root = Tk()

# create a menu
menu = Menu(root)
root.config(menu=menu)

filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)
filemenu.add_command(label="New", command=callback)
filemenu.add_command(label="Open...", command=callback)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=callback)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About...", command=callback)

mainloop()
```

```
root = Tk()

def hello():
    print "hello!"

# create a toplevel menu
menubar = Menu(root)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=root.quit)

# display the menu
root.config(menu=menubar)
```

```
root = Tk()

def hello():
    print "hello!"

menubar = Menu(root)

# create a pulldown menu, and add it to the menu bar
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="Open", command=hello)
filemenu.add_command(label="Save", command=hello)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)

# create more pulldown menus
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Cut", command=hello)
editmenu.add_command(label="Copy", command=hello)
editmenu.add_command(label="Paste", command=hello)
menubar.add_cascade(label="Edit", menu=editmenu)

helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="About", command=hello)
menubar.add_cascade(label="Help", menu=helpmenu)

# display the menu
root.config(menu=menubar)
```

```
root = Tk()

def hello():
    print "hello!"

# create a popup menu
menu = Menu(root, tearoff=0)
menu.add_command(label="Undo", command=hello)
menu.add_command(label="Redo", command=hello)

# create a canvas
frame = Frame(root, width=512, height=512)
frame.pack()

def popup(event):
    menu.post(event.x_root, event.y_root)

# attach popup to canvas
frame.bind("<Button-3>", popup)
```

```
counter = 0

def update():
    global counter
    counter = counter + 1
    menu.entryconfig(0, label=str(counter))

root = Tk()

menubar = Menu(root)

menu = Menu(menubar, tearoff=0, postcommand=update)
menu.add_command(label=str(counter))
menu.add_command(label="Exit", command=root.quit)

menubar.add_cascade(label="Test", menu=menu)

root.config(menu=menubar)
```

8.12.2.9.2. eine Tool-Bar einbauen

besteht aus einem Frame und Button's

```
from Tkinter import *

root = Tk()

def callback():
    print "called the callback!"

# create a toolbar
toolbar = Frame(root)

b = Button(toolbar, text="new", width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)

b = Button(toolbar, text="open", width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)

toolbar.pack(side=TOP, fill=X)

mainloop()
```

8.12.2.9.3. eine Status-Zeile (Status-Bar) einbauen

besteht aus einem Frame und Button's

```
class StatusBar(Frame):

    def __init__(self, master):
        Frame.__init__(self, master)
        self.label = Label(self, bd=1, relief=SUNKEN, an-
chor=W)
        self.label.pack(fill=X)

    def set(self, format, *args):
        self.label.config(text=format % args)
        self.label.update_idletasks()

    def clear(self):
        self.label.config(text="")
        self.label.update_idletasks()
```

```
status = StatusBar(root)
status.pack(side=BOTTOM, fill=X)
```

8.12.2.10. Umgang mit Standard-Dialogen

kommen direkt aus dem Betriebssystem
eigentlich gehören auch die Message-Boxen (→) mit dazu

Aufruf immer über Fehler-Behandlung empfehlenswert

```
try:
        fp = open(filename)
    except:
        tkMessageBox.showwarning(
            "Open file",
            "Cannot open this file\n(%s)" % filename
        )
        return
```

8.12.2.11. Listbox-Widget – Auswahl-Listen – List(en)-Boxen

```
from Tkinter import *

master = Tk()

listbox = Listbox(master)
listbox.pack()

listbox.insert(END, "a list entry")

for item in ["one", "two", "three", "four"]:
    listbox.insert(END, item)

mainloop()
```

```
listbox.delete(0, END)
listbox.insert(END, newitem)
```

```
lb = Listbox(master)
b = Button(master, text="Delete",
           command=lambda lb=lb: lb.delete(ANCHOR))
```

```
self.lb.delete(0, END) # clear
for key, value in data:
    self.lb.insert(END, key)
self.data = data
```

```
items = self.lb.curselection()
items = [self.data[int(item)] for item in items]
```

8.12.2.12. Options-Menüs – Auswahl-Schaltflächen

```
from tkinter import *

fenster=Tk()

auswahl=StringVar(fenster)
auswahl.set("weiss") # Vorgabe

optionsButton=OptionMenu(fenster,
                         auswahl,
                         "weiss", "gelb", "rot", "grün",
"blau", "schwarz")
optionsButton.pack()

def uebernehmen():
    print("Die Auswahl lautet(e): ", auswahl.get())
    fenster.quit()

buttonOK=Button(fenster, text="OK", command=uebernehmen)
buttonOK.pack()

mainloop()
```

Die print-Anweisung wird im IDLE-Fenster realisiert.

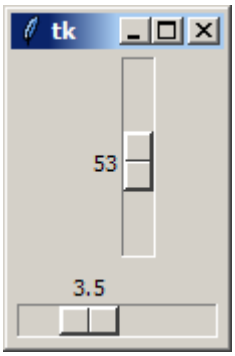
erstellen eines Options-Menüs aus einer Liste von Optionen

```
from Tkinter import *

# the constructor syntax is:
# OptionMenu(master, variable, *values)

OPTIONS = [
    "egg",
    "bunny",
    "chicken"
]

master = Tk()

variable = StringVar(master)
variable.set(OPTIONS[0]) # default value

w = apply(OptionMenu, (master, variable) + tuple(OPTIONS))
w.pack()

mainloop()
```

8.12.2.13. Scale-Widget – Gleiter / Regler

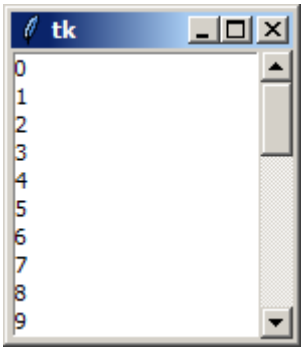
auch slider gnannt

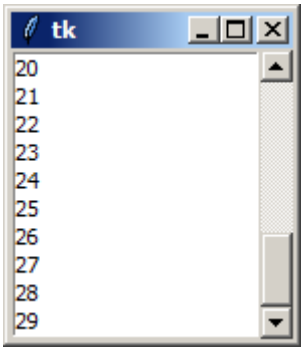
```
from tkinter import *

fenster=Tk()

schieber1=Scale(fenster, from_=0, to=100)
schieber1.pack()

schieber2=Scale(fenster, from_=0, to=12,
                resolution=0.5, orient=HORIZONTAL)
schieber2.pack()

mainloop()
```



8.12.2.14. Scrollbar-Widget - Bildlaufleisten

```
from tkinter import *

fenster = Tk()

laufleiste=Scrollbar(fenster)
laufleiste.pack(side=RIGHT, fill=Y)

auswahlListenBox=Listbox(fenster,
                         yscrollcommand=laufleiste.set)
for nummer in range(30):
    auswahlListenBox.insert(END, str(nummer))
auswahlListenBox.pack(side=LEFT, fill=BOTH)

laufleiste.config(command=auswahlListenBox.yview)

mainloop()
```





8.12.2.15. Widget x

8.12.x. Tkinter – stark, stärker, noch stärker Objekt-orientiert.

Alles was wir bisher mit Tkinter gemacht haben, war schon Objekt-orientiert. Wir benutzten Objekte, wie z.B. Tkinter selbst oder Labels und Buttun und die dazugehörigen Methoden. Wenn man es von Anfang an so macht, dann ist es auch irgendwie gar kein Problem. Wird man aber richtig Objekt-orientiert, dann ist die Welt für den Einsteiger-Programmierer schon schwerer zu durchschauen. Wer die nachfolgenden Programme und Erklärungen nicht gleich versteht, kann vielleicht erst einmal die Grundlagen der Objekt-orientierten Programmierung konsumieren. Ein Rücksprung hierher ist dann gut möglich und macht dann auch wieder Spaß. Graphische Oberflächen machen eben einfach die schöneren Programme.

8.12.x.1. nochmal "Hello Welt!"

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.QUIT = tk.Button(self, text="QUIT", fg="red",
                                            command=root.destroy)
        self.QUIT.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

einfache Dialoge

```
from Tkinter import *

class MyDialog:

    def __init__(self, parent):

        top = self.top = Toplevel(parent)

        Label(top, text="Value").pack()

        self.e = Entry(top)
        self.e.pack(padx=5)

        b = Button(top, text="OK", command=self.ok)
        b.pack(pady=5)

    def ok(self):

        print "value is", self.e.get()

        self.top.destroy()


root = Tk()
Button(root, text="Hello!").pack()
root.update()

d = MyDialog(root)

root.wait_window(d.top)
```

```
import tkSimpleDialog

class MyDialog(tkSimpleDialog.Dialog):

    def body(self, master):

        Label(master, text="First:").grid(row=0)
        Label(master, text="Second:").grid(row=1)

        self.e1 = Entry(master)
        self.e2 = Entry(master)

        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1, column=1)
        return self.e1 # initial focus

    def apply(self):
        first = int(self.e1.get())
        second = int(self.e2.get())
        print first, second # or something
  ...
    def apply(self):
        first = int(self.e1.get())
        second = int(self.e2.get())
        self.result = first, second

d = MyDialog(root)
print d.result
```

Überprüfung von Eingaben

```
...

    def apply(self):
        try:
            first = int(self.e1.get())
            second = int(self.e2.get())
            dosomething((first, second))
        except ValueError:
            tkMessageBox.showwarning(
                "Bad input",
                "Illegal values, please try again"
            )

    ...

    def validate(self):
        try:
            first= int(self.e1.get())
            second = int(self.e2.get())
            self.result = first, second
            return 1
        except ValueError:
            tkMessageBox.showwarning(
                "Bad input",
                "Illegal values, please try again"
            )
            return 0

    def apply(self):
        dosomething(self.result)
```

```
from Tkinter import *
import os

class Dialog(Toplevel):

    def __init__(self, parent, title = None):

        Toplevel.__init__(self, parent)
        self.transient(parent)

        if title:
            self.title(title)

        self.parent = parent

        self.result = None

        body = Frame(self)
        self.initial_focus = self.body(body)
        body.pack(padx=5, pady=5)

        self.buttonbox()

        self.grab_set()

        if not self.initial_focus:
            self.initial_focus = self

        self.protocol("WM_DELETE_WINDOW", self.cancel)

        self.geometry("+%d+%d" % (parent.winfo_rootx()+50,
                                  parent.winfo_rooty()+50))
```

```
        self.initial_focus.focus_set()

        self.wait_window(self)

    #
    # construction hooks

    def body(self, master):
        # create dialog body.  return widget that should have
        # initial focus.  this method should be overridden

        pass

    def buttonbox(self):
        # add standard button box. override if you don't want the
        # standard buttons

        box = Frame(self)

        w = Button(box, text="OK", width=10, command=self.ok, default=ACTIVE)
        w.pack(side=LEFT, padx=5, pady=5)
        w = Button(box, text="Cancel", width=10, command=self.cancel)
        w.pack(side=LEFT, padx=5, pady=5)

        self.bind("<Return>", self.ok)
        self.bind("<Escape>", self.cancel)

        box.pack()

    #
    # standard button semantics

    def ok(self, event=None):

        if not self.validate():
            self.initial_focus.focus_set() # put focus back
            return

        self.withdraw()
        self.update_idletasks()

        self.apply()

        self.cancel()

    def cancel(self, event=None):

        # put focus back to the parent window
        self.parent.focus_set()
        self.destroy()

    #
    # command hooks

    def validate(self):

        return 1 # override

    def apply(self):

        pass # override
```

Check-Boxen

```
def __init__(self, master):
        self.var = IntVar()
        c = Checkbutton(
            master, text="Enable Tab",
            variable=self.var,
            command=self.cb)
        c.pack()

    def cb(self, event):
        print "variable is", self.var.get()
```

weiterführende und Quell-Links:
http://www.python-kurs.eu/python_tkinter.php (tolles Tutorial)
http://www.wspiegel.de/tkinter/tkinter_index.htm (kurzes, aber informatives Tutorial)

Tk-Geometrie-Manager

-
-
-

```
>>>
```

8.12.x. diverse Tkinter-Beispiele

aus verschiedenen Quellen:

```
#grafik1.py
from Tkinter import *   #alle Funktionen des Moduls Tkinter werden importiert
fenster = Tk()          #Ein Objekt der Klasse Tk mit Namen fenster wird eingerichtet
fenster.mainloop()      #Die Methode mainloop aktiviert ein Tk-Fenster
```

```
#grafik2.py
from Tkinter import *
fenster= Tk()
fenster.etikett= Label(master=fenster,text= 'Hallo!')      #Ein Objekt der Klasse Label
                                                           #mit Namen fenster.etikett wird erzeugt.
fenster.etikett.pack()                                     #Mit der Methode pack() wird das neue
                                                           #Objekt etikett in die Darstellung des
                                                           #Anwendungsfensters fenster eingebaut.
fenster.mainloop()
```

```
#grafik3.py
from Tkinter import *
fenster= Tk()
fenster.etikett= Label(master=fenster,text= 'Hallo!',
font=('Comic Sans MS',14),fg='blue')           #als Schrifttyp wird Comic Sans MS
                                               #in der Schriftgröße 14;
                                               #Schriftfarbe ist Blau
fenster.etikett.pack()
fenster.title ('Formen')                       #Ueberschrift
leinwand=Canvas(fenster,width=800,height=600,bg="yellow")  #Mithilfe von Canvas-Objekten
                                               #werden Kreise, Rechtecke, Linien
                                               #oder Textobjekte generiert
leinwand.pack()
rechteck=leinwand.create_rectangle(40,20,160,80,fill="Moccasin")
kreis=leinwand.create_line(270,290,450,350,width=10,fill="Lightblue")
vieleck=leinwand.create_polygon(500,80,500,120,600,120,500,80, fill ="white")
streckenzug=leinwand.create_line(270,290,450,350,300,200,
arrow = LAST, width =10, fill = "blue")        #andere Werte fuer arrow: #FIRST, BOTH
spruch=leinwand.create_text(300,50,text="Aller Anfang ist schwer!"),
font=('Arial',14), fill="green")
fenster.mainloop()
```

Q: ???

8.13. Internet

8.13.x. Python und das http-Protokoll

Variante 1

```
import requests

adresse="http://www.lsp-dre.de"

antwort = requests.get(adresse)

print(antwort.ststus_code)
print(antwort.headers['content-type'])
print(antwort.encoding)
print(antwort.text[:80])
```

Variante 2

```
import urllib.request
adresse="http://www.lsp-dre.de"
seite=urllib.request.urlopen(adresse)
seiteninhalt=seite.read()
print(seiteninhalt)
seite.close()
```

Wichtig ist es, zumindestens für das erste Ausprobieren, eine einfache Internetseite abzufragen. Ansonsten kann die Antwort etlige Seiten lang sein. Im Beispiel-Fall ist das eine ganz einfach gestrickte Umleitung auf eine andere Internetseite.

```
>>>
b'\xef\xbb\xbf<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">\n<HTML>\n<HEAD>\n  <TITLE>lern-
soft-projekt: drews</TITLE>\n</HEAD>\n\n<BODY BGColor=#3333FF
Text=#FFFF99 Link=#FFFFFF VLink=#FFFF66>\n<H2>Homepage lern-soft-
projekt: drews</H2><HR>\n<P>Derzeit wird diese Domain nicht bedient.
Nutzen Sie bitte:<P>\n<DIV align="center"><H2><A href="http://www.lern-
soft-projekt.de/">www.lern-soft-
projekt.de</A></H2></DIV>\n</BODY>\n</HTML>'
>>>
```

Natürlich können wir nun den zurückgelieferten HTML-Text auch weiterverwenden.
Wollten wir einen Browser programmieren, müssten wir jetzt nach und nach alle Tags auswerten und in eine Seiten-Darstellung umsetzen. Dabei sollte dann das herauskommen, was uns ein anderer Browser (Internet-Explorer, Firefox, Opera, Chrome, Safari, …) uns auch liefern würde.
Mit Text wird das vielleicht noch recht einfach gehen, aber spätestens bei Bildern, Videos usw. usf. sind dann schon erweiterte Programmierkenntnisse notwendig.
Wir wollen den HTML-Text einfach nach dem Seiten-Titel durchsuchen. Darüber sollte eigentlich jede Internet-Seite verfügen.

die Beispiel-Seite im Browser Firefox

Wer sich schon mit HTML beschäftigt hat, der weiss, dass der Seiten-Titel zwischen den Tags <TITLE> und </TITLE> zu finden ist. Genau danach wollen wir jetzt suchen.

```
…
startpos=0
while True:
    startpos=seiteninhalt.find("<TITLE>",startpos)
    if startpos == -1:
        break
    endepos=seiteninhalt.find("</TITLE>",startpos)
    if endepos == -1:
        break
    print("gefundener Text:
",seiteninhalt[(startpos+7):endepos])
    startpos=endepos
print("Suche beendet!")
```

```
>>>
gefundener Text:  lern-soft-projekt: drews
Suche beendet!
>>>
```

Wenn wir das Programm ein wenig umgestalten, dann kann auch nach jedem anderen beliebigen Begriffspaar gesucht werden. Ob das Tags sind oder andere Begriffe, ist dabei egal.

```
…
print("-----------")
print("Suche:")
print("")startpos=0
starttag="<H2>"
endetag="</H2>"
while True:
    startpos=seiteninhalt.find(starttag,startpos)
    if startpos == -1:
        break
    endepos=seiteninhalt.find(endetag,startpos)
    if endepos == -1:
        break
    print("Text zwischen ",starttag," und ",endetag,":
",seiteninhalt[(startpos+len(starttag)):endepos])
    startpos=endepos
print("Suche beendet!")
```

```
…
-----------
Suche:

gesuchter Text zwischen  <H2>  und  </H2> :  Homepage lern-soft-projekt: drews
gesuchter Text zwischen  <H2>  und  </H2> :  <A href="http://www.lern-soft-
projekt.de/">www.lern-soft-projekt.de</A>
Suche beendet!
>>>
```

8.13.x. einfacher Web-Server

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import os

os.chdir("/tmp")

# CGIHTTP-Server auf Port 8080 starten
server = HTTPServer(("",8080), CGIHTTPRequestHandler)
server.serve_forever()
```

passendes CGI-script "cgi_test" unter /tmp abgespeichert

```
echo 'Content-Type: text/plain; charset=UTF-8'
echo
echo 'Hallo Welt!'
```

oder als Python-script "cgi_test.py" unter /tmp abgespeichert

```
print('''Content-Type: text/plain; charset=UTF-8
Hallo Welt!''')
```

8.13.x. Python und die eMail-Protokolle (smtp, pop3, imap)

```
# -*- coding: utf8 -*-

  # Mail-Versand mit dem Standard-Modul smtplib

  # Module smtplib und sys importieren
  import smtplib, sys

  # MIMEText aus dem Modul text des Sub-Pakets email.mime des Pakets email
  # importieren;
  # im Dateisystem z.B. unter /usr/lib64/python2.6/email/mime/text.py
  from email.mime.text import MIMEText

  # unser ASCII-Mailtext
  mail_text = '''
  Hello friends,
  this is a simple ASCII mail.
  '''

  # eine MIMEText-Nachricht erstellen
  msg = MIMEText(mail_text)

  # Header setzen
  msg['Subject'] = 'test mail'
  me = msg['From'] = 'otto@hrz.tu-chemnitz.de'
  you = msg['To'] = 'hot@hrz.tu-chemnitz.de'

  # Mail senden
  s = smtplib.SMTP()
  if len(sys.argv) > 1 and sys.argv[1] == 'd':
    # Kommandozeilenargument 1 lautet "d", daher Debug einschalten
    s.set_debuglevel(1)
  #s.connect(host = 'mailbox.hrz.tu-chemnitz.de')
  s.connect()
  s.sendmail(me, [you], msg.as_string())
  s.close()
```

Q: https://www-user.tu-chemnitz.de/~hot/PYTHON/

8.14. besondere mathematische Möglichkeiten in Python

8.14.1. imaginäre Zahlen

Notierung
2 + 3j
(2 + 3j)
complex(2,3**)**

jede Variable kann auch eine imaginäre Zahl beinhalten:
img_zahl = 2.5 - 1.5j
img-zahl**.real** liefert den Real-Teil, also hier 2,5
img_zahl**.imag** liefert den Imaginär-Teil, also hier -1,5 ➔ -1,5i

8.14.2. Matrizen (Matrixes)

Ob es nun Matrizen oder Matrixes heißt, wollen wir hier nicht vertiefen. Ich benutze Matrix für die Einzahl und Matritzen für die Mehrzahl. Das liesst sich aus meiner Sicht einfacher und jeder halbwegs (mathematisch) Eingeweihte, weiss, worum es geht.

Realisierung und Bearbeitung z.B. über geschachtelte Listen (s.a. kurze Einführung: → 8.4. Listen, die I. – einfache Listen)

```
def transponiere(matrix, bisIndex):
    for i in range(bisIndex):
        for j in range(i+1, bisIndex):
            matrix[i][j],matrix[j][i] = matrix[j][i],matrix[i][j]
    return matrix


def testeTransponieren(n):
    matrix= range(n)
    for i in range(n):
        matrix[i]=range(n)
    print("(Original-)Matrix")
    for i in range(n):
        print(matrix[i])
    transpoMatrix=transponiere(matrix)
    print("")
    print("transponierte Matrix")
    for i in range(n):
        print(transpoMatrix[i])
```

```
def multipliziere(matrix1, matrix2):
    laengeM1=len(matrix1)
    multMatrix=range(laengeM1)
    for i in range(laengeM1):
        multMatrix[i]=range(laengeM1)
        for j in range(laengeM1):
            multMatrix[i][j]=0
    for i in range(laengeM1):
        for j in range(laengeM1):
            summme=0
            for k in range(laengeM1):
                summe+=matrix1[i][k]*matrix2[k][j]
            multMatrix[i][j]=sum
    return multMatrix


def testeMultipizieren(laengeMatrix):
    matrixA=range(laengeMatrix)
    matrixB=range(laengeMatrix)
    print(matrixA)
    print(matrixB)
    for i in range(laengeMatrix):
        matrixA[i]=range(laengeMatrix)
        matrixB[i]=range(laengeMatrix)
        for j in range(laengeMatrix):
            matrixA[i][j]=i
            matrixB[i][j]=i
    print(matrixA)
    print(matrixB)
    matrixC=multipliziere(matrixA,matrixB)
    print(matrixC)
```

```
def berechneDeterminante(matrix):
    laengeM=len(matrix)
    if laengeM<=0:
        return 1
    else:
        if laengeM==1:
            return matrix[0][0]
        else:
            summe=0
            neg=-1
            for i in range(laengeM):
                neg=(-1)*neg
                matrixH=matrixcopy(matrix)
                for j in range(laengeM):
                    matrixH.pop(0)
                matrixH-pop(i)
                sum+=neg*matrix[i][0]*berechneDeterminante(matrixH)
    return summe
```

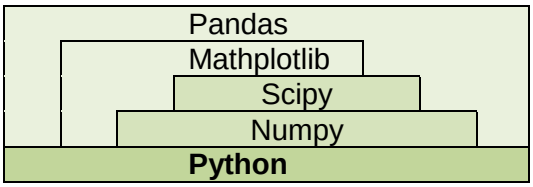
dieser Algorithmus hat eine Laufzeit von $O(2^n)$, es existiert aber auch einer mit $O(n^3)$

8.14.3. Python numerisch, Python für Big Data

auch für Data science, Maschinelles Lernen, Künstliche Intelligenz, …

numerisches Programmieren umfasst einen breiten Teil der Mathematik und meint das Arbeiten mit stetigen Variablen, numerische Analysen, Approximations-Algorithmen, …

in vielen Punkten Ersatz für das kostenpflichtige Matlab
hier genannte Module alle kostenfrei
keine Einschränken durch prohibitive / proprietäre Lizenzen bei der Weiterverwendung



Numpy

stellt die grundlegenden Daten-Typen für numerische Arbeiten und das Händling von Big Data zur Verfügung
dazu gehören mehrdimensionale Array's und Matrizen

Scipy

benutzt die Daten-Typen aus Numpy
bietet vor allem Funktionalitäten für Analysen usw. usf. an, wie z.B. Regression, FOURIER-Transformation, …

Mathplotlib

bietet Möglichkeiten der graphischen Darstellung von Daten an

```
1 import mathplotlib.pyplot as zeichnung
2
3 x = [1,2,3,4,5,6]
4 y = [1,4,9,16,25,36]
5 zeichnung.plot(x,y)
6 zeichnung.title("quadratische Funktion")
7 zeichnung.show()
```

Pandas

nutzt alle genannten Module
erweitert diese für Tabellen und Zeit-Reihen

8.15. Behandlung von Laufzeitfehlern – Exception's

try … except … else

Bsp: Zahlenraten
Computer wählt zufällig eine Zahl aus einem Zahlenbereich aus, hier 1 bis 100
der Nutzer soll die Zahl raten; der Computer gibt bein nicht-zutreffen zurück, ob die Zahl zu groß ode zu klein ist; Ziel sind besonders wenige Rate-Vorgänge zu brauchen.

```
from random import randint

geraten=False
SuchZahl=randint(1,100)
print("Der Computer hat eine Zahl erwürfelt?")
print()
zaehler=0
while not geraten:
    # Eingabe
    try:
        eing=int(input("Welche Zahl vermutest Du?: "))
    except ValueError:
        print("")
        continue    # --> Eingabe wiederholen
    # Auswertung
    zaehler+=1
    if eing > Suchzahl:
        print("vermutete Zahl ist zu groß!")
    elif ein < Suchzahl:
        print("vermutete Zahl ist zu klein!")
    else:
        geraten=True
print("Richtig!  ",zaehler," Versuche gebraucht")
print("Spiel-Ende")
```

Die Auswertung könnte auch im optionalen ELSE-Zweig stehen können.

```
>>>

>>>
```

try … except … finally

Bsp: Zahlenraten
Computer wählt zufällig eine Zahl aus einem Zahlenbereich aus, hier 1 bis 100
der Nutzer soll die Zahl raten; der Computer gibt bein nicht-zutreffen zurück, ob die Zahl zu groß ode zu klein ist; Ziel sind besonders wenige Rate-Vorgänge zu brauchen.

```
from random import randint

geraten=False
SuchZahl=randint(1,100)
print("Der Computer hat eine Zahl erwürfelt?")
print()
zaehler=0
while not geraten:
    # Eingabe
    try:
        eing=int(input("Welche Zahl vermutest Du?: "))
    except ValueError:
        print("")
        continue    # --> Eingabe wiederholen
    # Auswertung
    zaehler+=1
    if eing > Suchzahl:
        print("vermutete Zahl ist zu groß!")
    elif ein < Suchzahl:
        print("vermutete Zahl ist zu klein!")
    else:
        geraten=True
print("Richtig!  ",zaehler," Versuche gebraucht")
print("Spiel-Ende")
```

```
>>>

>>>
```

try … finally

raise

pass

leere Anweisung; z.B. als Platzhalter in definierten, aber noch nicht implementierten Funktionen / Klassen / …

8.16. Sortieren – eine Wissenschaft für sich

Dieses Kapitel könnte genauso gut unter dem Abschnitt (→) eingeordnet werden. Was durch den Anfänger vielleicht als überzogene, abgehobene, akademische Auseinandersetzung abgetan wird, ist in der Informatik ein Kernproblem: Wie bekommt man schnell und mit möglichst wenig Speicher-Aufwand eine Liste / ein Feld von Daten sortiert.
Die Algorithmik liefert viele Lösungen mit unterschiedlichen Vor- und Nachteilen. Einige Sortier-Verfahren wollen wir hier vorstellen und unter bestimmten Kriterien bewerten.
Für Anfänger-Listen-Größen von vielleicht maximal einigen hundert Werten machen die Unterschiede meist nicht viel aus. Bei großen - ev. sogar mehrdimensionalen Daten-Strukturen – bekommen die Kriterien dann schon eine andere Bedeutung.
Wir wollen hier versuchen die einzelnen Algorithmen nicht nur zu nennen, sondern auch zu erklären und an einer Beispiel-Datenreihe anzuwenden.
Wenn es geht und wenn es sinnvoll ist, dann werden wir auch die Zwischen-Zustände mittels eines abgewandelten Programm anzuzeigen, um das Verfahren auch in der Praxis zu erleben. Solche Zwischen-Anzeigen bieten sich auch an, wenn man einen Algorithmus auf die eigenen Daten-Strukturen anpasst. Selten geht alles beim ersten Mal glatt.
Anfängern sei empfohlen, sich zuerst einmal den Algorithmus herauszusuchen, bei dem man den Eindruck hat, man versteht ihn und die Umsetzung ins Programm. Dadurch wird die Fehlersuche vereinfacht. Später kann man sich dann den höheren Verfahren zuwenden.

8.16.x. Bubble-Sort

```
 1 def bubblesort(liste):
 2     laenge = len(liste)
 3     for i in range(laenge):
 4         geaendert = False
 5         for j in range(laenge-i-1):
 6             if liste[j] > liste[j+1]:
 7                 liste[j],liste[j+1] = liste[j+1],liste[j]
 8                 geaendert = True
 9         if not geaendert:
10             break
11     return liste
```

8.16.x. Quick-Sort

Anwendung des "Teile und herrsche"-Prinzips ("divide and conquer")
algemeines Prinzip zum Lösen von Problemen: Zerteile das Problem in kleinere und löse diese. Dabei darf das prinzip immer wieder angewendet werden, wir arbeiten also rekursiv
irgendwann sind die Teil-Probleme so klein, dass sie schon gelöst sind (Rekursions-Abbruch) oder einfach zu lösen sind

Quicksort besteht aus drei Elementen *(noch nicht perfekt!)*

- **Herrsche / Beauftragen / Befehlen** — wenn die Liste länger als ein Element ist, dann wird sie nach Teilen-Prinzip bearbeitet, ansonsten ist die Liste sortiert
  es kann das Zusammenfügen / Ausgeben erfolgen
- **Teilen** — aus der Liste wird (zufällig) ein Element elem ausgewählt und die Liste in zwei Teillisten zerlegt, wobei Liste1 alle kleineren Elemente als elem enthält und Liste 2 alle größeren oder gleichgroßen (/ anderen)
  die Teillisten werden dem Herrsche-Prinzip zur Prüfung übergeben
- **Zusammenfügen** — die sortierte Liste wird nun zusammengesetzt aus der sortierten Liste der kleineren Elemente, dem Element elem und der sortierten Liste der größeren Elemente

der Algorithmus stammt von HOARE 1962 ist einer der effektiven Sortier-Verfahren
besonders herausragend ist die Zeit-Effektivität

es werden durchschnittlich n log n Vergleiche benötigt

```
def quicksort(liste):

    def teile(links, rechts):
        i = links
        j = rechts - 1
        pivot = liste[rechts]

        while True:
            while liste[i] <= pivot and i < rechts:
                i+=1
            while liste[j] >= pivot and j > links
                j-=1
            if i < j:
                liste[i], liste[j] = liste[j], liste[i]
            else:
                break
        if liste[i] > pivot:
            liste[i], liste[rechts] = liste[rechts], liste[i]
        return i

    def sortieren(links,rechts):
        if links < rechts:
            teiler = teile(links,rechts)
            sortieren(links, teiler-1)
            sortieren(teiler+1, rechts)

    sortieren(0, laenge-1)
    return liste
```

ein Quick-Sort mit Anzeige

```
def quicksort(liste):
    if len(liste)>0:
        print("es wird sortiert: ", liste')
    if len(liste)<=1:
        return liste
    else:
        return quicksort([i for i in liste[1:] if i < liste[0])\
        + [liste[0]]\
        + quicksort([j for j in liste[1:] if j >= s[0]])
```

etwas kryptisch , aber auch so geht es:

```
def quicksort(liste):
    if len(liste) <= 1:
        return liste
    wahlelement = liste.pop()
    links  = [element for element in liste if element < wahlelement]
    rechts = [element for element in liste if element >= wahlelement]
    return quicksort(links) + [wahlelement] + quicksort(rechts)
```

8.16.x. Tree-Sort

8.16.x. Merge-Sort

```
def mergesort(liste):
    def mische(links, rechts):
        gemischt = []
        laengeLinks = len(links)
        laengeRechts = len(rechts)
        while laengeLinks != 0 and laengeRechts != 0:
            if links[0] <= rechts[0]:
                gemischt.append(links[0])
                links = links[1:]
            else:
                gemischt.append(rechts[0])
                rechts = rechts[1:]
        while laengeLinks !=0:
            gemischt.append(links[0])
            links = links[1:]
        while laengeRechts !=0:
            gemischt.append(rechts[0])
            rechts = rechts[1:]
        return gemischt

    def sortieren(liste):
        laenge = len(liste)
        if laenge<=1:
            return liste
        else:
            haelfte=laenge/2
            links = liste[0:haelfte]
            rechts = liste[haelfte:]
            links = sortieren(links)
            rechts = sortieren(rechts)
        return mische(links, rechts)

    return sortieren(liste)
```

8.16.x. Selection-Sort

```
def selectionsort(liste):
    laenge = len(liste)
    for i in range(laenge-1):
        minimum = i
        for j in range(i,laenge):
            if liste[j] < liste[minimum]:
                minimum = j
        liste[minimum],liste[i] = liste[i],liste[minimum]
    return liste
```

8.16.x. Insertion-Sort

```
def insertionsort(liste):
    laenge = len(liste)
    for i in range(1,laenge):
        wert = liste[i]
        j = i
        while j > 0 and liste[j-1] > wert:
            liste[j] = liste[j-1]
            j-=1
        liste[j] = wert
    return liste

```

8.16.x. Gnome-Sort

```
def gnomesort(liste):
    pos = 0
    laenge = len(liste)
    while pos < laenge-1:
        i = pos
        if liste[i] <= liste[i+1]:
            pos+=1
        else:
            liste[i], liste[i+1] = liste[i+1], liste[i]
            if pos !=0:
                pos-=1
            else:
                pos+=1
    return liste
```

8.16.x. Counting-Sort

```
def countingsort(liste):
    laenge = len(liste)
    if laenge == 0:
        return []
    listeA = [0] * (max(liste)+1)
    listeB = [""] * laenge
    for elem in liste:
        listeB[elem]+=1
    for i in range(1,len(listeB)):
        listeB[i]+=listeB[i-1]
    for elem in liste[::-1]:
        listeA[listeB[elem]-1] = elem
        listeB[elem]-=1
    return listeA
```

8.16.x. Radix-Sort

```
def radixsort(liste, k=10, d=0):
    laenge = len(liste)
    if laenge == 0:
        return []
    elif d == 0:
        d = max(map(lambda x: len(str(abs(x))), liste)
    for x in range(d):
        listeA = [[] for i in range(k)]
    for elem in liste:
        listeA[(elem / 10**x) % k].append(elem)
    liste = []
    for bereich in listeA:
        liste.extend(bereich)
    return liste
```

8.16.x. Tim-Sort

8.16.x. Heap-Sort

```
def heapsort(liste):

    return liste
```

8.16.x. Bucket-Sort

8.16.x. -Sort

8.16.x. Vergleich ausgewählter Sortier-Algorithmen

Algorithmus / Name	in place	stabil			Laufzeit-Verhalten B Best-Case	AVG durchschnittlich	W Worst-Case			
Selection-Sort	**ja**	**nein**			$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$			
Bubble-Sort	**ja**	**ja**			$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$			
Insertion-Sort	**ja**	**ja**			$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$			
Quick-Sort	**ja**	**nein**			$\Theta(n * \log(n))$	$\Theta(n * \log(n))$	$\Theta(n^2)$			
Heap-Sort	**ja**	**nein**			$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$			
Merge-Sort	**nein, (ja)**	**ja**			$\Theta(n * \log(n))$	$\Theta(n * \log(n))$	$\Theta(n * \log(n))$			
Tim-Sort	**nein**	**ja**			$\Theta(n)$	$O(n * \log(n))$	$O(n * \log(n))$			
Radix-Sort	**nein**	**ja**					$O(d * (n+k))$			
Counting-Sort	**nein**	**ja**					$O(n+k)$			
Bucket-Sort										

interessante Links:
https://www.toptal.com/developers/sorting-algorithms (Animationen zu den verschiedenen Sortier-Algorithmen)
https://www.youtube.com/watch?v=t8g-iYGHpEA (Sortierungen optisch und akustisch veranschaulicht)

(c,p) 2015 - 2020 lern-soft-projekt: drews

8.16.x. das Häufigste Element finden – der Modus

```
def mode(L):
    for i in range(0,100):
        for i in L:
            frequency[i] += 1
                return i
            if frequency[i] == max(frequency):
                frequency=[0]*10
```

```
>>>
```

interessante Links:
http://www.sortierkino.de (zum Zuschauen beim Sortieren; viele Algorithmen im Vergleich)

Beispiel-Implementierung

Q: https://github.com/MartinThoma/algorithms/blob/master/sorting.py

8.17. Nutzung weiterer (/ besonderer) graphischer Benutzer-Oberflächen

8.18. (die hohe Kunst der) Spiele-Programmierung

Nachdem wir einiges dazu schon beim Modul "pygame" besprochen haben (→ 8.9. das Modul "pygame") dringen wir nun noch etwas tiefer in den Sachvehalt ein.

interessante Links:
http://inventwithpython.com/inventwithpython_3rd.pdf (online-Version des Buches: AL SWEIGART: Invert Your Own Computer Games with Python 3rd Edition)
http://inventwithpython.com/makinggames.pdf (online-Version des Buches: AL SWEIGART: Making Games with Python & Pygame)

8.19. Python im Geheimen - Kryptologie

Begeben wir uns in die Welt von Alice und Bob, den beiden Haupt-Agenten in der Kryptologie.

8.19.0. Grundlagen

8.19.0.1. Codierung

"geheime" Codierungen

8.19.0.2. Chiffrierung

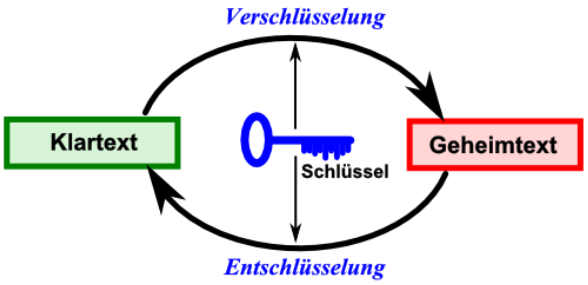
In den folgenden Kapiteln werden wir die Klartexte (unverschlüsselte Texte) grün oder grünlich hinterlegt darstellen. Wenn die Buchstaben-Art keine Rolle spielt, dann werden die Klatrtexte mit Groß-Buchstaben geschrieben.
Die verschlüsselten Texte (Geheimtexte) werden dagegen in Klein-Buchstaben in rot oder rötlich hinterlegt notiert.
Das Standard-Alphabet sind die 26 deutschen Buchstaben ohne Umlaute und ß. Oft wird auch auf das Leerzeichen verzichtet und die Wörter einfach hintereinander geschrieben. Erweiterte Alphabete nutzen Leerzeichen und / oder Ziffern und / oder Satz-Zeichen mit dazu. So etwas definieren wir dann bei den einzelnen Verfahren. Viele Algorithmen sind so ausgelegt, dass sie Nicht-Alphabet-Zeichen einfach ignorieren oder direkt übernehmen.

I.A. geht es vor allem um das Demonstrieren des Verfahren's. Der wichtigste Grund für die Wahl der standardisierten Alphabete ist aber die Vergleichbarkeit der verschiedenen Verfahren. Dabei interessiert uns immer das Agieren der Gegenseite. Kann Sie das Verfahren knacken?

Nicht's ist unangenehmer als eine Geheimschrift, von der man glaubt, sie sein Bomben-sicher und jeder kann aber in der Praxis mit wenig Aufwand mitlesen.

Die Einteilung von Geheim-Schriften / -Verfahren ist ein unendliches Thema. Wir beschränkn uns hier auf zwei elementare Möglichkeiten.

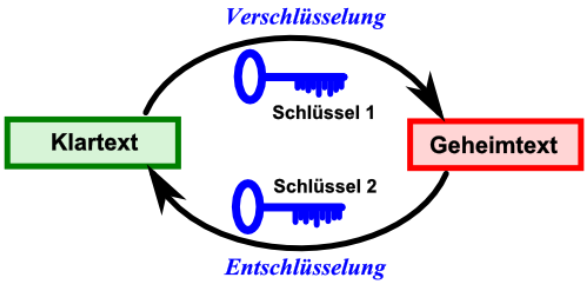
Für die erste Einteilung betrachtet man die Anzahl der verwendeten Schlüssel und das benutzte Verfahren. Wird nur ein Schlüssel und praktisch das gleiche verfahren für Ver- und Ent-Schlüsselung benutzt, dann sprechen wir von **symmetrischer Verschlüsselung**. Klassische Vertreter sind die CÄSAR-Chiffre (→) und (→).



symmetrische Verschlüsselung

Kommen dagegen zwei (zueinander gekoppelte) Schlüssel und praktisch auch zwei Verfahren zum Einsatz, dann handelt es sich um die **asymmetrische Verschlüsselung**.

Beispiele hierfür sind das RSA-Verfahren oder der DES-Algorithmus.



asymmetrische Verschlüsselung

Die zweite Einteilung bezieht sich auf die Art und Weise, wie der Geheimtext erzeugt wird.So kann man z.B. Geheimtexte durch Austauschen der Symbole erzeugen. Wir sprechen hier von **Substitution**. Typische Umsetzungen sind (→) und (→).

Eine weitere weitere Möglichkeit – Texte unleserlich zu machen – sind **Transposition**en. Hierbei bleiben die Symbole des Klartextes erhalten, aber ihre Positionen innerhalb des Textes werden verändert. (→) und (→) sind hier viel zitierte Chiffren.

Die dritte Art verändert die Klartext durch Hinzufügen von Symbolen. Dabei geht es zum Einen darum die Texte unleserlich (oder schwer leserlich) zu machen und zu Anderen sollen Häufigkeits-Analysen ausgetrickst werden. Kryptographen nennen diese Art der Geheimtext-Erzeugung **Erweiterung**. Als typische Vertreter dieser Gruppe können (→) und (→) genannt werden.

Aufgaben:

1. *Vergleichen Sie symmetrrische und asymmetrische Verschlüsselung! Nutzen Sie auch das Internet, um weitere typische Merkmale, Vor- und Nachteile zu erkunden!*
2. *Vergleichen Sie die Erzeugung von Geheimtexten durch Substitution, Transposition und Erweiterung anhand von jeweils mindestens 6 selbstgewählten Kriterien! Versuchen Sie gleichrangig Gemeinsamkeiten und Unterschiede zu finden!*
3. *Informieren Sie sich über weitere Einteilungs-Möglichkeiten und stellen Sie eine in Form eines kurzen Vortrages vor!*

Aufgaben:

4. *Erstellen Sie ein Stammbaum von Geheimsprachen und stellen Sie diese vor!*

8.19.1. symmetrische Verschlüsselung

Symmetrische Verschlüsselungen benutzen für die Ver- und Ent-Schlüsselung (Chiffrierung / Dechiffrierung) immer den gleichen Schlüssel. In praktischallen Fällen kann das gleiche – oder auch das umgekehrte (reverse) – Verfahren genutzt werden.
Das macht symmetrische Verfahren sehr effektiv. Mit Computern können sie sehr einfach umgesetzt werden. Das große Problem sind die Schlüssel. Sie müssen irgendwann ausgetauscht werden. Dieser Vorgang kann von Unbefugten mitgehört / manipuliert / … werden.
Viele der älteren symmetrischen Verfahren bieten durch eine recht geringe Schlüssel-Anzahl auch keine ausreichende Sicherheit mehr. Mit modernen Rechnern sind sie oft innerhalb weniger (milli-)Sekunden durch Brute-Force-Angriffe oder Häufigkeits-Analysen angreifbar.

8.19.1.x. CÄSAR-Verschlüsselung

Das Verfahren geht der Legende nach auf Gaius Julius CÄSAR (100 – 44 v.u.Z) zurück. Zu jener Zeit soll die Chiffre auch nicht gebrochen worden sein. Dazu gab es wahrscheinlich auch zu wenige Menschen, die sich mit Schrift und Alphabet auskannten.
CÄSAR's Verschlüsselung war einfach und effektiv. Er setzte dem Klartext-Alphabet ein zweites gegenüber, dass um 3 Positionen verschoben war. Buchstaben, die keine Entsprechung hatten, wurde an der anderen Seite angelegt.
Wahrscheinlich benutzte CÄSAR auch nur eine Möglichkeit – und zwar eben diese CÄSAR3-Verschiebung.

Beispiel: CÄSAR**3**

Symbole lfd. Nr.	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Klaralphabet	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Geheimalph.	**y**	**z**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**

Dadurch entsteht ein Buchstaben-Ring. Für andere CÄSAR-Verschlüsselungen wird der Ring einfach weitergeschoben.

Beispiel: CÄSAR**7**

Symbole lfd. Nr.	1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Klaralphabet	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Geheimalph.	**u**	**v**	**w**	**x**	**y**	**z**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**

Auch der Nachfolger von CÄSAR – der Kaiser AUGUSTUS – benutzte eine ähnliche Chiffre. Er benutzte die Verschiebung um einen Buchstaben und verwendete für das X (damals letzter Buchstabe im Alphabet) ein AA als Substituenten.
Die konkrete Umsetzung in Python schauen wir uns etwas später an. Zuerst diskutieren wir einige Programmier-Varianten bei einem speziellen Fall der CÄSAR-Verschlüsselung.

8.19.1.x. ROT13

ROT13 ist eine spezielle Variante der CÄSAR-Verschlüsselung. Genaugesagt handelt es sich um eine CÄSAR13-Chiffre.

Beispiel: CÄSAR**13**

Symbole lfd. Nr.	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16	17	18	19	20	21	22	23	24	25	26
Klaralphabet	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Geheimalph.	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**

Durch die symmetrische Teilung des Alphabet's ergeben sich einige praktische Besonderheiten.

Es kommt zu einer festen Zuordnung der Buchstaben von Klar- und Geheimtext-Alphabet. Dadurch funktionieren Ver- und Ent-Schlüsselung mit dem exakt gleichen Algorithmus bzw. der gleichen Funktion.

A	B	C	D	E	F	G	H	I	J	K	L	M
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Ursprünglich wurde die ROT13 auch nicht wirklich als Verschlüsselung eingesetzt, sondern als Mittel der sehr einfach und effektiven Verschleierung von Texten. Ursprünglich sollte damit im usenet zweideutige Witze und Texte auf den ersten Blick versteckt werden.

Dabei zielte man auch auf den Effekt hin, dass ein Leser einen ROT13-Text bewußt entschlüsselt. Damit ist er auch für sich verantwortlich, wenn er mit bestimmten Obzönigkeiten, sexuellen Anspielungen usw. usf. nicht klar kommt.

Er hätte es lassen können.

ROT13-Verschlüsselungen sind, wie alle klassischen CÄSAR-Chiffren sehr leicht durch Brute-Force- Angriffe oder Häufigkeits-Analysen knackbar. Mittlerweise sind sie ein Sinnbild für sehr schlechte kryptographische Verfahren.

In den folgenden Programmier-Beispielen zu den verschiedenen Chiffren wollen wir ein grundlegendes Schema benutzen.

1. Eingabe des Klartextes
2. Umwandlung des Klartextes in die notwendige Form (Groß-Buchstaben, ev. ohne Leerzeichen)
3. Erstellen einer Häufigkeits-Analalyse (zu Vergleichszwecken)
4. Verschlüsseln der Klartextes
   a) ev. Anzeige von Zwischenschritten
   b) Anzeige des verschlüsselten Textes
5. Erstellen einer Häufigkeits-Analyse vom Geheimtext
6. Entschlüsseln des Geheimtextes

Entwickeln wir ein solches Programm nun schrittweise für die ROT13-Verschlüsselung.
Zuerst bauen wir uns ein einfaches Programm ohne Funktionen. Diese führen wir dann in einer zweiten Entwicklungs-Reihe ein.
Starten wir mit einem einfachen Eingabe-Ausgabe-Rahmen, in den wir dann als nächstes die Umwandlung der Eingabe in Groß-Buchstaben integrieren.
Die beiden Alphabete legen wir als feste Listen an. Gerade beim ROT13-Verfahren ist dies ja eine der Basis-Vereinbarungen.

```
# ROT13 Ver- und Ent-Schlüsselung
# L. Drews; 2020

# Definitionen
klarAlpha=["A","B","C","D","E","F","G","H","I",
           "J","K","L","M","N","O","P","Q","R",
           "S","T","U","V","W","X","Y","Z"]
geheimAlpha=["n","o","p","q","r","s","t","u","v",
             "w","x","y","z","a","b","c","d","e",
             "f","g","h","i","j","k","l","m"]
laengeKlarAlpha=26

# Eingabe
klarText=input("Klartext  : ")


# Verschlüsselung
geheimText=klarText

print("Geheimtext: ",geheimText)


# Entschlüsselung
dechiffKlarText=geheimText


# Ausgabe
print("Klartext  : ", dechiffKlarText)

input()
```

Mit der Funktion upper() erhalten wir einen Groß-Buchstaben-Text von einem Text-Objekt. (s.a. 8.1.1. Objekt-orientierte Nutzung von Strings)

```
…
# Eingabe
klarText=input("Klartext  : ")
klarText=klarText.upper()
print("KLARTEXT  : ",klarText)


# Verschlüsselung
…
```

In der Verschlüsselung selbst bestimmen wir zuerst die Länge des Klartextes. Desweiteren wird ein leerer Geheimtext angelegt, denn wird dann mit der i-Schleife Zeichen für Zeichen auffüllen wollen.
Bei jedem Schleifendurchlauf separieren wir ein KlarSymbol. Dieses wird mittel j-Schleife im Klar-Alphabet gesucht und sich die Position gemerkt.

```
…
# Verschlüsselung
print("=========> Verschlüsselung ==========>")
laengeKlarText=len(klarText)
geheimText=""
for i in range(laengeKlarText):
    klarSymbol=klarText[i]
    pos=-1
    for j in range(laengeKlarAlpha):
        if klarSymbol==klarAlpha[j]:
            pos=j
            break
    geheimSymbol=geheimAlpha[pos]
    if pos>=0:
        geheimText+=geheimSymbol
    else:
        geheimText+=klarSymbol
print("Geheimtext: ",geheimText)


# Entschlüsselung
…
```

Mit Hilfe der Position holen wir aus das passende GeheimSymbol aus dem geheim-Alphabet und hängen es an den bisher bearbeiteten Geheimtext an. Für den Fall, dass wir kein passende Symbol gefunden haben (pos ist dann immer noch -1), übernehmen wir das Nicht-Alphabet-Symbol.

Für die Entschlüsselung benutzen wir genau den gleichen Algorithmus, nur umgetauschten klar- und geheim-Variablen. Hier macht sich eine sprechende Benennung wieder einmal bezahlt.

```
# Entschlüsselung
print("=========> Entschlüsselung ========>")
laengeGeheimText=len(geheimText)
dechiffKlarText=""
for i in range(laengeGeheimText):
    geheimSymbol=geheimText[i]
    pos=-1
    for j in range(laengeKlarAlpha):
        if geheimSymbol==geheimAlpha[j]:
            pos=j
            break
    klarSymbol=klarAlpha[pos]
    if pos>=0:
        dechiffKlarText+=klarSymbol
    else:
        dechiffKlarText+=geheimSymbol

# Ausgabe
```

Aufgaben:

1. ***Übernehmen Sie das Programm und die ergänzenden Programm-Abschnitte! Testen Sie das Programm mit verschieden Eingaben!***
2. ***Erweitern Sie das Programm um die Möglichkeit weitere Klartexte einzugeben! Ein Abbruch soll mit der Eingabe eines leeren Textes erfolgen!***
3. ***Verändern Sie die Ausgabe "`======> Verschlüsselung …`" so, dass für jedes chiffrierte Symbol ein Gleichheitszeichen angezeigt wird! Übernehmen Sie dieses dann auch für die Entschlüsselung"***

Natürlich gibt es weitaus schönere und effektivere Daten-Strukturen für die Alphabete.

So kann man zwei einhache Strings benutzen und dann durch sie durch interieren.

```
klarAlpha="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
geheimAlpha="nopqrstuvwxyzabcdefghijklm"
…
```

Diese Variante erscheint mir z.B. sehr gut für die flexible Erzeugung von Klar- und Geheim-Text-Symbolen zu sein.

Eine weitere Möglichkeit ist die Verwendung von einer strukturierten / geschachtelten Liste aus Symbol-Paaren.

```
rot13=[["A","n"], ["B","o"], ["C","p"], …
       …
      ]
…
```

Auch Tupel in Form von Dictonary's sind denkbar. Besonders wenn man Alphabete aus einer (JSON-)Datei einlesen möchte, spricht einiges für diese Variante.

```
rot13=[["A","n"], ["B","o"], ["C","p"], …
       …
      ]
…
```

Es geht aber auch ohne Vordefinition der Alphabete. Man kann ja auch die ASCII-Tabelle der Rechner selbst nutzen.
Natürlich muss man dann die neuen ASCII-Symbole immer berechnen. Der Algorithmus ändert sich also entscheident.

Klar-Alphabet			
Symbol	**ASCII**	**Symbol**	**ASCII**
A	065	a	097
B	066	b	098
C	067	c	099
…		…	…
L	076	l	108
M	077	m	109
N	078	n	110
O	079	o	111
…		…	
X	088	x	120
Y	089	y	121
Z	090	z	122

Geheim-Alph..	
Symbol	**ASCII**
n	110
o	111
p	112
…	
y	121
z	122
a	097
b	098
…	…
k	107
l	108
m	109

Aufgaben:

1. *Entscheiden Sie sich für eine neue Art der Alphabet-Darstellung bzw. die "Alphabet"-frei Version und erstellen Sie ein neues ROT13-Programm!*
2. *Drucken Sie Ihr Programm aus und veröffentlichen Sie es für eine Diskussion an der Tafel oder einem schwarzen Brett od.ä.!*
3. *Ein Mitschüler vertritt die Auffassung, man durch mehrfache Anwendung von Verschlüsselungen aufeinander die Sicherheit deutlich erhöhen kann. Auch beim ROT13-verfahren soll dies so sein. Setzen Sie sich mit dieser These auseinander!*

für die gehobene Anspruchsebene:

4. *Verändern Sie eine Programm-Version so, dass eine Text-Datei mit einem längeren Klartext eingelesen und angezeigt werden kann! Dieser Text soll dann verschlüsselt und danach wieder entschlüsselt angezeigt werden!*

In der Registry sollen in bestimmten Schlüsseln die Verläufe (des Internet-Browsing) gespeichert sein, die mit ROT13 verschlüsselt sind und wohl auch nicht gelöscht werden, wenn man den Verlauf löscht!

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\ [Unterverzeichnisse]
```

Dieses Logging lässt sich abschalten, wenn man in dem Verzeichnis einen neuen Schlüssel (DWORD) "NoLog" anlegt und diesem den Wert auf 1 setzt. Mit einem weiteren Schlüssel "NoEncrypt" mit dem Wert 1 kann die "Verschlüsselung" ausgeschaltet werden (mit 0 eben wieder einegschaltet)

8.19.1.x.1. ROT13 mit einer Funktion

Die zwei praktisch identischen Algorithmen für die Ver- und Entschlüsselung sind natürlich ein Dorn im Auge eine ("faulen") Programmierer's. Findet man irgendwann einen Fehler oder will man den Algorithmus berändern, dann muss man immer an zwei Stellen im Programm korrigieren. Erfahrungs.gemäß geht das schief. Irgend eine Stelle vergißt man oder ändert diese anders. Da sind dann Folge-Probleme schon vorprogrammiert.

So etwas schreit ja förmlich nach der Benutzung einer Funktion, die den einen Text in den anderen umwandelt. Was dabei Klar- und was Geheim-Text ist, ist ja egal, weil das Verfahren so schön symmetisch ist.

Wir bleiben hier mal bei der oben besprochenen Form der Alphabet-Darstellung in zwei Listen. Die Variabalen, die sich auf den Klartext bezogen, werden in der Funktion nun in **rein**-gehende Variablen umbenannt. Dementsprechend die geheim-Variablen auf **raus**.

```
def rot13(textRein):
    laengeTextRein=len(textRein)
    textRaus=""
    for i in range(laengeTextRein):
        reinSymbol=textRein[i]
        pos=-1
        for j in range(laengeKlarAlpha):
            if reinSymbol==klarAlpha[j]:
                pos=j
                break
        geheimSymbol=geheimAlpha[pos]
        if pos>=0:
            textRaus+=geheimSymbol
        else:
            textRaus+=reinSymbol
    return textRaus
```

Nun müssen wir aber auch noch beachten, dass wir aus kosmetischen Gründen die Klar- und geheim-Texte mit anderen Buchstaben versehen haben.

Das Hauptprogramm verkürzt sich nun natürlich deutlich durch die Funktions-Aufrufe.

Das Programm wird so auch deutlich übersichtlicher und verständlicher.

Änderungen und Erweiterungen können wir nun auch sehr gut vornehmen.

```
…
abbruch=False
while not abbruch:
# Eingabe
    klarText=input("Klartext  : ")

    if klarText>"":
        klarText=klarText.upper()
        print("KLARTEXT  : ",klarText)

    # Verschlüsselung
        print("======> Verschlüsselung =====>")
        geheimText=rot13(klarText)
        print("ROT13-Fkt.: ",geheimText)

        geheimText=geheimText.upper()

    # Entschlüsselung
        print("======> Entschlüsselung =====>")

    # Ausgabe
        print("ROT13-Fkt.: ",
              rot13(geheimText))
        print("-----------------------------")
        print()
    else:
        abbruch=True

print("Programm-Ende")
```

Aufgaben:

1. ***Der Aufbau der Verschlüsselungs-Zeile mit den Gleichheits-Zeichen entsprechend der umgewandelten Symbole hat einem Kunden sehr gut gefallen. Bekommen Sie das auch mit rot13-Funktion hin? Realisieren Sie die Funktion entsprechend ODER begründen Sie, warum das so nicht geht!***
2. ***Wandeln Sie Ihr 2. ROT13-Programm (mit der geänderten Daten-Struktur für die Alphabete bzw. mit dem geänderten Algorithmus) in ein Programm mit einer passenden rot13-Funktion um!***

8.19.1.x.2. Häufigkeits-Analyse

Alle einfachen CÄSAR-Chiffren – und ganz besonders die ROT13-Chiffre – sind für Analysen der Buchstaben-Häufigkeit empfindlich. Wir wollen die Buchstaben-Häufigkeit vor allem dazu benutzen, um verschiedene Verfahren miteinander zu vergleichen und zu bewerten.
Hier werden wir eine Funktion erstellen, die sich auf die Zählung und Anzeige der Symbole beschränkt. Für vergleichende Zwecke müsste man sonst vielleicht auch die Ergebnisse wieder zurückgeben.

```
…
def symbolHaeufigkeit(alphabet, analyseText):
    print(".. Häufigkeits-Analyse ..")
    anzahlSymbole=len(alphabet)
    haeufigkeit=[]
    for i in range(anzahlSymbole):
        haeufigkeit.append(0)
    laengeText=len(analyseText)
    for i in range(anzahlSymbole):
        aktSymbol=alphabet[i]
        for j in range(laengeText):
            if analyseText[j]==aktSymbol:
               haeufigkeit[i]+=1
    for i in range(anzahlSymbole):
        print("   ",format(alphabet[i],"2s"),
              format(haeufigkeit[i],"3d"),
              fomat(haeufigkeit[i]/laengeText*100,"6.2f"),"%")
    #return
…
```

In der obigen Funktion wird Alphabet-bezogen gearbeitet. Das Ergebnis soll in der Liste hauefigkeit gespeichert werden. Für jedes Symbol aus dem Alphabet wird zuerst einmal eine Null als Anfangs-Wert eingespeichert.
Danach wird wieder für jedes Symbol der Text nach allen Vorkommen durchsucht und die Häufigkeit inkrementiert.
Zum Schluß wird die Häufigkeit für jedes Symbol mit Anzahl und prozentualem Anteil ausgegeben.

Aufgaben:

1. ***Vereinfachen Sie die Häufigkeits-Analyse dahingehend, dass nur noch eine Schleife (`for i in range(anzahlSymbole):`) benutzt wird!***
2. ***Erweitern Sie die Analyse um die Erfassung der Nicht-Alphabet-Symbole und einer nahtloschen Ausgabe als "???"!***
3. ***Da die Zeilen für die Symbole nicht wirklich ausgenutzt werden, möchte der Kunde eine Ausgabe in mehreren Spalten, wobei die Spalten-Anzahl variabel gehalten werden soll!***

Hier eine mögliche Umsetzung einzelner Aspekte in einer erweiterten Funktion.

```
def symbolHaeufigkeit(alphabet, analyseText):
    print(".. Häufigkeits-Analyse ..")
    anzahlSymbole=len(alphabet)
    haeufigkeit=[]
    anzahlGefunden=0
    laengeText=len(analyseText)
    for i in range(anzahlSymbole):
        haeufigkeit.append(0)
        aktSymbol=alphabet[i]
        for j in range(laengeText):
            if analyseText[j]==aktSymbol:
               haeufigkeit[i]+=1
               anzahlGefunden+=1
    spalten=3
    aktSpalte=0
    for i in range(anzahlSymbole):
        print((8-spalten)*" ",format(alphabet[i],"2s"),
              format(haeufigkeit[i],"3d"),
              format(haeufigkeit[i]/laengeText*100,"6.2f"),"%",end="")
        aktSpalte+=1
        if aktSpalte==spalten:
            aktSpalte=0
            print()
    print((7-spalten)*" ",format("???","3s"),
          format(laengeText-anzahlGefunden,"3d"),
          format((laengeText-anzahlGefunden)/laengeText*100,"6.2f"),"%")
    #return
```

Aufgaben:

1. ***Erweitern Sie die Buchstaben-Häufigkeits-Analyse um kleine Histogramme (in Balken-Form)! Für jeweils gerundete 10 % könnte z.B. eine Raute (#) und für 5 % ein senkrechter Strich (\|) gesetzt werden.***
2. ***Verändern Sie die Funktion so, dass beim Funktions-Aufruf auch mit angegeben werden kann, in wievielen Spalten die Ausgabe erfolgen soll!***
3.

8.19.1.x. Umsetzung der CÄSAR-Verschlüsselung

Mit den Kenntnissen aus der Umsetzung der ROT13-Verschlüsselung können wir nun auch effektiv eine universelle CÄSAR-Ver- und Entschlüsselung programmieren.
Will man das Konzept der zwei Alphabete, wie vorne beschrieben, weiter benutzen. Dann muss man sich nach der Festlegung der Verschiebung das Geheim-Alphabet (neu) zusammenstellen. Dann kann man viele Funktionen mit wenigen Veränderungen übernehmen. Dafür spricht z.B., dass man diese Funktionen z.B. für CÄSAR13 ja schon getestet hat. Wie wir später bei der Umsetzung einer CÄSAR-Verschlüsselung mit einem Schlüssel noch sehen werden, ist diese zuerst etwas altbacken wirkende Methode, dann doch wieder sehr praktisch.
Aus meiner Sicht spricht hier aber mehr für eine flexible Umsetzung auf der Basis der ASCII-Zeichen.
Geht man von einem Ring-förmigen Alphabet aus, dann beschreiben die Funktionen:

$chiffriert_S(K) = (K + S) \bmod 26 = G$	K .. Position Klar-Alphabet-Symbol S .. Schlüssel-(Nummer) / Verschiebung
$dechiffriert_S(G) = (G - S) \bmod 26 = K$	G .. Position Geheim-Alphabet-Symbol

das grundsätzliche Vorgehen.
Bei den ASCII-Zeichen haben wir aber kein geschlossenes Alphabet. Leider sind auch die Kleinbuchstaben nicht direkt an die Groß-Buchstaben angeschlossen, was uns hier super helfen würde.
So bleibt nur eine Entscheidungs-bezogene Umsetzung entsprechend der Lage des / der Buchstaben zum Geheim-Buchstaben, welcher der CÄSAR-Chiffre entspricht.

```
# CÄSAR-Verschlüsselung
# 01.2020

# Eingabe Klartext
klarText=input("Klartext:  ")
verschiebung=eval(input("CÄSAR-Code (-26 .. 26): "))

# Vorbereitung
klarText=klarText.upper()
print("KLARTEXT:  ",klarText)
obenGeheimCodeGrenze=90-verschiebung
geheimText=""

# Verschlüsselung
laengeKlarText=len(klarText)
for i in range(laengeKlarText):
    aktKlarSymbol=ord(klarText[i])   #ASCII-Code
    if aktKlarSymbol>=65 and aktKlarSymbol<=90: # Großbuchstabe
        aktGeheimSymbol=aktKlarSymbol+verschiebung
        if aktGeheimSymbol>90:  # zu großer ASCII-Code
            aktGeheimSymbol-=26
        if aktGeheimSymbol<65:  # zu kleiner ASCII-Code
            aktGeheimSymbol+=26
        geheimText+=chr(aktGeheimSymbol) # Symbol aus Code
    else: # Nicht-Alphabet-Symbol
        geheimText+=chr(aktKlarSymbol)

print("Geheimtext: ",geheimText.lower())
```

Nach der Prüfung, ob es sich um ein Klartext-Symbol (also Großbuchstabe) handelt, wird die Verschiebung im ASCII-Code vorgenommen. Dabei können auch ASCII-Code's jenseits der Großbuchstaben entstehen. Hier korrigienen wir dann um die Alpahbet-Länge (hier 26).

Der aufmerksame Leser wird gleich bemerkt haben, dass ich dieses Mal nicht in die Kleinbuchstaben chiffriert habe. Da steckt einfach schon der Hintergedanke drin, später eine universelle Funktion aus dem Prototypen zu machen. Die Darstellung mit Klein- und Groß-Buchstaben ist ja nur Kosmetik und die möchte ich den Python-String-Funktionen überlassen.

Aufgaben:

1. ***Ergänzen Sie das obige Programm um eine Wiederholungs-Schleife, die solange chiffriert, bis ein Leertext eingegeben wird!***
2. ***Verändern Sie das Programm so, dass eine direkte Chiffrierung in die Kleinbuchstaben erfolgt!***
3. ***Planen Sie eine universelle Funktion für die CÄSAR-Chiffrierung und –Dechiffrierung! Welche Parameter braucht eine solche Funktion?***
4. ***Realisieren Sie eine universelle CÄSAR-Funktion!***
5. ***Erweitern Sie das Programm nun noch um die Dechiffrierung und eine Buchstaben-Häufigkeits-Analyse!***

8.19.1.x. moderne CÄSAR-Verschlüsselung mit Schlüssel

Natürlich ist der Begriff modern nicht wirklich Ernst gemeint. Durch das große Leistungs-Potential von Computern und vielen sehr cleveren Analyse-Methoden ist die CÄSAR-Chiffrierung nicht mehr Stand der Zeit. Aber mit ein paar Tricks kann man noch so einiges aus dieser "einfachen" Verschlüsselung herausholen.

Beispiel: CÄSAR7 mit Schlüsselwort BUCHLISTE

Symbole lfd. Nr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Klaralphabet	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Geheimalph.	**r**	**v**	**w**	**x**	**y**	**z**	**b**	**u**	**c**	**h**	**l**	**i**	**s**	**t**	**e**	**a**	**d**	**f**	**g**	**j**	**k**	**m**	**n**	**o**	**p**	**q**

Beispiel: CÄSAR7 auf ein Zufalls-Alphabet

Symbole lfd. Nr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Klaralphabet	**K**	**T**	**G**	**E**	**P**	**S**	**W**	**Y**	**J**	**Q**	**H**	**Z**	**I**	**A**	**V**	**O**	**R**	**C**	**M**	**N**	**U**	**X**	**B**	**D**	**F**	**L**
Geheimalph.	**u**	**x**	**b**	**d**	**f**	**l**	**k**	**t**	**g**	**e**	**p**	**s**	**w**	**y**	**j**	**q**	**h**	**z**	**i**	**a**	**v**	**o**	**r**	**c**	**m**	**n**

8.19.1.x. POLYBIOS-Verschlüsselung

auch unter Polybius zu finden
beschrieben vom griechischen Geschichtsschreiber POLYBIOS VON MEGALOPOLIS etwa um 200 bis 120 v.u.Z.

A	B	C	D	E	1
F	G	H	I	K	2
L	M	N	O	P	3
Q	R	S	T	U	4
V	W	X	Y	Z	5
1	2	3	4	5	

Basis ist die sogenannte Polybios-Matrix (Polybios-Quadrat), in der die Klar-Buchstaben Zeilen-weise notiert sind
Die Zeilen und Spalten werden durchnummeriert.
Die Zeilen- und Spalten-Nummern (Buchstaben-Koordinaten) werden zur Substitution genutzt.
Nehmen wir an, unser Klartext lautet:

SEHR GEHEIM

A	B	C	D	E	1
F	G	H	I	K	2
L	M	N	O	P	3
Q	R	**S**	~~T~~	~~U~~	**4**
V	W	~~X~~	Y	Z	5
1	2	**3**	4	5	

Praktisch wird nun jeder zu verschlüsselnde Buchstabe in der Matrix gesucht und dann zuerst immer die Zeilen- und dann die Spalten-Nummer notiert.
Aus dem **S** wird so **43** usw. usf.
Die Nummer konnten dann z.B. über auf dem Burg-Türmen oder –Mauern angeordneten Fackeln signalisiert werden.
Die Empfänger benutzten die gleiche Tabelle und praktisch das gleiche Verfahren (nur umgekehrt), um den Klartext aus den Geheimzeichen zu dechiffrieren.
Aus den Koordinaten **43** erhalten wir dann den entschlüsselten Buchstaben **S**.

A	B	C	D	E	1
F	G	H	I	K	2
L	M	N	O	P	3
Q	R	**S**	~~T~~	~~U~~	**4**
V	W	~~X~~	Y	Z	5
1	2	**3**	4	5	

In Python können wir für die POLYBIOS-Matrix ein mehrdimensionales Feld (Array, Vektor) nutzen (→ 6.6. Vektoren, Felder und Tabellen).

```
polybios=array(["A","B","C","D","E"],
               ["F", …],
               …)
```

Aufgaben:

1. *Verschlüsseln Sie den obigen Text bis zum Ende!*
2. *Überlegen Sie sich, wie das Verfahren verbessert werden! Machen Sie Vorschläge und erklären Sie, welche Veränderungen sich ergeben würden!*
3. *Setzen Sie die POLYBIOS-Verschlüsselung in ein Python-Programm um! Der zu verschlüsselnde Text soll als Eingabe in das Programm einfließen! Als Basis-Quadrat verwenden wir eine 6x6-Matrix mit allen Buchstaben und den Ziffern.*

Ein der praktischen Umsetzungen erfolgte als Klopf-Code in Gefängnissen, um z.B. über Rohrleitungen oder Wände Nachrichten zu übertragen.
Beim bifid-Verfahren (→) wird an eine POLYBIOS-Verschlüsselung noch eine Transposition angehängt, um die Koordinaten zu trennen.
Eine noch weitere Verbesserung erfolgte durch das ADFGX-Verfahren (→). Dieses wurde noch bis in den 1. Weltkrieg hinein verwendet.

8.19.1.x. bifid-Verschlüsselung

Bei der bifid-Chiffrierung wird die klassische POLYBIOS-Chiffrierung (Substitution) durch eine Faktionierung und eine Rück-Chiffrierung ergänzt.
Dazu werden die Koordinaten nicht hintereinander weg geschrieben, sondern in zwei Zeilen:
Nehmen wir an, unser Klartext lautet:

A	B	C	D	E	1
F	G	H	I	K	2
L	M	N	O	P	3
Q	R	**S**	T	U	**4**
V	W	X	Y	Z	5
1	2	**3**	4	5	

SEHR GEHEIM

Praktisch wird nun jeder zu verschlüsselnde Buchstabe in der Matrix gesucht und dann zuerst immer die Zeilen- und dann die Spalten-Nummer notiert.
Aus dem **S** wird so:

4
3

usw. usf.
Die beiden Zeilen:

4124 212123
3532 253542

wird dann die Symbolfolge:

41242121233532253542

Mit dieser Zifferfolge wird nun eine Rück-Verschlüsselung vorgenommen. D.h. die ersten zwei Ziffern sind die Koordninaten

41 24 21 21 23 35 32 25 35 42

für den ersten Geheim-Buchstaben:

Daraus ergibt sich ein Geheimtext, der auch Häufigkeits-Analysen stand hält:

qiffhpmkpr

a	b	c	d	e	1
f	g	h	i	k	2
l	m	n	o	p	3
q	r	s	t	u	**4**
v	w	x	y	z	5
1	2	3	4	5	

Wie in symmetrischen verfahren üblich läßt sich die Entschlüsselung durch das Umkehren des Verfahrens erreichen.
Zuerst wandeln wir den Geheimtext wieder in die Koordinaten um:

41 24 21 21 23 35 32 25 35 42

Dann wird die reine Ziffernkette

41242121233532253542

in der Mitte getrennt, um de Fraktionierung rückgängig zu machen:

4124 212123

3532 253542

Die Koordinaten oben und unten in den zwei Zeilen sind nun wieder die Basis für die Rück-Verschlüsselung in den Klartext.
Aus **43** wird so wieder der ursprüngliche Klartext-Buchstabe **S**.
In dieser Form gehen wir die nächsten Paare durch und erhalten den vollständigen Klartext zurück:

A	B	C	D	E	1
F	G	H	I	K	2
L	M	N	O	P	3
Q	R	**S**	T	U	**4**
V	W	X	Y	Z	5
1	2	**3**	4	5	

SEHR GEHEIM

Bei der Umsetzung in Python sollten wir jetzt deutlich planvoller vorgehen.
Da wir nun für Chiffrieren und Dechiffrieren immer jeweils eine POLYBIOS-Chiffrierung und –Dechiffrierung brauchen, ist die Nutzung von Funktion fast nicht mehr zu umgehen.

```
function chiffPolybios(polybios, zeichen):
    …
    return koord

function dechiffPolybios(koord, polybios):
    …
    return zeichen
```

Aufgaben:

1. *Planen Sie ein Programm zur Chiffrierung und Dechiffrierung nach dem bifed-Verfahren als Grob-Struktogramm!*
2. *Leiten Sie aus dem Grob-Struktogramm ein Funktions-orientiertes Python-Programm ab, in dem Sie dann Schritt-weise die Funktionen mit Leben füllen!*
3.

8.19.1.x. ADFGX-Verschlüsselung

in einer erweiterten 6x6-Form auch ADFGVX
basiert auf der POLYBIOS-Chiffre

der offizielle Name war "Geheimschrift der Funker 1918" oder kurz "GedeFu 18"

Bst.	MORSE-Zeichen
A	·—
D	—··
F	··—·
G	——·
V	···—
X	—··—

der Name ADFGX stammt von den Allierten, die die auffälligen Funksprüche nach den verwendeten Buchstaben charakterisierten
damals natürlich immer manuell durchgeführt
die Auswahl der Buchstaben wurde so gewählt, dass die MORSE-Zeichen sich besonders gut voneinander unterscheiden ließen

entwickelt vom deutschen Nachrichten-Offizier Fritz NEBEL (1891 - 1977)

verwendet wird als Basis das POLYBIOS-Quadrat
wieder ohne das J
auch die Reihenfolge der Buchstaben wird gedreht

Z	Y	X	W	V	a
U	T	S	R	Q	d
P	O	N	M	L	f
K	I	H	G	F	g
E	D	C	B	A	x
a	d	f	g	x	

desweiteren wird ein Schlüsselwort verwendet
als Beispiel hier: **Verschlüsselung**

V	E	R	S	C	a
H	L	U	N	G	d
Z	Y	X	W	T	f
Q	P	O	M	K	g
I	F	D	B	A	x
a	d	f	g	x	

dieses sollte optimaler-weise schön lang sein und alle Buchstaben nur einfach enthalten
sollten Buchstaben doppelt vorkomm,en werden sie im Einsatz einfach weggelassen
damit bleibt **VERSCHLUNG** übrig
das restliche Alphabet wird dann dahinter geschrieben

als Nächstes erfolgt die POLYBIOS-typische Substitution durch Beschriftung der Zeilen und Spalten (Koordinaten in der Matrix)
Soll z.B. der Klartext:

SEHR GEHEIM

V	E	R	S	C	a
H	L	U	N	G	d
Z	Y	X	W	T	f
Q	P	O	M	K	g
I	F	D	B	A	x
a	d	f	g	x	

verschlüsselt werden, dann wird aus dem **S** der Zwischen-Code **ag**.
Nachdem der Klartext so codiert wurde, erhalten wir:

ag ad da af dx ad da ad xa gg

Um die Angreifbarkeit gegen Häufigkeits-Analyse zu verbessern wird nun noch eine zweite Stufe der Verschlüsselung genutzt.
Dazu wird ein zweites Schlüsselwort (hier: **Krypto**) verwendet. Dieses Mal wird üblicherweise auf das Weglassen doppelter Buchstaben verzichtet. Das Verfahren funktioniert aber auch mit dem Weglassen. Wir verwenden hier jetzt nur ein kurzes Wort, da ja auch unser Zwischtext relativ kurz ist. Übliche Schlüsselwortlängen sind hier 15 bis 22 Zeichen.

Den Schlüsselwort-Buchstaben wird nun ihrer Position im klassischen Alphabet entsprechent eine Reihenfolge zugeordnet. Da im Alphabet das K aus Krypto der erste Buchstabe ist, erhalt es die Spalten-Nummer 1 usw. usf.
In die neue Tabelle wird nun der Zwischen-Code Zeilenweise notiert.
Die fehlenden Zeichen werden ausgefüllt. Hier die Buchstaben des Geheim-Alphabetes in umgekehrter Reihenfolge.

K	R	Y	P	T	O
1	**4**	**6**	**3**	**5**	**2**
a	g	a	d	d	a
a	f	d	x	a	d
d	a	a	d	x	a
g	g	x	g	f	d

Das Erzeugen des zu sendenden Geheimtextes erfolgt nun durch Auslesen der Spalten entsprechend der (aus dem 2. Schlüsselwort) abgeleiteten Reihenfolge.
Also wird zuerst die Spalte K und dann O usw. usf. hintereinander notiert.

aadg adad dxdg gfag daxf adax

In typischer Funker-Manier werden die Buchstaben in Fünfer-Gruppen übertragen, was fehlende oder falsch erkannte Zeichen leichter erkennen läßt.

aadga daddx dggfa gdaxf adax

Auch hier können die fehlenden Buchstaben beliebig ergänzt werden. Z.B. kännten wir noch ein a ranhängen. Damit sendet der Funken dann 25 Zeichen:

aadga daddx dggfa gdaxf adaxa

Zur Dechiffrierung geht man den umgekehrten Weg durch das Verfahren.
Zuerst werden die Fünfer-Gruppen aufgelöst und die Zeilen-Anzahl für die Transpositions-Tabelle aus der Buchstaben-Anzahl und der Schlüsselwort-Länge berechnet. Bei 25 Zeichen Geheimtext und der Schlüsselwort-Länge von 6 Zeichen ergeben sich 4 Zeilen (6 x 4 = 24 < 25).
Somit wird aus dem ungruppierten Geheimtext jetzt einer, der 4er Gruppen enthält:

aadg adad dxdg gfag daxf adax a

Das Schlüsselwort muss jetzt natürlich bekannt sein, damit die richtige Spalten-Reihenfolge ermittelt werden kann.
Die
Dabei bleiben die überzähligen Buchstaben in der letzten Spalte hängen und werden einfach ignoriert.

K	R	Y	P	T	O
1	**4**	**6**	**3**	**5**	**2**
a	g	a	d	d	a
a	f	d	x	a	d
d	a	a	d	x	a
g	g	x	g	f	d
		a			

Im zweiten Schritt werden wieder die Koordinaten rekonstruiert. Dazu wird die Hilfs-Tabelle wieder Zeilen-weise in 2er Gruppen ausgelesen:

ag ad da af dx ad da ad xa gg xg fd

Zum Schluß rekonstruieren wir aus den Koordinaten wieder die ursprünglichen Buchstaben. Aus dem **ag** wird so wieder das **S** usw. usf.

V	E	R	S	C	a
H	L	U	N	G	d
Z	Y	X	W	T	f
Q	P	O	M	K	g
I	F	D	B	A	x
a	d	f	g	x	

gebrochen durch die Krypto-Analyse des fanzösischen Georges PAINVIN (1918)

Aufgaben:

1. *Verändern Sie die Verschlüsselung so, dass mit einem 6x6-Quadrat für alle Buchstaben und Ziffern gearbeitet werden kann! Probieren Sie es einmal mit einem Partner un gegenseitiger Nachrichten-Übertragung aus!*
2. *Entwickeln Sie ein Python-Programm, welches das ADFGVX-Verfahren umsetzt! Der Klartext und die beiden Schlüsselwörter sollen eingebbar sein! Alle Zwischen-Schritte bzw. –Tabellen sollen anzeigbar sein. Im fertigen Programm sollten diese dann abschaltbar sein oder auskommentiert werden!*
3.

8.19.1.x. trifid-Verschlüsselung

von Franzosen Felix DELASTELLE (1840 - 1902) entwickelt
1902 beschrieben

arbeitet mit drei kleineren Polybios-ähnlichen Tabellen
dadurch ergeben sich für jeden Buchstaben 3 Koordinaten (Matrix, Zeile, Spalte)
diese Trigramme
Das Klartext-Alphabet kann hier schon mit einem Schlüsselwort verteilt eingetragen werden. Der Übersichtlichkeit verwe hier ein unverschlüsseltes Alphabet.

Matrix **1**			
	1	**2**	**3**
1	A	B	C
2	D	E	F
3	G	H	I

Matrix **2**			
	1	**2**	**3**
1	J	K	L
2	M	N	O
3	P	Q	R

Matrix **3**			
	1	**2**	**3**
1	S	T	U
2	V	W	X
3	Y	Z	+

Durch die drei 3x3 Tabellen kommen wir nun auch auf 27 mögliche Symbole. Da verwenden wir das vollständige Alphabet und ein Sonderzeichen, was z.B. als Leerzeichen dienen könnte.
Das Vorgehen ist wieder äquivalent zur POLYBIOS-Chiffre. Nur erhalten wir außer den beiden Koordinaten auch noch die Matrix-Nummer.
Verschlüsseln wir dieses Mal:

RICHTIG GEHEIM

Nun wird das **R** durch das Trigramm **233** codiert. Dieses wird wieder gleich auf drei Zeilen verteilt:

Matrix **1**			
	1	**2**	**3**
1	A	B	C
2	D	E	F
3	G	H	I

Matrix **2**			
	1	**2**	**3**
1	J	K	L
2	M	N	O
3	P	Q	R

Matrix **3**			
	1	**2**	**3**
1	S	T	U
2	V	W	X
3	Y	Z	+

2
3
3

Mit den anderen Symbolen gehen wir genauso vor. Das Leerzeichen ersetzen wir durch ein **+**. Insgesamt ergibt sich dann:

21113131111212
33131333232132
33322131222331

Als nächstes sollen Blöcke erstellt werden. Üblich sind 5 bis 7 Spalten als ein Block. Wir wählen hier die 5 als Block-Größe. Damit ergibt die folgende Struktur:

21113 13111 1212
33131 33323 2132
33322 13122 2331

Der letzte Block wird durch ein beliebiges Zeichen erweitert. Ich wähle hier das – eigentlich ungüstige – Leerzeichen.

21113 13111 12123
33131 33323 21323
33322 13122 23313

Der nächste Schritt ist spezifisch für das trifed-Verfahren. Die Blöcke werden – jeder für sich – Zeilen-weise in neue Trigramme zerlegt. Im ersten Block habe ich das durch unterschiedliche Farben gekennzeichnet:

21113 13111 12123
33131 33323 21323
33322 13122 23313

Diese neuen Trigramme verteilen wir wieder auf 3 Zeilen:

21313 11331 12133
13132 31312 23321
13332 13232 12233

und verschlüsseln mit den ursprünglichen Tabellen zurück:

jiuiw gczue dqhxu

Matrix 1			
	1	**2**	**3**
1	a	b	c
2	d	e	f
3	g	h	i

Matrix 2			
	1	**2**	**3**
1	j	k	l
2	m	n	o
3	p	q	r

Matrix 3			
	1	**2**	**3**
1	s	t	u
2	v	w	x
3	y	z	+

Dieser Geheimtext hält einer Häufigkeits-Analyse gut stand.

Die Dechiffrierung dreht das Verfahren einfach um.
Zuerst ermitteln wir die Koordinaten der Geheimtext-Buchstaben, sortieren sie innerhalb von Blöcken wieder um und verschlüsseln zurück.

Matrix 1			
	1	**2**	**3**
1	a	b	c
2	d	e	f
3	g	h	i

Matrix 2			
	1	**2**	**3**
1	j	k	l
2	m	n	o
3	p	q	r

Matrix 3			
	1	**2**	**3**
1	s	t	u
2	v	w	x
3	y	z	+

21313 11331 12133
13132 31312 23321
13332 13232 12233

→

21113 13111 12123
33131 33323 21323
33322 13122 23313

→

21113 13111 12123
33131 33323 21323
33322 13122 23313

→

211131311112123
331313332321323
333221312223313

Am Schluß folgt nun die Rück-Verschlüsselung mittels der Matrizen zum Klartext:
Aus dem Trigramm **233** wird so wieder der Klartext-Buchstabe **R**.
Der so rekonstruierte Klartext:

Matrix **1**			
	1	**2**	**3**
1	A	B	C
2	D	E	F
3	G	H	I

Matrix **2**			
	1	**2**	**3**
1	J	K	L
2	M	N	O
3	P	Q	R

Matrix **3**			
	1	**2**	**3**
1	S	T	U
2	V	W	X
3	Y	Z	+

RICHTIG GEHEIM+

unterscheidet sich nur durch zusätzlich angefügte Plus-/Leer-Zeichen.

Für ein Python-Programm könnte man sicher wieder mehrdimensionale Felder benutzen (→ 6.6. Vektoren, Felder und Tabellen).
Hier scheint mir eine Chiffren-Tabelle – bzw. zwei – auf der Basis von Dictonary's vielleicht günstiger.

```
trifedChiff={
    "A": 111,
    "B": 112,
    …
    }

trifedDechiff={
    111: "A",
    112: "B",
    …
    }
```

alternativ mit Tupeln der drei Koordinaten

```
trifedChiff={
    "A": [1,1,1],
```

Eine strukturierte Liste wäre ebenfalls möglich. Hier spart man sich dann auch die doppelte Darstellung in zwei Dictonary's.
Hier sind mehrere Varianten denkbar.

```
trifed={
    {"A",[1,1,1]},
    {"B",[1,1,2]},
    …
    }
```

```
trifed={
    {"A",111},
    {"B",112},
    …
    }
```

```
trifed={
    {"A",1,1,1},
    {"B",1,1,2},
    …
    }
```

Aufgaben:

1. ***Vereinbaren Sie im Kurs ein reichlich langes Schlüsselwort für die Dictory-Struktur!***
2. ***Erstellen Sie nun ein Programm, dass die trifed-Ver- und Entschlüsselung zeigt! Als Block-Größe verwenden wir die 3, um auch Zahlen-Operationen für die Transposition zu ermöglich!***
3.

für das gehobene Anspruchsniveau:

4. ***Setzen Sie das trifed-Verfahren ohne Einschränkungen um! Es sollen sowohl das Schlüsselwort für die drei Matrizen sowie die Block-Größe frei gewählt werden können!***

(c,p) 2015 - 2020 lsp: dre

8.19.1.x. Four-Square-Verschlüsselung

von Franzosen Felix DELASTELLE (1840 - 1902) entwickelt
benutzt als Basis-Alphabet Buchstaben-Paare (Digraphen, Bigramme)
Substitutions-Chiffre
damit werden aus den 26 Monographen (Monogramme) 676 Digraphen, die nun deutlich schwierger durch Häufigkeits-Analysen angreifbar sind
hierfür wären auch sehr lange Texte notwendig

Zur Ver- und Entschlüsselung werden 4 Quadrate (daher der Name) verwendet. Jeweils diagonal sind die Klartext- sowie die Geheimtext-Alphabete notiert. Im Beispiel wird, wie bei DELASTELLE auf das Q verzichtet. Alternativ wäre ein Verzicht auf das J möglich.
Modernere Verfahren nutzen 6x6-Felder. Dann passen auch noch die Ziffern mit hinein.
Zur Erhöhung der Verschlüsselung werden zwei Schlüsselwörter an den Anfang der Geheimtext-Alphabete gesetzt.

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	R	S	T	U
V	W	X	Y	Z

v	e	r	s	c
h	u	l	n	g
a	b	d	f	i
j	k	m	o	p
t	w	x	y	z

f	o	u	r	s
a	e	b	c	d
g	h	i	j	k
l	m	n	p	t
v	w	x	y	z

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	R	S	T	U
V	W	X	Y	Z

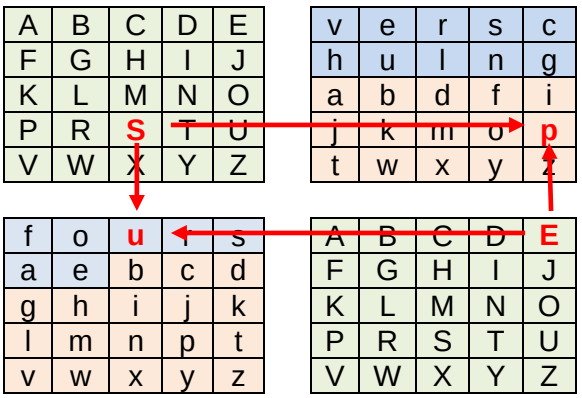
Doppelte Buchstaben werden weggelassen. Hinter den Schlüsselwörtern folgt das restliche Alphabet.
Der zu verschlüsselnde Klartext

z.B.: SEHR GEHEIM

wird zuerst in Digraphen zerlegt:

SE HR GE HE IM

Nun wird der erste Buchstabe im oberen Klartext-Quadrat gesucht und der zweite aus dem unteren. Nun werden Waagerechten und Senkrechten in die Geheimtext-Quadrate gezogen und dort die Geheimtext-Zeichen abgelesen.
Aus **SE** wird so **pu**.
Das Verfahren wird nun Digraph für Diagraph fortgesetzt.

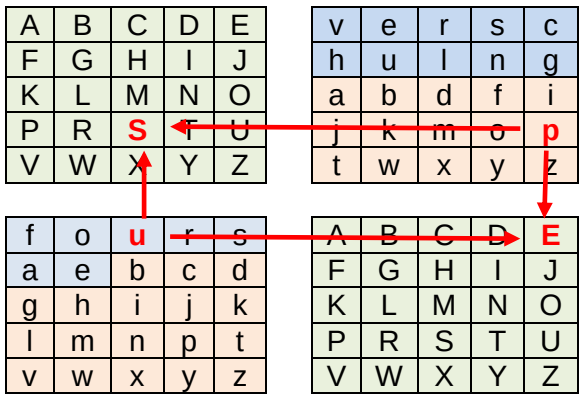


Aufgaben:

1. *Verschlüsseln Sie den Resttext!*
2. *Vereinbaren Sie mt einem Partner aus dem Kurs ein Schlüsselwort-Paar und verschlüsseln Sie damit eine kurze Nachricht!*
3.

Da es sich um ein symmetrisches Verfahren handelt verwenden wir die gleichen Schlüsselwörter und das inverse Verfahren für die Dechiffrierung.
Der empfangene Geheimtext wird wieder in Digraphen zerlegt:

z.B.: **pu** …



und dann der erste Buchstabe im oberen und der zweite im untereb Geheim-Alphabet gesucht. Die Klartext-Buchstaben ergeben sich wieder über die Waagerechten und Senkrechten.
So bekommen wir **SE** aus dem Klartext zurück.

In Python können wir hier mehrdimensionale Felder (Array's, Vektoren) nutzen (→ 6.6. Vektoren, Felder und Tabellen).
Die Geheimtext-Felder können natürlich erst nach Eingabe der Schlüsselwörter belegt werden.

```
klaroben=array(["A","B","C","D","E"],
               ["F", …],
               …)
geheimoben=array(["v","e","r","s","c"],
                 ["", …],
                 …)
…
```

Aufgaben:

1. *Tauschen Sie die verschlüsselten Nachrichten (vom ersten Aufgabenblock) und entschlüsseln Sie diese!*
2. *Schreiben Sie ein Programm, das einen einzgebenen Text ver- bzw. entschlüsselt! Die Schlüsselwörter sollen ebenfalls jeweils einzugeben sein! Als Alphabet benutzen wir alle Buchstaben und die Ziffern.*
3.

Aber auch Lösungen über verkettete Liste oder Tupel sind denkbar.

mögliche Listen

```
klaroben={ {"A","B","C","D","E"},
           {"F", …},
            …
         }
geheimoben={ {"v","e","r","s","c"},
             {"", …},
              …
           }
…
```

eine andere Listen-Variante mit weniger Such-Aufwand könnte so aufgebaut sein

```
klaroben={ {"A",0,0}, {"B",0,1}, …

geheimoben= { {"v",0,0}, {"e",0,1}, …
```

8.19.2. asymmetrische Verschlüsselung

interessante Links:

http://inventwithpython.com/hackingciphers.pdf (online-Version des Buches: AL SWEIGART: Hacking Secret Ciphers with Python)

8.20. Code verbessern und optimieren

Laufzeit wird durch viele Faktoren beeinflusst

Leistungs-Parameter des Computers / der ausführenden Maschine

Datenmenge
im Allgemeinen steigt der Berechnings-Aufwand nicht linear, meist expotentiell, selten überexportentiell

Lade-Technik
Laden und Arbeiten mit Daten im Speicher schneller als auf Fetsplatte
lokales Netzwerk ist noch langsamer
am langsamsten ist das Internet

Programmier-Stil / -Paradigmen
unnötige / ungünstige Befehle

Algorithmen / Algorithmen-Analyse
ungünstige Lage von nicht gebrauchten Funktions-Aufrufen inSchleifen
stattdessen speichern des Wertes in einer Variable und dann Nutzung der Variable in der Schleife

8.21. Test-gestütztes Programmieren mit Python

```
1  def test(algorithmus, testBedingung = True):
2
3      assert algorithmus([]) == []
4      assert algorithmus([2]) == [2]
5      assert algorithmus([4,2,2]) == [2,2,4]
6      assert algorithmus([1,2,4,6,7,9]) == [1,2,4,6,7,9]
7      assert algorithmus([9,8,5,5,3,2,1]) == [1,2,3,5,5,8,9]
8      if testBedingung:
9          assert algorithmus([0,-3,5,-9,13] == [-9,-3,0,5,13]
10
11 if __name__ == "__main__":
12     test(bubblesort, False)
13     test(bubblesort)
       test(quicksort)
```

9

9. Python, informatisch – Datenstrukturen, Klassen, Automaten, …

Ähnlich wie Mathematiker leben Informatiker in einer eigenen Modell-Welt. Nicht umsonst gelten sie als Nerds oder Guru's oder was es sonst auch noch für (böse) Bezeichnungen für die liebsten Menschen der Welt gibt (;-).
Beim genauen Hinschauen sind die Modell-Objekte genau so clever, wie die unzähligen mathematischen Operationen und Techniken. Wenn man irgendwelche Dinge am Computer tut, dann werden uns viele benutzte Informatik-Modelle gar nicht so recht bewusst. Kaum einer weiss, dass die Druck-Aufträge in einer Warteschlange verwaltet werden. Der Druck-Auftrag, der zuerst kommt, wird auch zuerst ausgedruckt. Erst wenn einer der Aufträge den Drucker blockiert, dann werden wir uns vielleicht die Warteschlange ansehen (Doppelklicken auf das Drucker-Symbol in der Task-Leiste) und den störenden Auftrag dort löschen.
Mit der Datenstruktur Baum haben wir dagegen alle schon zu tun gehabt, zumindestens, wenn wir einen Computer mehr als einmal praktisch genutzt und Daten gespeichert haben. Die Ordner in einem Laufwerk sind genau so eine Baum-Struktur. Aber auch die Laufwerke selbst sind wieder eine Baum-Struktur. Sie haben die gemeinsame Wurzel "Computer".
Keller und Ringe sind wieder eher verborgene Objekte. Aber auch sie werden für das ordnungsgemäße bzw. gewohnte Funktionieren eines PC gebraucht.

Definition(en): Datenstruktur
Eine Datenstruktur ist der Informatik eine Vereinbarung zur Organisation und Speicherung von Daten.

Definition(en): Datenstruktur
Eine Datenstruktur ist der Informatik eine Vereinbarung zur Organisation und Speicherung von Daten.

Einteilung nach (maximalen) Anzahl der Nachfolger auf ein Objekt möglich

maximal ein Nachfolger:
Liste

maximal zwei Nachfolger:
Binär-Baum

beliebig viele Nachfolger:
(allgemeiner) Baum
Netz

Zuerst werden wir allerdings etwas genauer auf die sogenannten Tupel eingehen. Sie sind keine klassische Datenstruktur oder gar ein Informatiker-Modell. Sie sind eher eine Spezialität von Python.

9.1. Tupel

Tupel sind Aufzählungen von Daten(-Objekten)
können, müssen aber nicht, den gleichen Typ haben

man kann sie als Paare oder Gruppen verstehen
beim Lesen des Quelltextes erscheinen viele Anweisungen kryptisch oder falsch
Anflug von Trick-Programmierung; meist aber elegante und effektive Lösungen, die in anderen Programmiersprachen viele Anweisungen oder eine etwas aufwändigere Programmierung erfordert hätten.
deshalb sollte die Anweisungen mit Tupel gut kommentiert werden

Tupel-Elemente werden in **runde Klammern** notiert
im Prinzip fest definierte und unveränderliche Listen
Zugriff aber – wie üblich bei Indixies – in eckigen Klammern

praktisch fast alle Operationen, wie bei Listen (→ 8.2.3. Listen, die I. – einfache Listen und 9.7. Listen, die II. – objektorientierte Listen) möglich
Tupel lassen sich aber nicht ergänzen oder ändern, nach ihrer Erzeugung sind sie unveränderlich
lassen sich aber aneinanderreihen

ein-elementige Tupel (Singleton) müssen nach dem (ersten) Element noch ein Komma aufweisen!

```
…
# Vertauschen von Werten
if kleinereZahl > groessereZahl:
    kleinereZahl, groessereZahl = groessereZahl, kleinereZahl
…
```

```
//Tauschen in PASCAL
hilfVar:=kleinereZahl;
kleinereZahl:=groessereZahl;
groessereZahl:=hilfVar;
```

Fast jede andere Programmiersprache braucht eine Hilfsvariable oder einen Kellerplatz zum zeitweisen Abspeichern der einen Zahl, damit diese für die Übernahme der anderen bereitsteht, erst dann kann die zweite mit dem Wert aus der Hilfsvariable versehen werden

9.2. Mengen

keine echte Informatik-Daten-Struktur, aber wichtiges Element in Python

Mengen sind Sammlungen von Objekten in denen jedes Objekt nur einmal vorkommt, eine Reihenfolge oder Ordnungs-Struktur gibt es nicht
in der Mathematik weden Mengen in geschweiften Klammern notiert
eine Menge ohne ein Element ist eine leere Menge

9.2.1. Mengen – einfach

9.2.1.1. Mengen-Erstellung

in Python gibt es die veränderlichen Mengen, die mit **set()** erstellt werden und es gibt unveränderliche Mengen für deren Erstellung die Funktion **frozenset()** zuständig ist

```
>>> menge1=set([1,3,5,4,2,3])
>>> menge2=set([2.1, 0.0, 5.3, 7.9])
>>> menge3=set("Farbenspiel")
>>> menge4=set(["gelb", "grün", "rot", "blau", "blau", "blau"])
>>> menge1
{1, 2, 3, 4, 5}
>>> print(menge1)
{1, 2, 3, 4, 5}
>>> menge2
{0.0, 5.3, 7.9, 2.1}
>>> menge3
{'p', 'l', 's', 'F', 'r', 'b', 'a', 'i', 'n', 'e'}
>>> menge4
{'rot', 'gelb', 'grün', 'blau'}
>>>
```

Die Ausgabe erfolgt immer schön ordentlich in geschweiften Klammern. Ev. mehrfach auftauchende Objekte werden eliminiert

```
>>> alphabet=frozenset("abcdefghijklmnopqrstuvwxyz")
>>> print(alphabet)
frozenset({'l', 'k', 'o', 'v', 'i', 'p', 't', 'f', 'a', 'z', 'j',
'n', 'g', 'm', 's', 'w', 'b', 'q', 'u', 'x', 'r', 'h', 'y', 'd', 'c',
'e'})
>>> alphabet
frozenset({'l', 'k', 'o', 'v', 'i', 'p', 't', 'f', 'a', 'z', 'j',
'n', 'g', 'm', 's', 'w', 'b', 'q', 'u', 'x', 'r', 'h', 'y', 'd', 'c',
'e'})
>>>
```

Interessant ist hierbei, dass frozenset's scheinbar anders zusammengestellt werden, als normale Mengen (set's). Die sind sortiert, während die Elemente im frozenset scheinbar willkürlich auftauchen, obwohl sie im Ursprungs-Objekt sortiert vorkamen.
Die Ausgabe verdeutlicht uns immer, dass wir es hier mit einer feststehenden / unveränderlichen Menge zu tun haben.

9.2.1.2. Mengen-Operationen

Ein existierendes Frozenset kann in Python niemals das Ergebnis einer Mengen-Operation werden, da mit ihnen die Unveränderlichkeit verbunden ist. Aber sie können natürlich Argument bzw. Operant sein.

einfache Operationen

len(menge)
gibt die Anzahl der Elemente in der Menge zurück

min(menge)

max(menge)

elem **in** menge

elem **not in** menge

teilmenge **<=** menge
ist True, wenn teilmenge eine Teilmenge von der Menge menge ist

teilmenge **<** menge
ist True, wenn teilmenge eine echte Teilmenge von der Menge menge ist

menge1 **|** menge2
erzeugt neue (Vereinigungs-)Menge von menge1 und menge2; die alle Elemente von beiden Mengen enthält

menge1 **&** menge2
erzeugt neue (Schnitt-)Menge von menge1 und menge2, die nur gemeinsame Elemente enthält

menge **-** teilmenge
erzeugt neue Menge, die alle Elemente von menge enthält, außer sie kommen in teilmenge vor
Differenz-Bildung

menge1 **^** menge2

erzeugt neue (Vereinigungs-)Menge von menge1 und menge2, außer den Elementen, die in beiden mengen enthalten sind
ergibt symmetrische Differenz oder auch Vereinigungs-Menge – Schnitt-Menge

typischen Mengen-Operationen

Zum Veranschaulichen der Mengen-Operationen erstellen wir uns zwei einfache Mengen:

```
>>> menge1=set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> menge2=set([0, 2, 4, 6, 8, 10, 12])
>>> menge1
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> menge2
{0, 2, 4, 6, 8, 10, 12}
>>>
```

Aus der Mathematik kennen wir als typische Mengen-Operationen die Vereinigung, den Durchschnitt und die Differenz.

Vereinigung
Unter der Vereinigung von Mengen versteht man die Gesamt-Menge aus den Teilmengen. In beiden Mengen mehrfach vorkommende Elemente sind in der Ergebnis-Menge natürlich nur einmal vorhanden.

```
>>> menge1 | menge2
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}
>>>
```

Durchschnitt
Der Durchschnitt zweier Mengen beschreibt die Menge der gemeinsamen Elemente aus beiden Mengen. In der Ergebnis-Menge kommen diese Elemente nur einmalig vor.

```
>>> menge1 & menge2
{0, 8, 2, 4, 6}
>>>
```

Differenz
Bei der Differenz von Mengen werden aus der ersten Menge, die Elemente entfernt, die auch in der subtrahierten Menge vorkommen, entfernt.
Die Differenzen zweier Mengen sind i.A. nicht symmetrisch bzw. kommutativ. D.h. normalerweise ist Menge1 – Menge2 ≠ Menge2 – Menge1 (alternative Notierung: Menge1 \ Menge2 ≠ Menge2 \ Menge1).

```
>>> menge1 - menge2
{1, 9, 3, 5, 7}
>>> menge2 - menge1
{10, 12}
>>>
```

Durchschnitt und Vereinigung sind dagegen kommutativ.

Bearbeitung in Schleifen etc.

Mengen lassen sich ebenfalls mit Schleifen durchlaufen. Wir benötigen wieder einen Interator (eine Laufvariable), um auf die einzelnen Elemente zuzugreifen.

iter(menge)
liefert einen Interator für die Menge

9.2.1.x. automatische Mengen-Generierung

Ähnlich, wie bei den Listen (→ 8.4.0.5. Listen-Erzeugung – fast automatisch) lassen sich auch Mengen automatisch generieren. Die Konstrukte unterscheiden sich praktisch nicht. Lediglich die Verwendung der Schlüsselwörtchen set bzw. frozenset kommt hinzu.

```
>>> quadrate=set(i**2 for i in range(16))
>>> quadrate
{0, 1, 64, 225, 4, 36, 100, 196, 9, 169, 16, 49, 81, 144, 25, 121}
>>>
```

Natürlich hätte man die Mengen für die Veranschaulichung der Mengen-Operationen (s.a.w.v.) auch mittels Generator (i for i in range(10)) erzeugen können.

```
>>> menge1=set(i for i in range(10))
>>> menge2=set(i*2 for i in range(7))
>>> menge1
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> menge2
{0, 2, 4, 6, 8, 10, 12}
>>>
```

Weitere Möglichkeiten können gerne in der Listen-Besprechung nachgeschlagen werden (→ 8.4.0.5. Listen-Erzeugung – fast automatisch).

9.2.2. Mengen – objektorientiert

menge1.**add**(element | menge2)
fügt das Element oder eine Menge2 der Menge1 hinzu

menge.**clear**()
löscht alle Elemente aus der Menge
es entsteht eine leere Menge

menge.**discard**(element)
das Element wird aus der Menge entfernt, wenn es dann dort enthalten ist

menge.**pop**()
liefert ein zufällig gewähltes Element aus der Menge zurück. Das Element selbst wird aus der Menge entfernt!

menge.**remove**(element)
das Element wird aus der Menge entfernt, wenn es dann dort enthalten ist, wenn es nicht vorhanden ist, gibt es eine Fehler-Meldung (KeyError)

die aufgezählten Operationen gelten nur für normale Mengen (set's), da sie Veränderlichkeit unterstellen
alle nachfolgenden Operationen gelten für set's bzw. frozenset's

menge.**copy**()
erstellt eine (flache) Kopie der Menge

menge1.**difference**(menge2)
berechnet die Differenz von Menge1 und Menge2 (menge1 – menge2 bzw. menge1 \ menge2)

menge1.**intersection**(menge2)
erstellt den Durchschnitt aus beiden Mengen

menge1.**union**(menge2)(
vereint die beiden Mengen

menge1.**issubset**(menge2)
prüft, ob Menge1 eine Teilmenge von Menge2 ist
liefert True bzw. False zurück

menge1.**issuperset**(menge2)
prüft, ob Menge1 die Obermenge von Menge2 ist
liefert True bzw. False zurück

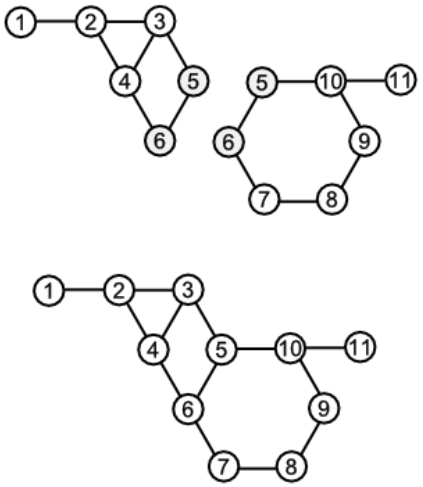
9.2.3. Anwendung von Mengen

9.2.3.1. ein bißchen Graphen

Graphen sind geometrische Objekte, die durch Knoten und Kanten beschrieben werden. Jeder Knoten (hier nummeriert) hat mindestens eine Verbindung (/ Kante) zu einem anderen.

Graphen werden z.B. zur Beschreibung von Wege- oder Raum-Plänen benutzt. Die Einmündungen bzw. Kreuzungen oder eben die Räume entsprechen den Knoten. Die möglichen Verbindungen (Wege oder Türen) zwischen den Knoten sind die Kanten.

Im nachfolgenden Programm werden neben "normalen" Mengen (set's) auch feste Menge (frozenset's) und Tupel (→ 9.1. Tupel) verwendet.



zwei vorgegebene Graphen (oben)
und der gemeinsame Graph (unten)

```
def findeNachbarKnoten(graph, knoten):
# Funktion zum Durchsuchen des Graphen nach den Nachbarknoten
# zu einem vorgegebenem Knoten
    alleKnoten, alleKanten = graph
    KantenDesKnoten = set(k for k in alleKanten if knoten in k)
    NachbarKnoten = set()
    for k in KantenDesKnoten:
        NachbarKnoten = NachbarKnoten | k
        NachbarKnoten= NachbarKnoten - set([knoten])
    return NachbarKnoten

def vereinigeGraphen(graph1, graph2):
# Funktion zur Verbindung von zwei Graphen (gemeinsame Knoten
# müssen gleiche Bezeichnung in beiden Graphen haben (mind. einer notw.!)
    return (graph1[0] | graph2[0], graph1[1] | graph2[1])

# =======Beispiel-Graphen (Daten)

Graph1Knoten={1,2,3,4,5,6}
Graph1Kanten= set(frozenset(k)
                  for k in [(1,2), (2,3), (2,4), (3,4), (3,5), (4,6), (5,6)])
Graph1=(Graph1Knoten, Graph1Kanten)

Graph2Knoten={5,6,7,8,9,10,11}
Graph2Kanten= set(frozenset(k)
                  for k in [(5,6), (5,11), (6,7), (7,8), (8,9), (9,10), (10,11)])
Graph2=(Graph2Knoten, Graph2Kanten)

# =======Hauptprogramm (Beispiel)
GesamtGraph=vereinigeGraphen(Graph1, Graph2)
print("Gesamtgraph: ...")
print("     Knoten: ", end='')
for i in GesamtGraph[0]:
    print(i, end='; ')
```

```
print()
print("     Kanten: ", end='')
for j in GesamtGraph[1]:
    print(tuple(j), end='; ')
print()
print("---------------------------------")
eingabe=1
while eingabe>0:
    eingabe=eval(input("Für welchen Knoten werden die Nachbarn gesucht?"
                       +" (Abbruch mit 0) ?: "))
    if eingabe>0:
        NachbarKnoten=findeNachbarKnoten(GesamtGraph, eingabe)
        print("Der Knoten",eingabe,"hat die / den Nachbarknoten: ",end='')
        for n in NachbarKnoten:
            print(n, end=', ')
        print()
    print()
input()
```

```
>>>
Gesamtgraph: ...
     Knoten: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
     Kanten: (2, 4), (1, 2), (5, 6), (6, 7), (3, 5), (8, 9), (3, 4),
(8, 7), (2, 3), (4, 6), (10, 11), (11, 5), (9, 10),
---------------------------------
Für welchen Knoten werden die Nachbarn gesucht? (Abbruch mit 0) ?: 4
Der Knoten 4 hat die / den Nachbarknoten: 2, 3, 6,

Für welchen Knoten werden die Nachbarn gesucht? (Abbruch mit 0) ?: 1
Der Knoten 1 hat die / den Nachbarknoten: 2,

Für welchen Knoten werden die Nachbarn gesucht? (Abbruch mit 0) ?: 0

>>>
```

9.3. Dictonary's - Wörterbücher

übersetzt Verzeichnis, praktisch eine Sammlung von Daten-Paaren, einem Schlüsselwert (Key) und einem Datenwert oder Eintrag (Value)
entspricht also einem Wörterbuch, deshalb auch gerne für direkte Übersetzungen benutzt. In der gesprochenen Sprache aber eher nur für einzelne Worte geeignet.
Angabe in geschweiften Klammern
prinzipiell Listen-artige Struktur; kann durch eine Liste aus zwei-stelligen Listen ersetzt werden
für "mehr-sprachige" Wörterbücher ist u.U. eine "mehr-spaltige" ("mehr-stellige") Liste besser geeignet.
(dann benötigt man auch keine gespiegelten Dictonary's, wobei dass auch nur programmtechnisch interessant ist)

Notierung auch mehrzeilig möglich, dann muss die erste geschweifte Klammer in der Defintions-Zeile stehen und die abschließende Klammer hinter der letzten Zeile.
Die Wörterbuch-Einträge sollten Zeilen-weise notiert werden.
Dictonary's eigenen sich auch gut zum Abspeichern und Einlesen aus einer Text-Datei.

ebenfalls keine typische Daten-Struktur, Mischung aus Daten-Strukturen für die interne Daten-Verwaltung und -Verarbeitung
besonders auch bei der Speicherung der Daten bedeutsam

Anhängen eines neuen Eintrags
dictonary_name[neuer_schlüssel] = neuer_eintrag

Bestimmen der Länge / Größe / Einträgezahl eines Dictonary's
len(dictonary_name**)**
jedes Paar zählt als ein Eintrag

Löschen eines Eintrages durch Angabe des Schlüssels
del dictonary_name[schlüssel]

ist die Position bekannt, dann kann auch ein Löschen über den Index erfolgen
del(dictonary_name[index]**)**

weitere Objekt-orientierte Möglichkeit des Löschens eines Eintrages über **.pop()**

```
Koordinaten = {
"Erlangen": [56,23],
"Berlin": [34,51],
"München": [12,23],
"Rostock": [11,45]
}

for key in Koordinaten:
      print( key, Koordinaten[key])
```

```
>>>
München [12, 23]
Rostock [11, 45]
Erlangen [56, 23]
Berlin [34, 51]
```

```
Hauptstädte={
    "Baden-Württemberg":"Stuttgart",
    "Bayern":"München",
    "Berlin":"Berlin",
    "Brandenburg":"Potsdam",
    "Bremen":"Bremen",
    "Hamburg":"Hamburg",
    "Hessen":"Wiesbaden",
    "Mecklenburg-Vorpommern":"Schwerin",
    "Nordrhein-Westfalen":"Düsseldorf",
    "Niedersachsen":"Hannover",
    "Rheinland-Pfalz":"Mainz",
    "Saarland":"Saarbrücken",
    "Sachsen":"Dresden",
    "Sachsen-Anhalt":"Magdeburg",
    "Schleswig-Holstein":"Kiel",
    "Thüringen":"Erfurt"
}

for k in hauptstädte.items():
    print(k[0]+ " hat die Hauptstadt " + k[1])
```

```
>>>
Baden-Würtemberg hat die Hauptstadt Stuttgart
Bayern hat die Hauptstadt
Berlin hat die Hauptstadt Berlin
Brandenburg hat die Hauptstadt Potsdam
Bremen hat die Hauptstadt Bremen
Hamburg hat die Hauptstadt Hamburg
Mecklenburg-Vorpommern hat die Hauptstadt Schwerin
Nordrhein-Westfalen hat die Hauptstadt Düsseldorf
Niedersachsen hat die Hauptstadt Hannover
Rheinland-Pfalz hat die Hauptstadt Mainz
Saarland hat die Hauptstadt Saarbrücken
Sachsen hat die Hauptstadt Dresden
Sachsen-Anhalt hat die Hauptstadt Magdeburg
Schleswig-Holstein hat die Hauptstadt Kiel
Thüringen hat die Hauptstadt Erfurt
```

```
def Stadtstaaten():
    StadtstaatenListe=[]
    for land in Hauptstädte.items():
        if land[0] == land[1]:
            StadtstaatenListe.append(land[0])
    return StadtstaatenListe


def spiegeln(Dict):
    SpiegelDict={}
    for eintrag in Dict.items():
        SpiegelDict[eintrag[1]]=eintrag[0]
    return SpiegelDict
```

Schlüssel und Werte eines Wörterbuch's spiegel

```
woerterbuch = {
    1: "uno"
    2: "due"
    3: "tres"
    '0': "zero"
}

print(woerterbuch)

getauschtesWoerterbuch = {}
for schluessel, wert in woerterbuch.items():
    if wert not in getauschtesWoerterbuch:
        getauschtesWoerterbuch[wert] = []
    getauschtesWoerterbuch[wert].append(schluessel)


print(getauschtesWoerterbuch)
```

Aufgaben:

1. *Erstellen ein Wörterbuch und ein kleines Anzeige-Programm (für alle Einträge) für die nächstkleinere Verwaltungs-Einheit Ihres Bundeslandes / Stadtstaates!*
2. *Ergänzen Sie dann eine – sich wiederholende – Abfrage eines beliebigen Schlüssels aus Ihrem Dictonary mit Anzeige des zugehörigen Eintrages in Form eines vollständigen Satzes! Der Abbruch der Eingabe soll bei der Eingabe eines leeren Textes erfolgen, fehlerhafte Eingabe sollen als solche auf dem Bildschirm vermerkt werden!*
3. *Erstellen Sie ein Deutsch-Englisch- und Englisch-Deutsch-Wörterbuch-Programm, dass bekannte Wort-Paare anzeigt und unbekannte lernt! (Wir gehen von einer immer richtigen Eingabe der Vokabeln aus!) Das Anfangs-Vokabular soll mindestens 30 Wort-Paare bzw. die Vokabeln der letzten Unterrichts-Module enthalten.*

9.3.1. Erfassen von unbekannten Objekten und Zählen der Objekte in einem Wörterbuch

Da man Dictonary's mit beliebigen Schlüsseln betreiben kann und auch die Aufnahme-Men ge an unterschiedlichen Objekten nicht begrenzt ist, bieten sie sich für das Zählen von irgendwelchen Objekten an.
Damit das Dictonary sowohl in der Funktion – als auch im Haupt-Programm nutzbar ist, definiert man es vor der Notierung der Funktion (im Haupt-Programm).

```
Anzahl={}

def zaehle(objekt):
  Anzahl[objekt]+=1

print(Anzahl)
zaehle("Maus")
print(Anzahl)
```

Es gibt allerdings ein Problem: In unserem Anzahl-Wörterbuch gibt es gar keine Schlüsselwörter.
Habe ich eine definierte Menge, kann ich sie im Vorfeld gleich mit definieren. Universeller – aber auch nicht perfekt – ist die nebenstehende Version:
Jetzt wird immer dann, wenn kein passendes Objekt in dem Anzahl-Wörterbuch gefunden wird, ein neues mit dem Zähl-Wert 1 angelegt.
Überlegen wir uns nun noch ein kleines Test-Programm für unsere Wörterbuch-Struktur.
Dabei wird auch die Effektivität der Speicherung deutlich.

```
Anzahl={}

def zaehle(objekt):
  if objekt in Anzahl:
    Anzahl[objekt]+=1
  else:
    Anzahl[objekt]=1

zaehle("Maus")
print(Anzahl)
```

Statt in einem Feld von 100 Objekten, werden jetzt nur die Objekte erfasst und zählend gespeichert, die wirklich bei einem Zufalls-Erzeugungs-Verfahren entstehen.
Bei einer weiteren Nutzung des Wörterbuches muss man ev. beachten, dass es bestimmte Einträge eben nicht gibt. Um hier keinen Laufzeitfehler zu bekommen, muss man dann vor der Benutzung immer die Existenz abtesten!

```
from random import randint

Anzahl={}

def zaehle(objekt):
  if objekt in Anzahl:
    Anzahl[objekt]+=1
  else:
    Anzahl[objekt]=1

for _ in range(50):
  zaehle(randint(1,20))
print(Anzahl)
```

```
>>>
{1: 4, 2: 1, 3: 5, 5: 3, 6: 3, 7: 2, 8: 3, 9: 3, 10: 6,
11: 6, 12: 1, 13: 2, 14: 2, 15: 3, 16: 3, 17: 1, 19: 1,
20: 1}
>>>
```

Aufgaben:

1. ***Verbessern Sie das obige Programm zum Zählen der Zufallszahlen um eine verständliche Ausgabe!***
2. ***Erstellen Sie eine Programm-Version eines Würfel-Programms, das 200x würfelt und die Wurf-Anzahl hinterher als Balken-Diagramm aus Rauten darstellt!***
3.

9.3.2. Objekt-orientierte Operationen mit Dictonary's

Löschen eines Eintrages aus dem Dictonary über den Index
dictonary_name**.pop(**index**)**
die Funktion liefert ein reduziertes Dictonary zurück

dictonary_name**.clear()**
löscht den Inhalt des Dictonary

dictonary_name**.copy()**
erzeugt eine flache Kopie von dictonary_name
erzeugt Alias-Dictonary

dictonary_name**.items()**
gibt eine Liste aller gespeicherter Schlüssel mit ihren Werten im Dictonary (jeweils Tupel-weise) zurück

dictonary_name**.keys()**
gibt eine Liste aller gespeicherter Schlüssel im Dictonary zurück

dictonary_name**.key()**

dictonary_name**.values()**
gibt eine Liste aller gespeicherter Werte (Value's) im Dictonary zurück

dictonary_name**.values()**
gibt eine Liste aller gespeicherter Werte im Dictonary zurück

dictonary_name**.get(schluessel, alternativwert)**
liefert den Wert zum Schlüssel zurück (wenn dieser existiert); sonst den Alternativwert

dictonary_name**.setdefault(schluessel, inhalt)**
setzt im Schlüssel-Eintrag/Inhalt-Paar mit dem angegebenen Schlüssel (wenn dieser vorhanden ist) den Inhalt → dic[schluessel] = inhalt
wenn Eintrag mit Schlüssel schon vorhanden ist, dann wird dessen aktueller Inhalt zurückgeliefert

dictonary_name**.iteritems()**
zum Durchlaufen aller Schlüssel-Eintrag/Inhalt-Paare in einer for-Schleife

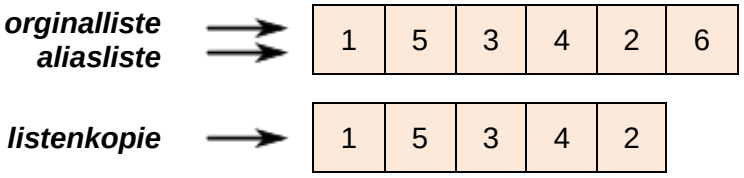
dictonary_name**.iterkeys()**
zum Durchlaufen aller Schlüssel in einer for-Schleife

dictonary_name**.itervalues()**
zum Durchlaufen aller Einträge einer for-Schleife

9.7. Listen, die II. – objektorientierte Listen

Im Abschnitt "Listen, die I. (→ 8.4. Listen, die I. – einfache Listen)haben wir die Listen ganz einfach betrachtet. Nun gehen wir zur Objekt-orientierten Nutzung über. Das hört sich vielleicht irgendwie kompliziert an, ist es aber gar nicht. Vom Objekt-orientierten Ansatz merken wir kaum etwas. Nur die übliche Objekt-orientierte Schreibung bzw. der Aufruf der Funktionen über die Punkt-Schribweise erinnert an sie. Ach ja, und jetzt heißen die Funktionen Methoden. Da werden wir uns aber schnell eingewöhnen.

Wir greifen auf auf das Listen-Beispiel aus dem ersten kapitel zurück. Wer will, kann ja noch mal schnell nachschlagen (→ 8.4. Listen, die I. – einfache Listen).



einige Methoden auf Listen lassen Operationen zu, die man eher den Keller- bzw. Warteschlangen-Datenstrukturen zuordnen würden
hie wird wieder die herausragende Rolle der Listen als Daten-Objekte (in Python und auch sonst) sichtbar

listenname.**remove**(element)
löscht aus der Liste das angegebene Element

??? listenname**.del(**index**)**
löscht aus der Liste das Element an der Index-Position

listenname**.count(**element**)**
zählt, wie häufig ein Elementes in einer Liste ist

listenname**.append(**element**)**
hängt ein Element an die Liste mit dem angegebenen Namen an
werden mehrere Elemente angehängt, dann erfolgt dies als ein Element, also als eigene Liste, das Ergebnis wäre eine Liste am Ende der (alten) Liste
Rückgabewert ist None, dieser muss aber nicht entgegengenommen werden

listenname**.extend()**
hängend ein oder mehrere Elemente einzeln an eine Liste an

listenname**.insert(**index, element**)**
fügt ein Element in eine Liste an der indizierten Position ein

element=listenname**.pop()**
liefert das letzte Element der Liste zurück, es wird aus der Liste entfernt!

einige Programmiersprachen kennen neben pop auch noch peek bei peek wird das letzte Element aber nicht entfernt, sondern nur gelesen; in python lässt sich hierfür liste[-1] benutzen

listenname**.sort()**
sortiert die Elemente einer Liste aufsteigend
bietet als Argument noch die Schlüsselwörtchen key und reverse
mit reverse wird die Liste absteigend (also umgekehrt) sortiert

listenname.sort(reverse=True)

mit dem Wörtchen key kann man der Sortierfunktion noch eine einargumentige Funktion übergeben, die als Sortierkriterium dienen soll z.B. die Länge der Strings (→ len())

listenname.sort(key=len)

die Schlüsselwörtchen lassen sich auch gemeinsam verwenden

durch den Allias wird lediglich ein weiterer Zeiger (genannt Variable) auf die gleiche Liste gelegt

beim Verändern der einen Liste wird die "andere" Liste mit geändert

nur mit Deep-Kopie kann eine unabhängige Kopie erstellt werden

```
>>> a=[3,2,1]
>>> a
[3, 2, 1]
>>> b=a
>>> b
[3, 2, 1]
>>> a.sort()
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>>
```

auch bei Übernahme einer Alias-Liste in eine andere Liste wird nur der Zeiger übernommen

Veränderungen an der "originalen" Alias-Liste wirken sich auch eine weitere Alias-Nutzung aus

```
>>> a=[3,2,1]
>>> a
[3, 2, 1]
>>> b=["A", "B","C"]
>>> b
["A", "B", "C"]
>>> a.append(b)
>>> a
[1, 2, 3,["A", "B", "C"]]
>>> b.reverse
>>> b
["C", "B", "A"]
>>> a
[1, 2, 3, ["C", "B", "A"]]
>>>
```

listenname**.reverse()**
sortiert die Elemente einer Liste absteigend

listenkopie = copy.deepcopy(originalliste)

liste = ['a', 'b', 'c']]
print(' '.join(liste)) ➔ 'abc'

Listen:
Vorteile:

- effektive Speichernutzung
- schnelles Einspeichern (Anhängen)
- einfache Algorithmen (suchen (, entfernen, einfügen an Position))

Nachteile:

- allgemein Arbeits-aufwändiger
- langsames Suchen

9.8. Keller

auch Stack (engl. = Stapel, Haufen)

was auf dem Stapel als letztes abgeladen wird, muss als erstes wieder entnommen werden, um z.B. an tiefer liegende / früher eingespeicherte Daten zu erreichen

bekannt z.B. aus der Rekursion (→ 8.4.2. Rekursion) dort ist Kellerspeicher zwingend notwendig, allerdings vom Nutzer unbemerkt

Größe einer KELLER_Datenstruktur wird im Wesentlichen vom verfügbaren / hierfür reservierten Speicher bestimmt
ansonsten Anzahl der Einträge beliebig

nur wenn Speicher des Rechners nicht mehr für die Größe des Kellerspeichers ausreicht (bei zu vielen Rekursionen), dann kommt es zum Fehler

LIFO-Speicher (Last-In-First-Out) oder Stack

alternatives Speicher-Prinzip ist die (Warte-)Schlange (→ 9.9. Warteschlangen) oder der FIFO-Speicher (First-In-First-Out)

Definition(en): Stack / Keller
Ein Keller- oder Stack-Datenstruktur ist eine lineare Anordnung von gleichartig zu bearbeitenden Daten-Obbjekten (Daten-Einträgen), die nach dem LIFO-Prinzip verwaltet werden.

zum Keller gehörenden Grund-Operationen:
nachsehen (top) → obersten Eintrag ansehen / auslesen ohne es zu entfernen
einlagern / einspeichern (push) → neues Element (oben) auf den alten Stapel legen
wegnehmen / ausspeichern (pop) → obersten Eintrag entnehmen

Stack	**Typ / Klasse**
liste: Liste (von Objekt)	Objekt
gibListe(): Liste setzListe(liste: Liste) istLeer(): Wahrheitswert einspeichern(eintrag: Objekt) ausspeichern(): Objekt lesen(): Objekt	Methoden

eine sehr einfache Implementierung eines Kellers (Stack's) über eine interne Liste:

```
def keller():
    liste = []

    def raus():
        if not istleer():
            return liste.pop()

    def rein(element):
        liste.append(element)

    def istleer():
        return len(liste)==0

    return raus, rein, istleer

REIN, RAUS, ISTLEER = keller()
```

Q und Fkt.W: ???; angelehnt an Objekt-orientierter Prog.

Listen-basiert
push = kellerliste.append
pop = kellerliste.pop

```
# Menü-System + Keller-Speicher mit Listen-Opp's

def anzeigenListe(Liste):
    for elem in Liste:
        print("[",elem,"]",end='  ')
    print("|<=")

def elementeZaehlen(Liste):
    anzahl=0
    for _ in Liste:
        anzahl+=1
    return anzahl

def istLeereListe(Liste):
    anzahl=elementeZaehlen(Liste)
    if anzahl==0:
        return True
    else: return False

def leerenListe(Liste):
    while not istLeereListe(Liste):
        Liste.pop()


# Main
KellerSpeicher=[]
maxMenuePunkte=3
auswahl=1
while auswahl>0 and auswahl<=maxMenuePunkte:
    print("")
    print("aktueller Keller-Speicher:")
    anzeigenListe(KellerSpeicher)
    print("")
    print("Auswahl-Menü")
    print("============")
    print("<1> .. Einspeichern (Push)")
    print("<2> .. Ausspeichern (Pop)")
    print("<3> .. Speicher leeren")
    print(".. ")
    print("<0> .. Programmende")
    auswahl=-1
    while auswahl<0 or auswahl>maxMenuePunkte:
         auswahl=eval(input("Ihre Wahl: "))
    if auswahl==1:
         eingabe=input("Was soll eingespeichert werden?: ")
         KellerSpeicher.append(eingabe)
    elif auswahl==2:
         if istLeereListe(KellerSpeicher):
             print("Keller-Speicher ist LEER.")
         else:
             ausgabe=KellerSpeicher.pop()
             print("Element: [", ausgabe, "] aus dem Speicher gelesen
und entfernt.")
    elif auswahl==3:
         leerenListe(KellerSpeicher)
    else:
        break

print("Ende...")
```

```
aktueller Keller-Speicher:
|<=

Auswahl-Menü
============
<1> .. Einspeichern (Push)
<2> .. Ausspeichern (Pop)
<3> .. Speicher leeren
<4> ..
<1> ..
..
<0> .. Programmende
Ihre Wahl: 1
Was soll eingespeichert werden?: 3

aktueller Keller-Speicher:
[ 3 ]  |<=
```

etwas aufwändigere Implementierung (Q: de.wikipedia.org)

```
class Stack(object):
    def __init__(self):
        self.maxindex=5
        self.topindex=0
        self.speicher = [0,0,0,0,0,0,0,0,0,0,0]

    def isEmpty(self):
        return self.topindex==0
    def isFull(self):
        return self.topindex==self.maxindex
    def push(self,element):
        if not self.isFull():
            self.topindex+=1
            self.speicher[self.topindex]=element
    def pop(self):
        if not self.isEmpty():
            self.topindex-=1
    def top(self):
        if not self.isEmpty():
            return self.speicher[self.topindex]

    def DisplayStack(self):
        M=self.topindex

        while M>0 :

            print("|     ",self.speicher[M],"         |")
            M-=1
        print("------------------|")



if __name__== "__main__":                              #Beispiel
    myStack=Stack()
    print(myStack.isFull())
    print(myStack.isEmpty())
    myStack.push(5)
    myStack.push(3)
    myStack.DisplayStack()
    print(myStack.isEmpty())
    myStack.push(13)
    myStack.DisplayStack()
```

9.9. Warteschlangen

FIFO-Prinzip (First In First Out)

Queue (sprich: kju)

Beispiele:
Kasse im Supermarkt
Warten beim Frisör / Arzt / …
Abarbeitung von Überweisungen bei einer Bank

Definition(en): Warteschlange / Queue
Eine Warteschlangen- bzw. Queue-Datenstruktur ist eine lineare Anordnung von gleichartig zu bearbeitenden Daten-Objekten (Daten-Einträgen), die nach dem FIFO-Prinzip verwaltet werden.

zum Keller gehörenden Grund-Operationen:
nachsehen (front) → ersten Eintrag ansehen / auslesen ohne ihn zu entfernen
anhängen / einspeichern (enqueue) → neues Element (hinten) auf die alten Schlange anhängen / kontatenieren
entfernen / ausspeichern (dequeue) → ersten / vordersten Eintrag entnehmen

alternatives Speicher-Prinzip ist der Keller (→ 9.8. Keller) bzw. der Stack (Stapel) oder der LIFO-Speicher

Queue	**Typ / Klasse**
liste: Liste (von Objekt)	Objekt
gibListe(): Liste setzListe(liste: Liste) istLeer(): Wahrheitswert einspeichern(eintrag: Objekt) ausspeichern(): Objekt lesen(): Objekt	Methoden

für eine Implementierung über eine einfache Liste:

links-orientiertes Arbeiten:
insert(0,x) zum Einspeichern (neue Liste ::= x, alte Liste) (Erweiterung links)
pop() Ausspeichern / Entnehmen des letzten Element's (Verkürzung rechts)

rechts-orientiertes Arbeiten:
append(x) anhängen eines Eintrags an die Liste (einspeichern) (Erweiterung rechts)
pop(0) Ausspeichern / Entnehmen des 1. Element's (Verkürzung links)

```
# Menü-System + Warteschlangen-Speicher mit Listen-Opp's

def anzeigenListe(Liste):
    print("->|",end='  ')
    for elem in Liste:
        print("[",elem,"]",end='  ')
    print("|=>>")

def elementeZaehlen(Liste):
    anzahl=0
    for _ in Liste:
        anzahl+=1
    return anzahl
    # Alternative
    # return len(Liste)

def istLeereListe(Liste):
    anzahl=elementeZaehlen(Liste)
    if anzahl==0:
        return True
    else: return False
    # Alternative
    # if len(Liste)==0: return True
    # return False

def leerenListe(Liste):
    while not istLeereListe(Liste):
        Liste.pop()


# Main
WarteschlangenSpeicher=[]
maxMenuePunkte=3
auswahl=1
while auswahl>0 and auswahl<=maxMenuePunkte:
    print("")
    print("aktueller FIFO-Speicher (Warteschlange):")
    anzeigenListe(WarteschlangenSpeicher)
    print("")
    print("Auswahl-Menü")
    print("============")
    print("<1> .. Einspeichern (Push)")
    print("<2> .. Ausspeichern (Pop)")
    print("<3> .. Speicher leeren")
    print(".. ")
    print("<0> .. Programmende")
    auswahl=-1
    while auswahl<0 or auswahl>maxMenuePunkte:
         auswahl=eval(input("Ihre Wahl: "))
    if auswahl==1: # Einspeichern
         eingabe=input("Was soll eingespeichert werden?: ")
         WarteschlangenSpeicher.insert(0,eingabe)
```

```
    elif auswahl==2: # Ausspeichern
         if istLeereListe(WarteschlangenSpeicher):
             print("Keller-Speicher ist LEER.")
         else:
             ausgabe=WarteschlangenSpeicher.pop()
             print("Element: [", ausgabe, "] aus dem Speicher gelesen
und entfernt.")
    elif auswahl==3: # Speicher leeren
         leerenListe(WarteschlangenSpeicher)
    else: # Ende
        break

print("Ende...")
```

9.10. Bäume

9.11. Graphen

siehe auch bei Listen II
siehe auch bei Mengen → 9.2.3.1. ein bißchen Graphen

9.12. endliche Automaten

9.13. TURING-Automaten

9.14.

10. Python für spezielle Fälle

10.1. Python in Zusammenarbeit mit anderen Anwender-Programmen

interessante Links:
https://automatetheboringstuff.com/ (online-Version des Buches: AL SWEIGART: Automate the Boring Stuff with Python – Practical Programming for Total Beginners)

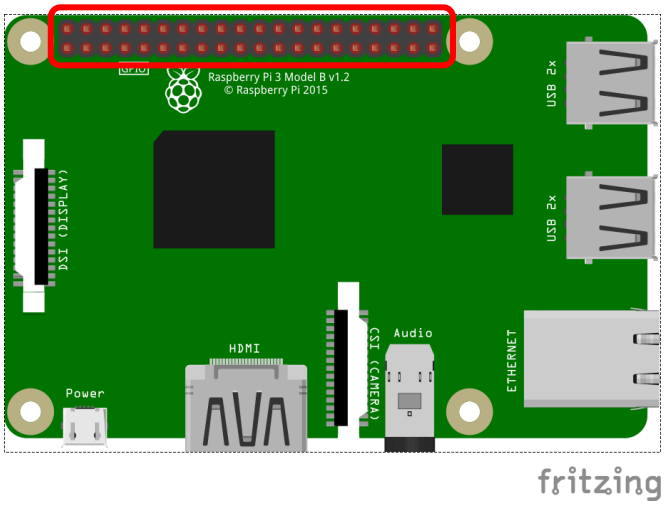
10.2. Steuerung externer Hardware (RaspberryPi, Arduino)

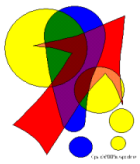
10.2.1. Raspberry Pi und Verwandte

10.2.1.0. Kurzbeschreibung und allgemeine Einführung zu Raspberry Pi

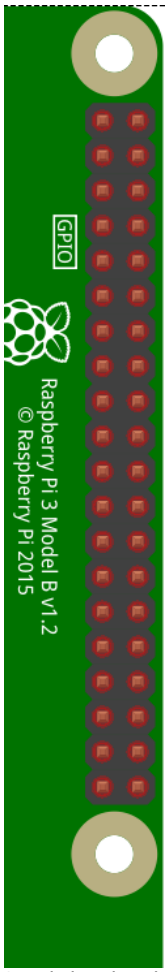
10.2.1.1. die GPIO-Schnittstelle

GPIO ist eine 40-polige Anschluß-Leiste mit verschiedenen Ein- und Ausgängen zum Board. Dient dem Anschluß von Zusatz-Platinen (Shield's) oder von elektrischen / elektronischen Schaltungen.





Achtung:
Die Benutzung der Pin's muss exakt eingehalten werden. Ein Wechsel auf andere Pin's ist nur mit genauer Vorüberlegung möglich.
Ein Verwechslung von Pin's oder Polungen kann zur Zerstörung von Bauelementen und / oder der Raspberry-Platine führen.
Schaltungen sollten vor der Benutzung immer noch einmal kontrolliert werden (4-Augen-Prinzip empfohlen!). Erst wenn alles übereinstimmt, dann als Letztes Masse oder Spannung-Pin einstecken!



Q: Fritzing; bearb.: dre

	innen	*außen*	
1	3,3V	5V	2
3	2	5V	4
5	3	Gnd	6
7	4	14	8
9	Gnd	15	10
11	17	18	12
13	27	Gnd	14
15	22	23	16
17	3,3V	24	18
19	10	Gnd	20
21	9	25	22
23	11	8	24
25	Gnd	7	26
27	ID_SD	ID_SC	28
29	5	Gnd	30
31	6	12	32
33	13	Gnd	34
35	19	16	36
37	26	20	38
39	Gnd	21	40
Pin	**Name**		

10.2.1.2. Steuerung über die GPIO-Schnittstelle

```
# LED an GPIO23 blinken
import RPi.GPIO as GPIO           # Bibliothek für GPIO-Steuerung
import time                       # Bibliothek für Zeitsteuerung

GPIO.setmode(GPIO.BCM)            # GPIO-Namen verwenden
pin=23                            # GPIO-Pin
GPIO.setup(pin, GPIO.OUT)         # GPIO-Pin auf Ausgabe

zeit=0.5                          # Hell-Dunkel-Wartezeit
while True:                       # Endlos-Schleife
    GPIO.output(pin, GPIO.HIGH)   # Pin ein
    time.sleep(zeit)
    GPIO.output(pin, GPIO.LOW)    # Pin aus
    time.sleep(zeit)
```

```
# LED per Taster EIN und AUS
import RPi.GPIO as GPIO               # Bibliothek für GPIO-Steuerung
import time                           # Bibliothek für Zeitsteuerung

GPIO.setmode(GPIO.BCM)                # GPIO-Namen verwenden
pinAus=23                             # GPIO-Pin
pinEin=24                             # GPIO-Pin
GPIO.setup(pinAus, GPIO.OUT)          # GPIO-Pin auf Ausgabe
GPIO.setup(pinEin, GPIO.IN)           # GPIO-Pin auf Eingabe

zeit=0.5                              # Hell-Dunkel-Wartezeit
for i in range(3):                    # 3x wiederholen (blinken)
    GPIO.output(pinAus, GPIO.HIGH)       # LED an
    time.sleep(zeit)                     # warten
    GPIO.output(pinAus, GPIO.LOW)        # LED aus
    time.sleep(zeit)                     # warten

while True:                          # Endlosschleife
    if GPIO.input(pinEin) == 0:      # Abfrage Eingabe-Pin = geschlossen
        GPIO.output(pinAus, GPIO.LOW)  # Ausschalten
    else:
        GPIO.output(pinAus, GPIO.HIGH) # Einschalten
```

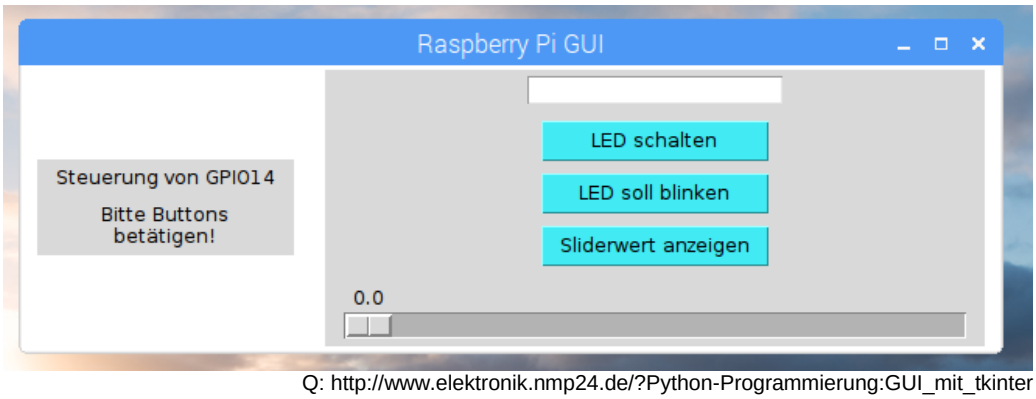
die Endlos-Schleifen lassen sich mit [Strg] + [c] abbrechen

10.2.1.3. direkte Steuerung der IO-Port

10.2.1.4. Objekt-orientiertes Programmieren

eine Sensor-Klasse
→ http://www.forum-raspberrypi.de/Thread-tutorial-einfuehrung-in-objektorientierte-programmierung-mit-python

10.2.1.5. GUI mit Tkinter



Q: http://www.elektronik.nmp24.de/?Python-Programmierung:GUI_mit_tkinter

```
#GUI für das Ein- und Ausschalten einer LED an GPIO14
from tkinter import * #Grafikbibliothek
root = Tk() # Fenster erstellen
root.wm_title("Raspberry Pi GUI") # Fenster Titel
root.config(background = "#FFFFFF") # Hintergrundfarbe des Fensters
#GPIO- und time-Bibliothek:
import RPi.GPIO as GPIO
import time
#festlegen, dass GPIO-Nummern verwendet werden:
GPIO.setmode(GPIO.BCM)
#GPIO14 als Ausgang:
GPIO.setup(14, GPIO.OUT)

# Hier werden zwei Frames erzeugt:
leftFrame = Frame(root, width=200, height = 400) # Frame initialisieren
leftFrame.grid(row=0, column=0, padx=10, pady=3) # Relative Position und Seiten-
abstand (padding) angeben
# Hier kommen die Elemente des linken Frames rein
leftLabel1 = Label(leftFrame, text="Steuerung von GPIO14")
leftLabel1.grid(row=0, column=0, padx=10, pady=3)
leftLabel2 = Label(leftFrame, text="Bitte Buttons\nbetätigen!")
leftLabel2.grid(row=1, column=0, padx=10, pady=3)

rightFrame = Frame(root, width=400, height = 400)
rightFrame.grid(row=0, column=1, padx=10, pady=3)

# Hier kommen die Elemente des rechten Frames rein
# callback1 ist der Handler von Button B1

def callback1():
    varLEDStatus = GPIO.input(14)
    if varLEDStatus == 0:
        GPIO.output(14, GPIO.HIGH)
```

```
        E1.delete(0, END)
        E1.insert(0, "LED ist eingeschaltet")
    else:
        GPIO.output(14, GPIO.LOW)
        E1.delete(0, END)
        E1.insert(0, "LED ist ausgeschaltet")

def callback2():
    for i in range(5):
        GPIO.output(14, GPIO.HIGH)
        time.sleep(0.5)
        GPIO.output(14, GPIO.LOW)
        time.sleep(0.5)

def callback3():
    print (Slider.get())
    E1.delete(0, END)
    E1.insert(0,  "Slider = ")
    E1.insert(12,  Slider.get())

buttonFrame = Frame(rightFrame)
buttonFrame.grid(row=1, column=0, padx=10, pady=3)

E1 = Entry(rightFrame, width=20)
E1.grid(row=0, column=0, padx=10, pady=3)

B1 = Button(buttonFrame, text="LED schalten", bg="#42ebf4", width=15, com-
mand=callback1)
B1.grid(row=1, column=0, padx=10, pady=3)
B2 = Button(buttonFrame, text="LED soll blinken", bg="#42ebf4", width=15, com-
mand=callback2)
B2.grid(row=2, column=0, padx=10, pady=3)
B3 = Button(buttonFrame, text="Sliderwert anzeigen", bg="#42ebf4", width=15,
command=callback3)
B3.grid(row=3, column=0, padx=10, pady=3)

Slider = Scale(rightFrame, from_=0, to=100, resolution=0.1, orient=HORIZONTAL,
length=400)
Slider.grid(row=3, column=0, padx=10, pady=3)

root.mainloop() # GUI wird upgedated. Danach keine Elemente setzen
```

Q: http://www.elektronik.nmp24.de/?Python-Programmierung:GUI_mit_tkinter

Links:

https://tutorials-raspberrypi.de

10.2.1.6. programmiertes Spielen mit microsoft Minecraft

Programm-Beispiele (gelbbräunlich (hell oliv)) stammen aus dem conrad / Franzis Adventskalender Programmieren mit Minecraft 2018

Python und Minecraft stellen auf dem Raspberry Pi eine besonders Preis-günstige Kombination dar. Praktisch hat man nur die Hardware-Kosten für den Pi und die notwendigen Zubehör-Teile. Weder Python noch Minecraft kostet auf dem Pi Geld. Microsoft hat für den Pi eine kostenlose Variante von Minecraft spendiert. Für erste Erfahrungen ist das schon mehr als genug. Wir können alle wesentlichen Funktionen von Mindecraft programmieren und damit unser eigenes Spiel erstellen.
Für Python-Anfänger ist das Objekt-orientierte sicher eine sehr große Herausforderung. Aber die schon fortgeschrittenen Programmierer mit Grundkenntnissen in OOP werden sicher sehr schnell zurecht kommen.

Position des Spieler auswerten

Die GPIO verfügt über 16 (???) Digital-Ausgänge. Das heißt diese können durch Programme AN oder AUS geschaltet werden. Für die meisten Schaltungen bedeutet das für AN liegt eine Spannung von 5 V an und bei AUS ist es 0 V. An den digitalen Ausgängen sind nur diese beiden Zustände zugelassen. Zwischen-Größen sind nicht möglich. Dies ist nur an analogen Port's möglich (→).
In diesem Projekt soll die Position des Spieler in der Würfel-Welt erfasst und ausgewertet werden. Wenn der Spieler einen bestimmten Bereich in der Minecraft-Welt erreicht hat, dann soll eine zweite LED (rot) leuchten.
an Pin 23 und Pin 25 muss jeweils eine LED entweder mit integriertem Vorwiderstand oder ein solcher in Reihe mit einer normalen LED geschaltet werden (das lange Beinchen der LED's steht für den Plus-Pol und kommt an den Pin
das andere (kürzere) Beinchen ist auf Minus bzw. Masse zu schalten

unbedingt die richtige Polung und auch die Pin-Auswahl beachten, ansonsten köönten die Bauelemente oder gar der Pi beschädigt werden

am Besten Aufbau auf einem SteckBrett

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()

GPIO.setmode(GPIO.BCM)
GPIO.setup(23, GPIO.OUT)
GPIO.setup(25, GPIO.OUT)

while True:

  p = mc.player.getTilePos()

  if (p.x==8 and p.z>=3 and p.z <=4):
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc
Setzen der Pin-Namen lt. BCM
Initialisierung des Port 23 für Ausgabe
Initialisierung des Port 25 für Ausgabe

Endlos-Schleife (wird durch Spiel unterbrochen)
Holen der akt. Spieler-Position und Speichern in p
Prüfen ob Spieler in einem bestimmten Be-

```
    GPIO.output(23, True)
    GPIO.output(25, False)
  else:
    GPIO.output(25, True)
    GPIO.output(23, False)
```

reich ist (hier x=8 und z zwischen 3 und 4)
Wenn ja, **dann** Ausgabe 1 (Spannung) auf Pin 23 und 0 (keine Spannung) auf Pin 25
Sonst
Ausgabe 1 (Spannung) auf Pin 25 und 0 (keine Spannung) auf Pin 23

an Pin 23 und Pin 25 muss jeweils eine LED entweder mit integriertem Vorwiderstand oder ein solcher in Reihe mit einer normalen LED geschaltet werden (das lange Beinchen der LED's steht für den Plus-Pol und kommt an den Pin
das andere (kürzere) Beinchen ist auf Minus bzw. Masse zu schalten

unbedingt die richtige Polung und auch die Pin-Auswahl beachten, ansonsten köönten die Bauelemente oder gar der Pi beschädigt werden

am Besten Aufbau auf einem SteckBrett

Spiel muss vor dem Starten des Python-Programmes laufen
ev. Warn-Hinweise wegen des Zugriffs auf die GPIO können ignoriert werden, Programm läuft schon im Hintergrund

Material eines Block's auswerten

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

import time

mc = minecraft.Minecraft.create()

rot = 0
gelb = 1
gruen = 2
Ampel = [18,23,25

GPIO.setmode(GPIO.BCM)
GPIO.setup(Ampel[rot], GPIO.OUT, ⮐
initial = True)
GPIO.setup(Ampel[gelb], GPIO.OUT, ⮐
initial = False)
GPIO.setup(Ampel[gruen], GPIO.OUT, ⮐
initial = False)

try:

  while True:

    p = mc.player.getTilePos()

    mat = mc.getBlock(p.x,p.y-1,p.z)

    if mat == 98:

      GPIO.output(Ampel[gelb], True)
      time.sleep(0.6)
      GPIO.output(Ampel[rot], False)
      GPIO.output(Ampel[gelb], False)
      GPIO.output(Ampel[gruen], True)
      time.sleep(2)
      GPIO.output(Ampel[gruen], False)
      GPIO.output(Ampel[gelb], True)
      time.sleep(0.6)
      GPIO.output(Ampel[gelb], False)
      GPIO.output(Ampel[rot], True)
      time.sleep(2)
except KeyboardInterrupt:


  GPIO.clearup()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Bibliothek mit Zeit- Funktionen einbinden
Erstellen eines Minecraft-Objektes mit dem Namen mc

Definition von Variablen (quasi als Konstanten)

Liste der für die Ampel (angeschlossene LED's benutzte Port's
Setzen der Pin-Namen lt. BCM
Initialisierung des Port lt. Ampel-Liste für Ausgabe
… für Gelb

… für Grün

nachfolgender Block wird ausprobiert (Abbruch folgt später!)
Endlos-Schleife (wird durch Spiel unterbrochen)
Holen der akt. Spieler-Position und Speichern in p
Material des Block's unter dem Spieler auslesen
Prüfen ob Material den Code 98 hat
Wenn ja, **dann** übliche Lichtschaltung einer Ampel
kurze Pause 0,6 s

lange Pause 2 s

Abbruch des probierten Block's durch eine Tatstatur-Unterbrechung (praktisch Tastendruck)
GPIO-Port-Belegung / -Benutzung löschen

⮐ Zeilenumbruch nur für die Darstellung des Quelltextes (Layout), bedeutet, dass in der Zeile weitergeschrieben wird

Schlag mit dem Schwert auswerten

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()
LED = [18,23,25]

GPIO.setmode(GPIO.BCM)
for i in LED:
  GPIO.setup(i,GPIO.OUT,initial=↵
  False)

try:

  while True:

    for hit in mc.events. ↵
    pollBlockHits():
      bl = mc.getBlockWithData(↵
      hit.pos.x, hit.pos.y, hit.pos.y)
      if bl.id == block. ↵
      GLOWSTONE_BLOCK.id:
        for i in LED:
          GPIO.output(i.True)
          time.sleep(0.05)
          GPIO.output(i.False)

except KeyboardInterrupt:
  GPIO.cleanup()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc

Setup der GPIO-Schnittstelle im BCM-Modus und auf Ausgaben; Benutzung der Port's ais der LED-Liste

nachfolgender Block wird ausprobiert (Abbruch folgt später!)
Endlos-Schleife (wird durch Spiel unterbrochen)
für alle Ereignisse (hits) aus der aktuellen Event-Liste dieses Block's
Erfassen der aktuellen Block-Daten

Wenn der akt. Block von dem angegebenen Material ist, **dann** …
lasse die die LED's hintereinander

kurz
aufblitzem

Abbruch durch eine Tastatur-Betätigung
dann noch GPIO nullen

auf externes Ereignis (z.B. Tasten-Druck) reagieren (und Blöcke erstellen)

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import mcpi.block as block
import RPi.GPIO as GPIO
import time

mc = minecraft.Minecraft.create()

t1 = 8
GPIO.setmode(GPIO.BCM)
GPIO.setup(t1,GPIO.IN, GPIO.PUD_DOWN)

try:
  while True:
    if GPIO.input(t1)==True:
      p = mc.player.getTilePos()
      mc.setBlocks(p.x-1, p.y,↵
 p.z-1, p.x+1, p.y, p.z+1,block.SAND)
      mc.player.setPos(p.x,p.y+1,p.z)

      time.sleep(0.2)
except KeyboardInterrupt:
  GPIO.cleanup()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft

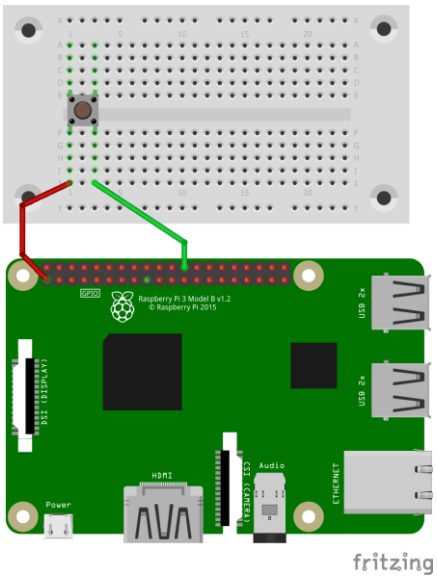
Bibliothek für die Ansteuerung der GPIO-Schnittstelle

Erstellen eines Minecraft-Objektes mit dem Namen mc
Festlegen des Port's für die Eingabe (Taste)
…
Einschalten des Pull-down-Widerstandes auf dem Rasp Pi
…

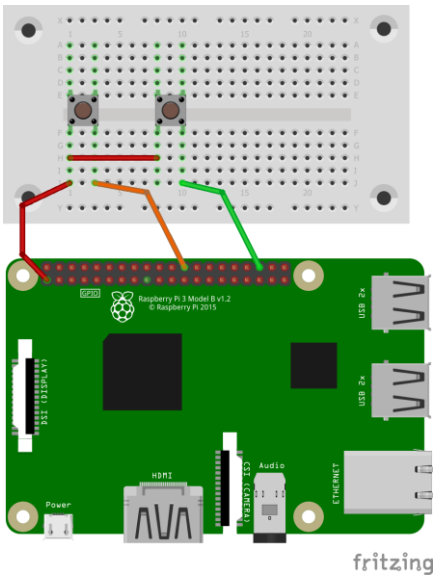
Wenn die Taste gedrückt ist, **dann** …

Erstellen von neuen Blöcken als 3x3-Fläche unter dem Spieler (Material ist Sand)
Korrektur der Spieler-Position auf der Sand-Fläche
…

ein Taster wird zwischen "3,3V" (innen Pin 1) und Port 8 (außen 12. Pin (Nr. 24)) geschaltet
s.a. Bau-Plan rechts



an einem weiteren Pin läßt sich ein weiterer Taster betreiben (siehe Aufbau-Plan unten)
diesem können wir unabhängig vom ersten Taster eigene Befehle zuordnen



Es werden die Port's 16 und 8 benutzt, die liegen auf den GPIO-Pin's 36 und 24. Die roten Draht-Brücken sind für den Masse-Kontakt beider Taster zuständig.

Im folgenden Programm soll 3 Blöcke vor der Spieler-Position ein Baum gebaut und mit der anderen Taste soll der Baum wieder entfernt werden. Dazu ändert man den Typ eines Block's einfach auf Luft (Code: 0).

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import mcpi.block as block
import RPi.GPIO as GPIO
import time

mc = minecraft.Minecraft.create()

t1 = 8
t2 = 16
GPIO.setmode(GPIO.BCM)
GPIO.setup(t1,GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(t2,GPIO.IN, GPIO.PUD_DOWN)

try:
  while True:
    if GPIO.input(t1)==True:
      p = mc.player.getTilePos()
      mc.setBlock(p.x+3, p.y, p.z,↵
 block.WOOD)
      mc.setBlocks(p.x+2, p.y+1,↵
 p.z-1, p.x+4,p.y+2, p.z+1,↵
 block.LEAVES)

    if GPIO.input(t2)==True:
      p = mc.player.getTilePos()
      mc.setBlock(p.x+3, p.y, p.z,↵
 block.AIR)
      mc.setBlocks(p.x+2, p.y+1,↵
 p.z-1, p.x+4,p.y+2, p.z+1, block.AIR)

      time.sleep(0.2)
except KeyboardInterrupt:
  GPIO.cleanup()
```

…

Festlegen des Port's für die Eingabe (Taste1) und für Taste 2
…
Einschalten der Pull-down-Widerstände auf dem Rasp Pi

…

Wenn die Taste1 gedrückt ist, **dann** …

Erstellen von neuen Blöcken als 3x3-Fläche unter dem Spieler (Material ist Sand)
Korrektur der Spieler-Position auf der Sand-Fläche
…

Material-Code	dt. Bezeich-nung	ID
AIR	Luft	0
STONE	Stein	1
GRASS	Gras	2
DIRT		3
COBBLESTONE		4
WOOD_PLANKS		5
SAPLING		6
BEDROCK		7
WATER_FLOWING		8
WATER		8
WATER_STATIONARY		9
LAVA_FLOWING		10
LAVA		10
LAVA_STATIONARY		11
SAND	Sand	12
GRAVEL		13
GOLD_ORE	Gold-Erz	14
IRON_ORE	Eisen-Erz	15
COAL_ORE		16
WOOD	Holz	17
LEAVES		18
GLASS		20
LAPIS_LAZULI_ORE		21
LAPIS_LAZULI_BLOCK		22
SANDSTONE	Sandstein	24
BED		26
COBWEB		30
GRASS_TALL		31
WOLL	Wolle	35
FLOWER_YELLOW		37
FLOWER_CYAN		38
MUSHROOM_BROWN		39
MUSHROOM_RED		40
GOLD_BLOCK		41
IRON_BLOCK		42
STONE_SLAB_DOUBLE		43

Material-Code	dt. Bezeich-nung	ID
STONE_SLAB		44
BRICK_BLOCK		45
TNT	Sprengstoff	46
BOOKSHELF		47
MOSS_STONE		48
OBSIDIAN	Obsidian	49
TORCH		50
FIRE	Feuer	51
STAIRS_WOOD		53
CHEST		54
DIAMAND_ORE	Diamand-Erz	56
DIAMAND_BLOCK		57
CRAFTING_TABLE		58
FARMLAND	Ackerland	60
FURNACE_INACTIVE		61
FURNACE_ACTIVE		62
DOOR_WOOD	Holztür	64
LADDER		65
STAIRS_COBBLESTONE		67
DOOR_IRON	Eisentür	71
REDSTONE_ORE		73
SNOW	Schnee	78
ICE	Eis	79
SNOW_BLOCK	Schnee-Block	80
CACTUS	Kaktus	81
CLAY		82
SUGAR_CANE		83
FENCE		85
GLOWSTONE_BLOCK		89
REDROCK_INVISIBLE		95
STONE_BRICK		98
GLASS_PANE		102
MELON	Melone	103
FENCE_GATE		107
GLOWING_OBSIDIAN		246
NETHER_REACTOR_CORE		247

Q: Begleitheft zu: conrad / Franzis Adventskalender Programmieren mit Minecraft 2018; erw. drews

eine LED dimmen (frequentes Signal ausgeben)

Problem: LED's lassen sich nicht wie Glühlampen dimmen. Bei diesen kann man durch eine größere oder kleiner Betriebs-Spannung die Leuchtstärke verändern. LED's brauchen einen minimale Spannung – quasi zum Zünden – und die Betriebsspannung kann auch nicht so einfach variiert werden, da es schnell zur Überlastung (Zerstörung) kommen kann.
Die analogen Port's des Rasp Pi (GPIO) sind nicht so auch nicht für eine Steuerung von LED's geeignet.
Der technische Ausweg ist ein Verändern von Leucht- und Dunkel-Phasen. Damit das ständige AN und AUS nicht als Blinken erkannt wird, benutzt man eine Arbeits-Frequenz von hier 50 Hz.

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()

LED = 25
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED,GPIO.OUT,initial=False)

pwm = 0
l = GPIO.PWM(LED,50)
l.start(pwm)

try:
  while True:
    p = mc.player.getTilePos()
    if p.z>=5 and p.z<=15:
      pwm = 10*(15-p.z)

    l.CangeDutyCycle(pwm)
    time.sleep(0.1)

except KeyboardInterrupt:
  l.stop()
  GPIO.cleanup()
```

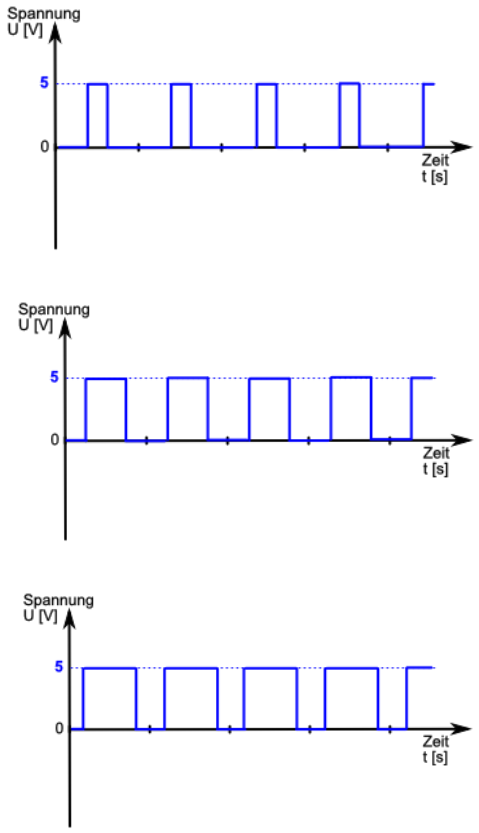
notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc
LED an Port 25
Initialiserung der GPIO

Tast-Verhältnis auf Null setzen ()
Erzeugen eines PWM-Objektes als l
Starten der Ausgabe

Wenn bestimmte Z-Position benutzt wird, dann wird Tast-Verhältnis schrittweise (in Abhängigkeit von der Z-Position) erhöht
Setzen des neuen Ausgabe-Signals
System-Wartezeit

…
Ausgabe am Port wird beendet

Ein Tast-Verhältnis von 0 bedeutet bei einem Rechteck-Signal einen minimalen AN-Teil. Praktisch ist der Pegel ständig auf 0 V gelegt.
Sind die AN-Phasen nur kurz (s.a. oberes Diagramm), dann blitzt die LED nur kurz auf. Da dies mit einer Freuenz von 50 Hz (so initialisiert im Programm) passiert, sehen wir das als sehr schwaches Leuchten (Licht-Summe).
Bei einem Tastverhältnis von 50 sind AN- und AUS-Teil jeweils 50% - also gleichlang (s.a. mittlere Abb.).
Physikalisch entspricht das der halben Lichttärke. Da unser optischer Sinn aber nicht linear funktioniert, erkennen wir das immer noch als relativ dunkel.
Je länger der AN-Teil wird, umso heller wird uns die LED erscheinen.
Setzt man den PWM-Wert auf 100, dann ist ein Dauer-AN – also ständig 5 V – am betreffenden Port anliegend. Damit ist die Leuchtstärke ausgeschöpft (zumindestens im regulären Bereich).

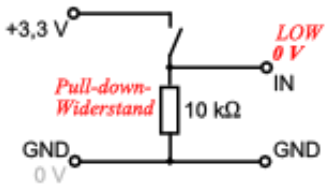


einen digitalen Pegel (/ Sensor-Kontakt) auswerten

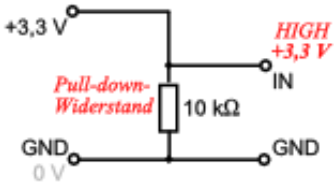
```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()

LED = 18
K1 = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED,GPIO.OUT,initial=False)
GPIO.setup(K1, GPIO.IN)

try:
  while True:
    if GPIO.input(K1) == False:
      GPIO.output(LED, True)
      mc.setBlocks(3,2,4,3,2,5,↵
 block.GOLD_ORE)
    else:
      GPIO.output(LED, False)
      mc.setBlocks(3,2,4,3,2,5,↵
 block.COAL_ORE)
    time.sleep(0.05)
except KeyboardInterrupt:
  GPIO-cleanup()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc

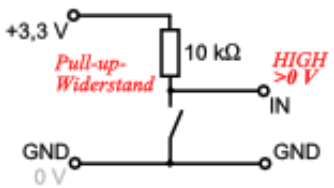
Die digitalen Eingänge an Schaltkreisen sind nicht eindeutig auf Null oder Eins gesetzt. Die interne Schaltung ermöglicht häufig stark schwankende Werte, die nicht eindeut auswertbar sind. Deshalb sorgt man mit einer einfachen Widerstands-Schaltung dafür, dass immer eindeutige Signale anliegen. Bei Schaltkreisen spricht man beim Null-Signal vom sogenannten LOW-Pegel, der üblicherweise einer Spannung von 0 Volt entspricht. Am Eingang (IN) kommt in der nebenstehenden Schaltung keine Spannung an. Der Widerstand sorgt dafür, dass irgendwelche anderen Spannungen in die Masse (GND, Erdung) abfließen können. Da der Widerstand das Signal auf LOW bzw. DOWN runterzieht, spricht man von einem Pull-down-Widerstand.



Wird nun der Schalter geschlossen, dann liegt die volle Spannung (hier +3,3 V) am eingang (IN) an. Diese Spannung wird eindeutig als HIGH bzw. Eins interpretiert und die nachfolgenden (internen) Schaltungen reagieren entsprechend.

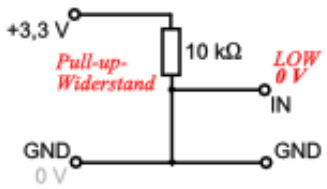


Das Schatungs-Prinzip lässt sich auch umdrehen.
Hier nutzt man zum Erzeugen eines HIGH-Signals einen sogenannten Pull-up-Widerstand. Dieser reduziert zwar die Betriebsspannung (hier: +3,3 V) auf einen deutlich niedrigeren Wert, aber dieser ist immer größer als Null. Er zieht in quasi über 0 V hoch. Somit interpretiert der Schaltkreis ein HIGH-Signal. Interpretieren heißt hier, dass die internen logischen Schaltkreise umschalten.



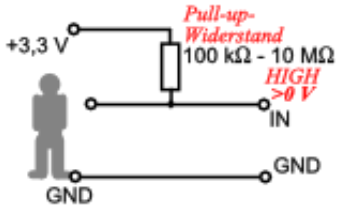
Drückt man nun den Taster, dann schließt man den Eingang (IN) mit der Masse (GND) kurz. Es liegt keine Spannung mehr am Eingang an und dieses bedeutet in der Schaltungs-technik eben ein LOW-Signal.

Die Widerstands-Werte können relativ weit gefächert sein. Je nach gewünschter Empfindlichkeit und Schaltsicherheit können Widerstände von mehr als 1 kW bis hoch zu einigen MW verwendet werden. Für praktische Schaltungen kann man die optimalen Werte mit Drehwiderständen ermitteln und dann gegen einen passenden einfachen Widerstand ersetzen.



Für eine konkrete "Sensor"-Schaltung nutzen wir die folgende Pull-up-Schaltung. Dabei wird kein Taster oder Schalter genutzt, sondern unser normaler Haut- und Körper-Widerstand. Wir werden somit zum elektrischen Leiter und der minimale Strom, der nun durch uns fließt, kann eine elektronische Schaltung beeinflussungen.

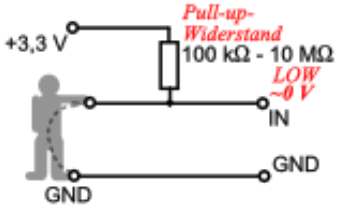
Der etwas größer gewählte Widerstand wird zwischen Strom-Versorgung (hier: +3,3 V) und dem Eingang (IN) des Schaltkreises eingebaut. Es fließt ein kleiner Strom, der für einen HIGH-Pegel am Eingang sorgt.

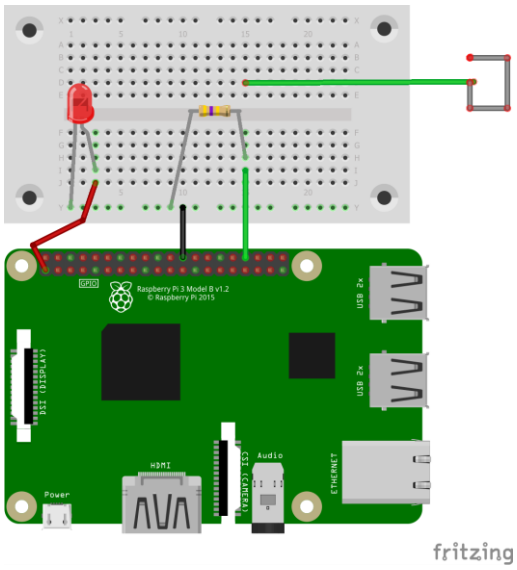


Durch die Berührung des IN-Kontaktes kommt es quasi zu einem "Kurzschluß" über den menschlichen Körper und seiner Haut. Der Pegel wird gegen 0 V gezogen, wass der Schaltkreis als LOW-Pegel interpretiert.

Natürlich ist es kein echter Kurzschluß. Unsere Haut und unser Körper haben einen deutlich messbaren elektrischen Widerstand. Feuchte Finger verringern den Widerstand weiter.

Weil dementsprechend sehr trockene Haut einen sehr großen Widerstand hat, kann es sein, dass die Schaltung nicht auslöst. Dann hilft z.B. das Anfeuchten der Hand oder der Finger.





eine RGB-LED ansteuern (beliebige Farbe ausgeben)

Bei LED's haben wir schon verschiedene Farben gesehen. Die Farbauswahl ist aber durch die verfügbaren Halbleiter beschränkt.

Aber auch hier können wir wieder unser Auge austricksen. Bringt man jeweils eine LED der drei Grundfarben in einem Gehäuse unter, dann kann man durch die unterschiedliche Ansteuerung der Einzelfarben eine optische Farb-Mischung erreichen. Statt der Farben Rot-Gelb-Blau wird aus historischen Gründen das Farbsystem RGB benutzt. Hier sind die drei Grund-Farben **R**ot-**G**rün-**B**lau. Auch aus diesen drei Farben lassen sich alle Farben – einschließlich weiß – mischen.

Ältere RGB-LED's hatten keine integrierten Vorwiderstände. In den modernen RGB-LED's sind passende Vorwiderstände eingebaut, so dass eine optimale Farbmischung möglich ist. Für "weiß" müssen ja alle drei Grundfarben gleichstark leuchten. Wir gehen hier vereinfacht von drei gleich großen Widerständen aus. Sie betragen 220 Ω. Bei älteren RGB-LED's schalten wir einfach die Vorwiderstände auf dem Steckbrett zwischen Plus-Pol (digitaler Ausgang) und der RGB-LED.

Böse Frage zwischendurch:

von Max Neugierig: "Kann man nicht nur einen einzelnen Vorwiderstand auf die Masse-Seite verwenden? Das würde doch immer zwei Widerstände einsparen, oder?"

Zuerst wollen wir nur die drei Grund-Farben nutzen – also die Einzel-LED's an- und ausschalten. Dadurch sehen wir die Elementar-Farben Rot-Grün-Blau.

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc

Im nächsten Schritt sollen entweder zwei oder alle drei Einzel-LED's gemeinsam angesteuert werden. Dadurch erreichen wir erste Farbmischungen. Die RBG-LED leutet dann z.B. violett, wenn die rote und die blau Elementar-LED angesteuert wird. Wenn man genau hinsieht, erkennt man noch die Einzel-Komponenten. Aus einiger Entfernung ist nur noch die Mischfarbe zu erkennen.

Richt vielseitig wird unsere RGB-LED, wenn wir die einzelnen Farben "dimmen". Wir können dadurch jede beliebige Farbe mischen.
Nun wollen wir einen schleichenden Farbverlauf programmieren.

eine LED dimmen (frequentes Signal ausgeben)

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc

eine LED dimmen (frequentes Signal ausgeben)

```
#!/usr/bin/python

import mcpi.minecraft as minecraft
import RPi.GPIO as GPIO

mc = minecraft.Minecraft.create()
```

notw. Zeile für Kommandozeilenstart von Python-Programmen
Bibliothek für die Verbindung mit Minecraft
Bibliothek für die Ansteuerung der GPIO-Schnittstelle
Erstellen eines Minecraft-Objektes mit dem Namen mc

10.2.2. Aduino und Verwandte

Die Grund-Konzeption der Arduino's ist eine andere, als die bei den Raspberry Pi's. Die Arduino's sind kleine Board's, die sich als Physical-Computing-Plattform verstehen. Dabei sollen sie bestimmte Steuerungs-Aufgaben übernehmen.
Die Programmierung erfolgt ursprünglich über eine C-ähnliche Sprache. Die Programme nennen sich Sketche und können die Arduino's zu beeindruckenden Leistungen bringen.
Seit ein paar Jahren ist auch eine Programmierung mit Python möglich. Dazu müssen aber diverse Vorarbeiten erledigt werden, die wir später vorstellen.

10.2.2.0. Kurzbeschreibung und allgemeine Einführung zu Arduino

sind programmierbare Kleinst-Rechner
Programme werden auf PC erstellt und dann auf den Arduino übertragen
dort laufen sie dann eigenständig (unabhängig vom Programmier-PC)

derzeit sind "Arduino Uno"-Clone für deutlich unter 10 Euro zu haben
ähnlich ist es bei dem kleineren Arduino Micro, diese werden in speziellen Versionen – oft als IoT-Board's – mit WLAN- oder Bluetooth-Schnittstelle angeboten, da liegt der Preis dann aber auch zwischen 20 und 30 Euro
einige – nicht ganz 100%ig kompartible – Uno-Platinen sind sogar unter 5 Euro aus China beziehbar, sie benötigen einen extra Treiber und sind dann kompartibel.
Interssant sind Boxen, die neben der Platine meist Unmengen von elektronischen Bauelementen und Kabeln enthalten. Auch hier sind die Preise sehr interessant und man bekommt ein Bastel-Set zwischen 8 und 50 Euro. Bei den größten Set's erschlagen einen die Möglichkeiten. Da besteht eher die Sorge, dass man da garnicht alles probieren kann. Vielfach fehlen auch Dokumentationen, die den gesammten Bauteile-Bestand erfassen.

10.2.2.1. Einrichtung einer Umgebung für Programmierung eines Arduino mit Python

es gibt verschiedene Herangehensweisen
Bei der Ersten müssen Programmier-Rechner und Arduino immer verbunden sein und die Arduino's können auch nicht selbstständig (weiter-)arbeiten. Das Zauberwort heißt hier pyFirmata (→ Variante mit pyFirmata:).
Nach einem ähnlichen Prinzip funktioniert pySerial (→). Diese Variante wird dann nachfolgend erläutert (→ Variante mit pySerial:). Man kann sich frei zwischen pySerial und pyFirmata entscheiden. Beide Varianten müssen nicht parallel auf dem Rechner installiert werden.
Die zweite Herangehensweise setzt auf eine interne Umsetzung des Python-Programm's in einen C-Code, welcher dann compiliert auf dem Arduino (auch selbstständig) laufen kann. Das Arbeits-Prinzip nennt man Cross-Compiling. Der Arduino kann dann auch im standalone-Betrieb – also ohne den Programmier-Rechner - arbeiten. Dazu weiter hinten Genaueres (→ Variante mit Shed Skin:).

Variante mit pyFirmata:

direkte Interpretation eines Python-Programm's auf dem Arduino ist nicht möglich, da auf dem Arduino solch komplexe Software, wie ein Interpreter, nicht laufen kann.
Man kann aber einen Arduino über die USB-Verbingung steuern.

Notwendig ist die sogenannte pyFirmata-Stelle für Python, also eine Erweiterung (ein Modul). Die holen wir uns von github.com (→).

Auf dem Computer, mit dem programmiert werden soll, muss zuerst die Arduino-IDE istalliert werden. Die gibt es auf arduino.cc zum Downloaden.

Nun muss auch das Python-System für die Zusammenarbeit mit dem Arduino konfiguriert werden. Das geht z.B. mit:

pip install pyfirmata

___
in der Arduino-Programmier-Umgebung den passenden Arduino am richtigen Port einstellen

"File" "Examples" "Firmata" ("Datei" "Beispiele" "Firmata") den Sketch "StandardFirmata" herunterladen und auf den Arduino hochladen

___

Die Python-Programm müssen dann immer das pyFirmata-Modul importieren:

```
from pyfirmata import Arduino, util
import time
```

Als nächstes setzen wir die Kommunikation auf die zugewiesene USB-Schnittstelle:

```
board = Arduino("COM3")
```

und schon kann das übliche Programmieren beginnen.

Das klassische Start-Programm ist bei den Arduino's ein LED-Blink-Programm. Im einfachsten Fall nutzen wir die Board-interne LED am Pin 13.
Natürlich kann auch eine externe LED mit beliebiger Farbe angesteuert werden. Dazu muss dann eine LED mit einem Vorwiederstand () zwischen Pin X und Gnd (Pin) gesteckt werden. Am Besten sind Aufbauten mit sogenannten Bread-Board's geeignet. Die lassen sich schnell und sicher zusammenstecken.
Den Bauplan sehen wir nebenan.

Im Python-Editor erstellen wir nun den eigentlichen Arbeits-Teil unseres Programm's:

```
while True:
   board.digital[13].write(1)
   time sleep(0.3)
   board.digital[13].write(0)
   time sleep(0.3)
```

Aufgaben:

1. *Ändern Sie die Blink-Frequenz der LED! (Hell- und Dunkel-Phasen sollen aber unbedingt gleichlang sein!)*
2. *Nun soll die LED am Pin 14 angeschlossen werden! Welche Veränderungen müssen am Board und im programm vorgenommen werden?*
3. *Schreiben Sie ein Programm, dass unaufhörlich SOS blinkt!*
4. *Konzipieren und erstellen Sie nun ein Programm, dass einen beliebigen einzugebenen Text per LED als MORSE-Zeichen sendet!*

Variante mit pySerial:

installieren mit:

```
pip install pyserial
```

Programmieren:

```
import serial  # ist pySerial

board = serial.Serial('/dev/cu.usbmodem143311', 9600,timeout=2) # bzw. Port unter Win

while True:
   board.write('Testtext')
   antwort = board.readline()
   print("Antwort des Board's: ", antwort)



print("Verbindung zum Board mit beliebiger Taste unterbrechen!")
…
try:
   while True:
      …
      board.digital[13].write(1)
      …
      board.digital[13].write(0)
except KeyboardInterrupt:
   board.exit()
   print("Verbindung zum Board ist unterbrochen."
...
```

```
import serial

verbindung = serial.Serial('/dev/tty.usbserial', 9600)

while True:
    print verbindung.readline()
```

Variante mit Shed Skin:

Cross-Compiler
übersetzt Python-Programm in ein C/C++-Programm
derzeit wird nur ein begrenter Sprach-Umfang von Python unterstützt

→ http://shedskin.github.io/ (Download, …)
→ https://en.wikipedia.org/wiki/Shed_Skin (engl. Wikipedia-Artikel)
→ https://shedskin.readthedocs.io/en/latest/ (Dokumentation)

10.2.2.x. Spezialfall UDOO

Kombination aus Raspberry-Pi-ähnlichem Grundsystem mit Arduino-Hardware

sehr Leistungs-fähig, aber weniger bekannt

Kleinserien-Produktion über Projekt (Finanzierung über ??? Founding (Kickstarter.com))

10.2.3. FRANZIS – Experimentierplatine mit FT232R

USB/Seriell-Wandler
praktisch Schnittstelle zwischen Betriebssystem / Anwender-Programmen und Experimentier-Hardware

leider nur unter Linux verwendbar und somit auch noch ein Rechner notwendig
(Nutzung über virtuelle Maschine noch nicht geprüft)

Kombination mit graphischer Oberfläche Gtk
Fenster-Manager Gnome
Gtk selbst ist auch für Windows verfügbar

erhältlich bei FRANZIS
Preis 99 Euro (etwas teuer)
im Schnäppchen-Angebot für 49 Euro auch immer noch recht teuer
dafür bekommt man schon einen vollwertigen Raspberry Pi mit Zubehör (mSD-Karte, Bauelemente, …)

10.2.4. TI-Innovator

ab Oktober 2020
Programmierung über TI-Nspire

Befehle sind über die den Werkzeug-Button eingefügt. Dort sind Menü's mit den verfügbaren Python-Befehlen zusammengestellt.
Die Befehle können dann nach Art der Block-Programmierung zusammengestellt werden.

gute Hilfe sind die notwendigen Bibliotheks-Aufrufe (Importe), die als Befehl gleich immer oben in den betreffenden Menü's aufgezählt sind

education.ti.com/de

auf der /fr gibt es die TI-83
ev. auch amerikanische TI-Seite nutzen

https://education.ti.com/de/activities/ti-codes

ti-unterrichtsmaterialien.net

10.2.4.y. externe Hardware

RGB-Array

anzuschließen über 4 Drähte

```
from ti_hub import *
rgb=rgb_array()
rgb.set(position,rot,gruen,blau)
sleep(zeit)
rgb.all_off()
```

```
rgb.set_all(rot,gruen,blau)



from ti_hub import *
from random import *

while get_key()!="esc":
    position=randint
    rot=randint(0,255)
    gruen=randint(0,255)
    blau=randint(0,255)
    rgb.set(position,rot,gruen,blau)
    sleep(1)
rgb.all_off()



mit Liste:

liste=[]
liste=[0 for i in range(16)]   #Null-befüllte Liste

for pos in range(0,16):
    print(liste[i])
    rgb.set(pos,



gibt wert als Binär-Zahl auf dem RGB-Array aus
rgb.pattern(wert)



misst den (Gesamt-)Strom:
wert=rgb.measurement()
```

Aufgaben:

1.
2. **Lassen Sie die einzelnen LED's nach und nach in der gleichen Farbe leuchten! Dabei soll sich die Intensität immer leicht erhöhen.**
3. **Jede LED soll zufällig ausgewählt mit einer zufälligen Farbe belegt werden. Das Programm soll solange laufen, bis die ESC-Taste gedrückt wird.**
4. **Erstellen Sie ein Programm, dass immer eine einzelne LED für 1 Sekunde in einer frei gewählten Farbe leuchten lässt und dann durch die nächste abgelöst wird!**
5. **Lassen Sie einen wieder einen "Leuchtpunkt" wandern! Dieses Mal soll die zuletzt benutzte LED mit halber Stärke nachleuchten!**
6. **Realisieren Sie ein Programm, dass immer eine zufällig ausgewählte LED das gesamte Farbspektrum durchläuft!**

7. *Realisieren Sie ein kleines Spiel bei dem der menschliche Spieler und Ihr Programm jeweils eine Zahl zwischen 1 und 8 setzen! Die gewählten Zahlen werden als gegenläufige Leuchtpunkte in den beiden Reihen des RGB-Array dargestellt. Der menschliche Spieler hat eine blaue Reihe, Ihr Programm eine gelbe. Wenn sich die Punktreihen überschneiden erhält der Spieler mit der größten Zahl einen Minus-Punkt, ansonsten bekommt derjenige einen Punkt, der die längste Reihe hatte. Bei Gleichstand erhält jeder einen Punkt / Minus-Punkt! Die Plus-Punkte werden als grüne Punkte und die Minus-Punkte als rote Leucht-Punkte angezeigt. Gewonnen hat derjenige, der zuerst 8 Gewinn-Punkte hat. Ihr Programm darf die aktuelle Eingabe des menschlichen Spielers nicht für die aktuelle Entscheidung benutzen, darf sich aber alle alten Eingaben merken und für eine Strategie auswerten.*
   (Realisieren Sie das Programm ev. in zwei groben Schritten: Zuerst nur die Anzeige der Spiel-Züge. Die Ergebnisse können dann auf der Shell ausgegeben werden. Im zweiten Schritt kann dann die Spielstand-Anzeige erfolgen.)
   Zusatz: Lassen Sie sich eine passende Gewinn-Anzeige einfallen (Leucht-Show)!

10.3. Datenbank-Zugriff mit Python

→ https://www.hdm-stuttgart.de/~maucher/Python/html/SQLite.html#connection-und-cursorobjekt-erzeugen

10.3.1. SQLite 3

from sqlite import dbapi2 as sqlt

10.3.1.0. Verbindung herstellen

außer der Verbindung brauchen wir noch ein Cursor-Objekt
es stellt quasi die imaginäre Shell (Konsole, Benutzeroberfläche) dar, es ist im Programm so, als würden wir auf die Shell Anweisungen etc. in SQL schreiben

verb = sqlt.connect(Datenbankname) # Datewnbankname ist Unterverzeichnis + Datenbank

eine temporäre, lokale (nicht-persistierende) Datenbank lässt sich mit:

verb = sqlt.connect(:memory:)
curs = verb.cursor()

erzeugen. Die zweite Anweisung erzeugt einen SQL-Cursor.

10.3.1.1. Erstellen einer Tabelle

curs.execute("'create table if not exists schueler (name text, vorname text, gebdatum date, masse real, groesse integer)'")

Datentyp in Python	Datentyp in SQLite	
None	NULL	
int	INTEGER	
float	REAL	
str (UTF8)	TEXT	
unicode	TEXT	
buffer	BLOB	

10.3.1.2. Hinzufügen von Datensätzen zu einer Tabelle

```
curs.execute('''insert into schueler values ('Mustermann','Klaus','01.01.2000',63.7,177)''')
…
verb.commit()    # eigentliche Speicherung der vorher angegebenen Datensätze
```

10.3.1.3. Aktualisieren eines Datensatzes in einer Tabelle

```
geschlechterListe = ["m","w", …]   # soviele Listen-Einträge, wie Datensätze in Tabelle
curs.execute('''alter table schueler add column geschlecht text''')
for geschl in enumerate(geschlechterListe):
   curs.execute("update schueler set geschlecht=(?) where owid=(?)",
                [geschlechterListe[geschl],geschl+1])
verb.commit()    # eigentliche Aktualisierung der vorher spezifizierten Werte in der
                   erweiterten Tabelle
```

10.3.1.4. Löschen eines Datensatzes aus einer Tabelle

```
curs.execute('''delete from schueler where vorname="Klaus" ''')
…
verb.commit()    # eigentliche Löschung der vorher spezifizierten Datensätze
```

10.3.1.5. Löschen einer Tabelle

```
curs.execute('''drop table if exists schueler''')
```

10.3.1.z. Beenden der Verbindung

weitere Beispiele:

```
import sqlite3

conn = sqlite3.connect('daten/Kontakte.dat')
curs = conn.cursor()
curs.execute("CREATE TABLE personen(PID, Vorname, Name, eMail)")

DatenListe=[(1,"Monika","Musterfrau","musterfrau@webb.de"),
            (2,"Klaus","Mustermann","muma@tee-online.de"),
            (3,"Prof. Lisa","Klug","Prof.L.Klug@uni-mustern.de)]

for Elem in DatenListe:
    curs.execute("Insert INTO personen VALUES (?,?,?,?)", Eleme)

…

curs.close()
conn.close()
```

curs.fetchone() liefert eine Ergebnis-Zeile zur Anfrage als Tupel
curs.fetchmany(n) liefert n Ergebniszeilen zur Anfrage als n-Tupel von Tupeln
curs.fetchall() liefert alle Ergebniszeilen zur Anfrage als Tupel von Tupeln

10.4. Web-Server-Anwendungen mit dem (Micro-)Framework Flask

unterstützt Generierung von Seiten auf einem Web-Server

allgemeines Handling:
auf einem Intranet- oder Internet-Server läuft ein Webserver (Dienst der die Bereitstellung von http-Seiten realisiert)
auf einem Client läuft ein Browser (Betrachter-Programm für http-Seiten)
nach Aufruf der Server-Adresse z.B.: www.lern-soft-projekt.de oder 127.0.0.1 (localhost; Webserver läuft auf dem eigenen Rechner) wird eine Seite (normal index.htm od.ä.) angefordert (get-Methode) (ev. wird mit Fehler-Meldung geantwortet
Webserver schickt HTML-Code der angeforderten http-Seite an den Browser
Browser setzt den HTML-Code in eine anzeigbare Seite um oder gibt Fehler-Meldungen aus
Client / Browser können nach Interaktionen neue Inhalte (nach-)laden / liefern / anzeigen

10.4.0. Erzeugung einer Web-Seite mit Python (Wiederholung)

10.4.1. das Framework Flask

dient hauptsächlich der Trennung von Daten und Darstellung
der Programmierer soll sich nicht mehr vorrangig um die Darstellung seiner Daten kümmern, das übernimmt im Wesentlichen das Framework
der Programmierer stellt seine Daten bereit und nutzt vorgefertigte Methoden / Funktionen usw. um die Darstellung fast von alleine dem Framework zu überlassen

bringt u.a. das Kachel-Design mit
sehr gut für Anzeigen etc. von IoT-Daten oder Info-Daten geeignet

ev. muss vorher ein Web-Server installiert und / oder eingerichtet werden

minimales Hello-Welt-Programm mit Flask

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def index():
    return "Hallo Welt!"
```

der große Vorteil von Flask (wie auch von anderen Frameworks), dass diese mit Templeates (Schablonen) a´rbeiten können. Die Templates enthalten bestimmte Stellen, an denen dann durch einfache Funktionen / Methoden die Inhalte eingefügt werden.
So ist z.B. in einem Template festgelegt, dass eine Überschrift mit einem Rahmen versehen sein soll, größer, unterstrichen und fett dargestellt werden soll. Auch Farben usw. lassen sich definieren.
Der Programmierer ruft jetzt nur eine Funktion auf, wie z.B. generiereÜberschrift() und übergibt dieser nur den Text. Die ganze Einstell-Arbeit übernimmt das Framework auf der Basis der bereitgestellten Schablone. Ist für eine – quasi parallel laufende – "andere" Webseite ein anderes Template festgelegt, dann sieht die gleiche Überschrift dort eben anders aus. Darum braucht sich der Programmierer nicht zu kümmern. Das Layout gestaltet ein Designer.
Dieses Prinzip kennt man von vielen Webseiten / Social-Media-Seiten, wo man seine individuellen Farb-Einstellungen etc. vornehmen kann.

```
from flask import Flask                                  Import des Flask-Frameworks
app = Flask(__name__)                                    Flask-Applikation def.
@app.route('/')                                          Funktionsdekorator
@app.route('/reserve')                                   alternativer Dekorator

def index()                                              beliebiger Funktionsname
    return render_template("template.html,               Darstellungs-Aufruf unter Nut-
                  var1="Hallo", var2="Welt")             zung eines Templates und zu-
                                                         sätzlichen Daten
```

Dieser Python-Code und Flask benötigen aber dazu das passende Template:

```
<!doctype html>
<html>
  <head>
      <title>
        Seite: Hallo-Welt
      </title>
  </head>
  <body>
    <h1>Überschrift der Hello-Welt-Seite</h1>
    übergebene Daten sind: {{var1}} und {{var2}}
    zusammen: {{var1}} {{var2}}!
  </body>
</html>
```

Achtung! Im Gegensatz zu Python sind bei HTML die Einrückungen nur Mittel zur übersichtlicheren Darstellung. Sie können vollständig weggelassen werden und sogar alles fortlaufend in eine Zeile geschreiben werden.

Mit weiteren Web-Techniken können weitere Verbesserungen / Funktions-Erweiterungen etc. erreicht werden. So kann man mit CSS (Cascade Style Sheets) Format-Vorlagen und andere Layout-Parameter (z.B. für ein Corpurate Design) definieren. Diese werden dann im HTML-Code zugewiesen.

```
…
  <head>
      <link rel="stylesheet" href=stylsheet.css">
  </head>
…
```

JavaScript eigente sich z.B. um Anpassungen der Seite vornehmen zu lassen oder Berechnungen durchzuführen.
Die Java-Scripte werden üblicherweise am Ende des Body-Bereiches angegeben.

```
…
  <body>
    …
    <script scr="funktion.js">
  </body>
…
```

Das folgende Python-Programm ist ein schönes – aber auch schon recht komplexes - Beispiel für eine Flask-Anwendung. Für die Nutzung der oben erwähnten Adafruit-Experimentier-Platine oder eines ähnlichen IoT-Systems soll eine Kachel-Oberfläche dienen. Die einzelnen Kacheln diesen entweder der Anzeige von Meßwerten, dem Ein- bzw. Ausschalten oder dem Wechsel zu Unterseiten usw.
Die Datei ist auf github.com gehostet und kann über:

git clone https://github.com/openHPI/Embedded-Smart-Home-2017.git

in das aktuelle Verzeichnis kopiert werden. Dort ist die nachfolgend angezeigte smarthome.py enthalten. Sie dient als Steuerzentrale des gesamten Projektes.

```
from flask import Flask, render_template, request
from flask_bootstrap import Bootstrap

from tiles import SimpleTile, TileManager
from helper import PageContext

app = Flask(__name__)
Bootstrap(app)

@app.route('/')
def main():
    tiles = [
        SimpleTile("Licht", "#EEEE00", "light/"),
        SimpleTile("Heizung", "#FF0000", "heaters/"),
        SimpleTile("Sicherheit", "#30FF00", "security/"),
        SimpleTile("Wasser", "#0000FF", "water/"),
        SimpleTile("Extrapunkt 1", "#00FFFF", "/"),
        SimpleTile("Extrapunkt 2", "#FF00FF", "/"),
        SimpleTile("Extrapunkt 3", "#A0FFA0", "/"),
        SimpleTile("Extrapunkt 4", "#00A0FF", "/"),
    ]

    manager = TileManager(tiles)
    context = PageContext("Smarthome Projekt", "Home")
    return render_template("main.html", tilerows=manager,
context=context)

@app.route('/light/')
def light():
    living_room = True
    sleeping_room = False

    if("living_room" in request.args):
        living_room = True if request.args["living_room"] ↵
        == "on" else False

    if("sleeping_room" in request.args):
```

```
        sleeping_room = True if request.args["sleeping_room"]↵
        == "on" else False

    tiles=[]

    tile = SimpleTile("Wohnzimmer: ", "", "?living_room=")
    tile.items[0].text += "an" if living_room else "aus"
    tile.link += "off" if living_room else "on"
    tile.bg = "#AAFF00" if living_room else "#338800"
    tiles.append(tile)

    tile = SimpleTile("Schlafzimmer: ", "", "?sleeping_room=")
    tile.items[0].text += "an" if sleeping_room else "aus"
    tile.link += "off" if sleeping_room else "on"
    tile.bg = "#6666BB" if sleeping_room else "#333388"
    tiles.append(tile)

    manager = TileManager(tiles)
    context = PageContext("Smarthome Projekt", "Licht",↵
              [["/", "Home"]])
    return render_template("main.html", tilerows=manager,↵
                           context=context)

if   name   == "  main  ":
    app.run(debug=True)
```

10.4.2. die Flask-Erweiterung bootstrap

10.4.3. Programmierung der Web-Oberfläche und Darstellung von Meßwerten

```
from flask import Flask, render_template, request
from flask_bootstrap import Bootstrap

from tiles import SimpleTile, TileManager
from helper import PageContext

import sqlite3
from flask import g

import requests, json

app = Flask(__name__)
Bootstrap(app)
app.config['BOOTSTRAP_SERVE_LOCAL'] = True
```

```
DATABASE='database.sqlite'

def get_db():
    db = getattr(g,'_database', None)
    if db is None:
        db=g._datbase=sqlite3.connect(DATABASE)
    return db

def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (tv[0] if rv else None) if one else rv

@app.teardown_appcontext
def close_connecting(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()



@app.route('/')
def main():
    temp = query_db("SELECT wer, einheit FROM sensoren ↵
                    ORDER BY zeit DESC", one=True
    print(temp)    #Kontrollanzeige auf Konsole
    tiles = [
        SimpleTile("Licht", "#EEEE00", "light/"),
        SimpleTile("Heizung", "#FF0000", "heaters/"),
        SimpleTile("Sicherheit", "#30FF00", "security/"),
        SimpleTile("Wasser", "#0000FF", "water/"),
        SimpleTile("Innentemperatur: " + temp[0] + " " +↵
                    temp[1], "#FF0000", "/"),
        SimpleTile("Außentemperatur", "#00FF00", "/"),
        SimpleTile("Luftfeuchtigkeit", "#0000FF", "/"),
        SimpleTile("Helligkeit", "#FFFF00", "/"),
    ]

    manager = TileManager(tiles)
    context = PageContext("Smarthome Projekt", "Home")
    return render_template("main.html", tilerows=manager,↵
                           context=context)

@app.route('/light/')
def light():
    living_room = True
    sleeping_room = False

    if("living_room" in request.args):
        living_room = True if request.args["living_room"]↵
        == "on" else False

    if("sleeping_room" in request.args):
        sleeping_room = True if request.args["sleeping_room"]↵
         == "on" else False

    tiles=[]

    tile = SimpleTile("Wohnzimmer: ", "", "?living_room=")
    tile.items[0].text += "an" if living_room else "aus"
    tile.link += "off" if living_room else "on"
```

gibt Verbindung zur Datenbank zurück

Datenbank-Abfrage
one bestimmt, ob nur 1 Wert zurückgeliefert werden soll

```
    tile.bg = "#AAFF00" if living_room else "#338800"
    tiles.append(tile)

    tile = SimpleTile("Schlafzimmer: ", "", "?sleeping_room=")
    tile.items[0].text += "an" if sleeping_room else "aus"
    tile.link += "off" if sleeping_room else "on"
    tile.bg = "#6666BB" if sleeping_room else "#333388"
    tiles.append(tile)

    manager = TileManager(tiles)
    context = PageContext("Smarthome Projekt", "Licht",↵
                          [["/", "Home"]])
    return render_template("main.html", tilerows=manager, ↵
                           context=context)

if __name__ == "__main__":
    app.run(debug=True)
```

fehlt requests bzw. gibt es dahingehend Fehler-Meldungunegn, dann mus die Bibliothek nachinstalliert werden:

pip3 install requests

Um z.B. externe Wetter-Daten mit anzuzeigen braucht man eine Daten-Quelle für solche Informationen. Dazu ist es bei openweathermap.org sich einen Account-Schlüssel zu besorgen und damit dann rund 60 Wetter-Info-Pakete pro Stunde runterzuladen. Die Daten kommen als JSON-Datei und müssen mittels json-Bibliothek in ein JSON-Objekt umgewandelt werden. Dazu brauchen wir die importierte json-Bibliothek. Natürlich könnte man den Text auch per Hand selbst zerlegen (parsen). Das ist aber recht aufwendig in der Programmierung. Da nutzen wir lieber die vorgeferigten und geprüften Methoden / Funktionen von json.

```
def main():
    temp = query_db(…
    r = requests.get('http://api.openweathermap.org/data/↵
        2.5/weather?g=Rostock,de&appid=  eigene ID')
    weatherdata = json.loads(r.text)
    temp_out = round/weatherdate['main']['temp'] - 273.15,1)
    weathersymbol = '<img src="http://openweathermap.org/img↵
     /w' + weatherdata['weather'][0]['icon'] + ".png">'

    …
    SimpleTile("Außentemperatur: " + str(tempout + " C <br>"↵
     + weathersymbol, "#FF00FF", "/"),
    SimpleTiel(…
```

Den API-Key muss man sich bei openWeatherMap.org besorgen.
Es gibt auf openWeatheMap.org auch API's, die eine Abfrage von Wetter-Vorhersagen erlauben.

10.5. Web-Applikationen mit Django

recht freies, kompakter und beherrschbares Framework
legt Wert auf effektives Programmieren (alles möglichst nur einmal programmieren (Don't Repeat Yourself → DRY))
mit eigenem Web-Server zum schnellen Testen / Ausprobieren

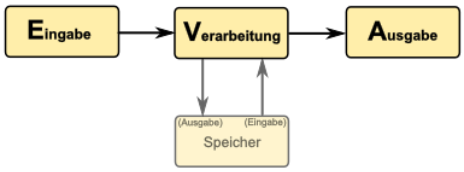
Nachteile / Probleme:
Struktur nicht selbsterklärend

Links:
http://www.django-workshop.de/ (gutes aufgebautes Tutorial (dt.)) → Beispiel: Rezept-Sammlung
[scheinbar ist die Version des Tutorials veraltet (0.4), einschließlich der verwendeten Programm-Versionen (Python 2?!, 3.3, Django 1.4); ab und zu ist nicht klar, was auf welcher Ebene gemacht werden muss; keine Fehler-Hinweise; es fehlt das Hintergrundwissen, um die Zusammenhänge zu verstehen; einige Texte wirken sehr abstrakt]

10.6. MicroPython für Microcontroller

u.a. Q:
/µP_Q1/ … Thomas WALDMANN (Vortrag: "Einführung in ESP32 Microcontroller + MicroPython"; EASTERHEGG 2018; https://media.ccc.de/v/V8W9DL)

Es muss nicht immer ein großer Rechner sein, um mit Python zu arbeiten. Wir haben ja schon gesehen, dass die kleinen Raspberry Pi's ebenfalls mit Python daherkommen und nicht wirklich Leistungs-schwächer sind. Natürlich muss man hier die allgemeine Leistungsfähigkeit des Grundsystem's beachten.
Es geht aber noch kleiner. Microcontroller sind minimalste Datenverarbeitungs-Systeme und zielen stark auf den IoT-Bereich ab.
Microcontroller – oft auch als Experimentier-Board's oder IoT-Bausteine bezeichnet - verfügen auf kleinsten Raum über alle EVAS-Teile eines Informatiksystems.
Besonders effektive Microprozessoren steuern unzählige Ein- und Ausgabe-Möglichkeiten.



Klassiker sind sicher die Arduino's. Sie waren und sind noch zu langsam und zu Speicherarm für Python.
Neuere Hardware ist da um Längen besser. Zu den neuen Sternen am Himmel zählen z.B. die ESP-Bausteine.
Fast allen Microcontrollern ist eine sehr offene Hard- und Software gemeinsam. Zwar ist nicht alles OpenSource oder völlig frei zugänglich, aber die offene Arbeitskultur von Hard- und Software-Herstellern erzeugt eine schnelle und breite Nutzung in allen Bereichen.

Vorteile:

- relativ einfache Programmierung (im Vergleich zum sonst üblichen C/C++)
- unendliches Laufen eines Programms auf minimalster Technik
- praktisch fast unbegrenztes direktes und indirektes Ansprechen von Sensoren und Aktoren
- langfristiger und relativ unabhängiger Betrieb über PowerBank-Stromversorgung möglich
- Unterstützung vieler Protokolle und der üblichen Hardware / Peripherie

Nachteile:

- etwas eingeschränkter Befehls-Umfang
- neue Bibliotheken / Module notwendig
- etwas umständliche Handhabung zwischen Entwicklung und Programm-Lauf
- Größe der Programme meist durch relativ kleine Speicher begrenzt
- auf MicroPython umgestellte Systeme müssen wieder speziell auf den üblichen Microcontroller-Betrieb (übliche C/C++-Programmierung od.ä.) umgestellt werden
-

fehlende Funktionen / Features (im Vergleich zum Standard-Python):

- keine Unterstützung von Unicode
- Leerzeichen zwischen Literalen (Zahlen) und Schlüsselwörtern notwendig
- geänderte Methoden-Auflösung bei geschachtelten Klassen
- nur eine Oberklasse festlegbar
- unterschiedliche Ausgabe-Formate für **float**-Zahlen
- für von **int** abgeleitete Typen ist kein Typ-Umwandlung möglich

- Slicing in Listen eingeschränkt
- **eval()** hat keinen Zugriff auf lokale Variablen
- in Generator-Funktionen wird **__exit__()** nicht aufgerufen
- Byte-Array's werden nicht unterstützt
- String-Methode **.endwith()**, **.ljust()** und **.rjust()** nicht implementiert
- **__del__** als spezielle Methode nicht implementiert
- **self** wird als ein Argument gezählt
- begrenzte Unterstützung von Namespace's
- Zeichenketten-Verarbeitung mit Schlüsselwörtern (z.B. **encoding**) nicht möglich
- lokale Variablen werden bei **locals()** nicht einbezogen
- Nutzer-definierte Attribute in Funktionen werde nicht unterstützt
- spezieller Umgang mit property-Getter
- die **__path__**-Eigenschaft von Modulen wird als relativer Pfad ausgegeben
- Verkettung von Exception's nicht implementiert
- die Methode **Exception.__init__** gibt es nicht
- keine Nutzer-definierten Attribute in Builtin-Exception's
- bei Fehler-Anzeigen in while-Schleifen werden Zeilennummern anders gezählt
- für Bytes-Objekte ist eine **.format()**-Methode verfügbar
- die Nutzung von **step != 1** in Byte-Objektensowie in Tuple und Listen nicht möglich
- Instanzen von **str**-Unterklassen können nicht mit **str**-Instanzen verglichen werden
-

fehlende / geänderte Funktionen / Features in Modulen:

- im **array**-Module
  - keine Suche nach Integer möglich
  - Löschen (**del()**) von Elementen nicht möglich
  - Nutzung von **step !=1** nicht möglich
- builtin's
- kein zweites Argument bei **next()** möglich
- im **collections**-Modul
  - **deque** nicht implementiert
- im **json**-Modul
  - nicht-serialisierte Einträge erzeugen keine Exception's
- im **struct**-Modul
  - zuviele Argumente in der **.pack()**-Methode werden nicht beachtet
- im **sys**-Modul
  - die Attribute **.stdin**, .**stdout** und **.stderr** lassen sich nicht überschreiben
-

Bedeutung des MicroPython in der Microcontroller-Welt:

- leichter zu programmieren als C/C++
- weite und immer weiter steigende Verbreitung von Microcontrollern
-

bekannte Forks zum MicroPython

- CircuitPython
- PyCom

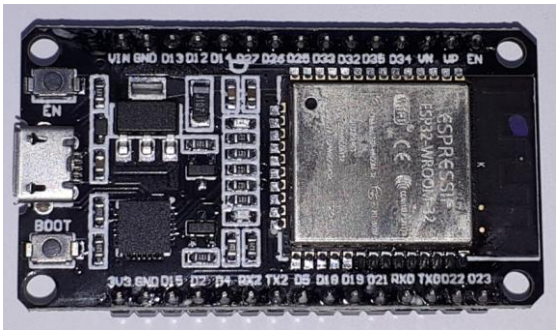
Warum funktioniert das Arbeiten mit Python auf einem Microcontroller, wenn sonst immer auf einem extra (Host-)Rechner editiert und kompiliert werden muss?

Normalerweise sind die Interpreter und Compiler moderner Programmiersprachen sehr große Programme. Der Speicher und meist auch die Leistung der CPU der Microcontroller reicht nicht für sie aus. Auch bei Python ist das so.
Bei MicroPython wird ein extra kleiner Interpreter mit eingeschränkter Leistung verwendet. Wir haben das oben schon thematisiert.
Zum anderen bedient man sich eines Trick's. Normalerweise werden ja immer die kompilierten Programme – also Binär-Dateien – auf den Microcontroller übertragen. Dies nennen wir flashen. Das (fest integrierte und unveränderliche) Boot-System und die variable Firmware des Microcontroiller's isind so ausgelegt, dass sie die gefundene Binär-Datei vom letzten Flashen startet und unermüdlich abarbeitet.
Beim Python wird der Mini-Interpreter geflasht und dazu ein klassisches Datei-System erzeugt. Das Boot-System des Microcontroller's startet also den aufgeflashten MicroPython-Interpreter und dieser kommuniziert zum Einen über die serielle Schnittstelle oder arbeitet ein gefundenes py-Programm ab.

10.6.x. MicroPython für microbit

lässt sich auch mit dem pyCraft-Tool (→) bedienen

10.6.x. MicroPython für ESP-32-Microcontroller



ESP-8266 und das Nachfolge-Modell ESP-32 sind deutlich leistungsfähiger als die sonst üblichen Arduino's
verschiedene Ausführungen und Hersteller

sie unterscheiden sich vorrangig in der Anzahl der herausgeführten – und damit nutzbaren – Pin's

auffallend ist die extrem funktionell breite Auslegung des Microcontroller
man kann fast schon sagen, alles was das IoT- und Bastler-Herz liebt und braucht ist sehr effektiv im ESP umgesetzt
praktisch billige Massen-Ware, je nach Ausstattung zwischen 6 und 30 Euro
bei den teureren Varianten sind dann oft schon Display's mit dabei
(i.A. insgesamt günstiger als Aduino-System (einschließlich der verschiedenen Billig-Clone)
besonders herausregend in dieser Preisklasse die breite Unterstützung von WLAN und Blue-tooth

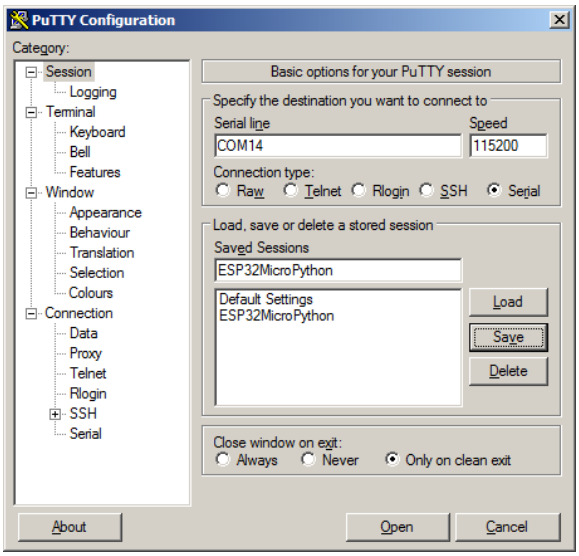
10.6.x.0. Vorbereiten des ESP für MicroPython

Download des Image von der MicroPython-Webseite (→ http://micropython.org/download)
weiterhin ein Fork unter (→) verfügbar (teilweise Leistungs-fähiger, aber eben Spezial-Lösung!)
nicht immer unbedingt das neueste Daily-Build verwenden, da hier schnell Bug's drin sein können, dafür sind aber alte Bug's im Allgemeinen bereinigt

zuerst die alte Firmware auf dem ESP löschen
esptool.py –chip esp32 –port /dev/ttyUSB0 erase_flash

neue Firmware (Minimal-OS + MicroPython) hochladen
esptool.py –chip esp32 –port /dev/ttyUSB0 write_flash –z 0x1000 esp32_firmware.bin

Verbindung zum ESP-32-MicroPython über ein Konsolen-Programm
hier PuTTY

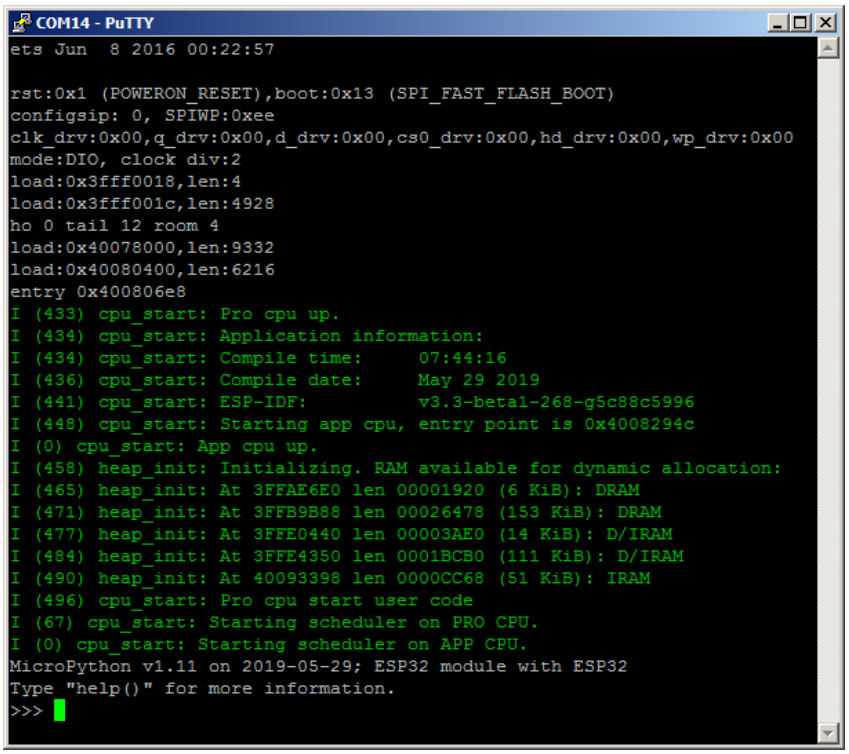


wir brauchen eine serielle Kommunikation (also: Serial) mit den Parametern:
Serial line: COM14 (USB-Port)
Speed: 115200 (übliche Baud-Rate für die Kommunikation mit dem ESP-32

ich habe mir die Parameter gleich unter einem passenden Namen abgespeichert

kommuniziert über die USB-serielle Schnittstelle mit dem MicroPython

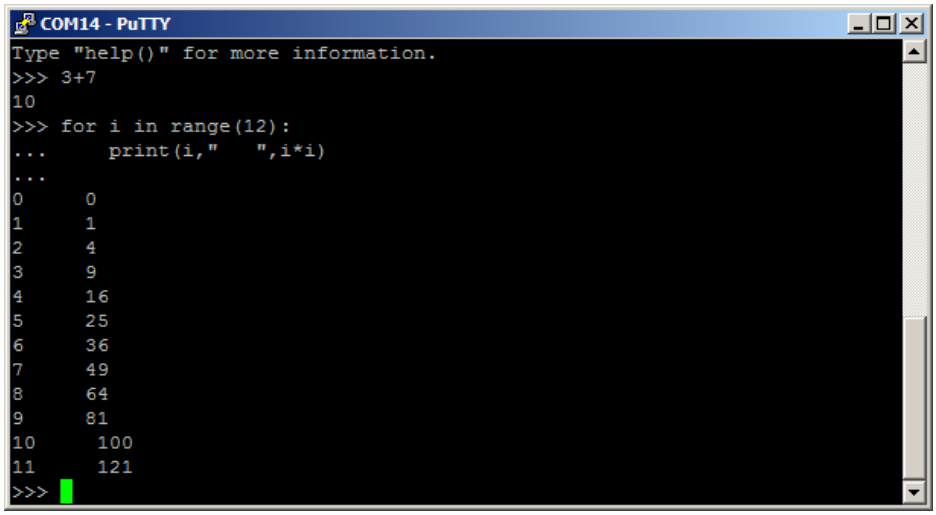
zuerst bekommt man eine Status-Information zum MicroPython und einigen Ressourcen angezeigt



den seriellen Monitor kann man jetzt auch weiter auflassen und mit den MicroPython kommminizieren

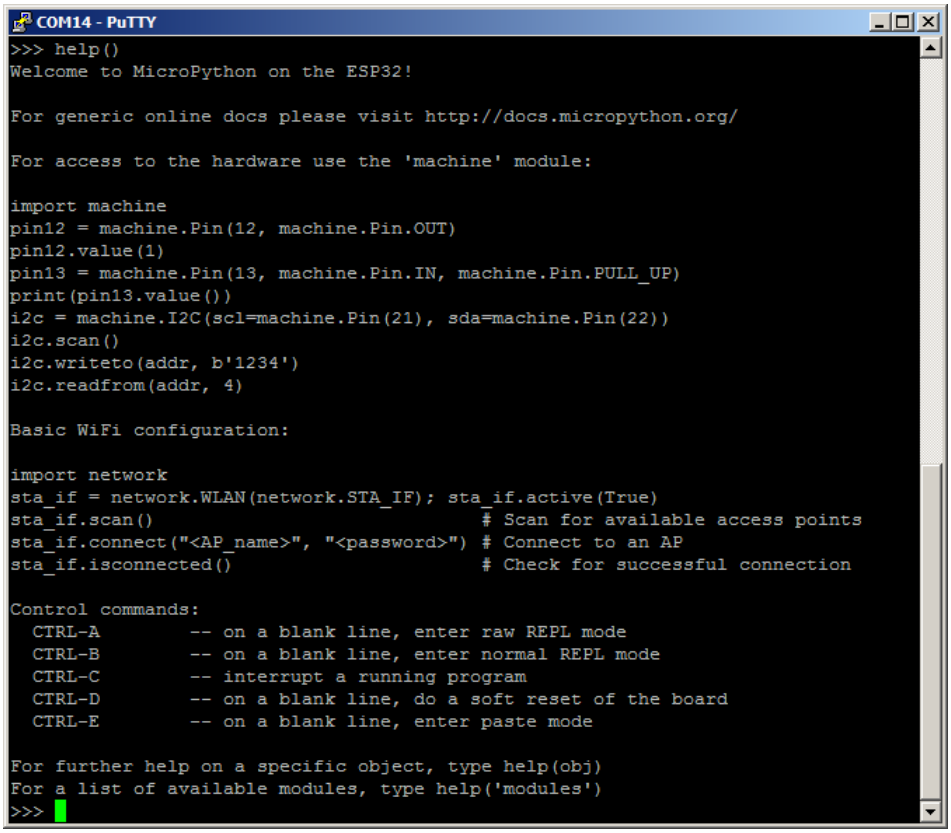
im Prinzip sind wir jetzt im interaktiven Modus, so wie wir ihn ja schon vom großen Python mit IDLE kennen

hier zwei kurze und einfache Beispiele

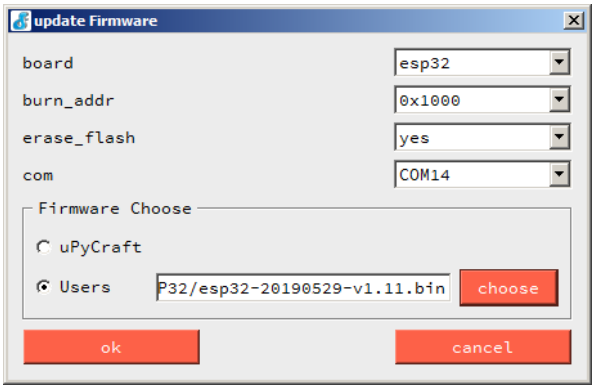


dieser Modus nennt sich REPL (Read-Evalute-Print-Loop)

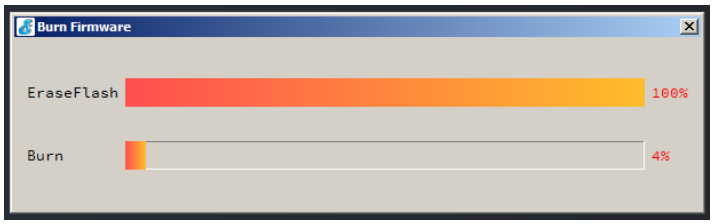
mit help() kann man sich die elementaren Hilfetexte für das MicroPython ansehen



10.6.x.0.1. das Tool uPyCraft



Alternativ – und insgesamt deutlich komfortabler – lässt sich die MicroPython-Firmware auch mit dem nachfolgend besprochenem Programm (uPyCraft) erledigen.
Falls das Programm noch nicht installiert ist bzw. noch unbekannt ist, dann bitte zuerst weiter hinten lesen (→ Installation und Beschreibung des Hilfs-Programms uPyCraft).
Dies bringt auch ein eigenes Image mit. Ich bleibe hier bei dem offiziellen von der micropython.org-Webseite.
Die Lade-Adresse (burn_addr) ist offiziell die 0x1000. In einigen Anleitungen zu uPyCraft wird dagegen empfohlen, dem Programm die Entscheidung zu überlassen.
(Dann kann es sein, dass als Adresse die 0x0 angegeben ist. Bei mir klappte es mit beiden Adress-Angaben.)
Nach der Bestätigung beginnt das Löschen des Flash-Speichers auf dem ESP. Vorhandene Dateien gehen verloren.



Auf das Löschen folgt das Schreiben (Burn, Brennen) des MicroPython in den Speicher.
Diese Alternative zum oben beschrieben Flashen ist vor allem dann interessant, wenn das uPyCraft dann zur Verfügung steht und man mal wieder ein neues MicroPython aufsetzen muss.

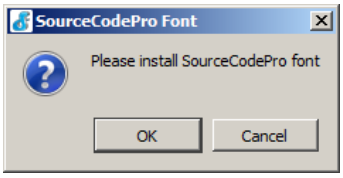
notwendige Treiber
https://github.com/Tasm-Devil/Micropython-Tutorial-for-esp32/archive/master.zip
downloaden und entpacken
so in das Dateisystem kopieren, dass der workspace-Ordner als Unterordner im Programm-Ordner der uPyCraft.EXE liegt
alternativ den workspace-Ordner anders einstellen

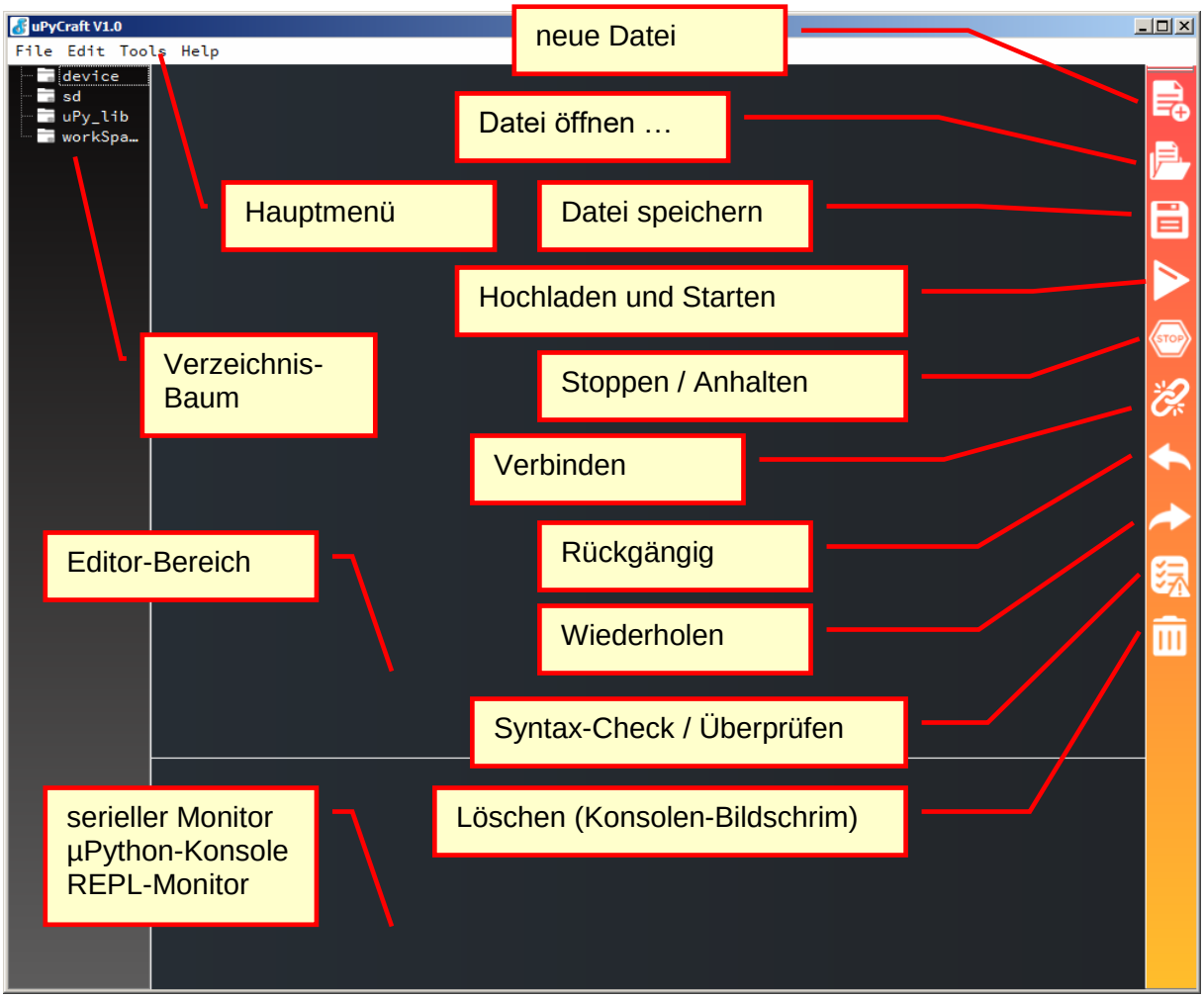
Zum ersten Ausprobieren reicht die serielle Konsole. Irgendwann müssen wir Dateien auf den ESP tranferieren. Dazu benötigt man ebenfalls ein spezielles Programm.
Das Programm **uPyCraft** ist ein sehr flexibles Werkzeug zum Arbeiten mit MicroPython auf einem Microcontroller.

Installation und Beschreibung des Hilfs-Programms uPyCraft

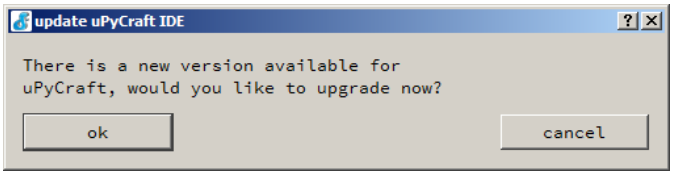
Das u im Namen des Programms steht dabei für µ - also micro. Im Internet hat sich diese Ersetzung bei vielen Projekten manifestiert.
Der Download erfolgt von der Seite https://randomnerdtutorials.com/uPyCraftWindows. Man erhält eine funktionsfähige EXE. Diese kann an eine beliebige Stelle kopiert werden – u.a. auch auf einen USB-Stick mit einem portableApps-System.

Nach dem Start der EXE kommt eine Bitte, eine spezielle Schrift-Art zu installieren.
Das kann man tun. Gleich dannach öffnet sich das Programm-Fester





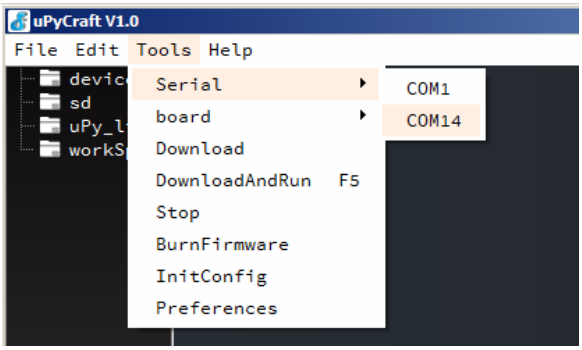
U.U. bietet uPyCraft jetzt an, eine aktuellere Version herunterzuladen.
Bei mir brach das Upgrade immer mit einer Fehler-Meldung ab.
Auf der Projekt-Webseite war aber auch keine neuere Version aufgelistet.



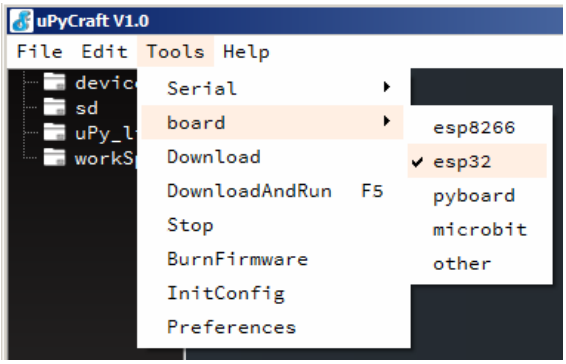
ev. kommt auch noch eine Nachfrage, ob man Beispiel-Dateien aktualisieren will. Auch die sollte man tun.

Die grundlegenden Werkzeuge aus der rechten Symbol-Leiste und die wichtigsten Elemente des Programms sind in der obigen Abbildung aufgezeigt.

Zum Testen der IDE muss zuerst einmal der richtige COM-Port unter "Tools" "Serial" ausgewählt werden. Üblicherweise ist es einer mit einer höheren Nummer.
Ist nur COM1 verfügbar, dann sollte man den USB-Treiber aktualisieren.

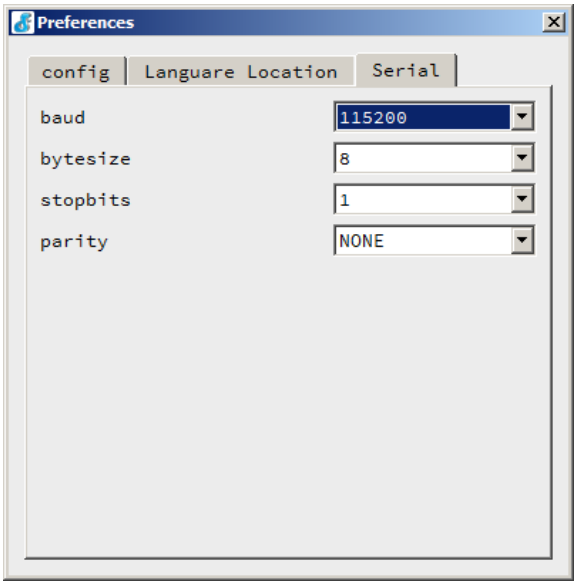


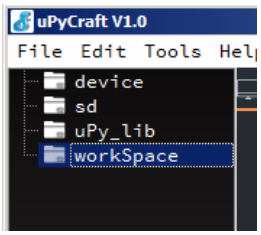
Als nächstes wählt man bei "Tools" "board" das zu benutzende Board.
Wer ein microbit mit Python programmieren will, wird hir auch fündig.



"Tools" "Preferences"

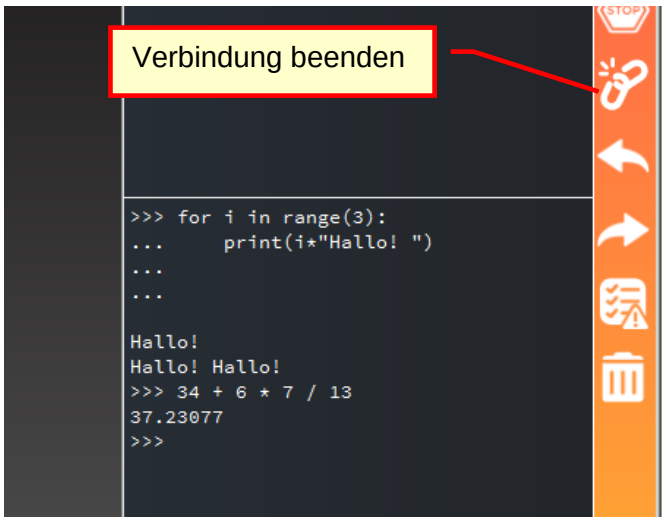
eigentlich nur der Reiter "Serial" interessant, da hier ev. die Übertragungs-Parameter für die USB-seriell-Schnittstelle eingestellt werden.





Klick auf "workSpace" führt zur Ordner-Auswahl. Hier bestimmt man einen Odner, indem sich der "WorkSpace" – also der "Arbeits-Ordner" befindet.
Nicht den "workSpace"-Ordner selbst auswählen, sondern das übergeordnete Verzeichnis, in dem sich eben "workSpace" befindet.
In workSpace müssen sich die Ordner und Dateien befinden, die wir uns von github (https://github.com/Tasm-Devil/Micropython-Tutorial-for-esp32/archive/master.zip) runtergeladen und dann entpackt haben.

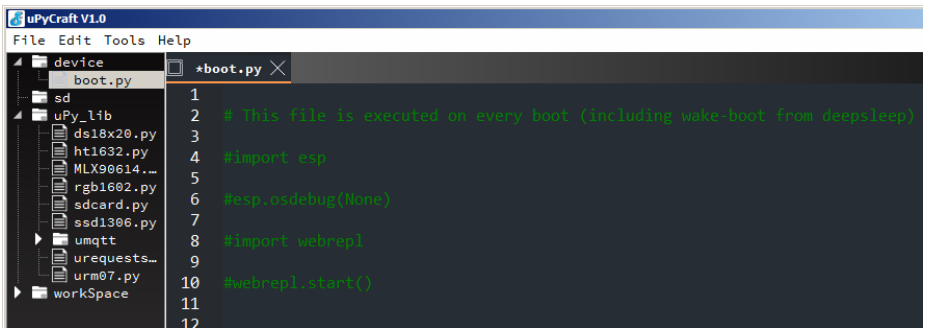
Mit "Connect" stellt man die Verbindung zur MicroPython-Konsole dar. Hier arbeitet man dann im REPL-Modus.
Praktisch entspricht dies dem inetraktiven Modus des "großen" Python.
Über "Disconnect" (Verbindung beenden") wird die Kommunikation zum ESP beendet. Dies ist z.B. notwendig, wenn wir andere Aufgaben – wie das Hochladen von Dateien – durchführen wollen. Ees ist immer nur eine Verbindung über den USB-Anschluß möglich.



??? Übertragen einer neuen Firmware

Im "Burn Firmware"-Dialog beim board "esp32" einstellen und erase_flash auf "yes" setzen, dann bestätigen

Ein frisch geflashtes MicroPython bringt schon eine **boot.py** mit. Diese können wir für unsere Zwecke anpassen.



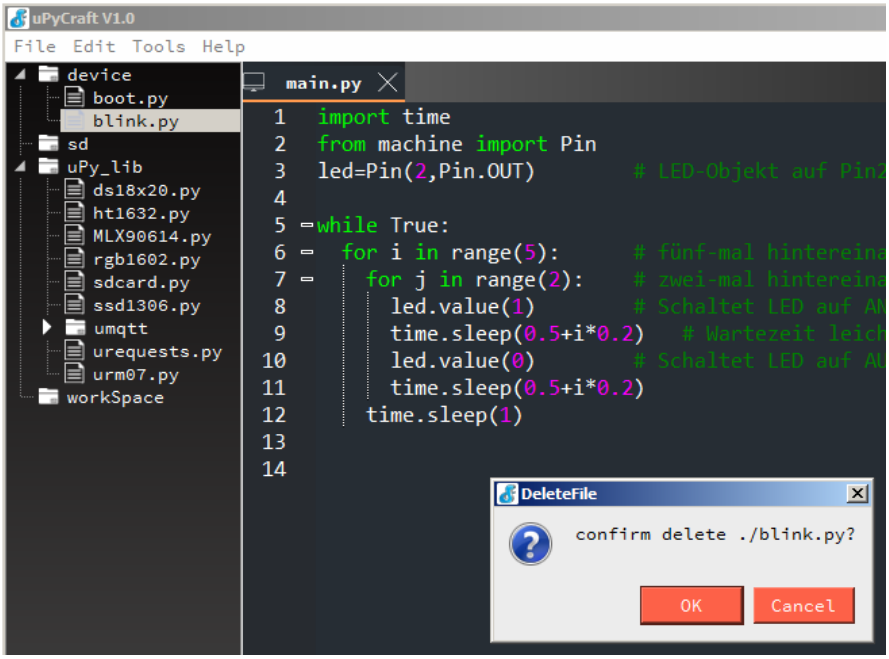
Die meisten Treiber-Dateien aus dem workSpace-Ordner müssen auf den ESP kopiert werden.
Einen neuen Ordner legt man durch Rechts-Klick auf das übergeordnete Verzeichnis an. Es gibt im Kontext-Menü nur den einen Eintrag "New dir".

Sollte sich der Verzeichnis- und Datei-Baum nach Aktionen nicht ändern, dann frischt ein "File" "Reflush Directory" die Ansicht wieder auf.

Löschen einer Datei auf dem ESP

Der Ordner "device" im Verzeichnis-Baum links zeigt uns die vorhandenen Dateien auf dem Microcontroller.

bei mir gab es Fehlermeldung und kein Löschen



Bearbeiten von Dateien im oberen mittleren Bereich (Editor) möglich.
Das "Speichern unter …" (Save as) erfolgt standardmäßig im Workspace.
Da die Start-Programme immer main.py heißen müssen, bietet sich im Workspace ein Abspeichern in einem Projekt-Ordner an.

Deutsche Umlaute usw. werden bei einem erneuten Laden (Open = Öffnen) in chinesische Schriftzeichen umgesetzt. Man sollte also durchgehend – auch bei den Kommentaren – auf spezielle Zeichen verzichten.

Das eigentliche Laden der Programm-Dateien usw. erfolgt über "Hochladen und Starten" (DownloadAndRun).
Vorher sollte man nochmal die "Verbindung" (Tools → Serial und → board) überprüfen. Bestimmte fehlende Angaben führen auch schnell mal zu undefinierten Zuständen, in denen dann nur noch ein vollständiger Neustart des Systems (oder gar ein Neuflashen des MicroPython) hilft.
Die Bezeichnung "Download" ist sicher etwas ungewöhnlich, wir spechen eher von Upload (Hochladen). Vielleicht ist es aus der Sicht des ESP-Systems bzw. des MicroPython gedacht.

nach dem erfolgreichen Hochladen (Download) erhält man im seriellen Monitor die Anzeige:

exec("main.py",,results())

meine Programme führten – auch nach einem Reset am ESP zu keiner Reaktion

ganz im Gegenteil, es kam zum dauerhaften Abbruch / Verklemmen der USB-Verbindung es half nur noch Neuflashen des ESP mit MicroPython's (funktionierte aber mit uPyCraft)

Wenn uPyCraft anstandslos läuft und alle Aufgaben machbar sind, dann kann jetzt direkt bei → 10.6.x.1. Arbeiten mit MicroPython weitergelesen werden.

Will oder kann man uPyCraft nicht benutzen, dann bleiben einige Komandozeilen-Programme übrig, die zum Hochladen von Dateien auf den ESP gedacht sind.

Hochladen von Dateien z.B. möglich mit ampy (Adafruit MicroPython Tool)

Installation über:
pip install adafruit-ampy

Hilfe aufrufen:
ampy --help

in Windows einstellen des AMPY_PORT über:
set AMPY_PORT=COM1

so ähnlich können auch AMPY_BAUD und AMPY_DELAY gesetzt werden
set AMPY_BAUD=115200
set AMPY_DELAY=0.5

die Hilfe zu **ampy**:

```
$ ampy --help
Usage: ampy [OPTIONS] COMMAND [ARGS]...
  ampy - Adafruit MicroPython Tool
  Ampy is a tool to control MicroPython boards over a serial
  connection.  Using ampy you can manipulate files on the board's
  internal filesystem and even run scripts.
Options:
  -p, --port PORT  Name of serial port for connected board.  Can
                   optionally specify with AMPY_PORT environemnt
                   variable.  [required]
  -b, --baud BAUD  Baud rate for the serial connection (default
                   115200).  Can optionally specify with AMPY_BAUD
                   environment variable.
  --version        Show the version and exit.
  --help           Show this message and exit.
Commands:
  get    Retrieve a file from the board.
  ls     List contents of a directory on the board.
  mkdir  Create a directory on the board.
  put    Put a file or folder and its contents on the...
  reset  Perform soft reset/reboot of the board.
  rm     Remove a file from the board.
  rmdir  Forcefully remove a folder and all its...
  run    Run a script and print its output.
```

10.6.x.0.2. Nutzung eines ESP mit microPython unter Linux

Arbeiten im REPL-Modus

```
$ cu -l /dev/ttyUSB0 -s 115200
```

```
>>> import machine
>>> pin = machine.Pin(5, machine.Pin.OUT)
>>> pin.value(True)
```

Äquivalent zu Blinky

```
>>> import machine
>>> import time
>>> pin = machine.Pin(5, machine.Pin.OUT)
>>> while True:
...     pin.value(True)
...     time.sleep(1)
...     pin.value(False)
...     time.sleep(1)
```

Nutzung des Filesystems

```
>>> import os
>>> os.listdir()
['boot.py']
>>> datei = open('hallo.txt', "w")
>>> datei.write("Hallo Welt!")
11
>>> datei.close
>>> os.listdir()
['boot.py', 'hallo.txt']
```

```
>>>
```

weiterhin Arbeiten mit WebREPL, den mpy-utils oder upip (micropython package manager) möglich

notwendige Dateien und Verzeichnisse der ESP32 Repo-Files
/
drivers/
extmod/
lib/
mpy-cross/
py/
tools/

sowie Ordner für die konkrete Plattform, z.B.
esp32/
-- Makefile
-- modesp.c
-- modmachine.c
-- modnetwork.c
-- modsocket.c
-- moduos.c
-- modutime.c

```
>>> import esp
>>> dir(esp)
['  name  ', 'flash read', 'flash write', 'flash erase',
'flash_size', 'flash_user_start']
>>> esp.flash_size()
4126345
```

esp-idf/components/spi_flash/include/esp_spi_flash.h
Beispiel für die c-Funktions-Deklarationen in den Header-Dateien

```
size_t spi_flash_get_chip_dir();
```

esp32/modesp.c
c-Code

```
#include "esp_spi_flash.h

STATIC mb_obj_t esp_flash_size(void){
  return mp_obj_new_int_from_uint(spi_flash_get_chip_size());
}
STATIC MP_DEFINE_CONST_FUN_OBJ_0(esp_flash_size_obj,
esp_flash_size);

STATIC const mp_rom_map_elem_t esp_module_globals_table[] = {
    { MP_ROM_QSTR___name__), MP_ROM_QSTR(MP_QSTR_esp) }.
    { MP_ROM_QSTR_flash_size), MP_ROM(&esp_flash_size_obj) },
};
STATIC MP_DEFINE_CONST_DICT(esp_module_globals,
esp_module_globals_table);

const mp_obj_module_t esp_module = {
    .base = {},
    .globals = (mp_obj_dict_t*)&esp_module_globals,
};
```

esp/mpconfigport.h

```
extern const struct _mp_obj_module_t esp_module;

#define MICROPY_PORT_BUILTIN_MODULES \
    { MP_OBJ_NEW_QSTR_esp), (mp_obj_t)&esp_module }, \
```

esp32/Makefile

```
SRC_C = \ modesp.c \
```

```
>>> import esp
>>> dir(esp)
['__name__', 'flash_read', 'flash_write', 'flash_erase',
'flash_size', 'flash_user_start']
>>> esp.flash_size()
4126345
```

10.6.x.0.3. Esp-Tool

Download als GitHub-ZIP von https://github.com/espressif/esptool

Installation mit

pip install esptool

wenn das nicht funktioniert kann man auch:

python –m pip install esptool

oder:

pip2 install esptool

probieren. (ev. auch vorher pip aktualisieren())

Ganz neue Versionen des ESP-Tool's müssen manuell installiert werden. Das sollte aber den Profi's vorbehalten sein. Es handelt sich dann meist um frische Entwicklungs-Version, deren Stabilität nicht sicher ist. Die stabilen Version sind immer über pip installierbar.
Für eine manuelle Installation gibt man:

python setup.py install

Gleiches kann man mit pySerial machen:

pip install pyserial oder easy_install pyserial oder apt-get install python-serial

Letzteres funktioniert natürlich nur unter Linux.

Die ESPtool's stellen die folgenden Kommando's zur Verfügung

ESPtool-Kommando's

- **verify_flash**
- **dump_mem**
- **load_ram**
- **read_mem**
- **write_mem**
- **read_flash_status**

- **write_flash_status**
- **chip_id**
- **make_image**
- **run**

ESP-Pin	serieller Pin (Host-Schnittstelle)
TX (GPIO1)	RX (empfangen)
RX (GPIO3	TX (senden)
Ground (GND)	Ground / Masse

Achtung! Die ESP's nutzen 3,3V-TTL-Spannung, während die üblichen Geräte-Schnittstellen 5V (Standard RS-232) benutzen. Hier muss also ein passender Adapter verwendet werden!
(Zwischen dem Standard-Pin (5V) und dem ESP-Pin (3V3) kommt ein Widerstand von 1kΩ und auf der ESP-Seite zwischen dem ESP-Pin und Ground ein 2,2 kΩ Widerstand. In der anderen Richtung – also bei der Daten-Übertragung vom ESP zur Standard-Schnittstelle benötigt man praktisch keine Anpassung, da die gelieferten 3,3 V als gültiges Signal akzeptiert wird.)

Wem die Kommandozeilen-Version (reines ESPtool) nicht so liegt, kann auch das Programm ESP8266-Flasher (→http://www.dietrich-kindermann.de/Downloads/ESP8266-Flasher-x32-Installer.zip) benutzen. Im Vorfeld muss allerdings die graphische Oberfläche wxPython installiert werden, da diese vom Flasher benutzt wird.

python –m pip install wxpython

Um das den ESP8266-Flasher unabhängig von einer Python-Installation (- also z.B. auf einem Fremd-Rechner -) benutzen zu können, kann man sich mit dem PyInstaller ein selbstständiges Installations-Paket erstellen.
Dazu wird zuert PyInstaller installiert:

python –m pip install pyinstaller

Der PyInstaller benötigt noch den UPX-Packer (→https://upx.github.io/). Dies ist ein klasisches Pack- und Entpack-Programm, dass sich auf ausführbare Packete spezialisiert hat.

Alternativ kann des NSIS-Installer benutzt werden.

Backup und Restore (Sichern und Wiederherstellen) der offiziellen Firmware von einem ESP-Mircocontroller

Wenn noch nicht geschehen, ESPtool installieren. Dazu in der Konsole in den Ordner mit dem entpsckten ESPtool wechseln (oder im Windows-Explorer / Arbeitsplatz / Computer) bei gedrückter [⇧]-Taste das Kontext-Menü zum Ordner öffnen und dann "Eingabeaufforderung hier öffnen" auswählen.

python setup.py install
pip install pyserial

angenommen es handelt sich um einen 4MB-Flash-Speicher auf dem ESP und der ESP hängt am USB-Port COM8, dann lauten die Befehle

```
python esptool.py –b 115200 –port COM8 read_flash 0x000000 0x400000 ⏎
flash4M.bin
```

```
python esptool.py erase_flash

python esptool –b 115200 –port COM8 write_flash –flash_freq 80m 0x000000
flash4M.bin
```

Die variablen Teile sind farblich hervorgehoben. Die Datei-Namen (bin-Dateien) sind hier natürlich nur Beispiele.

10.6.x.1. Arbeiten mit MicroPython

optimale ESP32-Hardware sind die WROVER-Versionen, da sie zusätzlichen Speicher onboard haben
dieser spRAM ist für größere Python-Programme dann auch notwendig
umgesetzt wurde Python 3 in einer abgespeckten – aber prinzipiell funktionsfähigen – Version
bei anderen Microcontrollern muss immer genau geprüft werden, was geht und was nicht

MicroPython-System muss einmalig auf den Microcontroller gespielt werden
dann gibt es zwei Betriebs-Möglichkeiten

Nutzungs-Möglichkeiten von MicroPython auf einem Microcontroller

- **interaktiver Interpreter** — REPL-Console (Read-Evaluate-Print-Loop)
  - lokale Version (→ 10.6.x.y.1. interaktiver Modus - REPL)
  - Internet-fähige / Netzwerk-Version (→ 10.6.x.y.2. interaktiver und Internet-fähiger Modus - WebREPL)
- **AutoRun-Modus "AutoStart"-Modus** — führt nach dem Reboot / Reset automatisch die **boot.py** und die **main.py** aus
  ansonsten können beliebige Dateien im Board-eigenen Dateisystem bearbeitet / genutzt werden
  (→ 10.6.x.y.3. "Autostart"-Modus)

in der boot.py lassen sich z.B. bestimmte Hilfs-Funktionen / -Makro's / WLAN initieren usw. ablegen, die beim Starten des ESP abgearbeitet werden

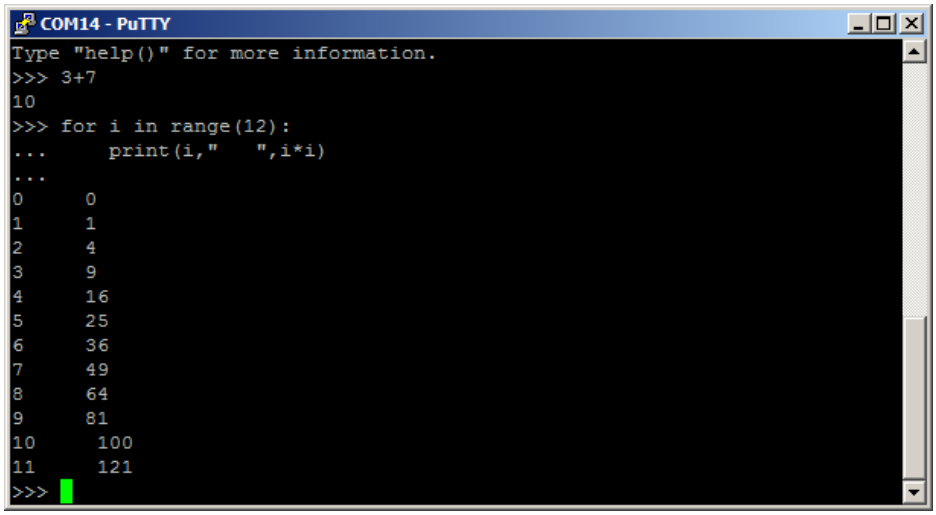
10.6.x.y.1. interaktiver Modus - REPL

praktisch identisch mit dem interaktiven Modus des normalen Python-Interpreter's
REPL steht für Read-Evaluate-Print-Loop
der MicroPython-Interpreter ließt die Kommandozeile (Read), überprüft und übersetzt dann das Kommando (Evaluate), was letztendlich zu einer Reaktion (üblich wohl eine Ausgabe mit print())
Das ganze läuft – wie üblich für Microcontroller – in einer Endlos-Schleife (Loop).

Die Menü-Befehle fehlen im REPL-Modus, so dass hier keine Erstellung oder Nutzung von Quellcode-Dateien erfolgen kann. Das muss man extern auf einem echten Rechner mit Editor oder Python-System erledigen.

nach der Verbindung über eine serielle Konsole, können die üblichen interaktiven Befehle oder Programm-Strukturen erledigt werden

hier zwei kurze und einfache Beispiele



viele Tools zum Arbeiten mit dem MicroPython nutzen genau diesen Modus und vereinfachen nur die Nutzung
man braucht dann i.A. nur noch ein Programm (eben dieses Tool), um sinnvoll mit dem Microcontroller zu arbeiten
Beispiele sind uMyCraft, …

Im REPL-Modus sind vor allem die internen Sensoren sowie die anschließbaren Sensoren und Aktoren interessant. Man kann die verschiedensten Busse und Port frei nutzen. Ein isoliertes Arbeiten des Microprocessors mit den Sensoren und Aktoren ist so nicht wirklich möglich. Dafür muss man dann den "Autostart"-Modus (→ 10.6.x.y.3. "Autostart"-Modus) verwenden.

10.6.x.y.2. interaktiver und Internet-fähiger Modus - WebREPL

benötigt wird ein Web-Client, den man unter https://github.com/micropython/webrepl downloaden bzw. gehostet unter http://micropython.org/webrepl nutzen kann

import webrepl_setup
Konfigurieren des Web-Clients

import webrepl
webrepl.start()

webrepl.start(password='meinPaswort')

10.6.x.y.3. "Autostart"-Modus

der Modus heißt nicht wirklich so, der Name beschreibt aber schon, was hier passiert
ein Python-Programm – abgespeichert als **main.py** – wird automatisch nach einem Reboot gestartet und ausgeführt
zum Boot-System gehört auch eine weitere mögliche Python-Datei, die **boot.py** . In dieser können noch vor dem Aufruf der main.py bestimmte Einstellungen gemacht und Vorbereitungen getroffen werden.

MicroPython-Tools

- **esptool.py** — notwendig, um das MicroPython-Image (minimales Betriebssystem (Miniatur-RealTime-OS vom Board-Hersteller) und MicroPython-Interpreter) auf das Board zu flashen
  Löschen des Flash-Speichers
- **mpy-utils**
  - **mpy-fuse** — mounten des ESP als beschreibbares Datesystem
  - **mpy-upload** — hochladen einer Datei auf den ESP
- **Terminal** — klassisches Terminal (serieller Monitor)
  (Start mit: screen /dev/ttyUSB0 115200
  Beenden mit: [Strg] [a] , [k]

esptool gibt es unter https://github.com/espressif/esptool zum Downloaden
Installation über pip
pip install esptool

bei Problemen, alternativ:
python –m pip install esptool oder pip2 install esptool

weiterhin manuelle Installation möglich:
python setup.py install

oder wiederum alternativ:
pip install pyserial oder easy_install pyserial oder apt-get install python-serial

mounten des Dateisystem ist etwas langsam

anders benannte Programme lassen sich aber auch von dern MicroPython-Konsole mit:

import DateiName (ohne **.py** (also quasi als Modul))

starten

unter Windows USB-Port-Angabe mit: -p COM1

scheinbar werden mit uPyCraft gedownloadete (hochgeladene) Python-Programme gleich gestartet
auf der Konsole steht dann

exec(….)

ein Umbenennen nach main.py scheint für den normalen Start nicht notwendig zu sein

ESP mit neuem Programm starten (unter Windows)

1. **main.py erstellen** — z.B. mit IDLE oder einfachem Editor; "Start"-Datei mus als **main.py** gespeichert werden
2. **ESP-Dateisystem mounten**
3. **main.py hochkopieren**
4. **ESP-Dateisystem unmounten**
5. **ESP resetten**

Ergebnisse können auf seriellem Monitor angezeigt werden (quasi Ausgabe-Bildschirm)
geeignet ist z.B. PuTTY. Dieses Programm startet auch ohne Installation aus beliebigem Verzeichnis.
Für portableApps gibt es eine eingebaute Version, die über das portableApps-System auch automatisch geupdatet wird.
Aber es sind natürlich auch andere seriellen Konsolen geeignet.

ESP mit neuem Programm starten (unter Linux (auch Raspberry Pi möglich))

1. **main.py erstellen** — z.B. mit IDLE oder einfachem Editor; "Start"-Datei mus als **main.py** gespeichert werden
2. **ESP-Dateisystem mounten**

   ```
   mpy-fuse mnt
   ```

   esp32-mount
3. **main.py hochkopieren**

   ```
   mpy-upload Datei
   ```
4. **ESP-Dateisystem unmounten**

   ```
   fusermount –u mnt
   ```
5. **ESP resetten**

mit esp32-terminal auf seriellen Monitor zum ESP zugreifen

Abfrage von freiem Speicher etc.

```
import gc
gc.mem_free()
```

verfügbare Funktionen über gc. abfragbar (Code-Ergänzungs-System)

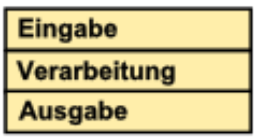
Garbage-Collection anstoßen
```
gc.collect()
```

10.6.x.4. elementare Programmierung mit MicroPython

In den folgenden Kapiteln besprechen wir die Programmierung mit Python auf einem Microcontroller (hier vorrangig der ESP32). Das ist eine Wiederholung vieler Abschnitte und Themen von weiter vorne in diesem Skript. Ich möchte hier aber eine auskoppelbare Einheit für Nutzer erstellen, die sich nur mit Microcontrollern und MicroPython auseinandersetzen wollen oder müssen.
Wer die Grundlagen nicht mehr braucht und sich gleich mit den Spezialitäten der Microcontroller beschäftigen möchte kann jetzt zu → springen.
Unter elementarer Programmierung verstehe ich nur einfachste Elemente einer Programmiersprache, die zu den absoluten Grundlagen zählen. Sie folgt gleich im nächsten Abschnitt (→ 10.6.x.4. elementare Programmierung mit MicroPython). Dazu gehören vorrangig Ein- und Ausgaben (auf Konsolen-Niveau) sowie einfache Verzweigungen und Schleifen. Python ist hier nur das spezielle Mittel.
Die klassische Programmierung (→ 10.6.x.5. klassische Programmierung mit MicroPython)beschäftigt sich aus meiner Sicht mit Listen, Wörterbücher (Dictonary's) Funktionen, Objekten usw. Sie gehören zu einem Niveau, bei dem die modernen Aspekte der Programmierung sowie die speziellen Möglichkeiten von Python eine Rolle spielen. Der Pythongrundgebildete Leser wird hier hin und wieder die Einschränkungen des MicroPython spüren. Für alle anderen ist es die Besprechung einer Leistungs-fähigen Programmiersprache.
Im Anschluß daran folgt die Geräte-nahe Programmierung. Hier kommen nun die Merkmale und Fähigkeiten der Microcontroller deutlich zum Vorschein. Deshalb nennen ich das auch spezielle Programmierung (→ 10.6.x.6. spezielle Programmierung mit MicroPython). Es ist nicht auszuschließen, dass sich dieser Teil nicht – so wie dargestellt – auf jeden anderen Microcontroller übertragen läßt.

10.6.x.4.1. Ausgaben

Abweichend vom EVA-Prinzip beginnen wir mit den Ausgaben. Dies sollten wir können, damit andere Leistungen eines Programm's von uns zumindestens kontrolliert werden können.
Ein Programm ohne Ausgaben ist praktisch nutzlos. Dabei müssen Ausgaben nicht immer auf dem Bildshirm erfolgen. Oft werden Daten auch ausgedruckt, in eine Datei gespeichert oder einem Aktor (z.B. Ein- und Auschalten einer LED) zugewiesen.
In der Programmierung hat sich die Block-Darstellung in sogenannten Struktogrammen durchgesetzt. Sie dienen der Veranschaulichung von Algorithmen (Programm-Abläufen) unanhängig von einer konkreten Programmiersprache. Gute Struktogramme lassen sich in sehr viele Programmiersprachen übertragen.



Struktogramme sind immer große Blöcke (Rechtecke), die intern in kleinere unterteilt werden (können).
Man ließt ein Struktogramm immer von oben nach unten. D.h. im Beispiel beginnt das Programm mit der Eingabe. Es folgt eine Verarbeitung (der Daten) und schließt mit einer Ausgabe ab.
Der klassische Ausgabe-Befehl in Python ist **print()**. In die Klammern können Kommagetrennt mehrere verschiedenartige Ausdruck-Elemente notiert werden.
Schauen wir uns zuerst den Ausdruck jeweils eines einzelnen Elementes an, um dann die Zusammenstellung zu längeren Ausdrucken zu besprechen.

Ausdruck-Element	Beispiel	Erläuterung / Bemerkungen / Hinweise

Text / Zeichenkette
 	
Verkettungen
	`print("Hallo Welt!")`
	
	`print("Hallo "+"Welt!)`
		beide print-Befehle erzeugen ein: **`Hallo Welt!`** auf dem Bildschirm
		bei einer Verkettung werden Zeichenketten mit einem Plus verbunden

Zahl
 	
Berechnung
	`print(24)`
	
	`print(2.713)`
	
	`print(24+7)`
		druckt **`24`** auf dem Bildschirm
		
		druckt die Kommazahl **`2.713`** aus **(die Zahlen-Darstellung entspricht dem englischen Stil / Format! Der Punkt ist der Dezimal-Trenner)**
		
		druckt den berechneten Betrag von 24+7, also **`31`**, aus

Variablen-Wert	`print(PI)`
	
	`print(x)`
	`print(Hallotext)`
		druckt die Kreiszahl π aus: 3.1412
		
		haben die Variablen **x** und **Hallotext** vorher einen Wert bekommen, dann wird dieser ausgedruckt, egal ob das eine Zahl oder ein Text ist
		ansonsten erscheint eine Fehler-Meldung

Der Unterschied zwischen Texten und Variablen ist die Notierung mit bzw. ohne Anführungsstriche. Statt den doppelten Anführungsstrichen sind auch einfache erlaubt. Sie müssen aber immer paarweise – also am Beginn und am Ende des Textes benutzt werden.
Jeder Befehl wird einzeln in die MicroPython-Konsole eingegeben. Nach einem [Enter] erscheint sofort die Ausgabe in der Zeile darunter.Dieses Wechselspiel von eingegebenen Befehlen und die sofortigen Ausgaben des Python-System's nennt man den **interaktiven Modus**.
Der Python-Kenner wird nun sagen, dass geht aber alles auch einfacher. Das stimmt! Praktisch hätten wir die print-Befehle und die zugehörigen Klammern weglassen können. Im interaktiven Modus sind sie nicht notwendig. Da wir aber später echte und vor allem größere Programme schreiben wollen, gewöhnen wir uns schon mal an die Schreibweise mit Befehl.

Aufgaben:

1. *Prüfe zuerst anhand der oben angegebenen Informationen, ob die folgenden Befehle ordnungsgemäße Ausgaben (im interaktiven Modus) erzeugen! (Die Befehle der letzten Zeile werden hintereinander mit jeweils einem Enter eingegeben!)*

a)	print("Python")	b)	print "Hallo"	c)	print(24 + "*")
d)	write("Jetzt aber!"	e)	print("Test Nr. 5)	f)	print(3*"+++")
g)	x = 56 print(x)	h)	print(3*x) a = "4" print(x+a)	i)	c = 4 print("x+c")

2. *Prüfen Sie nun die Befehle im MicroPython-System! Welche Überraschung(en) gibt es?*
3. *Berichtigen Sie die fehlerhaften Befehle und prüfen Sie diese im MicroPython!*
4. *Erzeugen Sie die folgenden Ausgaben!*

a) Drucken Sie den folgenden Satz als ganzes aus!
Python ist schon eine tolle Programmiersprache.
b) Drucken Sie den Satz nun als Zusammensetzung von einzelnen Wörtern!
c) Erzeugen Sie die Ausgabe Ihres Namens aus einzelnen Buchstaben!
d) Lassen Sie Python das Ergebnis der folgenden Berechnung ausdrucken! Rechnen Sie zuerst im Kopf!
5 * 3 + (21 – 7) * -2 / 2

Oben wurde schon erwähnt, dass sich mehrere Ausgabe-Elemente Komma-getrennt hintereinander in einem print-Befehl unterbringen lassen. Das kann man z.B. nutzen, um ordentliche Ergebnis-Sätze auszugeben. Die Ausgabe-Elemente können frei kombiniert werden. Auch deren Anzahl ist nicht begrenzt. Insgesamt sollte eine normale Ausgabe aber eine Zeile nicht überschreiten.

```
print("das Volumen des 5x5x5 Würfels beträgt: ", 5*5*5)
```

Zur allgemeinen Beschreibung von Befehlen benutzen wir gerne die folgende Syntyx-Darstellung:

```
print()
print(ausgabeelement)
print(ausgabeelement, ausgabeelement)
print(ausgabeelement { , ausgabe} )
```

Die oberste Zeile beschreibt eine leere Ausgabe. Das entspricht einer Leerzeile. Die blau angezeigten Elemente / Zeichen sind notwendige Details (für die Info-Profi's: Terminale).
In der zweiten Zeile ist der Syntax einer einfachen Ausgabe dargestellt. Das kursiv geschriebene *ausgabeelement* ist ein Platzhalter für eine korrekte Ausgabe. Der Name könnte durch einen anderen ersetzt werden – z.B. *druckobjekt* od.ä. Hier sprechen wir dann von einem Nicht-Termininal.
Sollen zwei Ausgabeelemente ausgegeben werden, dann müssen sie durch ein Komma (,) voneinander getrennt in der Klammer aufgezählt werden. Das Komma ist also vorgeschrieben (also ein Terminal).
In der 4. Zeile wird mit den rot geschriebenen geschweiften Klammern ({ }) gekennzeichnet, was sich beliebig oft hintereinander wiederholen darf: immer ein Komma und dann ein neues

Ausgabeelement. Die geschweiften Klammern dienen hier nur zum Kennzeichnen der Wiederholung. Sie werden nicht mitgeschrieben. Sie sind sogenannte Meta-Symbole im Syntax. Die nächste Zeile ist die gemeinsame Syntax aller obigen Zeilen:

print(**[** *ausgabeelement* **]** | *ausgabeelement* **{** , *ausgabeelement* **}**)

Die eckigen Klammern (**[ ]**) stehen für eine Option. Das Elemenet kann sein, muss es aber nicht. Der senkrechet Strich (**|**) kennzeichnet die Alternative – also entweder das links oder das rechts vom Strich. Das Lesen von Syntax-Darstellungen ist zuerst immer etwas gewöhnungsbedürftig. Später wird es zum effektiven Mittel, um die Möglichkeiten von Befehlen effektiv darzustellen.

Aufgaben:

1. Realisieren Sie die nachfolgenden Ausgaben!

a) Stellen Sie in einem print-Befehl den Text "Die Summe beträgt: " und die berechnete Zahl aus 34+41+26 in einer Zeile zusammen!

c) Geben Sie nachfolgende Berechnung so aus, dass die Zahlen separat im print-Befehl auftauchen (die sollen später durch Variablen ersetzt werden)! Die Berechnung des Ergebnisses darf direkt im print-Befehl erfolgen.
12,5 + 24 – 48 / 4 = …

b) Die obige Rechnung soll mit in den Text eingebunden werden. Dabei dürfen die Zahlen nicht in der Zeichenkette vorkommen, sondern müssen separat eingeabeitet werden!

d)

2.

Ausgaben auf Aktoren usw. besprechen wir erst später (→ 10.6.x.6. spezielle Programmierung mit MicroPython), da diese vom verwendeten Microcontroller abhängig sind.

???formatierte Ausgaben

10.6.x.4.2. Variablen, Zuweisungen und Berechnungen

Be den Ausgaben haben wir schon nebenbei mit Variablen gearbeitet. Die am meisten verwendete ist sicher x. In Python-Programmen können wir alle Zeichenketten, die mit einem Buchstaben beginnen und dann von Buchstaben, Ziffern und dem Unterstrich gefolgt werden. Gute Programmierer verwenden sprechende Variablen, d.h. solche deren Namen ihren Verwendungszweck beschreibt. Zum Einen verbessert das die Lesbarbeit von Programmen und zum anderen werden Verwechslungen oder versehentliche Doppelbenutzungen vermieden. Oft werden durch die sinnvolle Benennung von Variablen ihre Rollen in Algorithmen deutlicher. Dadurch lassen sich Programmierfehler schneller finden.
Das Volumen in einer Formel kann also z.B. mit **x**, **v**, **V**, **volumen** oder **Volumen** als Variable benutzt werden. Es wird gleich klar, dass die beiden letzteren am besten zu verstehen sind. In Python-Programmen wird die Groß- und Kleinschreibung unterschieden. D.h., dass volumen und Volumen zwei unterschiedliche Variablen sind. Man sollte sich auf eine Art der Schreibung in seinen Programmen festlegen. Üblich sind klein-geschriebene Variablennamen. Mit Unterstrichen oder Groß-Buchstaben kann man längere Variablennamen wieder besser lesbar machen, z.B.:

seitenSumme oder seiten_summe

Beim ersten Benutzen muss einer Variable ein Wert zugewiesen werden. Das passiert ganz einfach mit einem Gleichheitszeichen (**=**). Der Variablenname muss links stehen, der Wert rechts. Als Werte kommen Zahlen und Berechnungen, Texte und Verkettungen oder andere Variablen infrage. Eine weitere Möglichkeit stellen Funktionen dar, wobei die Eingabe-Funktion (→ 10.6.x.4.3. Eingaben) sicher die verständlichste ist.
Gültige Variablen-Deklarationen sind:

```
x = 4
y = x
HalloText = "Good morning!"
seite = 12
wuerfelVolumen = seite * seite * seite
```

Dagegen ist es z.B. falsch, die folgenden Ausdrücke zu benutzen:

4 = x
Hallo

Beim Versuch, solche Ausdrücke zu übersetzen (zu interpretieren) erzeugt der Python-Interpreter eine Fehlermeldung.

Einige spezielle Zuweisungen (Listen, Objekte, …) besprechen wir, wenn wir dort angekommen sind. Das Zuweisungs-Prinzip ist immer gleich.
Hat eine Variable erst einmal einen Wert, dann kann sie für Berechnungen und Funktionen benutzt werden. Die klassischen Rechen-Operationen haben wir ja schon so nebenbei mit vorgestellt.
Besonders muss vielleicht noch einmal auf die Multiplikation mit dem Sternchen (*****) und die Division mit dem Schrägstrich (**/**) hingewiesen werden.
Besondere Operatoren sind zwei aufeinanderfolgende Sternchen (******) als Potenz-Operator und das Prozent-Zeichen (**%**) als Modulo-Operator. Die Modulo-Operation ist die Berechnung des Restes einer ganzzahligen Division. Man verwendet die Modulo-Operation z.B. zum Bestimmen von Teilbarkeiten (s.a. → 10.6.x.4.4. Alternativen, Verzweigungen) oder in der Kryptographie.
Elementare Funktionen sind z.B. **sin()**, **cos()** oder **tan()**. Die Wurzel-Funktion wird mit **sqrt()** aufgerufen. Mit **abs()** erhält man den Absolut-Wert des Wertes in der Klammer. Die Werte in der Klammer einer Funktion sind die Argumente. Sie werden zur Berechnung des Funktionswertes benutzt. (Dazu gleich noch etwas mehr →). Dort stellen wir auch noch weitere Funktionen vor.

10.6.x.4.3. Eingaben

Eingaben von Sensoren usw. besprechen wir erst später (→ 10.6.x.6. spezielle Programmierung mit MicroPython), da diese vom verwendeten Microcontroller abhängig sind.

10.6.x.4.4. Alternativen, Verzweigungen

10.6.x.4.5. Wiederholungen, Schleifen

10.6.x.4.6. eingebaute und mitgelieferte Funktionen

10.6.x.5. klassische Programmierung mit MicroPython

10.6.x.5.1. Listen und Listen-Verarbeitung

10.6.x.5.2. Wörterbücher, Dictonary's

10.6.x.5.3. Lesen und Schreiben von Dateien, Datei-Verarbeitung

10.6.x.5.4.

10.6.x.6. spezielle Programmierung mit MicroPython

10.6.x.4. weitere spezielle Programm-Beispiele und -Schnipsel

Besonders wichtige Bibliotheken / Module sind machine und network. Sie stellen Objekte und Funktionen / Methoden für die spezielle Hardware – den speziellen Microcontroller – zur Verfügung.
Weiterhin werden oft Bibliotheken bzw. Module zu den benutzten Sensoren und Aktoren benötigt. Diese ersparen uns viel Programmier-Aufwand.
Um den RAM-Verbrauch möglichst gering zu halten, sollte man es sich angewöhnen nur die unbedingt notwendigen Bestandteile aus den Modulen zu laden.

klassisches Einstiges-Programm "Blink"
läßt die onboard-LED blinken

```
from time import sleep
from machine import Pin

led=Pin(12,Pin.OUT)

while True:
  led.value(True)
  sleep(0.2)
  led.value(False)
  sleep(0.2)
```

einen Pin Pulsweiten-moduliert ansteuern
mögliche Werte von 0 bis 1023

klassische Anwendungen für diese Ansteuerungs-Art sind zu "dimmende" LED's und Servo-Motoren

```
from time import sleep
from machine import Pin
from machine import PWM

led=Pin(12,Pin.OUT)

puls = PWM(led)
while True:
```

```
  puls.freq(200)
  sleep(1)
  puls.freq(500)
  sleep(2)
  puls.freq(1000)
  sleep(3)
```

einen Touch-Port abfragen
insgesamt 10 Touch-Eingabe möglich (0 .. 9)

```
from machine import TouchPad

touchpin = Pin(2)
touchpad = TouchPad(touchpin)
while True:
  print("Touch-Wert = ",touchpad.read())
  sleep(1)
```

LED-Ring (NeoPixel) ansteuern

LED-Streifen oder –Ringe usw. gehören heute zu den peppigen Accessoire's
zumeist sind hier die LED's einzeln ansteuerbar, bei einfarbigen LED's funktionoiert dann zumindestens das Ein- und Aus-Schalten
sind RGB-LED's verbaut, dann kann man eigentlich fast immer jede einzelne LED in einer speziellen Farbe leuchten lassen
wichtig ist hier immer, dass nach dem Setzen / verändern von Werten, diese auf den NeoPixel-Ring rausgeschrieben werden müssen

```
import machine, neopixel
import time
import random

def test(np):
  n = mp.n
  b = 5    # Helligkeit
  sl = 10   # Kurzschlafzeit

  time.sleep_ms(1000)
  np.fill((0,0,0))
  time.sleep_ms(1000)

  for i in range(n):
    np[i] = (b,0,0)
    np.write()
    time.sleep_ms(sl)

  time.sleep_ms(1000)
  np.fill((0,0,0))
  time.sleep_ms(1000)

  for i in range(n):
    np[i] = (0,b,0)
    np.write()
    time.sleep_ms(sl)
```

```
  time.sleep_ms(1000)
  np.fill((0,0,0))
  time.sleep_ms(1000)

  for i in range(n):
    np[i] = (0,0,b)
    np.write()
    time.sleep_ms(sl)

  time.sleep_ms(1000)
  np.fill((0,0,0))
  time.sleep_ms(1000)

  for i in range(n):
    np[i] = (b,b,b)
    np.write()
    time.sleep_ms(sl)

def demo1(np):
  n = np.n

def demo2(np):
  n = np.n
  # Kreis
  for i in range(4*n):
    for j in range(n):
      np[j]=(0,0,0)
    np[i%n]=(255,255,255)
    np.write()
    time.sleep_ms(25)
  # Band
  for i in in range(4*n):
    for j in range(n):
      np[j] = (0,0,128)
    if (i//n)%2 == 0:
      np[i%n] = (0,0,0)
    else:
      np[n-1-(i%n)] = (0,0,0)
    np.write()
  # Säubern
  for i in range(n):
    np[i] = (0,0,0)
  np.write()

def demo3(np):
  n = np.n
  b = 5    # Helligkeit
  sl = 1   # Kurzschlafzeit

  for i in range(10000):
    np.fill((0,0,0))
    i = random.randint(0,23)
    r = random.randint(0,100)
    b = random.randint(0,100)
    g = random.randint(0,100)
    np[i] = (r,g,b)
    np.write()
    time.sleep_ms(sl)


np = neopixel.NeoPixel(machine.Pin(15),24,timing=0)

test(np)
demo1(np)
```

```
demo2(np)
demo3(np)
```

Q: /µP_Q1/ (leicht geändert: dre)

kleine OLED-Display's sind bei einigen ESP-Bausteinen gleich mit aufgelötet. Sie ermöglich die Anzeige einiger Text-Zeilen oder kleiner Grafiken
in den meisten Fällen sind die OLED's allerdings Monochrom, was aber für die einfachen Möglichkeiten unserer Microcontroller schon super ausreicht.

ansprechen des OLED-Display's (wenn vorhanden)

screen.py

```
from machine import I2C, Pin
import time

from ssd1306 import SSD1306_I2C

_i2c = I2C(sda=Pin(5), scl=Pin(4))
_display = SSD1306_I2C(128,64,_i2c)

def text(t):
  _display.fill(0)   # OLED löschen
  lines = t.splitlines()
  y = 0
  for line in lines:
    _display.text(line,0,y)
    y += 10
  _display.show()
```

da OLED über den i2c-Bus an die Pin's 4 (Clock) und 5 (Data) angeschlossen ist, benötigt man den obigen "Treiber" für eine Text-Ausgabe
Hinweise: 128,64 stehen für Breite und Höhe des Display's in Pixel; Display wird als ganzes angesteuert, es erfolgt kein Scrollen

main.py

ein Sternchen im Ping-Pong-Modus über eine integrierte OLED-Anzeige wandern lassen

```
from time import sleep
from screen import text

def pingpong(i):
  txt('\n\n' + ' ' * i + '*' + ' ' * (15 - i) + '\n\n\n')
  sleep(0.1)

while True:
  for i in range(0,15,1):
    pingpong(i)
  for in in range(15,0,-1):
    pingpong(i)
```

Q: /µP_Q1/

Abfrage eines Licht-Sensors

```
from machine import Pin, I2C
from bh1750 import BH1750
from screen import text
from time import slep

i2c=I2C(scl=Pin(14), sda=Pin(13))

sensor BH1750(i2c)

while True:
  lum = sensor.liminance(BH1750.ONCE_HIRES_1)
  print("Lumineszenz =", lum)
  balken = "#" * int(lum/100)
  text("Lumineszenz-Sensor:\n\n%s\n % balken)
  sleep(0.5)
```

Q: /µP_Q1/ (leicht geändert: dre)

das Programm zeigt den aktuellen Meßwert auf dem seriellen Monitor an. Das OLED-Display wird zusätzlich zur Visualisierung der Lichtstärke aus Balken-Diagramm verwendet.

Wenn der ESP eins kann, dann ist das WLAN. Bei vielen Bausteinen ist gleich von der Herstellung schon ein kleines WLAN-Scan-Programm aufgespielt. Häufig werden die verschiedenen einfachen WLAN-Scanner auch als "Hallo Welt"-Programm der ESP-Welt verstanden.

WLAN-Scan

```
from network import WLAN, STA_IF
from time import sleep

wlan = WLAN(STA_IF)
wlan.active(True)

while True:
  nets = wlan.scan()
  print("Scan-Ergebnis: ================")
  for net in sorted(nets):
    print(net)
  print()
  sleep(2)
```

Q: /µP_Q1/ (leicht geändert: dre)

STA_IF … steht für den Stations-Modus des WLAN (praktisch als Client eingerichtet)

funktionierende Funktion zum Verbinden des ESP mit einem AccessPoint
ESP fungiert als einfache WLAN-Station

```
def verbinden():
  import network

  ssid = "????"
  passwort = "????"
```

```
meinwlan = networ.WLAN(network.STA_IF)
meinwlan.active(True)
if not meinwlan.isconnected():
  print("Verbindung zum WLAN herstellen …")
  meinwlan.conncet(ssid,passwort)
  while not meinwlan.connected():
    pass
print("aktuelle Netzwerk-Konfiguration:",meinwlan.ifconfig())
```

als Funktion mit Argumenten könnte verbinden() auch so aussehen

```
def verbinden(ssid,passwort):
  import network

  meinwlan = …
```

Empfang und Zurücksenden von UDP-Nachrichten / -Paketen (Echo-Funktion)

```
#include <ESP8266WiFi.h>
#include <WiFiUDP.h>

// The ESP-12 has a blue LED on GPIO2
#define LED 2

// Name and password of the access point
#define SSID "Pussycat"
#define PASSWORD "supersecret"

// The server accepts connections on this port
#define PORT 5444
WiFiUDP udpServer;

// Buffer for incoming UDP messages
char udp_buffer[WIFICLIENT_MAX_PACKET_SIZE+1];


/** Receive UDP messages and send an echo back */
void process_incoming_udp()
{
    if (udpServer.parsePacket())
    {
        // Fetch received message
        int len=udpServer.read(udp_buffer,sizeof(udp_buffer)-1);
        udp_buffer[len] = 0;

        // Display the message
        Serial.print(F("Received from "));
        Serial.print(udpServer.remoteIP());
        Serial.print(":");
        Serial.print(udpServer.remotePort());
        Serial.print(": ");
        Serial.println(udp_buffer);

        // Send echo back
        udpServer.beginPacket(udpServer.remoteIP(),
udpServer.remotePort());
        udpServer.print(F("Echo: "));
```

```
        udpServer.print(udp_buffer);
        udpServer.endPacket();

        // Execute some commands
        if (strstr(udp_buffer, "on"))
        {
            digitalWrite(LED, LOW);
            udpServer.println(F("LED is on"));
        }
        else if (strstr(udp_buffer, "off"))
        {
            digitalWrite(LED, HIGH);
            udpServer.println(F("LED is off"));
        }
    }
}


/** Optional: Notify about AP connection status changes */
void check_ap_connection()
{
    static wl_status_t preStatus = WL_DISCONNECTED;

    wl_status_t newStatus = WiFi.status();
    if (newStatus != preStatus)
    {
        if (newStatus == WL_CONNECTED)
        {
            digitalWrite(LED, LOW);

            // Display the own IP address and port
            Serial.print(F("AP connection established, listening on
"));
            Serial.print(WiFi.localIP());
            Serial.print(":");
            Serial.println(PORT);
        }
        else
        {
            digitalWrite(LED, HIGH);
            Serial.println(F("AP conection lost"));
        }
        preStatus = newStatus;
    }
}


/** Runs once at startup */
void setup()
{
    // LED off
    pinMode(LED, OUTPUT);
    digitalWrite(LED, HIGH);

    // Initialize the serial port
    Serial.begin(115200);

    // Give the serial monitor of the Arduino IDE time to start
    delay(500);

    // Use an external AP
    WiFi.mode(WIFI_STA);
    WiFi.begin(SSID, PASSWORD);
```

```
    // Start the UDP server
    udpServer.begin(PORT);
}


/** Main loop, executed repeatedly */
void loop()
{
    process_incoming_udp();
    check_ap_connection();
}
```

Q: http://stefanfrings.de/esp8266/

TCP-Server

```
#include <ESP8266WiFi.h>

// The ESP-12 has a blue LED on GPIO2
#define LED 2

// Name and password of the access point
#define SSID "Pussycat"
#define PASSWORD "supersecret"

// The server accepts connections on this port
#define PORT 5333
WiFiServer tcpServer(PORT);

// Objects for connections
#define MAX_TCP_CONNECTIONS 5
WiFiClient clients[MAX_TCP_CONNECTIONS];

// Buffer for incoming text
char tcp_buffer[MAX_TCP_CONNECTIONS][30];


/**
 * Collect lines of text.
 * Call this function repeatedly until it returns true, which
indicates
 * that you have now a line of text in the buffer. If the line does
not fit
 * (buffer to small), it will be truncated.
 *
 * @param source The source stream.
 * @param buffer Target buffer, must contain '\0' initiallly before
calling this function.
 * @param bufSize Size of the target buffer.
 * @param terminator The last character that shall be read, usually
'\n'.
 * @return True if the terminating character was received.
 */
bool append_until(Stream& source, char* buffer, int bufSize, char
terminator)
{
    int data=source.read();
    if (data>=0)
    {
```

```
        int len=static_cast<int>(strlen(buffer));
        do
        {
            if (len<bufSize-1)
            {
                buffer[len++]=static_cast<char>(data);
            }
            if (data==terminator)
            {
                buffer[len]='\0';
                return true;
            }
            data=source.read();
        }
        while (data>=0);
        buffer[len]='\0';
    }
    return false;
}


/** Optional: Notify about AP connection status changes */
void check_ap_connection()
{
    static wl_status_t preStatus = WL_DISCONNECTED;

    wl_status_t newStatus = WiFi.status();
    if (newStatus != preStatus)
    {
        if (newStatus == WL_CONNECTED)
        {
            digitalWrite(LED, LOW);

            // Display the own IP address and port
            Serial.print(F("AP connection established, listening on
"));
            Serial.print(WiFi.localIP());
            Serial.print(":");
            Serial.println(PORT);
        }
        else
        {
            digitalWrite(LED, HIGH);
            Serial.println(F("AP conection lost"));
        }
        preStatus = newStatus;
    }
}


/**
 * Put new connections into the array and
 * send a welcome message.
 */
void handle_new_connections()
{
    WiFiClient client = tcpServer.available();
    if (client)
    {
        Serial.print(F("New connection from "));
        Serial.println(client.remoteIP().toString());

        // Find a freee space in the array
        for (int i = 0; i < MAX_TCP_CONNECTIONS; i++)
```

```
        {
            if (!clients[i].connected())
            {
                // Found free space
                clients[i] = client;
                tcp_buffer[i][0]='\0';
                Serial.print(F("Kanal="));
                Serial.println(i);

                // Send a welcome message
                client.println(F("Hello World!"));
                return;
            }
        }
        Serial.println(F("To many connections"));
        client.stop();
    }
}


/** Receive TCP messages and send echo back */
void process_incoming_tcp()
{
    static int i=0;

    // Only one connection is checked in each call
    if (clients[i].available())
    {
        // Collect characters until line break
        if
(append_until(clients[i],tcp_buffer[i],sizeof(tcp_buffer[i]),'\n'))
        {
            // Display the received line
            Serial.print(F("Empfangen von "));
            Serial.print(i);
            Serial.print(": ");
            Serial.print(tcp_buffer[i]);

            // Send an echo back
            clients[i].print(F("Echo: "));
            clients[i].print(tcp buffer[i]);

            // Execute some commands
            if (strstr(tcp_buffer[i], "on"))
            {
                digitalWrite(LED, LOW);
                clients[i].println(F("LED is on"));
            }
            else if (strstr(tcp_buffer[i], "on"))
            {
                digitalWrite(LED, HIGH);
                clients[i].println(F("LED is off"));
            }

            // Prepare the buffer to receive the next line
            tcp_buffer[i][0]='\0';
        }
    }

    // Switch to the next connection for the next call
    if (++i >= MAX_TCP_CONNECTIONS)
    {
        i=0;
    }
```

```
}


/** Executes once during start*/
void setup()
{
    // LED off
    pinMode(LED, OUTPUT);
    digitalWrite(LED, HIGH);

    // Initialize the serial port
    Serial.begin(115200);

    // Give the serial monitor of the Arduino IDE time to start
    delay(500);

    // Use an external AP
    WiFi.mode(WIFI_STA);
    WiFi.begin(SSID, PASSWORD);

    // Start the TCP server
    tcpServer.begin();
}


/** Main loop, executed repeatedly */
void loop()
{
    handle_new_connections();
    process_incoming_tcp();
    check_ap_connection();
}
```

Q: http://stefanfrings.de/esp8266/

Links:

http://docs/mircopython.org/en/latest/esp8266/ (Dokumentation in Entwicklung, muss für ESP-32 interpretiert werden)

https://randomnerdtutorials.com/getting-started-micropython-esp32-esp8266/ (online Arbeitsanleitung)

10.6.x.5. spezielle Module für ESP-32-Microcontroller

Info- und Quellcode-Q: docs.micropython.org/en/latest/library/index.html (Quellcode's leicht geändert)

Einige der Module sind auch für andere Microcontroller verfügbar. Meist sind diese Board-spezifisch, d.h. sie müssen auf der micropython-Website als Download-Paket genauestens ausgewählt werden.
In der jeweiligen Nutzung kann zu veränderten Notierungen und Varianten – im Vergleich zu den folgenden Darstellungen – kommen.

10.6.x.5.1. Modul "machine"

import machine

machine.freq()
liefert die aktuelle Prozessor-Frequenz zurück (in Hz)

machine.freq(240000000)
setzt die Prozessor-Frequenz auf 240 MHz

Deep-sleep-Modus ()

machine.deepsleep(100000))
Versetzt den Microcontroller für 100 s in den Tiefschlaf-Modus (Stromspar-Modus)
ohne Parameter wird der Microcontroller dauerhaft in den Tiefschlaf-Modus versetzt
ein weiteres Stromsparen ist durch Setzen / Einschalten von internen Pull-up-Widerständen möglich
p1 = Pin(4, Pin.IN, Pin.PULL_HOLD)

```
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
   print("Microcontroler ist aufgeweckt!")
```

RTC (realtime clock)

from machine import RTC

rtc = RTC()
rtc.datetime((Jahr, Monat, Tag, Stunde, Minuten, Sekunden, MilliSekunden))
über die Abfrage eines NTC-Servers ist eine recht genaue Zeitsynchronisierung möglich

rtc.datetime()
gibt das aktuelle Datum und die Zeit zurück

Zähler / Timer

from machine import Timer

Zaehler = Timer(-1)
Zaehler.init(period=1000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
Zähl-Einheit sind MilliSekunden

Zaehler.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))

Pin's / GPIO

from machine import Pin

verfügbar sind die Pin's 0 .. 19, 21 .. 23, 25 .. 27 und 32 .. 39
abhängig von den Pin's, die auf dem Board nach außen geführt wurden (je nach Hersteller und Board-Art unterschiedlich)
weiterhin gilt:

- Pin 1 ist für TX und Pin 3 für RX der seriellen Verbindung über UART in Gebrauch
- die Pin's 6 .. 8, 11, 16 und 17 sind für die Verbindung mit dem eingebauten Flash-Speicher in Gebrauch und können nicht anderweitig verwendet werden
- Pin's 34 .. 39 sind nur als Input-Pin's nutzbar (und haben auch keinen internen Pull-up-Widerstand)
- der Pull-Wert kann mittels Pin.PULL_HOLD auf einen anderen Wert gesetzt werden, z.B., um sie Strom-sparend im DeepSleep-Modus zu nutzen

meinpin = Pin(4, Pin.OUT)
setzt Pin 4 bei der Initialisierung als Ausgabe-Port

meinpin = Pin(5, Pin.OUT, value=1)
setzt Pin 3 bei der Initialisierung als Ausgabe-Port sofort auf HIGH

meinpin.on()
schaltet den Pin auf HIGH

meinpin.value(0 | 1)
schaltet den Pin auf LOW bzw. HIGH

meinpin.off()
schaltet den Pin auf LOW

meinpin = Pin(6, Pin.IN)

meinpin.value()
gibt 0 oder 1 (für LOW bzw HIGH) zurück

meinpin = Pin(7, Pin.IN, Pin.PULL_UP)

aktiviert den internen Pull-up-Widerstand

meinpin = Pin(8, Pin.OUT, value=1)

PWM (pulse width modulation)

from machine import Pin, PWM

pulsweitmod = PWM(Pin(0))

pulsweitmod.freq()

pulsweitmod.freq(500)

pulsweitmod.duty()

pulsweitmod.duty(100)

pulsweitmod.deinit()

pulsweitmod2 = PWM(Pin(9), freq=10000, duty=1000)
Pulsweiten-Ausgabe in einer Funktion initialisieren

es sind Frequenzen von 1 Hz bis 40 MHz möglich

ADC (analog to digital conversion)

from machine import ADC

analdigwand = ADC(Pin(32))
Eingangs-Spannungs-Pegel: 0 .. 1,0 V; Auflösung auf 12 bit → Ergebniswerte: 0 .. 4'095

analdigwand.read()

analdigwand.atten(ADC.ATTN_11DB)
Einschalten einer Dämpfung → Eingangs-Spannungs-Pegel: 0 .. 3,6 V
weitere zugelassene Dämpfungswerte: ATTN_0DB (bis 1,0 V), ATTN_2_5_DB (bis 1,34 V), ATTN_6DB (bis 2,0 V)

analdigwand.width(ADC.WIDTH_9BIT))
Ändern der Auflösung auf 9 bit → Ergebniswerte: 0 .. 511

weitere Auflösungen: WIDTH_10BIT (0 .. 1023), WIDTH_11BIT (0 .. 2047), WIDTH_12BIT (0 .. 4095)

Eingangs-Spannungen über 3,6 V können den Microcontroller zerstören!

SPI-Bus (serial peripheral interface)

Bus-System von Motorola
synchron, seriell, Master-Slave-System

from machine import Pin, SPI

spi=SPI(baudrate=100000, polarity=1, phase=0, sck= Pin(0), mosi=Pin(2), miso=Pin(4))
Initialisierung des SPI-Busses mit einen Signal-Übertragungs-Rate von 100'000 Bd
sck .. serial clock (Bus-Takt)
MOSI (seltener auch SIMO oder SDO (serial data out)) .. Master Output + Slave Input
MISO (seltener auch SOMI oder SDI (serial data input)) .. Master Input + Slave Output
SDO und SDI werden i.A. aus der Sicht des Device's benannt, d.h. die Leitungen müssen sich kreuzen
die Polarität und die Phase können die Werte 0 oder 1 annehmen und steuern Datenabnahme; Phase bestimmt welcher Flankenwechsel ausgewertet wird; die Polarität, ob die steigende (0) oder fallende Flanke (1) das Signal ist

spi.init(baudrate=200000)

spi.read(AnzahlBytes)

spi.read(AnzahlBytes, Adresse)
Adresse (von MOSI) z.B. 0xff

puffer = bytearray(100)
spi.readinto(puffer)

spi.readinto(puffer, Adresse)

spi.wirte(b'abcdef')
Schreiben von 6 Bytes an MOSI

puffer = bytearray(5)
spi.write_readinto(b'12345', puffer)
Schreiben der Bytes an MOSI und Lesen von MISO in den Puffer

spi.write_readinto(puffer, puffer)
Schreiben des Puffers an MOSI und Lesen von MISO in den Puffer

SPI-Hardware-Bus
beim ESP-32 sind zwei Hardware-Kanäle steuerbar
Pin's sind festgelegt

HSPI (id=1) → sck = 14, mosi = 13, miso = 12
HSPI (id=2) → sck = 18, mosi = 23, miso = 19

from machine import Pin, SPI

hspi =SPI(1, 100000, sck= Pin(14), mosi=Pin(13), miso=Pin(12))
vspi = SPI(2, baudrate = 100000, polarity=0, phase=0, bits=8, firstbit=0, sck=Pin(18), mosi=Pin(23), miso=Pin(19))

I2C-Bus

eigentlich I2C für inter-integrated circuit
serieller Daten-Bus von Philips
technisch identisch mit Two-Wire-Interface (von Amtel)
beides sind Zwei-Draht-Schnittstellen
praktisch 3 Leistungen: Betriebs-Spannung V_{DD} sowie zwei – über Pull-up-Widerständen angeschlossene Arbeits-Leitungen (Takt (SDL) und Daten (SDA))
gearbeitet wird mit positiver Logik (LOW = 0 (max. 0,3 V_{DD}) und HIGH = 1 (min. 0,7 V_{DD})
bei Daten-Übertragungen ist das 1. Byte die Slave-Adresse (die 7 niedrigen Bits bilden die eigentliche Adresse, das 8. Bit bestimmt, ob Slave Daten empfangen (0 / LOW) bzw. senden (1 / HIGH) soll
bestimmte Adressen sind für Sonderzwecke reserviert (insgesamt 112 Slave's ansprechbar)

from machine import Pin, I2C

i2c = I2C(scl = Pin(5), sla = Pin(4), freq = 100000)
i2c.readfrom(Adresse, AnzahlBytes)
Adresse z.B. 0x3a

i2c.writeto(Adresse, Wert)

puffer = bytearray(10)
i2c.writeto(Adresse, puffer)

OneWire-Treiber ()

from machine import Pin
import onewire

eindraht = onewire.OneWire(Pin(12))
aktiviert eine OneWire-Bus an der GPIO12

eindraht.scan()
Gibt eine Liste der gescannten Device's zurück

eindraht.reset()
Setzt den Bus zurück

eindraht.readbyte()
Ließt ein Byte vom Bus

eindraht.writebyte(Adresse)
Adresse könnte z.B. 0x12 sein

eindraht.write(Wert)
Wert wird als Bytes verstanden

eindraht.select_rom(b'12345678')

speziell für Temperatur-Sensoren DS18S20 und DS18B20

```
import time, ds18x20
ds = ds18x20.DS18x20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(1000)
für rom in roms:
   print("gemessene Temperatur: ",ds.read_temp(rom))
```

LED-Leisten bzw. -Ringe (NeoPixel)

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)
neopix = NeoPixel(pin, AnzahlLEDs))
```
erstellt eine NeoPixel-LED-Reihe an der GPIO0

```
neopix[0] = (255,255,255)
neopix.write()
```
Setzt die erste LED auf weiß
die wirkliche Anzeige / Ausgabe erfolgt erst mit .write()

```
r,g,b = neopix[0]
```
liefert die Farbwerte der ersten LED zurück

für ESP ist eine weitere Low-Level-Ansteuerung möglich:

```
import esp
esp.neopixel_write(pin, rgb_buf, is800khz)
```
800 kHz ist die Default-Einstellung, praktisch sind auch 400 kHz möglich (timing=0)

Touch-Eingabe (capacitive touch)

```
from machine import TouchPad, Pin

touch = TouchPad(Pin(14))
```
Aktivieren des Touch-Modus für GPIO14 (Touch6)

```
touch.read()
```

Auslesen des Touch-Pin's (gelesener Wert wird bei Berührung (deutlich) kleiner)

Benutzen der Touch-Eingänge für das Aufwecken aus dem Tiefschlaf-Modus
import machine
from machine import TouchPad, Pin
import esp32

touch = TouchPad(Pin(14)
touch.config(500)
esp32.wake_on_touch(True)
machine.lightsleep()
ESP wird in den Tiefschlaf-Modus versetzt, solange der Touch-Sensor an GPIO14 (Touch6) gedrückt ist

DHT (Umweltsensoren, Temperatur-Luftfeuchte-Sensor)

häufig genutzter Kombinations-Sensor DHT11 für Luftfeuchtigkeit (Humidity) und Temperatur
Sensor arbeit an allen Pin's

import dht
import machine

humtemp = dht.DHT11(machine.Pin(4))
Aktivieren des Sensors für GPIO4

humtemp.measure()
eine Messung abfragen

humtemp.temperature()
Gibt Temperatur in °C zurück

humtemp.humidity()
Gibt die relative Luftfeuchtigkeit in Prozent zurück

10.6.x.5.2. Modul "esp"

import esp

esp.osdebug(None)
Ausschalten der Debugging-Mitteilungen

esp.osdebug(0)
Umleiten der Debugging-Mitteilungen auf UART(0)

esp.flash_size()

esp.flash_user_start()

esp.flash_erase(SektorNummer)

esp.flash_write(ByteOffset, Puffer)

esp.flash_read(ByteOffset, Puffer)

10.6.x.5.3. Modul "esp32"

import esp32

esp32.hall_sensor()
Auslesen des (internen) Hall-Sensors

esp32.raw_temperature()
Auslesen des (CPU-internen) Temperatur-Sensors (Angabe in °F)

esp32.ULP()
Zugriff auf den ULP-Coprozessor (ultra low power; Stromspar-Coprozessor)

10.6.x.5.4. Modul "network"

import network

MeinWLAN = network.WLAN(network.STA_IF)
konfigurieren des (eigenen) WLAN-Moduls im Stations-Modus

MeinWLAN.active(True | False)
Ein- bzw. Aus-Schalten des WLAN-Moduls

MeinWLAN.scan()
Scannen des WLAN's nach AccessPoint's

MeinWLAN.isconnected()
Prüfen, ob Station mit dem AccessPoint verbunden ist

MeinWLAN.connect(SSID, Passwort)
Verbindung zum AccessPoint herstellen

MeinWLAN.config('mac')
gibt die MAC-Adresse zurück

MeinWLAN.ifconfig()
gibt die IP-Adresse, die Netzwerk-Maske, den Gateway und die DNS-Adresse zurück

MeinAccessPoint = network.WLAN(network.AP_IF)
konfigurieren des (eigenen) WLAN-Moduls als AccessPoint

MeinAccessPoint.config(essid=WLANName)
Festlegen des Namens für das WLAN

MeinAccessPoint. active(True | False)
Ein- bzw. Aus-Schalten des WLAN-Moduls

10.6.x.5.5. Modul "time"

import time

time.sleep(Sekunden)

time.sleep_ms(MilliSekunden)

time.sleep_us(MikroSekunden)

StartZeit = time.ticks_ms()
Laufzeit = time.ticks_diff(time.ticks_ms(), StartZeit)
Laufzeit ermitteln

time.sleep(Sekunden)

10.6.x.y. Sprach-Elemente vom MicroPython (Kurz-Übersicht / Spicker)

(formatierte) Ausgabe:

ausgabe:=wert | *berechnung* | *"Text"* | *'Text'*

```
print()
print(ausgabe)
```

Verzweigung:

```
if bedingung:              # Einleitung und Test/Bedingung
      befehle              #  Then-/Dann-/Wahr-Zweig  (eingerückt!!!  mehrzeilig
möglich)
{elif bedingung:           # zusätzliche(r) untergeordnete(r) Test/Bedingung
      befehle}             # untergeord. Then-/Dann-/Wahr-Zweig
[else:                     # optionaler Else-/Sonst-/Falsch/Rest-Zweig
      Befehle]
```

Schleifen:

```
while bedingung:           # while True:    # Endlosschleife
                                            # (meist break notwendig)
      befehle
      {continue}           # Sprung zum nächsten Schleifendurchlauf /-anfang
      {befehle
      break}               # Sprung hinter Schleife (noch hinter ELSE)
{else:
      befehle}

for laufvariable in liste|tupel:  # _ als laufvariable, wenn kein Gebrauch in
      befehle                                          Schleife geplant
      [verzweigung : break]            # vorzeitiger Abbruch der Schleife

for laufvariable in range([untere_grenze, ]obere_grenze[, schrittweite]):
      befehle
```

erweiterter Spicker für das "normale" Python (→ Python-Spicker)

10.7. Python auf und mit Taschenrechnern

10.7.x. Texas Instruments-Rechner

11. Üben, üben und nochmals üben

hier folgen Aufgaben unterschiedlichster Schwierigkeitsgrade und Komplexitäten
keine Abfolge, wie im Skript
einfache Sammlung verschiedener – in diversen Quellen gefundener – Aufgabenstellungen oder (informatischer) Probleme

nicht täuschen lassen, Aufgaben, die leicht oder einfach zu lösen scheinen, können sich als echte Programmier-Diamanten herausstellen. Dagegen können Aufgaben mit seitenlangen Aufgabenstellungen mit ein paar Zeilen Quelltext erschlagen werden. Man erinnere sich an die Wundertüte Rekursion (→ 8.4.2. Rekursion)

im Allgemeinen hilft nur probieren
eine Lösung die ich oder ein anderer als leicht einschätze, kann für jemanden Anderes ein unlösbares Problem sein, aber es geht natürlich auch anders herum. So manche Aufgabe, für die ich viele Zeilen Quelltext brauche erledigt ein findiger Programmierer mit genial wenigen Zeilen. So ist die Welt, und das ist gut so!

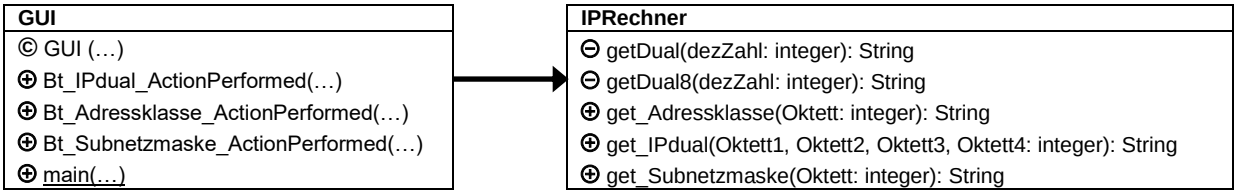
nicht unendlich in eine Aufgabe reinsteigern; Grenzen setzen; auf das Wesentliche konzentrieren
gibt es scheinbar unlösbare Hindernisse / Probleme, dann Problem / Sachverhalt (z.B. im Quelle-Text) kurz notieren; dann erst mal eine andere Aufgabe (zum Ablenken) erledigen

11.x. Aufgaben aus der Abiturprüfung Informatik MV

aus rechtlichen Gründen wurden die Formulierungen der Aufgabenstellungen geändert
die eigentliche Aufgabenstellung bleibt aber erhalten

11.x.y. Abitur 2010

Rechner hat 177.122.66.99/16



Klasse IPRechner

11.x. Aufgaben der Landesolympiade Informatik MV

aus rechtlichen Gründen wurden die Formulierungen der Aufgabenstellungen geändert
die eigentliche Aufgabenstellung bleibt aber erhalten

11.x.y. 2014/2015

11.x.y.z. Sekundarstufe II

Literatur und Quellen:

/1/ SANDE, Warren D.; SANDE, Carter:
Hello World! – Programmieren für Kids und Anfänger.-München: C. Hanser Verl..-2., akt. u. erw. Aufl.
ISBN 978-3-446-43906-4

/2/ LINGL, Gregor:
Python für Kids.-Heidelberg, München, Landsberg, Frechen, Hamburg: bhv Verl. / mitp Verl.-4. Aufl.
ISBN 978-3-8266-8673-3

/3/ WEIGEND, Michael:
Python 3 – Lernen und professionell anwenden.- Heidelberg, München, Landsberg, Frechen, Hamburg: mitp Verl.-5., akt. Aufl.
ISBN 978-3-8266-9456-1

/4/ ARNHOLD, Werner:
Lieben Sie PYTHON?-IN: LOG IN, 21(2001) Heft 2.-S. 18 ff.-Berlin: LOG IN Verl.
ISSN 0720-8642

/5/ MONK, Simon:
Raspberry Pi programmieren – Alle Befehle, und es klappt mit dem Raspberry.-Haar bei München: Franzis Verl.; 2014
ISBN 978-3-645-60261-7
auch sonst als reine Python-Einführung sehr empfehlenswert

/6/ VON LÖWIS, Martin; FISCHBECK, Nils:
Das Python-Buch – Referenz der objektorientierten Skriptsprache für GUIs und Netzwerke.-Bonn: Addison-Wesley-Verl., 1997.- 1. Aufl.
ISBN 3-8273-1110-1

/7/ ERNESTI, Johannes; KAISER, Peter:
Python 3 – Das umfassende Handbuch.-: Rheinwerk Verl..- 4. Aufl. 2015
ISBN 978-3-8362-3633-1

/8/ :
.-: Verl..- Aufl.
ISBN 978-3-

/8/ :
.-: Verl..- Aufl.
ISBN 978-3-

/8/ :
.-: Verl..- Aufl.
ISBN 978-3-

/A/ Wikipedia
http://de.wikipedia.org

Die originalen sowie detailliertere bibliographische Angaben zu den meisten Literaturquellen sind im Internet unter http://dnb.ddb.de zu finden.

Abbildungen und Skizzen entstammen den folgende ClipArt-Sammlungen:

/A/

andere Quellen sind direkt angegeben.

Alle anderen Abbildungen sind geistiges Eigentum:

/I/ lern-soft-projekt: drews (c,p) 1997-2020 lsp: dre
für die Verwendung außerhalb dieses Skriptes gilt für sie die Lizenz:

CC-BY-NC-SA

Lizenz-Erklärungen und –Bedingungen: http://de.creativecommons.org/was-ist-cc/
andere Verwendungen nur mit schriftlicher Vereinbarung!!!

verwendete freie Software:

- **Inkscape** von: inkscape.org (www.inkscape.org)
- **CmapTools** von: Institute for Human and Maschine Cognition (www.ihmc.us)

⊞- (c,p) 2015 - 2020 lern-soft-projekt: drews -⊟
⊞- 18069 Rostock; Luise-Otto-Peters-Ring 25 -⊟
⊞- Tel/AB (0381) 760 12 18 FAX 760 12 11 -⊟