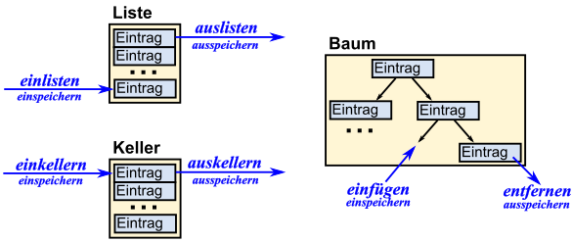
Informatik

für die Sekundarstufe I + II

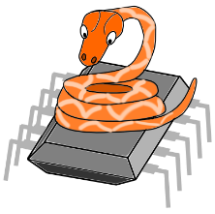
- Programmieren mit Python – Teil 1: für Einsteiger

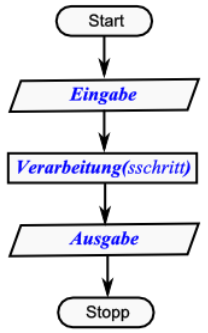
Autor: L. Drews

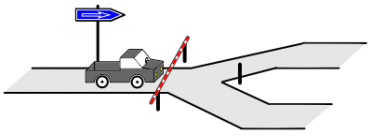


Grüner Baum-Python
(s) Morelia viridis
Q: de.wikipedia.org (Mwx)

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



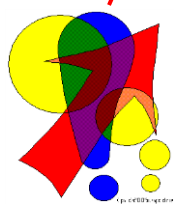




teilredigierte Version 0.11e (2025)

(c,p) 2015 - 2025 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.

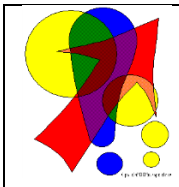
(7) Der Autor behält sich das Recht vor, diese Bestimmungen zu ändern.

(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	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 6

1. Einstieg und Grundlagen 9

1.1. Geschichte und Namensgebung 9

1.2. Warum Python? 10

grundlegende Python-Konzepte 14

2. Vorbereitung (Installation) 16

2.1. Python auf Windows-Rechnern 16

2.2. Python auf Linux-Rechnern 18

2.3. Python auf dem Raspberry Pi 18

2.5. Python auf Android-Systemen 18

2.5.1. Pydroid3 IDE 18

2.6. Python auf dem MacOS 19

2.7. Python auf dem Taschenrechner 19

2.8. Python auf Microcontrollern 20

2.9. Python online (ausprobieren) 21

3. Zugriff auf das Python-System 22

3.1. die Python-Shell 22

3.1.1. Eingaben an der Shell 23

3.1.2. IDLE als Python-Konsole 23

3.1.2. fortgeschrittene Mathematik 25

3.1.3. mehrzeilige Eingaben an der Shell 26

3.1.4. mehrere Befehle in eine Zeile 27

3.1.2.1. Mathematik für Informatiker – binäres Rechnen 28

3.1.3. Eingaben und Daten merken - Variablen 31

3.1.3.1. besondere Variablen und spezielle Möglichkeiten für Variablen in Python 33

3.2. Arbeiten mit Scripten 36

3.2.1. Grundlagen DOS bzw. Komandozeile (Eingabeaufforderung, Terminal) 36

3.2.2. Aufruf fertiger Python-Skripte 37

3.3. die interne Benutzer-Oberfläche 39

3.3.x. Hilfe(n)! 39

3.4. Nutzung anderer Benutzer-Oberflächen 41

3.4.1. gut geeignete Editoren für die Verwendung mit Python 41

3.4.1.1. Sublime Text 41

3.4.1.2. Geany 42

3.4.1.3. Notepad++ 42

3.4.1.4. Komodo Edit 42

3.4.x. Eclipse 43

3.4.x. Spyder 44

3.4.x. LiClipse 45

3.4.x. Anaconda 45

3.4.x. WinPython 45

3.4.x. Komodo IDE 46

3.4.x. Thonny 46

3.4.x. SciTE 46

3.4.x. TigerJython 47

3.4.x. Editoren im Internet – online-Editoren 48

3.4.x.1. w3schools.com 48

3.4.x.2. TigerJython 48

3.4.x.3. repl.it 49

3.4.x.3. ??? ... 49
3.4.x. microsoft Visual Studio Code mit Jupyter-Erweiterung ... 50
3.5. Snap for Python ... 51
Windows ... 51
Linux ... 51
MacOS ... 51
4. erste einfache Programme mit Python ... 52
4.1. Kommentare ... 55
4.2. Planung eines Programms und Umsetzung in Python ... 56
ergänzende Bemerkungen zu Vaiablen und Daten-Typen ... 61
5. Was passiert mit dem Quelltext? ... 62
5.1. Und es geht doch! – aus dem Python-Quelltext eine EXE erstellen ... 65
5.2. Fehlersuche ... 66
5.3. Stil-Regeln für Python-Programmierer ... 70
Linter 72
5.4. agile Software-Entwicklung ... 72
6. grundlegende Sprach-Elemente von Python ... 74
6.1. Ausgaben ... 74
6.1.1. Anpassen von Zahlen für Ausgaben ... 79
6.1.2. formatierte Ausgaben ... 80
6.1.2.1. formatierte Ausgaben mit der format-Funktion ... 81
6.1.2.2. Verwendung von Platzhaltern in Ausgabetexten ... 83
6.1.2.3. Kombination von Platzhaltern und format-Funktion ... 84
6.2. Eingaben ... 86
6.2.1. unschöne Eingabe-Effekte in Python-Programmen ... 88
6.3. Verarbeitung ... 91
Operatoren ... 93
6.4. Kontrolle(n) ... 96
6.4.1. Verzweigungen ... 97
6.4.1.1. einfache Verzeigungen ... 97
einseitige Auswahl / bedingte Ausführung ... 97
zweiseitige Auswahl / vollständige Verzweigung ... 100
6.4.1.2. geschachtelte Alternativen ... 107
6.4.1.3. Mehrfach-Verzeigungen ... 110
6.4.1.4. Optimierung des Quellcode's – DRY- und EVA-Prinzip ... 114
6.4.2. Schleifen ... 119
6.4.2.1. bedingte Schleifen ... 120
Berechnung der Kreiszahl Pi mit der Methode von ACHIMEDES ... 124
Berechnung der Quadratwurzel von x nach der Formel von HERON ... 126
Berechnung der n-ten Wurzel ... 127
Eingabe-Kontrolle mittels Schleife ... 127
Fehler-Analyse in Schleifen ... 128
6.4.2.2. Sammlungs-bedingte Schleifen ... 132
6.4.2.3. Zähl-Schleifen ... 138
6.4.2.4. besondere Kontrollstrukturen in Schleifen ... 140
6.4.2.5. Und was ist mit nachprüfenden / Fuß-gesteuerten Schleifen? ... 143
6.4.2.6. Anwendungs-Beispiel: lineare Regression ... 147
Beispiel für Daten in zwei Listen ... 147
6.5. Unterprogramme, Funktionen usw. usf. ... 149
6.5.1. Allgemeines zu Funktionen in Python ... 151
6.5.2. Funktionen ohne Rückgabewerte ... 153
6.5.3. echte Funktionen – Funktionen mit Rückgabewerten ... 155
6.5.4. Funktionen mit Standard-Werten als Parameter ... 156

6.5.5. Funktionen mit einer variablen Anzahl von Parametern 157
6.5.6. Funktionen mit Funktionen als Parameter 157
6.5.7. Generator-Funktionen – Funktionswerte schrittweise 158
6.5.8. Interator-Funktionen – Funktionswerte noch wieder anders 160
6.6. Vektoren, Felder und Tabellen 162
6.6.1. Felder mit unterschiedlichen Datentypen 166
6.7. ein bisschen Statistik 167
6.7.1. Zufallszahlen 167
6.8. die Python-Schlüsselwörter im Überblick 172
Python-Spicker 178
Eingabe: 178
(formatierte) Ausgabe: 178
Verzweigung: 178
Schleifen: 178
Funktion: 179
Bibliotheken: 179
Objekt / Klasse: 179
Literatur und Quellen: 180

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
https://cscircles.cemc.uwaterloo.ca/using-website-de/

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

Warum Programmieren?

Steuern von Geräten, Computern usw. usf.
Entwickeln von Spielen, …

aber Programmieren und Programmieren Lernen erfüllt auch andere wichtige Funktionen:

man versteht, wie Programme und damit auch Computer usw. gerade so funktionieren
eigene Projekt-Ideen umsetzen
einfach nur Lernen wie man Probleme – ev. auch im Team – Lösen kann

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

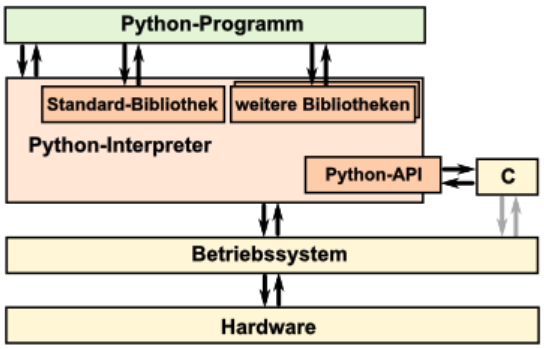
nicht ganz übliche / moderne Programm-Strukturen, wenn objektorientiert programmiert werden soll

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

(c,p) 2015 - 2025 lsp: dre

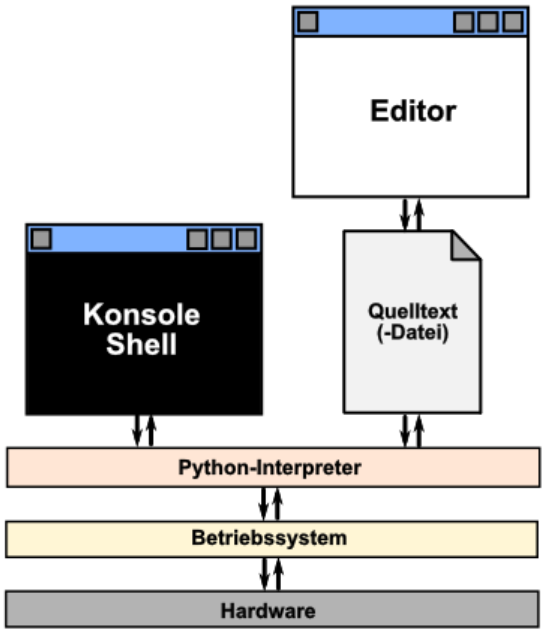
grundlegende Python-Konzepte

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.

Auf → https://portablepython.com/wiki/Download/ gibt es ebenfalls eine Python-Distribution, die sich für den portablen Einsatz eignet. Besonders hervorzuhenem sind die vielen – schon integrierten – Bibliotheken. Viele davon sind für den erweiterten schulischen Einsatz sehr wichtig. Ich denke dabei z.B. an **NymPy** (→ 8.6.4. Modul / Bibliothek NumPy) und **Matplotlib** (→ 8.6.5. Modul / Bibliothek Matplotlib). Hier die Liste der integrierten Bibliotheken:

- PyScripter
- NymPy
- SciPy
- Matplotlib
- PyWin32
- NetworkX
- Lxml
- PySerial
- PyODBC
- PyQt
- IPython
- Pandas

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.

2.6. Python auf dem MacOS

Python3IDE

benötigt neue(ste) OS-Version

für einfache Programmier-Versuche unterwegs ausreichend

hier soll auch noch mal auf die Besonderheiten der Mac-Tastatur hingewiesen werden

die zusätzlichen Graphik-Zeichen erhält man über die [option]-Taste (statt [Alt Gr] von Standard-Tastaturen)

Installations-Anleitung:
neueste Version von der Python-Seite herunterladen (→ https://www.python.org/downloads/)
die heruntergeladene .pkg-Datei öffnen
den Anweisungen folgen

Installation eines sehr guten Text-Editor's (Sublime Text)
herunterladen der aktuellen Version von (→ https://www.sublimetext.com/3)
installieren ???

wenn die Text-Dateien ordnungsgemäß mit .py-Dateiendung versehen wird, dann kann das Programm von der Konsole, aus dem Datei-Manager oder direkt in Sublime-Text über den Menü-Eintrag "Build" gestartet werden

2.7. Python auf dem Taschenrechner

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

Umsetzung von MicroPython

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

Auch für einige Taschen- und CAS-Rechner von Texas Instruments steht ein aktuelles / aktualisertes Betriebssystem mit Python zur Verfügung. Es handelt sich ebenfalls um ein MicroPython.

auch für die Steuerung des TI-Innovator's und des TI-Rover benutzbar
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.))

2.8. 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.

2.9. Python online (ausprobieren)

→ https://repl.it/

→ http://pythonfiddle.com/

repl.i

→ https://www.programiz.com/python-programming/online-compiler/

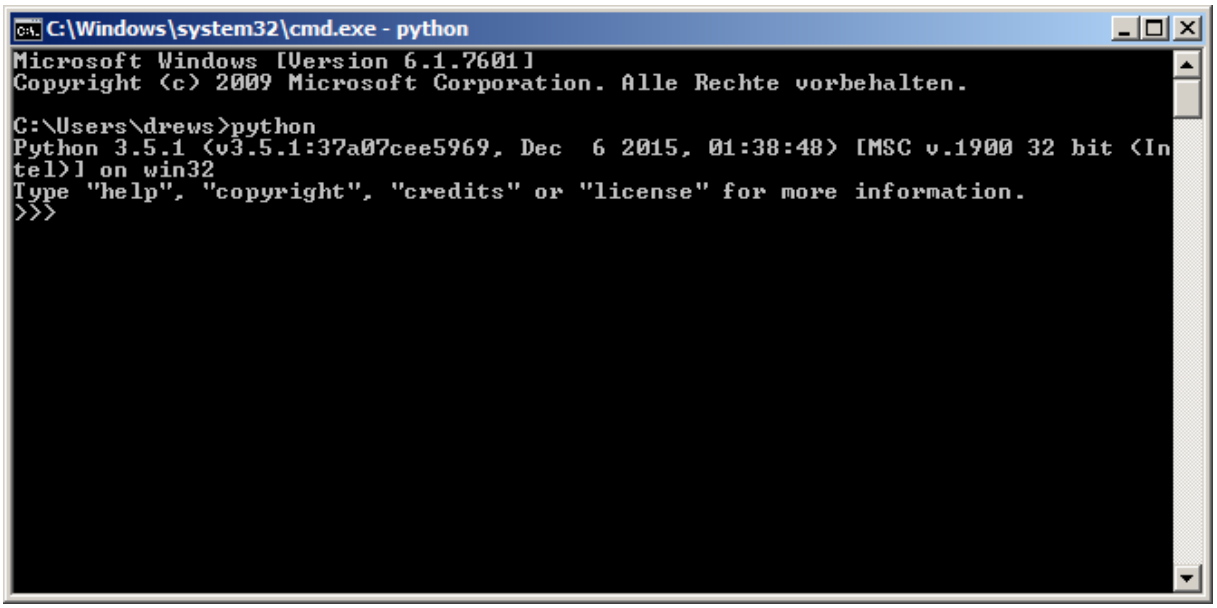
???
→ http://pythontutor.com/

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

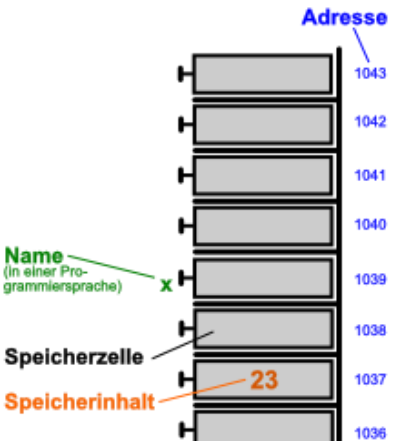
2. ***Erkunden Sie, was die Operatoren &&, || und ! bewirken! Arbeiten Sie dazu mit kleinen Zahlen und stellen Sie sich die Eingaben und Ausgaben in Ihren Mitschriften binär (untereinander) dar! (&& und || sind zweistellige, innere Operatoren; ! ist ein einstelliger Präfix-Operator)***
3. ***Probieren 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 beschrifte "Behälter" oder "Container" in denen etwas aufbewahrt wird. In der Informatik heißen Variablen exakt Bezeichner. Mitlerweile versteht man Variablen auch als Objekte.
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.

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

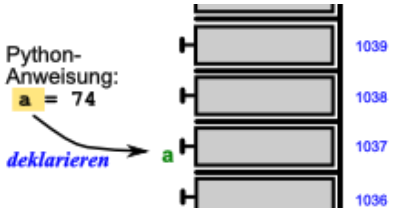
Natürlich gibt es Variablen auch in Python. Jeder Variable muss zuerst einmal ein Name zugeordnet werden. Anders als in verschiedenen anderen Programmiersprachen braucht man die Variablen vorher zu deklarieren (definieren). D.h. man muss nicht vorher sagen bzw. zuerst festlegen, was man in der Variable 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.



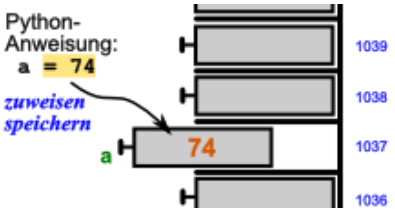
Python gibt bestimmten Speicherzellen nun den internen Variablen-Namen und bei der Wertzuweisung mit "=" wird festgelegt, welche Zahl oder welcher Text in die Schublade getan werden soll (Initialisierung). Besser spricht man statt "ist gleich" bei einer Wertzuweisung von "ergibt sich aus"! Das trifft 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.



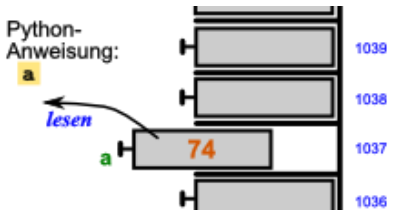
Nun wird der zweite Teil der Anweisung ausgeführt. In die Speicherzelle a wird der Wert 74 eingespeichert. Man spricht auch von einer Zuweisung.
Im Fall des Hauptspeichers (RAM's) bleibt dieser Inhalt nun solange erhalten, wie der Speicher mit Strom versorgt wird oder in die Speicherzelle ein neuer Wert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



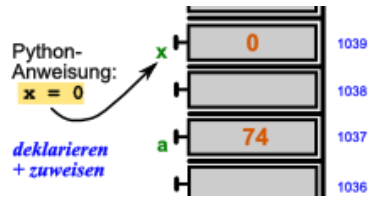
Unter bestimmten Bedingungen sorgt das Python-System dafür, dass nicht mehr gebrauchte Variablen aus dem Speicher entfernt werden. Praktisch wird auch nur der Name aus der Namensliste gestrichen, so dass hierüber kein Zugriff mehr erfolgen kann. Der Inhalt der Speicherzelle bleibt erhalten, ist aber 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
>>>
```

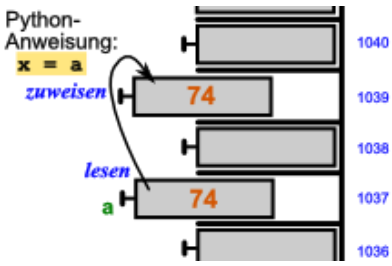
Mit der nebenstehenden Anweisung wird die Variable x im Speicher angelegt und ihr der Wert 0 zugewiesen.
Solcheine initiale Belegung (Anfangsbelegung) sollte man sich für jede Variable angewöhnen.

```
>>> 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
```

In Python können die "Schubladen" unterschiedlich groß sein. Darum kümmert sich Python selbst. Zahlen brauchen meist weniger Platz als Texte. Jeder Buchstabe bzw. jedes Zeichen in einem Text benötigt z.B. mindestens 1 Byte (in modernen Rechnern auch schnell 2 Byte). Der Speicher-Bedarf für Zahlen wird später thematisiert (→ 8.2 Datentypen und Typumwandlungen).
Bei großen Daten-Mengen muss der Speicher-Bedarf beachtet werden. Ansonsten kann es zu Programm-Abbrüchen kommen.

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)

7

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 ergibt sich aus 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:

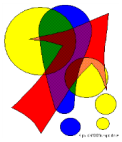
1.

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:

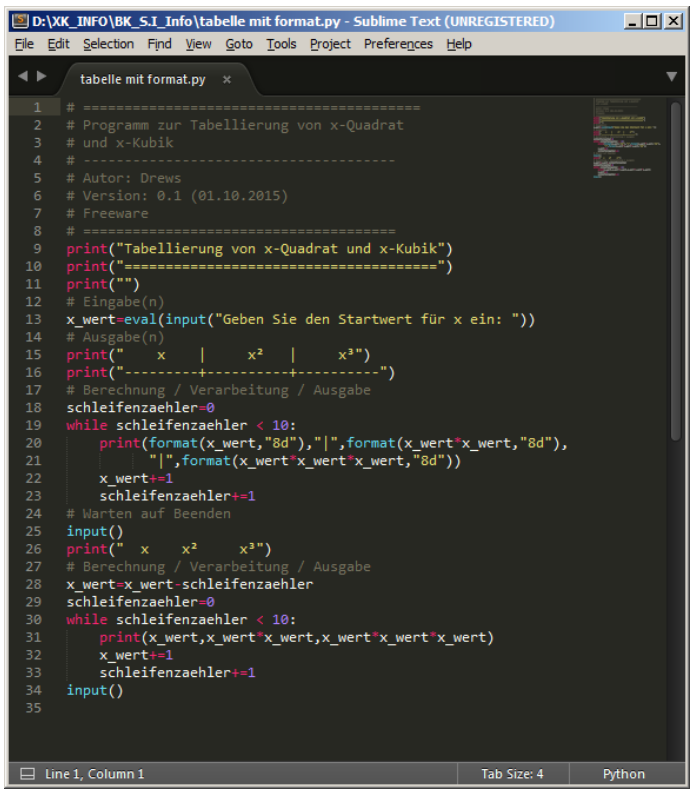
Never touch a running system.

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

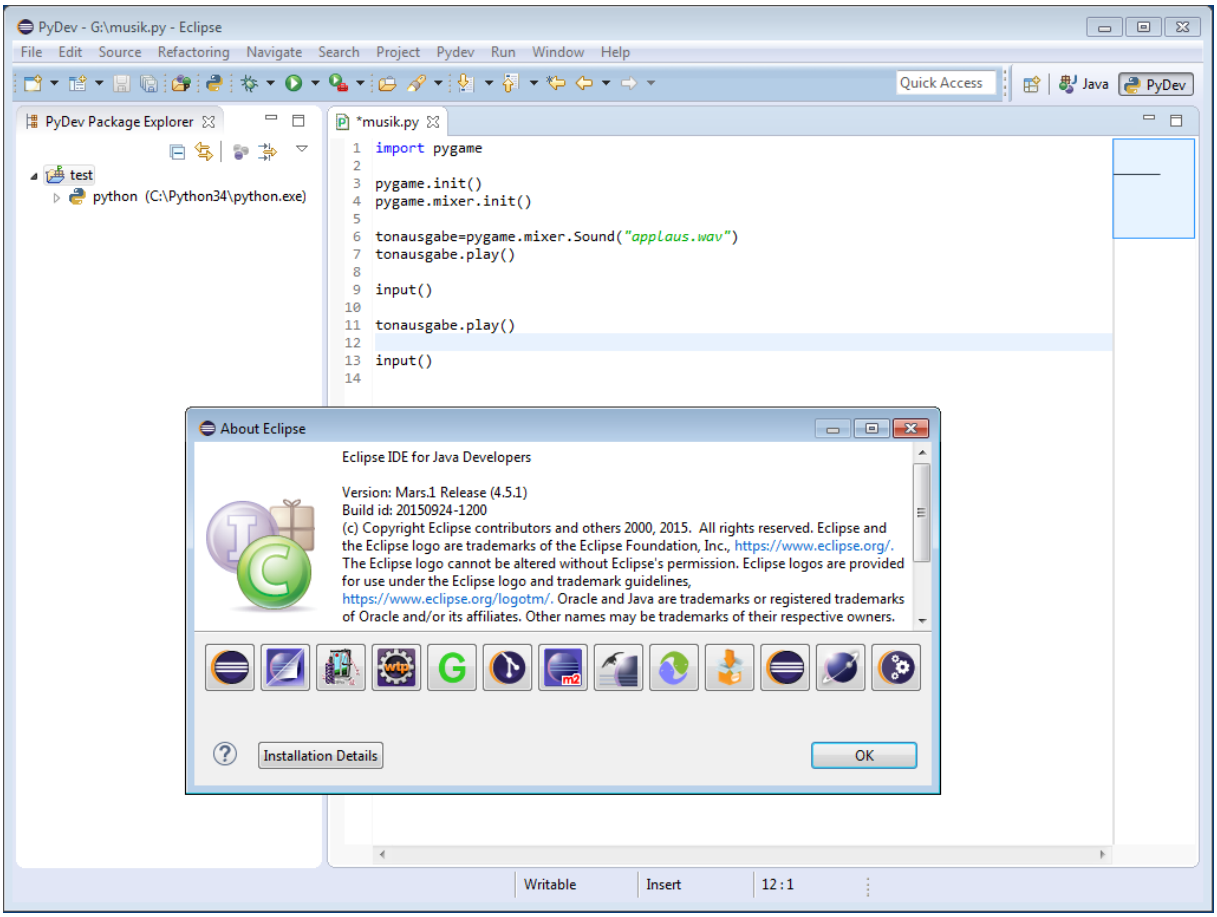
kein Debugger

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



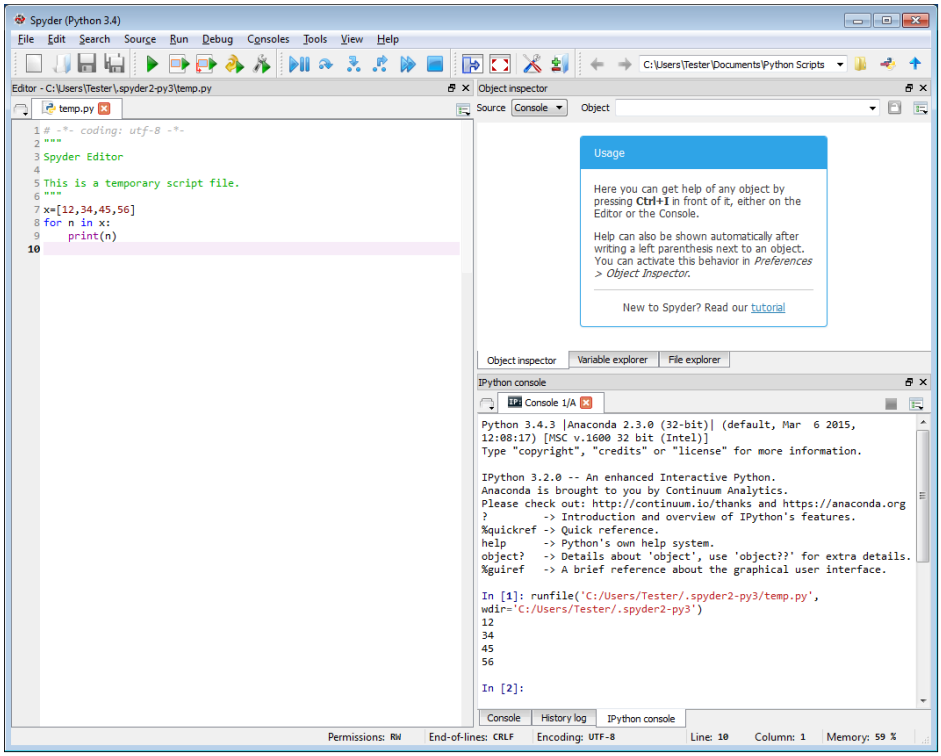
Eclipse in der Version Mars1 mit installiertem PyDev

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.

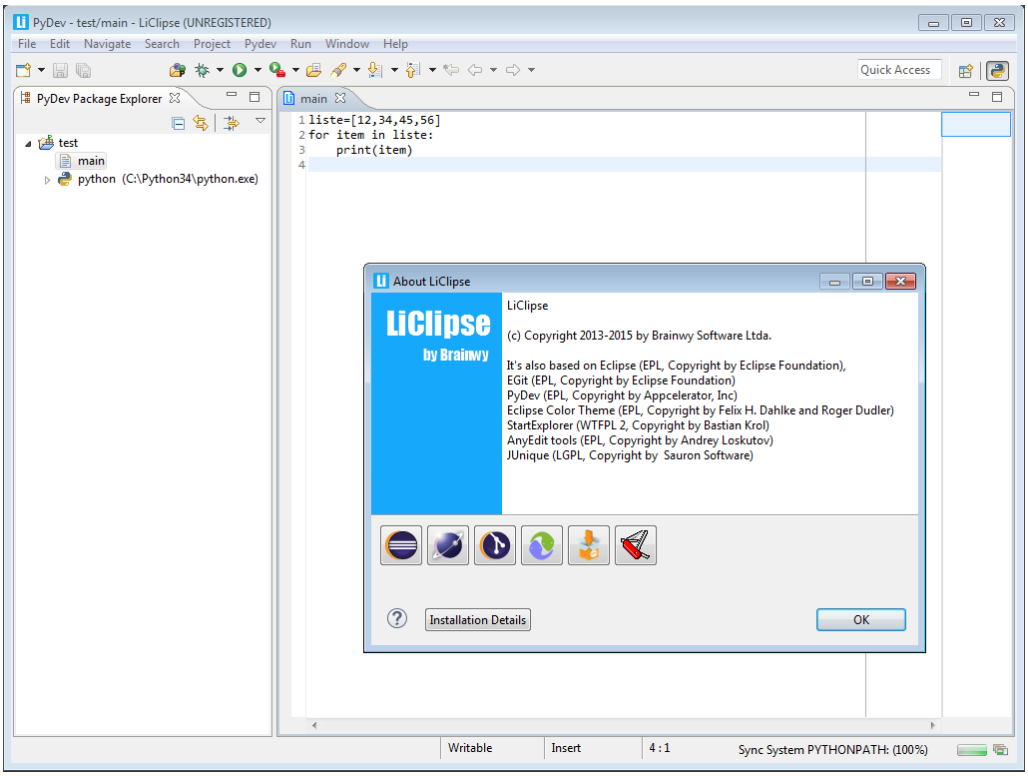
Links:
https://www1.ethz.ch/foss/news/course_python/configEclipse

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

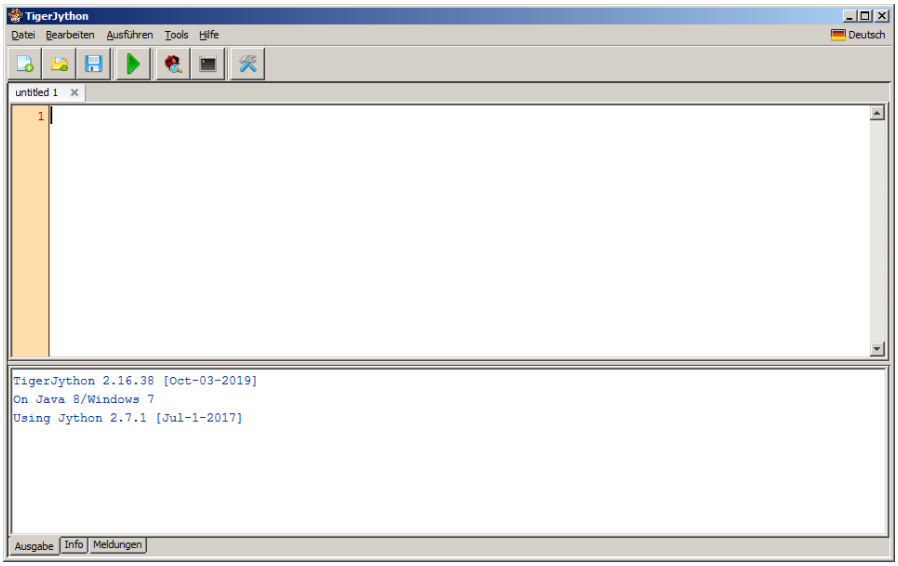
http://thonny.org/
für Windows, Mac und Linux

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

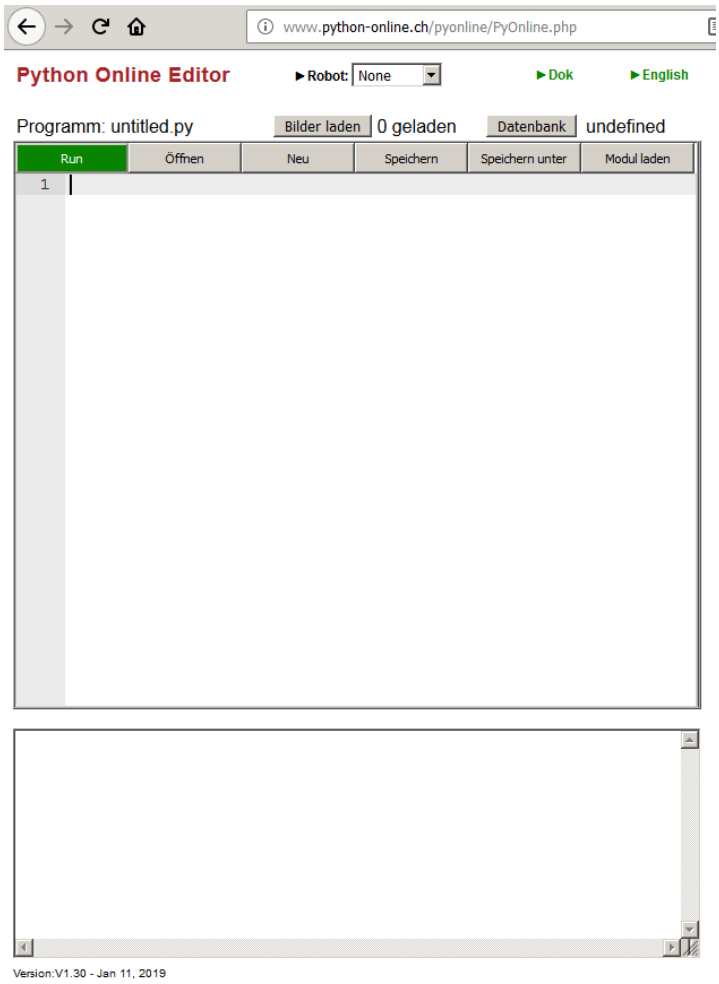
eine der besten Seiten; natürlich / leider alles englisch
https://www.w3schools.com/ (sehr viele Sprachen, …) ! keine Anmeldung notwendig

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. ???

3.4.x. microsoft Visual Studio Code mit Jupyter-Erweiterung

etwas aufwändiges Konstrukt aus Editor (ms Visual Studio Code) und einem Desktop-Dialog und –Dokumentations-System (Jupyter)

Nachteile

- sehr gewaltig und unübersichtlich für Anfänger
- lenkt von der eigentlichen Programmierung ab
- wenn Probleme auftauchen, ist die Ursache nicht immer der eigenen programmierung zuzuordnen; es gibt weitere – zusätzliche – Fehlerquellen
- …

bietet viele Vorteile

- sehr guter und flexibler Editor
- Dialog-System (immer Wechsel zwischen Eingabe und Ausgabe) → Konsolen-bzw. Skript-orientiertes Arbeiten
- dazu lassen sich Dokumentations-Abschnitte einfügen (Erklärungen usw. usf.)
- leichter Umstieg auf andere Programmiersprachen (die fast alle mit ms VSC editierbar sind)
- …

für fortgeschrittene Nutzer aber ohne weiteres empfehlenswert
sehr gut geeignet für Dialog- und Einstellungs-orientierte Sequenzen von Befehls-Ketten (z.B. bei der Erstellung / bearbeitung von Künstlichen Netzen / Systemen zum Maschinellen Lernen / …

bei **microsoft learn** gibt es dazu einen Kurs:
https://learn.microsoft.com/de-de/training/modules/

praktisches Vorgehen / Handling stellen wir im Skript-Teil 2 "Python für Fortgeschrittene" vor (→ 8.22.2. Jupyter-Erweiterung in microsoft Visual Studio Code)

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

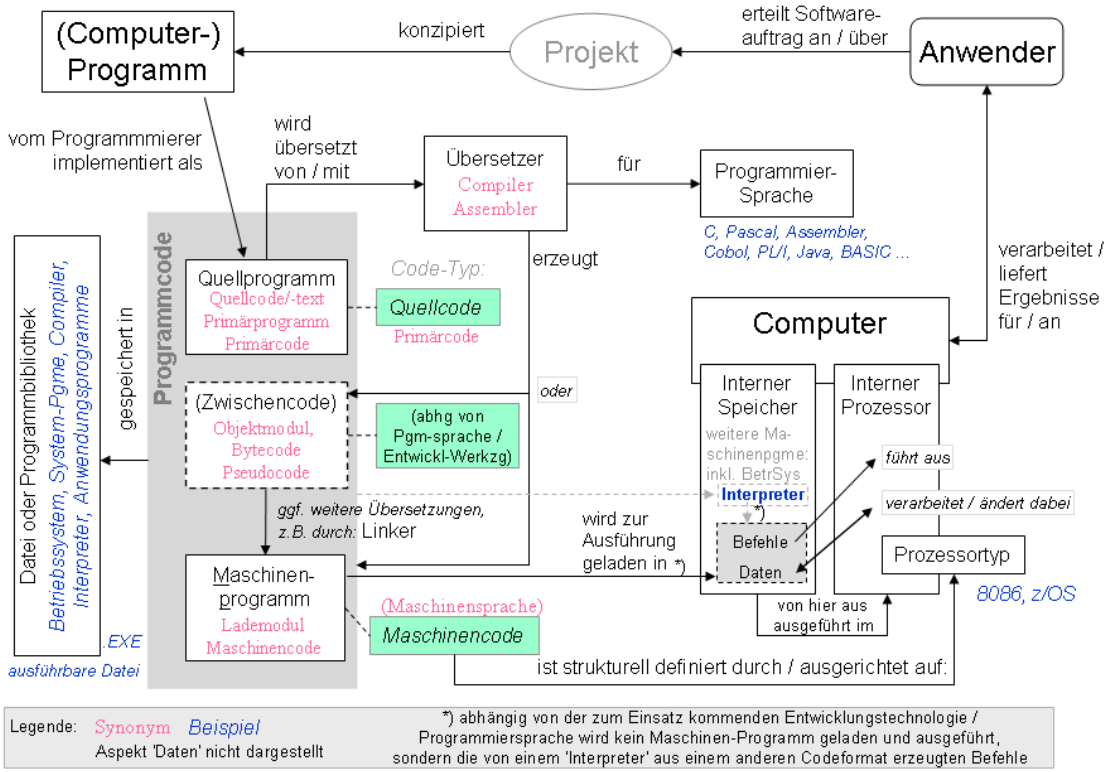
Gute Programmierer bauen diese Struktur auch im Quelltext ihrer Programme nach.
Der Ablauf eines Programms sollte immer vorgeplant werden. Zumindestens bei etwas komplizierteren Programmen kommt man dann nicht mehr ohne Vorplanung aus. U.U. werden bei größeren Programm-Projekten nur bestimmte kritische 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.

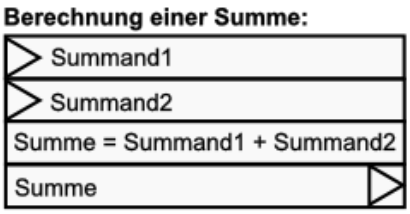
```
…
# Ausgabe(n)
for i in range(4):
    print("")
…
```

```
…
""" 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

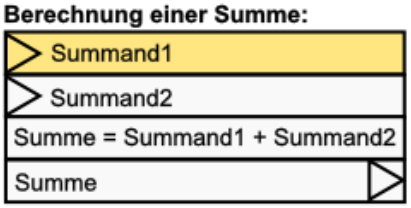
…	… Python

	`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.

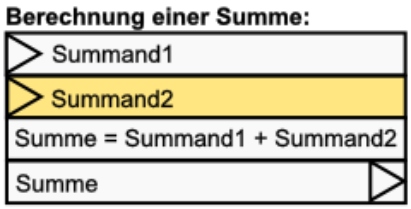
```
…
# Eingabe(n)
summand1 = input("Geben Sie den ersten Summanden ein: ")
…
```

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:
```

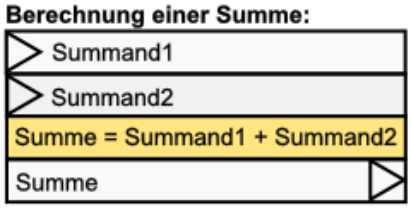
Im Fehlerfall müssen wir wieder zum Quelltext wechsel und die Fehler beseitigen. Dann wird wieder gespeichert und ausprobiert. Dieses muss man solange wiederholen, bis dieser Teil des Programms funktioniert.

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.

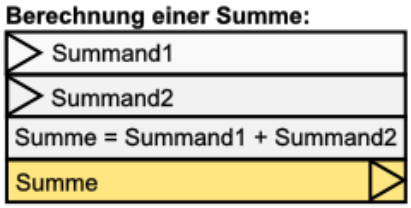
Erfahrene Programmierer geben natürlich gleich mehrere Zeilen ein und testen dann.

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!

ergänzende Bemerkungen zu Vaiablen und Daten-Typen

int mit einem Werte-Bereich von -9’223’372’036’854’775’808 bis 9’223’372’036’854’775’807 (-9 Trillionen bis 9 Trillionen (also knapp von -10^{18} bis 10^{18}))
Das entspricht dem Maximum, was in einer 64bit-Variablen möglich ist

float für Gleitkommazahlen ebenfalls als 64bit-Variable
durch spezielle Verteilung der Bit’s für Matisse und Exponent kommt man auf einen möglichen Bereich von $-1{,}797’693’134’862’315’7 * 10^{-308}$ bis $+2{,}225’073’858’507’201’4 * 10^{308}$

komplexe Zahlen lassen sich als Summe (besser auch in Klammern) aus reelen und imaginären Teil zusammensetzen 4+5j

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

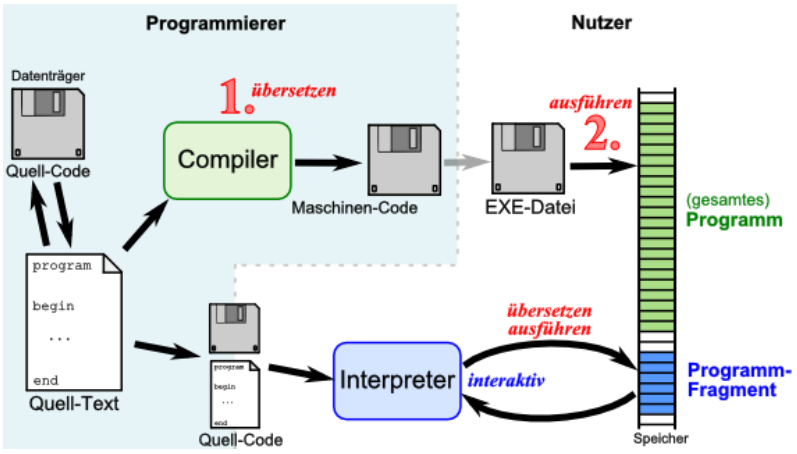
Übersetzung einer Programmiersprache in Maschinen-Code kann auf zwei Arten erfolgen:
entweder mit

- **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

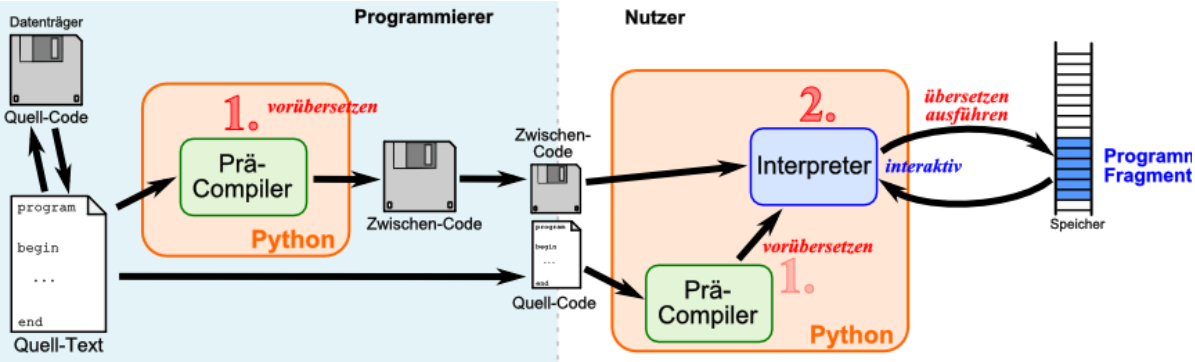
- **Compiler** — Übersetzung erfolgt hier in einem Stück vor der Nutzung; es wird zumeist ein echtes ausführbares Programm (EXE-Datei) erzeugt; das ausführbare Programm kann beliebig oft und ev. auch parallel ausgeführt werden; Nutzung ohne den Quell-Text und den Compiler möglich

Der Compiler nimmt den gesamten Quell-Text und übersetzt ihn in Maschinen-Code. Dieser wird dann in eine ausführbare Datei (i.A. eine EXE) gespeichert und kann dann beliebig oft ausgeführt werden. Meist kann die EXE-Datei auch weitergegeben werden und auf einem anderen Rechner ausgeführt werden. Dieser Rechner benötigt kein Übersetzungs-Programm, da der Quell-Text ja vollständig in Maschinen-Code übertragen wurde.



Compilierte Programme sind sehr schnell, da eine erneute Überprüfung und Übersetzung nicht mehr notwendig ist.
Ein Interpreter geht bei der Übersetzung anders vor. Er übersetzt immer nur zusammengehörende Teile des Quell-Code's und führt diese sofort aus.
Der Vorteil ist hier, dass eine ev. aufwändige Compilierung eines (großen) Programm's nicht erfolgen muss – es wird nur der derezit gebrauchte Teil übersetzt. Geht etwas beim Übersetzen oder in der Nutzung (z.B. Bedienfehler) schief, dann erfolgt eine Fehler-Meldung. Dies kann sofort im Quell-Text verbessert werden und eine erneute Interpretation erfolgen.
Interpreter sind auch wesenltliche einfacher zu erstellen und deutlich kleinere Programme, als Compiler.

Der Nutzer benötigt aber bei der Interpreter-Version auch immer wieder dieses Übersetzungs-Programm (also den Interpreter) auf seinem Rechner.
Prinzipiell ist Python eine Interpreter-Sprache. Der Quell-Text wird also während der Benutzung / dem Aufruf in Maschinen-Befehle umgesetzt. Aber beim genauen Betrachten wird allerdings eine Kombination aus Compiler und Interpreter benutzt. Dadurch werden die Vorteile beider Übersetzungs-Techniken zusammengeführt.



Der Quell-Text wird vom Compiler-Teil des "Interpreter's" zuerst in einen Zwischen-Code (Byte-Code) übersetzt. Dieser Code ist dann vom Interpreter lesbar. DerByte-Code selbst ist Maschinen-unabhängig, d.h. jeder Compiler erzeugt den gleichen Byte-Code aus einem Quell-Text. Der Byte-Code wird im Hintergrund verarbeitet. Der "normale" Nutzer übersieht diesen Code wahrscheinlich.
Der Zwischen-Code (Byte-Code) wird dann vom Interpreter, der nun als virtuelle Maschine fungiert, Maschinen-abhängig abgearbeitet. Für jeden Rechner mit einem anderen Betriebssystem muss es also von Python einen speziellen Interpreter geben.
Im Detail werden die Daten / Dateien in etwa so verarbeitet. Wenn der ***py***-Quelltext eines Moduls oder aus einer anderen py-Datei importiert wird, dann wird eine (sichtbare) Bytecode-Datei abgelegt. Diese hat die Endung ****.pyc***. Zumeist liegen die Dateien im Unterordner **__pycache__** des Python-Systems. Die pyc-Datei kann mit dem Interpreter jedes anderen Systems abgearbeitet werden.

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)

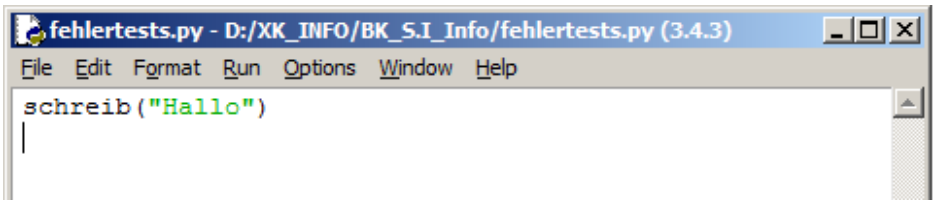
5.1. Und es geht doch! – aus dem Python-Quelltext eine EXE erstellen

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



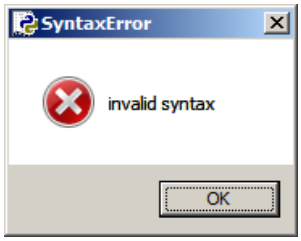
zu interpretierender Quelltext

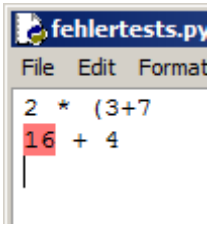
Gleich bei den ersten Übungen tauchen erfahrungsgemäß die ersten Fehler beim Interpretieren des Quellcodes auf.
Manche Fehler sind offensichtlich. Vor allem dann, wenn noch Fehler-Informationen mit ausgegeben werden.

```
>>>
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

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?***

Bei fehlerhaften Python-Programmen werden häufig die folgenden Fehler angezeigt:

- **SyntaxError** besagt, das Elemente von Phyton nicht richtig geschrieben oder angeordnet (z.B. fehlende schließende Klammern) worden sind (praktisch Rechtschreib- bzw. Grammatik-Fehler)
  der (fehlerhafte) Quellcode lässt sich nicht in ein ausführbares Programm (Maschinencode) übersetzen
  häufig fehlt ein Doppelpunkt oder es fehlen schließende Klammern oder die Ende-Kennzeichen von Texten

- **NameError** zeigt an, dass eine Variable benutzt wird, der vorher noch kein Wert zugewiesen wurde
  häufig wurde der Variablen-Name nur falsch geschrieben und die Groß- und Kleinschreibung nicht beachtet
  weiterhin tritt der Fehlertyp auf, wenn eine Funktion aufgerufen / benutzt wird, die nicht definiert oder importiert wurde

- **IdentationError** ist ein Einrückungsfehler; entweder wurden keine notwendigen Einrückungen vorgenommen oder die Einrückungen sind unterschiedlich groß
  durch einheitliches Benutzen von Tab-Stop's bzw. Leerzeichen (empfohlen werden 4) leicht zu korrigieren und zu vermeiden

- **ValueError** tritt immer dann auf, wenn zwar der richtige Daten-Typ verwendet wurde, aber der Inhalt (Wert = engl.: value) nicht für die Funktion usw. geeignet ist

- **TypeError** wird angezeigt, wenn für eine Operation / Funktion Daten eines nicht geeigneten bzw. zugelassenen Typ's verwendet oder übergeben wurden
  tritt auch auf, wenn eine Eingabe (standardmäßig ein Text) an eine Rechen-Operation oder –Funktion übergeben wird
  z.B. kann eine Funktion, die ein ganze Zahl erwartet nicht mit einer Kommazahl oder einem Text aufgerufen werden

- **ImportError** weist darauf hin, dass die Bibliothek / das Modul oder die spezifizierten Funktionen nicht verfügbar sind oder falsch geschrieben wurden

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.
Ein anderer Programmierer oder z.B. der Kursleiter sind Personen, die ein Programm einfach nur lesen. Sie müssen und können nicht zwangsläufig die Gedankengänge des Programmierer's verstehen. Programmieren ist heute zudem immer Team-Arbeit. Das Lesen von Programmen passiert deutlich häufiger als das Schreiben und Korrigieren. Deshalb müssen Programme immer auch für fremde Leser verständlich angelegt und geschrieben werden.
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 (Leer-Zeichen):

- 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 (=)
- *ein Leerzeichen vor und hinter dem Vergleichszeichen (z.B.: ==)*

***keine* Leer-Räume:**

- zwischen Funktionsnamen und der öffnenenden Klammer
- hinter einer öffnenden Klammer, vor einer schließenden Klammer
- vor einem Doppelpunkt (von Verzweigungen und Schleifen)

Bezeichner:

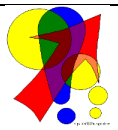
- Variablennamen, Funktionen, Attribute in Kleinbuchstaben (ev. mit Unterstrichen)
- geschützte Attribute mit führendem Unterstrich
- private Attribute mit führendem doppelten Unterstrich
- Klassen und Exceptions mit Großbuchstaben beginnend (und ev. auch intern für Wörterbeginn)
- Modul-weite Konstanten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

→
→
→

Grob-Gliederung eines Programm's

- **ev. Kommentare zum Programm / zur Datei** — Leistung des Programm's, Autor, Datum, Version, Lizenz

Leerzeile — *mindestens*

- **Importe**

Leerzeile — *mindestens*

- **Funktions-Definitionen** — ev. mit Kommentaren

2 Leerzeilen — *mindestens*

- **(Haupt-)Programm / Initialisierung** — möglichst mit Kommentar z.B.: `#MAIN`

(c,p) 2015 - 2025 lsp: dre

Linter

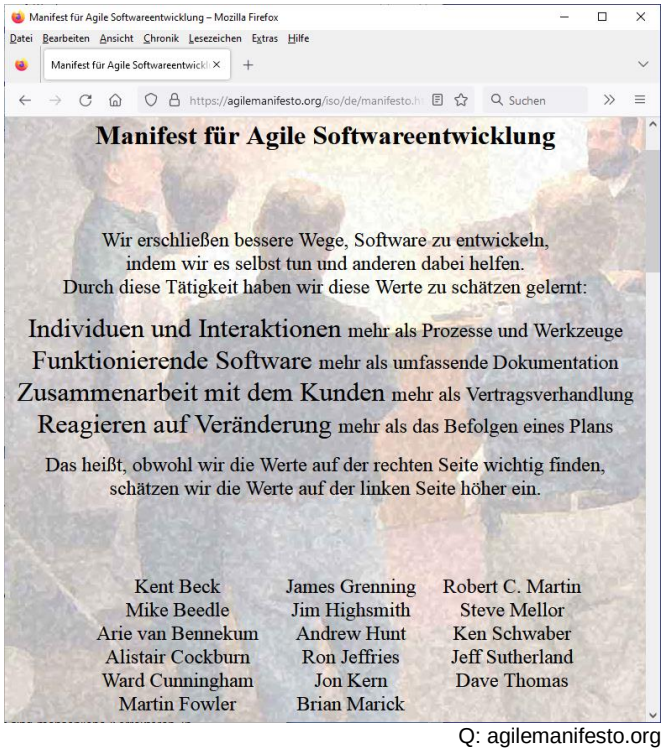
Ein Linter ist ein Programm, das Programm-Code analysiert und geeignete Kommentare zurückgibt.
Es werden dabei:

- Code-Formatierungen
- sinnlose Code-Zeilen
- mögliche, unbeabsichtige Fehler-Quellen

betrachtet.
Linter ergeben immer nur Empfehlungen. Besser als Linter sind menschliche Korrektoren. In der Praxis arbeiten Programmierer in Zweier-Team. Einer programmiert und der andere schaut dem ersten über die Schulter.

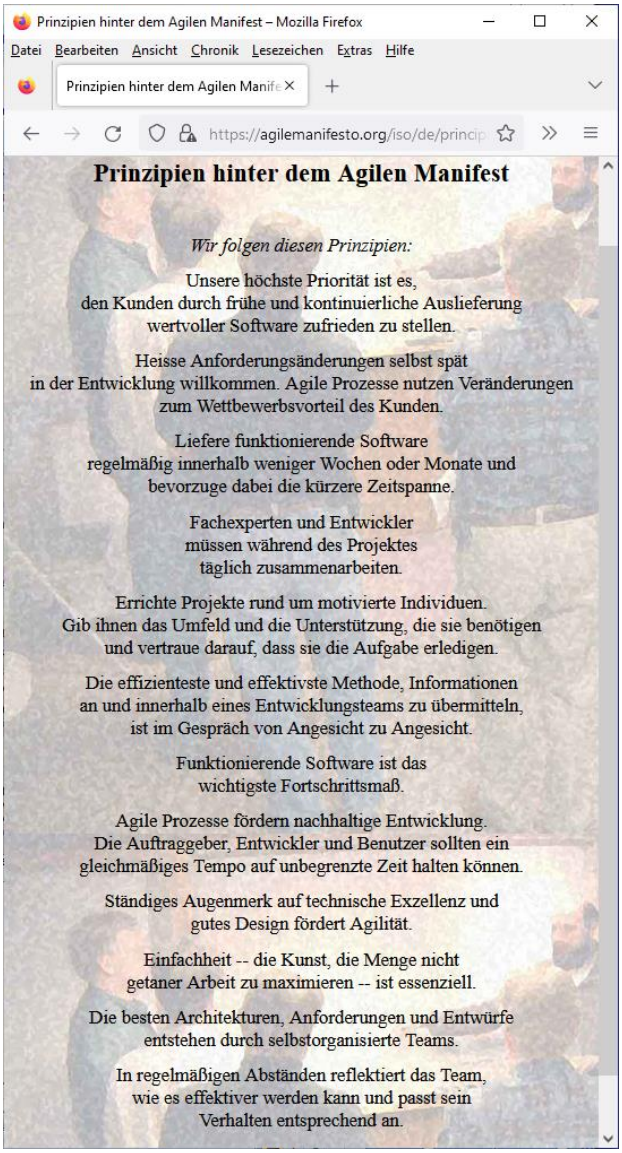
5.4. agile Software-Entwicklung

mehr eine Arbeits-Einstellung für Programmierer, als Programmier-Regeln



Q: agilemanifesto.org

PDCA-Zyklus



Q: agilemanifesto.org

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

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

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. Ausgaben mit Platzhaltern

In Programmen kommt es sehr häufig vor, dass man immer die gleichen Ausgaben oder Ausgaben nach einem bestimmten Muster machen muss. Hierfür kann man Texte / Strings mit Platzhaltern verwenden. Diese kann man sich wie die Freistellen in einem Lücken-Text vorstellen. In Python muss die Lücke aber einen Namen bekommen, damit das System weiss, an welche Stelle die Lücke ist was in diese geschrieben werden soll.
Den eigentlichen Lückentext kennzeichnen wir durch ein f vor dem Text / String. Die Lücke selbst wird durch geschweifte Klammern

```
name = "Hans"
lueckentext = f"Hiermit begrüßen wir Dich, {name}, zu Phyton"}

print(lueckentext)
```

Aufgaben:

1. *Überlegen Sie sich, was bei den Programmen passiert, wenn man das f vor dem Lückentext vergisst?*
2. *Probieren Sie es einfach mal aus!*
3. *Erstellen Sie eine Befehls-Sequenz / ein Programm, das einen beliebigen Namen in den Text nach dem Muster "Hallo Jule!" einfügt!*

In einem Lückentext können mehrere Lücken vorkommen und auch mehrfach der gleiche Wert in unterschiedliche Lücken eingesetzt werden.
Lücken, in die der gleiche Inhalt geschrieben werden soll, bekommen den gleichen Bezeichner. Hier im Beispiel ist dies **name**.

```
name = "Hans"
lueckentext = f"Hallo {name}! Wer kennt {name} schon?"}

print(lueckentext)
```

Für unterschiedliche Inhalte müssen unterschiedliche Bezeichner benutzt werden. Im nächsten Beispiel sind das die bezeichner **name** und **vorname**.

```
vorname = "Hans"
name = "Müller"
l_text = f"Hallo {vorname}! Wer kennt {vorname} {name} schon?"}

print(l_text)
```

Aufgaben:

1. ***Erstellen Sie eine Befehls-Sequenz / ein Programm, das einen beliebigen Namen in den Text nach dem Muster "Jeder kennt Jule. Hallo Jule!" einfügt!***
2. ***Eine Ausgabe soll nach dem Muster "Ein Apfel, zwei Äpfel, drei Äpfel , …, viele Äpfel." für beliebige Objekte / Substantive dienen! (Hier in der männlich bzw. sächlichen Form. Es kann aber auch gerne der Text für weibliche Substantive geschrieben werden.)***
3. ***Erstellen Sie zusätzlich eine englisch-sprachige Ausgabe! Die Objektnamen bleiben gleich***

In Ausgaben können zusätzliche Angaben gemacht werden. Besonders häufig benötigt man einen gewollten oder eben nicht gewollten Zeilen-Umbruch. Auch andere Behandlungen des Endes eines **print()**-Befehl's sind möglich.

`end="\n"`	normaler Zeilen-Umbruch nach der Ausgabe (muss nicht angegeben werden)	
`end="\n\n\n"`	mehrere Zeilen-Umbrüche hintereinander (hier 3)	
`end=""`	kein Zeilen-Umbruch, der nächste **print()**-Befehl erzeugt Ausgabe direkt dahinter	
`sep=" "`	beschreibt die Trennzeichen-Sequenz zwischen Komma-getrennten Ausgaben eines **print()**-Befehl's (ein Leer-Zeichen ist Standard, der nicht extra angegeben werden muss)	
`sep="__"`	setzt zwischen die Ausgaben-Teile zwei Unterstriche (z.B.)	

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. *Zuguter Letzt 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()
```

Näheres zu Variablen (/Objekt-Bezeichnern) haben wir schon weiter vorn besprochen (→ 3.1.3. Eingaben und Daten merken - Variablen).
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.

Eingaben von Texten ist immer sehr aufwändig. Vor allem beim Testen von Programmen kann das schnell nervig werden. Längere Text können aus einer Text-Datei (Typ-Endung .txt) eingelesen werden.

```
dateiName = "Beispiel.txt"
with open(dateiName) as datei:  # with sorgt auch für das Schließen der Datei am
                                # Ende
    inhalt = datei.read()

dateiName = "Beispiel.txt"
with open(dateiName) as datei:
    inhalt = datei.read().splitelines()   # zerlegt inhalt in Zeilen (gespeichert
                                            als Liste von Strings
```

falsche Eingaben können über Schleifen (→ 6.4.2. Schleifen) solange immer wieder abgearbeitet werden, bis die Eingabe den Anforderungen entspricht

eine weitere Möglichkeit der Kontrolle sind excetion's (→ 🕮 **Python - professionell**)

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.

Bei der Abfrage / Eingabe und der folgenden Typ-Umwandlung kann aber eine weiteres Problem auftauchen. Der Nutzer gibt eine Zahl als Wort (also String) ein oder verwendet nicht-zugelassenene Zeichen (z.B. ausversehen einen Doppelpunkt).
Nun lässt sich der Eingabe-String nicht in eine Zahl konvertieren. Python quittiert dies mit einer Value-Fehlermeldung. Die Reaktion auf solche Fehler, die erst während des Programm-Laufes auftreten (="Laufzeit-Fehler") kann man durch eine gezielte Fehler-Behandlung kompensieren. Dazu muss man die mögliche Fehler-Quelle vorher abstecken und einigen zusätzlichen Quell-Code einbauen. Dazu später mehr. Für unsere ersten Programmier-Versuche sind solche Strukturen zu sperrig. Wir gehen in den nächsten Kapiteln davon aus, dass die Eingaben Typ-grecht erfolgen.
Wie man die Laufzeit-Fehler in Python abfängt zeigen wir dann im Abschnitt → 8.15. Behandlung von Laufzeitfehlern – Exception's.
Natürlich kann man von anfangan solche Strukturen einbauen. Das empfehle ich aber nur für fortgeschrittene Programmierer, die von einer anderen Programmiersprache zu Python umsteigen.

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

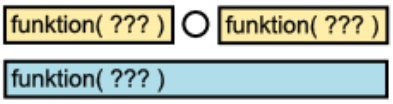
Der Verarbeitungs-Teil eines Programmes enthält alle Operationen, welche die eingegebenen Daten in die Ergebnisse umwandelt. Saclich liegt der Verarbeitungs-Teil nach dem EVA-Prinzip zwischen Eingaben und Ausgaben.
Häufig sind Verarbeitung und Ausgabe stark miteinander verwoben, so dass keine exakte 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.

Operatoren

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

Ope-rator	Name	Beschreibung	Beispiel	Ergebnis
if .. else..		WENN-DANN-SONST Verzweigung		
if .. elif .. else ..		Mehrfach-Auswahl		
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.

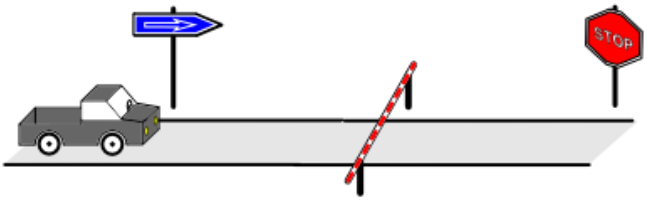
```
Anweisung ::= Variable = Variable
Anweisung ::= Variable = Variable Operator Variable | Term

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:

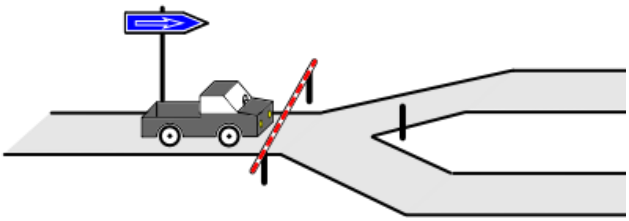
WENN zwei alternative Wege in einem Programm zur Verfügung stehen,
DANN muss eine Entscheidung gefällt werden, welcher der Wege nun genommen werden soll / muss.

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

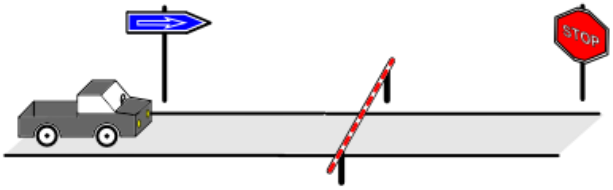


wobei der ELSE-Zweig i.A. optional ist – also bei Bedarf einfach weggelassen werden kann. In dem Fall geht es dann gleich hinter der Alternative1 weiter.
Eine vollständige Verzweigung wird auch zweiseitig genannt, eine ohne ELSE-Zweig heißt einseitige Verzweigung.

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.



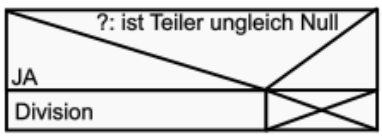
Wollen wir z.B. eine Division programmieren, dann ist das sicher kein allgemeines Problem:

```
…
# 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:

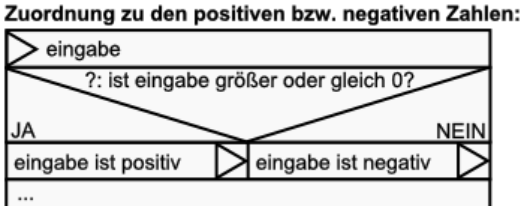
1. *Zeichnen Sie das Struktogramm für die vollständige Bearbeitung der Division mit zwei if-Anweisungen!*
2. *Erstellen Sie ein Programm, das für eine einzugebene (ganze) Zahl prüft, ob es sich um eine gerade oder ungerade Zahl handelt!*
3. *Erstellen Sie ein Programm, das die Teilbarkeit einer einzugebenen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

Python lässt auch kombinierte Vergleiche zu:

5 < 9 <= 13 ergibt True (Prüfung erfolgt bezüglich der 9)

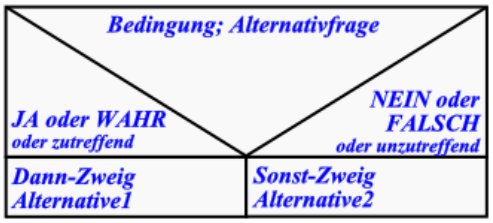
zweiseitige Auswahl / vollständige Verzweigung

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.

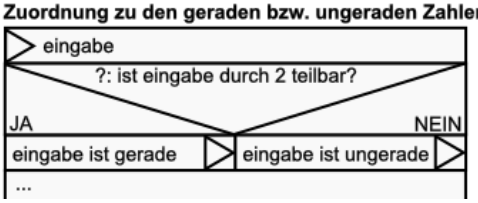
Im nachfolgenden Code-Schnipsel ist die Bedingung anders gestellt. Dadurch tauschen THEN- und ELSE-Zweig.

```
…
# 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
^	XOR	exklusives ODER	(1 == 2) ^ (2 == 1) (1 == 2) ^ (2 == 2)	False True
&	bitweise UND			
|	bitweise ODER			

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.")
…
```

!!!: Fehler-hafter Quell-Code!!!

Aufgaben:

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

Listen / Collection's lassen sich auch gemeinschaftlich analysieren:

```
boolListe = [True, False, False, True, True]
```

`all(boolListe)`	ergibt False, da zwei Elemente nicht wahr sind
`any(boolListe)`	ergibt True, da mindestens ein Element wahr ist
`sum(boolListe)`	ergibt 3, weil drei wahre Werte in der Liste enthalten sind (True wird als 1 gespeichert, False als 0)

Schaltjahr-Prüfung nit kombinierten logischen Operatoren

```
jahr = int(input("Welches Jahr soll geprüft werden: "))
if jahr % 4 == 0 and jahr % 100 != 0 or jahr % 400 == 0:
    print(jahr,"ist ein Schaltjahr")
else:
    print(jahr,"ist KEIN Schaltjahr")
```

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?*

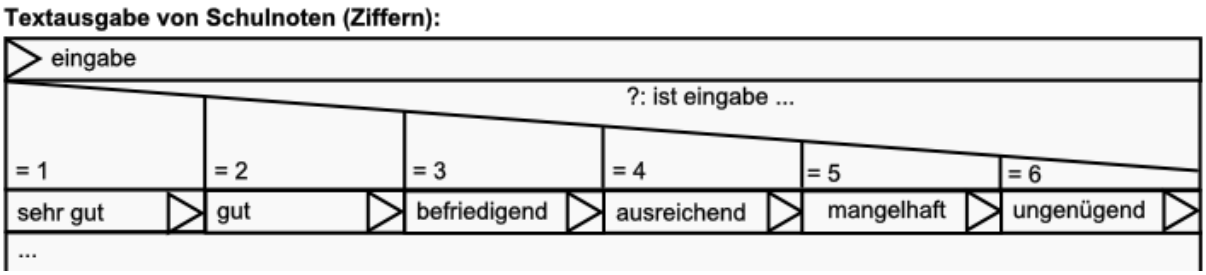
a)	`3 == 3`	b)	`456 <= 289`	c)	`4 == '4'`
d)	`5 == 3+2`	e)	`"Eingabe" == 0`	f)	`"Name" <= "Vorname"`
g)	`8.0 == 8`	h)	`"Hallo!" == "Hallo! "`		
i)	`"Ei" is "Ei"`	j)	`a = 23`	k)	`24 // 8 == 4`
l)	`5 in [2, 3, 5, 7]`	m)	`4 not in [9, 3, 2, 4 ,6, 8, 13, 1, 99]`		
n)	`"Bio" not in ["Astro", Bio, "Chem",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

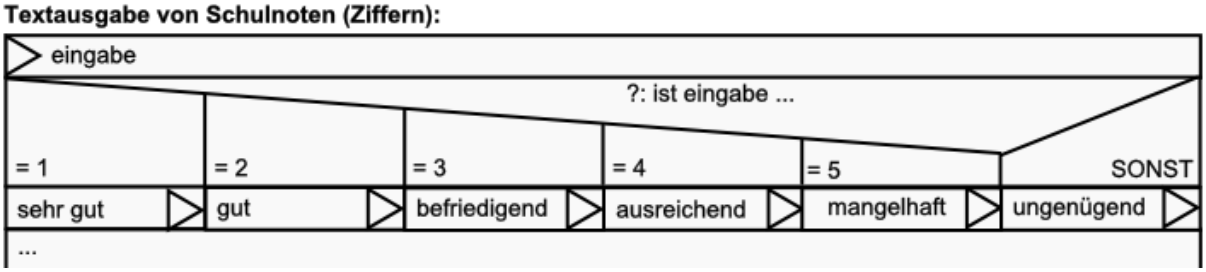
Neben den einfachen Verzweigungen kennen viele Programmiersprachen Mehrfach-Verzweigungen. Meist lautet das Schlüsselwort dann SWITCH, CASE oder so ähnlich. Python geht bei den Mehrfach-Verzweigungen einen ganz einfachen Weg – es erweitert einfach die "normale" Verzeigung.
Im nächsten Beispiel sollen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



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:



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:

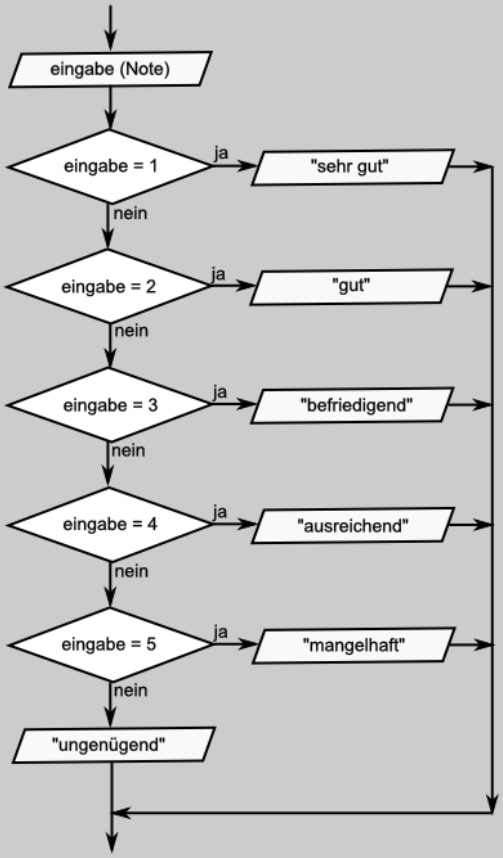
```
…
# Mehrfach-Alternative
if temp < 19.0:
    print("zu kalt")
elif temp > 19 and temp < 20:
    print("kühl")
elif temp > 20 and temp < 22:
    print("angenehm")
elif temp > 22:
    print("angenehm")
else: # nicht ausgewertete Fälle
    print("es ist ein Fehler aufgetreten!")
    print("Melden Sie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

6.4.1.4. Optimierung des Quellcode's – DRY- und EVA-Prinzip

Das DRY-Prinzip (don't repeat yourself) besagt, dass man möglichst alle Dinge nur einmal in einem Programm codiert. Auch späteres erneutes Programmieren sollte vermieden werden.
Für einen Anfänger ist dies zuerst etwas schwierig. Er muss sich noch zu viel auf seinen Code konzentrieren. Aber spätestens, wenn ein funktionierender Code vorliegt, sollte man sich um eine Anpassung an die informatischen Prinzipien kümmern. Nerdigen Code wird man später nicht wirklich in ein Team einbringen und für einen selbst wird es auch sehr uneffektiv, immer das Gleiche mehrfach zu codieren.

```
1  // Schaltjahr-Programm
2
3  eingabe = input("Schaltjahr-Prüfung für Jahr?: ")
4  jahr = int(eingabe)
5
6  if jahr < 0:
7      print("es muss ein positives Jahr sein!")
8  else:
9      if jahr % 4 == 0:  // ?durch 4 teilbar
10         if jahr % 100 == 0: //? durch 100 teilbar
11             if jahr % 400 == 0: // ?durch 400 teilbar
12                 println("kein Schaltjahr")
13             else:
14                 println("ist Schaltjahr")
15         else: // ? %100
16             println("ist Schaltjahr")
17     else: // ? %4
18         println("kein Schaltjahr")
```

Auf den ersten Blick ist es ein ordentliches Programm. Was den Informatiker stört, sind die mehrfachen Ausgaben. Würde man das Programm für eine andere Sprache umschreiben wollen, dann müsste man auch wieder doppelt arbeiten. Weiterhin ist die Mischung aus Verarbeitung und Ausgabe ungünstig. Beide Bereiche sollten möglichst vollständig voneinander getrennt werden. Dann kann man die Ausgabe auch in unterschiedlicher Form erledigen, z.B. wie bisher üblich auf der Konsole, oder in einer graphischen Oberfläche.
Sollte unser Schaltjahr-Problem noch einige weitere Male im Programm auftauchen, dann müssen wir auch wieder mehrfach den Code notieren oder reinkopieren. Derzeit haben wir noch keine echte Lösung dafür, aber mit Funktionen (→ 6.5. Unterprogramme, Funktionen usw. usf.) wird das später bestens gehen. Derzeit überlegen wir uns nur, wie man die Information von der Verarbeitung zur Ausgabe transportiert. Wir wollten ja wissen, ob es sich um ein Schaltjahr handelt oder nicht. Also handelt es sich eigentlich um einen Wahrheitswert. Genau so eine Variable nutzen wir nun. Auch das Auftreten eines Fehlers erfassen wir nur, merken uns diesen und geben ihne dann u.U. später aus.

```
// Schaltjahr-Programm

//Eingabe
eingabe = input("Schaltjahr-Prüfung für Jahr?: ")
jahr = int(eingabe)

//Verarbeitung
ist_schaltjahr = False
fehler = 0
if jahr < 0:
    fehler = 1
else:
    if jahr % 4 == 0:  // ?durch 4 teilbar
        if jahr % 100 == 0: //? durch 100 teilbar
            if jahr % 400 == 0: // ?durch 400 teilbar
                ist_schaltjahr = False
            else:
                ist_schaltjahr = True
        else: // ? %100
            ist_schaltjahr = True
    else: // ? %4
        ist_schaltjahr = False

//Ausgabe
if fehler > 0:
    println("Es ist ein Fehler aufgetreten.")
else:
    if ist_schaltjahr:
       println(jahr," ist ein Schaltjahr")
    else:
       println(jahr," ist kein Schaltjahr")
```

Natürlich müssen wir unser Programm gründlich testen. Dabei werden wir merken, dass es nicht ganz exakt arbeitet.

Aufgaben:

1. ***Recherchieren Sie, welche Regeln zu den Schaltjahren in den verschiedenen Kalendern festgelegt wurden!***
2. ***Korrigieren Sie Ihr Programm, so dass es ordnungsgemäß funktioniert!***
3. ***Prüfen Sie Ihr Programm mit mindestens 10 weiteren Jahres-Zahlen, die Sie im Kurs gemeinsam auswählen, um möglichst viele Sonder-Fälle zu testen!***

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!*

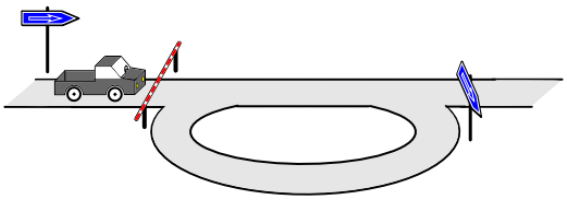
für die gehobene Anspruchsebene:

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.

DRY-Prinzip
don't repeat yourself

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.

Struktogramm: Kopfgesteuerte Schleife

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!

zulässige Schachtelung von Schleifen

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.

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 =  3.14159265358979267190875361848 2
geforderte Genauigkeit e =  0.00000000000000000010000000000
   ---> nach:  26  Interationen
 zum Vergleich System-Pi =  3.14159265358979311599796346854 4
>>>
```

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.

Eingabe-Kontrolle mittels Schleife

derzeit wohl noch nicht verständlich, aber ein gutes Mittel um z.B. Ganzzahl-Eingaben zu erzwingen

```
wert = None      # wert bekommt einen internen Wert, der für NICHTS steht
while not isinstance(wert, int):   # wiederholt solange bis der eingegebene wert
                                     vom Typ int ist
    wert = input("Gib eine Zahl ein: ")   # eigentliche Eingabe
    try    # versuchsweise Durchführung der nächsten Zeilen
       wert = int(wert)    # Umwandeln des Eingabe-Strings in eine Zahl
    except ValueError:
        print("Es sind nur Ziffern zugelassen!")
```

Fehler-Analyse in Schleifen

Schleifen stellen häufig große Fehler-Quellen dar. Eine einfache Variante ist die Anzeige von Werten vor und nach der Schleife.
Die zusätzlichen Anzeigen sollten unbedingt mit Kommentaren versehen werden, damit man sie nachher wieder gezielt entfernen oder auskommentieren kann.

```
anz=0
sum=0
print("anz=",anz) #für Tests
print("sum=",sum) #für Tests
while anz<3:
    sum=sum+anz
    anz=anz+1
print("anz=",anz) #für Tests
print("sum=",sum) #für Tests
anz=0
```

später auskommentieren

```
# print("anz=" …
# print("sum=" …
```

Vor allem die Vorgänge innerhalb der Schleifenkörper werden schnell zum MysteriWm. welcher Wert wird da wirklich bei einem Durchlauf verrechnet?
Will man allerdings auch die Schleifen-Variablen verfolgen, dann bleibt einem nur die Anzeige von Werten innerhalb der Schleife. Für Test-Zwecke kann man dann vielleicht die Anzahl der Durchläufe künstlich herabsetzen. Wenn's mit den kleinen Zahlen läuft, dann kann man sich auch an die größeren wagen und ev. nur noch spezielle Werte anzeigen (z.B. mit Bedingung).

```
anz=0
sum=0
while anz<3:
    sum=sum+anz
    anz=anz+1
    print("anz=",anz) #für Tests
    print("sum=",sum) #für Tests
anz=0
```

später auskommentieren

Nicht jeder Algorithmus kann so ohne weiteres implementiert werden. Auch der Eingriff in laufende Systeme ist nicht so ohnen weiteres möglich. Oft steht auch nur ein Struktogramm od.ä. zur Verfügung. An diesem soll dann die Funktionsfähigkeit geprüft werden.
In vielen Fällen helfen Verfolgungs- bzw. Variablen-Protokolle beim Finden von Fehlern.
Wir gehen hier mal davon aus, dass uns der Algorithmus nur als Quell-Text vorliegt und wir diesen offline prüfen müssen.

```
anz=0
sum=0
while anz<3:
    sum=sum+anz
    anz=anz+1
anz=0
```

In einer ersten Version extrahieren wir die kritischen Befehle der Schleife und ordnen sie horizonzal an.

Befehle	`while anz<3:`	`sum=sum+anz`	`anz=anz+1`

Das bietet den Vorteil, dass eine Schleifen-Situation immer in einer Zeile dargestellt wird. Bei vielen Befehlen innerhalb des Schleifenkörpers wird diese Darstellungs-Variante aber auch schnell sehr breit und unübersicht.
Als nächstes analysieren wir die Eintritts-Bedingungen – also welche Werte vor der Schleife vorliegen.

Befehle	`while anz<3:`	`sum=sum+anz`	`anz=anz+1`
Eintritt		sum=0	anz=0

Nun gehen wir Durchlauf für Durchlauf durch die Schleifen-Befehle. Beim ersten Durchlauf erhalten wir:

Befehle	`while anz<3:`	`sum=sum+anz`	`anz=anz+1`
Eintritt		sum=0	anz=0
1. Durchlauf	0 < 3 **true**	sum=0 + 0 **0**	anz=0 + 1 **1**

Hilfe / Berechn.

Der obere Teil (also die Berechnung über der Strichel-Linie) ist vor allem für die ersten versuche / Durchläufe als Hilfe zu empfehlen. Zuoft glaubt man etwas programmiert zu haben, was aber gar nicht in den Befehlen ausgedrückt wird. Man kann die Ausdrücke ja auch nur mit Bleistift schreiben, um so die wesentlichen Inhalte – also die Variablen-Werte – deutlicher erkennen zu können.
Nach und nach ergänzt man nun die Zeilen für die nächsten Durchläufe:

Befehle	`while anz<3:`	`sum=sum+anz`	`anz=anz+1`
Eintritt		sum=0	anz=0
1. Durchlauf	0 < 3 **true**	sum=0 + 0 **0**	anz=0 + 1 **1**
2.	1 < 3 **true**	sum=0 + 1 **1**	anz=1 + 1 **2**
3.	2 < 3 **true**	sum=1 + 2 **3**	anz=2 + 1 **3**
4.	3 < 3 **false**		

Beim 4. Durchlauf erhalten wir beim WHILE-Ausdruck ein false zurück. Damit wird die Schleife nicht mehr ausgeführt und wir können die restliche Zeile streichen

4.	3 < 3 **false**		

Das gesamte Protokoll sieht dann für unser Beispiel so aus:

Befehle	**`while anz<3:`**	**`sum=sum+anz`**	**`anz=anz+1`**
Eintritt		sum=0	anz=0
1. Durchlauf	0 < 3	sum=0 + 0	anz=0 + 1
	true	**0**	**1**
2.	1 < 3	sum=0 + 1	anz=1 + 1
	true	**1**	**2**
3.	2 < 3	sum=1 + 2	anz=2 + 1
	true	**3**	**3**
4.	3 < 3		
	false		
Austritt		sum=3	anz=3

Bei Schleifen mit vielen Befehlen kann man sich ev. auf die Befehle mit den interessierenden Variablen beschränken. Aber Achtung, es müssen alle Befehle enthalten sein, die in irgendeiner Form mit den zu beobachtenden Variablen zusammenhängen!
Als erste Vereinfachung kann man dann auf die Hilfen mit den Vergleichen und Berechnungen verzichten.
Ist man dann etwas geübter in der Analyse von Schleifen-Variablen, dann bietet sich ein weiter vereinfachtes Protokoll an. Hierbei wird auf die einzelnen Berechnungen usw. verzichtet und nur noch die Variable und ihr Wert notiert. Allerdings ist hier immer sehr gründlich zu arbeiten.

Durchlauf	**Bedingung**	**sum**	**anz**
vorher	---	0	0
1	true	0	1
2	true	1	2
3	true	3	3
4	false	---	---
danach		3	3

Aufgaben:

1. ***Analysieren Sie zuerst die nachfolgende Schleife ohne Hilfsmittel! Welche Werte erwarten Sie beim Schleifen-Austritt? Begründen Sie!***

```
1  prod=0
2  sum=0
3  anz=0
4  offset=2
5  while anz<=5:
6      anz=anz+1
7      sum=sum+anz
8      prod=prod*anz+offset
```

2. ***Erstellen Sie sich ein Verfolgungs-Protokoll für alle Variablen!***
3. ***Vergleichen Sie Ihre Erwartungs-Werte mit den Daten aus dem Protokoll! Wenn Sie sich vertan haben, versuchen Sie zu ergründen, an welcher Stelle Sie einen Denkfehler gemacht haben!***

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 ist hier einfach mehr möglich!

wenige andere Programmiersprachen bieten ein ähnliches Konzept

Sammlungen können Aufzählungen und Listen sein. Den Bezug zwischen der Zähl-Schleifen-Anweisung und der Aufzählung wird über das Schlüsselwörtchen **in** hergestellt.
Die Schleifen-Anweisung arbeitet dann eben alle Werte ***in*** der Sammlung ab.
Eine Aufzählung beinhaltet Werte in einem runden Klammer-Paar **( )**, bei einer Liste sind die Werte in eckige Klammer **[ ]** notiert. Die Werte selbst sind Komma-getrennt.
Die Unterschiede zwischen Aufzählungen (exakt: Tupel genannt) und Listen sind für uns hier nicht relevant. Diese besprechen wir dann später.

```
# Countdown
for wert in (10, 9, 8, 7, 6, 5, 4, 3, 2, 1):
    print(wert)
print("Start")
# Warten auf Beenden
input()
```

```
>>>
10
9
8
7
6
5
4
3
2
1
Start
>>>
```

Auch hier brauchen wir eine Schleifen-Variable. Bei uns ist das dieses Mal wert. In dieser Variable steckt die Kennung für den konkreten Schleifen-Durchlauf. Die Kennungen werden nach und nach der Aufzählung entnommen und abgearbeitet.
Die Aufzählung kann auch in einer Variable gespeichert werden. Das mach Sinn, wenn man diese öfter benötigt.

```
werte = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
# Countdown
for wert in werte:
    print(wert)
print("Start")
# Warten auf Beenden
input()
```

Man kann sogar bei der Variablen-Zuweisung auf die Klammern verzichten. Aber gerade bei unübersichtlichen Sammlungen sollte man die Grenzen sauber abstecken.

im Schleifen-Kopf kann auch auf eine Liste (Sammlung) zurückgegriffen werden, 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 (→ 8.2.3. Listen, die I. – einfache Listen, 9.7. 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!*

Während Listen veränderlich sind, also z.B. geleert, erweitert oder Elemente daraus gelöscht werden können, sind Tupel (Aufzählungen) unveränderlich. Man kann also ein Tupel als konstante Liste verstehen.
Intern ist die Verarbeitung von Tupel etwas schneller, als die der Listen. Wenn man also feste Aufzählungen hat, dann sollte man zu Tupel greifen. Für alle anderen Fälle sind Listen immer die richtige Wahl. Mit Hilfe von Tupeln lassen sich auch gut Gruppen von Variablen bzw. Werten programmieren. Ein schönes Beispiel ist die Berechnung neuer Punkt-Koordinaten als x-y-Paar (→ (x, y) = ….).

```
# ========================================
# 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!
4. *Überlegen Sie sich, was das folgende Programm leistet! Geben Sie den Code dann in Python ein und prüfen Sie, ob Sie mit Ihrer Vorüberlegung recht hatten!*

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

# Verarbeitung der Aufzählung
for zahl in zahlenliste:
    print(zahl," --> ",zahl*zahl)
# Warten auf Beenden
input()
```

um sich z.B. alle Objekt-Namen aus einem Modul (einer Bibliothek) anzeigen zu lassen, kann man die Funktion dir() benutzen

```
import math
for obj in dir(math):
    print(obj)
```

auch Zeichenketten sind iterierbar:

```
for buchstabe in zeichenkette:
    print(buchstabe)
```

mit Tupeln in einer abzuarbeitenden / Sammel-Liste kann man in der Schleife auch mit mehreren Iteratoren (Lauf-Variablen) arbeiten:

```
for a, b, c in [(2,5,9), (2,3,2), (1,5,9), (2,5,1), (3,7,9)]:
    print(a * b + c)
```

die Kombination von Schleifen und darin genutzten Funktionen lassen sich mit der **map()**-Funktion zusammenfassen
das ergibt kompakten, aber schwer lesbaren Code
in der Praxis sind map-basierte Programme effektiver als die originalen / expandierten

orginal:

```
liste = ["abcdef","ghij","klmnopqrt","rstuvwxyz"]
elementLaenge = []
for element in liste:
    elementLaenge.append(len(element))
```

ergibt die Liste (elementLaenge): [6, 4, 9, 9]

komprimiert mit map():

```
elemenLaenge = list(map(len, liste))
```

map() erwartet die zu nutzende Funktion (nur den Namen!) und das zu nutzende Objekt (Liste od.ä.)

das Gegenstück zu map() ist die **filter()**-Funktion
auch sie erwartet eine zu verwendende Funktion und eine Liste von zu nutzenden Objekten
hier muss die Funktion aber unbedingt einen Boolean-Wert zurückliefern, bzw. das Funktions-Ergebnis wird als bool betrachtet!

sollen vor map und filter keine Funktionen definierte werden (worden sein), dann kann man kurze Berechnungen etc. über die anonyme Funktion **lambda** direkt in die map()- bzw. filter()-Funktion notieren:

```
… list(map(lambda x: x = x+1, zahlenListe)
… list(filter(lambda x: x < 100, zahlenListe)
```

ändert man (Sammlungs-)Listen od.ä. innerhalb der Schleife, dann ist zu beachten, dass Python sich über einen internen Index durch die Liste hangelt, wenn ein Element der Liste entfernt wird, merkt Python das nicht, es arbeitet mit dem nächsten Index-Element weiter
besser ist es, eine neue Ergebnis-Liste anzulegen und dort die Ergebnisse Element-weise sammeln

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**)**

mit den drei Parametern lassen sich auch absteigende Folgen erstellen:

range(Obergrenze, Untergrenze, neg_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

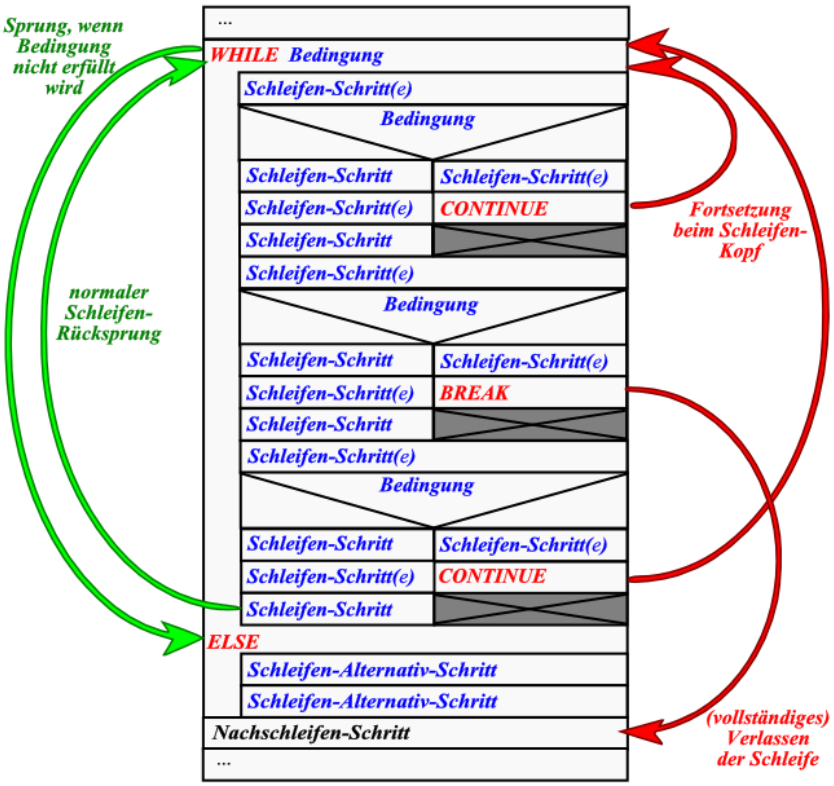
c)
```
*
*#
*#*
*#*#
…
```
bis 20 Zeichen in der letzten Zeile

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.

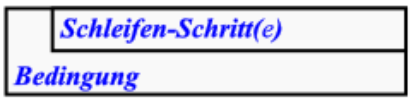
```
…
abbruch=False
while not abbruch:
    # auszuführender Code
    …
    if bedingung:
        abbruch=True
    …
    # Ende des auszuführenden Codes in der Schleife

# 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. Erstellen Sie auf einem kariertem Blatt Papier aus den vorgegebenen Code-Schnipseln ein Programm, welches die fortlaufende Summe aus einzugebenen Werten (Ganzzahlen) berechnet und anzeigt! Die Eingabe und Summen-Bildung soll mit der Eingabe einer Null beendet werden!
(Vereinbarung: Auf dem kariertem Papier erfolgt eine notwendige Einrückung mit mindestens zwei Kästchen!)

`eingabe = -1`	`print("Berechnung der fortlaufenden Summe von Ganzzahlen")`		
`print("aktuelle Summe:",summe)`		`eingabe = int(eingabe)`	`print()`
`print()`	`summe = 0`	`print("Abbruch mit Eingabe von 0")`	
`summe = summe + eingabe`		`eingabe = input("neuer Wert: ")`	
`while not eingabe == 0:`		`input("Programm-Ende bestätigen")`	

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 auf einem kariertem Blatt Papier aus den geeigneten Code-Schnipseln ein Programm, welches das fortlaufende Produkt aus einzugebenen Werten (Ganzzahlen) berechnet und anzeigt! Die Eingabe und Produkt-Bildung soll mit der Eingabe einer Null beendet werden!
(Vereinbarung: Auf dem kariertem Papier erfolgt eine notwendige Einrückung mit mindestens zwei Kästchen!)

`eingabe = -1`	`print("Berechnung des fortlaufenden Produktes von Ganzzahlen")`		
`print("aktuelles Produkt:",summe)`	`print("Abbruch mit Eingabe von 0")`	`print()`	
`if produkt != 0:`	`while produkt > 0:`	`eingabe = input("neuer Wert: ")`	
`produkt = produkt + eingabe`	`if eingabe != 0:`	`eingabe = int(eingabe)`	
`produkt = eingabe * produkt`	`eingabe = int(input("neuer Wert: "))`		
`print("aktuelle Summe:",produkt)`	`input("Programm-Ende bestätigen")`	`print()`	
`while not eingabe == 0:`	`print("aktuelles Produkt:",produkt)`	`summe = 0`	
`produkt = eingabe + 2`	`eingabe = ganzzahl(Eingabe)`	`Produkt = 1`	

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 Linien-Diagramm 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.
Anne → kein Isogramm; Stamm → kein Isogramm

für absolute Freak's:

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

6.4.2.6. Anwendungs-Beispiel: lineare Regression

In der experimentielle Forschung werden wir immer wieder mit Datensätzen konfrontiert, für die auf den ersten Blick nicht klar ist, ob zwischen zwei größen ein Zusammenhang existiert. Gerade bei wenigen Daten struen die Messwerte doch sehr häufig.
Mit der sogenannten Regression kann geprüft werden, ob es einen Zusammenhang gibt oder eben nicht.
Der einfachste Fall ist die lineare Regression. Hierbei wird getestet ob zwischen zwei größen ein linearer Zusammenhang existiert. Dabei nutzt man die Methode der kleinsten fehler-Quadrate. In dieser wird die Gerade so berechnet, dass die Abweichungen – exakt deren Quadrate – möglichst klein sind.

Für eine lineare Funktion vom allgemeine Typ y = m x + n ergibt sich für:

$$m = \frac{j \cdot \sum(x \cdot y) - \sum x \cdot \sum y}{j \cdot \sum x^2 - (\sum x)^2}$$

und für:

$$n = \frac{\sum y - m \cdot \sum x}{j}$$

wobei j die Anzahl der Daten-Paare ist.
Beim Analysieren der beiden Formeln fällt auf, dass mehrere Summen gebraucht werden. Diese können entweder beim Durchlaufen der Daten-Liste oder eines Array's gebildet werden. Aber auch wenn man die Daten quasi online eingeben will / muss, lassen sich die Summen gut bilden. Am Ende werden diese dann zu m und n verrechnet.

Beispiel für Daten in zwei Listen

```
x_werte = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
#Ziel: 4x+3
y_werte =[2.9, 7.2, 10.9, 15.3, 18.6, 23.0, 27.4]
#Ziel: x^2 (Quadrate)
#y_werte = [0.0, 1.1, 1.9, 8.9, 16.2, 25.0, 36.3]

j = len(x_werte)  # WertePaar-Zähler
if len(y_werte)!=j:
    print("Fehler: ungleiche Anzahl x- und y-Werte")
else:
    sum_x = 0
    sum_y = 0
    sum_xy = 0
    sum_x2 = 0
    for i in range(j):
        sum_x += x_werte[i]
        sum_y += y_werte[i]
        sum_xy += x_werte[i]*y_werte[i]
        sum_x2 += x_werte[i]*x_werte[i]
    print("Summe X:",sum_x,"   Summe Y:",sum_y,"   Summe X*Y:",sum_xy,
          "   Summe X*X:",sum_x2)
    m = (j*sum_xy-sum_x*sum_y)/(j*sum_x2-sum_x*sum_x)
    n = (sum_y - m*sum_x)/j
    print("Anstieg m:",m,"   Schnitt der Abszisse n:",n)
```

Wir bekommen so eine Gerade. Ob diese aber einen echten Zusammenhang darstellt oder einfach nur blind berechnet ist, kann mittels Korrelations-Koeffizienten r berechnet werden.

$$r = \frac{\sum(x - \bar{x}) \cdot \sum(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \cdot \sum(y - \bar{y})^2}}$$

Somit erweitern wir den Else-Zweig:

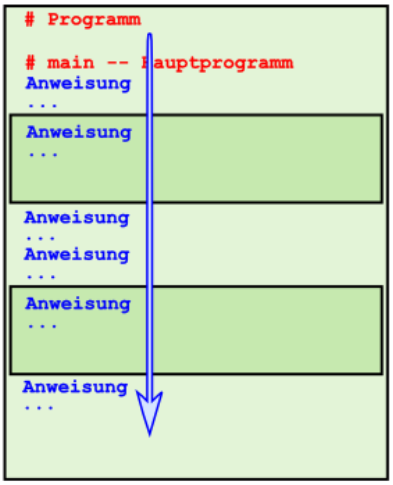
```
mx = sum_x/j
my = sum_y/j
sum_dx = 0
sum_dy = 0
sum_dxy = 0
for i in range(j):
    sum_dx += x_werte[i]-mx
    sum_dy += y_werte[i]-my
    sum_dxy += (x_werte[i]-mx)*(y_werte[i]-my)
print("Summe Abweichungen X:",sum_dx,"   Summe  Abw. Y:",sum_dy,
      "   Summe Produkt Abw. XY:",sum_dxy)
r = (sum_dx*sum_dy)/math.sqrt(sum_dx*sum_dx*sum_dy*sum_dy)
```

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.



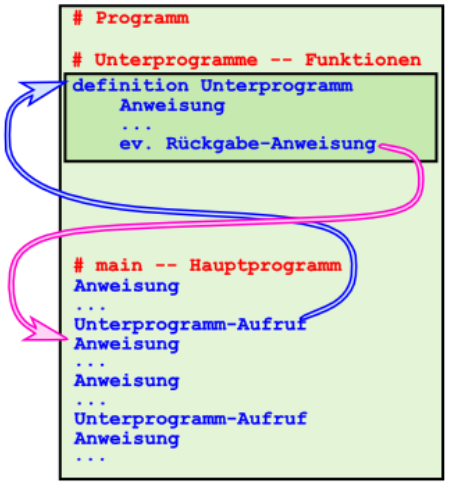
Beim wiederholten Schreiben können – neue / weitere – Fehler auftreten. Beim Kopieren kann man ev. vergessen, das Variablen wieder neu gestartet werden müssen oder – weil sie noch woanders benutzt werden – sie einer Umbenennung bedürfen.
Problematisch ist es auch, wenn sie die Abschnitte als Fehler-behaftet 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.

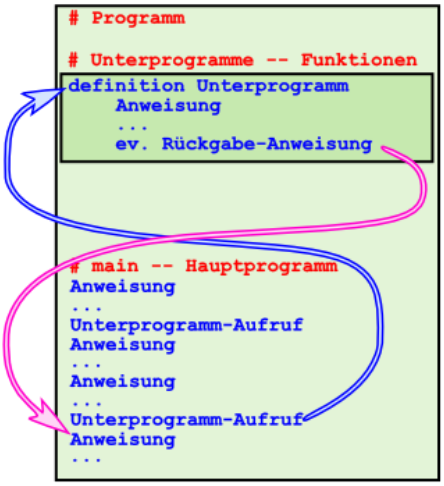
Günstiger wäre / ist es, jeden Programm-Teil nur einmal zu schreiben. Braucht man dann den speziellen Sequenz-Teil, ruft man ihn auf und kehrt danach wieder zur Haupt-Sequenz zurück.
Solche Teil-Sequenzen werden Unter-Programme, Prozeduren und / oder Funktionen genannt. Einige Programmiersprachen unterschieden 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.

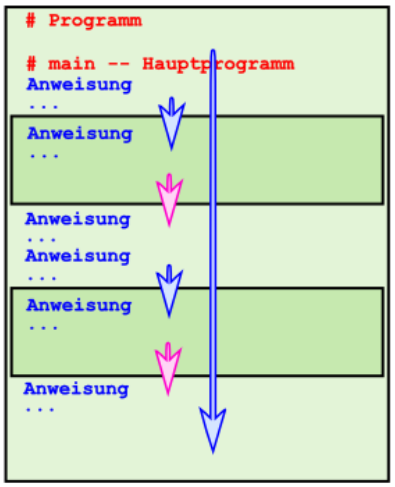
Danach wird mit der nächsten Anweisung im Haupt-Programm fortgesetzt..

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 Benennung 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.

Vorteile von Funktionen
• **verbesserte Struktur des Programm's**
• **Vermeidung von Code-Dopplungen**
• **Erleichterung der Code-Pflege und Fehler-Bereinigung**
• **einfache Wiederverwendbarkeit**
• **erleichtertes Verständnis des Quell-Textes**
•
•
•
•

Funktionen gehören zur Klasse **function**

Der Gültigkeits-Bereich (Scoping) von Variablen ist abhängig von ihrer Erst-Deklaration (erste Wert-Belegung).
In einer Funktion angelegte Variablen gelten nur innerhalb der Funktion. Die lokales Anlage hat Vorrang vor einer globalen Anlage. Ihre Instanz wird nach dem Verlassen der Funktion automatisch gelöscht. Man kann auch später nicht mehr auf deren Wert zugreifen.
Globale – also außerhalb der Funktion definierter – Variablen gelten auch innerhalb. Es dürfen aber keine Funktions-interne Neudefinitionen vorhanden sein.
Werden einer Funktion bei den Parametern eine Variable übergeben, dann wird deren Wert der Parameter-Variablen zugeordnet. Die Parameter-Variable kann bedenkenlos benutzt werden.
Die Argument-Variable (äußere bzw. globale Variable) sollte innerhalb der Funktion nicht verändert werden!

Im Zweifelsfall sollte man sich die Werte vor, innerhalb und nach dem Funktions-Aufruf auch mal anzeigen lassen. Die Anzeigen können dann später auskommentiert werden.

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 allgmeine 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!*

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, den oder die Parameter in Klammern hinter dem Funktionsnamen aufzuzählen.

die Variablen, die in der Funktions-Definition angegeben werden, heißen Parameter
die Werte, die beim Aufruf der Funktion mitgegeben werden, heißen Argumente
in den normalen Fällen muss die Anzahl der Argumente beim Aufruf, genausogroß sein, wie die Anzahl der Parameter bei der Definition der Funktion

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

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

…
```

```
…
# Quadrat-Funktion
def quadrat(arameter):
    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(parameter1, parameter2):
    return parameter1 + parameter2

…
```

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 Argument(-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 Argumenten 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!***

def funktions_name(***Parameter(-Liste)***):
    # Funktions-Inhalt
    return ***Rückgabe-Wert***

ergebnis_variable = funktions_name(***Argument(-Liste)***)

6.5.4. Funktionen mit Standard-Werten als Parameter

hinter dem Parameter in der Funktions-Deklaration wird mit einem Zuweisungs-Zeichen der Standard-Wert angegeben

dieser wird verwendet, wenn keine Angabe für den Parameter getätigt wird
wird dagegen ein Wert angegeben, dann überschreibt er den Standard-Wert

```
def ausgabe(x = 'ok'):
    print("Ergebnis ist ",x)


# Main
ausgabe()

ausgabe("fehlerhaft")
```

6.5.5. Funktionen mit einer variablen Anzahl von Parametern

z.B. print()
funktioniert ohne, mit einem und auch vielen Argumenten

def funktionsname(ArgumentZaehler=anzahl, *variableArgumente): …

Funktionen können auch Listen oder Tupel usw. zurückgeben

```
import math
def loeseQuadratGleichungPQ(p,q):
    hilf = math.sqrt(p**2/4 - q)
    nullstellen = (-p/2 + hilf, -p/2 - hilf)
    return nullstellen
```

6.5.6. 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)

Aufgaben:

1.
2. *Erstellen Sie ein Programm, dass eine Begrüßung für den Nutzer ausgibt! Der Name der Nutzers soll vorher eingeben werden und der Funktion, die willkommen() heißen soll, übergeben werden.*
3.

6.5.7. 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.

6.5.8. 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:

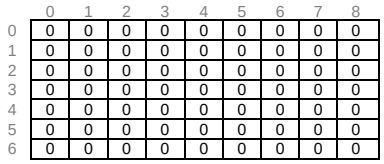
```
feld = [ ]
zeilen = 4
spalten = 6
for spa in range(spalten):
        feld.append(range(zeilen))
        for zei in range(zeilen):
                feld[zei][spa] = 0
```

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

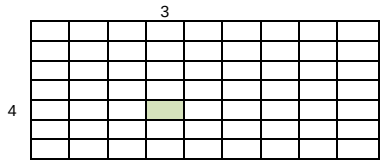
```
import numpy as npy

feld = npy.zeros((7,9))
```



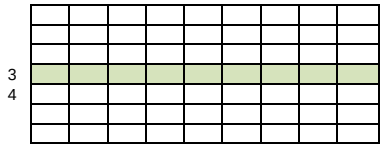

```
wert = feld[4][3]
```

oder

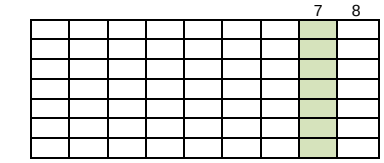
```
wert = feld[4,3]
```



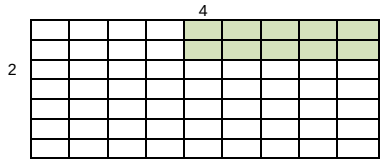

```
sicht = feld[ 3:4, :]
```



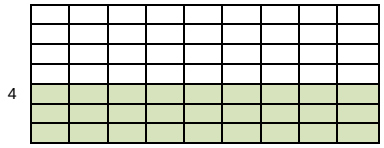

```
sicht = feld[ :, 7:8]
```



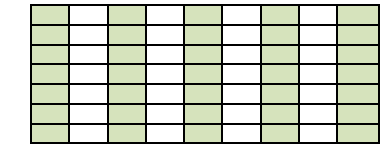

```
sicht = feld[ :2, 4:]
```



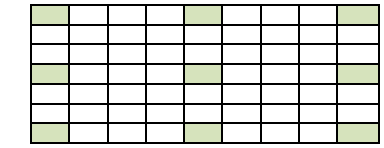

```
sicht = feld[ 4:, :]
```




```
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 hier 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 ausgedruckt.

```
>>>
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 Feldern 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!*

Oft braucht man in der Statistik aber Zufalls-Werte zwischen 0 und 1. Hierfür nutzen wir die Funktion **random()** aus der Bibliothek random.

```
from random import random

for i in range(10):
    print(random())
```

```
>>>
0.8337455442781
0.1755767061503858
0.3853434899800302
0.792967393485768
0.5036802632097241
0.34405052476700704
0.9737482138245568
0.1504658029464917
0.32370271239696813
0.8812756510874483
```

Wenn Sie das obige Programm ausprobieren sollten, dann erhalten Sie ganz sicher andere Werte. Das ist schließlich Sinn und Zweck eines Zufalls-Generators (Würfel's).
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.

Analyse des Würfel's mittels eines Dictonary (→ Teil 2)

```
from random import randint

Haeufigkeit = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}

# Würfelschleife
for i in range(60):
    wurf = randint(1, 6)
    haeufigkeit[wurf] += 1

# Ausgabe
print(haeufigkeit)
```

Testen mit 6, 60, 600, 6000, 60000, 600000
berechnen der prozentualen Abweichung von Zielwert (eines gerechten Würfel's)

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)

(c,p) 2015 - 2025 lsp: dre

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 !)
```

Literatur und Quellen:

siehe letzter Teil!

Abbildungen und Skizzen entstammen den folgende ClipArt-Sammlungen:

/A/

andere Quellen sind direkt angegeben.

Alle anderen Abbildungen sind geistiges Eigentum:

lern-soft-projekt: drews (c,p) 1997 – 2025 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 - 2025 lern-soft-projekt: drews -⊟
⊞- drews@lern-soft-projekt.de -⊟
⊞- http://www.lern-soft-projekt.de -⊟
⊞- 18069 Rostock; Luise-Otto-Peters-Ring 25 -⊟
⊞- Tel/AB (0381) 760 12 18 FAX 760 12 11 -⊟