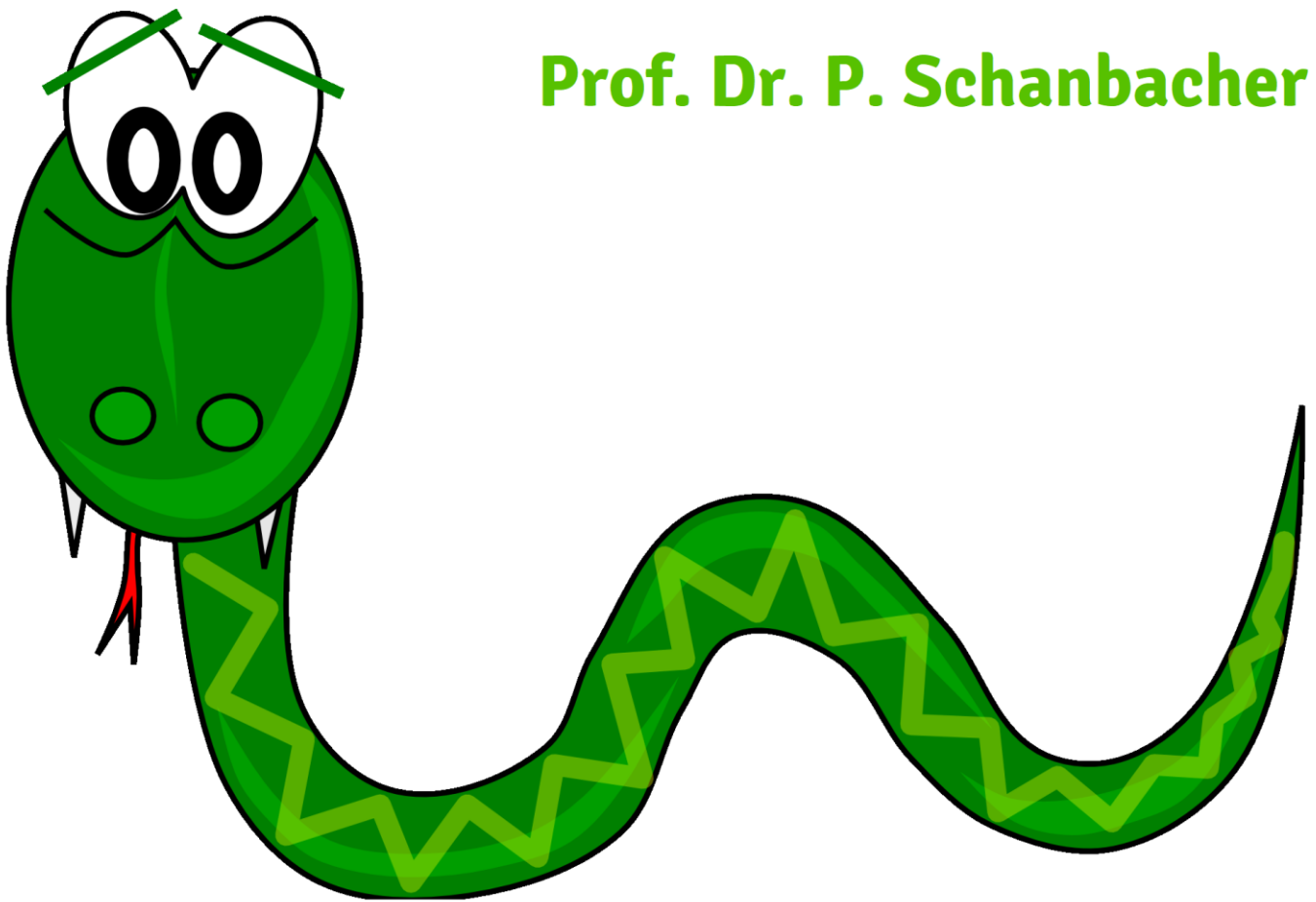


Schnell &
einfach

Python lernen - in nur einem Tag

Prof. Dr. P. Schanbacher



Python-programmieren.de

PYTHON LERNEN - in nur einem Tag

Inhalt

Einführung

Stunde 1: Installation

Stunde 2: Datenstrukturen

Stunde 3: Kontrollstrukturen

Stunde 4: Funktionen, Module, Pakete

Stunde 5: Hilfreiche Pakete

Stunde 6: Debuggen und Fehler

Stunde 7: Klassen

Stunde 8: Daten

Stunde 9: Grafiken (Matplotlib)

Lösungen

Bonus: Spickzettel

PYTHON LERNEN in nur einem Tag

Einführung

Diese Einführung ist für alle geschrieben, die wenig Text lesen und viel Python lernen wollen.

Warum Python?

- leicht zu lernen
- weit verbreitet
- kostenlos und Open Source
- Plattform unabhängig (Windows / Linux / MacOS)
- viele Pakete für Data Science

Lernstrategie

Dieses Buch ist zwar so geschrieben, dass man es in einem langen Arbeitstag (9 Stunden) durcharbeiten könnte. Dennoch ist es besser 9 Tage 1 Stunde zu lernen. Jedes Kapitel sollte ca. eine Stunde benötigen. Manche Kapitel aufgrund des großen Themas dauern evtl. etwas länger zum Durcharbeiten. Um Deine Zeit maximal effektiv zu nützen, wird keine Theorie und nur relativ wenig ausführliche Erläuterungen ausgeführt. Stattdessen werden prägnante Beispiele gezeigt, welche Du leicht nachprogrammieren kannst und Dir viel eher im Gedächtnis bleiben. Bei Beispielen wird zunächst der Code und danach die Ausgabe gezeigt (*AUSGABE*:). Mach Dir immer klar, was Du als Ausgabe erwartest und kontrolliere, ob dies der tatsächlichen Ausgabe entspricht.

Zur Vertiefung gibt es am Ende jedes Kapitels Aufgaben. Die Lösungen findest Du im Anhang. Nicht beim Fußball zuschauen macht einen Profispieler - sondern nur das selbst spielen. Genauso ist es beim Programmieren. Ein guter Programmierer wird nur, wer selbst viel programmiert und probiert. Viel Spaß und effektives Lernen mit diesem praxisnahen Buch!

Projekt

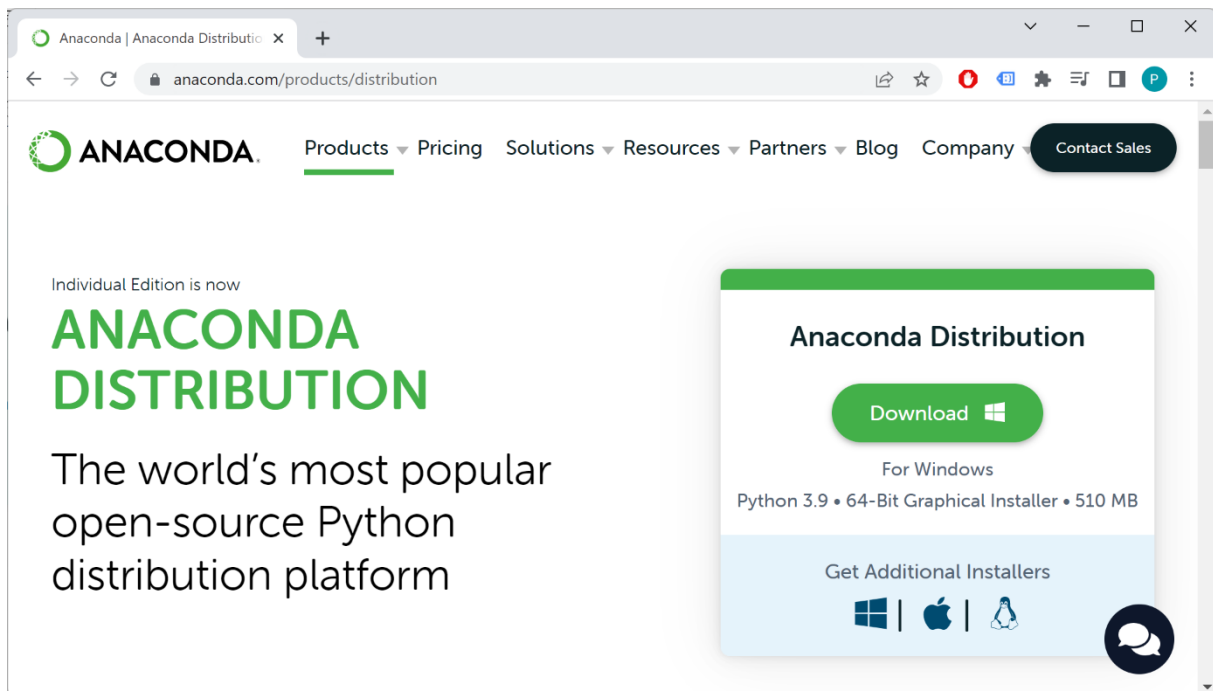
Ein Projekt, welches sich durch das ganze Buch zieht, ist das Telefonbuch. Ziel ist es für unterschiedliche Namen die Telefonnummer zu speichern. In fast jedem Kapitel wirst Du dem Projekt etwas hinzufügen. Versuche die Aufgaben des Projektes erstmal eigenständig zu lösen, bevor Du dir die (bzw. eine mögliche) Lösung anschaust.

Stunde 1: Installation

Anaconda Package

Um Programmieren zu können, benötigt man zunächst eine geeignete Entwicklungsumgebung (IDE = integrated development environment). Das kostenlose Anaconda Paket bietet eine erstklassige Lösung. Die richtige Version kann

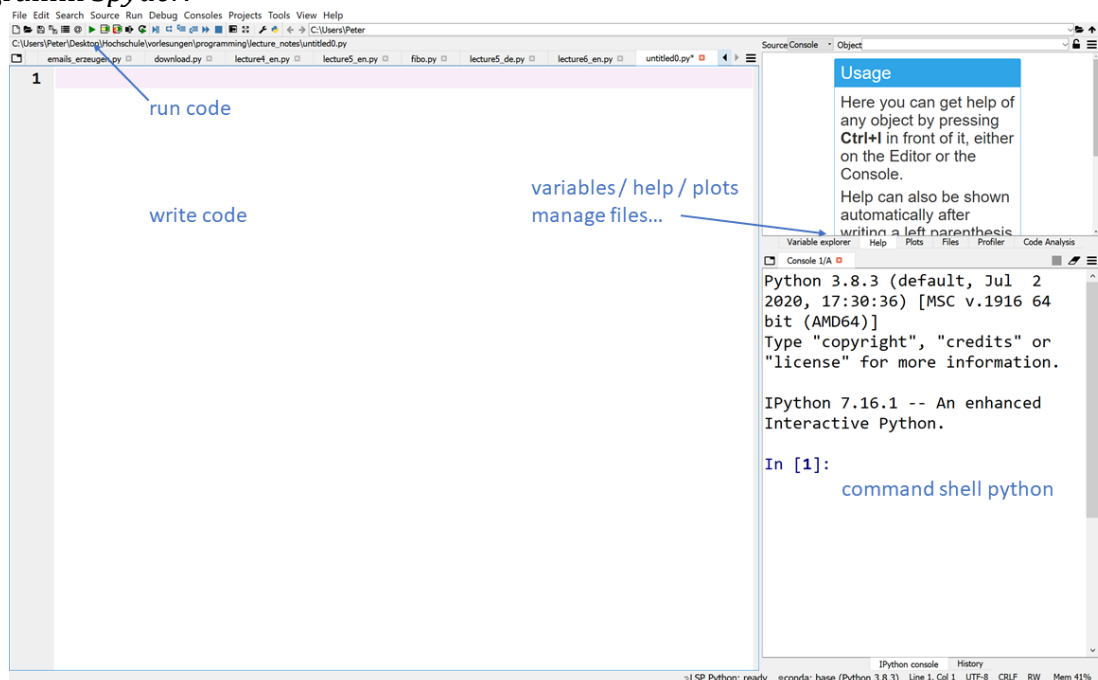
auf <https://www.anaconda.com/products/distribution> heruntergeladen werden (Windows, Linux, MacOS).



Führe die Installation mit den Standardeinstellungen aus. Eine Eingabe von persönlichen Daten oder gar Kreditkarteninformationen sind zur Installation der Privat-Version nicht notwendig.

Spyder

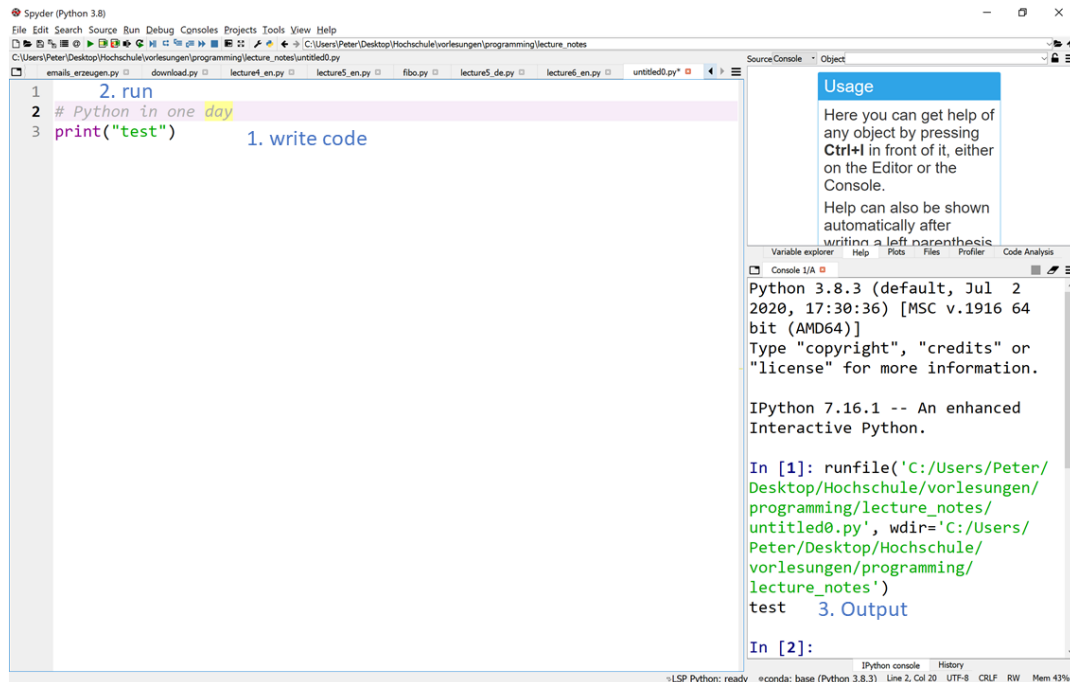
Öffne nach der Installation das Programm *Spyder*.



Es gibt ein Feld, in welchem Du Dein Programm schreibst (links). Daneben gibt es ein Fenster in welchem Du Hilfe, Variablen, Dateien, etc. anschauen kannst (oben rechts). Die Ausgabe (bzw. eine Command-Line für die Eingabe von einzelnen Befehlen) findet sich unten rechts.

erstes Programm

Schreibe nun Dein erstes Programm und gib `print("test")` in das große Eingabefeld ein (1). Drücke dann den grünen "Run"-Pfeil (2); oder gehe über das Menü (Run -> Run bzw. F5 drücken). Beobachte nun die Ausgabe `test` im Ausgabefeld (3).



Der `print` Befehl gibt einen Wert, Zahl oder sonstiges im Ausgabefeld aus.

Beachten die ebenfalls die Zeile `# Python in one day`. Mit dem `#` startet ein Kommentar in dieser Zeile. Dies hat keine Auswirkung auf das Programm, sondern dient nur als Notiz. Ein Programm sollte immer gut kommentiert werden. Insbesondere am Anfang sollte kurz beschrieben werden, was das Programm tut und wer es wann erstellt hat.

Variablen

Der große Vorteil eines Computers besteht darin ein Programm für unterschiedliche Werte schnell ausführen zu können. Deshalb sind Variablen welche meist durch den Benutzer oder den Programmierer gesetzt werden, der Kern jedes Programms. Eine Variable wird immer mit `variablenname = wert` gesetzt. Im Unterschied zu den meisten Programmiersprachen musst Du in Python eine Variable nicht explizit einem Typ zuordnen. Dies geschieht automatisch im Hintergrund. Gib folgendes in das Programm ein.

```
a = 1
b = 3.141 # Achtung . statt ,
c = True
```

In [21]:

```
d = "Some Text"
print(a,b,c,d)
```

AUSGABE:
1 3.141 True Some Text

Es werden 4 Variablen a,b,c,d erstellt. Diese decken auch die 4 wichtigsten Variabentypen ab:

- int (integer): a ist eine ganzzahlige Variable
- float: b ist eine Gleitkommazahl (mit Dezimaltrenner .)
- boolean: c ist ein Boolean Variable und kann nur die Werte True oder False annehmen
- string: d ist eine Text Variable. Der Text wird zwischen zwei " Zeichen (oder alternativ ' Zeichen) geschrieben.

Wähle im Fenster rechts oben den "Variable Explorer" aus. Nun siehst Du die definierten Variablen, den jeweiligen Typ und den zugehörigen Wert.

Name	Type	Size	Value
a	int	1	1
b	float	1	3.141
c	bool	1	True
d	str	1	Some Text

Variable explorer Help Plots Files Profiler Code Analysis

Mit dem print Befehl kannst Du die Variablen ausgeben. Die einzelnen Variablen müssen mit einem Komma getrennt werden.

In Python kannst Du auch eine Variable überschreiben. Bsp. mit c = "one" hast Du aus einer Integer Variable eine Text Variable (mit Text "one") gemacht. Es ist jedoch guter Stil eine Variable bei einem Typ zu belassen.

Die Variablennamen sind immer eine zusammenhängende Zeichenkette. meine Variable (mit Leerzeichen) ist somit kein zulässiger Variablenname. Im Gegensatz dazu ist meine_Variable oder meineVariable richtig. Gebe den Variablen einen Namen aus dem sich schließen lässt, wofür die Variable gut ist. meinSumme=10 ist somit besser als xyz123=10 (wobei beides möglich ist).

Berechnungen

Du kannst auch die gängigen Rechenoperationen ausführen.

```
e = a + b # e = 1 + 3.141
print(e)
```

AUSGABE:
4.141

berechnet $e = a + b = 1 + 3.141 = 4.141$. Die gebräuchlichsten Rechenoperationen sind:

In [4]:

Operator	Bedeutung	Beispiel
+	Addition	$2+3 = 5$
-	Subtraktion	$2-3 = -1$
*	Multiplikation	$2*3 = 6$
/	Division	$17/5 = 3.66$
//	Floor Division	$17//5 = 3$
%	modulo (Rest)	$17 \% 5 = 2$
**	Potenz	$2**3 = 8$
round()	Runden	$\text{round}(2.3) = 2$

Wie üblich gilt Punkt vor Stich. D.h. $3+2*5 = 13$.

Aufgabe 1.1: Setze eine Variable `x` als eine Zahl. Teste ob die Zahl gerade ist (Ausgabe 0) oder ungerade (Ausgabe 1).

Tipp: Die Modulo Operation. Die Lösung findest Du am Ende des Buchs.

Aufgabe 1.2: Definiere eine Variable mit (D)einer Größe (in m) und eine mit (D)einem Gewicht (in kg). Berechne den Body-Mass $\text{BMI} = \text{Gewicht} / \text{Größe}^2$. Diesen am besten auf eine Stelle runden. Gib am Ende aus: "Der Body-Mass-Index beträgt BMI".

String Operationen

Es ist auch teilweise möglich Rechenoperationen bei Strings zu verwenden.

Bsp. `print(3*"Py" + "thon")` führt zur Ausgabe "PyPyPython".

Auf einzelne Buchstaben kann per wie folgt zugegriffen werden. Hierbei bedeutet -1 der vorletzte Buchstabe. Wie in der Computerwissenschaft üblich fängt Python mit 0 an zu zählen. Somit bedeutet 0: vom ersten Buchstaben (Index 0) bis zum Ende. 1:4 bedeutet vom zweiten Buchstaben *inklusive* (Index 1) bis *exklusive* 5ten Buchstabe (Index 4).

In [24]:

```
txt = "Python"
print(txt[1:4]) # Buchstaben 2,3,4
print(txt[:-2]) # bis vorletzter Buchstabe
```

AUSGABE:

```
yth
Pyth
```

Beachte nochmals den Index der einzelnen Buchstaben von vorne bzw. von hinten gezählt.

P y t h o n

0 1 2 3 4 5

-6 -5 -4 -3 -2 -1

Folgende String Operationen sind oft hilfreich:

In [19]:

```
print("python".capitalize())      # Python, Großbuchstaben am Anfang
print("PythonPython".count("thon")) # 2, zählt Anzahl "thon"
print("python".find("y"))         # 1, position von "y"
print("tho" in "python")         # True, test ob "tho" in "python"
print("python".isnumeric())      # False, test ob "python" eine Zahl
print("python".replace('p','j')) # jython, ersetze "p" mit "j"
print("python".split('t'))       # ["py","hon"], split bei "t"
print(str().join(["p", "yt", "hon"])) # pyt -> verknüpft Texte
print("python".upper())         # nur Großbuchstaben
print(len("python"))            # 6, Anzahl Buchstaben
```

Aufgabe 1.3: Erstelle eine Variable mit Deinem vollen Namen. Selektiere Deinen Vornamen (finde das Leerzeichen) und gebe diesen aus.

Stunde 2: Datenstrukturen

Damit ein Programm überhaupt funktionieren kann, müssen meist mehrere Daten gespeichert und verarbeitet werden. Dies geschieht in Form von einfachen Variablen (bsp. Zahlen oder Texte) oder komplexeren Datenstrukturen (Listen, Matrizen, Wörterbüchern, etc.).

Es zahlreiche Datenstrukturen. Im Folgenden die 3 wichtigsten Datenstrukturen:

- Listen (lists): eine Auflistung verschiedener Werte. Beispielsweise eine Liste mit Namen.
- Wörterbücher (dictionaries): Eine Liste mit Schlüsseln (keys) und Werten (values). Beispielsweise Namen als Schlüssel und Telefonnummern als Werte.
- Mengen (sets). Eine ungeordnete Liste mit eindeutigen Einträgen.

Listen (Lists)

Der häufigste Datentyp sind Listen. Listen können als Folge von Komma getrennten Werten geschrieben und ausgegeben werden.

In [20]:

```
quadrade = [1,4,9,16,25]
print(quadrade)
```

AUSGABE:

```
[1, 4, 9, 16, 25]
```

Auf einzelne Einträge kann per [index] zugegriffen werden. Der erste Wert ist am Index [0]. Mit negativen Zahlen zählt man von hinten; d.h. der letzte Eintrag ist auf [-1]. Abschnitte werden mit [Start:Ende] selektiert (exklusive Eintrag an Position [Ende]).

In [21]:

```
print(quadrate[0])
print(quadrate[-1])
print(quadrate[0:3])
```

AUSGABE:

```
1
25
[1, 4, 9]
```

Bestehende Werte können verändert und neue Werte hinzugefügt werden.

In [23]:

```
quadrate = [1000,4,9,16,25]
quadrate.append(6**2)      # 6^2 dazu
quadrate[0] = 1            # korrigere ersten Wert 1000 zu 1
print(quadrate)
buchstaben = ["P", "Q", "R", "C"] # PQRC
buchstaben.append("D")      # neuer Eintrag D -> PQRCD
buchstaben.pop(0)           # lösche ersten Eintrag -> QRCD
buchstaben[0:2] = ["A", "B"] # überschreibe erste beide Werte ABCD
print(buchstaben)
```

AUSGABE:

```
[1, 4, 9, 16, 25, 36]
['A', 'B', 'C', 'D']
```

Alle Elemente können aus der Liste gelöscht werden, indem man diesem einfach eine leere Liste zuordnet.

In [25]:

```
buchstaben[:] = []
print(buchstaben)    # leer -> []
```

AUSGABE:

```
[]
```

Listen können verschachtelt werden. Beispielsweise mit

In [27]:

```
personen = [ ["Peter", 37], ["Conny", 35], ["Jonah", 32] ]
print(personen)
print(personen[1][0]) # [1] -> wähle von der zweiten Liste ["Conny", 35]
                     # [0] -> das erste Element "Conny"
```

AUSGABE:

```
[['Peter', 37], ['Conny', 35], ['Jonah', 32]]
Conny
```

Aufgabe 2.1: Mit folgendem Code erhältst Du eine Liste mit den Buchstaben Deines Namens. Verändere einige Buchstaben und füge die Buchstaben wieder zu einem String zusammen.

In [29]:

```
name = []
name[:] = "dein name"
# TODO: ändere einige Buchstabe und füge diese zusammen
```

Die Reihenfolge ist bei Listen relevant. Editiert man das erste Element, so ist entschieden, welches Element aktuell vorne steht. Die Reihenfolge kann beispielsweise umgedreht werden (letzte Element am Anfang); oder alphabetisch sortiert werden.

In [2]:

```
einkauf = ["Milch", "Apfel", "Apfel", "Käse", "Joghurt"]
einkauf.reverse()      # von hinten nach vorne
print(einkauf)
einkauf.sort()         # alphabetische Reihenfolge
print(einkauf)
```

AUSGABE:

```
['Joghurt', 'Käse', 'Apfel', 'Apfel', 'Milch']
['Apfel', 'Apfel', 'Joghurt', 'Käse', 'Milch']
```

Folgende Operationen sind für Listen hilfreich.

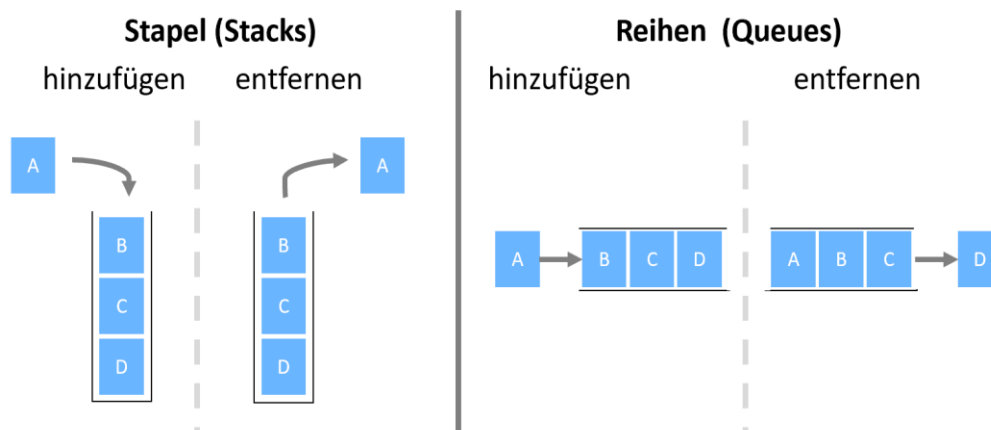
In [1]:

```
einkauf = ["Milch", "Apfel", "Apfel", "Käse", "Joghurt"]
print(einkauf.count("Apfel")) # 2 Apfel in einkauf
print(einkauf.index("Käse"))  # index 3
einkauf.append("Schokolade")  # füge Schokolade
einkauf.pop(3)                # lösche "Käse" at index 3
einkauf.remove("Joghurt")     # lösche Joghurt
print(einkauf)
einkauf.clear()               # einkauf=[]
```

AUSGABE:

```
2
3
['Milch', 'Apfel', 'Apfel', 'Schokolade']
```

Stapel (Stacks) und Reihen (Queues)



Listen können als Stapel (stacks) verwendet werden. Der letzte Teller der auf den Stapel gesetzt wurde, ist der erste Teller der wieder weggenommen wird (last-in, first-out).

In [96]:

```
stack = [1,2,3]
stack.append(4)
stack.append(5)
print("Append", stack)
stack.pop()      # lösche letztes Element
stack.pop()
print("Pop", stack)
```

AUSGABE:

```
Append [1, 2, 3, 4, 5]
Pop [1, 2, 3]
```

Listen können auch als Reihe (wie eine Warteschlange) verwendet werden. Die letzte Person die gekommen ist, ist die letzte die in den Bus einsteigt (first-in, first-out).

In [97]:

```
queue = ["A", "B", "C"]
queue.append("D")
queue.append("E")
print("Append", queue)
queue.pop(0)    # lösche erstes Element (index=0)
queue.pop(0)
print("Pop", queue)
```

AUSGABE:

```
Append ['A', 'B', 'C', 'D', 'E']
Pop ['C', 'D', 'E']
```

Wörterbuch

Wörterbücher (Dictionaries) enthalten Paare bestehend aus zusammengehören Schlüssel und Werte. Die Schlüssel müssen eindeutig sein. Beispiele sind (eindeutige) Postleitzahl und die dazugehörige Stadt. Ein Dictionary wird mit geschweiften Klammern erstellt {key1:value1, key1:value1}.

In [51]:

```
plz = {10115:"Berlin", 80331:"München", 70173:"Stuttgart",
70180:"Stuttgart"}
print(plz)
```

AUSGABE:

```
{10115: 'Berlin', 80331: 'München', 70173: 'Stuttgart', 70180: 'Stuttgart'}
```

Elemente können hinzugefügt, geändert oder gelöscht werden mit

In [52]:

```
plz[10115] = "Berlin Mitte"    # ändere Eintrag
plz[20095] = "Hamburg"         # füge neuen Eintrag hinzu
del plz[80331]                 # lösche München
print(plz)
```

AUSGABE:

```
{10115: 'Berlin Mitte', 70173: 'Stuttgart', 70180: 'Stuttgart', 20095: 'Hamburg'}
```

Teste, ob ein Element in der Liste ist und gebe alle Werte und Schlüssel aus:

In [55]:

```
print("New York" in plz)    # NY ist nicht im dict
print(list(plz.keys()))    # list der Städte
print(list(plz.values()))  # list der PLZ
print("Die Postleitzahl 20095 hat", plz[20095])
```

AUSGABE:

```
False
[10115, 70173, 70180, 20095]
['Berlin Mitte', 'Stuttgart', 'Stuttgart', 'Hamburg']
Die Postleitzahl 20095 hat Hamburg
```

Aufgabe 2.2: Erstelle ein Dictionary mit dem Namen und Alter Deiner Freunde. Gib die Namen aus.

Mengen

Eine Menge (Set) ist eine ungeordnete Liste eindeutiger Elemente (d.h. doppelte Elemente werden nicht mehrfach berücksichtigt). Mengen werden ähnlich zu Listen definiert; jedoch mit einer geschweiften Klammer {item1, item2}.

In [57]:

```
korb = {"Milch", "Apfel", "Apfel", "Apfel", "Apfel", "Apfel"}
print(korb)                # nur ein Apfel
print("Banane" in korb)    # keine Banane im Korb
```

AUSGABE:

```
{'Apfel', 'Milch'}
False
```

Mengen Operationen sind unter anderem ohne -, oder |, und &.

In [60]:

```
einkaufsliste = {"Apfel", "Käse", "Joghurt"}
print("ohne:", korb - einkaufsliste) # elemente in korb aber nicht in liste
print("oder:", korb | einkaufsliste) # element in korb oder liste (oder beides)
print("und:", korb & einkaufsliste) # elements in korb und liste
```

AUSGABE:

```
ohne: {'Milch'}
oder: {'Apfel', 'Käse', 'Milch', 'Joghurt'}
und: {'Apfel'}
```

Löschen

Mit dem del (für delete) lassen sich einzelne oder mehrere Elemente aus einer Datenstruktur löschen. Beispiele für das Löschen bei Listen und Wörterbücher sind hier:

In [62]:

```
# Listen
meine_liste = ["A","B","C","D","E"]
del meine_liste[1:3] # löscht Elemente an Index 1 und 2 ("B", "C")
del meine_liste[-1] # löschen vom letzten Element "E"
print("Liste:",meine_liste)    # "A", "D"

# Wörterbuch
plz = {10115:"Berlin", 80331:"München", 70173:"Stuttgart",
70180:"Stuttgart"}
del plz[10115]    # lösche Berlin
print("Wörterbuch:",plz)

AUSGABE:
Liste: ['A', 'D']
Wörterbuch: {80331: 'München', 70173: 'Stuttgart', 70180: 'Stuttgart'}
```

Aufgabe 2.3: Bestimme die Anzahl der eindeutige Buchstaben Deines Namens. Gib die Buchstaben Deines Namens ohne Vokale (aeiou) aus.

Echte Kopien

Python legt nur echte Kopien von Datentypen an, wenn es der Programmierer explizit mit slicing [:] oder .copy verlangt. Am besten wird es mit folgendem Code klar.

In [4]:

```
meine_liste = [0,1,2]
tmp = meine_liste    # keine echte Kopie
tmp[0] = 1000        # verändert auch meine_liste
print(meine_liste)    # [1000,1,2]
```

```
AUSGABE:
[1000, 1, 2]
```

eine echte Kopie wird wie folgt erstellt

In [5]:

```
meine_liste = [0,1,2]
tmp = meine_liste[:] # oder meine_liste.copy()
tmp[0] = 1000        # does not change my_list
print(meine_liste)    # [0,1,2]
```

```
AUSGABE:
[0, 1, 2]
```

Dieses Verhalten führt oft zu Fehlern. Man erzeugt eine (unechte) Kopie und ändert diese. Später möchte man auf die ursprüngliche Variable zurückgreifen und stellt fest, dass sich diese geändert hat.

Inputs

Per input kann man den Benutzer einen String eingeben lassen. Der Text erscheint in der Konsole, wo der Benutzer die Eingabe machen kann.

In [16]:

```
a = input("Bitte etwas eingeben: ") # bsp. Eingabe "Hallo"
print("Eingabe war",a)              # Ausgabe
```

AUSGABE:

```
Bitte etwas eingeben: hallo
Eingabe war hallo
```

Möchte man die Eingabe in eine Zahl umwandeln kann man dies mit int oder float erreichen.

In [18]:

```
a = int(input("Wie alt sind Sie?"))
print("Sie sind",a, "Jahre alt.")
```

AUSGABE:

```
Wie alt sind Sie?132
Sie sind 132 Jahre alt.
```

Wird statt einer Zahl, ein Text (bsp. *Hallo*) eingegeben, so kann dieser nicht in eine Zahl umgewandelt werden. Das Programm endet mit einer Fehlermeldung.

Projekt:

1. Erstelle ein Telefonbuch (als Wörterbuch) mit dem Namen und Telefonnummer von einigen Freunden.
2. Füge zwei Inputs hinzu, um einen zusätzlichen Namen und Telefonnummer einzulesen. Füge den Namen und Telefonnummer dem Telefonbuch hinzu.
3. Füge einen weiteren Input hinzu und lösche diese im Telefonbuch.

Hinweis: Verwende einen String um eine Telefonnummer richtig zu speichern (bsp. mit "+49.." oder eine führenden Null "0178")

In [2]:

```
telefonbuch = {"Tick":"+4912345", "Trick":"01781234", "Track":"01798765"}
name = input("Name zum hinzufügen:")
phone = input("Nummer:")
telefonbuch[name]=phone
name_loeschen = input("Name zum löschen:")
del telefonbuch[name_loeschen]
print(telefonbuch)
```

AUSGABE:

```
Name zum hinzufügen:Donald
Nummer:012345
Name zum löschen:Tick
{'Trick': '01781234', 'Track': '01798765', 'Donald': '012345'}
```

Stunde 3: Kontrollstrukturen

Kontrollstrukturen sind das Herz des Programmierens. Es gibt Abfragen (if-Statements) und Schleifen (for und while). Für beides ist es wichtig zwei Variablen bzw. eine Variable mit einem Wert vergleichen zu können.

Vergleiche

Ein einfaches Gleichheitszeichen = ist eine Zuweisung. `a=5` bedeutet, dass die Variable den Wert 5 annimmt. Ein doppeltes Gleichheitszeichen `==` ist eine Abfrage. `a==5` entspricht der Frage, *ist a 5?*. `a==5` nimmt den Wert wahr (`True`) an, sofern die Variable `a` tatsächlich den Wert 5 besitzt. Anderenfalls ist `a==5` falsch (`False`). Ebenso kann getestet werden ob eine Variable kleiner `<`, kleiner gleich `<=`, größer `>` oder ungleich `!=` einem anderen Wert ist. Auch zwei Variablen können so verglichen werden.

In [7]:

```
a = 5          # a bekommt Wert 5
b = a+1        # b=6
print(a == 5)  # hat a Wert 5? -> True
print(a < 5)   # ist a<5? -> False
print(a != b)  # 5 ungleich 6? -> True
```

AUSGABE:

```
True
False
True
```

Bei Strings ist der Vergleich in alphabetischer Reihenfolge. So kommt `"A"` kleiner als `"Z"`, da es weiter vorne im Alphabet kommt.

In [10]:

```
print("A" < "Z")    # True
print("A" != "A")   # False
```

AUSGABE:

```
True
False
```

Bei Listen geht der Vergleich elementweise durch und entscheidet beim ersten unterschiedlichen Element. Verglichen wird das *ite* Element der einen Liste mit dem *iten* Element der anderen Liste.

In [5]:

```
print([1,2,3] < [1,2,4]) # True
print([1,2,3] < [2,0,0]) # True (1<2)
print([1,2,3] < [1,0,9]) # False 2>0
```

AUSGABE:

```
True
True
False
```

Vergleiche kann man auch sequentiell durchführen, so bedeutet z.B. $a < b < c$, dass die Bedingung gelten muss $a < b$ *und* $b < c$.

In [33]:

```
print(1<2<3)  # -> 1<2  und  2<3
```

```
AUSGABE:  
True
```

Man kann verschiedene Bedingungen mit **and** (und), **or** (oder), **not** (nicht) verknüpfen. Die Bedingungen verhalten sich wie im normalen Sprachgebrauch. Eine korrekte Aussage ("gerade lerne ich Python") *und* (and) eine falsche Aussage ("gerade lerne ich Kobra") gibt insgesamt eine falsche Aussage.

In [63]:

```
print(True and False) # I learn python and kobra -> False  
print(True or False)  # I learn python or kobra -> True  
print(False or True and not False) # -> False or (True and (not False))
```

```
AUSGABE:  
False  
True  
True
```

Beim letzten Beispiel ist zu beachten, dass die Operatoren unterschiedliche Prioritäten haben. **Not** hat die höchste Priorität und **or** die tiefste Priorität. Also wird zunächst das **not** Statement ausgeführt (not False wird zu True), danach das **and** Statement (True and True wird zu True). Am Ende wird das **or** Statement ausgeführt (False or True wird zu True).

If Statement

Um eine Abfrage zu machen wird das If-Statement verwendet. Es hat die Struktur If... elif... elif... else.

```
If Bedingung1:  
    Aktion1 # wenn Bedingung1 erfüllt ist.  
elif Bedingung2:  
    Aktion2 # wenn Bedingung2 (aber nicht Bedingung1) erfüllt ist.  
elif Bedingung3:  
    Aktion3 # wenn Bedingung3 (aber Bedingung1 oder Bedingung2) erfüllt  
ist.  
else:  
    Aktion4 # wenn weder Bedingung1, Bedingung2 noch Bedingung3
```

Die einzelnen Blöcke der Aktionen müssen mit der Tab Taste  eingerückt sein. So weiß Python wie viele Zeile ausgeführt werden müssen, wenn eine Bedingung erfüllt ist. Es ist auch möglich eine nur eine einzelne If-Abfrage ohne elif oder else auszuführen (andersrum jedoch nicht).

Folgendes Beispiel lässt eine Zahl eingeben. Eine negative Zahl wird auf 0 gesetzt. Außerdem wird ausgegeben, ob die eingegebene Zahl negativ, null oder positiv ist.

In [33]:

```
a = int(input("Bitte ganze Zahl eingeben:"))  
if a<0:
```

```

    a = 0
    print("Negative Zahl wurde auf 0 gesetzt.")
elif a == 0:
    print("Zahl ist 0")
else:
    print("Zahl ist positiv")

```

AUSGABE:

Bitte ganze Zahl eingeben:324
Zahl ist positiv

Ein weiteres Beispiel:

In [22]:

```

a = int(input("Ihr Alter: "))
if a < 18:
    print("Sie sind noch minderjährig")
elif a < 67:
    print("Sie sind noch nicht im regulären Rentenalter")
else:
    print("Ihnen einen schönen Ruhestand")

```

AUSGABE:

Ihr Alter: 324
Ihnen einen schönen Ruhestand

Aufgabe 3.1: Lasse Dir vom Benutzer eine positive oder negative Zahl eingeben. Gib den absoluten Wert der Zahl aus (d.h. -3 wird zu 3).

Aufgabe 3.2: Lasse Dir vom Benutzer einen Namen eingeben. Frage ab, ob der Name "Tony", "Tom" oder "Tina" ist.

Projekt (bisher): Unser bisheriges Projekt sieht wie folgt aus.

In []:

```

telefonbuch = {"Tick":"+4912345", "Trick":"01781234", "Track":"01798765"}
name = input("Name zum hinzufügen:")
phone = input("Nummer:")
telefonbuch[name]=phone
name_loeschen = input("Name zum löschen:")
del telefonbuch[name_loeschen]
print(telefonbuch)

```

Projektaufgabe: Passe das Projekt wie folgt an: Es soll eine Auswahlfunktion geben, bei welcher der Benutzer nach einer Auswahl gefragt wird (er soll eine Zahl von 1-4 eingeben):

1. ein neuer Namen und Telefonnummer soll hinzugefügt werden.
2. ein Name soll gelöscht werden.
3. ein Name soll gesucht werden und die dazugehörige Telefonnummer ausgegeben werden.
4. das Telefonbuch soll ausgegeben werden.

In [15]:

```
telefonbuch = {"Tick":"+4912345", "Trick":"01781234", "Track":"01798765"}

print("\nBitte auswählen:")
print("1: Person hinzufügen")
print("2: Person entfernen")
print("3: Person suchen")
print("4: alle Personen anzeigen")
x = input()
if x == "1":    # füge Person hinzu
    name = input("Name zum hinzufügen:")
    phone = input("Nummer:")
    telefonbuch[name] = phone
elif x == "2":  # lösche Person
    name = input("Name zum entfernen:")
    if name in telefonbuch: # checke ob Person im Telefonbuch
        del telefonbuch[name]
elif x == "3":  # Suche Person
    name = input("Name zum suchen:")
    if name in telefonbuch:
        print(name, "hat die Nummer", telefonbuch[name])
elif x == "4":  # gebe Telefonbuch aus
    print(telefonbuch)
else:
    print("ungültige Auswahl.")

print("Ende:", telefonbuch)
```

AUSGABE:

```
Bitte auswählen:
1: Person hinzufügen
2: Person entfernen
3: Person suchen
4: alle Personen anzeigen
2
Name zum entfernen:Trick
Ende: {'Tick': '+4912345', 'Track': '01798765'}
```

While Loop

Mit einer While-Schleife kann man eine dauernde Ausgabe kontinuierlich wiederholen, solange eine Bedingung erfüllt ist. Es ist wichtig sicherzustellen, dass die Bedingung irgendwann mal nicht mehr erfüllt ist. Ansonsten läuft die Schleife für immer (das Programm kann mit Ctrl + C abgebrochen werden oder in Spyder auf das rote Rechteck bei der Konsole). Die Struktur ist wie folgt:

```
index = 0
while BEDINGUNG:
    TUE ETWAS
    index = index+1
```

Alles was im While Block ausgeführt werden soll, muss wie bei der If-Bedingung eingerückt sein. Folgendes Programm läuft solange $a < 10$. In jedem Schritt wird a um eines erhöht, damit die Bedingung irgendwann verletzt ist und $a \geq 10$ ist. Der Aktionsblock der While Schleife ist eingerückt und wird mehrmals ausgeführt. Der

Nachfolgende Code, welcher nicht mehrmals ausgeführt werden soll, ist nicht mehr eingerückt.

In [30]:

```
a = 0
while a < 10:
    a = a+1
    print(a, end=",")
print("ende")
```

AUSGABE:

1,2,3,4,5,6,7,8,9,10,ende

Anmerkung: normalerweise endet der Print Befehl mit einer neuen Zeile (d.h. end = "\n"). Damit die Ausgabe nicht mit 10 Zeilen erfolgt, werden die Ausgaben nur mit einem Komma getrennt (d.h. end = ","). Alternativ könnte man die Bedingung auch anpassen. Der folgende Code verwendet anstatt a<10, solange a nicht 10 ist.

In [29]:

```
a = 0
while not(a==10):
    a = a+1
    print(a, end=",")
```

AUSGABE:

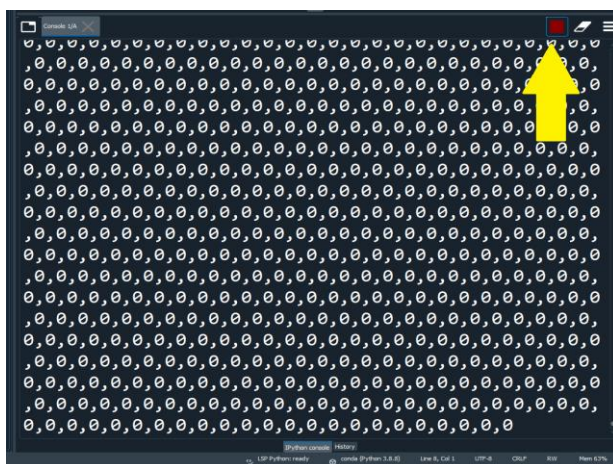
1,2,3,4,5,6,7,8,9,10,

Stell sicher, dass die Bedingung irgendwann nicht mehr erfüllt ist. Ein häufiger Fehler ist zu vergessen, den Index zu erhöhen.

In []:

```
a = 0
while a<10:
    # ERROR: hier fehlt a = a + 1
    print(a, end=",")
```

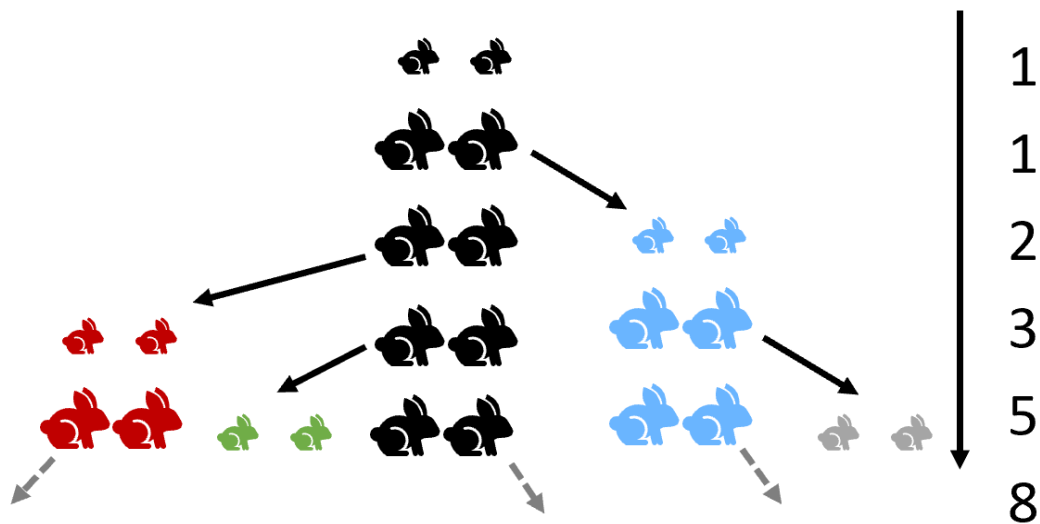
Das Programm endet niemals, da a nicht um eins erhöht wird. Mit CTRL+C oder einem Klick auf das rote Quadrat kann das Programm unterbrochen werden.



Aufgabe 3.3: Lasse den Benutzer so lange sein Alter eingeben, bis die Eingabe positiv ist; d.h. bei negativem Alter wird erneut gefragt.

Ein kompliziertes Beispiel sind die Fibonacci Zahlen. Diese starten mit 1 und 1. Die nächste Zahl ist jeweils die Summe der beiden vorangegangenen Zahlen. Die ersten Fibonacci Zahlen sind 1,1,2,3,5. Die nächste Zahl wäre die 8 als Summe von 3 und 5. Fibonacci-Zahlen treten auch in der Natur auf. Beispiel:

1. ein Hasenpaar braucht einen Monat, um geschlechtsreif zu werden
2. jedes erwachsene Hasenpaar wirft jeden Monat ein neues Paar Hasenkinder



Folgendes Programm berechnet alle Fibonacci Zahlen unter 100:

In [18]:

```
a = 0
b = 1

while b < 100:
    print(b, end=",")
    bneu = a+b # nächste Zahl = Summe der vorangegangenen Zahlen
    a = b      # a um eines verschieben
    b = bneu   # b um eines verschieben
```

AUSGABE:

1,1,2,3,5,8,13,21,34,55,89,

Ende Durchlauf Wert von a und b

1 a=1, b=1+1=2

2 a=b=2, b=2+1=3

3 a=b=3, b=3+2=5

Aufgabe 3.4: Was ist das Ergebnis, wenn man alle Zahlen von 0 bis 100 addiert; d.h. $0+1+2+3+4+\dots+99+100=?$

Bemerkung: Diese Frage wurde Gauss in der Grundschule gestellt. Er hat die Antwort sofort, da er die Zahlen geeignet umstellte:

$0+1+2+3+4+\dots+97+98+99+100=$
 $(0+100) + (1+99) + (2+98) + (3+97) + \dots + (49+51) + 50 =$
 $100 + 100 + 100 + \dots + 100 + 50 = 50 \cdot 100 + 50 = \mathbf{5050}.$

For Schleife

Eine For Schleife ist ähnlich zu einer While Schleife. Bei der For Schleife ist jedoch bekannt wie viele Runden durchlaufen werden müssen. Will man beispielsweise alle Fibonacci Zahlen unter 1000 erhalten, würde man die While Schleife verwenden. Es ist (für die meisten) unklar wie viele Runden das Programm läuft (es wären 16). Eine For Schleife würde man eher bei der Aufsummierung aller Zahlen von 0 bis 100 verwenden. Hier ist bekannt dass 101 Runden (inklusive 0) benötigt werden. Die Struktur einer For-Schleife ist wie folgt:

```
for index in Iterierbares_Objekt:
    TUE ETWAS
```

ein Beispiel ist:

In [28]:

```
for i in [0,1,2,3,4]:
    print(i, end=",")
```

AUSGABE:

0,1,2,3,4,

Will man beispielsweise wissen, wie viele Buchstaben jedes Wort eines Satzes hat, kann man folgendes verwenden:

In [27]:

```
satz = "Heute lerne ich Python programmieren"
woerter = satz.split()
for w in woerter:
    print("'",w, "' hat ", len(w), " Buchstaben ", sep="")
```

AUSGABE:

```
'Heute' hat 5 Buchstaben
'lerne' hat 5 Buchstaben
'ich' hat 3 Buchstaben
'Python' hat 6 Buchstaben
'programmieren' hat 13 Buchstaben
```

Bemerkung: `print(a,b)` gibt a und b mit einem Leerzeichen getrennt aus.
Mit `print(a,b,sep="")` wird kein Leerzeichen ausgegeben.

Range Statement

Um schnell durch etwas zu iterieren, kann man den `Range(start, stop, schritte)` verwenden. Wird nur ein Wert angegeben, dann ist das das Ende `Range(stop)`. Wie bei Python üblich geht man von `start` bis zu `stop-1`.

In [32]:

```
for i in range(10): # von 0 bis 9
    print(i, end=", ")
```

AUSGABE:
0,1,2,3,4,5,6,7,8,9,

Der nachfolgende Code geht von -10 bis 110 (exclusive) in 30iger Schritten: -10,20,50,80 (ohne 110).

In [35]:

```
for i in range(-10,110,30):
    print(i,end=", ")
```

AUSGABE:
-10,20,50,80,

Aufgabe 3.5: Gib alle geraden Zahlen von 0 bis 10 aus.

Aufgabe 3.6: Berechne die Summe aller Zahlen von 1 bis 100 (d.h. $1+2+3+\dots+100=?$) mit einer For Schleife.

break & continue

Um das Verhalten in einer Schleife zu beeinflussen können `break` und `continue` verwendet werden.

break

Das `break` statement beendet die Schleife komplett. Folgendes Beispiel verdeutlicht dies:

In [6]:

```
i = 0
while True: # Bedingung immer True -> Schleife läuft für immer
    i = i+1
    print(i, end=", ")
    if i > 10:
        break # beende Schleife
```

AUSGABE:
1,2,3,4,5,6,7,8,9,10,11,

wir sehen, dass die While Schleife für immer ausgeführt wird (Bedingung = True). Um irgendwann doch zu einem Ende zu gelangen wird die Schleife abgebrochen (`break`)

wenn i über 10 ist. Das `break` statement wird 2-mal eingerückt, da es sich sowohl in der `while` Schleife als auch in der `If`-Abfrage befindet.

continue

Das `continue` statement führt den Rest des Durchlaufs nicht weiter aus, sondern springt sofort zum nächsten Durchlauf. Dies siehst Du an dem folgenden Beispiel, welches ausgibt ob eine Zahl gerade oder ungerade ist. Es ist zu beachten, dass das `continue` statment doppelt eingerückt ist; einmal für die `for`-Schleife und einmal für das `If`-Statment.

In [51]:

```
for num in range(2,10):
    if num%2 == 0: # num ist teilbar durch 2
        print(num,"ist gerade")
        continue # springe direkt zum nächsten Durchlauf ohne den Rest
weiter auszuführen
        print(num,"ist ungerade")
```

AUSGABE:

```
2 ist gerade
3 ist ungerade
4 ist gerade
5 ist ungerade
6 ist gerade
7 ist ungerade
8 ist gerade
9 ist ungerade
```

Beispiel: Primzahlen

Schaue Dir ein komplexeres Beispiel an. Hier werden verschiedene Elemente (`for`, `if`, `break`) zusammengebracht. Eine Primzahl n ist eine Zahl die sich durch keine Zahl von 2 bis $n-1$ teilen kann (d.h. nur durch sich selbst und 1 teilbar ist). Als Beispiel dafür ist 7 eine Primzahl, da $7 \bmod x \neq 0$ für alle $x \in 2, 3, 4, 5, 6$.

9 ist jedoch keine Primzahl da $9 \bmod 3 = 0$, d.h. 9 ist durch 3 teilbar. Um zu testen, ob eine Zahl eine Primzahl ist, muss man also einen Teiler für die Zahl finden. Konkret wird ein x gesucht, für welches $n \bmod x = 0$. Im folgenden Code wird zunächst angenommen, dass n eine Primzahl ist. Wenn ein Teiler von n gefunden wird, ist n keine Primzahl (`prim=False`). Die Schleife kann direkt abgebrochen werden. Falls die Schleife komplett durchlaufen wird, ohne einen Teiler zu finden, ist n eine Primzahl und die Variable `prim` bleibt `True`. Am Ende wird noch ausgegeben, ob n eine Primzahl ist, oder nicht (`If`).

In [45]:

```
n = 7
prim = True
for x in range(2,n): # von 2 bis n-1
    if n%x == 0:      # ist x ein Teiler von n?
        prim = False
        print(n, "=", x, "*", n//x)  # n = x * n/x
        break
if prim == True:
    print(n, "ist eine Primzahl")
else:
```

```
print(n, "ist keine Primzahl")
```

AUSGABE:
7 ist eine Primzahl

Aufgabe 3.7: Gib alle Primzahlen zwischen 0 und 100 an.

Tipp: Setze den besprochenen Primzahlen-Code in eine Schleife von 0 und 100. Passen den Code danach leicht an.

Aufgabe 3.8: Frage den Benutzer wiederholt nach eine Pizza-zutat. Höre auf, sobald der Benutzer keine Angaben mehr macht (`input=""`). Gib alle Pizzazutaten aus (bsp. `["Käse", "Schinken", "Tomaten"]`)

Iterieren durch Datenstrukturen

Du hast bereits Listen, Wörterbücher und Sets kennengelernt. Folgende Beispiele zeigen, wie Du durch diese Iterieren kannst.

In [52]:

```
meine_liste = ["Tim", "Tom", "Tina"]
mein_woerterbuch = {"a":1, "b":2, "c":3}
mein_set = {"x", "y", "z"}
```

```
for l in meine_liste:
    print(l, end=",")
print("<- Listen")
```

AUSGABE:
Tim,Tom,Tina,<- Listen

```
for w in mein_woerterbuch:
    print(w, "hat den Wert", mein_woerterbuch[w], end=",")
print("<- Wörterbuch")
```

AUSGABE:
a hat den Wert 1,b hat den Wert 2,c hat den Wert 3,<- Wörterbuch

```
for s in mein_set:
    print(s, end=",")
print("<- Menge")
```

AUSGABE:
x,y,z,<- Menge

List Comprehension

Ein schneller Weg um durch Listen zu iterieren oder diese zu erzeugen ist mit der Kontrollsequenz `[FUNK(x) for x in LISTE if BEDINGUNG]`. Hierzu folgende Beispiele:

In [54]:

```
vec = [-4, -2, 0, 3, 5]
print("Originalliste", vec)
print("Doppelte Werte:", [x*2 for x in vec])
print("Ohne Negative:", [x for x in vec if x >= 0])
```

```
print("Function abs(.):", [abs(x) for x in vec])
```

AUSGABE:

```
Originalliste [-4, -2, 0, 3, 5]
Doppelte Werte: [-8, -4, 0, 6, 10]
Ohne Negative: [0, 3, 5]
Function abs(.): [4, 2, 0, 3, 5]
```

Bei Wörterbüchern funktioniert dies ähnlich:

In [55]:

```
print("Wörterbuch:", [{x:x**2} for x in range(6)])
```

AUSGABE:

```
Wörterbuch: [{0: 0}, {1: 1}, {2: 4}, {3: 9}, {4: 16}, {5: 25}]
```

Dies kann auch mit mehreren Werten gerechnet werden. Beispielsweise wie viele Kombinationen zwei Würfel zu werfen gibt es, bei denen die Würfel unterschiedliche Augenzahlen aufweisen?

In [59]:

```
wurf = [(x,y) for x in range(1,7) for y in range(1,7) if x!=y]
print(wurf)
print("Es gibt", len(wurf), "Möglichkeiten")
```

AUSGABE:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (2, 3), (2, 4), (2, 5), (2, 6), (3, 1), (3, 2), (3, 4), (3, 5), (3, 6), (4, 1), (4, 2), (4, 3), (4, 5), (4, 6), (5, 1), (5, 2), (5, 3), (5, 4), (5, 6), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5)]
Es gibt 30 Möglichkeiten
```

Task 3.9: Erstelle eine Liste der Quadratzahlen $0^2, 1^2, 2^2, \dots, 10^2$.

Projektaufgabe: Setze das bisherige Projekt in eine Schleife. Beende die Schleife wenn der Benutzer "0" als Input bei der Auswahl eingibt. D.h. die Auswahlmöglichkeiten sind:

1. Beende das Programm
2. ein neuer Name und Telefonnummer soll hinzugefügt werden.
3. ein Name soll gelöscht werden.
4. ein Name soll gesucht werden und die dazugehörige Telefonnummer ausgegeben werden.
5. das Telefonbuch soll ausgegeben werden.

In []:

```
x = None # wähle irgendein Startwert für x
          # (aber nicht "0", da Schleife sonst beendet wird)
while x!="0": # solange die Eingabe nicht "0" ist,...
    print("\nBitte auswählen:")
    print("0: Programm beenden")
```

```
print("1: Person hinzufügen")
print("2: Person entfernen")
print("3: Person suchen")
print("4: alle Personen anzeigen")
x = input()
if x == "1":
    name = input("Name zum hinzufügen:")
    phone = input("Nummer:")
    telefonbuch[name] = phone
elif x == "2":
    name = input("Name zum entfernen:")
    if name in telefonbuch:
        del telefonbuch[name]
elif x == "3":
    name = input("Name zum suchen:")
    if name in telefonbuch:
        print(name, "hat die Nummer", telefonbuch[name])
elif x == "4":
    print(telefonbuch)
elif x == "0":
    print("Auf Wiedersehen!")
else:
    print("ungültige Auswahl.")
```

Stunde 4: Funktionen und Module

Bisher hast Du den Code Zeile für Zeile geschrieben. Hierbei gibt es zwei Nachteile:

1. will man mehrmals etwas ähnliches ausführen, muss der Code kopiert werden.
2. langer Code wird sehr unübersichtlich. Wäre der gesamte Code von großen Projekten in einer Datei wäre der Code praktisch nicht zu warten. Mehrere Personen könnten nicht an einem Projekt schreiben und man müsste jedes Mal die Stelle suchen, welche man anpassen möchte.

Aus diesem Grund gibt es die Möglichkeit einen Teil des Codes in eine Funktion oder sogar in ein Modul auszulagern. Diese Möglichkeiten werden im Folgenden diskutiert.

Funktionen

Funktionen werden mit dem Begriff `def FUNKTIONSNAME()` definiert. Sobald eine Funktion einmal definiert ist kann die Funktion mit `FUNKTIONSNAME()` aufgerufen werden. Das folgende Beispiel verdeutlicht dies:

In [52]:

```
def hallo():  
    print("Guten Tag")  
  
hallo() # aufruf der Funktion
```

AUSGABE:
Guten Tag

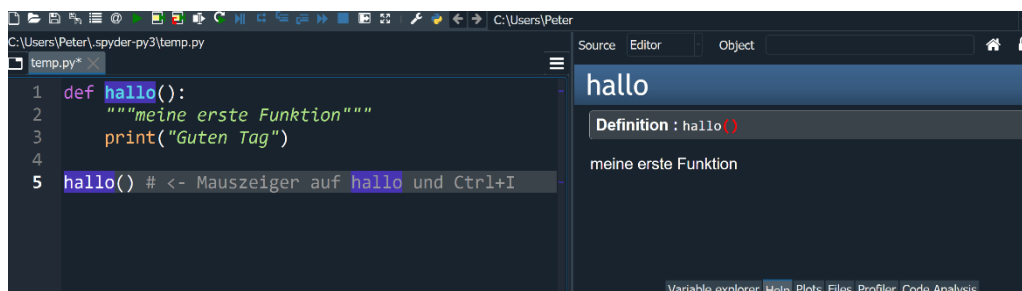
Insbesondere wenn Du einen längeren Code hast, den Du mehrmals an unterschiedlichen Stellen ausführen möchtest, ist es hilfreich diesen nur mit einem Befehl (hier `hallo()`) aufrufen zu können.

Du kannst auch eine Beschreibung mit `"""Beschreibung"""` hinzufügen.

In []:

```
def hallo():  
    """meine erste Funktion""" # <- Beschreibung  
    print("Guten Tag")  
  
hallo() # <- Mauszeiger auf hallo und Ctrl+I
```

Wenn Du nun mit dem Mauszeiger auf den letzten Aufruf `hallo()` gehst und `Ctrl+I` drückst, öffnet sich ein Fenster für die Dokumentation statt dem Variable Explorer.



Es gibt die Möglichkeit der Funktion Variablen zu übergeben. Dazu muss dies bei der Definition der Funktion festgelegt werden.

In [53]:

```
def meineSumme(x,y): # erhalte x=3, y=5
    print(x, "+", y, "=", x+y)

meineSumme(3,5) # rufe meineSumme auf und übergebe 3 und 5
```

AUSGABE:

3 + 5 = 8

Die Werte, welche übergeben werden, nennt man Argumente. Bei obigem Beispiel wird zwar die Summe ausgegeben, aber im Hauptprogramm kann nicht mehr mit dem Ergebnis weiterrechnen. Wenn die Summe beispielsweise danach in eine Datei geschrieben werden soll, wäre dies so nicht möglich. Funktionen können jedoch mit dem `return` Befehl auch Werte an das Hauptprogramm zurückgeben. Dies geht wie folgt:

In [54]:

```
def meineSumme(x,y): # erhält die Werte aus dem Hauptprogramm
    return(x+y)      # gibt Summe x+y wieder zurück

erg = meineSumme(3,5) # übergibt 3 und 5 und erhält ein Ergebnis zurück
print("Das Ergebnis ist", erg) # erg könnte nun auch in einer Datei
                             gespeichert werden
```

AUSGABE:

Das Ergebnis ist 8

Task 4.1: Erstelle eine Funktion welche den Vorname und Nachname als Argument erhält und den vollständigen Namen (Vorname + Nachname) zurückgibt.

Task 4.2: Erstelle eine Funktion `istGerade(x)` welche zurückgibt ob `x` gerade (d.h. `True`) oder ungerade (`False`) ist.

Task 4.3: Erstelle eine Funktion `istPrimzahl(n)` welche zurückgibt ob `n` eine Primzahl ist, oder nicht. Für die Bestimmung ob eine Zahl eine Primzahl ist, siehe oben.

Mehrere Rückgabewerte

Sollen mehre Werte zurückgegeben werden, dann schreibe diese in eine Liste. Das folgende Beispiel gibt alle Fibonaccizahlen unterhalb eines Wertes zurück.

In [10]:

```
def fib(n):
    fib_liste = []
    a,b = 0,1 # a = 0 und b = 1
    while b <= n:
        fib_liste.append(b)
        a,b = b,a+b
    return fib_liste

print(fib(30)) # gebe alle Fibonacci-Zahlen unter 30 aus
```

AUSGABE:

```
[1, 1, 2, 3, 5, 8, 13, 21]
```

Funktionsbeschreibung

Fügt man direkt hinter der Definition der Funktion einen mehrzeiligen String ("""abc""") ein, so dient dieser der Beschreibung der Funktion. Das kurze Beispiel einer Funktion für die Division veranschaulicht dies.

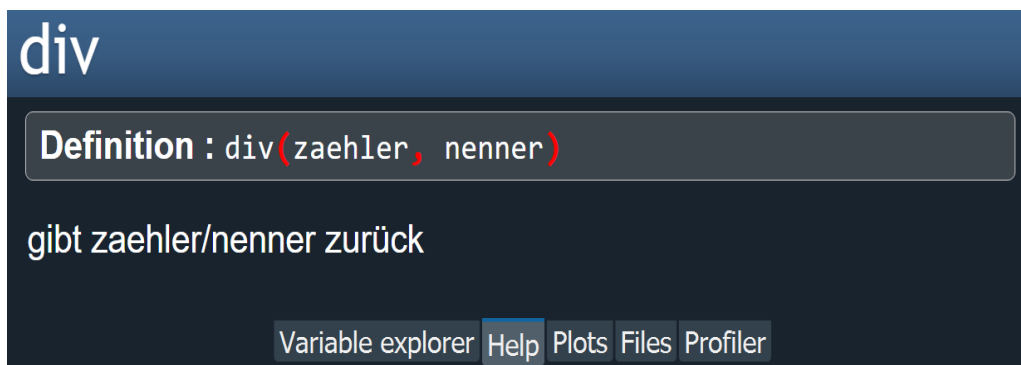
In [11]:

```
def div(zaehler, nenner):  
    """gibt zaehler/nenner zurück"""  
    quotient = zaehler/nenner  
    return quotient  
print(div(1,2))  # <- auf div klicken und dann Ctrl+I
```

AUSGABE:

```
0.5
```

Klicke auf `div` im Funktionsaufruf, so wird im rechten oberen Feld unter "Help" eine Funktionsbeschreibung angezeigt.



Du siehst, dass die Funktion `div` zwei Argumente `zaehler` und `nenner` erhält. Außerdem erhältst Du die Funktionsbeschreibung "gibt zaehler/nenner zurück". Klicke nun wieder auf "Variable Explorer" um zur Ansicht für die Variablen zurückzukehren.

Argumente

Einer Funktion können mehrere Argumente übergeben werden `funktionsname(argument1, argument2, argument3)`. Bei Funktionen mit vielen möglichen Parametern ergeben sich hierbei zwei Probleme:

1. für manche Argumente möchte man evtl. einen Standardwert verwenden. Dies lässt sich festlegen.
2. die Reihenfolge der Argumente kann unklar werden. Hier kann man die Argumente direkt mit Namen ansprechen.

Standardwerte

Wenn Du einen Standardwert bei der Definition der Funktion festlegen möchtest, kannst Du diesen mit `funktionsname(argument=standardwert)` festlegen. Betrachte folgendes Beispiel mit der Division mit einem Standardwert für den Nenner von 1.

In [14]:

```
def div(zaehler, nenner = 1):    # Standard nenner=1
    quotient = zaehler/nenner
    return quotient
print(div(1,2))    # überschreibt den Standardwert. 1/2
print(div(1))      # verwendet für Nenner den Standard = 1. 1/1
#print(div()) -> Fehler da zaehler fehlt
```

AUSGABE:

```
0.5
1.0
```

In dem Beispiel sieht man, dass

1. wenn ein Wert für Nenner übergeben wird, wird der Standardwert überschrieben.
2. wenn kein Wert für Nenner übergeben wird, wird der Standardwert verwendet.

Achtung: Für das Argument `zaehler` wurde kein Standardwert festgelegt. Damit ist ein Aufruf `div()` ohne Argument ist nicht möglich und würde zu einem Fehler führen.

Task 4.4: Erstelle eine Funktion `begruessen(gruss)` welche den String `gruss` ausgibt. Falls die Funktion ohne Argument aufgerufen wird, soll ein allgemeiner Gruß genannt werden.

Positional vs. Keyword Argumente

Bisher wurden Argument stets nach Ihrer Position vergeben. D.h. beim Aufruf von `function(1,"test")` wäre die 1 das erste Argument und "test" das zweite Argument. Du kannst aber der Funktion direkt mitteilen welches Argument zu welchen Variablen gehört. Das folgende Beispiel zeigt zunächst einen Aufruf mit Positions-Argumenten und danach mit Keyword-Argumenten. Bei den Keyword-Argumenten ist die Reihenfolge nicht relevant.

In [16]:

```
# Positional vs. Keyword
def satz(subj, verb, obj):
    print(subj,verb,obj)

# positional arguments
satz("Peter", "lernt", "Python")
# keyword arguments: Variablen der Funktion werden direkt angesprochen.
satz(obj="den Ball", verb="wirft", subj="Peter")
```

AUSGABE:

```
Peter lernt Python
Peter wirft den Ball
```

Task 4.5: Erstelle erneut eine Funktion `bmi(groesse, gewicht)` welche den BMI (Gewicht/Größe²) zurückgibt. Rufe die Funktion mit Keyword-Argumenten auf.

Mehrere Argumente

Falls eine größere evtl. sogar unbekannte Anzahl von Argumenten übergeben wird, kann man dies mit den Spezialoperatoren `*` und `**` tun:

1. Positions-Argumente `*` erhalten eine Liste (Achtung `*` ist hier keine Multiplikation).
2. Keyword-Argumente `**` erhalten ein Wörterbuch (Achtung `**` ist hier keine Potenz).

Schaue Dir die Verwendung in folgenden Beispielen an. Zunächst soll die Summe einer unbekannten Anzahl von Zahlen berechnet werden.

In [37]:

```
# unbekannte Anzahl positionale Parameter
def berechne_Summe(*args):
    mySum = 0
    for x in args:
        mySum = mySum + x
    return (mySum)

print(berechne_Summe(1, 2, 3, 4))
# berechne_Summe(arg1=1, arg2=2) -> ERROR
```

AUSGABE:

10

Die Funktion erhält eine Reihe von Keyword Argumenten.

In [36]:

```
def bsp_keyword(**kwargs):
    print('kwargs: ', kwargs)
    for i in kwargs:
        print("Argument", i, "hat den Wert", kwargs[i])

bsp_keyword(arg1=1, arg2=2, arg3=3)
# bsp_keyword(1,2,3) # ERROR
```

AUSGABE:

```
kwargs: {'arg1': 1, 'arg2': 2, 'arg3': 3}
Argument arg1 hat den Wert 1
Argument arg2 hat den Wert 2
Argument arg3 hat den Wert 3
```

Beide Ideen können auch kombiniert werden:

In [38]:

```
def bsp_beide(*args, **kwargs):
    print("Positional:", args)
```

```
print("Keyword:", kwargs)
```

```
bsp_beide("a","b",c="1",d="2")
```

AUSGABE:

Positional: ('a', 'b')

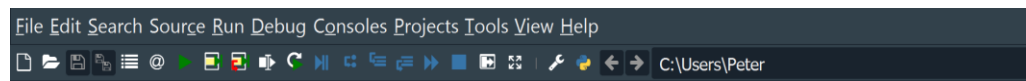
Keyword: {'c': '1', 'd': '2'}

Task 4.6: Erstelle eine Funktion *meinMinimum(..)* welche mehrere Zahlen erhält und die kleinste Zahl davon zurückgibt.

Module

Ein Programm wird unübersichtlich, wenn es allzu lange wird. Außerdem definiert man oft Funktionen, welche man in anderen Programmen ebenfalls verwenden möchte. Deshalb ist es gut, wenn man das Programm in unterschiedliche Dateien speichert. Beispielsweise eine Datei zum Laden der Daten, eine Datei zur Auswertung, und eine Datei zum Darstellen und Speichern der Ergebnisse. Diese unterschiedliche Python Dateien nennt man *Module*. Diese enden mit *.py* und haben den Modulname in einer Variable `__name__`.

Um ein Beispiel zu sehen, speichere folgendes Skript in einer Datei *my_module.py* im Arbeitsverzeichnis (wird rechts oben in Spyder angezeigt; hier *C:\users\peter*).



In [91]:

```
# speichere dies als my_module.py im Arbeitsverzeichnis, im Beispiel  
C:\Users\Peter
```

```
def berechne_summe(*args):  
    return sum(*args)
```

In [93]:

```
import my_module # sucht my_module.py im Arbeitsverzeichnis  
print(my_module.berechne_summe([1,2,3]))
```

AUSGABE:

6

eine Funktion kann auch einen lokalen Namen bekommen

In [94]:

```
summe = my_module.berechne_summe  
print(summe([1,2,3,4]))
```

AUSGABE:

10

Projektaufgabe: Passe das Projekt an, indem die folgenden Funktionen erstellt werden:

- `hinzufuegen(telefonbuch, name, nummer)`: fügt eine Person dem Telefonbuch hinzu
- `entfernen(telefonbuch, name)`: löscht die Person aus dem Telefonbuch
- `suchen(telefonbuch, name)`: gibt Telefonnummer von der entsprechenden Person zurück
- `anzeigen(telefonbuch)`: gibt alle Einträge zurück

In []:

```
def hinzufuegen(telBuch, name, nummer):
    telBuch[name] = nummer

def entfernen(telBuch, name):
    if name in telBuch:
        del telBuch[name]

def suchen(telBuch, name):
    if name in telBuch:
        return (name + " hat die Nummer " + telBuch[name])
    else:
        return name + " nicht gefunden"

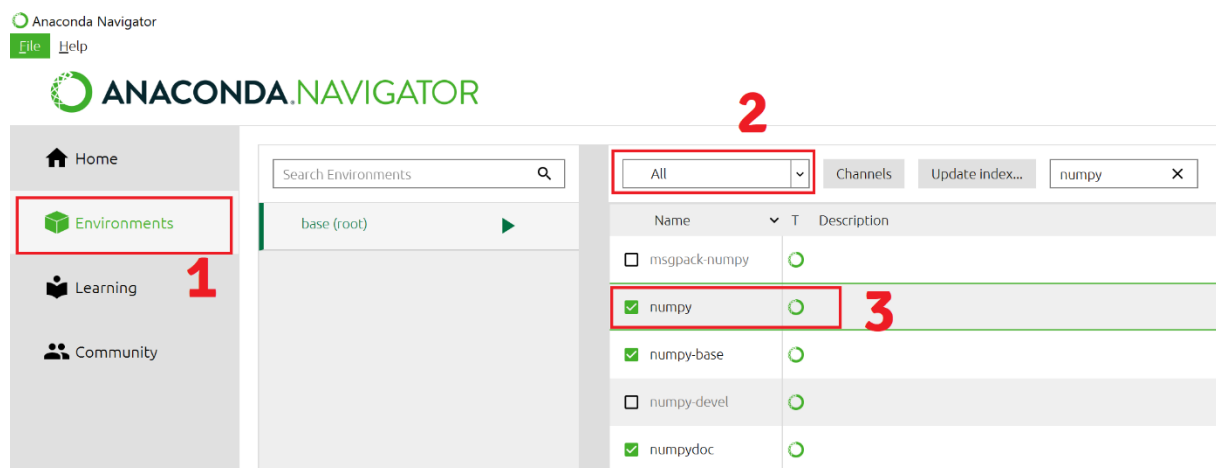
def anzeigen(telBuch):
    for k,v in telBuch.items():
        print(k, "hat Nummer", v)

x = None
while x!="0":
    print("\nBitte auswählen:")
    print("0: Programm beenden")
    print("1: Person hinzufügen")
    print("2: Person entfernen")
    print("3: Person suchen")
    print("4: alle Personen anzeigen")
    x = input()
    if x == "1":
        name = input("Name zum hinzufügen:")
        phone = input("Nummer:")
        hinzufuegen(telefonbuch, name, phone)
    elif x == "2":
        name = input("Name zum entfernen:")
        entfernen(telefonbuch, name)
    elif x == "3":
        name = input("Name zum suchen:")
        print(suchen(telefonbuch, name))
    elif x == "4":
        anzeigen(telefonbuch)
    elif x == "0":
        print("Auf Wiedersehen!")
    else:
        print("ungültige Auswahl.")
```

Bemerkung: Es erscheint etwas umständlich zu sein, dafür erst noch eine Funktion zu definieren. Aber falls das Programm umfangreicher wäre und noch gewisse Checks gemacht werden würden, wäre es sehr sinnvoll diese Schritte in eine Funktion zu verschieben.

Pakete

Pakete sind Projekte von anderen Programmierern welche unterschiedlichen Funktionen enthalten. Diese müssen zunächst installiert werden. Öffne dazu den *Anaconda Navigator* und gehen Environments (1). Gib im Dropdown-Menü *All* ein (2). Nun kannst Du nach Paketen suchen und durch die Auswahl installieren (3).



Installiere die folgenden Pakete:

- numpy
- pandas
- matplotlib

Pakete (Packages) bringen eine Struktur in unterschiedliche Module. Die Struktur ist `PAKET.MODULNAME`. Hier ist eine Struktur des Numerisches Pakets *numpy*.

```
numpy/          # Paketname
  fft.py        # fast fourier transform modul
  linalg.py     # lineare algebra modul
  matlib.py     # matrix calculation modul
  random.py     # random sampling modul
    rand()      # function to generate random ints
    randint()
    shuffle()
```

Das Paket *numpy* hat also ein Modul *random* welches gewisse Funktionen wie *rand()* enthält. Es gibt unterschiedliche Möglichkeiten nun das Modul zu laden und die Funktion aufzurufen:

In [96]:

```
from numpy.random import rand # importiere Funktion
print(rand())

import numpy.random           # importiere modul
```

```
print(numpy.random.rand())      # modul muss immer gezielt angesprochen  
werden
```

```
import numpy.random as r      # importiere modul und mit Abkürzung r  
print(r.rand())
```

AUSGABE (bei Dir anders, da Zufallszahlen):

```
0.7663916003136391  
0.1988336779897335  
0.11858871559667128
```

Task 4.7: Die Funktion `randint(a,b)` im Paket Modul `random` vom Paket `numpy` gibt eine Zufallszahl zwischen `a` und `b` (exklusive). Lassen den Computer würfeln (Zahl zwischen 1 und 6). Nun soll der Benutzer raten welche Zahl gewürfelt wurde. Lasse den Benutzer so lange raten, bis er auf die richtige Zahl getippt hat.

Stunde 5: Hilfreiche Pakete

Schöne Ausgaben (formatted Strings)

Um Texte schön auszugeben, können String formatiert werden. Ein formatierter String wird entweder durch ein vorangestelltes f"..." oder ein nachgelagertes "...".format() dargestellt. In dem String werden die entsprechenden Variablennamen mit geschweiften Klammern eingegeben f"{VARNAME}".

In []:

```
year = 2050
event = 'Wahl'
print(f'Das Ergebnis der {event} im Jahr {year}') # Variante 1
print('Das Ergebnis der {} im Jahr {}'.format(event, year)) # Variante 2
```

Generell gibt es 3 Möglichkeiten bei .format die richtigen Variablen zu wählen: per Reihenfolge, durch Positionsangabe oder durch den Namen.

In [64]:

```
print('Ergebnis {} im Jahr {}'.format(event, year)) # Reihenfolge
print('Ergebnis {1} im Jahr {0}'.format(year, event)) # Position
print('Ergebnis {v1} im Jahr {v2}'.format(v2=year, v1=event)) # Namen
```

AUSGABE:

```
Ergebnis Wahl im Jahr 2050
Ergebnis Wahl im Jahr 2050
Ergebnis Wahl im Jahr 2050
```

Die Ausgaben können entsprechend formiert werden:

- {f} Fließkommazahl
- {%} Prozentzahl
- {<} linksbündig
- {>} rechtsbündig
- {^} zentriert

Dies kann mit einer festen Anzahl reservierter Zeichen verbunden werden. {2.1%} entspricht 2 Stellen vor dem Punkt und eine Stelle nach dem Punkt wird für den Prozentwert reserviert.

In [56]:

```
ja = 4
nein = 6
print('{:2} Ja Stimmen {:.1%}'.format(ja, ja/(ja+nein)))
```

AUSGABE:

```
4 Ja Stimmen 40.0%
```

Der folgende Code stellt die Ausgaben nicht sehr schön dar.

In [57]:

```
liste = ["Alexander", "Jenny", "Mia"]
for l in liste:
```

```
print(l, "ist da")
```

AUSGABE:

```
Alexander ist da
Jenny ist da
Mia ist da
```

besser ist (vielleicht?):

In [61]:

```
liste = ["Alexander", "Jenny", "Mia"]
for l in liste:
    print("{:>9} ist da".format(l)) # reserviere 9 Zeichen und
    rechtsbündige Darstellung.
```

AUSGABE:

```
Alexander ist da
      Jenny ist da
      Mia ist da
```

Projektaufgabe: Bei der bisherigen Anzeige im Projekt werden die Namen nicht schön untereinander dargestellt. Passe die Funktion `anzeigen` entsprechend an.

In [4]:

```
telefonbuch = {"Kai": "0134", "Alexander": "953345"}
def anzeigen(telBuch):
    for k,v in telBuch.items():
        print("{:10} hat die Nummer {:10}".format(k,v))

anzeigen(telefonbuch)
```

AUSGABE:

```
Kai           hat die Nummer 0134
Alexander     hat die Nummer 953345
```

Rechnen

Das `math` Paket liefert viele wichtige Funktionen zu rechnen. Beispiele sind Summen (`fsum` für floating point sum) und Produkte (`prod`).

In [132]:

```
import math
liste = [1,2,3,4]
print("Summe von", liste, "=", math.fsum(liste)) # fsum = 1+2+3+4 = 10
print("Produkt von", liste, "=", math.prod(liste)) # prod=1*2*3*4 = 24
```

AUSGABE:

```
Summe von [1, 2, 3, 4] = 10.0
Produkt von [1, 2, 3, 4] = 24
```

Alle typischen trigonometrischen Funktionen wie Sinus (`sin`), Cosinus (`cos`), Exponentiell (`exp`) oder Logarithmus (`log`) findest Du im `Math` Paket.

In [133]:

```
print(math.sin(2*math.pi)) # sin(2*pi)=0 (hier ungefähr 0)
print(math.cos(2*math.pi)) # cos(2*pi) = 1.0
print(math.exp(1))         # e=2.718
print(math.log(math.exp(2))) # log(exp(2))=2
print(math.log(1024,2))     # log with base 2. 2^10 = 1024
```

AUSGABE:

```
-2.4492935982947064e-16
1.0
2.718281828459045
2.0
10.0
```

Es ist es möglich fehlende Werte, so genannte NaN (not a number) zu definieren. Fehlende Werte sind typisch, wenn der Benutzer etwas in ein Feld nicht eingetragen hat, oder eine Messstation keine Werte geliefert hat. Ähnlich lässt sich auch unendlich (inf für infinity) definieren und abzufragen.

In [75]:

```
x = float("nan") # fehlender Wert = nan
print("Fehlender Wert?", math.isnan(x))
x = float("inf") # unendlich = inf
print("Unendlich?", math.isinf(x))
```

AUSGABE:

```
Fehlender Wert? True
Unendlich? True
```

Task 5.1: Erstelle eine Liste mit fehlenden Werten (NaNs). Filtere nun diese fehlenden Werte aus der Liste und erhalte eine neue bereinigte Liste ohne fehlende Werte.

Möchtest Du mit Zufallszahlen arbeiten, hilft Dir das random Paket weiter.

In [81]:

```
import random
print(random.choice(["Schere", "Stein", "Papier"])) # Zufallswahl
print(random.sample(range(10),5)) # Ziehen ohne Zurücklegen
print(random.randrange(6))        # Zahlen von 0 - 5 (mit Zurücklegen)
print(random.random())             # Zufallszahl zwischen 0 und 1
```

AUSGABE (bei Dir anders, da Zufallszahlen):

```
Schere
[1, 6, 0, 8, 5]
1
0.8290468537066096
```

Task 5.2: Erstelle das Spiele-Schere-Stein Papier, bei denen der Benutzer gegen den den Computer spielt.

Im statistics Paket findest Du die üblichen statistischen Auswertungen wie Mittelwert (mean), Median (median) oder Varianz (variance)

In [82]:

```
import statistics
data = [2,3,4,5,90]
print(statistics.mean(data)) # Durchschnitt
print(statistics.median(data)) # Mittlerer Wert
```

```
print(statistics.variance(data)) # quadratische Abweichung zum Durchschnitt
```

AUSGABE:

```
20.8
4
1497.7
```

Vielleicht hast Du schon festgestellt, dass es bei Python zu merkwürdigen Rundungen kommt. Der Grund liegt darin, dass Zahlen im Binärsystem und nicht im Dezimalsystem gespeichert werden. D.h. beispielsweise wird eine 1.0 als 0.999999 gespeichert.

Hier sind ein paar Beispiele:

In [84]:

```
print(1.0 % 0.1)          # falscher rest 0.09999999999999999
print(sum([0.1]*10))      # falsche Summe 0.9999999999999999
print(0.7*1.05)           # 0.735
print(round(0.7*1.05,2))  # wird falsch abgerundet auf 0.73
```

AUSGABE:

```
0.099999999999999995
0.9999999999999999
0.735
0.73
```

Das Problem ist, dass sich viele falsche Rundungen stark aufaddieren können. Würde Deine Bank falsch runden, würde sich dies auf einen Millionenbetrag aufaddieren. Arbeitet man deshalb mit Dezimalzahlen und möchte explizit richtig runden, hilft das Paket `decimal` weiter.

In [85]:

```
from decimal import Decimal
```

```
print(Decimal("1.0") % Decimal("0.1"))          # rest 0
print(sum([Decimal("0.1")] *10))                # 1.0
print(round(Decimal("0.7")*Decimal("1.05"),2))  # 0.74
```

AUSGABE:

```
0.0
1.0
0.74
```

Zeiten (Time/Date)

Oft möchtest Du ein Datum oder eine Zeit berechnen. Dafür ist das `datetime` Paket. Mit folgenden Abkürzungen kannst Du Zeiträume von Jahren bis Mikrosekunden angeben:

- %a: Abkürzung Wochentag: So, Mo, ..., Sa
- %A: Wochentag: Sonntag, Montag, ..., Samstag
- %d: Tag im Monat (Zahl): 01, 02, ..., 31
- %b: Abkürzung Monat: Jan, Feb, ..., Dez
- %B: Monatsname: Januar, Februar, ..., Dezember
- %m: Monat (Zahl): 01, 02, ..., 12
- %y: Jahr ohne Jahrhundert: 00, 01, ..., 99

- %Y: Jahr: 0001, 0002, ..., 2022, 2023, ..., 9998, 9999
- %H: Stunde: 00, 01, ..., 23
- %M: Minute: 00, 01, ..., 59
- %S: Sekunde 00, 01, ..., 59
- %f: Mikrosekunde: 000000, 000001, ..., 999999

Nun ein Beispiel für ein Datum mit `datetime.date`:

In [103]:

```
# Datum
from datetime import date
jetzt = date.today()    # setzte Datum heute
print(jetzt)
print(jetzt.strftime("%m-%d-%y"))
print(jetzt.strftime("%d %b %Y ist ein %A am %d Tag vom %B"))
```

AUSGABE:
 2025-04-24
 04-24-22
 24 Apr 2025 ist ein Sunday am 24 Tag vom April

Nun definiere ein Datum (hier 1.1.2000) und berechne die Tage seit damals.

In [104]:

```
tag = date(2000,1,1)    # Definition eines Datums
print((jetzt-tag).days) # Tage seit 2000
```

AUSGABE:
 8149

Task 5.3: Berechne an welchem Wochentag Du geboren bist und wie viele Tage Du bereits lebst.

Siehe Dir nun ein Zeitpunkt (Datum + Uhrzeit an) mit `datetime.datetime` an.

In [105]:

```
from datetime import datetime

# jahr, monat, tag, stunde, min, sek, mikrosek
b = datetime(2021,1,31,5,59,1,123)
print(b)
print(b.year)
print(b.month)
print(b.day)
print(b.hour)
print(b.minute)
print(b.second)

now = datetime.now()
print("Die Uhrzeit ist", now.strftime("%H:%M:%S"))
```

```
AUSGABE:
2021-01-31 05:59:01.000123
2021
1
31
5
59
1
Die Uhrzeit ist 17:25:55
```

Zum Messen von Zeiten ist auch das Paket `time` hilfreich. Insbesondere kannst Du messen wie lange dein Programm läuft bzw. Dein Programm auch für eine gewisse Zeit pausieren (`time.sleep(SEK)`).

In [108]:

```
import time
sec1 = time.time()
print("Start")
time.sleep(2.5)
sec2 = time.time()
print("Ende")
print("Dauer:", sec2-sec1, "Sekunden")
```

```
AUSGABE:
Start
Ende
Dauer: 2.500286817550659 Sekunden
```

Task 5.3: Berechne alle Quadrate von 0 bis 1'000'000 mit einer

a. Schleife

b. list comprehension

Messe jeweils die Zeit für die Berechnung.

Bemerkung: Eine List comprehension sollte effizienter und somit schneller als eine Schleife sein.

Stunde 6: Debuggen und Fehler

In dieser Einheit lernst Du Fehlerbehandlungen und das Aufspüren von Fehlern. Es gibt unterschiedliche Fehlerarten. Die einfachsten Fehler sind Syntax Fehler (wie ein vergessener Doppelpunkt oder Klammer). In diesem Fall läuft das Programm nicht durch. Schwieriger sind Fehler, bei welchen das Programm erst durchläuft aber dann irgendwann abbricht (bsp. `TypeError` wenn man einen Text mit einer Zahl addiert). Am schwersten sind Fehler welche selten auftreten. Ein Beispiel ist ein `ZeroDivision` Fehler, welcher auftritt, wenn unerwarteterweise der Nenner gleich Null ist.

Debugging

Betrachte folgendes Beispiel, bei welchem der Code zunächst durchläuft, aber dann plötzlich mit einem Fehler abbricht. Generell gilt, dass $(5-i)/(5-i)=1$ ist. Ausser für $i=5$, dann erhält man einen `ZeroDivisionError` (Fehler aufgrund Teilung durch Null). Deshalb bricht das Programm beim letzten Durchlauf ab.

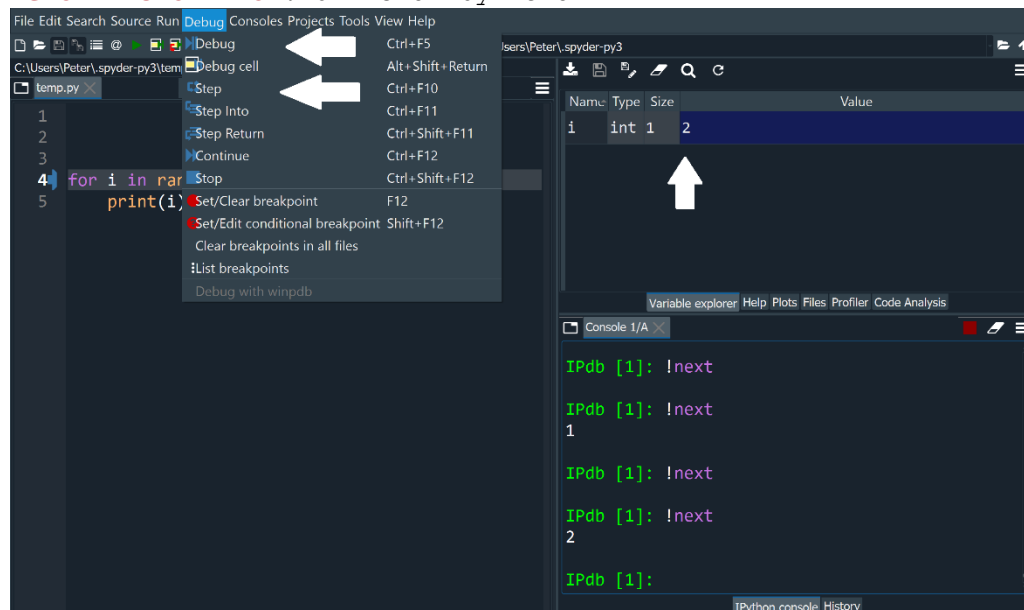
In [20]:

```
for i in range(6):  
    print((5-i)/(5-i))
```

```
1.0  
1.0  
1.0  
1.0  
1.0
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-20-2da2a2e0a284> in <module>  
      1 for i in range(6):  
----> 2     print((5-i)/(5-i))
```

ZeroDivisionError: division by zero



Wenn man hier unsicher ist, woher der Fehler eigentlich kommt, kann man in den Debug-Modus gehen: `Debug -> Debug` (oder `Ctrl+F5`). Dort kann man nun schrittweise das Programm ausführen (`Ctrl+F10`). Man kann nun langsam voranschreiten (mehrmals `Ctrl+F10` drücken) und bemerkt, wie im Variable Explorer sich die Variablen verändern.

Den Debug-Modus wird beim Programmende automatisch verlassen; oder man beendet ihn vorzeitig über Debug -> Stop (oder Ctrl+Shift+F12) Man kann auch einen Breakpoint (F12) bei einer Zeile setzen. Der Breakpoint wird über ein roter Punkt nach der Zeilennummer dargestellt. Durch einen Klick an diese Stelle kann der Breakpoint hinzugefügt oder entfernt werden. Startet man den Debug Modus, wird das Programm bis zum Breakpoint ausgeführt.

Fehlerbehandlung

Schau Dir im Folgenden nochmal häufige Fehler an:

- **SyntaxError:** Meist ist ein Doppelpunkt, Klammer, Hochkomma oder sonstige Zeichen zu wenig (oder zu viel)
- **ZeroDivisionError:** Teilen durch 0
- **NameError:** Variable oder Funktion mit dem Namen existiert nicht
- **TypeError:** Versuch einen Operator unerlaubt auf Variablen von unterschiedlichen Typen anzuwenden. Bsp. Text und Zahlen zu addieren.

```
while True print('Hello world')      # Syntax Error
File "<ipython-input-23-9c8223eb1092>", line 1
  while True print('Hello world')    # Syntax Error
      ^
SyntaxError: invalid syntax
```

In [23]:

```
10 * (1/0)                            # ZeroDivisionError

-----
ZeroDivisionError                     Traceback (most recent call last)
<ipython-input-22-90f9ab40ae0d> in <module>
----> 1 10 * (1/0)                    # ZeroDivisionError

ZeroDivisionError: division by zero
```

In [22]:

```
var()

-----
NameError                             Traceback (most recent call last)
<ipython-input-27-b7d7d398baa1> in <module>
----> 1 var()

NameError: name 'var' is not defined
```

In [27]:

```
'2' + 2                              # TypeError

-----
TypeError                             Traceback (most recent call last)
<ipython-input-25-2e274ed5f019> in <module>
----> 1 '2' + 2                      # TypeError

TypeError: can only concatenate str (not "int") to str
```

In [25]:

Um solche Fehler abzufangen verwendet man einen try:... except FEHLERNAME: Block. Wird kein Fehlername eingegeben try:... except: ..., werden alle Fehler abgefangen.

```
try:
```

In [30]:

```

    x = int(input("Bitte Zahl eingeben: ")) # Versuche die Zahl einzugeben
except ValueError:                        # Fange ValueError ab
    print("Das war leider keine Zahl.")

```

AUSGABE:

```

Bitte Zahl eingeben: abc
Das war leider keine Zahl.

```

Task 6.1: Passe den obenstehenden Programmcode wie folgt an: der Benutzer wird so lange wiederholt aufgefordert eine Zahl einzugeben, bis er tatsächlich eine Zahl eingegeben hat.

Sollen mehrere Fehler abgefangen werden, kannst Du mit mehreren `except` Blöcken arbeiten.

In [37]:

```

try:
    zaehler = float(input("Zähler eingeben:"))
    nenner = float(input("Nenner eingeben:"))
    print("Ergebnis=", zaehler/nenner)
except ValueError:          # fange ZeroDiv Fehler ab
    print("Keine Zahl eingegeben")
except ZeroDivisionError:   # fange ZeroDiv Fehler ab
    print("Kann nicht durch 0 teilen")
except:                    # fange sonstige Fehler ab
    print("Sonstige unerwartete Fehler")

```

AUSGABE:

```

Zähler eingeben:abc
Keine Zahl eingegeben

```

Es werden nicht nur Fehler abgefangen, welche direkt im `try` Feld passieren, sondern auch Fehler, welche in dort aufgerufenen Funktionen passieren. Das folgende Beispiel veranschaulicht dies:

In [42]:

```

def fehlerhafte_funktion():
    return 1+"abc"

try:
    fehlerhafte_funktion()
except TypeError as err: # Message err
    print('Fehler aufgetreten:', err)

Fehler aufgetreten: unsupported operand type(s) for +: 'int' and 'str'

```

Task 6.2: Fange alle Fehler ab (keine Zahl eingegeben, durch 0 geteilt), die bei der Berechnung des body-mass-index passieren können ($BMI = \text{gewicht(in kg)} / \text{groesse}^2$ (in m)).

Bemerkung: Eine negative Zahl als Eingabe führt zu keinem Python-Fehler. Damit muss dieses nicht mit einem `try` Statement abgefangen werden. Man kann dies aber mit einem `if` statement abfangen.

Namensräume

Ein häufiger Fehler ist, dass Variablen andere Werte als erwartet haben. Python hat so genannte Namensräume (Namespaces) in welchen auf unterschiedliche Variablen zugegriffen werden. Insbesondere gelten Variablen, welche innerhalb einer Funktion definiert werden, nur innerhalb dieser Funktion. Siehe Dir dazu folgendes Beispiel an:

In [76]:

```
def funktion1():
    name = "Conny"
    print("innerhalb der Funktion:",name)

def funktion2():
    global name
    name = "Jonah"
    print("innerhalb der Funktion:",name)

name = "Peter"
funktion1()
print("Ausserhalb von der Funktion:",name)
funktion2()
print("Ausserhalb von der Funktion:",name)
```

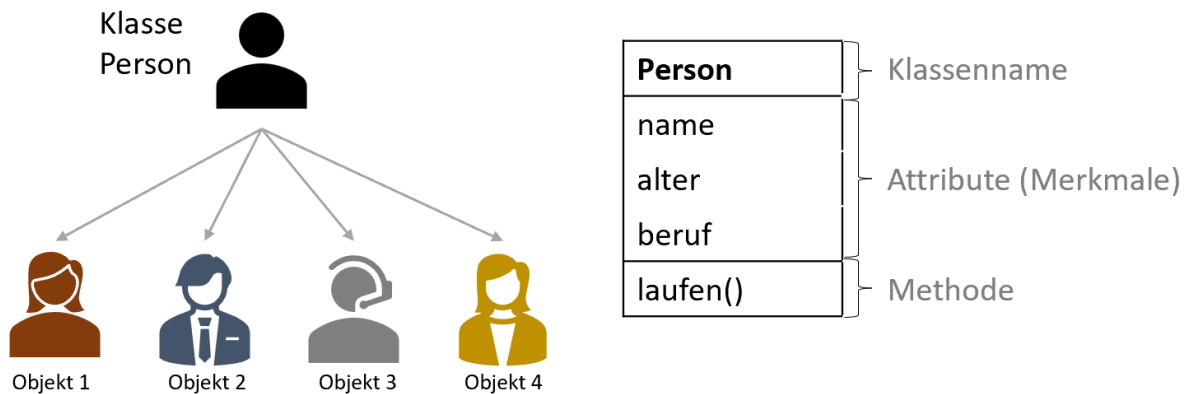
AUSGABE:

```
innerhalb der Funktion: Conny
Ausserhalb von der Funktion: Peter
innerhalb der Funktion: Jonah
Ausserhalb von der Funktion: Jonah
```

Du siehst, dass die Variable `name` im Hauptprogramm als "Peter" definiert wird. Innerhalb der Funktion `funktion1` wird ebenfalls eine Variable `name` mit dem Wert "Conny" definiert. Innerhalb der Funktion hat nun `name` den Wert "Conny". Geht man jedoch wieder ins Hauptprogramm stellt man fest, dass dort die ursprüngliche Definition "Peter" gilt. Anders verhält es sich wenn man bewusst innerhalb einer Funktion eine Variable als `global` definiert (hier "Jonah"). Diese behält dann auch über die Funktion hinaus ihre Gültigkeit.

Stunde 7: Klassen

Will man Beispielsweise mehrere Personen in einer Kartei anlegen, so könnte man in einer Liste unterschiedliche Personen anlegen. Man müsste davon abgetrennt unterschiedliche Funktionen definieren. Viel besser ist es eine Schablone "Person" zu haben und daraus unterschiedliche Personen zu erzeugen. Im Programmierslang sagt man, dass aus der Klasse "Person" mehrere Objekte bzw. Instanzen erzeugt werden.



Eine Person hat gewisse Merkmale (Alter, Größe, Gewicht) und Fähigkeiten (laufen, schwimmen, lernen). In einer Klasse heißen die Merkmale Attribute (Variablen in der Klasse) und Fähigkeiten Methoden (Funktionen in der Klasse).

Schaue nun Deine erste einfache Klasse "Person" an.

In [28]:

```
class Person:
    """Meine erste Klasse""" # Beschreibung
    def wer(self):
        return "Ich bin's"
```

Du siehst eine Klasse wird mit dem Schlüsselwort `class` `KLASSENNAME` definiert. Die Beschreibung ist nicht unbedingt notwendig, hilft aber dem Leser und funktioniert wie bei Funktionen (siehe Kapitel zu Funktionen). Dazu gibt es noch eine Method `wer()`. Diese enthält das Keyword `self`, was später nochmals genauer betrachtet wird. Eine Methode ist wie eine Funktion. Während aber eine Funktion von sich heraus aufgerufen werden kann, ist eine Methode immer an eine Klasse bzw. Objekt geknüpft.

Nun hast Du zwar die Schablone "Person" erstellt, aber noch keine Person erzeugt. Dies funktioniert wie folgt.

In [29]:

```
p = Person()
c = Person()
print(p.wer())
print(c.wer())
```

AUSGABE:

```
Ich bin's
Ich bin's
```

Nun hast Du zwar 2 Personen x und y erzeugt. Diese unterscheiden sich jedoch noch nicht. Bei der Erstellung eines Objekt (Initialisierung) können Werte übergeben werden. Wird eine Objekt erstellt, wird im Hintergrund die Methode `__init__` (für Initialisierung) aufgerufen. Beachte, dass bei `__init__` ein doppeltes Unterstrichen (`_`) verwendet wird.

In [31]:

```
class Person:
    def __init__(self, name):
        self.n = name
    def wer(self):
        return "Ich bin " + self.n

p = Person("Peter")      # Übergebe einen Namen
print(p.wer())           # Ich bin Peter

c = Person("Conny")
print(c.wer())
```

AUSGABE:

```
Ich bin Peter
Ich bin Conny
```

Wir sehen, dass sich nun beide Objekte (zumindest im Namen) unterscheiden. Nun nochmals zu dem seltsamen keyword `self`. Beim Aufruf einer Methode wird erstmal stets das Objekt selbst übergeben. D.h. Du kannst `p.wer()` als `wer(p)` lesen. Bei der Initialisierung wird somit das Attribut `p.n` mit dem Wert "Peter" deklariert. Analog bei der zweiten Person "Conny". Diese Attribute lassen sich auch direkt aufrufen.

In [34]:

```
print(p.n, "und", c.n)
```

AUSGABE:

```
Peter und Conny
```

Task 7.1: Erstelle eine Klasse `Komplex` welche eine Komplexe Zahl mit einem Realteil und einer Imaginärzahl definiert (https://de.wikipedia.org/wiki/Komplexe_Zahl). Eine komplexe Zahl soll bsp. mit `x = Komplex(3.0, -4.5)` (mit Realteil 3.0 und Imaginärteil -4.5) erstellt werden.

Betrachte nun so genannte Klassenattribute, welche denselben Wert für alle Objekte besitzen. Im unteren Beispiel ist es das Attribut `semester`. Daneben gibt es auch Objekteigene Attribute welche mit `self.ATTRIBUTNAME` deklariert werden. Hier ist es der Name `self.name`.

In [65]:

```
class StudentIn:
    semester = 1          # Klassenattribut gilt für alle Instanzen
    def __init__(self, name):
        self.name = name  # Instanz-variable speziell für alle Instanzen

p = StudentIn('Peter')
c = StudentIn('Conny')
print("Semester:",
```

```

        p.semester, "und", c.semester)    # gleich für alle Studierenden

print("Name:", p.name, "und", c.name)    # speziell für d und e

```

AUSGABE:

```

Semester: 1 und 1
Name: Peter und Conny

```

Angenommen Du möchtest eine Fachliste für alle Studenten erstellen.

In [5]:

```

class StudentIn:
    faecher = []          # Klassen variable falsch!
    def __init__(self, name):
        self.name = name
    def add_fach(self, fach):
        self.faeher.append(fach)

p = StudentIn('Peter')
c = StudentIn('Conny')
p.add_fach('Mathematik')
p.add_fach('Programmieren')
print(p.name, "Fächer:", p.faeher)
print(c.name, "Fächer:", c.faeher)    # Falsch: Fächer auch an Conny

```

AUSGABE:

```

Peter Fächer: ['Mathematik', 'Programmieren']
Conny Fächer: ['Mathematik', 'Programmieren']

```

Task 7.2: Wie muss oben genanntes Programm angepasst werden, damit nur "Peter" und nicht "Conny" die Fächer hinzugefügt werden?

Beispiel Fahrzeug

Schau Dir nun das ausführlichere Beispiel an. Dort wird eine Klasse "Fahrzeug" erstellt, welche bei Initialisierung den hersteller und das Jahr übergibt. Außerdem gibt es noch ein zusätzliches Objektspezifisches Attribut `self.km` welche am Anfang stets auf 0 gesetzt wird. Dieses Attribut kann aber über die Methode `erhohe_kmstand` verändert werden.

In [47]:

```

class Fahrzeug():
    """Klasse um Fahrzeug darzustellen"""

    def __init__(self, hersteller, jahr):
        self.hersteller = hersteller
        self.jahr = jahr
        self.km = 0          # Attribut km auf 0 gesetzt

    def beschreibung(self):
        return (self.hersteller +
                " aus dem Jahr " + str(self.jahr))

    def kmstand(self):
        print("Der Kilometerstand beträgt", self.km)

```

```

def erhoehe_kmstand(self, km):
    if km > 0:          # gefahrene km führen immer zu höherem km Stand
        self.km += km # d.h. self.km = self.km + km
    else:               # der km-Stand darf nicht zurückgesetzt werden.
        print("Der km-Stand kann nicht zurückgesetzt werden.")

def hupen(self):
    print("Tuuuut")

```

Nun wird ein Mercedes-Objekt erstellt und der Kilometerstand zunächst auf 30 erhöht und danach (erfolglos) versucht wieder um 10 zurückzusetzen.

In [49]:

```

m = Fahrzeug("Mercedes", 2021)
print(m.beschreibung())
m.kmstand()          # 0km
m.erhoehe_kmstand(30) # von 0km auf 30km
m.kmstand()          # 30km
m.erhoehe_kmstand(-10) # Meldung, km nicht zurücksetzbar
m.kmstand()          # immernoch 30km

```

AUSGABE:

```

Mercedes aus dem Jahr 2021
Der Kilometerstand beträgt 0
Der Kilometerstand beträgt 30
Der km-Stand kann nicht zurückgesetzt werden.
Der Kilometerstand beträgt 30

```

In obigem Beispiel wird klar, dass es sinnvoll ist die Attribute nicht direkt zu verändern. Um Attribute anzupassen, sollten immer Methoden verändert werden. Damit kann man sicherstellen, dass die Attribute vom richtigen Typ sind und nicht falsch verändert (im Beispiel reduziert) werden.

Um zu verhindern, dass ein Attribut direkt verändert wird, kannst Du den Namen des Attributs mit `__` beginnen lassen. In diesem Fall kann das Attribut nur über eigene Methode geändert werden. Andere Programmiersprachen nennen solche Attribute *privat*. Für das Verständnis siehe Dir dazu folgendes Beispiel an:

In [62]:

```

class Haus:
    def __init__(self):
        self.__maxpreis = 0 # privates Attribut mit __

    def verkauf(self):
        print("Verkauf für ", self.__maxpreis)

    # Methode zum Verändern des Preises
    def setzeMaximalPreis(self, p):
        self.__maxpreis = p

h = Haus()
h.setzeMaximalPreis(100) # ändern über Methode
h.verkauf()
print("Versuch Preis direkt zu ändern")
h.__maxpreis = 200 # Versuch Attribut direkt zu verändern
h.verkauf()

```

AUSGABE:

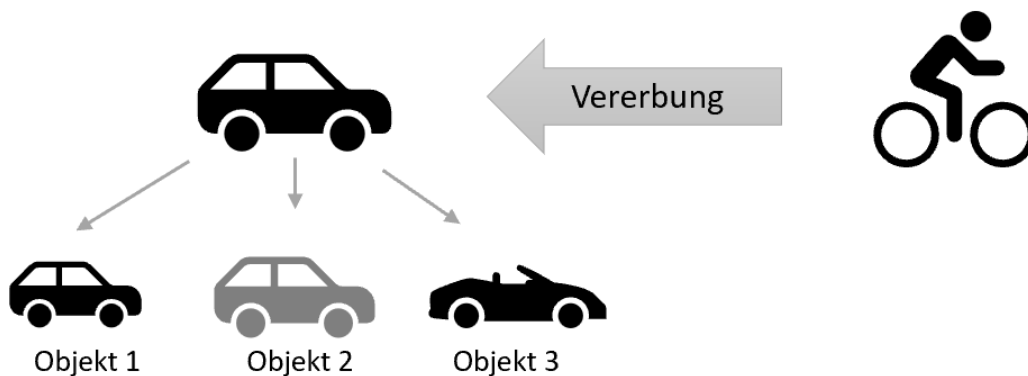
```
Verkauf für 100  
Versuch Preis direkt zu ändern  
Verkauf für 100
```

Du siehst, dass der Preis zwar mit der Methode, aber nicht direkt verändert werden konnte.

Task 7.3: Erstelle eine Klasse Konto mit einem privaten Attribut `kontostand` und einer Methode `setzeKontostand(wert)`. Der Kontostand soll nicht negativ werden können.

Vererbung

Möchte man eine Klasse Fahrrad erstellen, so kann man von der Struktur der Klasse Fahrzeug profitieren. Mit dem Syntax `class NEU_KLASSE(ELTERNKLASSE)` kann alle Attribute und Methoden erben.



Um bei der Initialisierung Attribute an die Elternklasse weiterzuleiten verwendet man `super().__init__(ATTRIBUTE)`.

In [19]:

```
class Fahrrad(Fahrzeug):  
    """Spezieller Fahrzeugtyp: Fahrrad"""  
    def __init__(self, hersteller, jahr):  
        super().__init__(hersteller, jahr)  
        # oder Fahrzeug.__init__(self, hersteller, jahr)  
  
    def hupen(self):    # überschreibe Methode  
        print("Fahrrad hat keine Hupe")  
  
    def klingel(self):  # neue Methode  
        print("Klingeling")
```

In der Klasse Fahrrad wird eine neue Methode `klingel()` erzeugt. Außerdem wird die Methode `hupen()` der Fahrzeugklasse überschrieben und durch eine neue ersetzt.

In [20]:

```
f = Fahrrad("BMX", 2019)    # Erzeuge eine Instanz Fahrrad  
print(f.beschreibung())    # Fahrrad erbt alle Methoden von Fahrzeug  
f.hupen()                  # die neue Methode hupen der Fahrradklasse  
f.klingel()                # Erweiterung der Fahrradklasse
```

AUSGABE:

BMX aus dem Jahr 2019
Fahrrad hat keine Hupe
Klingeling

Task 7.4: Erstelle eine Klasse `Person` mit den Attributen *name* und *alter*, sowie einer Methode *vorstellen()*. Erstelle eine abgeleitete Klasse `Arbeitnehmer` mit dem zusätzlichen Attribut *jobtitel*. Überschreibe die Methode *vorstellen()*.

Meist verfügt die Kinderklasse über mehr Funktionalität verfügt wie die Elternklasse. Allerdings kann man auch zu Vereinfachung führen. Ein Beispiel ist die Klasse `Quadrat` welche von der Klasse `Rechteck` erbt.

In [54]:

```
class Rechteck:
    def __init__(self, laenge, breite):
        self.laenge = laenge
        self.breite = breite
    def inhalt(self): # Flächeninhalt = länge*breite
        return self.laenge * self.breite

class Quadrat(Recteck):
    def __init__(self, laenge): # für Quadrat länge=breite
        super().__init__(laenge, laenge)
```

Ein Quadrat mit Länge 4 hat den Flächeninhalt (4·4=16)

In [55]:

```
q = Quadrat(4)
print(q.inhalt())
```

AUSGABE:

16

Mehrfachvererbung

Klassen können sequentiell vererben, beispielsweise Fahrzeug → Auto → Caprio. Aber eine Klasse kann per `class NEU_KLASSE(ELTERNKLASSE1, ELTERNKLASSE2)` auch von mehreren Klassen erben. Dies zeigt folgendes Beispiel.

In [60]:

```
class Mama():
    def __init__(self, name):
        self.name = name
    def vorstellen(self):
        print("Ich bin die Mama")
    def backen(self):
        print("Ich kann backen")

mum = Mama("Ute")
mum.vorstellen()

class Papa():
    def __init__(self, name):
        self.name = name
    def vorstellen(self):
```

```

        print("I bin der Papa")
    def kochen(self):
        print("Ich kann kochen")

dad = Papa("Roland")
dad.vorstellen()

class Ich(Mama, Papa):    # Mehrfachvererbung
    def __init__(self, name):
        Papa.__init__(self, name)

i = Ich("Peter")
i.backen()                # Methode der Klasse Mama
i.kochen()                # Methode der Klasse Papa
i.vorstellen()            # vorstellen() Methode der ersten Elternklasse

```

AUSGABE:

```

Ich bin die Mama
I bin der Papa
I bin der Papa
Ich kann backen
Ich kann kochen
Ich bin die Mama

```

Vertausche nun die Reihenfolge der Elternklasse zu Papa, Mama. Welches Ergebnis bekommst Du?

Polymorphismus

Unter dem Wort *Polymorphismus* verbirgt sich, dass mehrere unterschiedliche Klassen dieselbe Methode haben können. Wenn dies der Fall ist, kann man eine gemeinsame Schnittstelle anbieten, welche die Methoden unabhängig von der Klasse ausführen kann. Siehe dazu folgendes Beispiel:

In [65]:

```

class Adler:
    def fliegen(self):
        print("Adler kann fliegen")

class Katze:
    def fliegen(self):
        print("Katze kann nicht fliegen")

# gemeinsame Schnittstelle:
def starteFlug(vogel):
    vogel.fliegen()    # führt Methode von beiden Klassen aus

a = Adler()
k = Katze()

starteFlug(a)
starteFlug(k)

```

AUSGABE:

```
Adler kann fliegen
Katze kann nicht fliegen
```

Natürlich funktioniert dies nur wenn die notwendigen Methoden (hier `fliegen()`) bei allen Klassen implementiert ist.

Numerische Operationen

Will man zwei Objekte einer Klasse addieren, so ist nicht klar was passieren soll. Dasselbe gilt, wenn man ein Objekt printen möchte. `print(s1)` führt zu einer kryptischen Meldung, wo das Objekt aktuell gespeichert ist. Möchte man zwei Objekte addieren, führt es zum Programmabbruch. Python kann nicht erraten was ein sinnvolles Ergebnis der Addition sein soll.

In [73]:

```
class Strasse:
    def __init__(self, name, laenge):
        self.name = name
        self.laenge = laenge

s1 = Strasse("B1", 5)
s2 = Strasse("A12", 3)
print(s1)
s3 = s1+s2      # Type Error!

<__main__.Strasse object at 0x000001D61D6549D0>
-----
TypeError                                Traceback (most recent call last)
<ipython-input-73-75e21f9447fb> in <module>
      7 s2 = Strasse("A12", 3)
      8 print(s1)
----> 9 s3 = s1+s2      # Type Error!

TypeError: unsupported operand type(s) for +: 'Strasse' and 'Strasse'
```

Du kannst jedoch festlegen was bei der Addition passieren soll. Andere Operationen sind

- `+`: `object.__add__(self,other)`
- `-`: `object.__sub__(self,other)`
- `*`: `object.__mul__(self,other)`
- `/`: `object.__truediv__(self,other)`
- `//`: `object.__floordiv__(self,other)`
- `%`: `object.__mod__(self,other)`
- `**`: `object.__pow__(self, other)`
- `str`: `object.__str__(self) # für print(object)`

Im folgenden siehst Du, dass bei der Addition `__add__(self, other)` von zwei Straßen eine neue, längere Straße entstehen soll. Wenn ein Objekt angezeigt wird, wird `print(object)` dies zunächst mit `str` in einen String umwandeln. Die Methode `__str__` definiert bei der Umwandlung in einen String ausgegeben werden soll.

In [74]:

```

class Strasse:
    def __init__(self, name, laenge):
        self.name = name
        self.laenge = laenge

    def __add__(self, other): # self+other
        return Strasse("Neue Strasse",
                        self.laenge + other.laenge)

    def __str__(self): # für print
        return self.name + " ist " + str(self.laenge) + "km lang"

s1 = Strasse("B1", 5)
s2 = Strasse("A12", 3)

s3 = s1+s2 # s1.__add__(self, s2)
print(s3)  # __str__(s3)

```

AUSGABE:

Neue Strasse ist 8km lang

Task 7.5: Erstellen eine Klasse Vektor mit einem zwei-dimensionalen Vektor mit einem x und einem y Wert. Zwei Objekte von der Klasse Vektor sollen addiert und subtrahiert werden können.

a = Vektor(3,2)

b = Vektor(4,9)

c = a+b

Das Ergebnis soll ein neuer Vektor sein mit $x_c = x_a + x_b$ und $y_c = y_a + y_b$

Projekt: Bilde das Telefonbuch-Projekt nun als Klasse ab.

In []:

```

class Telefonbuch:
    def __init__(self):
        self.telBuch = dict()
    def hinzufuegen(self, name, nummer):
        self.telBuch[name] = nummer
    def entfernen(self, name):
        if name in self.telBuch:
            del self.telBuch[name]
    def suchen(self, name):
        if name in self.telBuch:
            return name + " hat die Nummer " + self.telBuch[name]
        else:
            return name + " nicht gefunden"
    def anzeigen(self):
        for k,v in self.telBuch.items():
            print("{:<10} hat Nummer {:<14}".format(k,v))

t = Telefonbuch()
x = None
while x!="0":
    print("\nBitte auswählen:")
    print("0: Programm beenden")
    print("1: Person hinzufügen")

```

```

print("2: Person entfernen")
print("3: Person suchen")
print("4: alle Personen anzeigen")
x = input()
if x == "1":
    name = input("Name zum hinzufügen:")
    phone = input("Nummer:")
    t.hinzufuegen(name, phone)
elif x == "2":
    name = input("Name zum entfernen:")
    t.entfernen(name)
elif x == "3":
    name = input("Name zum suchen:")
    print(t.suchen(name))
elif x == "4":
    t.anzeigen()
elif x == "0":
    print("Auf Wiedersehen!")
else:
    print("ungültige Auswahl.")

```

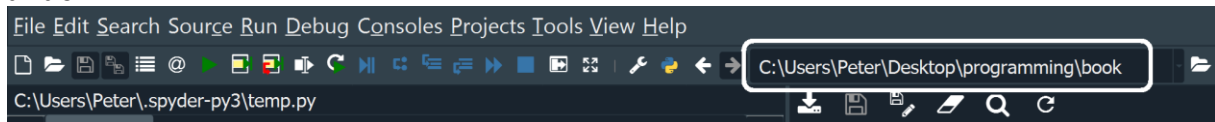
Du magst Dich fragen was hiervon der Vorteil ist. Für Dich selbst war es im aktuellen Beispiel keine Erleichterung. Wenn aber Du oder Deine Freunde nächstes Mal ein Telefonbuch erstellen wollt, könnt ihr einfach die Klasse `Telefonbuch` verwenden. Damit habt ihr bereits alle dazugehörigen Methoden und müsst nicht noch Funktionen suchen welche ihr evtl. braucht und dann selbst das Telefonbuch als Wörterbuch anlegen.

Stunde 8: Daten

Hat man Daten verarbeitet, möchte man diese oft dauerhaft speichern. Dazu ist es notwendig Dateien lesen und schreiben zu können.

Dateien lesen und schreiben

Wenn Du eine Datei schreibst, so wird diese im aktuellen Arbeitspfad (working directory) abgelegt. Diesen kannst Du in Spyder rechts oben ändern.



Mit dem Paket `os.getcwd()` (get current working directory) kannst Du Dir den Pfad auch ausgeben lassen.

In []:

```
import os
print(os.getcwd())
```

Öffne diesen Pfad im Explorer um zu überprüfen ob die Dateien tatsächlich dort abgelegt werden.

Eine Datei öffnest Du mit dem Befehl `open(DATEINAME, ART)`. Den Dateinamen kannst Du frei wählen. Gut ist beispielsweise *info.txt*, da die Datei dann automatisch in einem Texteditor geöffnet wird und Du Dir den Inhalt direkt anschauen kannst. `ART` ist ein String welcher folgende Werte annehmen kann:

- "w": Datei schreiben (write). Achtung: dies überschreibt den bisherigen Inhalt.
- "a": Datei hinzufügen (append). Der Inhalt wird dem bisherigen Inhalt angehängt.
- "r": Datei lesen (read). Dies ist die Standardeinstellung, wenn Du nichts angibst.
- "r+": Datei lesen und schreiben.

Es können beispielsweise direkt Strings geschrieben und gelesen werden. Wie bisher wird eine neue Zeile mit `"\n"` gestartet. `with open()` as `f` öffnet einen Eingabe-Ausgabeoperator `f`. Dieser kann lesen `f.read` und schreiben `f.write`. Dazu folgende Beispiele:

Datei speichern

In [121]:

```
# Datei schreiben
with open("meine_datei.txt", "w") as f:
    f.write('Dies ist die erste Zeile\nHier ist die zweite Zeile.')

# Noch einen zusätzlichen Satz anhängen
with open("meine_datei.txt", "a") as f:
    f.write('\nDritte Zeile.')
```

Gehe nun in deinen Explorer in Deinen Arbeitspfad. Dort müsstest Du eine neue Datei *meine_datei.txt* finden. Wenn Du diese in einem Texteditor öffnest, wirst Du den o.g. Text finden.

Bemerkung: Während Du die Datei im Texteditor geöffnet hast, kannst Du den Inhalt nicht (über)schreiben. Schließe also den Texteditor wieder.

Datei lesen

Nachdem Du die Datei geschrieben hast, kannst Du diese auch wie folgt in den String *daten* auslesen.

In [123]:

```
with open('meine_datei.txt') as f:
    daten = f.read()

print(daten)
```

AUSGABE:

*Dies ist die erste Zeile
Hier ist die zweite Zeile.
Dritte Zeile.*

Oft ist es hilfreich zu schauen, ob eine Datei vorhanden ist. Dies kannst Du mit `os.path.isfile(DATEINAME)` tun:

In [124]:

```
import os
if os.path.isfile("meine_datei.txt"):
    print("Datei ist vorhanden und kann gelesen werden")
else:
    print("Datei ist nicht vorhanden")
```

AUSGABE:

Datei ist vorhanden und kann gelesen werden

Json

Nun hast Du gesehen, wie Du Texte speichern und laden kannst. Möchtest Du aber eine Datenstruktur wie eine Liste oder ein Wörterbuch speichern, hilft es Dir nicht dieses als String zu lesen. Das einfachste ist hierbei das Paket `json` zu verwenden, um solche Datenstruktur zu speichern (`json.dump(.)`) und zu laden (`json.load(.)`). Schau Dir dazu folgendes Beispiel an:

```
import json
liste = [1,2,"text"]

# liste speichern
with open("info.txt", "w") as f:
    json.dump(liste, f)

with open("info.txt", "r") as f:
    geladene_liste = json.load(f)

print(geladene_liste, "ist vom Typ", type(geladene_liste))
```

AUSGABE:

```
[1, 2, 'text'] ist vom Typ <class 'list'>
```

In deinem Arbeitspfad wirst Du nun eine Datei *info.txt* finden, welche genau die Liste enthält. Der Vorteil von json ist, dass nachdem Du die Daten geladen hast, diese auch eine Liste und nicht nur ein String sind.

Projekt: Erweitere das Projekt wie folgt:

- wenn eine neue Instanz der Klasse Telefonbuch erstellt wird, soll versucht werden das Telefonbuch aus der Datei *tel.txt* zu laden. Wenn diese nicht vorhanden ist, soll ein leeres Wörterbuch als Telefonbuch verwendet werden. Hier musst Du `__init__` entsprechend anpassen.
- Füge der Klasse Telefonbuch neue Methode *speichern()* hinzu. Das Telefonbuch soll darin in der Datei *tel.txt* gespeichert werden.

In [130]:

```
import json
import os

class Telefonbuch:
    def __init__(self):
        if os.path.isfile("tel.txt"): # wenn Datei existiert
            with open("tel.txt", "r") as f:
                self.telBuch = json.load(f)
        else:
            self.telBuch = dict()
    def hinzufuegen(self, name, nummer):
        self.telBuch[name] = nummer
    def entfernen(self, name):
        if name in self.telBuch:
            del self.telBuch[name]
    def suchen(self, name):
        if name in self.telBuch:
            return name + " hat die Nummer " + self.telBuch[name]
        else:
            return name + " nicht gefunden"
    def anzeigen(self):
        for k,v in self.telBuch.items():
            print("{:<10} hat Nummer {:<14}".format(k,v))
    def speichern(self):
        with open("tel.txt", "w") as f:
            json.dump(self.telBuch,f)
        print("Daten in",os.getcwd(), "geschrieben.")

t = Telefonbuch()
x = None
while x!="0":
    print("\nBitte auswählen:")
    print("0: Programm beenden")
    print("1: Person hinzufügen")
    print("2: Person entfernen")
    print("3: Person suchen")
    print("4: alle Personen anzeigen")
    x = input()
```

```

if x == "1":
    name = input("Name zum hinzufügen:")
    phone = input("Nummer:")
    t.hinzufuegen(name, phone)
elif x == "2":
    name = input("Name zum entfernen:")
    t.entfernen(name)
elif x == "3":
    name = input("Name zum suchen:")
    print(t.suchen(name))
elif x == "4":
    t.anzeigen()
elif x == "0":
    print("Auf Wiedersehen!")
else:
    print("ungültige Auswahl.")

t.speichern() # speicher Wörterbuch in tel.txt

```

AUSGABE:

```

Bitte auswählen:
0: Programm beenden
1: Person hinzufügen
2: Person entfernen
3: Person suchen
4: alle Personen anzeigen
4
Peter      hat Nummer 3224

```

```

Bitte auswählen:
0: Programm beenden
1: Person hinzufügen
2: Person entfernen
3: Person suchen
4: alle Personen anzeigen
2
Name zum entfernen:Peter

```

```

Bitte auswählen:
0: Programm beenden
1: Person hinzufügen
2: Person entfernen
3: Person suchen
4: alle Personen anzeigen
4

```

```

Bitte auswählen:
0: Programm beenden
1: Person hinzufügen
2: Person entfernen
3: Person suchen
4: alle Personen anzeigen
0
Auf Wiedersehen!
Daten in C:\Users\Peter\Desktop\book geschrieben.

```

Wenn Du Spyder bisher geschlossen hattest, war Dein Telefonbuch weg. Nun ist dieses aber in der Datei *tel.txt* gespeichert. Wenn Du das Programm nun einen Tag später neu startest, hast Du weiterhin alle Deine Kontakte.

Daten laden (Excel, CSV)

Ein häufiges Format für Daten sind Excel-Dateien oder CSV-Dateien (Comma-Separated-Values). Diese speicherst und lädst Du am Besten mit pandas. Pandas arbeitet mit so genannten DataFrames welche mit `pandas.DataFrame(verschachtelte Liste)` erzeugt werden. Diese können bequem in einem Excel oder einer CSV Datei gespeichert werden.

In [69]:

```
import pandas as pd
df = pd.DataFrame([['Michael', 'Jackson', 1958, -400],
                  ['Britney', 'Spears', 1981, 70],
                  ['Adele', '', 1988, 220]],
                  columns=['Vorname', 'Nachname', 'Jahrgang', 'Vermögen (m)'])
df.to_excel("daten.xlsx", index=False) # als Excel
# oder alternativ
df.to_csv("daten.csv", index=False)   # als CSV
```

Nun ist die Excel bzw. Datei in Deinem Arbeitsverzeichnis gespeichert. Die Dateien können nun bequem per `pandas.read_excel` bzw. `pandas.read_csv` gelesen werden.

In [70]:

```
daten = pd.read_excel('daten.xlsx')
print(daten)
print("\noder als CSV Datei")
daten2 = pd.read_csv('daten.csv')
print(daten2)
```

AUSGABE:

	Vorname	Nachname	Jahrgang	Vermögen (m)
0	Michael	Jackson	1958	-400
1	Britney	Spears	1981	70
2	Adele	NaN	1988	220

oder als CSV Datei

	Vorname	Nachname	Jahrgang	Vermögen (m)
0	Michael	Jackson	1958	-400
1	Britney	Spears	1981	70
2	Adele	NaN	1988	220

Daten bearbeiten

pandas ist gut im Daten bearbeiten. Zwei der wichtigsten Operationen ist die Erstellung von neuen Spalten und die Selektion von Daten. Erstelle eine neue Spalte *weiblich*, welche 0 für *Michael Jackson* und 1 für den Rest ist (hier Britney und Adele).

In [71]:

```
# erstelle eine neue Spalte
df['weiblich'] = [0 if x == 'Michael' else 1 for x in df['Vorname']]
print(df)
```

AUSGABE:

	Vorname	Nachname	Jahrgang	Vermögen (m)	weiblich
0	Michael	Jackson	1958	-400	0
1	Britney	Spears	1981	70	1
2	Adele		1988	220	1

Per `DataFrame.loc(BEDINGUNG)` kann man einen Teil der Daten selektieren. Hier die Selektion von allen Personen mit Jahrgang vor 1980.

In [72]:

```
print(df.loc[df['Jahrgang'] < 1980])
```

AUSGABE:

	Vorname	Nachname	Jahrgang	Vermögen (m)	weiblich
0	Michael	Jackson	1958	-400	0

Willst Du eine einzelne Spalte anschauen, kannst Du dies direkt mit dem Namen:

In [74]:

```
df["Vermögen (m)"]
```

AUSGABE:

0	-400
1	70
2	220

Name: Vermögen (m), dtype: int64

Das Gesamtvermögen der Stars lässt sich mit `.sum()` berechnen. Hierbei ist bei "Michael Jackson" ein stark negativer Wert gegeben, so dass das Gesamtergebnis negativ ist. Ebenso ist es möglich den Mittelwert `.mean()` oder den Median `.median()` zu berechnen.

In [75]:

```
df["Vermögen (m)"].sum()
```

AUSGABE:

-110

Bist Du nur am Vermögen der weiblichen Stars interessiert, kannst Du gemäss dem Attribut "weiblich" gruppieren (`.groupby()`). Es gibt nun zwei DataFrames welche über `.get_group()` erhalten werden können. Summiert man nun über die Group auf, werden alle numerischen Attribute aufsummiert. Beim Jahrgang macht dies weniger Sinn, als beim Vermögen. Männer haben hier ein Vermögen von insgesamt -400 und Frauen von insgesamt 290.

In [86]:

```
grp = df.groupby("weiblich")
print(grp.get_group(1))
grp.sum()
```

AUSGABE:

	Vorname	Nachname	Jahrgang	Vermögen (m)	weiblich
1	Britney	Spears	1981	70	1
2	Adele		1988	220	1

	<i>Jahrgang</i>	<i>Vermögen (m)</i>
<i>weiblich</i>		
<i>0</i>	1958	-400
<i>1</i>	3969	290

Um ein Überblick über größere Datensätze zu erhalten, kannst Du dir die ersten 5 Zeilen mit `.head()` anschauen. Das folgende Programm lädt dazu den Datensatz `wine` von `sklearn` und gibt die ersten 5 Zeilen aus.

In [101]:

```
from sklearn.datasets import load_wine
import pandas as pd
wine=load_wine()
df=pd.DataFrame(data=np.c_[wine['data'],wine['target']],columns=wine['feature_names']+['target'])
df.head()
```

AUSGABE:

	<i>alcohol</i>	<i>malic_acid</i>	<i>ash</i>	<i>alcalinity_of_ash</i>	<i>magnesium</i>	<i>total_phenols</i>	<i>..... proline</i>	<i>target</i>
<i>0</i>	14.23	1.71	2.43	15.6	127.0	2.80	1065.0	0.0
<i>1</i>	13.20	1.78	2.14	11.2	100.0	2.65	1050.0	0.0
<i>2</i>	13.16	2.36	2.67	18.6	101.0	2.80	1185.0	0.0
<i>3</i>	14.37	1.95	2.50	16.8	113.0	3.85	1480.0	0.0
<i>4</i>	13.24	2.59	2.87	21.0	118.0	2.80	735.0	0.0

Um eine Zusammenfassung der einzelnen Attribute zu erhalten verwende `.describe()`. Es wird nun die Gesamtanzahl (count), der Mittelwert (mean), die Standardabweichung (std), das Minimum und Maximum (min, max) und die Quantile zu 25%, 50% und 75% angezeigt. Damit erhältst du einen groben Einblick in die Verteilung der Werte.

In [102]:

```
df.describe()
```

AUSGABE:

	<i>alcohol</i>	<i>malic_acid</i>	<i>ash ...</i>	<i>proline</i>	<i>target</i>
<i>count</i>	178.000000	178.000000	178.000000	178.000000	178.000000
<i>mean</i>	13.000618	2.336348	2.366517	746.893258	0.938202
<i>std</i>	0.811827	1.117146	0.274344	314.907474	0.775035
<i>min</i>	11.030000	0.740000	1.360000	278.000000	0.000000
<i>25%</i>	12.362500	1.602500	2.210000	500.500000	0.000000
<i>50%</i>	13.050000	1.865000	2.360000	673.500000	1.000000
<i>75%</i>	13.677500	3.082500	2.557500	985.000000	2.000000
<i>max</i>	14.830000	5.800000	3.230000	1680.000000	2.000000

Du kannst auch überprüfen, dass es keine Fehlenden Daten gibt.

```
df.isna().sum()
```

Der letzte Wert gibt ein *target* an, welches zwischen 0,1 oder 2 ist.

In [104]:

```
df['target'].value_counts()
```

AUSGABE:

```
1.0    71
0.0    59
2.0    48
Name: target, dtype: int64
```

Task 8.1: Wie viele Weine haben einen Magnesiumgehalt über 150?

Task 8.2: Gib den durchschnittlichen Alkoholgehalt für jede der drei Klassen *target* an.

Task 8.3: Speichere alle Weine mit einem *target* Wert von 1 als *wein.xlsx* ab.

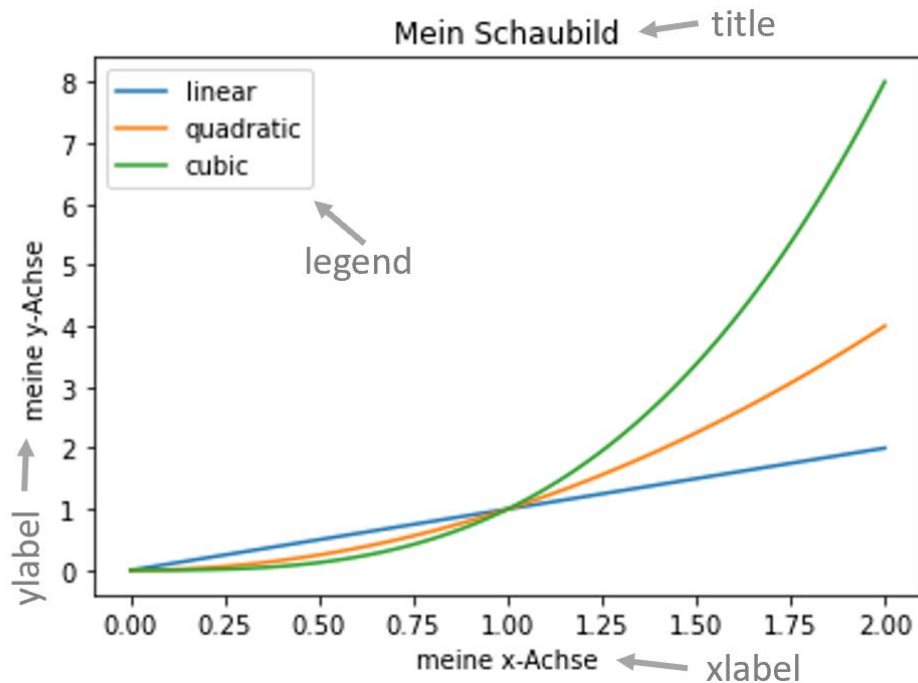
Stunde 9: Grafiken (Matplotlib)

Der beste Weg, um Daten darzustellen, ist eine Visualisierung. Menschen denken visuell. Wenn Du etwas präsentierst, kann sich Dein Publikum meistens danach nicht mehr an einzelne Tabellen und Zahlen erinnern. Eine Kurve jedoch bleibt im Gedächtnis.

Du lernst im Folgenden Linien Punkte, Säulen, Boxplots und Kuchendiagramme. Je nach Problem ist ein anderer Plot notwendig. Es gibt zwei Arten von Variablen. *Kategorische Variablen* sind Variablen welche zu einer begrenzten Anzahl von Kategorien gehören. Beispielsweise Länder (USA, China, Japan,...) oder Geschlecht (männlich, weiblich). *Stetige Variablen* sind Variablen welche die unterschiedlichsten Werte annehmen können. Beispiel Marktanteil (0%-100%) oder Einkommen ($f_0 - \infty$ Euro). Meist hat man zwei Variablen welche zusammenhängen: Bruttoinlandsprodukt (stetig) pro Land (kategorisch). Wann welcher Plot geeignet ist, entnimmst Du folgender Übersicht:

Vergleich	Geeignete Charts		Fragestellung	Beispiel
Teile (absolut)	Säulen	Wasserfall	• Größer bzw. kleiner • Davor bzw. danach • Reihenfolge	Verkaufte Produkte
Anteil (relativ)	Kuchen	gestapelte Säulen	• Anteil von... • Prozent von... • Beitrag zu...	Marktanteil
Zeitreihe	Säulen	Linien	• Veränderung (Trend) • Schwankungen	Aktienpreis
Verteilung	Histogramm	Boxplot	• Schwerpunkt • Häufigkeit zwischen... • Normal Verteilung	Körpergröße
Korrelation	Streudiag.	gruppierte Säulen	• Zusammenhang • Wächst mit...	Preis vs. Anzahl Verkäufe

Grafiken können in Python am besten mit dem Paket *Matplotlib* erzeugt werden. Folgende essenzielle Parameter können festgelegt werden:



Punkte und Linien

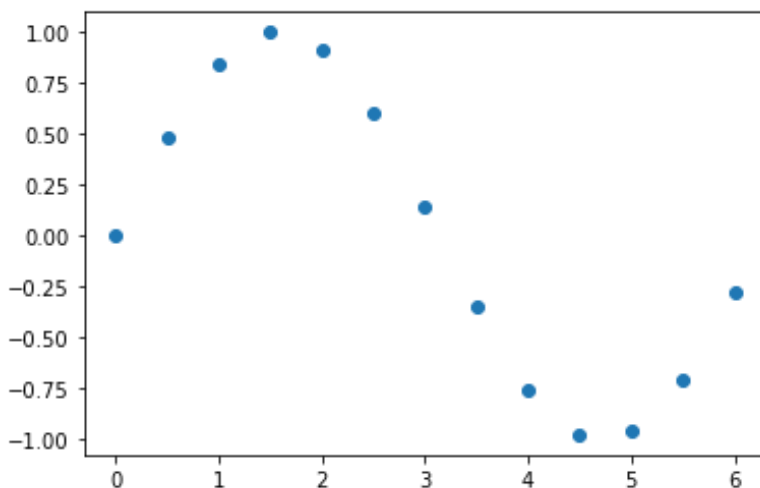
Um Punkte zu zeichnen verwendest Du `matplotlib.pyplot.scatter(x,y)`. Meist wird `matplotlib.pyplot` als `plt` geladen. Im Folgenden wird Funktion $f(x)=\sin(x)$ für gewisse Werte von x dargestellt.

In [188]:

```
import matplotlib.pyplot as plt
import numpy as np

# zerlege Interval von 0 bis 6 in 13 Teile
x = np.linspace(0,6, 13)
plt.scatter(x,np.sin(x)) # Punkte(x,f(x))
plt.show()
print("x-Werte:",x)
```

AUSGABE:



x-Werte: [0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6.]

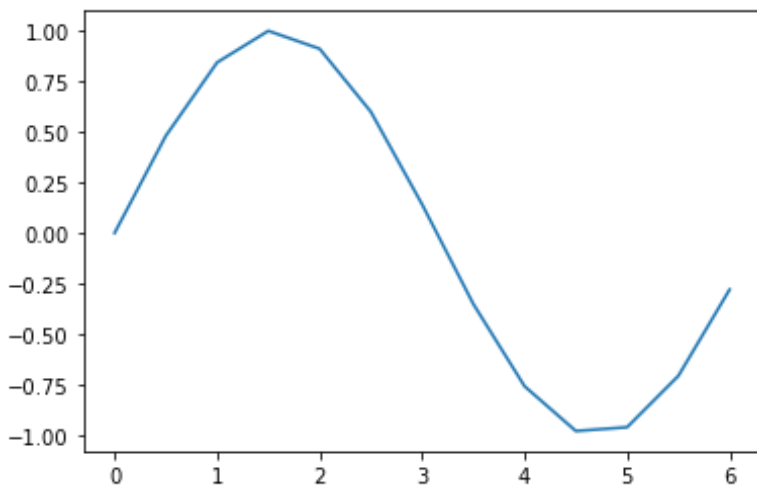
Ein Computer kann nicht alle stetigen Werte berechnen, sondern nur eine endliche Anzahl von Werten. Deshalb wird der Zahlenbereich 0-6 in 13 Werte unterteilt (`np.linspace(START, ENDE, ANZAHL)`). Für diese Werten wird `sin(x)` berechnet.

Da der Sinus eigentlich eine stetige Kurve ist, ist es besser ihn als Linie zu zeichnen. Dazu verwendest Du `matplotlib.pyplot.plot(x,y)`. Dies verbindet die obigen Punkte zu einer durchgezogenen Linie.

In [190]:

```
x = np.linspace(0,6, 13)
plt.plot(x,np.sin(x)) # plot(x,f(x))
plt.show()
print("x-Werte:",x)
```

AUSGABE:



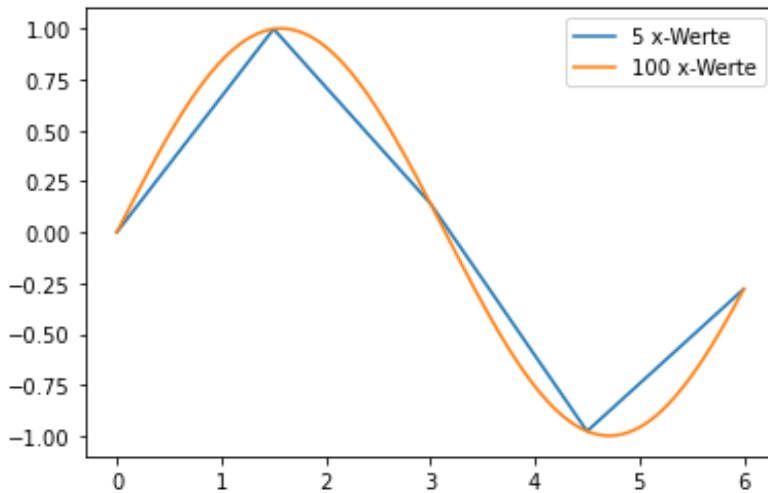
x-Werte: [0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6.]

Die Kurve erscheint es eckig, da die Punkte einfach mit geraden Linien verbunden werden. Erhöhe die Anzahl der Punkte (aktuell 13 so genannte Stützstellen). Die Kurve erscheint gleichförmiger (differenzierbar), da mehr Punkte auf engem Raum sind. Für wenige Punkte ist die Kurve erscheint die Kurve sehr eckig.

In [191]:

```
x1 = np.linspace(0,6, 5) # nur 5 Werte berechnet
plt.plot(x1,np.sin(x1), label="5 x-Werte")
x2 = np.linspace(0,6, 100) # 100 Werte berechnet
plt.plot(x2,np.sin(x2), label="100 x-Werte")
plt.legend()
plt.show()
```

AUSGABE:



Im obigen Beispiel wurden in einem Schaubild zwei Kurven eingetragen, welche jeweils eine eigene Bezeichnung (label) bekommen haben. Dazu wurde eine Legende (plt.legend()) angezeigt.

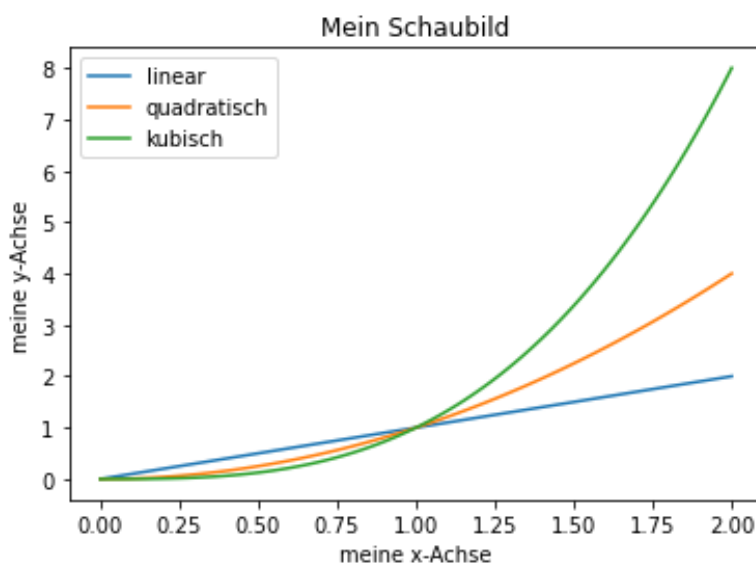
Du kannst ebenfalls einen Titel hinzufügen (plt.title()) und die X-Achse (plt.xlabel()) bzw. die Y-Achse (plt.ylabel()) bezeichnen.

Schau Dir dazu folgendes Beispiel mit $f(x)=x$, $f(x)=x^2$ und $f(x)=x^3$ an:

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
plt.plot(x,x, label="linear")
plt.plot(x,x**2, label="quadratisch") # plot(x,y)
plt.plot(x,x**3, label="kubisch")
plt.xlabel("meine x-Achse")           # X-Achsenbeschreibung
plt.ylabel("meine y-Achse")          # Y-Achsenbeschreibung
plt.title("Mein Schaubild")          # Großer Titel
plt.legend()                          # Legende anzeigen
plt.show()                           # Plot anzeigen
```

AUSGABE:



Task 9.1: Plote die Log-Funktion von 0.01 bis 6 (Tipp: `np.log(.)`).

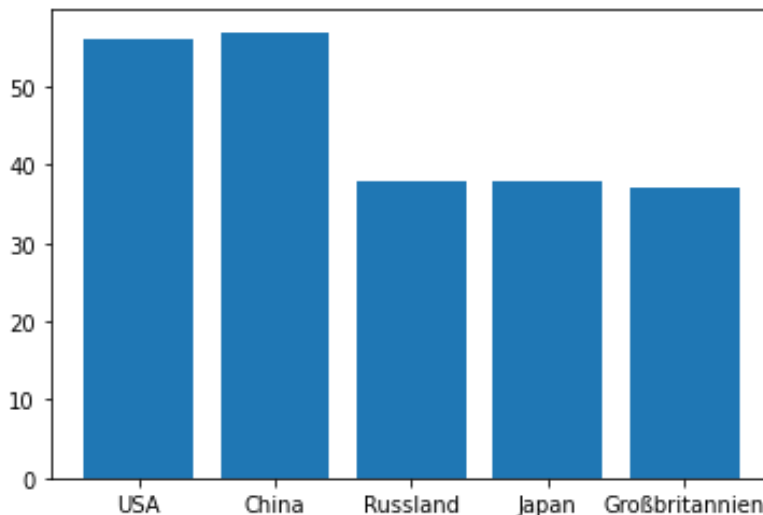
Säulen (bar)

Säulendiagramme sind besonders hilfreich, wenn unterschiedliche Gruppen (kategorische Variablen) bezüglich einer stetigen Variablen miteinander vergleicht. Beispielsweise die Olympia-medailen für unterschiedliche Länder.

In [211]:

```
laender = ["USA", "China", "Russland",  
           "Japan", "Großbritannien"]  
medaillen = [56, 57, 38, 38, 37]  
  
plt.bar(laender, medaillen)  
plt.show()
```

AUSGABE:

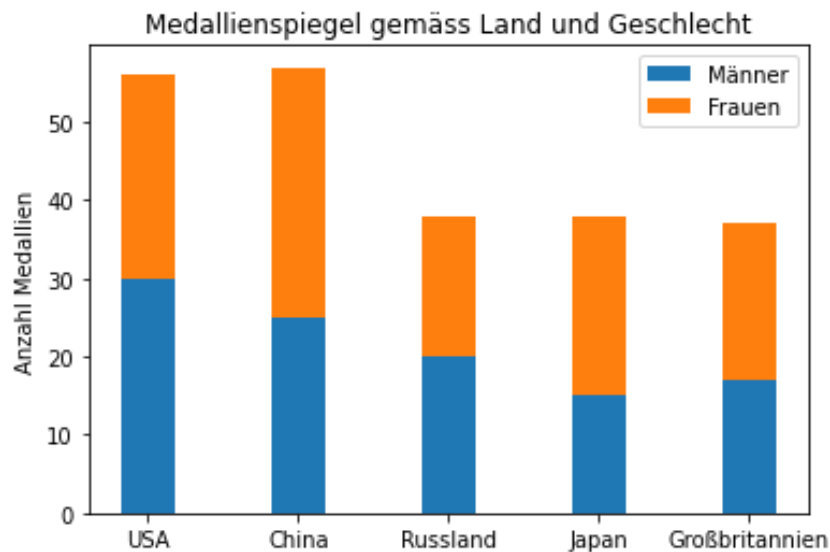


Nun ein etwas komplizierteres Beispiel. Der Anteil von Frauen und Männern soll dargestellt werden. Dies wird mit zwei Säulendiagrammen gemacht. Beachte, dass die Säule der Frauen nach der Säule der Männer anfängt (`plt.bar(.,bottom=maenner)`).

In [213]:

```
laender = ["USA", "China", "Russland",  
           "Japan", "Großbritannien"]  
maenner = [30, 25, 20, 15, 17]  
frauen = [26, 32, 18, 23, 20]  
  
x = np.arange(len(laender))  
breite = 0.35  
  
bar1 = plt.bar(laender, maenner, breite, label = "Männer")  
bar2 = plt.bar(laender, frauen, breite, bottom=maenner, label = "Frauen")  
  
plt.title("Medallienspiegel gemäss Land und Geschlecht")  
plt.ylabel("Anzahl Medallien")  
plt.legend()  
plt.show()
```

AUSGABE:

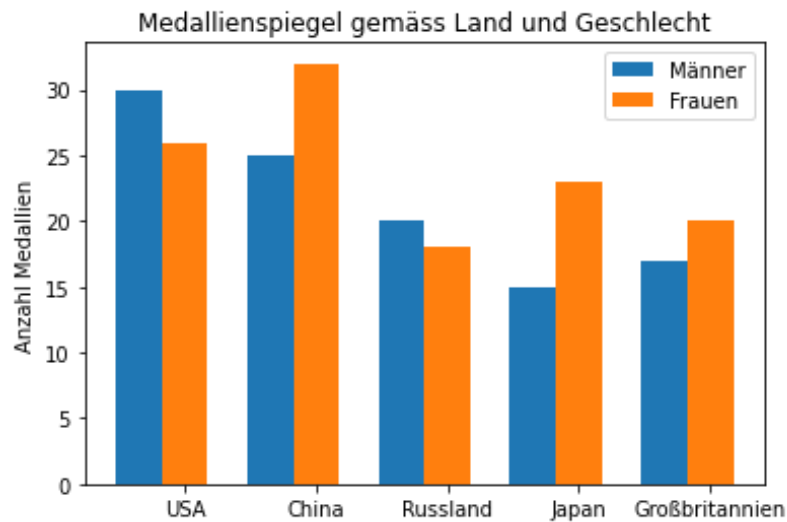


Wenn jedoch bsp. nur Frauen der Länder miteinander verglichen werden sollen, eignet sich die Darstellung nicht. Besser ist es die Säulen mit $x - \text{breite}/2$ bzw. $x + \text{breite}/2$ nebeneinander zu setzen.

In [204]:

```
laender = ["USA", "China", "Russland",  
           "Japan", "Großbritannien"]  
maenner = [30, 25, 20, 15, 17]  
frauen = [26, 32, 18, 23, 20]  
  
x = np.arange(len(labels))  
breite = 0.35  
  
bar1 = plt.bar(x-breite/2, maenner, breite, label = "Männer") # setze etwas  
links  
bar2 = plt.bar(x+breite/2, frauen, breite, label = "Frauen") # setze etwas  
rechts  
  
plt.title("Medallienspiegel gemäss Land und Geschlecht")  
plt.ylabel("Anzahl Medallien")  
plt.xticks(x + breite/2, laender) # bezeichne Elemente auf X-Achse mit  
Laendern  
plt.legend()  
plt.show()
```

AUSGABE:



Kuchen (Pie)

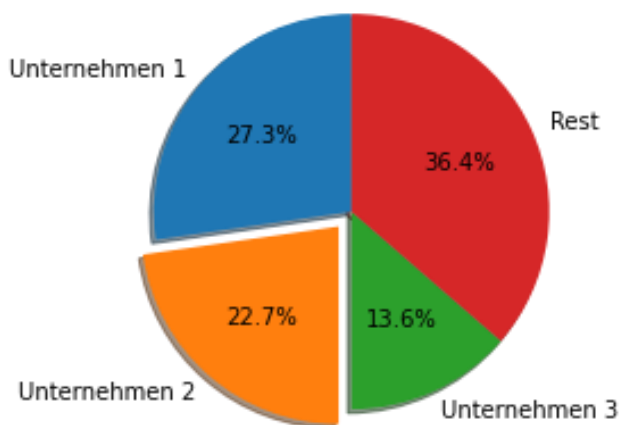
Ein Kuchendiagramm (Pie-chart) zeigt man meist, um die Aufteilung von deiner Sache aufzulisten. Beispielsweise wie groß der Marktanteil unterschiedlicher Unternehmen sind (wobei 100% der gesamte Markt ist).

In [217]:

```
names = ["Unternehmen 1", "Unternehmen 2", "Unternehmen 3", "Rest"]
value = [30, 25, 15, 40] # Umsatz der Unternehmen

explode = (0,0.1,0,0) # hebt zweites Unternehmen hervor
plt.pie(value, labels = names,
        shadow=True, startangle=90,
        autopct="%1.1f%%", explode=explode)
plt.show()
```

AUSGABE:



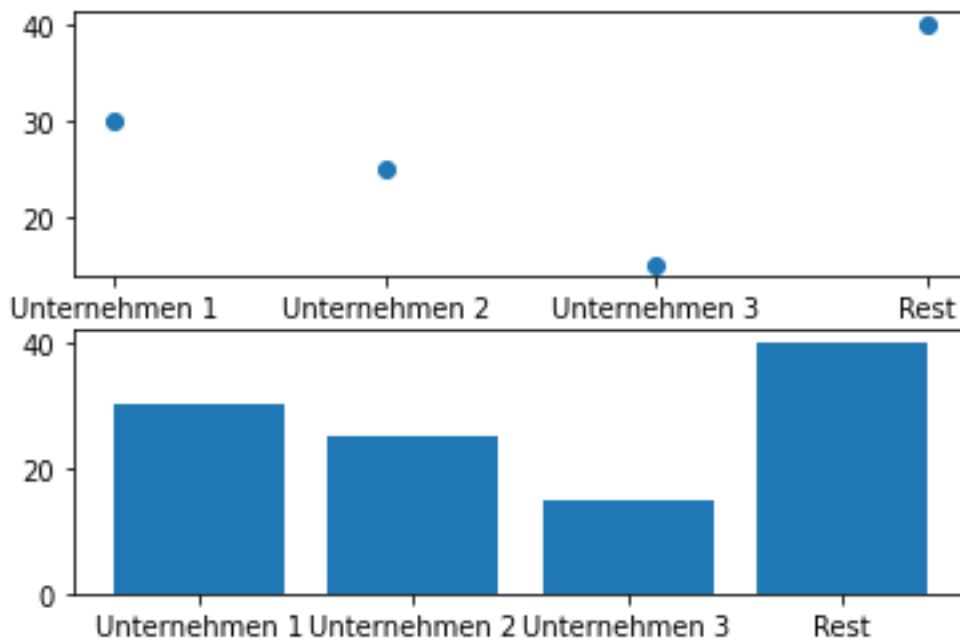
Punktwolke (Scatter)

Eine Punktwolke (Scatter-plot) hast Du bereits kennengelernt. Diese ist weniger sinnvoll bei kategorischen Variablen. Hier würde man eher ein Säulendiagramm verwenden. Im nächsten Beispiel lernst Du auch wie man mehrere Plots gleichzeitig erstellen kann.

In [232]:

```
fig, (ax1, ax2) = plt.subplots(2)    # erstelle zwei Plots
ax1.scatter(names, value)             # Scatterplot -> schlecht
ax2.bar(names, value)                 # bars -> besser
plt.show()
```

AUSGABE:

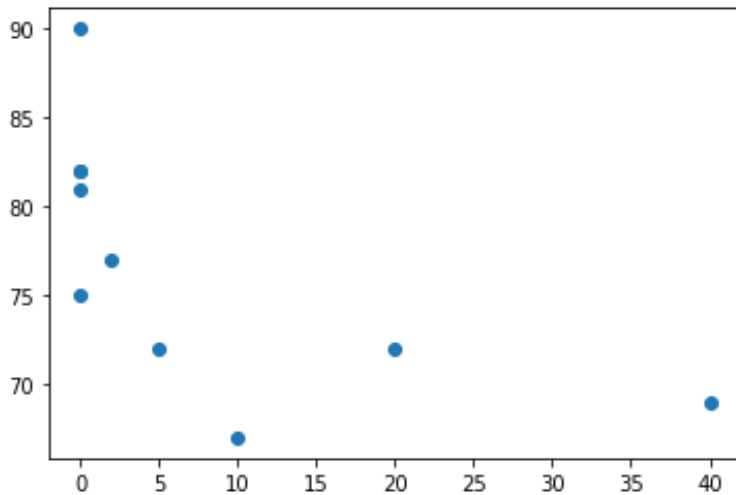


Scatterplots sind dann sinnvoll wenn man einen Zusammenhang zwischen zwei stetigen Variablen untersucht wird. Beispielsweise einen Zusammenhang zwischen erreichter Lebensdauer und Anzahl täglicher Zigaretten.

In [221]:

```
zigaretten = [ 0, 0, 10, 5, 0, 0, 0, 20, 2, 40]
lebensdauer = [75, 82, 67, 72, 82, 81, 90, 72, 77, 69]
plt.scatter(zigaretten, lebensdauer)
plt.show()
```

AUSGABE:



Task 9.2: Definiere für 4 Fächer die Anzahl der Stunden, die Du pro Woche in das jeweilige Fach investierst. Stelle dies in unterschiedlichen Charts dar.

Farben und Stricharten

Wenn Du unterschiedliche Kurven anzeigen willst, dann kannst Du die Farben (`plot(.,color)`), Liniendicke (`plot(.,linewidth)`) und die Linienart (`plot(.,linestyle)`) wählen. Farben gibt es alle möglichen Farben ("black", "grey", "green", "cyan", "orange",...). Für fast alle Farben gibt es die Möglichkeit die einen hellen Ton ("lightyellow") oder dunklen Ton ("darkred") davon zu bekommen. Es ist auch möglich RGB Farben mit Transparenz (rot, grün, blau, transparenz) darzustellen. Die Werte sind immer zwischen 0 und 1, bsp. (rot, grün, blau, transparenz) = (0.2, 0.3, 0.4, 0.1). Es können auch Hex-Farben in Hex-Code angegeben werden, bsp. "#ff11bb".

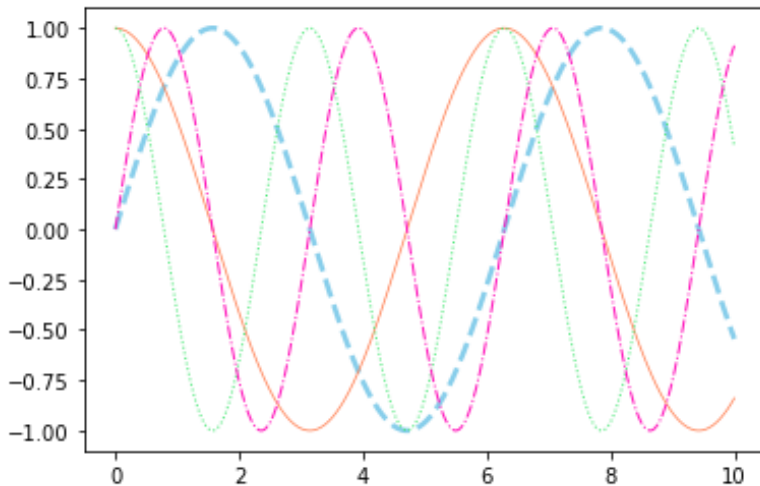
Die Liniendicke wird als Float angegeben. Bei der Linienart kann zwischen solid ("-"), dotted (":"), dashed ("--"), dashdot ("-.") gewählt werden.

In [254]:

```
x= np.linspace(0,10,500)
plt.plot(x,np.sin(x), linestyle="dashed",      # dashed blue
         linewidth=2.5, color="skyblue")
plt.plot(x,np.cos(x), linestyle="-",          # solid orange
         linewidth=0.75, color="coral")
plt.plot(x,np.cos(2*x), linestyle=":",        # dotted green
         linewidth=1.00, color=(0.2, 0.9, 0.4, 0.9))
plt.plot(x,np.sin(2*x), linestyle="-.",       # dashdot pink
         linewidth=1.00, color="#ff11bb")

plt.show()
```

AUSGABE:



Task 9.3: Erstelle ein Plot von $\cos(1/x)$ im Bereich 0.01 bis 0.1. Was passiert, wenn man zu wenige und genug Stützstellen angibt. Teste unterschiedliche Farben und Linienarten.

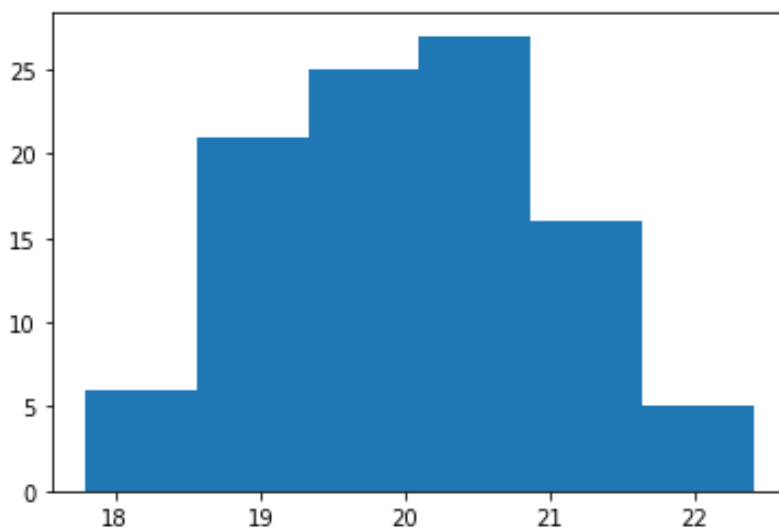
Histogramm

Wenn Du wissen willst, wie eine stetige Variable verteilt ist, hilft Dir am besten das Histogramm oder Box-Plot weiter. Betrachte beispielsweise eine Liste von 100 Studierenden. Nun möchtest Du wissen, wie die Alter verteilt sind; d.h. was ist das durchschnittliche Alter, was ist ein hohes bzw. tiefes Alter, in welchem Altersbereich sind viele bzw. wenige Studierende. Für ein Histogramm nimmst Du Altersklassen (bsp. $[0,18]$, $[18,19]$, $[19,20]$, $[20,21]$, $[21,22]$, $[22,100]$) und zählst wie viele Studierende in einer Klasse sind. Dies trägst Du dann in ein Schaubild (Histogramm) ab. In folgendem Beispiel werden 100 zufällig Alter um 20 Jahre erzeugt (Normalverteilung).

In [264]:

```
# erzeuge 100 normalverteilte Werte um Wert 20
alter = np.random.randn(100)+20
plt.hist(alter, bins=6) # Histogramm der Altersverteilung mit 6 Klassen
plt.show()
```

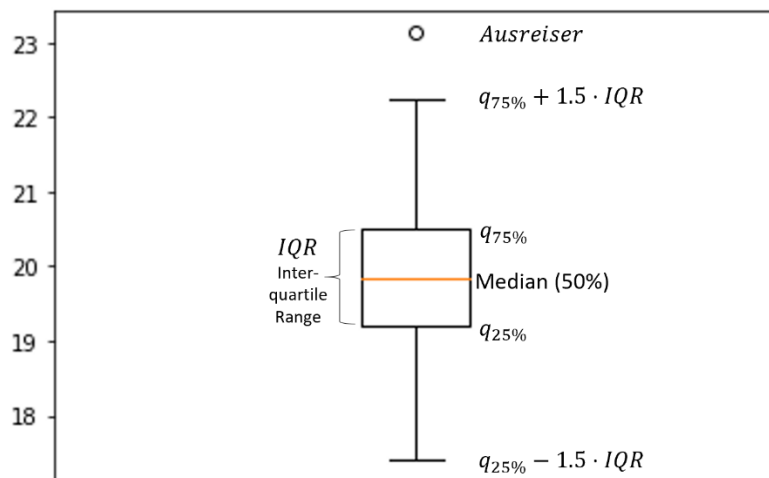
AUSGABE:



Das Histogramm gibt Dir eine gute Idee über die Altersverteilung der Studierenden. Bemerkung: Bei Dir wird das Histogramm (und die folgenden Plots) etwas anders aussehen, da die erzeugten Zufallszahlen sich jedes Mal ändern.

Boxplot

Eine Alternative ist der Boxplot. Hierbei werden die Alter erstmal aufsteigend sortiert. Ein Quantil sagt, welches Person gewählt wird. Bei 100 Personen entspricht das 25% Quantil $q_{25\%}$ der 25-jüngsten Person (d.h. 24 Personen sind jünger, 75 Personen sind älter). Das 75% Quantil $q_{75\%}$ ist die 25-älteste (bzw. 75 jüngste) Person. Der Median ($q_{50\%}$) entspricht der 50igsten Person. Das Boxplot ist nun wie folgt definiert:

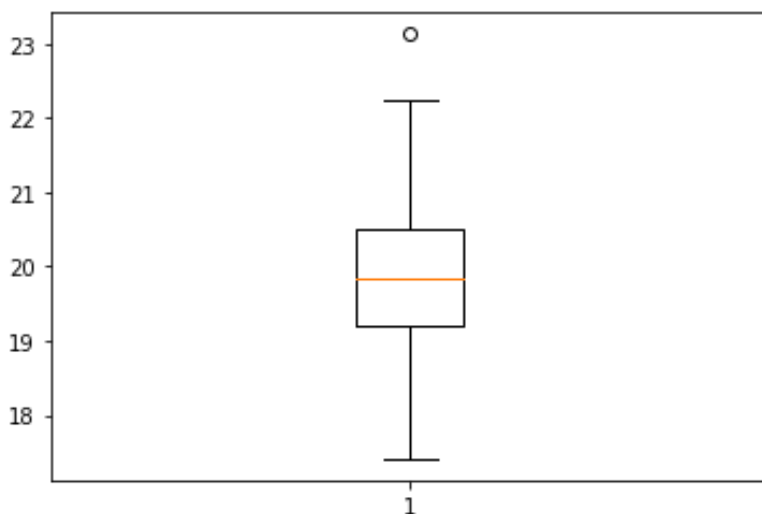


Schaue Dir dazu die Implementation mit Matplotlib an.

In [261]:

```
plt.boxplot(alter)
plt.show()
```

AUSGABE:



Du bekommst einen guten Einblick was der typische Wert (Median) bzw. typische Wertebereich (im IQR) ist.

Mehrere Variablen

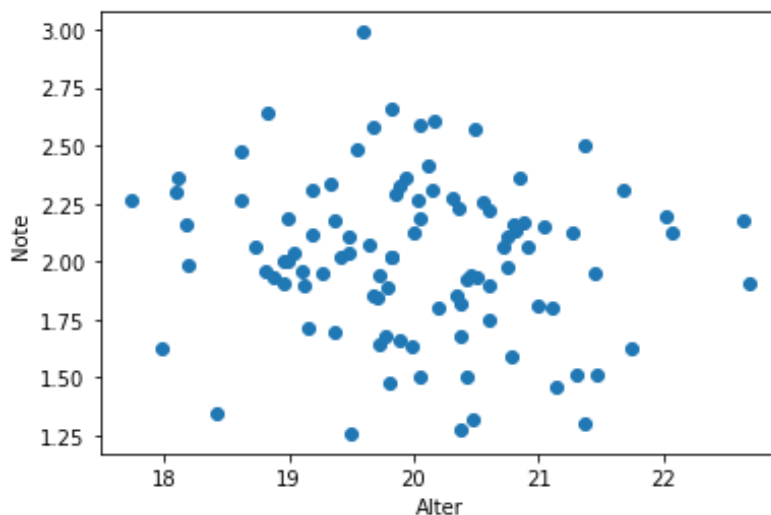
Möchtest Du den Zusammenhang von zwei stetigen Variablen ergründen, hilft Dir das bereits Scatterplot weiter. Ein Beispiel wäre, wenn Du nun den Zusammenhang zwischen Alter und Note ergründen möchtest.

In [3]:

```
# Normalverteilte Werte
alter = np.random.randn(100)+20
note = np.random.randn(100)/3+2

plt.scatter(alter,note)
plt.xlabel('Alter')
plt.ylabel('Note')
plt.show()
```

AUSGABE:



In obiges Beispiel ist kein Zusammenhang zwischen Note und Alter zu sehen. Dies war auch zu erwarten, da die Zufallszahlen für Alter und Note unabhängig voneinander erstellt wurden.

Task 9.4: Erstelle zwei mal 100 normal verteilte Zufallszahlen (x,y). Erstellen den plot für x und z, gegeben

1. $z = y$
2. $z = y+x$
3. $z = y-x$

Du solltest für 1. keinen Zusammenhang zwischen den Variablen finden. Für 2 und 3 solltest Du einen positiven (bzw. negativen) Zusammenhang finden. Positiver Zusammenhang bedeutet, wenn eine Variable groß ist, ist auch (tendenziell) die andere hoch.

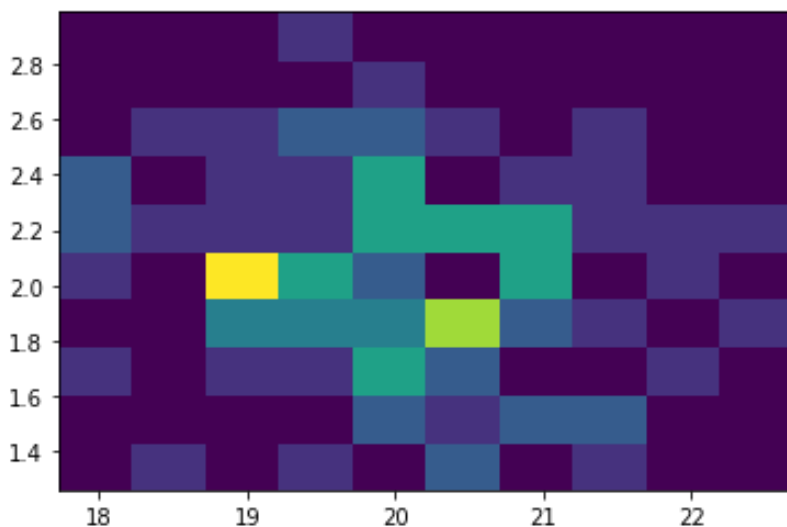
2D Histogramme

Das obige Beispiel kannst Du auch in einem Heatmap oder 2D Histogramm darstellen lassen. Hier siehst Du typische Wert. In unserem Beispiel scheint es keinen Zusammenhang zwischen Note und Alter zu geben.

In [5]:

```
plt.hist2d(alter,note)
plt.show()
```

AUSGABE:



3D Plotting

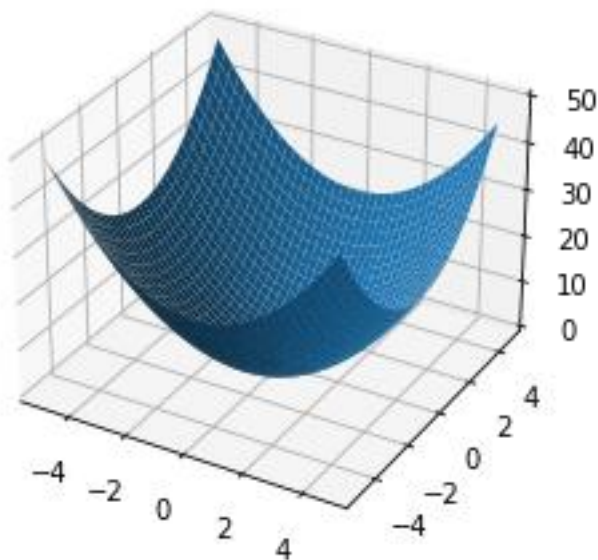
Wenn Du mehr als zwei Variablen darstellen möchtest, kannst Du dies über einen 3D Plot tun. Hierbei jedoch die Warnung, dass die Plots zwar oft toll aussehen, aber tatsächliche Zusammenhänge und Erkenntnisse schwer abzulesen sind. Am besten ist dies für Formeln mit zwei Variablen. Schau dazu die Darstellung von x^2+y^2 an.

In [10]:

```
X = np.arange(-5,5,0.25)
Y = np.arange(-5,5,0.25)
X,Y = np.meshgrid(X,Y)
Z = (Y**2+X**2) # oder np.sin(np.sqrt(X**2 + Y**2))

# zeige 3D-surface plot
surf = plt.gca(projection="3d").plot_surface(X,Y,Z)
```

AUSGABE:



Plots mit 4 Variablen

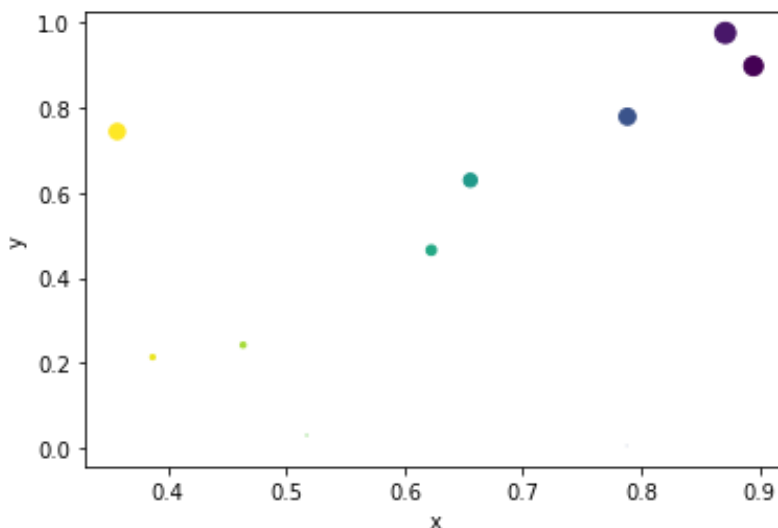
Bei 4 Variablen kann man auf die x-Achse, y-Achse, die Farbe und Größe der Plots übergehen. Allerdings gilt auch hier, dass die Übersichtlichkeit dann schnell nachlässt. Es kann jedoch hilfreich sein, wenn Größe bsp. den Umsatz repräsentiert und Farbe unterschiedliche Kategorien wie bsp. die Branche.

In [11]:

```
x = np.random.rand(10)
y = np.random.rand(10)
z = np.cos(x)
w = np.square(y*10)

plt.scatter(x,y, c=z, s=w) # color = z, size = w
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

AUSGABE:



Task 9.5: Erstelle einen Scatterplot mit zufälligen Zahlen x, y . Während die Farbe und Größe durch $z = 50 \cdot (x^2 + y^2)$ gegeben ist.

Kurzes Schlusswort

Herzlichen Glückwunsch!! Du hast es geschafft. Nun hast Du die Grundkenntnisse von Python verstanden. Du weißt wie man Variablen definiert, Schleifen erzeugt, Klassen erstellt, Funktionen aufruft, Dateien schreibt, Formeln berechnet und Grafiken darstellt. Das ist eine ganze Menge! Wende nun das Gelernte auf Dein Projekt an.

Schaue auch gern auf <https://python-programmieren.de/> für weitere hilfreiche Videos und Unterlagen. Dort lernst Du beispielsweise:

- eigene Oberflächen (Fenster)
- Web-Anwendungen
- Datenbanken mit Python selbst programmieren! Herzliche Einladung zu <https://python-programmieren.de/>.

Viel Spaß weiterhin mit Python! Dein Peter Schanbacher

Lösungen

Aufgabe 1.1: Setze eine Variable `x` als eine Zahl. Teste ob die Zahl gerade ist (Ausgabe 0) oder ungerade (Ausgabe 1).

In [10]:

```
x=5
print(x % 2)    # x mod 2 -> für gerade = 0, ungerade = 1
```

AUSGABE:

1

Aufgabe 1.2: Definiere eine Variable mit (D)einer Größe (in m) und eine mit (D)einem Gewicht (in kg). Berechne den Body-Mass BMI=Gewicht (in kg)/Groesse² (in m).. Diesen am besten auf eine Stelle runden. Gib am Ende aus: "Der Body-Mass-Index beträgt BMI".

In [13]:

```
size = 1.70
weight = 75
bmi = weight / size**2
bmi = round(bmi)
print("Der BMI beträgt", bmi)
```

AUSGABE:

Der BMI beträgt 26

Aufgabe 1.3:

In [23]:

```
name = "Peter Schanbacher"
p = name.find(" ")
print(name[:p])
```

AUSGABE:

Peter

Aufgabe 2.1: Mit folgendem Code erhältst Du eine Liste mit den Buchstaben von Deinem Namen. Verändere einige Buchstaben und füge die Buchstaben wieder zu einem String zusammen.

In [30]:

```
name = []
name[:] = "dein name"
name[0:4] = ["d", "e", "r", " "]
print(str().join(name))
```

AUSGABE:

der name

Aufgabe 2.2: Erstelle ein Dictionary mit dem Namen und Alter Deiner Freunde. Gib die Namen aus.

In [75]:

```
myDictionary = {'Peter':102, 'Conny':23, 'Kira':33, 'Jonah':24, 'Lian': 5}
```

```
print(myDictionary.keys())
```

AUSGABE:

```
dict_keys(['Peter', 'Conny', 'Kira', 'Jonah', 'Lian'])
```

Aufgabe 2.3: Bestimme die Anzahl der eindeutige Buchstaben Deines Names. Gib die Buchstaben Deines Namens ohne Vokale (aeiou) aus.

In [108]:

```
name = 'Peter'
myNameSet = set(name)
print(len(myNameSet))
myNameSet2 = {x for x in name if x not in 'aeiou'}
print(myNameSet2)
```

AUSGABE:

```
4
{'t', 'r', 'P'}
```

Aufgabe 3.1: Lasse Dir vom Benutzer eine positive oder negative Zahl eingeben. Gib den absoluten Wert der Zahl aus (d.h. -3 wird zu 3).

In [34]:

```
x = float(input("Bitte Zahl eingeben: "))
if x < 0:
    x = -x
print("Der absolute Wert beträgt:",x)
```

AUSGABE:

```
Bitte Zahl eingeben: -34
Der absolute Wert beträgt: 34.0
```

Aufgabe 3.2: Lasse Dir vom Benutzer einen Namen eingeben. Frage ab, ob der Name "Tony", "Tom" oder "Tina" ist.

In [16]:

```
name = input("Bitte Name eingeben")

if name == "Tony":
    print("Tony ist bekannt")
elif name == "Tom":
    print("Tom ist bekannt")
elif name == "Tina":
    print("Tina ist bekannt")
else:
    print("Name ist nicht bekannt")
```

AUSGABE:

```
Bitte Name eingebenTina
Tina ist bekannt
```

oder alternativ

In [64]:

```

name_list = ["Tony", "Tom", "Tina"]
name = input("Bitte Name eingeben:")
if name in name_list:
    print("Name ist bekannt")
else:
    print("Name ist nicht bekannt")

```

AUSGABE:

```

Bitte Name eingeben:Tony
Name ist bekannt

```

Aufgabe 3.3: Lasse den Benutzer solange sein Alter eingeben, bis die Eingabe positiv ist; d.h. bei negativem Alter wird erneut gefragt.

In [35]:

```

a = -1
while a<0:
    a = int(input("Bitte Alter eingeben:"))
print("Ihr Alter ist", a, "Jahre.")

```

AUSGABE:

```

Bitte Alter eingeben:-3
Bitte Alter eingeben:-3
Bitte Alter eingeben:5
Ihr Alter ist 5 Jahre.

```

Aufgabe 3.4: Was ist das Ergebnis wenn man alle Zahlen von 0 bis 100 addiert; d.h. $0+1+2+3+4+\dots+99+100=?$

In [9]:

```

my_sum = 0
i = 0
while i<=100:
    my_sum = my_sum+i
    i = i+1
print("0+1+2+...+100=",my_sum)

```

AUSGABE:

```

0+1+2+...+100= 5050

```

Aufgabe 3.5: Gib alle geraden Zahlen von 0 bis 10 aus.

In [50]:

```

for i in range(0,11,2):
    print(i, end=",")

```

AUSGABE:

```

0,2,4,6,8,10,

```

Aufgabe 3.6: Berechne die Summe aller Zahlen von 1 bis 100 (d.h. $1+2+3+\dots+100=?$) mit einer For Schleife.

In [11]:

```

my_sum = 0
for i in range(101): # i = 0,...,100 (w/o 101)
    my_sum = my_sum+i

```

```
print("0+1+2+...+100=",my_sum)
```

AUSGABE:

```
0+1+2+...+100= 5050
```

Aufgabe 3.7: Gib alle Primzahlen zwischen 0 und 100 an.

Tipp: Setze den besprochenen Primzahlen-Code in eine Schleife von 0 und 100. Passen den Code danach leicht an.

In [48]:

```
for n in range(2,101):
    prim = True
    for x in range(2,n):
        if n%x == 0:
            prim = False
            break
    if prim == True:
        print(n, end=",")
```

AUSGABE:

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,
```

Aufgabe 3.8: Frage den Benutzer wiederholt nach eine Pizza-zutat. Höre auf, sobald der Benutzer keine Angaben mehr macht (input=""). Gib alle Pizzazutaten aus (bsp. ["Käse", "Schinken", "Tomaten"])

In [49]:

```
pizza = []
while True:
    ing = input("Bitte geben Sie eine Zutat ein:")
    if ing == "":
        print(pizza)
        break
    else:
        pizza.append(ing)
```

AUSGABE:

```
Bitte geben Sie eine Zutat ein:Käse
Bitte geben Sie eine Zutat ein:Tomate
Bitte geben Sie eine Zutat ein:Pizza
Bitte geben Sie eine Zutat ein:
['Käse', 'Tomate', 'Pizza']
```

Task 3.9: Erstelle eine Liste der Quadratzahlen $0^2, 1^2, 2^2, \dots, 10^2$.

In [61]:

```
quadrat = [x**2 for x in range(11)]
print(quadrat)
```

AUSGABE:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Task 4.1: Erstelle eine Funktion welche der Vorname und Nachname als Argument erhält und den vollständigen Namen (Vorname + Nachname) zurückgibt.

In [56]:

```
def myName(vor, nach):  
    return vor + " " + nach  
print(myName("Max", "Mustermann"))
```

AUSGABE:

Peter Schanbacher

Task 4.2: Erstelle eine Funktion `istGerade(x)` welche zurückgibt ob `x` gerade (d.h. `True`) oder ungerade (`False`) ist.

In [7]:

```
def istGerade(x):  
    if (x%2==0):  
        return True  
    else:  
        return False
```

```
istGerade(3)
```

AUSGABE:

False

Task 4.3: Erstelle eine Funktion `istPrimzahl(n)` welche zurückgibt ob `n` eine Primzahl ist, oder nicht. Für die Bestimmung, ob eine Zahl eine Primzahl ist, siehe oben.

In []:

```
def istPrimzahl(n):  
    prime = True  
    for x in range(2,n):  
        if n%x==0: # lässt sich n durch x teilen -> keine Primzahl  
            prime = False  
            break  
    return prime
```

```
istPrimzahl(6)
```

Task 4.4: Erstelle eine Funktion `begruessen(gruss)` welche den String `gruss` ausgibt. Falls die Funktion ohne Argument aufgerufen wird, soll ein allgemeiner Gruss genannt werden.

In [18]:

```
def begruessen(gruss="guten Tag"):  
    print(gruss)  
begruessen("Hallo!")      # Hallo!  
begruessen()              # guten Tag
```

AUSGABE:

Hallo!
guten Tag

Task 4.5: Erstelle erneut eine Funktion `bmi(groesse, gewicht)` welche den BMI (Gewicht/Größe²) zurückgibt. Rufe die Funktion mit Keyword-Argumenten auf.

In [22]:

```
def bmi(groesse, gewicht):  
    bmi_value = gewicht/groesse**2  
    return round(bmi_value)  
h = float(input("Bitte Groesse (m):"))  
w = int(input("Bitte Gewicht (kg):"))  
print(bmi(gewicht=w, groesse=h))
```

AUSGABE:

```
Bitte Groesse (m):1.70  
Bitte Gewicht (kg):72  
25
```

Task 4.6: Erstelle eine Funktion *meinMinimum(..)* welche mehrere Zahlen erhält und die kleinste Zahl davon zurückgibt.

In [40]:

```
def meinMinimum(*zahlen):  
    return min(zahlen)  
  
print(meinMinimum(3,1,5))
```

AUSGABE:

```
1
```

Task 4.7: Die Funktion *randint(a,b)* im Paket Modul *random* vom Paket *numpy* gibt eine Zufallszahl zwischen *a* und *b* (exklusive). Lasse den Computer würfeln (Zahl zwischen 1 und 6). Nun soll der Benutzer raten welche Zahl gewürfelt wurde. Lasse den Benutzer so lange raten, bis er auf die richtige Zahl getippt hat.

In [12]:

```
from numpy.random import randint  
x = randint(1,7)  
#print(x)  
guess = -1 # guess auf einen Startwert der nicht gewürfelt werden kann  
while guess != x:  
    guess = int(input("Bitte Zahl raten (1-6):"))  
  
print("Richtig, die Zahl war", x)
```

AUSGABE:

```
Bitte Zahl raten (1-6):1  
Bitte Zahl raten (1-6):2  
Bitte Zahl raten (1-6):3  
Bitte Zahl raten (1-6):4  
Bitte Zahl raten (1-6):5  
Richtig, die Zahl war 5
```

Task 5.1: Erstelle eine Liste mit fehlenden Werten (NaNs). Filtere nun diese fehlenden Werte aus der Liste und erhalte eine neue bereinigte Liste ohne fehlende Werte.

In [73]:

```
liste = [1,2,3, float("nan"), 5]  
neue_liste = [x for x in liste if not math.isnan(x)]  
print(neue_liste)
```

AUSGABE:

[1, 2, 3, 5]

Task 5.2: Erstelle das Spiele-Schere-Stein Papier, bei denen der Benutzer gegen den Computer spielt.

In [86]:

```
import random
auswahl = ["Schere", "Stein", "Papier"]
rechner = random.choice(auswahl)
benutzereingabe = None
while benutzereingabe not in auswahl:
    benutzereingabe = input("Auswahl: Schere, Stein, Papier:")
if benutzereingabe == rechner:
    print("Unentschieden")
elif (benutzereingabe == "Schere" and rechner == "Papier") or \
      (benutzereingabe == "Stein" and rechner == "Schere") or \
      (benutzereingabe == "Papier" and rechner == "Stein"):
    print("Du hast gewonnen")
else:
    print("Du hast verloren!")

print("Rechner:", rechner, " vs. Du:", benutzereingabe)
```

AUSGABE:

Auswahl: Schere, Stein, Papier:Stein
Unentschieden
Rechner: Stein vs. Du: Stein

Task 5.3: Berechne an welchem Wochentag Du geboren bist und wie viele Tage Du bereits lebst.

In [109]:

```
from datetime import datetime
tag = date(1985,1,31)
print(tag.strftime("%d %b %Y war ein %A"))
alter = jetzt - tag
print(alter.days)
```

AUSGABE:

31 Jan 1985 war ein Thursday
13597

Task 5.4: Berechne alle Quadrate von 0 bis 1'000'000 (d.h. $0^2+1^2+\dots+1000000^2$) mit einer

a. Schleife

b. list comprehension

Messe jeweils die Zeit für die Berechnung.

Bemerkung: Eine List comprehension sollte effizienter und somit schneller als eine Schleife sein.

In [110]:

```
start = time.time()
z = []
```

```

for i in range(0,1000001):
    z.append(i**2)
stop = time.time()
print("Ende For Schleife:", stop-start)

start = time.time()
z = [k**2 for k in range(0, 1000001)]
stop = time.time()
print("Ende list comprehension:", stop-start)

AUSGABE:
Ende For Schleife: 1.110055923461914
Ende list comprehension: 0.8431367874145508

```

Task 6.1: Passe den obenstehenden Programmcode wie folgt an: der Benutzer wird so lange wiederholt aufgefordert eine Zahl einzugeben, bis er tatsächlich eine Zahl eingegeben hat.

In [32]:

```

while True:
    try:
        x = int(input("Bitte Zahl eingeben: "))
        break
    except ValueError:
        print("Das war leider keine Zahl. Bitte erneut eingeben.")
print("Sie haben", x, "eingegeben")

AUSGABE:
Bitte Zahl eingeben: 1
Sie haben 1 eingegeben

```

Task 6.2: Fange alle Fehler ab (keine Zahl eingegeben, durch 0 geteilt), die bei der Berechnung des body-mass-index passieren können (BMI=Gewicht (in kg)/Größe² (in m)).

In [43]:

```

while True:
    try:
        w = float(input("Bitte Gewicht angeben (in kg): "))
        break
    except ValueError: # ValueError -> keine Zahl
        print("Keine Zahl. Bitte erneut eingeben.")

while True:
    try:
        s = float(input("Bitte Größe eingeben (in m): "))
        break
    except ValueError: # ValueError -> keine Zahl
        print("Keine Zahl. Bitte erneut eingeben.")

try:
    print("BMI is {:.1f}".format(w/s**2))
except ZeroDivisionError as err: # durch 0 geteilt
    print('Kann nicht durch 0 teilen:', err)

AUSGABE:
Bitte Gewicht angeben: 72

```

Bitte Größe eingeben: 1.75
BMI is 23.5

Task 7.1: Erstelle eine Klasse Komplex welche eine Komplexe Zahl mit einem Realteil und einer Imaginärzahl definiert (https://de.wikipedia.org/wiki/Komplexe_Zahl). Eine komplexe Zahl soll bsp. mit `x = Komplex(3.0, -4.5)` erstellt werden.

In [40]:

```
class Komplex:
    def __init__(self, real, imag):
        self.r = real
        self.i = imag

x = Komplex(3.0, -4.5)
print("Komplexe Zahl =", x.r, " + i * ", x.i)
```

AUSGABE:

*Komplexe Zahl = 3.0 + i * -4.5*

Task 7.2: Wie muss das Programm angepasst werden, damit nur "Peter" und nicht "Conny" die Fächer hinzugefügt werden?

In []:

```
class Student:
    def __init__(self, name):
        self.name = name
        self.faecher = [] # erstellt leere Liste für jeden Student
    def add_fach(self, fach):
        self.faecher.append(fach)

p = Student('Peter')
c = Student('Conny')
p.add_fach('Mathematik')
p.add_fach('Programmieren')
print(p.name, "Fächer:", p.faecher)
print(c.name, "Fächer:", c.faecher)
```

Task 7.3: Erstelle eine Klasse Konto mit einem privaten Attribut `kontostand` und einer Methode `setzteKontostand(wert)`. Der Kontostand soll nicht negativ werden können.

In [63]:

```
class Konto:
    def __init__(self):
        self.__kontostand = 0

    def setzteKontostand(self, w):
        if w >= 0:
            self.__kontostand = w
        else:
            print("Kontostand darf nicht negativ werden")

    def nenneKontostand(self):
        print("Kontostand ist", self.__kontostand)

k = Konto()
k.setzteKontostand(-10)
k.nenneKontostand()
k.setzteKontostand(10)
```

```
k.nenneKontostand()
```

AUSGABE:

```
Kontostand darf nicht negativ werden
Kontostand ist 0
Kontostand ist 10
```

Task 7.4: Erstelle eine Klasse `Person` mit den Attributen *name* und *alter*, sowie einer Methode *vorstellen()*. Erstelle eine abgeleitete Klasse `Arbeitnehmer` mit dem zusätzlichen Attribut *jobtitel*. Überschreibe die Methode *vorstellen()*.

In []:

```
class Person:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def vorstellen(self):
        print("Hallo, ich bin", self.name, "und", self.alter, "Jahre alt.")

pers = Person("Peter", 35)
pers.vorstellen()

class Arbeitnehmer(Person):
    def __init__(self, name, alter, jobtitel):
        super().__init__(name, alter)
        self.jobtitel = jobtitel

    def vorstellen(self):
        print("Sehr geehrte Damen und Herren, mein Name ist",
              self.name, "und ich bin hier", self.jobtitel)

an = Arbeitnehmer("Conny", 32, "CEO")
an.vorstellen()
```

Task 7.5: Erstelle eine Klasse `Vektor` mit einem zwei-dimensionalen Vektor mit einem *x* und einem *y* Wert. Zwei Objekte von der Klasse `Vektor` sollen addiert und subtrahiert werden können.

```
a = Vektor(3,2)
b = Vektor(4,9)
c = a+b
```

Das Ergebnis soll ein neuer Vektor sein mit $x_c = x_a + x_b$ und $y_c = y_a + y_b$

In [75]:

```
class Vektor():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vektor(self.x+other.x, self.y+other.y)

    def __sub__(self, other):
        return Vektor(self.x-other.x, self.y-other.y)

    def __str__(self):
        return "X="+str(self.x)+" , Y="+str(self.y)
```

```

v1 = Vektor(3,2)
v2 = Vektor(4,9)
v3 = v1+v2
v4 = v1-v2

```

```

print(v1)
print(v2)
print(v3)
print(v4)

```

AUSGABE:

```

X=3,Y=2
X=4,Y=9
X=7,Y=11
X=-1,Y=-7

```

Task 8.1: Wie viele Weine haben einen Magnesiumgehalt über 150?

In [132]:

```

# lade Datensatz
from sklearn.datasets import load_wine
import pandas as pd
wine=load_wine()
df=pd.DataFrame(data=np.c_[wine['data'],wine['target']],columns=wine['feature_names']+['target'])
# Selektiere Weine mit Magnesium > Schwelle von 150
schwelle = 150
select = df.loc[df['magnesium'] > schwelle]
print(select)
print(select.ndim, "Weine haben einen Magensiumgehalt über", schwelle)

```

AUSGABE:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols
69	12.21	1.19	1.75	16.8	151.0	1.85
95	12.47	1.52	2.20	19.0	162.0	2.50

	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue
69	1.28	0.14	2.50	2.85	1.2
95	2.27	0.32	3.28	2.60	1.1

	od280/od315_of_diluted_wines	proline	target
69	3.07	718.0	1.0
95	2.63	937.0	1.0

2 Weine haben einen Magensiumgehalt über 150

Task 8.2: Gib den durchschnittlichen Alkoholgehalt für jede der drei Klassen *target* an.

In [136]:

```

# lade Datensatz wie bei 8.1
df.groupby('target')['alcohol'].mean()

```

AUSGABE:

```
target
0.0    13.744746
1.0    12.278732
2.0    13.153750
Name: alcohol, dtype: float64
```

Task 8.3: Speichere alle Weine mit einem *target* Wert von 1 als *wein.xlsx* ab.

In [135]:

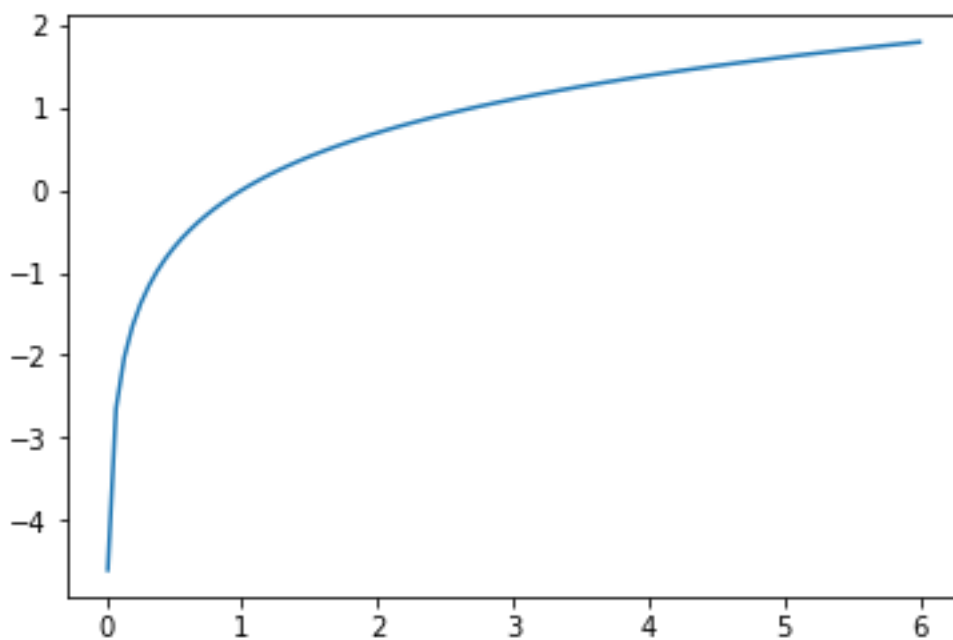
```
# lade Datensatz wie bei 8.1
grp = df.groupby('target')
select = grp.get_group(1)
select.to_excel("wein.xlsx", index=False) # als Excel
```

Task 9.1: Plote die Log-Funktion von 0.01 bis 6 (`np.log(.)`).

In [192]:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.01, 6, 100)
plt.plot(x, np.log(x))
plt.show()
```

AUSGABE:

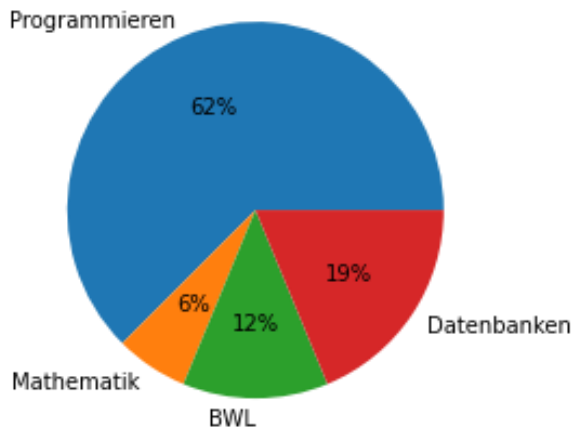


Task 9.2: Definiere für 4 Fächer die Anzahl der Stunden die Du pro Woche in das jeweilige Fach investierst. Stelle dies in unterschiedlichen Charts dar.

In [228]:

```
import matplotlib.pyplot as plt
label = ["Programmieren", "Mathematik",
         "BWL", "Datenbanken"]
sizes = [20, 2, 4, 6]
plt.pie(sizes, labels = label, autopct="%1.0f%%")
plt.show()
```

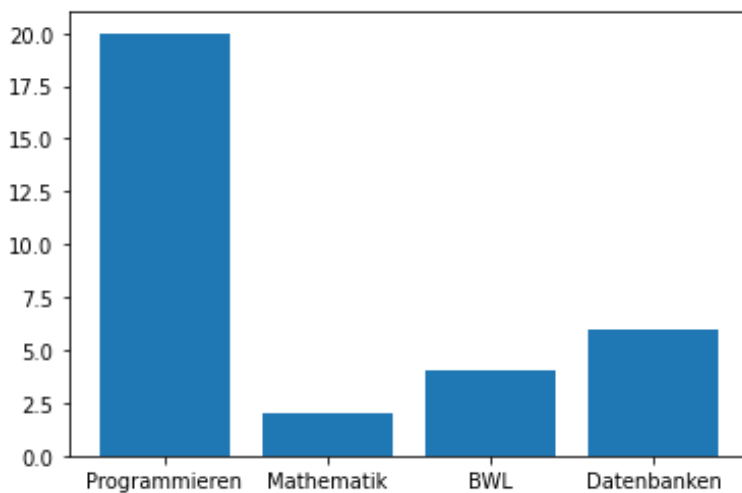
AUSGABE:



In [229]:

```
plt.bar(label, sizes)
plt.show()
```

AUSGABE:

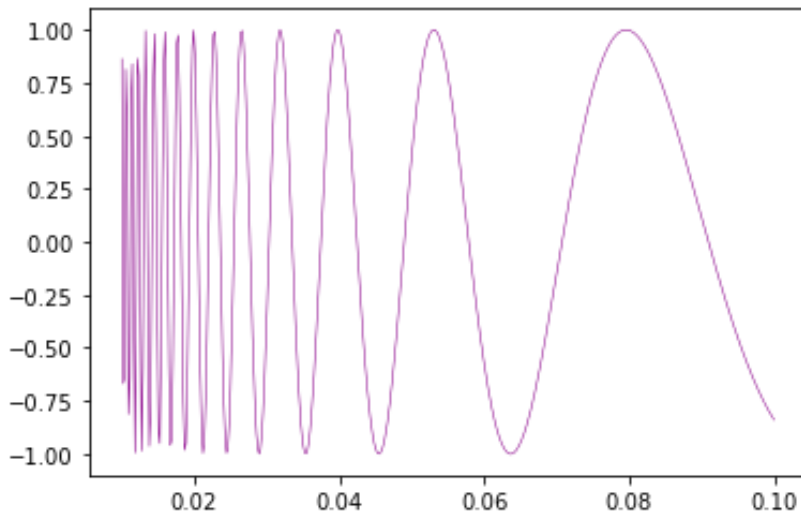


Task 9.3: Erstelle ein Plot von $\cos(1/x)$ im Bereich 0.01 bis 0.1. Was passiert, wenn man zu wenige und genug Stützstellen angibt. Teste unterschiedliche Farben und Linienarten.

In [259]:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.01, 0.1, 500)
y = np.cos(1/x)
plt.plot(x, y, linestyle="solid", color="darkmagenta", linewidth=0.5)
plt.show()
```

AUSGABE:



Task 9.4: Erstelle zwei mal 100 normal verteilte Zufallszahlen (x,y). Erstellen den plot für x und z, gegeben

1. $z = y$
2. $z = y+x$
3. $z = y-x$

Du solltest für 1. keinen Zusammenhang zwischen den Variablen finden. Für 2 und 3 solltest Du einen positiven (bzw. negativen) Zusammenhang finden. Positiver Zusammenhang bedeutet, wenn eine Variable groß ist, ist auch (tendenziell) die andere hoch.

In [4]:

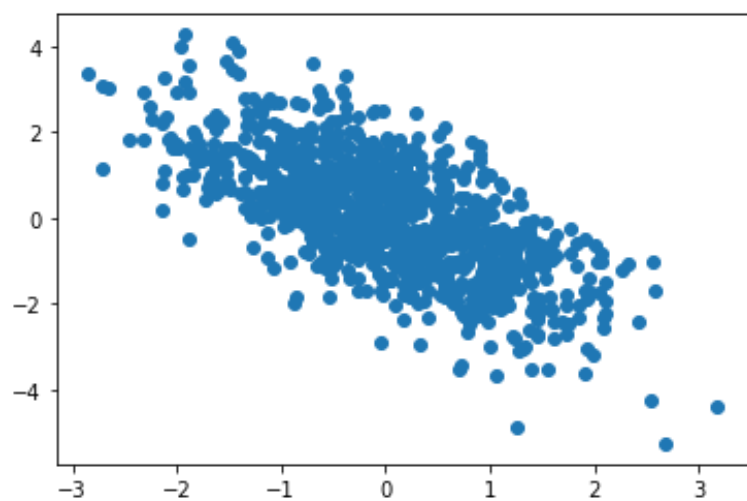
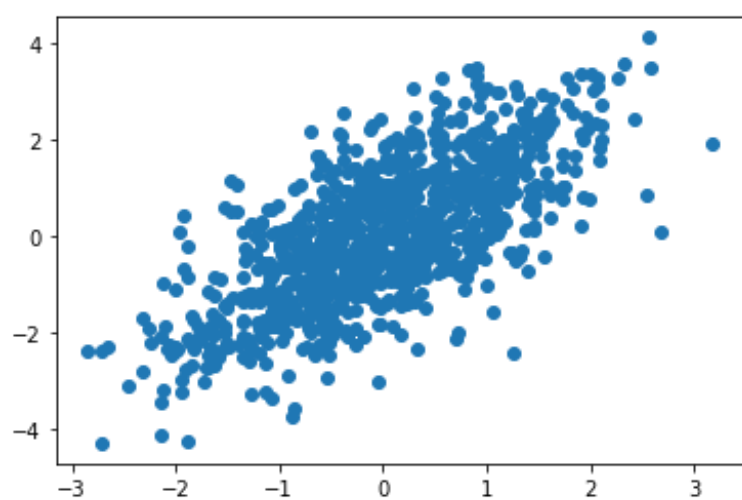
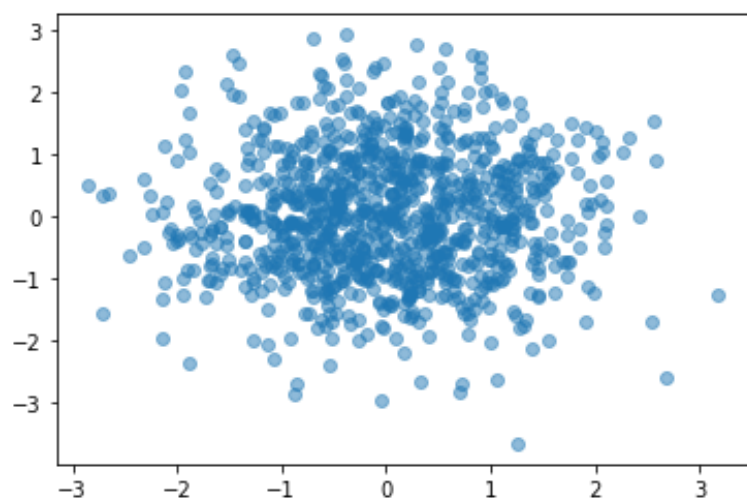
```
n = 1000
x = np.random.randn(n)
y = np.random.randn(n)

plt.scatter(x,y, alpha=0.5)
plt.show()

plt.scatter(x,y+x)
plt.show()

plt.scatter(x,y-x)
plt.show()
```

AUSGABE:



Spickzettel

In []:

```
# Variablen
ganzZahl = 1 # integer
text = "Python lernen" # Text or string
reelleZahl = 3.141 # float
text.split(" ") # -> [Python, lernen]
text.find("lern") # index 7

# Listen
liste = [1,2] # [1,2]
liste.append(3) # [1,2,3]
liste.pop(0) # [2,3]
print(liste)

# While
a = 0
while a<5:
    print(a)
    a=a+1

# For
for a in range(5):
    print(a)

# If
if a == 0:
    print("a ist Null")
else:
    print("a ist nicht Null")

# Funktionen
def summe(x,y):
    return x+y

summe(1,2) # return 1+2

# Fehler
try:
    1/0
except:
    print("Fehler")

# Klassen
class Elternklasse:
    pass # definiert nicht

class MeineKlasse(Elternklasse):
    klassenattribut = 0
    def __init__(self,arg):
        self.arg = arg
        self.attr = 1
    def summe(self, x,y):
        return x+y
m = MeineKlasse("Argument")
m.summe(1,2) # return 1+2
```