Python-Crashkurs

Patricia Gerbig

WISE 2025/26

Danke an Conrad Schweiker für das Korrekturlesen!

Contents

Für diesen Crashkurs verwenden wir Python 3, Numpy, Matplotlib und Scipy sowie den Texteditor Visual Studio Code und ein Terminal.
Der erste Teil ist für die Personen, die noch keine Programmiererfahrungen haben. Wir erklären euch in diesem Teil, wie man wie ein Programmierer denkt und unabhängig von der Sprache Probleme löst. Das heißt, es werden erst Konzepte erarbeitet, die dann in Code umgewandelt werden. Die Beispiele, die hier gezeigt werden, werden zwar in Python sein, aber ihr seid dazu aufgerufen, euch erst mal weniger Gedanken über die Syntax zu machen und eher die Konzepte dahinter zu verstehen. Dazu werden wir erst mal die Grundbegriffe erklären.

0 Hinweise zur Installation von Python

Es gibt viele Wege Python zu installieren, wenn es überhaupt notwendig ist. Denn auf vielen Betriebssystemen ist Python, sogar Python3, bereits vorinstalliert.
Der Grund wieso es trotzdem ein Thema ist, ist wegen der Version von Python3. Denn mit den Versionen ändert sich zum Beispiel mal die Syntax oder die Art und Weise wie eine Funktion funktioniert. Dies kann dazu führen, dass auf zwei Geräten der gleiche Code unterschiedliche Ergebnisse hervorbringen kann. Und wenn man an Abgaben von Übungszetteln denkt, ist das nicht gerade vorteilhaft.
Deshalb macht es Sinn, immer die neuste Version von Python3 installiert zu haben. Vorinstallierte Versionen hängen jedoch gerne einige Monate bis Jahre zurück. Deshalb sollte man natürlich darauf achten die neuste Version zu installieren. Aber was heute das neuste ist kann schnell veraltet sein. Der zweite Punkt, der bei der Installation berüchsichtigt werden sollte ist deshalb, dass es eine einfache und schnelle Möglichkeit gibt Updates zu installieren. Und das im Idealfall nicht nur für Python selbst, sondern auch für alle Libraries (quasi Erweiterungen von Python), die wir nutzen werden.

0.1 Was wird benötigt für diesen Crashkurs?

Für diesen Crashkurs wird folgendes benötigt:

- Python3 (idealerweise die neuste Version)
- ein Texteditor (z.B. Visual Studio Code)
- für Numerik relevante Python Libraries:
  - NumPy
  - Matplotlib
  - Scipy

0.1.1 Hinweis zur Wahl des Texteditors

Unter einigen Programmierern ist die Wahl des Texteditors eine sehr leidenschaftliche und identitätsstiftende Entscheidung, weshalb es viele Artikel gibt, die die verschiedenen Optionen vergleichen. Hier wollen wir es aber simple und pragmatisch halten.
Während des Python Crashkurses werde ich alles in **Visual Studio Code (VSCode)** oder **VSCodium** demonstrieren.

Wieso VSCode/VSCodium?

- Es ist nur sehr wenig Einrichtungsaufwand nötig und die Einrichtung ist, im Vergleich zu anderen Texteditoren, sehr intuitiv. Zum Beispiel können Extensions wie in App Stores installiert werden und die Einstellungen können graphisch angepasst werden.
- Der Texteditor ist gut rein *graphisch* verwendbar. Man muss sich also keine Keyboard Shortcuts und Befehle angucken, um ihn verwenden zu können.
- Er ist auf allen drei gängigen Betriebssystemen (Linux, MacOS und Windows) verfügbar.
- Erweiterungen und Individualisierungen sind einfach möglich. Es ist also ein Texteditor, der bis zu einem bestimmten Punkt mit euren Ansprüchen mitwächst.

Was ist VSCode bzw. VSCodium und wo liegt der Unterschied?

Visual Studio Code ist ein Texteditor von Microsoft. Der Source Code, in dem dieser programmiert ist, ist allerdings Open Source. Das bedeutet (um mal ganz stumpf Wikipedia zu zitieren), dass der Source Code öffentlich ist und von dritten eingesehen, geändert, genutzt und im Anschluss auch verteilt werden kann.
Wenn man sich VSCode dann von Microsoft runterlädt, wird dies allerdings lizensiert, sodass die Open Source *Eigenschaften* verloren gehen.
Welche Vor- und Nachteile Open Source bietet, soll gar nicht Thema dieses Crashkurses sein. Falls ihr jedoch Open Source Software bevorzugt, stellt VSCodium eine gleichwertige alternative zu VSCode dar.

Es steht euch natürlich frei für diesen Crashkurs jeden Texteditor zu verwenden, den ihr wollt. Falls ihr nur noch keine persönliche Präferenz habt, zum Beispiel weil ihr das erste Mal programmiert, ist es empfehlenswert erstmal mit einem der beiden zu starten.

0.2 Installationsanleitung

Vor dem Start der Installation sollten, unabhängig vom Betriebssystem, man einige Punkte berücksichtigen:

- Es sollte **keine andere Python Variante installiert** sein, die nicht vorinstalliert ist oder die über den gleichen Package-Manager verwaltet wird, wie den, der verwendet wird, um die neue Version zu installieren.
- Alle Software-Updates sollten vorher installiert werden (Damit nicht allzu lange zu warten, ist sowieso immer eine gute Idee).
- Es sollte eine stabile Internetverbindung bestehen (wenn ihr in der Uni seid, richtet euch zuerst **eduroam** ein anstatt _Free_Wifi-Berlin oder eurem Hotspot zu nutzen).

0.3 Linux

Auf Linux ist eine vernünfitge Installation glücklicherweise simpel, da Programme zur Paketverwaltung, sog. **Package-Manager** in den verschiedenen Linux-Versionen (Distributionen) bereits installiert sind. Mit deren Hilfe ist es sehr einfach, alles benötigte zu installieren.
Außerdem ist meistens bereits eine relativ aktuelle Version installiert (oder wird es sein, sobald ihr euer System vollständig aktualisiert habt).
Da die Package-Manager der unteschiedlichen Distributionen unterschiedlich heißen und verschiedene Befehle verwenden, gehe ich hier auf die gängigsten Distributionen ein, nämlich solche, die auf Debian basieren — dazu gehören u.a. Ubuntu, Linux Mint und natürlich Debian selbst — sowie Fedora.

Bevor du anfängst, dein System, bzw. Python zu aktualisieren, kannst du zunächst einmal schauen, ob du bereits die aktuellste Python-Version installiert hast. Dazu verwendest du unter allen Distributionen den folgenden Befehl:

```
python3 --version
```

Welche die neueste Version ist, kannst du hier herausfinden (aber bitte lade dir die dort nicht herunter). Falls du bereits die neuste Version hast, kannst du zu Schritt 3 deiner jeweiligen Distribution springen.

0.3.1 Linux Mint, Ubuntu, Debian

Diese Distributionen verwenden das **Apt** Paketmanagementsystem.

1. Führe ein Systemupgrade durch. Dazu führst du zunächst den folgenden Befehl aus:

   ```
   sudo apt update
   ```

   Danach führst du dann diesen Befehl aus:

   ```
   sudo apt upgrade
   ```

   dabei wirst du gefragt, ob du das Upgrade fortsetzen willst:

   ```
   Do you want to continue? [Y/n]
   ```

   Das bestätigst du dann, indem du `Y` tippst und mit der Enter-Taste bestätigst.
   Starte im Anschluss dein Gerät neu. Jetzt solltest du die neuste Python-Version haben.

2. Falls du nach dem Systemupgrade noch nicht die neuste Python-Version hast, kannst du es noch explizit nachträglich installieren. Dazu führst du den folgenden Befehl aus:

   ```
   sudo apt install python3
   ```

   Und bestätigst wieder mit `Y` und der Enter-Taste.
   Dies installiert automatisch die neuste Version, welche mit künftigen Systemupgrades automatisch aktualisiert wird.

3. Jetzt kannst du die benötigten Python Bibliotheken installieren.
   Dazu führst du den folgenden Befehl aus:

   ```
   sudo apt install python3-numpy python3-matplotlib python3-scipy
   ```

   Auch hier bestätigst du wieder die Installation.

4. Als letzten Schritt installierst du jetzt VS Codium.
   Dafür gibt es eine kürzere Variante, die ohne Weiteres erstmal **nur unter Ubuntu** funktioniert, da dort der **Snap** Paketmanager vorinstalliert ist:

   ```
   snap install codium --classic
   ```

   Unter **allen anderen genannten Distributionen** funktioniert die folgende Variante aber auch, ohne dass Snap installiert ist. Diese Variante wirkt zwar komplizierter, ist es aber nicht wirklich und ermöglicht dir zudem, dass du VS Codium wie gewohnt installieren und aktualisieren kannst.
   Dazu führst du zunächst den folgenden Befehl aus:

   ```
   wget -qO - https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/raw/master/pub.gpg \
       | gpg --dearmor \
       | sudo dd of=/usr/share/keyrings/vscodium-archive-keyring.gpg
   ```

   Darauf folgend führst du dann diesen Befehl aus:

   ```
   echo -e 'Types: deb\nURIs: https://download.vscodium.com/debs\nSuites: vscodium\nComponents
       : main\nArchitectures: amd64 arm64\nSigned-by: /usr/share/keyrings/vscodium-archive-
       keyring.gpg' \
   | sudo tee /etc/apt/sources.list.d/vscodium.sources
   ```

   Diese beiden Befehle haben nun VS Codium zu der Liste der installierbaren Pakete hinzugefügt, sodass du es nun wie gewohnt installieren kannst:

   ```
   sudo apt install codium
   ```

0.3.2 Fedora

Diese Distribution verwendet das **DNF** Paketmanagementsystem.

1. Führe ein Systemupgrade durch. Dazu führst du zunächst den folgenden Befehl aus:

```
sudo dnf upgrade --refresh
```

   Starte im Anschluss dein Gerät neu. Jetzt solltest du die neuste Python-Version haben.

2. Falls du nach dem Systemupgrade noch nicht die neuste Python-Version hast, kannst du es noch explizit nachträglich installieren. Dazu führst du den folgenden Befehl aus:

```
sudo dnf install python3
```

   Dies installiert automatisch die neuste Version, welche mit künftigen Systemupgrades automatisch aktualisiert wird.

3. Jetzt kannst du die benötigten Python Bibliotheken installieren.
   Dazu führst du den folgenden Befehl aus:

```
sudo dnf install python3-numpy python3-matplotlib python3-scipy
```

4. Als letzten Schritt installierst du jetzt VS Codium.
   Die folgende Variante wirkt zwar kompliziert, ist es aber nicht wirklich und und ermöglicht dir zudem, dass du VS Codium wie gewohnt installieren und aktualisieren kannst.
   Dazu führst du zunächst den folgenden Befehl aus:

```
sudo rpmkeys --import https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/-/raw/master/pub
    .gpg
```

   Darauf folgend führst du dann diesen Befehl aus:

```
printf "[gitlab.com_paulcarroty_vscodium_repo]\nname=download.vscodium.com\nbaseurl=https
    ://download.vscodium.com/rpms/\nenabled=1\ngpgcheck=1\nrepo_gpgcheck=1\ngpgkey=https
    ://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/-/raw/master/pub.gpg\nmetadata_expire
    =1h\n" | sudo tee -a /etc/yum.repos.d/vscodium.repo
```

   Diese beiden Befehle haben nun VS Codium zu der Liste der installierbaren Pakete hinzugefügt, sodass du es nun wie gewohnt installieren kannst:

```
sudo dnf install codium
```

0.4 Windows

Windows Installation

0.5 MacOS

Unter MacOS haben wir 2 Möglichkeiten:

1. Man kann sich mit dem Package-Manager **Homebrew** alle benötigten Packages installieren.
2. Man kann **Anaconda** nutzen, um die neuste Python Version und Package-Manager zu installieren.

Beide Varianten haben den Vorteil, dass man die seine Pakete installieren und *sauber* updaten, sowie alte, ungenutzte Pakete einfach entfernen kann. Bei anderen Wegen der Installation ist dies leider häufig nicht gegeben.
Keiner der beiden Optionen ist generell besser oder schlechter. Es hängt vor allem von euch ab, was ihr benötigt. Hier ein kleiner Vergleich:

	Pro	Contra
Homebrew	• Man kann mehr als nur Python darüber verwalten: z.B. auch VS-Codium. • Falls man sich doch umentscheidet, ist Homebrew deutlich einfacher vollständig zu deinstallieren, im Vergleich zu Anaconda.	• Falls man irgendwann mal mehr oder spezielle Python-Packages brauchst, ist die Auswahl leider nur auf die wichtigsten beschränkt. (Man kann mit Pip und Virtual Environments arbeiten, wenn ein Paket nicht auf Homebrew verfügbar ist, aber ein bisschen einarbeiten muss man sich dafür schon.)
Anaconda	• Es gibt eine riesige Auswahl an Packages für Python. • Falls man bestimmte Versionen von einem Paket brauchst, ist die Verwaltung davon ziemlich easy.	• Man muss sich etwas einarbeiten, damit alles funktioniert. • Eine vollständige Deinstallation ist nicht ganz so intuitiv.

Table 1: Vergleich: Homebrew und Anaconda

Wenn ihr Programmieranfänger seit, empfehle ich aber prinzipiell eher Homebrew.

0.5.1 Installation mit Homebrew

1. Das Gerät auf Software-Updates prüfen und diese ggf. installieren. (Man findet diese unter *Einstellungen* → *Allgemeines*)

2. Als Nächstes wollen wir überprüfen, welches Python standardmäßig genutzt wird. Dies können wir mit folgendem Befehl überprüfen:

   ```
   % which python3
   ```

   Die Ausgabe sollte */usr/bin/python3* sein. (Falls nein, bittet um Hilfe bei der Installation)

3. Als nächstes installieren wir Homebrew.
   Dies machen wir, in dem unser Terminal öffnen (z.B. mit *Command* + *Space* und dann *Terminal* eingeben und *Enter* drücken). Dann öffnen wir unseren Browser und rufen brew.sh auf.
   Den darauf angezeigten Befehl (s. Figure 1) kopieren wir und fügen ihn in unser Terminal ein und klicken anschließend Enter.
   Jetzt erscheint eine Zeile, wo man sein Passwort für das Gerät eingeben muss. Dies muss man "blind" eingegeben, im Sinne von man sieht nicht was man eingibt und auch keine Platzhalter dafür.

   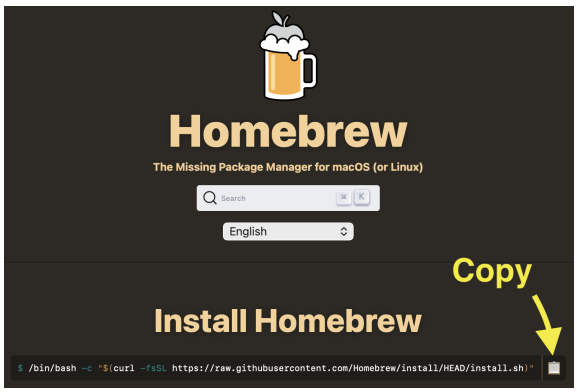

   Figure 1: Homebrew Startseite mit Installationsbefehl

   Die Installation von Homebrew kann jetzt einige Zeit dauern, da ggf. die Command Line Tools noch installiert werden müssen. Falls diese noch nicht installiert sind, bekommt ihr im Terminal eine Benachrichtigung, in der ihr gefragt werden, ob ihr mit der Installation einverstanden seid. Nach der Bestätigung wird die Homebrew-Installation fortgesetzt.

4. Sobald die Installation fertig ist, können wir ohne den letzten Schritt Homebrew noch nicht verwenden. Probiere dies gerne aus, in dem du brew help in dein Terminal eingibst. Du solltest dann eine Fehlermeldung bekommen, dass der Befehl noch nicht gefunden sei.
   Damit der Befehl gefunden wird, kannst du folgenden Befehl in dein Terminal ausführen:

```
% (echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> /Users/deinnutzername/.
    zprofile
```

deinnutzername sollte dabei entsprechend angepasst werden. Allerdings wird dir am Ende der Installation von Homebrew im Terminal auch der angepasste Befehl angezeigt, so dass du ihn einfach herauskopieren kannst.

Starte jetzt einmal dein Gerät neu und gebe erneut in dein Terminal *brew help* ein. Jetzt sollte keine Fehlermeldung mehr auftauchen und du solltest die wichtigsten Befehle angezeigt bekommen, wie in dem Terminal rechts.

```
% brew help
Example usage:
  brew search TEXT|/REGEX/
  brew info [FORMULA|CASK...]
  brew install FORMULA|CASK...
  brew update
  brew upgrade [FORMULA|CASK...]
  brew uninstall FORMULA|CASK...
  brew list [FORMULA|CASK...]

Troubleshooting:
  brew config
  brew doctor
  brew install --verbose --debug FORMULA|CASK

Contributing:
  brew create URL [--no-fetch]
  brew edit [FORMULA|CASK...]

Further help:
  brew commands
  brew help [COMMAND]
  man brew
  https://docs.brew.sh
```

Falls dies nicht der Fall ist, überprüfe zunächst, ob deine *.zprofile* Datei korrekt aussieht, indem du folgenden Befehl ausführt, um sie zu öffnen:

```
open -a TextEdit .zprofile
```

In dieser Datei solltest du folgendes angezeigt bekommen:

```
1 eval "$(/opt/homebrew/bin/brew shellenv)"
```

Wenn dies nicht der Fall ist, kopiere die Zeile in die Datei und speichere diese und starte dein Gerät erneut.
Probiere nun erneut, ob du mit dem Befehl *brew help* keine Fehlermeldung und die richtige Ausgabe bekommst.

5. Jetzt startet die eigentliche Installation und danke Homebrew geht die echt schnell.
   Kopiere einfach folgenden Befehl (zur Installation der neusten Python-Version, sowie den benötigten Libraries) in dein Terminal:

   ```
   % brew install python3 numpy scipy python-matplotlib
   ```

   Wenn du Zeit sparen möchtest, kannst du ein weiteres Terminal-Fenster öffnen (mit Rechtsclick auf das Terminal-Symbol im Dock und dann *New Window* (bzw. Neues Fenster auswählen). Du kannst aber auch warten, bis der vorherige Befehl vollständig ausgeführt wurde. Füge nun folgenden Befehl (zur Installation von VSCodium) in dein Terminal:

   ```
   % brew install --cask vscodium
   ```

6. Als letztes testen wir, ob die Installation erfolgreich war. Dafür geben wir erneut *which python3* ins unser Terminal ein.
   Die Ausgabe sollte jetzt folgendermaßen aussehen:

   ```
   % which python3
   /opt/homebrew/bin/python3
   ```

0.5.2 Installation mit Anaconda

Anaconda Installation hinzufügen?

0.6 VSCodium einrichten

1 Grundbegriffe der Programmierung

1.1 Funktion, Parameter & Rückgabewert

Eine **Funktion** ist eine Zusammenfassung von Anweisungen. Diese Funktion ist beliebig oft aufrufbar. Sie hat einen Namen, einen Parameter (auch Übergabewert genannt) und einen Rückgabewert. Diese können aber auch leer sein.
Betrachten wir als erstes Beispiel eine mathematische Funktion: $f : \mathbb{R} \to \mathbb{R}$; $f(x) = x^2$. Dann lässt sich diese so als Code schreiben:

```
def f(x):
    return x**2
```

Dabei ist x der Parameter und $x**2$, was die Python-Schreibweise für x^2 ist, der Rückgabewert unserer Funktion.
Ein anderes Beispiel wäre eine Funktion, die "Hello World" ausgibt:

```
def helloWorld():
    print("Hello World")
    return
helloWorld() #so wird die Funktion aufgerufen
```

Wenn man diese Funktion aufruft, würde in der Konsole ein "Hello World" erscheinen. Die Parameter und der Rückgabewert der Funktion sind allerdings leer.
Die Raute leitet dabei einen Einzeilen Kommentar ein. Das heißt, alles was hinter dieser Raute in der Zeile steht wird nicht ausgeführt.
Eine Funktion kann natürlich auch mehrere Parameter und Rückgabewerte haben. Die Funktion $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$; $g(x, y) = x \cdot y$ würde sich so schreiben lassen:

```
def g(x,y):
    return x*y
```

1.2 Variable & Datentypen

Eine Variable hat einen Namen, einen Datentyp und ggf. einen Startwert. Wir können erst mal folgende Datentypen unterscheiden:

Boolean: Dies ist ein Datentyp, der genau zwei verschiedene Werte haben kann: True oder False. Demnach hier eine vollständige Liste an Beispielen:

```
bool1 = True
bool2 = False
```

Integer: Ein Integer ist ein ganzzahliger Wert, ungenau ausgedrückt die ganzen Zahlen $\mathbb{Z}$. Der Vergleich hinkt ein bisschen, da die Darstellung ab einer bestimmten Größe schwierig wird. Genau genommen ist ein Integer eine ganze Zahl in dem Intervall $[-2^{63}, 2^{63} - 1]$, wenn man standardmäßig von einem 64-bit Integer ausgeht. (Was das ist und wieso das so ist, werdet ihr in CompMath I lernen.) Für den Fall, dass doch größere Zahlen benötigt werden, gibt es allerdings noch Datentypen wie long Integer, in dem mehr Stellen gespeichert werden können.
Beispiele sind für Integer sind also:

```
int1 = 1
int2 = -50
int3 = 1397
```

Float: Variablen vom Datentyp Float sind Kommazahlen, oder eher Gleitkommazahlen, d.h. grob gesagt eine annähernde Darstellung einer reellen Zahl. Genaueres dazu erfahrt ihr auch in CompMath I.
Beispiele für Floats sind dann:

```
float1 = 67.2
```

```
float2 = 3.14159265359
float2 = 2.0
float3 = 1.72e+3 #=1720.0
float4 = 7e-4 #=0.0007
```

Standard Floats können gespeichert werden in einem Bereich von $\pm 2.2250738585072014e-308$ bis $\pm 1.7976931348623157e+308$, wobei Bereich eigentlich irreführend ist, da wir bei Zahlen mit zu vielen Nachkommastellen die Zahl eigentlich nur approximieren können. Dabei wird die Gleitkommadarstellung verwendet. Wie genau diese funktioniert, erfahrt ihr in CompMath I.

String: Ein String ist eine Zeichenkette.
In Python wird diese in Anführungszeichen gesetzt, also:

```
string1 = "Hello World"
string2 = 'Hello World'
```

List: Eine Liste ist eine geordnete Sammlung von Elementen.

```
list1 = [1,2,3,4]
list2 = [3.7, 2.9, 5.3]
list3 = [True, False]
list4 = [True, 1, 3.4, 'Hallo']
```

Wie man in den Beispielen sieht, kann eine Liste (in Python) verschiedene Datentypen enthalten. Dazu dürfen sich die Elemente doppeln, sind veränderbar, es kann also der Wert einzelner Elemente geändert werden und es können welche hinzugefügt und gelöscht werden. Außerdem hat jedes Element einen Index. Dieser beginnt bei 0.
Eine besonderheit in Python ist, dass Strings quasi Listen sind, im Sinne von, dass man auf ihnen die gleichen Operatoren ausführen kann. Mehr dazu bei den Listenoperatoren.

Es gibt natürlich noch viele weitere Datentypen in Python. Im dritten Teil werdet ihr auch noch die Numpy-Arrays kennen lernen. Die je nachdem wir man sie initialisiert, d.h. jenachdem welchen "Wert" bzw. eher Form man ihnen zuweist, eher mit einem Vektor oder einer Matrix vergleichbar sind.
Das sind aber schon so gut wie alle vorerst relevanten Datentypen, die ihr in den nächsten Semestern brauchen werdet. Irgendwann könntet ihr noch den Datentypen Dictionary oder komplex begegnen. Ersteres ist eine geordnete Sammlung von Datenwerten in Paaren und zweiteres ein Datentyp um komplexe Zahlen darzustellen.
Es ist vielleicht gut die Existenz dieser Datentypen im Hinterkopf zu behalten.
In vielen Programmiersprachen muss man den Datentyp einer Variable mit angeben. Dies ist in Python nicht der Fall.

1.3 Operatoren

Operatoren sind eigentlich auch nur Funktionen, die manchmal eine 'ungewöhnliche' Notation haben. Zunächst lassen sich Operatoren in zwei Kategorien unterteilen:

- **Unäre Operatoren** haben nur einen Operanden. Ein typisches Beispiel dafür ist die Negation.
- **Binäre Operatoren** haben zwei Operanden. Ein Beispiel dafür wäre die Addition, Subtraktion oder Multiplikation von zwei Zahlen.

Zusätzlich kann man die Operatorenschreibweise auch noch unterteilen:

- Bei **Präfix**-Operatoren wird der Operator vor den Operanden geschrieben. Ein Beispiel dafür die Negation $\neg x$.
- Bei **Infix**-Operatoren wird der Operator in die Mitte der beiden Operanden geschrieben. Beispiele dafür sind die Addition $x + y$, oder das logische und $x \wedge y$.
- **Postfix**-Operatoren schreiben den Operator hinter den Operanden. Ein Beispiel, was wir gleich noch genauer kennen lernen werden, ist das aufrufen des i-ten Elements einer liste listenname[i].

Wir können die Operatoren aber noch weiter unterteilen und zwar in **Vergleichsoperatoren**, **logische Operatoren**, **arithmetische Operatoren** und **Listenoperatoren**, wobei die Namen wahrscheinlich selbsterklärend sind.
Folgende Operatoren gibt es in Python:

	unär			binär		
Vergleichsoperatoren				==	Gleichheit	Boolean, Integer, Float, List, String
				!=	Ungleichheit	Boolean, Integer, Float, List, String
				<	kleiner als	Integer, Float, List, String
				>	größer als	Integer, Float, List, String
				<=	kleiner gleich	Integer, Float, List, String
				>=	größer gleich	Integer, Float, List, String
logische Operatoren	not	Negation	Boolean	or	oder	Boolean
				and	und	Boolean
arithmetische Operatoren	+	ändert nichts	Integer, Float	+	Addition	Integer, Float
	-	Kehrt das Vorzeichen um	Integer, Float	-	Subtraktion	Integer, Float
				*	Multiplikation	Integer, Float
				**	Power	Integer, Float
				//	ganzzahlige Division	Integer, Float
				%	Modulo	Integer
				/	Division	Float
Listenoperatoren	[i]	i-te Stelle	List, String	+	Verkettung	List, String
				*	Wiederholung	List, String
				[i:j]	Teilliste i bis j-te Stelle	List, String
				in	Test auf Enthalten-sein	List, String

Table 2: Übersicht der wichtigsten Operatoren

Für viele dieser Operatoren mag es sehr intuitiv sein, wie diese funktionieren, z. B.:

```
int1 = 1 + 3
int2 = 5 // 2 # = 2
float1 = 5 / 2 # = 2.5
bool1 = 4 != 3 # = True
bool2 = 52 < 5 # = False
```

Etwas unintuitiver kann es am Anfang bei den Listenoperatoren sein. Wichtig dabei ist, zu beachten, dass wir anfangen, bei Null zu zählen.
Und auch, wenn es erst mal etwas komisch wirkt, dass die Listenoperatoren sich auch auf Strings anwenden lassen, hilft es, sich einfach vorzustellen, dass Strings auch nur eine Liste von einzelnen Zeichen sind.

```
list1 = [True, 1, 3.4, 'Hallo']
list1[2] #3.4
String1 = 'Hello'
String2 = 'World'
String3 = String1 + String2 # = 'Hello World'
String3[4] # = 'l'
String4 = 3*String1 # = 'HelloHelloHello'
```

1.3.1 Indizierung einer Liste

Ein besonders wichtiger Operator, ist das Zugreifen auf bestimmte Elemente einer Liste.
Wichtig dabei ist, dass man immer daran denkt, dass in Python eine Liste **0-Indiziert** ist, d.h. wir fangen an bei 0 zu zählen.
Beispiel:

```
my_list = [1,3,5,7,9]
my_list[0] #1
my_list[1] #3
```

Man kann Listen auch 'slicen', also Teile davon ausgeben.
Der Slice-Operator ist folgendermaßen aufgebaut: **listenname[start:stop:step]**
Alle dieser Parameter sind Integer die alle **optional** sind. D.h. wenn man sie nicht angibt, gibt es einen Defaultwert der angenommen wird.
Bei start ist es 0, stop ist es die länger des Liste und step hat den Defaultwert 1.
Dieser Operator wird spätestens sehr wichtig, wenn man anfängt mit Matrizen zu arbeiten und kann uns jetzt auch schon dabei helfen zu verstehen, wie Listen in Python indiziert werden:

```
my_list = [1,3,5,7,9]
my_list[1:3] #[3,5]
my_list[1:] #[3,5,7,9]
my_list[0:5] #[1,3,5,7,9]
my_list[5] #IndexError: list index out of range
```

Die Art und Weise welche Elemente der Liste zurückgegeben werden scheint manchmal reicht unintuitiv zu sein: Wieso wird wenn man das 1-te bis 3-te Element haben möchte nur das 1-te und 2-te ausgegeben? Und eigentlich ist es ja auch schon komplett unintuitiv, dass 2-te Element das 1-te zu nennen.

Wann welche Elemente einer Liste zurückgegeben werden, kann man sich gut veranschaulichen, in dem man statt die Elemente zu zählen die Kommas zählt (s.h. Bild): Also, dass my_list[1] das Element nach dem ersten Komma zurückgibt, also die 3. Demnach gibt my_list[1:3] die Elemente zwischen dem 1-ten und 3-ten Komma zurück. my_list[0:5] gibt alle Elemente zwischen dem 0-ten und 5-ten Komma zurück, also die komplette Liste.

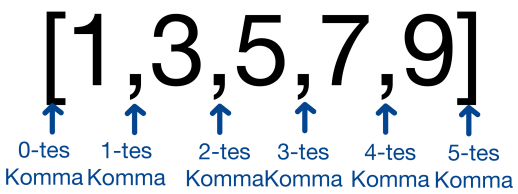

Figure 2: Veranschaulichung Indizes einer Liste

my_list[5] dagegen sollte das Element nach dem 5-ten Komma zurück geben. Nach dem 5-ten Komma gibt es aber kein weiteres Element mehr, weshalb man in dem Fall eine Fehlermeldung bekommt.

1.3.2 Modulo

Die Division mit Rest bzw. Modulorechnung kennen einige bestimmt schon. Aber da man es beim Programmieren häufig benötigt und ihr auch im Theorieteil von CoMa I davon nicht verschont bleibt, zum Beispiel beim Umrechnen von Zahlen in verschiedene Zahlensysteme, sollen hier noch mal alle auf den gleichen Stand gebracht werden:
Modulo ist, wie oben in der Tabelle schon steht, eine arithmetische Operation. Diese gibt uns den Rest der ganzzahligen Division zurück. Hier ein paar Beispiele:

$$10 \bmod 4 = 2, \text{ da } 10//4 = 2(\text{Rest } 2)$$
$$53 \bmod 10 = 3, \text{ da } 53//10 = 5(\text{Rest } 3)$$
$$24 \bmod 2 = 0, \text{ da } 24//2 = 12(\text{Rest } 0)$$

Auch wenn es oben in der Tabelle anders steht, lässt sich in Python der Modulooperator auch mit Float-Zahlen verwenden. Dass dies möglich ist, wurde bewusst weggelassen, denn auch wenn es theoretisch gesehen erst mal gar nicht so sinnlos klingen mag, gibt es, wenn man mit Computern arbeitet, aber einen wichtigen Aspekt, der die Modulorechnung für Floats unbrauchbar machen kann: Rundungsfehler.

Nehmen wir uns eine reelle Zahl, z. B. π, dann sollte $3 \cdot \pi \bmod \pi = 0$ sein.
Wir wollen testen, ob eine Zahl ein Vielfaches von π ist, und definieren uns eine Variable mit den ersten zehn Nachkommastellen. Überprüfen wollen wir das mit der Approximation von π, die uns Python zur Verfügung stellt.

```
import math
mypi = 3.1415926536
print(100*mypi % math.pi)
```

Die Ausgabe ist dann nicht 0:

```
1.0206946399193 84e-09
```

Ihr werdet in CoMa noch sehr viel über Rundungsfehler anderer arithmetischer Operationen erfahren, weswegen wir das auch nicht vorweg nehmen werden. Nur vor allem bei der Modulorechnung ist es wichtig, dies von Anfang an im Hinterkopf zu behalten.
In den nächsten beiden Kapiteln werden wir uns mit Kontrollstrukturen beschäftigen. Kontrollstrukturen sind dazu da, um den Ablauf eines Programmes zu steuern. Man kann dabei zwei Arten von Kontrollstrukturen unterscheiden: Fallunterscheidungen und Schleifen.

1.4 Fallunterscheidung

Wozu Fallunterscheidungen grob da sind, lässt sich, glaube ich, am Namen erkennen. Für simple Unterscheidungen kann man einfach mit **If-else-Anweisungen** arbeiten:

- **if**: Eine if-Anweisung besteht aus einer Bedingung und einem Codeblock mit Anweisungen, der, wenn diese Bedingung wahr ist, ausgeführt wird.
- **else**: Eine else-Anweisung wird ausgeführt, wenn keine andere Bedingung wahr ist.
- **elif**: Kurzform für else if. Dies ist ein else, was erneut an eine Bedingung geknüpft ist, bevor der zugehörige Code-Block ausgeführt wird. Dadurch lassen sich Verschachtelungen vermeiden.

Um mal ein mathematisches Beispiel zu geben:
Wir haben die Fallunterscheidung $f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & \text{sonst} \end{cases}$
Diese lässt sich dann in Python so ausdrücken durch

```
def f(x):
    if x >= 0:
        return 1
    else:
        return 0
```

Speziell in Python kommen dabei noch die conditional expressions hinzu, welche im zweiten Teil des Crashkurses erklärt werden. Es gibt zwar noch weitere Formen der Fallunterscheidung, diese werden aber erst mal für euch nicht weiter relevant sein.

1.5 Schleifen

Eine Schleife besteht genauso wie eine if-Anweisung aus einer Bedingung und einem Code-Block. Der große Unterschied besteht jedoch darin, dass dieser Code-Block so lange ausgeführt wird, bis eine Abbruchbedingung eintritt.
In Python unterscheiden wir zwischen zwei Arten von Schleifen: der For-Schleife und der While-Schleife.

1.5.1 For-Schleife

Die allgemeine Syntax der For-Schleife lässt sich folgendermaßen beschreiben:

```
for Variable in iterierbares-Objekt:
    ... Code-Block ...
```

Wenn wir zum Beispiel einmal einzeln alle Elemente einer Liste printen wollen, können wir das so machen:

```
list = [1,2.5,'Hallo',False]
for x in list:
    print x
```

In CoMa, aber auch in der Programmierung im Allgemeinen, wird man For-Schleifen häufig benutzen, um über einen bestimmten Zahlenabschnitt zu iterieren.
Das könnt ihr, ohne in diesem Moment zu sehr ins Detail zu gehen, mit der range()-Funktion:

```
for i in range(1,9):
    print i
```

Die Ausgabe ist dann:

```
1
2
3
4
5
6
7
8
```

Wie die Funktion genau funktioniert, erfährt ihr im zweiten Part.

1.5.2 While-Schleife

Kürze do While Teil

While-Schleifen funktionieren so, dass sie ihre Anweisungen bzw. den Code-Block so lange ausführen, wie die Bedingung, an die sie geknüpft sind, wahr ist. Das so zu sagen, klingt für jeden Programmierer erst mal richtig, ist, wenn man es ganz genau nehmen will, aber vielleicht doch etwas zu leicht ausgedrückt. Denn wenn man das so formuliert, würde das heißen, dass die Schleife sofort abbricht, sobald die Bedingung nicht mehr wahr ist. Es kommt aber auf den Zeitpunkt der Überprüfung an.
Dabei lassen sich While-Schleifen in zwei Kategorien unterteilen: While-do-Schleifen und Do-while-Schleifen. In Python gibt es aber nur erstere.
Hier ein Beispiel für eine Schleife. Das Programm gibt solange "Hallo" aus, bis x nicht mehr kleiner 5 ist.

```
x = 0
while x < 5:
    print("Hallo")
    x = x + 1
```

Der größte Unterschied zwischen einer For- und einer While-Schleife für den Computer ist, dass die Anzahl der Durchläufe einer for-Schleife bereits vorher bekannt ist, bei einer While-Schleife allerdings nicht.
Bei einer For-Schleife wird entweder über eine Liste oder ein listenähnliches Objekt (wie die range()-Funktion) iteriert. Die Länge dieser steht beim Aufruf der Schleife fest und kann nicht mehr so geändert werden, dass dies von der For-Schleife verwendet werden kann. Damit ist gemeint, man kann die Liste zwar nach dem Start der For-Schleife ändern, aber diese Änderungen werden nicht mehr auf diese For-Schleife übertragen werden.
Bei einer While-Schleife hingegen, kann der 'flow' beliebig ablaufen. Deshalb sollte man immer darauf achten, dass ab einen bestimmten Punkt die Bedinung nicht mehr erfüllt ist, da man sonst eine unendlich-Schleife startet. (Falls ihr das jetzt ausprobieren wollt, in der Regel kann man diese mit Strg bzw. Cmd + C wieder abbrechen)
Man kann eine Schleife übrigens nicht nur mit der Nichterfüllung der Schleifenbedingung oder dem Ende der Liste unterbrechen: Es gibt Statements in Python, mit denen ihr den Ablauf weiter beeinflussen könnt, auf welche wir später noch eingehen.

1.6 Input & Output eines Programmes

Der Input und Output eines Programmes sind, wie die Namen schon sagen, die Eingabe, welche wir dem Programm übergeben, und die Ausgabe, die uns das Programm zurückgibt. Bisher war die Ausgabe, die

wir hatten, immer in Textform im Terminal mit der Funktion print() erzeugt. Das ist aber bei Weitem nicht die einzige Form eines Outputs, die wir haben können. Man kann z. B. den Text auch in eine Datei schreiben, Plots zum Beispiel in Form von Bildern zurückgeben und noch vieles mehr. Wie das funktioniert, werdet ihr auch im letzten Kapitel zu Matplotlib lernen.
Damit ihr aber auch wisst, wie ihr Dinge in eurem Programm eingeben könnt, möchten wir euch jetzt die **input()-Funktion** zeigen.
Diese Funktion bekommt einen Parameter **prompt** übergeben, welcher der String ist, der vor der Eingabe angezeigt werden soll.
Um die Eingabe abzuspeichern, müssen wir diese als Variable speichern:

```
name = input('Wie lautet dein Name: ')
print('Hallo ' + name)
```

Die Ausgabe sieht dann so aus:

```
Wie lautet dein Name: Bob
Hallo Bob
```

Die Eingabe wird immer als String gespeichert.
Es gibt noch andere Möglichkeiten, wie ihr eurem Programm Argumente übergeben könnt, und zwar mit **Command Line Arguments**.
Da das etwas verwirrend sein kann am Anfang und ihr das erst mal auch gar nicht in CoMa I zwingend braucht, findet ihr dazu was im Anhang.

1.7 Iterative und rekursive Programmierung

Zum Abschluss von diesem Part werden wir uns Rekursion angucken. Rekursion ist im Gegensatz zu dem, was wir uns bisher angeguckt haben, kein 'Bestandteil' einer Programmiersprache, sondern eine Art und Weise, zu programmieren.
Bislang waren alle Beispiele, die wir uns angeguckt haben, iterativ programmiert. Doch was genau heißt das überhaupt? **Iteration** ist das lateinische Wort für Wiederholung. Im Bezug auf die Programmierung heißt das, dass eine mehrfache Ausführung von Funktionen stattfindet oder es mehrere Anweisungen gibt. Dies wird dann durch Schleifen realisiert und mithilfe einer Abbruchbedingung beendet.
Rekursion ist das lateinische Wort für zurücklaufen. Man spricht von Rekursion, wenn sich eine Funktion immer wieder selbst aufruft, bis eine Abbruchbedingung eintritt.
Zur Veranschaulichung nehmen wir die Fakultät $n!$:

```
def fak_iterativ(n):
    res = 1
    for i in range(1,n+1):
        res = res*i
    return res

def fak_rekursiv(n):
    if n == 0:
        return 1
    else:
        return n*fak_rekursiv(n-1)
```

Um das Prinzip besser zu verstehen, kann es helfen, die verschiedenen Funktionen auf dem Papier anhand von einer konkreten Zahl einmal durchzurechnen.
Grundsatzlich gilt: Jede rekursive Lösung eines Problems lässt sich iterativ umsetzen und jede iterative Lösung lässt sich auch rekursiv umsetzen. Was sinnvoller ist, lässt sich dabei nicht so einfach sagen, da es dabei nicht nur auf das Problem ankommt, sondern auch auf die Anforderungen, die man selbst an die Lösung hat. Iterative Lösungen bieten zum Beispiel häufig den Vorteil, dass sie besser performen. Rekursion kann, auch wenn es am Anfang manchmal schwieriger zu verstehen ist, deutlich übersichtlicher sein und kommt häufig auch mit weniger Quellcode aus, ist allerdings auch speicherintensiver.

2 Python-Basics

2.1 Ausführen des Python-Codes

Bevor wir uns die Python-Syntax angucken, gucken wir uns an, wo die überhaupt ausgeführt werden kann. Dafür gibt es zwei Möglichkeiten: in der Command Line oder in einer Datei.
Für die erste Variante brauchen wir nur unsere Konsole. Um Python zu öffnen, geben wir python3 in diese ein. Das sieht dann folgendermaßen aus:

```
$ python3
Python 3.9.7 (default, Oct 12 2021, 22:38:23)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Darin können wir dann Python-Code ausführen:

```
>>> print('Hallo')
Hallo
>>> 21 + 42
63
```

Um Python dann wieder zu verlassen, können wir quit() in die Konsole eingeben. Die andere Möglichkeit ist, eine Datei zu erstellen mit der Endung '.py'. In diese Datei schreibt ihr dann euren Quellcode. Ausführen könnt ihr diesen dann, indem ihr in VSCode auf das Dreieck in der oberen rechten Ecke geht oder indem ihr in der Konsole 'python3 /Pfad/zur/datei.py' eingebt.
Wenn ihr im gleichen Verzeichnis wie euer Programm Python startet, könnt ihr außerdem direkt einzelne Funktionen aufrufen:

```
>>> import dateiname
>>> dateiname.funktionsname(parameter)
... Output ...
```

2.2 Syntax

Die Python-Syntax ist vor allem bekannt dafür, Pseudo-Code sehr ähnlich zu sein. Die Syntax ist also sehr intuitiv und kommt ohne viele Klammern aus. Dafür ist es wichtig, die einzelnen Anweisungen richtig einzurücken.
Folgender Code wäre korrekt:

```
1 for i in range(0,5):
2     print(i)
```

Dagegen würde folgendes nicht funktionieren bzw. sogar eine Fehlermeldung geben:

```
1 for i in range(0,5):
2 print(i)
```

Wie viel ihr einrückt, ist euch überlassen, wichtig ist, dass ihr es tut.
Wie man Variablen initialisiert, also den Wert zuweist, funktioniert in Python, wie wir im ersten Part gesehen haben, auch sehr intuitiv.
Es gibt jedoch ein paar Schreibweisen, die nicht genauso in jeder anderen Programmiersprache funktionieren. Eine wären die Mehrfachzuweisungen. Statt

```
1 a = 5
2 b = 3.1415
3 c = 'Hallo Welt'
```

kann man auch Folgendes machen:

```
1 a, b, c = 5, 3.1415, 'Hallo Welt'
```

Wenn man verschiedenen Variablen den gleichen Wert zuweisen möchte, kann man das außerdem so machen:

```
a = b = c = 1
a,b,c = 1
```

Wichtig ist dabei zu erwähnen, das bei der ersten Variante auf den gleichen Speicherplatz verwiesen wird. Bei Speicherplatz schonenden Datentypen wie Integern wird dies in der Regel noch kein Problem darstellen. Wenn wir uns später aber Numpy-Arrays angucken, hat die erste Schreibweise die Konsequenz, dass bei einer Änderung von b, auch a und c geändert werden.
Deshalb ist die zweite Schreibtweise zu bevorzugen.

Wenn man programmiert, wird man häufig etwas wie

```
xnum = xnum + 1
xnum = xnum * 2
xnum = xnum - 1
xnum = xnum / 2
```

schreiben müssen. Aber um euren Fingern diese unzumutbare Tipparbeit zu ersparen, gibt es dafür folgende kürzere Variante:

```
xnum += 1
xnum *= 2
xnum -= 1
xnum /= 2
```

Bei der Typzuweisung gibt es in Python jedoch noch eine spezielle Besonderheit:

2.3 Compiler vs. Interpreter

An der Stelle wird es kurz ein bisschen technisch, denn wir gucken uns an, wie unser Computer überhaupt mit einem Programm-Code bzw. Source-Code umgeht. Dies ist allerdings sehr relevant um zu verstehen, wie Datentypen in Python einer Variable zugewiesen werden und um allgemein besser zu verstehen wie bestimmte Fehlermeldungen zustande kommen können.
Bei Umgang des Computers mit dem Source-Code gibt es zwei Möglichkeiten:
Entweder wird der Code mit einem **Compiler** ausgeführt. In dem Fall wandelt der Compiler den kompletten Source Code (bei kleinen Programmen wie wir sie schreiben in der Regel eine Datei) in Machine Code um. Also eine Sprache, die der Computer leicht lesen kann. Diesen Machine Code kann man dan ausführen.
Die andere Möglichkeit ist die Verwendung eines **Interpreters**. In diesem Fall wird der Source-Code Zeile für Zeile gelesen und sofort ausgeführt. Python verwendet einen Interpreter.
Das wichtigste was man sich daraus mitnehmen muss, ist wann Fehler erkannt werden:
Bei einem Compiler fällt ein Fehler beim kompilieren auf. Der Machine Code kann nicht entstehen und somit kann nichts von dem Source Code ausgeführt werden.
Beim Interpreter dagegen kann wegen der zeilenweise Ausführung das Programm solange ausgeführt werden, bis dem Interpreter ein Fehler auffällt.
Wichtig ist auch, dass beachtet wird wann man Funktionen definiert: Bei einem Compiler reicht es, wenn die Funktion irgendwo im Programm definiert ist. Wenn bei einem Interpreter aber ein Funktionsaufruf auftaucht, der weiter 'oben' im Programm definiert wurde, dann gilt die Funktion als nicht definiert.

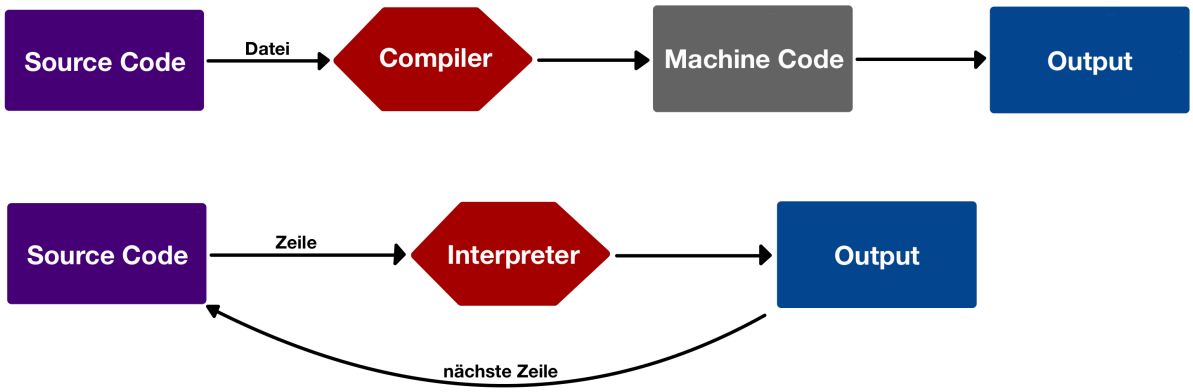

Figure 3: Compiler vs. Interpreter

2.4 Dynamische Typisierung

In den meisten Programmiersprachen, zum Beispiel in C, findet die Typprüfung statisch, d. h. während der Kompilierung statt. In Python ist das anders, dort findet die Typisierung dynamisch, also während der Laufzeit statt.
In der Praxis bedeutet das, dass man in Python im Gegensatz zu anderen Programmiersprachen nicht den Datentyp einer Variable mit angeben muss, sondern der Datentyp automatisch zugewiesen wird. Das klingt erst mal sehr einladend, sich nicht mit Datentypen und Typisierung in Python zu beschäftigen, aber spätestens nach dem ersten Fehler, der darauf zurückzuführen ist, werdet ihr euch sehr dankbar sein, wenn ihr es doch getan habt. Denn manchmal führt es auch zu Problemen:
Das wahrscheinlich häufigste Problem entsteht durch die Typzuweisung bei Integer und Floats. Dabei gilt stets zu beachten: 3 hat den Datentyp Integer, 3.0 hat den Datentyp Float. Wenn ihr zum Beispiel eine For-Schleife habt, könnt ihr sagen:

```
for i in range(0,3):
    print(i)
```

Aber Folgendes wirft einen Fehler aus:

```
for i in range(0,3.0):
    print(i)
```

Um so was zu vermeiden, gibt es zum Glück Funktionen, die den Datentyp verändern können.
Aber von Anfang an: Die Funktion, um den Datentyp auszugeben, heißt type().

```
print(type(3))
print(type(4.3))
print(type(1.99999))
print(type('Hallo'))
print(type([1, 2.5, 'Hallo']))
```

Die Ausgabe sieht dann folgendermaßen aus:

```
<class 'int'>
<class 'float'>
<class 'float'>
<class 'str'>
<class 'list'>
```

Falls ihr Anweisungen an die Bedingung knüpfen wollt, dass eine Variable einen bestimmten Datentypen hat, könnt ihr das folgendermaßen machen:

```
print(type(3) is int)
print(type(4.1) is int)
print(type([1, 2.5, 'Hallo']) is list)
print(type(5) is not str)
```

```
True
False
True
True
```

Wichtig bei der dynamischen Typisierung in Python ist, dass der Datentyp einer Variable sich verändern kann, ohne dass man ihn selbst aktiv ändert.
Zum Beispiel:

```
x = 3 #Datentyp von x: Integer
x = x // 2 #Datentyp von x: Integer (da ganzzahldivision verwendet)
x = x / 2 #Datentyp von x: Float
x = x / 3 #Datentyp von x: Float
```

Das heißt, wir können Operatoren auf Integer andwenden, die eigentlich für Floats gedacht sind. Der Rückgabewert ist dann allerdings immer ein Float.
Es kann passieren, dass wir Operatoren die Floats zurückgeben trotzdem verwenden müssen, aber letzendlich einen Integer brauchen. Für so einen Fall ist es möglich den Datentypen aktiv zu ändern. Dies wird als **type casting** bezeichnet.
Um den Datentypen zu casten, nimmt man die Funktion mit dem Namen von eurem Ziel-Datentyp:

```
a, b, c, d = int(3.0), int(2.2), int(1.9999), float(5)
print(a)
print(type(a))
print(b)
print(type(b))
print(c)
print(type(c))
print(d)
print(type(d))
```

```
3
<class 'int'>
2
<class 'int'>
1
<class 'int'>
5.0
<class 'float'>
```

Man kann sich natürlich denken, dass man nicht jeden Datentyp in jeden konvertieren kann. Wenn man versucht, einen String, der ein Wort enthält, in ein Integer zu konvertieren, gibt es eine Fehlermeldung.

2.5 Mehr zur Ablaufsteuerung

else bei Schleifen In Python ist es möglich, für die Schleifenbedingung auch eine Else-Anweisung zu geben:

```
x = 1
while x< 10:
    print("x ist kleiner 10")
    x += 3
else:
    print("x ist groesser 10")
```

Die Ausgabe dieses Programmes wäre dann:

```
x ist kleiner 10
x ist kleiner 10
x ist kleiner 10
x ist groesser 10
```

break, continue & pass Wie im ersten Part schon erwähnt, gibt es in Python auch noch andere Mittel, um den Programmablauf in einer Schleife zu steuern, als die Bedingung, an die sie gebunden ist. Dafür stellt Python zwei Statements zur Verfügung:

stiltechnische fraglichkeit erwähnen

- **break** gibt einem die Möglichkeit, eine Schleife vorzeitig zu unterbrechen, auch wenn die Bedingung, an die die Schleife eigentlich gebunden ist, noch nicht False geworden ist. Wichtig ist dabei, zu beachten, dass wenn man mithilfe von break eine Schleife abbricht, der dazugehörige Else-Block nicht ausgeführt wird.
- Durch **continue** wird der aktuelle Schleifendurchlauf unterbrochen und es wird direkt beim nächsten Schleifendurchlauf wieder angesetzt.

Hier mal ein konkretes Beispiel:

```
x = 0
while x<=10:
    if(x%2 != 0): #Pruefen ob x ungerade ist
        x += 1
        continue
    if(x == 8):
        break
    print("x hat den Wert", x)
    x += 1
else:
    print("x ist groesser 10")
```

Und so sieht die Ausgabe dann aus:

```
x hat den Wert 0
x hat den Wert 2
x hat den Wert 4
x hat den Wert 6
```

Das **pass**-Statement ist eine Anweisung die garnichts macht. Das mag erstmal sinnlos klingen, ist es aber garnicht. Denn normalerweise wenn man programmiert schreibt man nicht den ganzen Code runter und führt ihn aus. Wenn man das tun würde, würde man sehr lange Zeit damit verbringen alle Fehler im Code zu finden und am Ende funktioniert es doch irgendwo doch nicht so wie man es sich am Anfang gedacht hat und man muss alles nochmal neu schreiben.
Also schreibt man nach und nach den Code und guckt ob die einzelnen Teile funktionieren. Dabei kann es natürlich vorkommen, dass man bestimmte Teile erst später implementieren will. Bestimmte Code-Blöcke in Python wegzulassen, würde allerdings einen Syntax-Fehler erzeugen. Dieses Beispiel funktioniert zum Beipiel nicht:

```
x=1
while x <= 6:
    if (x%2 != 0):
    else:
        print("x ist ungerade")
    x += 1
```

Man würde folgende Fehlermeldung zurückbekommen:

```
else:
    ^
IndentationError: expected an indented block
```

Da kommt die pass-Anweisung ins Spiel. Denn wenn wir in den if-Block einfach pass setzen bekommen wir die gewollte Ausgabe:

```
x=1
while x <= 6:
    if (x%2 != 0):
        pass
    else:
```

```
        print("x ist ungerade")
    x += 1
```

```
x ist ungerade
x ist ungerade
x ist ungerade
```

Der einzige Zweck dieses Statements ist also uns die Arbeit während des programmierens zu erleichtern. In fertigen Programmen kommt diese Anweisung für gewöhnlich garnicht vor.

Conditional Expressions Ein Conditional Expression also wörtlich übersetzt 'bedingter Ausdruck' ist eine Kontrollstruktur in Python, welche Abhängig von einer Bedingung zwei verschiedene Werte annehmen kann.
Angenommen wir haben ein Boolean 'bool'. Wenn bool den Wert 'True' hat, soll eine Variable den Wert 100 annehmen, wenn der Wert 'false' ist, 5. Wenn man dies mit den bisherigen gelernten Werkzeugen umsetzen würde, könnte das so aussehen:

```
if bool:
    var = 100
else:
    var = 5
```

Mit ein Conditional Expression geht dies auch schlanker:

```
var = (100 if bool == True else 5)
```

Die Klammern sind dabei nicht unbedingt notwendig, jedoch erhöhen sie die Übersichtlichkeit.
Die 'Form' vom dem Conditional Expression lässt sich ganz gut merken, da diese sich aus der englischen Sprache ableitet: A **if** Bedingung **else** B.
Besonders praktisch können Conditional Expressions in Kombination mit der print()-Funktion sein. Als Beispiel hier eine Funktion, die ausgibt ob eine Zahl 'x' in dem Intervall $I = [5, 10]$ liegt:

```
print("x ist in I" if (x>5 and x<10) else "x ist nicht in I")
```

2.6 Dokumentationen lesen

Auch wenn es lästig erscheint: Man kann sich beim Programmieren viel Zeit sparen, wenn man in der Lage ist Dokumentationen zu lesen, da dies bei vielen Problemen schneller geht, als zu versuchen jemanden mit genau dem gleichen Problem auf Stackoverflow zu finden. Außerdem entwickelt sich auch die Programmiersprache Python regelmäßig weiter, so dass eine Lösung die bei einer Person vor 5 Jahren funktioniert hat, vielleicht in eurer Python Version nicht mehr funktioniert.
Deshalb gucken wir uns jetzt einmal an wie man so eine Dokumentation liest.
Im Allgemeinen sieht die Dokumentation einer Funktion so aus:

Rückgabedatentyp **Funktionsname** (**arg1**, **arg2**, arg3=1)

Der Rückgabedatentyp stellt dabei den Datentyp dar den eine Funktion zurückgibt. Die Übergabeparameter arg1 und arg2 sind Argumente, die die Funktion auf jedenfall benötigt. Bei arg3 ist bereits ein Wert dahinter spezifiziert. Das heißt, dass dieses **optional** ist und im Falle, dass kein drittes Argument spezifiziert wird es den angegebenen Wert annimmt, den **Defaultwert**.

Die Python Dokumentation findet man auf docs.python.org. Bei der Verwendung sollte man darauf achten die richtige Python Version auszuwählen.
Eine andere gute Seite ist devdocs.io. Dort findet ihr nicht nur die Python Dokumentation, sondern auch die Dokumentation von vielen anderen Programmiersprachen und Libarys.

2.7 Die range()-Funktion

In CompMath wird es sehr häufig vorkommen, dass ihr eine Liste über einen bestimmten Zahlenabschnitt benötigt.
Bevor ihr anfangt, immer eine Liste händisch einzutippen, möchte ich euch die range()-Funktion ans Herz

legen, die genau für diesen Zweck existiert und die wir im letzten Teil schon mal im Zusammenhang mit der For-Schleife gesehen haben. Kurz gesagt erstellt diese eine Folge bzw. eine Sequenz an Zahlen.
Die range()-Funktion bekommt bis zu drei Parameter übergeben: range(start, stop, step).

- **start (optional)**: legt den Anfangswert euer Folge fest. Dieser Parameter ist optional. Der **Defaultwert** ist 0, d. h. wenn ihr diesen Parameter nicht angebt, funktioniert die Funktion so, als hättet ihr 0 angegeben.
- **stop**: legt den Endwert euer Folge fest. Dieser Parameter ist nicht optional, d. h. wenn ihr den nicht mit angebt, funktioniert die Funktion gar nicht bzw. wirft einen Fehler aus. Ihr werdet gleich noch in den Beispielen sehen, dass euer Endwert nicht mehr dazuzählt. Also wenn der Endwert 15 ist, ist der letzte Wert der Folge 14.
- **step (optional)**: legt die Schrittweite eurer Folgen fest. Dieser Parameter ist optional und der Defaultwert ist 1.

In Python2 hat die Funktion noch eine Liste zurückgegeben, was relativ ineffektiv ist. In Python3 ist das deshalb wie gesagt nicht mehr der Fall. Da bekommt man ein range-Object, welches auch Sequence genannt wird. Deshalb wird die range()-Funktion auch manchmal range()-type genant. Beispiel:

```
x = range(3,7)
print(x)
```

```
range(3,7) #Ausgabe in Python3
[3,4,5,6] #Ausgabe in Python2
```

Um zu veranschaulichen, wieso ihr über die Folge trotzdem iterieren könnt, kann man diese mit der Funktion list() in eine Liste umwandeln:

```
x = range(3,7)
print(list(x))
```

Dann ist die Ausgabe in Python3 die gleiche Liste wie in Python2.
Hier sind noch ein paar weitere nützliche Beispiele:

```
list1 = list(range(5))
list2 = list(range(-1,-11,-1))
list3 = list(range(5,-1,-1))
list4 = list(reversed(range(6))) #Die reversed()-Funktion gibt eine Sequenz umgekehrt zurueck
print(list1)
print(list2)
print(list3)
print(list4)
```

Die Ausgabe ist dann:

```
[0, 1, 2, 3, 4] #list1
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10] #list2
[5, 4, 3, 2, 1, 0] #list3
[5, 4, 3, 2, 1, 0] #list4
```

Es kann natürlich auch praktisch sein, eine Sequenz mit z. B. 0.5 Schrittweite zu haben. Das wäre mit der range()-Funktion nicht umsetzbar, da diese eine Fehlermeldung gibt, sobald man Floats als Parameter übergibt. Eine Lösung dafür bietet Numpy mit der Funktion numpy.arange(), die wir im letzten Part kennenlernen werden.

2.8 List Comprehension

List Comprehension ist eine Möglichkeit, aus einer bereits bestehenden Liste eine neue zu erstellen. Der Grundaufbau dafür ist:

```
newlist = [Expression for item in list if condition == True]
```

In echten Code sollte man übrigens eine Liste niemals 'list' nennen, da ansonsten es dem Interpreter schwer fällt auseinander zu halten, wann die Funktion zur Datentypumwandlung "list()" gemeint ist und

wann die Variable list. (Das gleiche gilt für int, str, float, usw.)
Um mal ein Beispiel für List Comprehension zu geben: Wir haben eine Liste mit allen Zahlen von 1 bis 20 und wollen eine neue Liste mit allen geraden Zahlen. Dann können wir das folgendermaßen machen:

```
list1 = list(range(1,21))
newlist1 = [x for x in list1 if x % 2 == 0]
print(newlist1)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Ein etwas komplizierteres Beispiel: Wir haben eine Liste von Tupeln. Im nullten Eintrag der Tupeln befindet sich ein String mit dem Namen einer Person und im ersten Eintrag das Alter. Wir wollen eine neue Liste haben, wo die Namen aller Personen drauf sind, die 20 oder jünger sind:

```
list2 = [('Bob', 18), ('Amelie', 21), ('Katja', 20), ('Jonas', 23), ('Luisa', 19), ('Achmed',
    19), ('Tara', 17)]
list2_a = [x[0] for x in list2 if x[1] <=20 ]
list2_b = [x for x,y in list2 if y <= 20]
print(list2_a)
print(list2_b)
```

```
['Bob', 'Katja', 'Luisa', 'Achmed', 'Tara']
['Bob', 'Katja', 'Luisa', 'Achmed', 'Tara']
```

2.9 Namens-Konventionen

Für Programmiersprachen gibt es nicht nur eine Syntax an die man sich halten muss, sondern auch Konventionen, an die man sich halten sollte. Zum einen für einen guten Programmierstil, aber auch um die Lesbarkeit für andere Programmierer zu gewährleisten.
Prinzipiell gibt es erstmal ein paar allgemeine Regeln für Namen, die so bei allen Programmiersprachen gelten:

- Vergebe aussagekräftige und konsistente Namen: Dieser Punkt wurde der Übersichtlichkeit halber bislang ein bisschen vernachlässigt und Variablen nach dem Datentypen und nicht nach dem Inhalt benannt. In Programmcode sollte man dies allerdings nicht so machen. Konsistent in dem Fall heißt, dass man gerne ein Schema bei der Benennung innerhalb eines Programms haben sollte.
- Sonderzeichen, Leerzeichen, Umlaute usw. sind potenziell schwierig: Selbst wenn sie bei euch gerade funktionieren, kann dies auf anderen Geräten trotzdem zu Problemen führen.
- Wenn man mehrere Wörter in einem Namen verwendet, sollte man (in Python) Snake Case oder Camel Case verwenden. In anderen Sprachen gibt es auch noch die Kebab Case und Pascal Case Schreibweisen. Diese sind in Python aber nicht üblich.
  - **Snake Case**: Trennung der Wörter mit Hilfe eines Unterstrichs. z.B. 'snake_case'
  - **Camel Case**: Kennzeichnung eines neuen Wortes durch Großschreibung, wobei der erste Buchstabe trotzdem klein geschrieben wird. z.B. 'camelCase'.
  - **Pascal Case**: (nicht üblich in Python) Kennzeichnung eines neuen Wortes durch Großschreibung, wobei der erste Buchstabe groß geschrieben wird. z.B. 'PascalCase'
  - **Kebab Case**: (nicht üblich in Python) Trennung der Wörter mit Hilfe eines Bindestriches. z.B. 'kebab-case'.
- Vorsicht mit reservierten Wörtern: Wenn man Wörter in Python verwendet, die eigentlich schon für Funktionen reserviert sind oder ein Schlagwort darstellen (wie z.B. if, else, for, int, list, usw.), dann fällt einen das bei der Verwendung der Variable auf die Füße, oder spätestens, wenn man das nächste mal versucht die Funktion aufzurufen. Wenn man z.B. eine Liste list nennt, dann ist die Funktion list() von da an nicht mehr aufrufbar.
- Verwende, wenn möglich, nicht allzu lange Namen, denn es kann den Code unübersichtlich machen.

2.10 Libraries

Bevor wir anfangen uns mit bestimmten Python-Libraries auseinanderzusetzen, sollten wir 2 Sachen klären:

1. Was sind Libraries (im Kontext des Programmierens)?
2. Wie verwendet man Libraries in Python?

zu 1.) Libraries sind Sammlungen von vordefinierten Datentypen und Funktionen, sowie Routinen und Skripten, mit dem Zweck gängige Probleme auszulagern, damit andere Programmierer dadurch schneller und besser programmieren können.
zu 2.) Dies haben wir bereits vorher gesehen, nämlich im Kapitel 1.3.2, als wir eine vordefinierte Approximation von π verwenden wollten und dafür die Mathebibliothek *math* importiert haben.
Unsere 3 am Anfang installierten Libraries, können wir also folgendermaßen importieren:

```
1 import numpy
2 import scipy
3 import matplotlib
```

Wenn wir nach dem Importieren ein Bestandteil einer Library verwenden wollen, müssen wir dennoch kenntlich machen, aus welcher Bibliothek wir diesen nehmen, in dem wir voran *libraryname.* schreiben. So würde man z.B. ein NumPy-Array verwenden:

```
1 import numpy
2 narr = numpy.array([1,2,3,4])
```

Bei Matplotlib gibt es noch eine Besonderheit: Überwiegend verwendet man ein spezifisches Modul, nämlich **pyplot**. Demnach würde man Funktionen aus diesem folgendermaßen aufrufen:

```
1 import matplotlib
2
3 matplotlib.pyplot.plot([1,2,3,4])
4 matplotlib.pyplot.show()
```

Dies ist natürlich echt umständlich und davon raten auch die Matplotlib-Entwickler selbst ab (weshalb es auch immer häufiger nicht mehr funktioniert) und weshalb man glücklicherweise einen *alias* erstellen kann:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 narr = np.array([1,2,3,4])
5 plt.plot(narr)
6 plt.show()
```

Es gibt noch weitere Varianten zum Importieren, die äquivalent sind oder wo man nicht einmal den Alias der Bibliothek davor schreiben muss. Da das davor schreiben des Library-Alias vor dem verwendeten Bibliotheksbestandteil aber die Lesbarkeit erhöht, werden wir erstmal dabei bleiben.

Und eine kurze Anmerkung, bevor es mit NumPy weitergeht: Vergleicht mal die Art, wie Bibliotheken importiert werden mit dem Importieren eures eigenen Skripts in dem interaktiven Modus, wie wir es im Kapitel 2.1 getan haben.

3 Beyond the Python-Basics

3.1 weitere Datentypen & Funktionen

3.1.1 Tuples

Tupel hinzufügen

3.1.2 Dictionaries

Dictionaries sind Datentypen, die letzendlich genau das tun, was man sich unter dem Begriff vorstellt: Sie ordnen jeden Element der Kategorie 1 ein Element der Kategorie 2 zu. Zum Beispiel jeden deutschen Wort die englische Übersetzung, jedem Produkt einen Preis oder jeden Zeichen ein Ascii-Code:

```
asciitable = {'A': 65,
              'B': 66,
              'C': 67,
              'D': 68,
              'E': 69,
              'F': 70,
              'G': 71,
              'H': 72,
              'I': 73,
              'J': 74,
              'K': 75,
              'L': 76,
              'M': 77,
              'N': 78,
              'O': 79,
              'P': 80,
              'Q': 81,
              'R': 82,
              'S': 83,
              'T': 84,
              'U': 85,
              'V': 86,
              'W': 87,
              'X': 88,
              'Y': 89,
              'Z': 90}
```

Was ist der ASCII-Code?

Da Computer bekanntlich nur Zahlen im Binärsystem abspeichern können, brauchen wir für jedes Zeichen eine Zahl, die dem zugeordnet werden kann. Dabei kommt der ASCII-Code ins Spiel:
ASCII steht für **A**merican **S**tandart **C**ode for **I**nformation **I**nterchange und ist die standardisierte Kodierung für Zeichensätze.
Mittlerweile sind aber sehr viele Zeichen hinzugekommen, wie z.B. Umlaute, aber auch Smileys und Schriftzeichen aus sämtlichen anderen Sprachen, weshalb ASCII mit seinen 128 Zeichen heutzutage zwar die Grundlage für moderne Zeichencodierung darstellt, es aber sehr viele Erweiterungen gibt.

Wenn wir jetzt den ASCII Wert eines Buchstabens erfahren wollen, können wir uns dieses folgendermaßen ausgeben lassen:

```
print(asciitable["B"]) #Output: 66
print(asciitable["R"]) #Output: 82
```

Diese Paare in Dictionaries nennt man **Schlüssel-Wert-Paare** (oder **Key-value-Pairs**), wobei der erste Wert immer der Schlüssel und der zweite der Wert ist.
Bevor man die Dictionaries eintippt, gibt es auch für diesen Datenypen eine Funktion um einen anderen Datentypen in ein Dictionary zu konvertieren, die dict()-Funktion:

```
ascii_list = [['A', 65], ['B', 66], ['C', 67], ['D', 68]]
print(dict(ascii_list)) #Output: {'A': 65, 'B': 66, 'C': 67, 'D': 68}
```

Wenn man die Schlüssel und Wert aber nicht in einer 2-dimensionalen Liste, sondern in 2 seperaten Listen gegeben hat, dann kann man die zip()-Funktion verwenden:

```
ascii_Keys = ['A', 'B', 'C', 'D', 'E', 'F']
ascii_Values = range(65, 71)
print(dict(zip(ascii_Keys, ascii_Values))) #Output: {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E':
    69, 'F': 70}
```

Bei der richtigen Verwendung von Dictionaries sollte man aber auch einige Sachen beachten:
Dictionaries sind seit Python 3.7 geordnete Sammlungen von Elementen. Davor waren sie ungeordnet. Wenn man sich also nicht sicher ist, ob man die neuste Python Version hat, lohnt es sich das zu überprüfen.

Wenn in der Schlüssel-Spalte mehrmals der gleiche Wert enthalten ist, wird der letzte davon ausgegeben:

```
asciitable = {'A': 65,
              'B': 66,
```

```
                'B': 67,
                'D': 68}
print(asciitable["B"]) #Output: 67
```

Prinzipiell kann man immer nur dem Schlüssel des Dictionaries einen Wert zuordnen. Wenn man auch einen Wert einen Schlüssel zugeordnet möchte, kann man z.B. eine Funktion schreiben, die ein zweites 'umgedrehtes' Dictionary erstellt oder über die Elemente des Dictionaries iteriert und diese überprüft und bei Übereinstimmung des Werts den Schlüssel ausgibt.

Wenn man für einen Schlüssel mehrere Werte aus verschiedenen Kategorien hat, zum Beispiel ein Englisches Wort und die deutsche, französische und spanische Übersetzung oder die ASCII-Tabelle nicht nur in Zeichen und Dezimalzahl, sondern auch noch binärzahl haben möchte, dann kann man sich dafür ein Dictionary der Dictionaries erstellen oder ein Subdictionary in dem Dictionary erstellen:

```
ascii_letters = ['A', 'B', 'C', 'D', 'E', 'F']
ascii_dezimal = range(65, 71)
ascii_binary = list(map(lambda x: bin(x)[2:], ascii_dezimal))

super_dict = {
    ascii_letters[i]: {'decimal': ascii_dezimal[i], 'binary': ascii_binary[i]}
    for i in range(len(ascii_letters))
}

print(super_dict) #Output: {'A': {'decimal': 65, 'binary': '1000001'}, 'B': {'decimal': 66, '
    binary': '1000010'}, 'C': {'decimal': 67, 'binary': '1000011'}, 'D': {'decimal': 68, 'binary
    ': '1000100'}, 'E': {'decimal': 69, 'binary': '1000101'}, 'F': {'decimal': 70, 'binary':
    '1000110'}}
print(super_dict['B']) #Output: {'decimal': 66, 'binary': '1000010'}
print(super_dict['D']['decimal']) #Output: 68
print(super_dict['F']['binary']) #Output: 1000110
```

3.1.3 Die enumerate()-Funktion

enumerate()-Funktion in Python tut letzendlich genau das, was ihr name sagt: Sie numeriert Elemente einer iterierbaren Variable:

```
weekdays = ['Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag', 'Sonntag']
print(list(enumerate(weekdays))) #Output: [(0, 'Montag'), (1, 'Dienstag'), (2, 'Mittwoch'), (3,
    'Donnerstag'), (4, 'Freitag'), (5, 'Samstag'), (6, 'Sonntag')]
```

Der default start-Wert ist, wie üblich, 0. Wenn wir von 1 anfangen wollen zu zählen, dann können wir dies auch spezifizieren:

```
weekdays = ['Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag', 'Sonntag']
print(list(enumerate(weekdays, start=1))) #Output: [(1, 'Montag'), (2, 'Dienstag'), (3, '
    Mittwoch'), (4, 'Donnerstag'), (5, 'Freitag'), (6, 'Samstag'), (7, 'Sonntag')]
```

Standardmäßig gibt die enumerate()-Funktion ein enumerate-Object zurück. Wenn wir darauf die list()-Funktion anwenden, bekommen wir die Tupel-in-Liste-Schreibweise zurück. Da wir jeder Zahl einen Wochentag zugeordnet haben, stellt dieses Beispiel aber auch ein optimaler Kandidat für einen Dictionary dar. Dies ist genauso einfach zu implementieren:

```
weekdays = ['Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag', 'Sonntag']
print(dict(enumerate(weekdays, start=1))) #Output: {1: 'Montag', 2: 'Dienstag', 3: 'Mittwoch',
    4: 'Donnerstag', 5: 'Freitag', 6: 'Samstag', 7: 'Sonntag'}
```

3.1.4 Sets

Extra: Set-Datentyp hinzufügen

3.2 Anonyme Funktionen & Funktionen höherer Ordnung

Extra: Definition Anonyme Funktion & Funktionen hoeherer Ordnung

3.2.1 Die map()-Funktion

Die map()-Funktion ist besonders praktisch, wenn man mit iterierbaren Datentypen wie Listen oder Tupeln arbeitet. Sie bekommt als erstes Argument eine Funktion und als zweites einen iterierbare Variable übergeben und wendet dann diese Funktion auf jedes einzelne Element an. Zurückgegeben bekommt man ein map-Object, was auch ein iterierbarer Datentyp ist. Mit der list()-Funktion kann man sich dieses einfach als Liste darstellen lassen.
Beispiel: Wir haben eine Liste gegeben und wollen jedes Element in der Liste quadrieren. Ohne map()-Funktion könnten wir über die Liste iterieren und jedes Element einzelnd quadrieren und dabei entweder in einer neuen Liste speichern oder das Element der alten Liste überschreiben. Mit der map()-Funktion, können wir dies ein bisschen kürzer ausdrücken:

```
list1 = [1,2,3,4,5]
def quad(x):
    return x**2
list2 = list(map(quad, list1))
print(list2) #Output: [1, 4, 9, 16, 25]
```

3.2.2 Der Lambda-Operator

Mit dem Lambda-Operator erzeugte Funktionen können 'anonym' sein. Das heißt, die Funktion hat keinen Namen. Sie kommen ursprünglich aus der funktionellen Programmierung.
Ein Lambda-Operator in Python, der $f(x, y) = x^2 + y^2$ implementiert sieht folgendermaßen aus:

```
f = lambda x,y: x**2 + y**2
f(2,3) #Aufruf der Lambda-Funktion
```

Mal abgesehen davon, dass man sich mit dem lambda-Operator die ein oder andere Zeile Code spart, wird die Eleganz des Operators vor allem sichtbar in Kombination mit der map()-Funktion, hier am Beispiel der zu quadrierenden Liste:

```
list1 = range(1,6) #equivalent zu [1,2,3,4,5]
list2 = list(map(lambda x: x**2, list1))
print(list2) #Output: [1, 4, 9, 16, 25]
```

3.2.3 join()-Funktion

3.2.4 reduce()-Funktion

3.2.5 filter()-Funktion

Extra: join()-Funktion

Extra: reduce()-Funktion

Extra: filter()-Funktion

Extra: generators, objects & yield

4 NumPy-Basics

4.1 Wozu braucht man NumPy?

In der CoMa-Vorlesung kommt man relativ schnell an den Punkt, wo man mit Vektoren und Matrizen rechnet und auch in den Programmieraufgaben damit arbeiten muss.
Dabei stellt sich als Erstes die Frage, wie man Vektoren und Matrizen überhaupt in Python implementiert? Mit unserer bisherigen Erfahrung mit Python, könnten wir zum Beispiel Vektoren als Liste speichern und Matrizen als Liste von Listen.

Beispiel: $e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
e1 = [1, 0, 0]
I3 = [[1, 0, 0],
      [0, 1, 0],
      [0, 0, 1]]
```

Was erstmal nach einer guten Lösung aussieht, bringt aber schnell ein paar Probleme mit sich, die bei den Python Listen aus mathematischer Sicht nicht sinnvoll sind: Beispielsweise kann eine Liste in Python verschiedene Datentypen enthalten. Diese Flexibilität ist aus der reinen Programmierer-Sicht bestimmt

praktisch, aber wenn man Funktionen für Matrizen und Vektoren implementieren möchte, wäre es auf dauer schon ziemlich aufwendig jedes mal überprüfen zu müssen, ob die übergebene Liste auch die Eigenschaften einer Matrix hat.
Dies ist einer der Gründe, wieso für mathematische Zwecke folgender Datentyp bevorzugt wird:

4.2 Das NumPy-Array

Das NumPy-Array ist ein Datentyp aus der Python-Erweiterung NumPy, welcher durch klassische Listen erzeugt wird und die Eigenschaften der Liste so modifiziert, dass mathematische Operationen auf diesen effektiv angewendet werden können.
Um deutlich zu machen, wieso NumPy-Arrays und keine Listen verwendet werden sollten, hier eine Übersicht, worin sich diese unterscheiden.

Python-Liste	NumPy-Array
Es können verschiedene Datentypen enthalten sein.	Dies ist ein **homogener** Datentyp, d.h. es können nur Daten vom gleichen Datentyp enthalten sein. Wenn unterschiedliche Datentypen enthalten sind, gibt es entweder eine Fehlermeldung oder die Elemente werden angepasst.
Listen sind **dynamisch** (d.h. ihre Länge ist Variabel)	Die Länge (bzw. hier eher Dimension) steht von Anfang an fest.
Für jedes Element werden bestimmte Metadaten gespeichert, wie den Datentypen und ein Pointer (dies ist ein Datentyp, der auf eine bestimmte Position im Speicher zeigt[1])	Es werden Metadaten für die gesamte Liste gespeichert, wie z.B. welcher Datentyp enthalten ist und die Form (Dimension) und weitere mathematisch relevante. In der Regel sind es aber insgesamt weniger als bei einer Liste.
Die Elemente einer Liste werden nicht zwangsläufig angrenzend gespeichert (Eine Konsequenz der Flexibilität). Dies sorgt dafür, dass die Liste im Vergleich ein ineffizienter ist.	Die Elemente werden *zusammenhängend* gespeichert.
Die hohe Flexibilität (sowohl bzgl. der Größe, als auch der enthaltenen Datentypen) geht mit einer schlechteren Performance bei numerischen Berechnungen einher.	Der Datentyp hat eine vergleichsweise **gute Performance bei numerischen Operationen** (schließlich wurde er auch dafür erstellt).
Die arithmetischen Operatoren sind nicht zwangsläufig implementiert oder nicht so implementiert, wie man es für die Arbeit mit Vektoren und Matrizen benötigen würde. (Beispiel: Addition verkettet zwei Listen, Subtraktion ist nicht definiert)	Die arithmetischen Operatoren funktionieren überwiegend so, wie in der Mathematik benötigt. (Beispiel: Addition addiert elementweise, Subtraktion subtrahiert elementweise)

Table 3: Vergleich: Liste und NumPy-Array

Einige nennen NumPy-Arrays auch einfach nur **Arrays**. Formal betrachtet ist dies korrekt, denn NumPy-Arrays haben die Eigenschaften nach der Definition eines Arrays. Jedoch verwenden viele, dadurch, dass es *nativ* keinen Array-Datentyp in Python gibt, Array als auch Synonym für Liste. Deshalb ergibt es kommunikationstechnisch Sinn immer spezifisch NumPy-Array und Liste zu verwenden.

Doch wie erzeugt man NumPy-Arrays eigentlich? Dies macht man mit einer Funktion, die ein (N-Dimensionales) NumPy-Array erzeugt. Am gängigsten ist die Funktion numpy.array(). Das einzige nicht-optionale Argument ist dabei eine Liste.
Wir könnten das obige Beispiel also folgendermaßen implementieren:

ndarray - der wahre Datentyp hinter np array und erzeugen von arrays

[1] Ja, dies ist eine starke Vereinfachung. Aber dies ist kein C oder C++ Kurs!

Variante 1:

```
import numpy as np

e1 = [1, 0, 0]
I3 = [[1, 0, 0],
      [0, 1, 0],
      [0, 0, 1]]
e1, I3 = np.array(e1), np.array(I3)
```

Variante 2:

```
import numpy as np

e1 = np.array([1, 0, 0])

I3 = np.array([[1, 0, 0],
               [0, 1, 0],
               [0, 0, 1]])
```

Es können allerdings nicht aus allen Listen NumPy-Arrays erzeugt werden.

Hinzufügen, welche Listen zur Erzeugung von NumPy Arrays verwendet werden koennen & bei welchen man aufpassen muss

4.2.1 Nützliche Matrizen

Es gibt viele Matrizen, die in der Mathematik häufig verwendet werden und wo es echt zeitsparend wäre, wenn man diese nicht jedes Mal eintippen müsste bzw. welche man auch flexibel in bzgl. der Größe erzeugen kann.

4.2.2 Gitter

4.3 Matrixoperatoren & -funktionen

4.3.1 Matrixoperatoren

4.3.2 Indizierung von Matrizen

4.3.3 Inverse einer Matrix & Lineare Gleichungssysteme

4.4 mathematische Konstanten & Funktionen

5 Matplotlib-Basics

Anhang

6 Dateien lesen und schreiben

Die letzten praktischen Funktionen die wir uns in diesen Kapiteln angucken, sind die Funktionen zum lesen und schreiben von Dateien. Dies kommt ein vor allem zu gute, wenn man zum Beispiel eine Datei mit vielen Daten einlesen oder ändern will oder in nächstem Kapitel ein Plot als Bild speichern will.

Öffnen einer Datei Um die Datei zu lesen oder zu schreiben, müssen wir sie erstmal öffnen. Dafür verwenden wir die **open()-Funktion**. Dieser Funktion übergibt man eine Datei, (bzw. wenn die Datei nicht im selben Verzeichnis wie die Python-Datei ist muss man auch dem Pfad zur Datei übergeben) und einen Modus, also open("Dateiname.endung" , "Modus")
Betrachten wir erstmal 4 verschiedene Modi:

- **read**: Schreibweise: r. Das bedeutet, dass die Datei nur zum lesen geöffnet wird.
- **write**: Schreibweise: w. Damit öffnet man die Datei nur zum schreiben.
- **append**: Schreibweise: a. Die Datei wird wieder nur zum schreiben geöffnet. Im Gegensatz zum Write-Modus wird beim bestehen einer gleichnamigen Datei diese nicht überschrieben, sondern erweitert.
- **read and write**: Schreibweisen: r+, w+, a+. Dadurch wird die Datei zum lesen und schreiben geöffnet. Der einzige Unterschied zwischen r+,a+ und w+ ist, dass bei w+ eine vorher bestehende Datei überschrieben wird.

Es gibt noch weitere Modi, welche die Dateien im Binärmodus öffnen. Dies ist aber erstmal nicht wichtig für uns.
Die open()-Funktion gibt uns ein File Obejct zurück.
Nach dem man alles mit der Datei gemacht hat was man tun wollte, sollte man die Datei wieder schließen. Dies nicht zu tun, erzeugt einem in neueren Python-Versionen keinen Fehler mehr, gehört aber zum guten Programmierstil. Denn wenn man zum Beispiel eine Datei mehrfach öffnet bearbeitet und nicht wieder schließt, kann es vorkommen, das Änderungen nicht vorgenommen werden oder andere Programme können die Datei dann möglicherweise garnicht öffnen. Das schließen der Datei können wir mit der **close()-Funktion** tun.

Wir haben eine Datei 'namen.txt' mit folgendem Inhalt:

```
Bob 18
Amelie 21
Katja 20
Jonas 23
Luisa 19
Achmed 19
Tara 17
```

Das öffnen der Datei würde dann so möglich:

```
1 datei = open("namen.txt","r")
2 #Operationen auf unserer Datei
3 datei.close()
```

Es gibt aber noch eine weitere (elegante) Möglichkeit, wie wir die Datei öffnen und schließen können, ohne dass wir die close()-Funktion dafür verwenden: Dafür verwenden wir das **with**-Statement. Dieses wird im Allgemeinen dafür eingesetzt, wenn man Operationen anwendet, die auch wieder korrekt deinitialisiert werden müssen, wie das schließen unserer Datei. Das with-Statement überträgt dabei die Verantwortung für deinitialisieren auf das Objekt selber.
Unser Code von oben würde dann so aussehen:

```
1 with open("namen.txt","r") as datei:
2     #Operationen auf unserer Datei
```

Lesen der Datei Nachdem wir die Datei geöffnet haben, ist das Lesen der Datei jenachdem was man machen will und wie die Daten gespeichert sind sehr einfach oder ein bisschen friemelig.
Versuchen wir unsere Datei erstmal **Zeilenweise auszulesen**, das geht nämlich ziemlich einfach, denn wir können Zeilenweise über ein File-Object iterieren:

```
1 with open("namen.txt","r") as datei:
2     for zeile in datei:
3         print(zeile)
```

```
Bob 18
Amelie 21
Katja 20
Jonas 23
Luisa 19
Achmed 19
Tara 17
```

Als nächstes wollen wir die Datei so auslesen, dass wir unsere 'list2' von oben bekommen, also jede Zeile wird zu einem Listenelement und alle mit einem Leerzeichen getrennten Teile werden in einem Tupel getrennt. Dies können wir mit der Methode split tun, welcher wir ein Leerzeichen übergeben. Zurückgeben tut uns diese Funktion eine Liste, wobei jede Zeile eine Unterliste ist.

```
1 list2 = []
2 with open("namen.txt","r") as datei:
3     for zeile in datei:
```

```
        zuordnung = zeile.split(" ") #Teilen der Zeile bei jedem Leerzeichen, gibt Liste zurck
        list2 += [(zuordnung[0], int(zuordnung[1]))] #Verkettung der Listen und konvertieren von
     string zu integer
    print(list2)
```

Wenn wir etwas aus einer Datei auslesen, wird der Inhalt erstmal als String interpretiert. Deshalb müssen wir die Zahlen dementsprechend noch konvertieren. Unsere Ausgabe sieht dann so aus:

```
[('Bob', 18), ('Amelie', 21), ('Katja', 20), ('Jonas', 23), ('Luisa', 19), ('Achmed', 19), ('
    Tara', 17)]
```

Schreiben in eine Datei Als letztes wollen wir in unsere Datei schreiben. Als erstes wollen wir weitere Daten einer Person Anhängen, dafür nehmen wir Anne, 22. Dafür müssen wir an die oben erwähnten Modi der open()-Funktion erinnern: Denn für das anhängen existiert doch der append-Modus.
Mit der write()-Funktion schreiben wir dann in die Datei. Dieser Funktion müssen wir einen String übergeben.

```
neueDaten = ('Anne', 22)
with open("namen.txt","a") as datei:
    datei.write(neueDaten[0]+" "+str(neueDaten[1])+"\n")
```

Die Escape Sequence \n macht einen Zeilenumbruch in einem String. Man kann dies also auch verwenden um mit der print()-Funktion etwas über mehrere Zeilen auszugeben.
Eine **Escape Sequence** ist ein Zeichenkombination, welche eine andere Bedeutung hat als die literarische Bedeutung dieser Zeichen. Die wahrscheinlich häufigste Escape Sequence die wir verwenden sind die Anführungszeichen ' oder " um einen String einzuleiten. Abgesehen von diesen Beispiel werden die meisten Escape Sequenzen in Python mit einem Backslash \eingeleitet. Wenn wir versuchen würden diesen so print("\") auszugeben, würden wir eine Fehlermeldung bekommen. Deshalb ist dieses Symbol auch über eine Escape Sequence definiert. So würde es nämlich funktionieren:

```
print("\\")
```

```
\
```

to be continued ...

Todo list