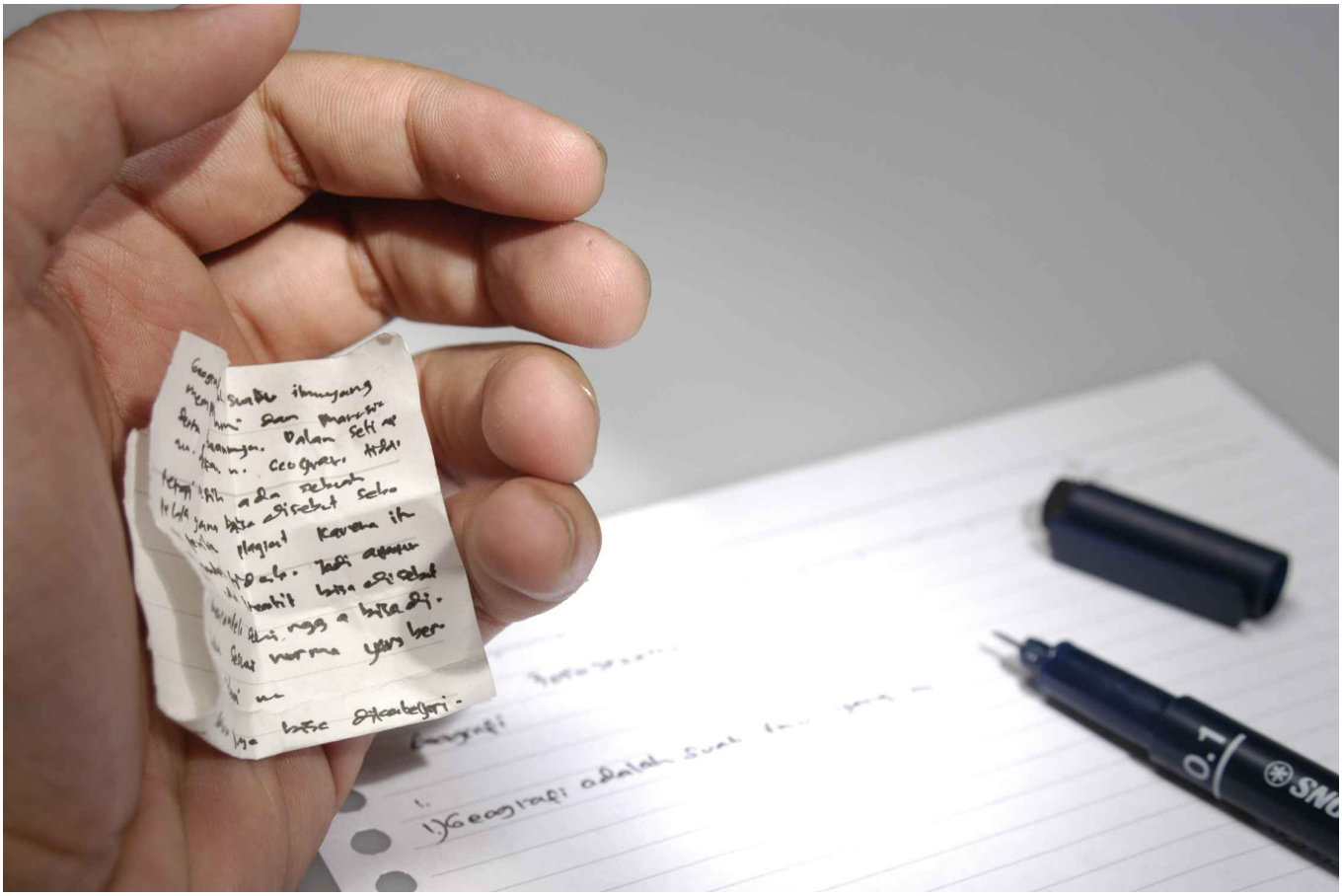
Cheat Sheet 1.0



Dieses Notebook soll Ihnen als Spickzettel dienen und alle bisher gelernten Werkzeuge an einer Stelle zusammenfassen. Sie können diess Notebook auch sehr gerne als Basis für Ihren eigenen Spicker nutzen oder dieses Notebook mit Notizen anreichern.

Bitte stellen stellen Sie in dieser Sitzung unbedingt Fragen zu Konzepten und Werkzeugen, mit denen Sie noch Schwierigkeiten haben!

Choose your own adventure...

- **Strings**
- **Variablen**
- **Methoden**
- **Integers (ganze Zahlen)**
- **Floats (Kommazahlen)**
- **Vergleichsoperatoren**
- **Listen**
- **Slicing**
- **Mutability**
- **Dictionarys**
- **if/elif/else (Bedingungen)**
- **for-Schleifen**
- **Kombination von Bedingungen und Schleifen**
- **Funktionen**
- **Module**
- **Dateien lesen**
- **Dateien schreiben**
- **User-Input mit `input()`**
- ... oder haben Sie Fragen zu anderen Themen?

Strings

Alle Daten in Python haben einen Datentyp. Der erste Datentyp, den wir kennengelernt haben, ist "String".

In Variablen können wir Daten aller Typen speichern.

Variablenzuweisung bzw. Initialisierung

- Variablennamen wählen - Regeln für Variablennamen:
  - Muss mit Buchstabe oder "_" beginnen, keine Zahlen zu Beginn
  - Keine Sonderzeichen oder reservierte Namen (Funktionsnamen, Operatoren)
  - Bedenken Sie: Groß-/Kleinschreibung wird vom Python-Interpreter beachtet
- mit dem einfachen "=" einen Wert zuweisen

In [ ]:

```
string_1 = "Eine Zeichenkette aus (fast) beliebigen Zeichen"
print(string_1)
```

In [ ]:

```
string_2 = "\"  # Geht das?
print(string_2)
```

Nein, denn `\` ist ein sog. Escape-Character, mit dem wir auch Zeichen, die in Python besondere Bedeutung haben, als Werte benutzen. Deshalb wird das zweite Anführungszeichen nicht als Ende des Strings interpretiert. Ein abschließendes Anführungszeichen fehlt diesem String, daher die Fehlermeldung.

In [ ]:

```
string_3 = "C:\\Users\\Admin"
string_4 = "C:/Users/Admin"  # Die Schreibweise mit "/" spart Tipparbeit
print(string_3)
print(string_4)

print("-----")

string_5 = "\\\\@//"
print(string_5)  # Was wird hier ausgegeben?

print("-----")

string_6 =  """ "\\" """
print(string_6)  # Und hier?
```

Sequenzen von 3 Anführungszeichen (jeweils " oder ') können zum Markieren von Strings benutzt werden.

In [ ]:

```
string_7 = 'Auch ein String'
print(string_7)

print("-----")

string_8 = """Hier auch"""
print(string_8)

print("-----")

string_9 = ''' Auch ein String '''
print(string_9)
```

In [ ]:

```
string_10 = """"""hallo""""""
print(string_10)  # Was wird hier ausgegeben? Und warum?
```

Methoden

Methoden sind datentypspezifische Funktionen. Deshalb treten Fehler auf, wenn z.B. eine Stringmethode auf eine Liste angewendet wird.

In [ ]:

```
string_1 = "Eine Zeichenkette aus (fast) beliebigen Zeichen"

string_1.lower()
print(string_1)

print("-----")

string_11 = string_1.lower()  # Kopie in neuer Variable aufbewahren
print(string_11)
```

- Strings sind ein unveränderlicher Datentyp.
- String-Methoden erstellen eine Kopie des Ursprungs-Strings.
- Der ursprüngliche String wird nicht verändert.
- Deshalb muss der Output einer Methode einer Variable zugewiesen werden, um nach dem Funktionsaufruf darauf zugreifen zu können.

Weitere String-Methoden: `upper()`, `replace()`, `count()` ... Für weitere Methoden siehe Vorlesungsskript (Notebook vom 15.10.2019)

Integers

Integers sind positive oder negative ganze Zahlen.

In [ ]:

```
zahl_1 = 1
print(zahl_1)

zahl_2 = 2
print(zahl_2)

zahl_3 = 3
print(zahl_3)

print((zahl_1 + zahl_2) * zahl_3)
```

- Befehle werden "von innen nach außen" ausgeführt, d.h.
- Eingebettete Operationen werden zuerst verarbeitet.

Floats - Gleitkommazahlen

Gleitkommazahlen sind eine Annäherung an die reellen Zahlen. Warum nur eine Annäherung? Mehr Informationen finden Sie hier: https://www.youtube.com/watch?v=PZRI1IfStY0 (https://www.youtube.com/watch?v=PZRI1IfStY0)

In [ ]:

```
zahl_4 = 0.1
print(type(zahl_4))

zahl_5 = 3*zahl_4
print(zahl_5)

print(zahl_5 == 0.3)  # Oh weh!

print("-----")

print(zahl_5 < 0.3)
print(zahl_5 > 0.3)
```

- Es werden nur Annäherungen an tatsächliche Zahlenwerte gespeichert.
- Rundungsfehler passieren!

Vergleichsoperatoren

Mit Vergleichsoperatoren können Sie testen, ob die folgenden Relationen zwischen zwei Werten bestehen:

- `==` : Gleichheit
- `!=` : Ungleichheit
- `<` , `>` : Kleiner als, Größer als
- `<=` , `>=` : Kleiner oder gleich, Größer oder gleich

Neben Zahlen können auch andere Werte so miteinander verglichen werden, z.B. Strings, Listen.
Vergleichen Sie immer nur Objekte des gleichen Typs miteinander!

In [ ]:

```
print(3.2 <= 5.8)
print(4 >= 4)

print("ABC" != "CBA")

print("ABC" > "ACB")   # Was bedeutet "größer als" bei Strings?
print("ABC" < "ACB")
```

Listen

- Eckige Klammern `[ ... ]` begrenzen Listen.
- Listen können leer oder mit Inhalt erzeugt werden (vgl. liste_0 und liste_1)
- Werte jedes beliebigen Datentyps können als Element in einer Liste gespeichert werden
- Elemente einer Liste können verschiedenen Typs sein. Auch komplexe Typen (z.B. verschachtelte Listen) sind möglich!

In [ ]:

```
liste_0 = []
liste_1 = ["item_1", "item_2", "item_3", "item_4"]

print(liste_0)
print(liste_1)

print("-----")

print(liste_1[2])
print(liste_1[-1])
```

- Listenelemente können per Index aufgerufen werden.
- die Indizierung beginnt mit 0
- Mit negativen Indizes kann vom Ende der Liste aus zurück gezählt werden
- Das Element am Schluss der Liste hat bei dieser Zählweise immer den Index `-1`

Slicing

- Mit der Slicing-Schreibweise kann man auf Teile von Listen zugreifen
- `[:]` slicet die gesamte Liste
- Der erste Indexwert bezeichnet den Startindex (inklusiv) des Slice.
- Der zweite Wert bezeichnet das Ende (exklusiv) des Slice.
- Über negative Indizes als Endwert können die n letzten Elemente ausgeschlossen werden.

In [ ]:

```
liste_1 = ["item_1", "item_2", "item_3", "item_4"]

liste_2 = liste_1[0:2]  # Wieviele Einträge hat die Liste?
print(liste_2)

print("-----")

liste_3 = liste_1[0:-2]  # Wie liste_1 abzgl. der letzten beiden Elemente
print(liste_3)
```

Veränderbarkeit von Listen

Im nächsten Codeblock werden zwei Variablen gleichgesetzt, Sie zeigen damit auf die **selbe** Liste.

In [ ]:

```
liste_1 = ["item_1", "item_2", "item_3", "item_4"]
liste_4 = liste_1

print(liste_4)
```

Listen sind ein **veränderlicher** Datentyp (Mutability). Im Anschluss werden Listenelemente einer bereits vorhandenen Liste geändert...

In [ ]:

```
liste_1 = ["item_1", "item_2", "item_3", "item_4"]
liste_4 = liste_1

#####################################################

liste_4[1] = "Einfach mal was anderes machen."
liste_4[3] = "Und hier auch. Jetzt auch mit Ümläutön."

print(liste_4)
# print(liste_1)  # Was wird hier ausgegeben?
```

- Durch die Gleichsetzung im vorangegangenen Codeblock aus gleich!
- Die ursprüngliche Liste `liste_1` ist damit also verändert worden, die ursprünglichen Werte sind weg!
- Listen sind "mutable", veränderlich. (Mutability: https://docs.python.org/3/reference/datamodel.html (https://docs.python.org/3/reference/datamodel.html))
- Falls Sie tatsächlich eine Liste kopieren möchten, können Sie das z.B. mit Slicing tun.
- Nun bleibt `liste_1` erhalten und `liste_4` enthält alle Elemente, die auch in `liste_1` enthalten waren
- Verändern wir jetzt `liste_4` , wird `liste_1` nicht mit verändert.

In [ ]:

```
liste_1 = ["item_1", "item_2", "item_3", "item_4"]
liste_4 = liste_1

################################################

liste_4 = liste_1[:]
liste_4[3] = "Mit ohne Umlaute"

print("Liste 4:")
print(liste_4)
print("Liste 1:")
print(liste_1)
```

Dictionarys

- Genau wie Listen können Dictionarys entweder leer oder mit Inhalten initialisiert werden.
- Die Elemente von Dictionarys sind immer **Paare von Schlüsseln und Werten**.
- Schlüssel müssen eindeutig sein und müssen einen unveränderlichen Datentyp haben.
- Die Werte können beliebige Typen haben.

In [ ]:

```
empty_dict = {}
filled_dict = {"a": 0, "e": 0, "i": 0, "o": 0, "u": 0}

print(empty_dict)
print(filled_dict)
```

Mit der "Index"-Schreibweise können in Dictionarys

1. neue Schlüssel-Wert-Paare zu einem Dictionary hinzugefügt und auch
2. der Wert eines bereits im Dictionary enthaltenen Schlüssels geändert werden

In [ ]:

```
empty_dict = {}
filled_dict = {"a": 0, "e": 0, "i": 0, "o": 0, "u": 0}

############################################################

empty_dict["a"] = 1
print(empty_dict)

filled_dict["a"] = 1
print(filled_dict)
```

- Um Werte bereits vorhandener Schlüssel **schrittweise zu verändern** (z.B. beim Zählen von Vokalen), können folgende Schreibweisen angewendet werden.
- Diese Schreibweise bedeutet, dass der Interpreter erst nachschaut, welchen Wert der Schlüssel "i" im Dictionary bisher hatte, zum bisherigen Wert 1 addiert und den neuen Wert für diesen Schlüssel ins Dictionary schreibt.
- Natürlich können Sie auch andere Schrittweiten als 1 bestimmen
- Neben dem `+=` -Operator stehen auch noch diese weiteren Operatoren zur Verfügung: `-=`, `*=`, `/=`.

In [ ]:

```
empty_dict = {"a": 1}
filled_dict = {"a": 1, "e": 0, "i": 0, "o": 0, "u": 0}

filled_dict["i"] = filled_dict["i"] + 1
print(filled_dict)

filled_dict["i"] += 2
print(filled_dict)

print("-----")

# empty_dict["e"] += 1  # Was läuft hier schief?
# print(empty_dict)

# print(empty_dict)
# empty_dict["a"] =+ 1  # Und was läuft hier schief?
# print(empty_dict)     # Achten Sie auf den Operator...
```

Die folgenden Dictionary-Methoden liefern eine Sequenz aller Schlüssel bzw. aller Werte eines Dictionarys. Vorsicht: Die Reihenfolgen der Listenelemente innerhalb der beiden Listen müssen nicht miteinander korrespondieren!
Ob die Reihenfolge eingehalten wird, ist von der Python-Version abhängig. Ab Python 3.7 bleibt die Reihenfolge erhalten.

In [ ]:

```
filled_dict = {'a': 1, 'e': 0, 'i': 3, 'o': 0, 'u': 0}

##########################################################

schluessel = filled_dict.keys()     # alle Schlüssel des Dictionarys
print(schluessel)

werte = filled_dict.values()        # alle Werte des Dictionarys
print(werte)
```

Bedingungen

- Entscheidungen im Code treffen
- Je nachdem, wie die Daten aussehen, bestimmte Teile des Programms ausführen oder überspringen
- mehrere `elif` -Bedingungen und genau ein `else` -Statement sind möglich - aber jeweils nicht obligatorisch

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

print(liste_4)

print("-----")

if len(liste_4[1]) > 8:
    print(liste_4[1])
```

- Das `else` -Statement kann sinnvoll sein, um sicher zu gehen, dass der gesamte `if` -Block ausführt wurde.
- Falls die Bedingung nicht erfüllt ist, ignoriert der Interpreter die Befehle im `if` -Block und springt direkt in den `else` -Block.
- Bedingungen sind häufig in Schleifen eingebettet, um Entscheidungen zu automatisieren.

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

if len(liste_4[0]) > 8:
    print(liste_4[0])
else:
    print("Hier gibt es nichts zu sehen.")
```

`For` -Schleifen

- Wiederholte Anwendung von Code auf Daten in einem iterierbaren (Container-)Datentyp (Listen, Dictionarys)
- Die Funktion `range()` stellt eine iterierbare Sequenz von Ganzzahlen bereit:
  - Wird die Funktion mit nur einem Parameter `range(n)` aufgerufen, so wird die Sequenz 0, 1, ... , n-1 bereitgestellt.
  - Mit zwei Parametern `range(n,m)` kann man Beginn n (inklusiv) und Ende m (exklusiv) der Sequenz bestimmen: n , n+1, ..., m-1

In [ ]:

```
for i in range(-3, 6):
    print(i)
```

Was passiert in dem Code-Block oben?

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

for i in range(3):
    print(liste_4[i])
```

Was passiert in dem Code-Block oben?

In [ ]:

```
for i in range(0, 2):
    print(liste_4[i])
```

Was passiert in dem Code-Block oben?

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

for x in liste_4:
    print(x)
```

- Bisher haben wir durch ranges iteriert, also durch Sequenzen von Ganzzahlen. Da Listen aber auch Sequenzen sind, können wir direkt durch die Elemente von Listen iterieren.
- Im folgenden hat die Variable x in jedem neuen Schleifendurchlauf den Wert des jeweils aktuellen Listenelements.

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

##################################################

for i in range(len(liste_4)):
    print(liste_4[i])
```

- Hier wird ebenfalls durch die Elemente der liste_4 iteriert, aber mithilfe des Index. Der Effekt des print()-Aufrufs ist gleich.
- Allerdings haben wir in dieser Variante auch Zugriff auf den aktuellen Index. Das wäre in der Variante mit "for x in liste_4" nicht so einfach möglich gewesen.

- Konventionelle Namen für "Laufvariablen":
  - für Integers: **i**,
  - für Zeichen (characters) in Strings: **c**,
  - für Wörter in Wortlisten: **w**

Wie genau Sie die Variable nennen, spielt keine Rolle - es geht um Lesbarkeit und Verständlichkeit des Codes.

Bedingungen und For-Schleifen kombinieren

Stellen Sie sich vor, wir sind interessiert an allen Zahlen im Zahlenraum bis 50, die die Ziffern 3 und/oder 5 enthalten, und möchten uns diese ausgeben lassen.

In [ ]:

```
for i in range(1, 51):
    string_i = str(i)
    if "5" in string_i and "3" in string_i:
        print("Die Zahl " + string_i + " enthält die Ziffern 3 und 5.")
    elif "5" in string_i:
        print("Die Zahl " + string_i + " enthält die Ziffer 5.")
    elif "3" in string_i:
        print("Die Zahl " + string_i + " enthält die Ziffer 3.")
    else:
        print("Die Zahl " + string_i + " ist hier uninteressant. Die nächste bitte!")
```

- Wir iterieren durch die range 1 .. 50.
- Wir erzeugen eine Stringversion der Zahl.
- Sind die 5 und die 3 beide in der Stringversion der Zahl enthalten?
- falls ja: Ausgabe, dass beide enthalten sind.
- falls nein: weitere Bedingung prüfen - ist die 5 in der Stringversion der Zahl enthalten?
- falls ja: Ausgabe, dass die 5 enthalten ist.
- falls nein: weitere Bedingung prüfen - ist die 3 in der Stringversion der Zahl enthalten?
- falls ja: Ausgabe, dass die 3 enthalten ist.
- falls alle Bedingungen nicht erfüllt waren...
- ... Ausgabe, dass die aktuelle Zahl uninteressant ist.

Funktionen

- Code-Einheiten, die einen bestimmten Zweck erfüllen und mehrmals in einem Programm benutzt werden, können als Funktionen definiert werden.
- Funktionen helfen dabei, den Programm-Code übersichtlicher zu gestalten
- Sie helfen bei der Fehlervermeidung

Häufig möchte man Inhalte von komplexen Datentypen, wie Listen oder Dictionarys, ansehnlich ausgeben:

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

# Im Programm:
for item in liste_4:
    print(item)

print("-----")

# als Funktion:
def print_list(die_liste):  # Funktionsdefinitions: def als Startsymbol, Funktionsname
                            # und in Klammern die Funktionsparameter (optional).
    for item in die_liste:  # Der Funktionskörper mit den Befehlen ist eingerückt.
        print(item)         # Mehrere Einrückungen sind möglich bzw. nötig.
    return                  # Rückgabewert definieren (optional)

print_list(liste_4)
```

- Eine Funktion kann beliebig viele Parameter haben (auch 0).
- Als Parameter/Argumente wollen wir die Werte übergeben, die nötig sind, um das Ergebnis der Funktion zu ermitteln.
- Innerhalb der Funktion ist der übergebene Wert unter dem Parameternamen `die_liste` verfügbar.
- Die Funktion hatte keinen Rückgabewert, weil `print()` immer sofort ausgeführt wird.

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

#####################################################

# mit noch schönerem Output und mehreren Parametern:
def print_list_nicer(die_liste, der_ausgabetext):
    counter = 1  # Zählervariable
    for item in die_liste:
        print(der_ausgabetext + " " +  str(counter) + ". " + item)
        counter += 1

print_list_nicer(liste_4, "Uiuiui das")
```

- Das `return` -Statement ohne zusätzliche Angabe ist eine Kurzform von `return None` . Das bedeutet, dass die Funktion ein "leeres" Ergebnis zurückliefert (Datentyp `None`).
- Es ist bedeutungsgleich, ob wir ein leeres `return` -Statement schreiben oder `return` einfach vollständig weglassen. Für die bessere Lesbarkeit empfehlen wir Ihnen, immer `return` zu schreiben, auch wenn Sie keinen Rückgabewert verwenden möchten!

In [ ]:

```
liste_4  = ["item_1", "item_2", "item_3", "item_4"]

##################################################

def alles_zurueckgeben(eingabe_liste):
    result = ""
    for x in eingabe_liste:
        result = result + x + "\n"
    return result

rueckgabe = alles_zurueckgeben(liste_4)
print(rueckgabe)
```

- Die Funktion `alles_zurueckgeben(...)` enthält kein `print()` . Stattdessen wird ein String namens `result` nach und nach um zusätzliche Informationen erweitert. Zum Schluss wird der fertige String als Rückgabewert weitergereicht.
- Wenn `alles_zurueckgeben(...)` aufgerufen wird, sehen wir zunächst nichts. Wir können aber den Ergebnis-String speichern, weil er als Ergebnis von der Funktion zurückgegeben wird - oder wir betten den Funktionsaufruf in ein `print()` -Statement ein.

Strategien zum Schreiben von Funktionen

- Wie soll die Funktion heißen? Der Name soll möglichst aussagekräftig sein und die Aufgabe beschreiben, die von der Funktion erledigt wird. Beispiel:

```
def reverse_string
```

- Welche Daten soll die Funktion als Ausgangspunkt nehmen? Diese Werte können als Parameter übergeben werden. Beispiel:

```
def reverse_string(original_string):
```

- Welcher Wert soll am Ende der Funktion zurückgegeben werden? Sie können entweder den eingegebenen Wert überschreiben oder eine neue Variable anlegen (letzteres ist lesbarer). Beispiel:

```
def reverse_string(original_string):
  reversed_string = ""     # hier kümmern wir uns gleich drum
  return reversed_string
```

- Und schließlich: Wie soll die Aufgabe der Funktion Schritt für Schritt gelöst werden? Code im Funktionskörper ergänzen.

```
def reverse_string(original_string):
  reversed_string = original_string[::-1]  # Geheimtipp zum Umdrehen von Str
ings in Python!
  return reversed_string
```

- Bei sehr übersichtlichen Funktionen dürfen Sie komplexe return-Statements verwenden. Dabei wird die Operation, die den Rückgabewert ermittelt, direkt hinter `return` geschrieben. Tun Sie das nur, wenn Sie sich gut in Ihrem eigenen Code auskennen!

```
def reverse_string(original_string):
  return original_string[::-1]   # wir brauchen keine neue Variable mehr!
```

- Ganz am Schluss ist es eine gute Idee, mit mehreren Eingabewerten zu prüfen, ob Ihre Funktion auch richtig funktioniert.

```
print(reverse_string("Pythonkurs"))
print(reverse_string("Python, aber rückwärts"))
```

Module

- Module fügen der Python-Grundausstattung zusätzliche Funktionen hinzu
- Sie sollen in der Regel zu Beginn des Programms importiert werden. Das liegt daran, dass beim Importieren der Zustand des Interpreters verändert wird. Wenn das import- Statement z.B. in einer Funktionsdefinition steht, ist es sehr schwer, zu sehen, ob der Interpreter zu einem beliebigen Zeitpunkt im Programm schon einen erfolgreichen import ausgeführt hat oder nicht. Deshalb importieren wir immer am Anfang und auf der obersten Code-Ebene (also nicht in Schleifen, Funktionen oder Bedingungen eingebettet).
- Modul-Funktionen werden wie folgt aufgerufen: `<modulname>.<modul-funktion>`
- Wir haben bisher die Module random (Woche 3) und datetime (Woche 4) kennengelernt.

Das Modul random

Der Interpreter liest das Modul random ein und merkt sich alle Funktionen, die darin definiert werden.

- In unserem Programm können wir jetzt die Funktion `randint(...)` aus dem Modul `random` aufrufen.
- Der liste_5 wurde eine zufällige Zahl zwischen 1 und 100 angehängt. Bei der Ausgabe können wir sehen, welche Zahl das war.

In [ ]:

```
import random

liste_5 = []
for i in range(10):
    zufallszahl = random.randint(1,100)
    liste_5.append(zufallszahl)

print(liste_5)
```

Dateien schreiben, lesen und erweitern

Schreiben

1. Wir definieren wir eine neue Variable vom Typ String, in der wir den Dateinamen (mit Dateiendung) angeben. Die Dateiendung für Textdaten ist `.txt` . Wir werden aber später noch andere Endungen benutzen.
2. Die Datei wird mit der "with open()"-Syntax bereitgetellt:
   - der erste Parameter von `open()` ist der Dateiname
   - als nächstes geben wir mit `"w"` für **write** an, dass wir in eine Datei schreiben wollen
   - zusätzlich können wir die Zeichencodierung mittels `encoding="utf8"` festlegen. UTF-8 ist eine Zeichencodierung die mit Umlauten und vielen, vielen Zeichen mehr gut umgehen kann.
   - mit `as` gefolgt von einem neuem Variablennamen legen wir fest, wie wir die Datei, in die wir schreiben wollen, in unserem Programm bezeichnen.
3. Mit dem `print()` -Befehl können wir ausgaben in Dateien umleiten. Dazu müssen wir in den Klammern von `print()` außer dem Inhalt, der ausgegeben werden soll, auch den zusätzlichen Parameter `file=<Name der Datei mit Dateieindung>` angeben!

In [ ]:

```
dateiname = "texttesttext.txt"

with open(dateiname, "w", encoding="utf8") as outfile:
    print("Hat's was geschrieben?", file=outfile)
    # print(outfile.read())  # Man kann nix Lesen im Schreibmodus

print("Geschafft! Die Datei '" + dateiname + "' wurde erstellt.")
```

Lesen

1. Wie oben brauchen wir erstmal den Dateinamen einer bekannten Datei.
2. Wir benutzen die gleiche Syntax wie oben um die Datei zu öffnen:
    - Erster Parameter: Dateiname
    - Der Unterschied liegt im **zweiten Parameter: "r" für read**
    - Wir übernehmen auch `encoding="utf8"` als dritten Parameter
    - Nun muss wieder per `as` ein Dateibezeichner für den Verlauf des Programms festgelegt werden.

Nun können wir im Körper des `open()` -Statements mit `for` durch die Zeilen der Datei iterieren.

Wenn wir im "r"-Modus versuchen, Inhalte in die Datei zu schreiben, schlägt das fehl.

In [ ]:

```
pfad_u_dateiname = "c:/Users/Ben/Desktop/superstition.txt"

with open(pfad_u_dateiname, "r", encoding="utf8") as infile:
    for line in infile:
        print(line)
```

Statt durch die Zeilen zu iterieren, können wir uns auch die ganze Datei auf einmal anzeigen lassen.

Das Ergebnis der Operation `infile.read()` ist ein String. Hier wird dieser String gedruckt, aber wir könnten ihn auch in einer Variable speichern und dann unsere gewohnten Stringmethoden darauf anwenden.

Bei sehr großen Dateien ist es empfehlenswert, die Inhalte zeilenweise zu verarbeiten.

In [ ]:

```
dateiname = "texttesttext.txt"

with open(dateiname, "r", encoding="utf8") as infile:
    dateiinhalt = infile.read()
    print(dateiinhalt)
    # print("Täst, Töst", file=infile)  # Man kann nix schreiben im Lesemodus!
```

Append Mode

Wenn wir Dateien im Schreibmodus öffnen, werden diese Dateien neuerstellt bzw. der Inhalt bestehenden Dateien vollständig überschrieben, sodass evtl. vorher vorhandener Inhalt verloren geht.

Um Inhalt zu bereits bestehenden Dateien hinzuzufügen, können wir Dateien im Append Mode (Anfügemodus) öffnen. Dazu geben wir als zweiten Parameter des `open()` -Befehl `"a"` an.

Neues Material wird dann am Ende der Datei hinzugefügt.

In [ ]:

```
pfad_u_dateiname = "c:/Users/Ben/Desktop/superstition.txt"

weitere_zeile = "Oh no! Nah, nah, nah!"

with open(pfad_u_dateiname, "a", encoding="utf8") as append_file:
    print(weitere_zeile, file=append_file)
    # print(append_file.read())  # Man kann nix lesen im Anfügemodus!
```

User-Input während eines Programms

Jetzt, wo wir Funktionen geschrieben haben, können wir den User entscheiden lassen, auf welche Eingabewerte eine Funktion angewendet werden soll. Dazu verwenden wir das Kommando `input()` . Zuerst ein Beispiel für die grundlegende Verwendung von `input()` :

In [ ]:

```
original_word = input("Bitte ein Wort eingeben! ")
print("Sie haben das Wort \"" + original_word + "\" eingegeben.")
```

Der String in den Klammern beim Aufruf von `input` dient dazu, den User zu informieren, dass eine Eingabe erwartet wird.

Jetzt definieren wir unsere String-Umkehrungsfunktion von oben noch einmal und lassen uns dann drei eingegebene Worte nacheinander umdrehen:

In [ ]:

```
# Funktionsdefinition:
def reverse_string(original_string):
    reversed_string = original_string[::-1]
    return reversed_string

# Jetzt fragen wir nach drei Wörtern.
for i in range(3):
    input_word = input("Welches Wort soll umgedreht werden? ")
    # Das Ergebnis ist ein String, der von der Funktion mit
    # return zurückgegeben wird. Wir können ihn hier ausgeben.
    print("Ergebnis: " + reverse_string(input_word))
```

Falls wir noch Zeit haben, können wir hier sogar noch Bedingungen einbauen!

In [ ]:

```
# Funktionsdefinition:
def reverse_string(original_string):
    reversed_string = original_string[::-1]
    return reversed_string

# Jetzt fragen wir nach drei Wörtern.
for i in range(3):
    input_word = input("Welches Wort soll umgedreht werden? ")

    # Nur Wörter mit mehr als 1 Zeichen sollen umgedreht werden.
    if len(input_word) > 1:
        # Das Ergebnis ist ein String, der von der Funktion mit
        # return zurückgegeben wird. Wir können ihn hier ausgeben.
        print("Ergebnis: " + reverse_string(input_word))
    else:    # Bei kürzeren Wörtern...
        # ... sind wir faul und geben das Originalwort zurück.
        print("Ergebnis: " + input_word)
```