Bisherige Themen

Letzte Woche haben wir folgende Themen behandelt:

- Wie Sie mit Python "Hallo Welt" ausgeben können
- Wie Sie in VSCode Einstellungen ändern und Programme schreiben
- Wie Sie in VSCode Programme ausführen
- Wie Sie in Jupyter Notebooks Codebeispiele bearbeiten und ausführen können
- Dass bei verschachtelten Befehlen in Pythoncode immer zuerst der innere Befehl interpretiert wird, dessen Ergebnis dann für die Interpretation der äußeren Befehle verwendet wird (vgl. `print(3 + 5)`)
- Wie Kommentare in Pythoncode markiert werden
- Wie der Python-Interpreter, VSCode und das Terminal miteinander zusammenhängen
- Wie Sie in Python Variablen erzeugen und Werte in Variablen speichern können
- Wie Sie den Datentyp eines Werts ermitteln können

Falls Sie Fragen zu diesen Themen oder zu den Übungsaufgaben haben, sprechen Sie uns bitte an!

Und so geht es weiter:

Editor-Trick des Tages: Multicursor

In VSCode können Sie mehrere Zeilen auf einmal bearbeiten! Klicken Sie dazu an eine beliebige Stelle im Code und halten Sie dann die `Alt` -Taste gedrückt, während Sie an weiteren Stellen klicken.

Oder Sie klicken die erste Stelle an, drücken dann `Alt + Shift` und klicken gleichzeitig an eine zweite Stelle; die Cursors werden jetzt untereinander überall zwischen der ersten und der zweiten Position platziert.

Auch die Command Palette kann zusätzliche Cursors setzen. Drücken Sie `Ctrl + Shift + P` und tippen Sie `add cursor` ein, um dann eine der verfügbaren Optionen auszuwählen.

Probieren Sie es aus: Kopieren Sie den folgenden Text in den Editor und ergänzen Sie mit einem Multicursor in allen Zeilen die vollständige Jahreszahl ("2019" statt "'19").

```
08.10.'19
15.10.'19
22.10.'19
29.10.'19
05.11.'19
12.11.'19
19.11.'19
26.11.'19
03.12.'19
10.12.'19
```

Achtung: Welcher dieser Wege an Ihrem eigenen Gerät funktioniert, hängt u.a. vom Betriebssystem ab. Finden Sie heraus, welches Kommando bei Ihnen das richtige ist!

`for` : Schleifen für Sequenzen

Um auf die Elemente in einer Liste nacheinander zuzugreifen, schreiben wir eine **Schleife** (englisch: *loop*). Alle Befehle, die in der Schleife stehen, werden schrittweise immer wieder für jedes einzelne Element der Liste ausgeführt. Nach dem letzten Element wird die Schleife verlassen und die Befehle unterhalb der Schleife werden ausgeführt.

Klicken Sie in das Kästchen unten und drücken Sie `Ctrl + Enter` , um den Code auszuführen:

In [ ]:

```
# Liste definieren und Inhalt in einer Variable speichern
zahlen_bis_fuenf = [0, 1, 2, 3, 4, 5]

print("Start")                  # Das Wort "Start" ausgeben

for zahl in zahlen_bis_fuenf:   # Diese Zeile leitet die Schleife ein.
    print("Aktuelle Zahl:")     # Erster Befehl in der Schleife
    print(zahl)                 # Zweiter Befehl in der Schleife

# Befehle ab hier gehören nicht mehr zur Schleife.

print("Ende")                   # Das Wort "Ende" ausgeben
```

Das Konstrukt, das wir oben verwendet haben, heißt **for-Schleife**. Diese Schleifen entsprechen immer dem folgenden Muster:

```
for <element> in <collection>:
    <befehl1>
    <befehl2> # (optional)
    <befehl3> # (optional)
    <...>
```

Die Zeile, die mit `for` beginnt und mit `:` endet, heißt **Kopf** der Schleife. Die Zeilen, die darunter stehen und weiter **eingerückt** sind als der Kopf, heißen **Körper**. Befehle, die im Körper einer Schleife stehen, müssen immer gleich weit eingerückt sein (typischerweise 4 Leerzeichen pro Einrückungsebene).

Auf Englisch heißt Einrückung **indentation** (als Verb: **to indent**).

Befehle, die die gleiche Einrückung haben wie der Kopf der Schleife, werden außerhalb der Schleife ausgeführt, also nur einmal und nur vor/nach der Ausführung der Schleife.

Es ist dabei egal, ob die `collection` (beispielsweise eine Liste) als Variable übergeben wird oder als konkreter Wert. Sie können das Beispiel von oben also folgendermaßen umschreiben, ohne die Funktionalität des Programms zu verändern:

In [ ]:

```
#####################################
# +++ erste Lösung (siehe oben) +++ #
#####################################
zahlen_bis_fuenf = [0, 1, 2, 3, 4, 5]

print("Start")                  # Das Wort "Start" ausgeben

for zahl in zahlen_bis_fuenf:   # Diese Zeile leitet die Schleife ein.
    print("Aktuelle Zahl:")     # Erster Befehl in der Schleife
    print(zahl)                 # Zweiter Befehl in der Schleife

print("Ende")                   # Das Wort "Ende" ausgeben
```

In [ ]:

```
print("Start")                  # Das Wort "Start" ausgeben

for zahl in [1, 2, 3, 4, 5]:    # Diese Zeile leitet die Schleife ein.
    print("Aktuelle Zahl:")     # Erster Befehl in der Schleife
    print(zahl)                 # Zweiter Befehl in der Schleife

print("Ende")                   # Das Wort "Ende" ausgeben
```

Das `element` bekommt von Ihnen einen Namen, den Sie im Kopf der Schleife angeben. Im Beispiel oben lautet der Name `zahl`. Das jeweils aktuelle Element ist dann im Körper der Schleife unter diesem Variablennamen abrufbar. Deshalb wird im ersten Schleifendurchlauf die Zahl 1 ausgegeben, im zweiten die Zahl 2 usw.

Nachdem die Schleife **beendet** ist, behält die Variable den letzten zugewiesenen Wert.

Tipp: Da der Name des Elements sich pro Schleifendurchlauf auf genau ein Element bezieht, wählen wir typischerweise einen Namen im **Singular** (wie "zahl"). Der Name der Variable, in der die Liste gespeichert ist, steht hingegen im Plural (wie "zahlen_bis_zehn"). Wenn Sie sich an diese Konvention halten, ist Ihr Code gut lesbar, weil Kollektionen (wie Listen) und individuelle Elemente auf den ersten Blick unterschieden werden können.

Fortlaufende Listen von Zahlen kann Python übrigens für uns erzeugen, ohne dass wir jede Zahl aufschreiben müssen. Das ist nützlich, wenn wir eine lange Liste brauchen (z.B. Zahlen bis 100) oder die Anzahl der Schleifendurchläufe durch andere Werte im Programm bestimmt werden soll.

Eine weitere Benennungskonvention ist, dass wir in Schleifen wie der hier folgenden das aktuelle Element mit dem Buchstaben `i` bezeichnen. Das `i` steht für *Integer* oder *Index*.

In [ ]:

```
# Welcher Wert versteckt sich hinter range(10)?
for i in range(10):
    print(i)
```

In [ ]:

```
# range([min], max) erzeugt eine Kollektion, die alle Ganzzahlen
# zwischen min (inklusive; falls nicht angegeben, 0) und max (exklusive) enthält.
# Probieren Sie es aus!
for i in range(2,5):
    print(i)
```

Grundlegende Operationen

Wir haben jetzt also auf den aktuellen Wert in Variablen zugegriffen, um anzuzeigen, welchen Wert die Variable gerade hat. Für jeden Datentyp gibt es bestimmte Operationen, die zu neuen Werten führen. Einige Beispiele:

In [ ]:

```
print(1 + 4)
print("Erst das Wasser, " + "dann die Säure!")
print([1, 2, 3] + [4, 5, 6])
```

Probleme gibt es, wenn wir versuchen, Werte verschiedener Typen miteinander zu verknüpfen:

In [ ]:

```
print(333 + ": bei Issos Keilerei")
```

Es gibt mehrere Möglichkeiten, die gewünschte Ausgabe (Zeichenkette mit den angegebenen Zahlen am Anfang) zu erreichen. Wir verwenden hier im Kurs vorerst die folgende Methode:

In [ ]:

```
print(str(333) + ": bei Issos Keilerei")
```

Auch die anderen oben aufgelisteten Datentypen können mit dem Befehl `str()` explizit in Strings umgewandelt werden.

Eine Operation, mit der wir zwei Werte vergleichen können, ist `==` . Damit wird geprüft, ob die beiden Elemente, die verglichen werden, den gleichen Wert haben. Das Ergebnis ist vom Typ `bool` .

In [ ]:

```
print(4 == 4)                 # genau der gleiche Wert
print("words" == "Words")     # Groß- und Kleinschreibung wird unterschieden
print(1 == "1")               # verschiedene Datentypen
```

Genau das Gegenteil macht der Operator `!=` , der prüft, ob zwei Werte *ungleich* sind. Außerdem gibt es die Operatoren `>` , `<` , `>=` (größer oder gleich) und `<=` (kleiner oder gleich).

In [ ]:

```
print(4 != 4)                 # genau der gleiche Wert
print("words" != "Words")     # Großschreibung ist wichtig
print(1 != "1")               # verschiedene Datentypen
print(3 > 3)
print(4 <= 4)
```

Werte vom Typ `bool` können miteinander verknüpft werden, und zwar mit den Operatoren `and` , `or` , `not` .

Aufgabe

1. Können Sie vorhersagen, was der Output der folgenden Befehle ist? Sie können die Zellen mit `Ctrl + Enter` ausführen, um zu prüfen, ob Sie richtig lagen.

In [ ]:

```
a = "a"
b = "b"
print(a == b)
```

In [ ]:

```
a = "a"
b = "b"
print(a = b)
```

In [ ]:

```
print(True and not False)
```

In [ ]:

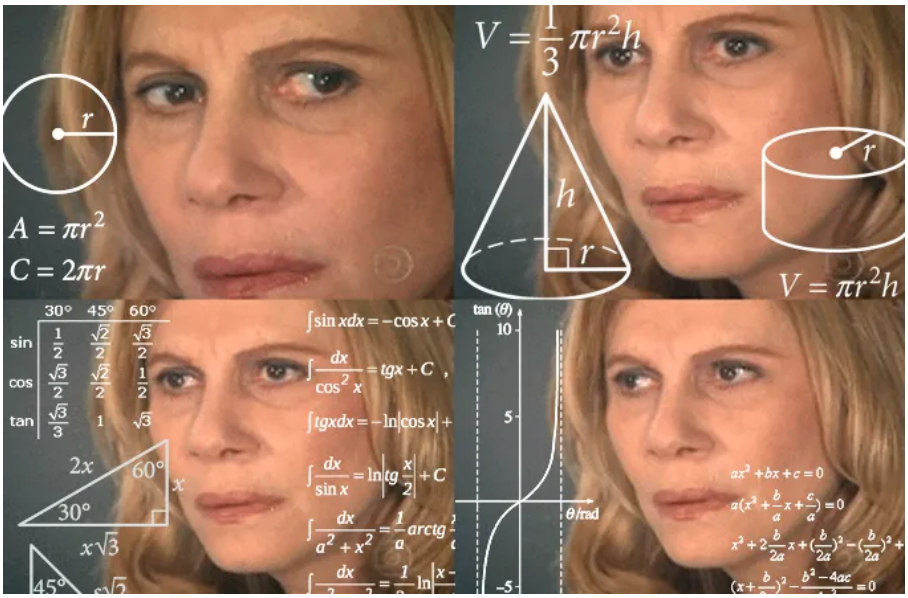
```
print(False or True)
```

Aufgabe (optional)

1. Strings können übrigens auch mit den Operatoren < und > verglichen werden. Können Sie experimentell herausfinden, welche Eigenschaft der Strings dabei geprüft wird? Vergleichen Sie so viele Strings, bis Sie eine Hypothese gefunden haben.

In [ ]:

```
# Ihre String-Vergleichstests
```



Bedingungen: `if`, `elif`, `else`

Wahrheitswerte und Vergleiche sind ein wichtiger Bestandteil unseres Werkzeugkastens, da sie uns helfen, Programmabläufe zu steuern. Wenn bestimmte Bedingungen erfüllt sind, soll eine Reihe von Befehlen ausgeführt werden; wenn andere Bedingungen erfüllt sind, sollen andere Befehle ausgeführt werden. Die **Syntax** für diese Steuerung sieht wie folgt aus:

In [ ]:

```
# Startwerte für unsere Variablen festlegen:
heizung_an = False
temperaturen = [24, 23, 19, 20, 17, 24, 14]

# Die leeren Zeilen haben keine Bedeutung und
# dienen nur der besseren Lesbarkeit :)

for temperatur in temperaturen:
    if temperatur <= 19:          # bei Temperaturen unter 19°C...
        heizung_an = True         # ... schalten wir die Heizung an

    elif 19 < temperatur < 21:    # bei Temperaturen zwischen 19°C und 21°C...
        heizung_an = True         # ... schalten wir die Heizung an

    else:                         # in jedem anderen Fall...
        heizung_an = False        # ... schalten wir die Heizung aus

    print("Temperatur: " + str(temperatur))
    print("Heizung an: " + str(heizung_an))
    print("---")
```

Mit `if`, `elif` und `else` können wir prüfen, ob die angegebenen Bedingungen erfüllt sind. Ein `if`-Block hat die folgende Struktur:

```
if <bedingung>:
    <befehl1>
    <befehl2 usw.>  # optional
elif <bedingung2>:  # optional
    <befehl3>
    <befehl4 usw.>  # optional
else:               # optional
    <befehl5>
    <befehl6 usw.>  # optional
```

Eine Bedingung kann alles sein, was sich zu einem Wahrheitswert umwandeln lässt - also praktisch alles. Wenn die Bedingung zum Wahrheitswert `False` führt, ist das `if` nicht zutreffend.

Wenn Sie nur den `if`-Block verwenden und die beiden anderen weglassen, wird bei Nichterfüllung der Bedingung einfach der ganze Block übersprungen.

Wenn Sie `if` in Kombination mit `elif` verwenden, wird bei Nichterfüllung der ersten Bedingung als nächstes die zweite Bedingung geprüft. Sie können beliebig viele `elif`-Blöcke aneinanderreihen.
Achtung: Sobald ein `if`- oder `elif`-Block ausgeführt wurde, werden die Bedingungen der verbleibenden `elif`-Blöcke nicht mehr geprüft und die Befehle dort übersprungen.

Wenn Sie `else` verwenden, werden bei Nichterfüllung der Bedingungen in den Blöcken davor stattdessen die Befehle im `else`-Block ausgeführt.

Aufgabe:

1. Verkürzen Sie das Heizungsprogramm, indem Sie einen der optionalen Blocks entfernen und die Bedingungen so umformen, dass das Verhalten gleich bleibt. Am einfachsten ist es, wenn Sie den gesamten Code in das Kästchen unten kopieren und dann anfangen, einzelne Zeilen zu ändern.
2. Ändern Sie die Ausgabe des Heizungsstatus so, dass nicht mehr `True` und `False` ausgegeben werden, sondern stattdessen "Heizung an" oder "Heizung aus", je nachdem, welchen Wert die Variable `heizung_an` gerade hat.

In [ ]:

```
# Ihre Lösung (1.)
```

In [ ]:

```
# Ihre Lösung (2.)
```

Operationen mit Strings

Da wir computerlinguistische Programme schreiben wollen, sind Zahlen, Zeichenketten und Listen die Datentypen, die wir ständig verwenden. Vor allem mit Strings wollen wir viel arbeiten.

Python stellt uns eine Reihe von Operationen zur Verfügung, die wir auf Strings anwenden können. Wir sprechen von **Methoden**.

Um eine Methode auf einen einzelnen String anzuwenden, schreiben wir beispielsweise:

```
"Hallo Welt".split()
```

Am Anfang steht der Originalstring (oder eine Variable, deren Inhalt ein String ist). Nach dem Punkt folgt der Name der Operation, die wir ausführen wollen. Das ist in diesem Fall `split`. Die öffnenden und schließenden Klammern sind dazu da, weitere Vorgaben für die Operation zu machen. Diese zusätzlichen Bedingungen in den Klammern werden **Parameter** genannt.

Hier also die allgemeine Form für solche Stringoperationen:

```
<string>.<operation>(<parameter1>, <parameter2 usw.>)
```

Jede Operation hat einen Rückgabetyp. Zum Beispiel ist die Rückgabe (das Ergebnis) von `"Hallo Welt".split()` eine Liste der Strings, die sich ergeben, wenn man den ursprünglichen String an allen Leerzeichen aufteilt:

In [ ]:

```
# Ausgabe: Typ von "Hallo Welt"
print(type("Hallo Welt"))

# Ausgabe: Ergebnis der Aufteilung von "Hallo Welt"
print("Hallo Welt".split())

# Ausgabe: Typ, den das Ergebnis der Aufteilung von "Hallo Welt" hat
print(type("Hallo Welt".split()))
```

Die optionalen Parameter für die Methode `split()` (https://docs.python.org/3.6/library/stdtypes.html#str.split) heißen `sep` und `maxsplit`. Damit können wir angeben, an welchem Substring der ursprüngliche String aufgeteilt werden soll und wie viele Teilungen erfolgen sollen. Wir könnten zum Beispiel einen String, der ein Datum enthält, an allen Punkten aufteilen:

In [ ]:

```
print("15.10.2019".split(sep="."))
```

Oder wir wollen den folgenden String am ersten `"-"` aufteilen, die anderen Vorkommen dieses Zeichens aber intakt lassen:

In [ ]:

```
print("2019-10-15".split(sep="-", maxsplit=1))
```

Die Parameter einer Methode haben eine vorgegebene Reihenfolge. In diesem Fall wird immer `sep` als erster Parameter erwartet, `maxsplit` als zweiter. Wir können den Namen der Parameter in diesem Fall weglassen, wenn wir die Parameter in der richtigen Reihenfolge einfügen:

In [ ]:

```
print("2019-10-15".split("-", 1))
```

Das geht allerdings nur, solange kein Parameter ausgelassen wird. Für `split` ist der Parameter `sep` optional, aber eventuell möchten wir trotzdem einen Wert für `maxsplit` angeben. Dann müssen wir den Namen des Parameters angeben.

So verwenden wir `split`, wenn wir am Standardseparator (dem Leerzeichen) splitten wollen, aber nur eine bestimmte Anzahl von Elementen brauchen:

In [ ]:

```
print("All that glitters is gold, only shooting stars break the mold".split(maxsplit = 5))
```

Nach dem ersten benannten Parameter müssen auch die folgenden Parameter benannt werden. Sonst schlägt der Befehl fehl, so wie hier:

In [ ]:

```
print("Hallo Welt".split(sep="l", 2))
```

Übrigens spricht nichts dagegen, längere Substrings als Wert für `sep` zu verwenden. Denken Sie nur daran, dass die Substrings in der Liste, die wir als Ergebnis bekommen, nicht mehr enthalten sind:

In [ ]:

```
print("Ich kenne die Weise, ich kenne den Text, ich kenn auch die Herren Verfasser".split(sep="enn"))
```

Fassen wir zusammen:

In [ ]:

```
print("Hallo Welt".split())                      # n Teilungen am Standardseparator

print("----------")

print("Hallo Welt".split("W"))                   # Teilung am angegebenen Separator
print("Hallo Welt".split(sep="W"))               # Teilung am angegebenen Separator

print("----------")

print("Hallo Welt".split(sep="l", maxsplit=2))   # 2 Teilungen am angegebenen Separator
print("Hallo Welt".split("l", 2))                # 2 Teilungen am angegebenen Separator
print("Hallo Welt".split("l", maxsplit=2))       # 2 Teilungen am angegebenen Separator

print("----------")

print("Hallo Welt".split(maxsplit=1))            # 1 Teilung am Standardseparator
```

Nachdem wir gesehen haben, wie man Strings mit Funktionen verarbeiten kann, hier einige wichtige Operationen, die wir auf Strings anwenden können. Beachten Sie, dass die Funktionen unterschiedliche Rückgabetypen haben.

Operation	Rückgabe
`s.split([sep, n])`	**Liste** aller Teilstrings von `s`, die durch den Separator voneinander getrennt sind. Wenn weder `sep` noch `n` angegeben werden, wird der String an jedem Leerzeichen geteilt. Mit `sep` kann man spezifizieren, an welchen Teilstrings der String zerteilt werden soll, und mit `n`, wieviele Zerteilungen erfolgen sollen.
`s.lower()`	**String:** Kopie von `s`, in der alle Buchstaben kleingeschrieben sind
`s.upper()`	**String:** Kopie von `s`, in der alle Buchstaben großgeschrieben sind
`s.islower()`	**Bool:** `True`, wenn alle Buchstaben in `s` kleingeschrieben sind; sonst `False`
`s.isupper()`	**Bool:** `True`, wenn alle Buchstaben in `s` großgeschrieben sind; sonst `False`
`sub in s`	**Bool:** `True`, wenn der Teilstring `sub` in `s` enthalten ist; sonst `False`. Groß- und Kleinschreibung wird unterschieden.
`len(s)`	**Integer:** Anzahl der Zeichen, die in `s` enthalten sind.
`s.count(sub)`	**Integer:** Anzahl, wie oft der Teilstring `sub` in `s` enthalten ist
`s.startswith(sub)`	**Bool:** `True`, wenn der Teilstring `sub` am Anfang von `s` steht; sonst `False`
`s.endswith(sub)`	**Bool:** `True`, wenn der Teilstring `sub` am Ende von `s` steht; sonst `False`
`s.find(sub)`	**Integer:** Position des ersten Vorkommens von `sub` in `s`; falls `sub` nicht enthalten ist, wird -1 zurückgegeben
`s.strip([chars])`	**String:** Kopie von `s` ohne alle Vorkommen der angegebenen `chars` an Beginn und Ende des Strings. Wird `chars` nicht angegeben, werden alle führenden und abschließenden Whitespaces von `s` entfernt.
`s.replace(old, new)`	**String:** Kopie von `s`, in der alle Vorkommen des Teilstrings `old` durch den String `new` ersetzt wurden.

Aufgabe (optional)

1. Einige der Funktionen in der Tabelle haben eine Variante mit einem r davor, z.B. `rsplit()` als Pendant zu `split()`. Prüfen Sie, für welche Funktionen das gilt. Können Sie experimentell herausfinden, was die Bedeutung dieser r-Varianten ist? Tipp: Die Funktion `strip()` hat auch eine l-Variante, `lstrip()`.

In [ ]:

```
# Ihre Tests
```

Besonderheiten bei Strings

Da wir Anführungszeichen `""` zur Markierung von Strings verwenden, können wir innerhalb von Strings zunächst keine Anführungszeichen schreiben. Es gibt einige Möglichkeiten, damit umzugehen.

Erstens erfüllen in Python die einfachen Anführungszeichen `''` die gleiche Funktion wie die doppelten Anführungszeichen `""`. Ein String wird immer mit der Art von Anführungszeichen beendet, mit der er begonnen wurde. Das bedeutet, dass die jeweils andere Form von Anführungszeichen im String wie ein normales Zeichen behandelt wird.

In [ ]:

```
print('"The time has come," the Walrus said, "to talk of many things"')
```

Zweitens ist es in Python möglich, mehrzeilige Strings zu erzeugen, indem wir an Anfang und Ende der Zeichenkette nicht nur ein Anführungszeichen setzen, sondern drei. Einzelne Anführungszeichen innerhalb dieses Strings sind dann unproblematisch.

In [ ]:

```
total_langer_string = """
"The time has come," the Walrus said,
"to talk of many things"
"""
print(total_langer_string)
```

Drittens können Anführungszeichen durch einen Backslash (`Alt Gr + ß`) **escapet** werden. Damit erkennt der Interpreter, dass das Zeichen direkt nach dem Backslash nicht als Anführungszeichen im Sinne von Programmcode behandelt werden soll, sondern nur als ein Zeichen innerhalb der Zeichenkette.

In [ ]:

```
print("\"The time has come,\" the Walrus said, \"to talk of many things\"")
```

Der Backslash hat noch mehr Funktionen. Wir können ihn verwenden, um innerhalb von Strings Zeilenumbrüche (`'\n'`) und Tabs (`'\t'`) zu markieren:

In [ ]:

```
print("If this be error and upon me proved,\nI never writ, nor no man ever loved.")
```

In [ ]:

```
print("ottos mops \t kotzt")
```

Das Zeichen `'\n'` entspricht übrigens auch solchen Zeilenumbrüchen, die wir ganz normal getippt haben. Der folgende String enthält vier Zeilen (beachten Sie das dreifache Anführungszeichen an Start und Ende des Strings). Um ihn an jedem Zeilenumbruch zu zerteilen, rufen wir `split()` mit dem Separator `"\n"` auf:

In [ ]:

```
print("""Am Grunde der Moldau wandern die Steine
Es liegen drei Kaiser begraben in Prag.
Das Große bleibt groß nicht und klein nicht das Kleine.
Die Nacht hat zwölf Stunden, dann kommt schon der Tag.
""".split("\n"))
```

Weil der Backslash diese Sonderfunktion erfüllt, muss er übrigens auch selbst escapet werden, wenn man ihn als bloßes Zeichen in einem String verwenden möchte. Wir schreiben dazu `\\` .

In [ ]:

```
print("D:\\Studium\\01-WiSe2019\\Python")
```

+ und +=

Manchmal wollen wir eine Variable am Anfang eines Programms erzeugen und sie dann nach und nach verändern. Dafür können wir beispielsweise schreiben:

```
anzahl_elemente = 0
for zahl in range(10):
    anzahl_elemente = anzahl_elemente + 1
```

Für diese Art von Anweisung, bei der die gleiche Variable einmal links und einmal rechts vom Zuweisungsoperator (`=`) steht, gibt es auch eine Kurzschreibweise:

```
anzahl_elemente = 0
for zahl in range(10):
    anzahl_elemente += 1
```

Beide Anweisungen machen faktisch das gleiche. Sie können sich entscheiden, welche Schreibweise Ihnen besser gefällt. Auch die Operatoren `-=` , `*=` und `/=` stehen zur Verfügung.

Achtung! Die Reihenfolge der Zeichen spielt eine Rolle! Wenn Sie `=+` schreiben statt `+=` , wird Ihr Programm sich nicht so verhalten wie vorgesehen. Können Sie sich vorstellen, warum?

Variablen überschreiben

Der Inhalt einer Variable kann sich im Laufe eines Programms verändern. In solchen Fällen brauchen wir nicht für jeden Wert einen neuen Variablennamen. Tatsächlich kann eine Variable auch in einer einzigen Zeile verändert werden, wenn wir folgendes schreiben:

In [ ]:

```
mein_string = "Nennt mich Ismael."
print(mein_string)

print("------")

mein_string = mein_string + " Als ich vor einigen Jahren – wie lange es genau her ist,
 tut wenig zur Sache – so gut wie nichts in der Tasche hatte und von einem weiteren Auf
enthalt auf dem Lande nichts mehr wissen wollte, kam ich auf den Gedanken, ein wenig zu
r See zu fahren, um die Welt des Meeres kennenzulernen."
print(mein_string)
```

Zusammenfassung

Sie haben heute gelernt,

- In VSCode mehrere Cursors auf einmal zu setzen
- Mit `for` Befehle in Schleifen zu verpacken, die pro Element einer Sequenz einmal ausgeführt werden
- Mit `if`, `elif`, `else` Befehle nur unter bestimmten Bedingungen auszuführen
- Verschiedene Operationen auf Strings anzuwenden
- Variablenwerte während der Laufzeit eines Programms zu überschreiben

Und morgen...

In der Übung am 16.10.2019 werden Sie üben, mit `if`-Blöcken und `for`-Schleifen die Abläufe in Ihrem Programm zu steuern. Sie werden sich außerdem intensiv mit Stringmethoden beschäftigen.