Inhalt

Rückblick

Python Programme sind Folgen von Ausdrücken

- Variablenzuweisung, Funktionsaufrufe
- Funktions- oder Klassendefinition
- Kontrollstrukturen (**`if`**, **`for`**,...)

Beispiel:

```
a = 42
b = 21
c = a + b
d = someFunc(c)     # Funktionsaufruf
xs = [0, 1, 2, 3]   # eine Liste
```

In unserem Kurs verwenden wir Jupyter Notebooks, die aus Code Zellen bestehen, in die man kleine Python-Programme schreiben und ausführen kann.

Python ist eine dynamisch typisierte Sprache, d.h.

- Variablen haben einen Typ
- der Typ einer Variablen wird von Python zur Laufzeit ermittelt
- Typ wird *nicht* explizit vom Benutzer festgelegt

Beispiel:

```
a = 42          # Variable a ist ein  int
b = 42.0        # b ist vom Typ  float
c = a * b       # Welchen Typ hat c?
xs = [a, b]     # xs ist vom Type  list
```

Die Funktion `type(x)` gibt den Typ von `x` zurück.

Probieren Sie es im Notebook aus!

Bisher haben wir uns im Wesentlichen auf die Datentypen **`int`** und **`float`** beschränkt - also ganze Zahlen und Fließkommazahlen. Das reicht natürlich nicht ganz aus. In Python gibt es eine Vielzahl weiterer Datentypen:

Numerische Werte	**`int`**, **`float`**, **`complex`**
Text	**`str`**
Sequenzen	**`list`**, **`tuple`**, **`range`**
Maps	**`dict`**
Mengen	**`set`**, **`frozenset`**
Bool'sche Werte	**`bool`**
Binäre Daten	**`bytes`**, **`bytearray`**, **`memoryview`**

Abbildung: Datentypen in Python (Auszug)

Python bietet Operatoren für Grundrechenarten:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division (Gleitkomma)
%	Modulo Operator (Rest)
//	Division (ganzzahlig)
**	Potenzieren

Zeichenketten

Natürlich wollen wir mit Python nicht nur Zahlen, sondern auch Text verarbeiten. Ein Stück Text ist in Python nichts anderes als eine Aneinanderreihung von Buchstaben bzw. Zeichen. Um Texte zu speichern, enthält Python den Datentyp **`str`** für Zeichenketten:

```
meinText = "Hallo!"
```

Der Code definiert eine Variable `meinText`, die die Zeichenkette bestehend aus den Zeichen `'H'`, `'a'`, `'l'`, `'l'`, `'o'` und `'!'` enthält.
Zeichenketten in Python beginnen und enden mit Anführungszeichen - entweder einfache oder doppelte.

Zugriff über den Index

Wir können über eckige Klammern jetzt auf die einzelnen Zeichen von unserem Text in der Variable `meinText` zugreifen, z.B.:

```
ersterBuchstabe = meinText[0]
```

Mit der **`print`** Funktion können wir uns die Variablen ausgeben lassen:

```
print(ersterBuchstabe)
```

Probieren Sie es im Notebook aus!

Es ist wichtig zu beachten, dass eine Zeichenkette etwas anderes darstellt, als z.B. eine Zahl. Im folgenden Beispiel wird eine Variable **a** mit der Zeichenkette **'4'** und **'2'** definiert. Die Variable **b** enthält die Zahl 42.

```
a = "42"
b = 42

vergleich = (a == b)
```

Was ergibt der obige Vergleich, d.h. welchen Wert enthält die Variable **vergleich**?

Probieren Sie es im Notebook aus!

Natürlich können wir nicht nur auf einzelne Buchstaben, sondern auch auf ganze Teile unserer **str** Variablen zugreifen. Dazu bieten Variablen vom Typ **str** die Möglichkeit auf *Bereiche* zuzugreifen, z.B.:

```
name = "Hello, world!"
mitte = name[1:4]
```

In diesem Falle greifen wir auf die Zeichen von Position 1 (einschließlich) bis Position 4 (ausgeschlossn) zu. Im obigen Beispiel erhalten wir also den Teil-String **ell** in der Variablen **mitte**.
Betrachten wir die folgende Variable:

```
name = "Hochschule Bochum"
stadt = ??
```

Mit welchen Indizes müssen Sie in diesem Beispiel auf die Variable **name** zugreifen, um in **stadt** nur noch das Wort **Bochum** zu erhalten?

Probieren Sie es im Notebook aus!

Weitere Funktionen mit `str`

Der Datentyp **`str`** stellt eine Menge nützlicher Funktionen bereit. So können wir z.B. alle Buchstaben in Groß- oder Kleinbuchstaben verwandeln:

```
name = "alice"
gross = name.upper()          # gross ist dann 'ALICE'
klein = name.lower()          # klein ist dann 'alice'
cap = name.capitalize()       # cap ist dann: "Alice"
name.islower()                # ergibt:  True
neu = name.replace('ice', 'ex')  # neu ist nun "alex"
```

Sie können diese Funktionen natürlich sehr leicht ausprobieren, indem Sie sich die neu definierten Variablen mit der **`print`** Funktion ausgeben lassen:

Probieren Sie es im Notebook aus!

Der Datentyp `list`

Bisher haben wir in Variablen immer nur einzelne Werte gespeichert. Als Beispiel haben wir die Variable `zahl`, die nur den Wert 42 enthält:

```
zahl = 42
```

Mit dem Datentyp **`list`** können wir nun mehrere Werte in einer einzelnen Variablen speichern. **`list`** ist also eine Liste von mehreren Werten und läßt sich in Python einfach mit eckigen Klammern definieren:

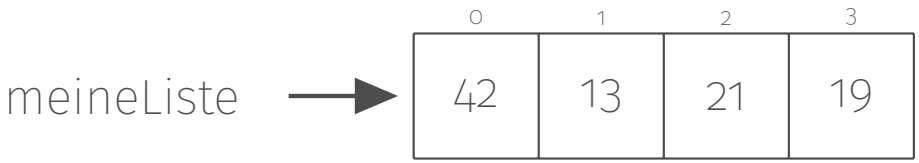
```
meineListe = [ 42, 13, 21, 19 ]
```

Dieser Code definiert eine Liste mit den Zahlen 42, 13, 21 und 19.

Betrachen wir den folgenden Code etwas genauer – was passiert dabei?

```
meineListe = [ 42, 13, 21, 19 ]
```

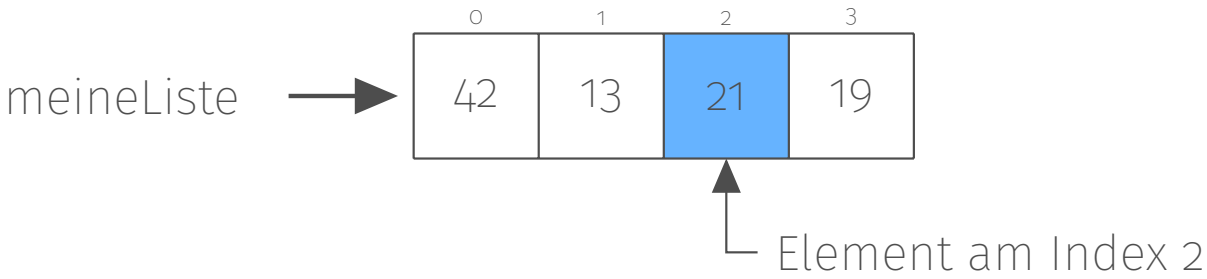
Python reserviert in ihrem Computer vier Speicherplätze und schreibt dort hintereinander die Werte 42, 13, 21 und 19:



Betrachen wir den folgenden Code etwas genauer – was passiert dabei?

```
meineListe = [ 42, 13, 21, 19 ]
```

Python reserviert in ihrem Computer vier Speicherplätze und schreibt dort hintereinander die Werte 42, 13, 21 und 19:




```
zahl = meineListe[2]   # zahl enthaelt den Wert 21
```

Wie die anderen Datentypen (z.B. **`int`**) unterstützen auch Listen ein paar einfache Operationen, z.B. `+` oder `*`:

```
xs = [1,2,3,4]      # Definition einer Liste
ys = [4,5,6,7]      # noch eine Liste

# Einige Operatoren funktionieren auch fuer Listen:
zs = xs + ys
ws = xs * 2
```

Welche Werte haben jetzt die Variablen `zs` und `ws`?

Probieren Sie es im Notebook aus!

Zugriff auf Teile einer Liste

Listen sind in ihren Eigenschaften ähnlich zu den Zeichenketten, die wir zuvor schon kennengelernt haben. Und wie beim Datentyp **`str`** können wir auch bei Listen auf einfache Weise auf Teil-Listen zugreifen:

Zugriff auf einzelne Elemente:

```
xs = [1, 2, 3, 4, 5, 6]
a = xs[3]  # Welchen Wert hat a ?
```

Zugriff auf Teil-Listen (*slicing*):

```
xs = [1, 2, 3, 4, 5, 6]
xxs = xs[0:4]              # Ein Teilbereich von xs
```

Probieren Sie es im Notebook aus!

Wieviele Elemente sind nun in einer Liste?

Mit der Funktion **len** kann man die Länge einer Liste – also die Anzahl der enthaltenen Elemente - berechnen:

```
xs = ['a', 'b', 'c', 'd']
laenge = len(xs)
```

Ist der Wert X in der Liste enthalten?

Mit dem Operator **in** kann gestestet werden, ob eine Liste ein bestimmtes Element enthält:

```
xs = [ 1, 2, 3, 4, 5 ]
y = 3
enthalten = y in xs

# enthalten ist nun   'True'
```

Viele weitere Möglichkeiten...

Natürlich existieren für Listen noch eine große Zahl weitere Funktionen, wie z.B.:

```
xs.append('e')     # Anhaengen eines weiteren Elements
xs.index('c')      # Berechnet den Index von 'c'
xs.remove('b')     # Entfernt ein Element
xs.sort()          # Sortiert die Liste
xs.reverse()       # Dreht die Reihenfolge um
```

Damit lassen sich Werte an eine Liste anhängen (*append*) oder aus einer Liste entfernen (*remove*) usw.; die Liste kann sortiert werden (*sort*) oder einfach umgedreht werden (*reverse*).
Einige von diesen Funktionen verändern die Liste selbst, andere erzeugen eine neue Liste mit den alten Werten.

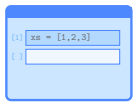
Spaß mit Listen:

Betrachen wir das folgende kleine Programm:

```
ys = [12, 37, 20, 38]
ys.append(24)
ys = ys * 2

if ys[2] % 20 == 0:
    ys.reverse()
```

Wie sieht `ys` nach Ablauf dieses Programms aus?



Probieren Sie es im Notebook aus!

Der Datentyp `tuple`

Tupel sind sehr ähnlich zu Listen, haben aber eine feste Länge. Damit lassen sich z.B. Paare oder Tripel definieren:

- Tupel fassen mehrere Elemente zusammen
- Tupel sind immutable - können nicht verändert werden
- Tupel haben daher feste Länge
- Zugriff auf Elemente wie bei einer Liste

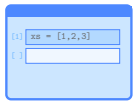
Tupel werden mit runden Klammern definiert und können (wie Listen) beliebige Elemente enthalten:

```
paar = (1,'A')              # ein 2er Tupel
dreier = (2, 'B', [1,2])    # 3er Tupel
```

Der Datentyp `tuple`

Tupel verhalten sich im Prinzip wie *nur lesbare* Listen. Nur lesbar bedeutet hier, dass man ein Tupel *nicht um weitere Elemente erweitern* kann. Bei Listen konnte man ja z.B. mit **append** weitere Elemente anhängen.

```
paar = (1, 'A')    # Definition des Tupels
x = paar[0]        # x = 1
y = paar[1]        # y = 'A'
```



Probieren Sie es im Notebook aus!

Zugriff auf Tupel-Werte

Manchmal ist klar, dass ein Tupel z.B. aus genau drei Werten besteht. In diesem Fall kann man die Werte auch direkt in Variablen auspacken, so dass man leicht darauf zugreifen kann:

```
mein_tupel = ('Vorlesung', '10:00', 'Mittwochs')

(name, uhrzeit, tag) = mein_tupel

# ist das Gleiche wie:
name = mein_tupel[0]
uhrzeit = mein_tupel[1]
tag = mein_tupel[2]
```

Funktionen mit mehreren Werten

Tupel eignen sich gut, um z.B. Funktionen zu schreiben, die mehr als nur ein Ergebnis haben. Zum Beispiel liefert die folgende Funktion zwei Ergebnisse zurück:

```
def meineFunktion(zahl):
    wert1 = zahl - 1
    wert2 = zahl + 1
    return (wert1, wert2)
```

Wenn man die Funktion jetzt z.B. mit dem Parameter 42 aufruft, ist das Ergebnis ein Tupel mit zwei Teilen:

```
erg = meineFunktion(42)
```

Was steht nun in der Variablen `erg`?

Probieren Sie es im Notebook aus!

Schleifen

Sie haben in den vorangegangenen Blöcken schon eine Reihe von Kontrollstrukturen kennengelernt, aus denen wir Python-Programme für bestimmte Abläufe erstellen können, z.B.

- bedingte Anweisungen mit **if** und ggf. **else**
- eigene Funktionen mit **def**

Beispiel:

```
x = 30
if x > 10:
  print("x ist mehr als 10!")
else:
  print("x ist weniger als 10!")
print(x)
```

Ein besonderes Merkmal von Python ist, dass hier Einrückungen benutzt werden, um zu definieren, welche Anweisungen in einer Bedingung ausgeführt werden und welche z.B. im **else** Fall:

```
if x > 4:
    print("Mehr als 4")
    x = x + 6
else:
    print("Weniger als 5")
print("Ende")
```

Zusammenhängender Block

ausgegeben - die Ausgabe `Mehr als 4` und die Erhöhung um 6 nur, wenn entweder x einen Wert enthält, der größer ist als 4. Andernfalls wird nur der Text `Weniger als 5` ausgegeben.. Im Falle dieses Beispiels wird z.B. als letzte Anweisung immer **Ende**, weil Sie nicht mehr Teil der beiden If/else Blöcke ist.

Wiederholungen mit Schleifen

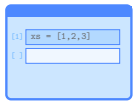
Ein zentrales Element der meisten Programmiersprachen ist die Möglichkeit, einen Block von Anweisungen mehrfach auszuführen. Es gibt in Python zwei wesentliche Arten von Schleifen: **`while`** und **`for`**.

Beispiel:

```
i = 0
xs = []
while i < 10:
  xs.append(i)
  i = i + 1
```

In diesem Beispiel wird der Block innerhalb der **`while`**-Schleife so lange ausgeführt, solange `i` einen Wert kleiner 10 hat.

- Welchen Wert hat `i`, wenn das Programm zu Ende ausgeführt wurde?
- Wie sieht die Liste in der Variablen `xs` nach Ende des Programms aus?



Probieren Sie es im Notebook aus!

Struktur der `while` Schleife

Eine while Schleife besteht aus dem Wort **`while`**, gefolgt von einer Bedingung und dann einem Block mit einer oder mehreren Anweisungen:

```
while BEDINGUNG:
    ANWEISUNG1
    ANWEISUNG2
    ...
```

Vor jedem Schleifendurchlauf wird überprüft, ob die Bedingung wahr ist, dann wird der Block der Schleife ausgeführt. Andernfalls ist die Schleife beendet und das Programm wird mit dem Code nach der Schleife weitergeführt.

Eine while Schleife wiederholt also so lange eine oder mehrere Anweisungen, solange ihre Bedingung wahr ist (**`True`**).

Schleifen mit `for`

Häufig möchte man Anweisungen für eine Reihe von Elementen immer wiederholen. Zum Beispiel um Elemente einer Liste zu zählen. Dazu bietet Python die **`for`** Schleifen an:

Beispiel:

```
xs = [1, 2, 3, 4]
for x in xs:
  print(x)
```

In diesem Beispiel wird eine Liste `xs` definiert und dann mit einer **`for`** Schleife die Anweisung `print(x)` für jedes Element der Liste ausgeführt.

Struktur der for Schleife

Eine **for** Schleife besteht also aus dem Wort `for`, gefolgt von einer Variablen, dem Wort `in` und einer Sequenz/Folge von Werten (z.B. einer Liste). Danach kommt ein Block mit einer oder mehreren Anweisungen:

```
for VARIABLE in SEQUENZ:
  ANWEISUNG1
  ANWEISUNG2
```

Was tut also das folgende kleine Programm?

```
xs = [1, 2, 3, 4, 5]
s = 0
for x in xs:
  s = s + x
print(s)
```

Probieren Sie es im Notebook aus!

for Schleife mit Strings

Die **for** Schleifen in Python sind für Listen und andere Folgen sehr nützlich. Das funktioniert z.B. auch mit Zeichenketten, die ja auch Folgen von Buchstaben sind.

Was macht z.B. das folgende Programm?

```
anzahl = 0
text = "Python ist eine recht einfache Sprache"

for b in text:
  if b == 'e':
    anzahl = anzahl + 1
print(anzahl)
```

Probieren Sie es im Notebook aus!

Sequenzen mit range

Manchmal ist es sehr mühsam, lange Folgen z.B. als Liste zu definieren. Stellen Sie sich vor, Sie müssten eine Folge der Zahlen von 1 bis 100 definieren. Python stellt für Zahlenfolgen Typ range:

```
seq1 = range(4)      # erzeugt die Folge 0, 1, 2, 3

seq2 = range(1,4)    # erzeugt die Folge 1, 2, 3

seq3 = range(1,9,3) # erzeugt die Folge 1, 4, 7
```

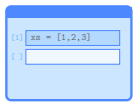
Mit **range** lassen sich nun Folgen für eine **for** Schleife leicht angeben. Zum Beispiel die Summe der Zahlen von 1 bis 99:

```
summe = 0
for zahl in range(100):
  summe = summe + zahl
```

Sequenzen mit range

range kann sehr nützlich sein. Überlegen Sie sich z.B. wie Sie mit **range** und einer **for** Schleife die

1. Summe der Quadratzahlen von 1 bis 10 berechnen
2. die Summe der geraden Zahlen zwischen 1 und 1000 ausrechnen.



Probieren Sie es im Notebook aus!