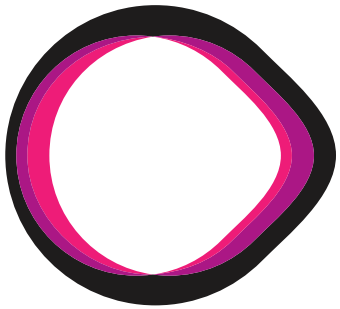
Python – Lektion 2
Verzweigungen, Schleifen und Funktionen



OST
Ostschweizer
Fachhochschule

Rückblick - Variablen und Datentypen

Datentyp	Beschreibung	False-Wert
`NoneType`	Indikator für nichts, keinen Wert.	`None`
Numerische Datentypen (unveränderlich)		
`int`	Ganze Zahlen	`0`
`float`	Gleitkommazahlen	`0.0`
`bool`	Boolesche Werte	`False`
`complex`	Komplexe Zahlen	`0 + 0j`
Sequenzielle Datentypen		
`str`	Zeichenketten oder Strings (unveränderlich)	`''`
`list`	Listen (veränderlich)	`[]`
`tuple`	Tupel (unveränderlich)	`()`
`bytes`	Sequenz von Bytes (unveränderlich)	`b''`
`bytearray`	Sequenz von Bytes (veränderlich)	`bytearray(b'')`
Mengen		
`set`	Menge mit einmalig vorkommenden Objekten	`set()`
`frozenset`	Wie `set` jedoch unveränderlich	`frozenset()`
Assoziative Datentypen		
`dict`	Dictionary (Schlüssel-Wert-Paare, veränderlich)	`{}`

Sequentielle Datentypen - Indizierung

Auf ein Element zugreifen:

```
x =   "   H   A   L   L   O   "
          0   1   2   3   4
         -5  -4  -3  -2  -1
```

`http://localhost:8888/notebooks/indizierung.ipynb`

- Index startet bei Null
- Indizierung über eckige Klammern
- Rückwärts über negative Indizes

Sequentielle Datentypen - Slicing

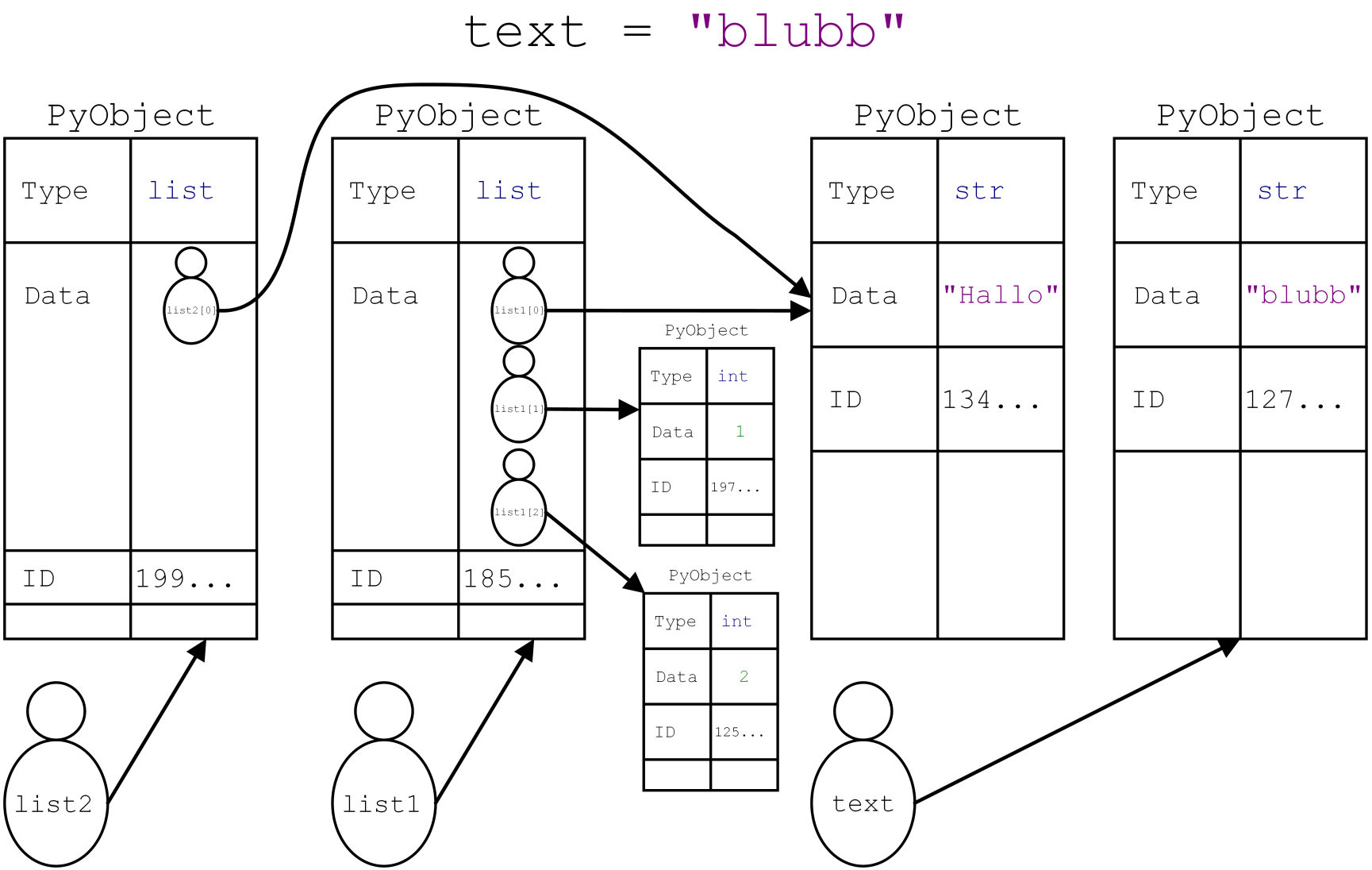
Einen Teil ausschneiden:

```
slice =  x[start:  end:  step]
```

`http://localhost:8888/notebooks/slicing.ipynb`

- `x[:end:step]`
- `x[start::step]`
- `x[start:stop]`
- Nicht vorhandener Bereich $\rightarrow$ Leerer String, Liste, Tuple etc

Variablen sind Referenzen auf Objekte



Heutige Themen

- Verzweigungen (`if`, `else`, `elif`)
- Schleifen (`for`, `while`)
- Funktionen definieren und benutzen

Verzweigungen

Die `if`-Anweisung:

```
if␣Bedingung:
␣␣␣␣Anweisung1
␣␣␣␣Anweisung2
```

- Anweisungen 1 & 2 nur ausführen, wenn die Bedingung **wahr** ist.

Alle Anweisungen im gleichen Codeblock müssen gleich eingerückt sein, z.B. mit vier Leerzeichen[a], sonst wird ein Fehler ausgegeben.

[a] `https://www.python.org/dev/peps/pep-0008/#indentation` Tabulatoren sind möglich, werden aber nicht empfohlen.

Verzweigungen

`if`-Anweisung mit `else`-Zweig:

```
if Bedingung:
    Anweisung1
    Anweisung2
else:
    Anweisung3
    Anweisung4
```

- Anweisungen 1 und 2, falls Bedingung **wahr**
- Anweisungen 3 und 4, falls Bedingung **unwahr**

Verzweigungen

`if`-Anweisung mit `elif`-Zweigen und `else`-Zweig:

```
if Bedingung1:
    Anweisung1
elif Bedingung2:
    Anweisung2
elif Bedingung3:
    Anweisung3
else:
    Anweisung4
```

- `elif` = `else if`

Python kennt keine switch-case-Anweisung.

- Bedingungen werden als boolesche Ausdrücke evaluiert

Boolesche Ausdrücke

- Für jeden Datentyp gibt es einen Wert, der als **unwahr** gilt:

Datentyp	**False-Wert**
`NoneType`	`None`
`int`	`0`
`float`	`0.0`
`bool`	`False`
`complex`	`0 + 0j`
`str`	`''`
`list`	`[]`
`tuple`	`()`
`bytes`	`b''`
`bytearray`	`bytearray(b'')`
`dict`	`{}`
`set`	`set()`
`frozenset`	`frozenset()`

- Jeder andere Wert gilt als **wahr**!

Bedingung mit vergleichenden Operatoren

Operator	Beschreibung
`x is y`	**wahr**, wenn `x` und `y` gleich sind
`x == y`	**wahr**, wenn `x` und `y` gleich sind
`x != y`	**wahr**, wenn `x` und `y` verschieden sind
`x < y`	**wahr**, wenn `x` kleiner als `y` ist[1]
`x <= y`	**wahr**, wenn `x` kleiner oder gleich `y` ist[1]
`x > y`	**wahr**, wenn `x` grösser `y` ist[1]
`x >= y`	**wahr**, wenn `x` grösser oder gleich `y` ist[1]

`http://localhost:8888/notebooks/is_is.ipynb`

- Unterschied zwischen == und `is` Operator:
  - == prüft Wertgleichheit
  - `is` prüft Objektgleichheit
- `is` Operator sollte nur bei Prüfung auf `None` verwendet werden!

[1] Nicht definiert für den Datentyp `complex`

Bedingung mit logischen Operatoren

Operator	Beschreibung
`not x`	**wahr**, wenn `x` **unwahr** und vice versa (Invertiert die Logik von `x`)
`x or y`	**wahr**, wenn `x` oder `y` **wahr** ist
`x and y`	**wahr**, wenn `x` und `y` **wahr** ist

```
x or y or z or ...
x and y and z and ...
```

Bedingung mit vergleichenden und logischen Operatoren

```
x < y and y >= z
```

```
x < y >= z
```

```
x < y() and y() >= z
```

```
x < y() >= z
```

- `y` bzw. `y()` wird mit `and` zweimal evaluiert.

Bedingung mit Keyword in

- Prüfen ob sich ein Wert in einem Objekt befindet:

`http://localhost:8888/notebooks/iter.ipynb`

- Der in-Operator kann auf jedes Objekt angewendet werden, der die Methode `__contains__()` implementiert, z.B. `str`, `list`, `tuple`, `dict`, `set`, ..., oder eigene Klassen.

Schleifen

Die `for`-Schleife:

```
for Wert in Sequenz:
    Anweisung1
```

- dient zur Iteration einer Sequenz
- Sequenz muss ein iterierbares Objekt sein
- Ein iterierbares Objekt ist ein Objekt, welches die Methode `__iter__()` besitzt, mit welcher auf ein Element nach dem anderen zugegriffen werden kann. Dazu gehören:
  `list`, `tuple`, `str`, `bytes`, `bytearray`, `set`, `frozenset`, `dict`

Zähl-Schleife mit `range(start, stop, step)` (exklusive `stop`):

```
for n in range(5):
    print(n)
```

Schleifen

Die `while`-Schleife:

```
while Bedingung:
    Anweisung1
```

- Anweisung 1 wird wiederholt, solange die Bedingung **wahr** ist.

Python kennt keine do-while-Schleife.

```
while Bedingung:
    Anweisung1
else:
    Anweisung2
```

- `else`-Teil wird ausgeführt, wenn die while-Condition `False` wird und die Schleife damit 'normal' verlassen wird

Schleifen

Die `while`-Schleife:

```
while Bedingung:
    Anweisung1
```

- Anweisung 1 wird wiederholt, solange die Bedingung **wahr** ist.

Python kennt keine do-while-Schleife.

```
for Wert in Sequenz:
    Anweisung1
else:
    Anweisung2
```

- Gibt es auch bei der `for`-schleife

Schleifen

`while`-Schleife abbrechen:

```
while Bedingung:
    Anweisung1
    if Fehler:
        break
    Anweisung2
else:
    Anweisung3
```

- `break` bricht die `while`-Schleife vorzeitig ab

Schleifen

Durchlauf beenden und zurück nach oben:

```
while Bedingung:
    Anweisung1
    if Ausnahme:
        continue
    Anweisung2
else:
    Anweisung3
```

- `continue` beendet den aktuellen Durchlauf und springt nach oben

`http://localhost:8888/notebooks/while.ipynb`

Funktionen

- Python besitzt eingebaute Funktionen:
  `https://docs.python.org/3/library/functions.html`

- und eine grosse Standard-Bibliothek, z.B.:

```
import time     # time.time(), time.sleep()
import math     # math.pi, math.cos()
import zipfile # ZIP-Dateien manipulieren
```

  `https://docs.python.org/3/library/`

Funktionen - Definition

- Einfache Funktionsdefinition:

```
def Funktionsname(Parameterliste):
    Anweisungen
```

- Beispiel:

```
def begruessung(vorname, nachname):
    print('Hallo', vorname, nachname)
```

- Der Funktionsname kann frei[2] gewählt werden.
- Parameternamen durch Kommas trennen
- Codeblock gleichmässig einrücken

[2] https://www.python.org/dev/peps/pep-0008/#function-and-variable-names

Funktionen - Rückgabewert

- Der Rückgabewert der Funktion ist `None`, falls nichts angegeben wird.

```
def begruessung(vorname, nachname):
    print('Hallo', vorname, nachname)
```

- `return`-Anweisung beendet den Funktionsaufruf mit Rückgabewert:

```
def division(a, b):
    return a/b
```

Funktionen - Rückgabewert

- Der Rückgabewert der Funktion ist `None`, falls nichts angegeben wird.

```
def begruessung(vorname, nachname):
    print('Hallo', vorname, nachname)
```

- `return`-Anweisung beendet den Funktionsaufruf mit Rückgabewert:

```
def division(a, b):
    if b != 0:
        return a/b
    else:
        return
```

- leere `return`-Anweisung liefert `None` zurück
- mehrere `return`-Anweisungen sind erlaubt, wie in C/C++

Funktionen - Docstrings

- Nach Funktionsdefinition sollte immer ein Docstring[3][4] eingefügt werden:

```
def summe(a, b):
    '''Gibt die Summe a + b zurueck.'''
    return a + b
```

http://localhost:8888/notebooks/funktionen.ipynb

[3] https://www.python.org/dev/peps/pep-0257/#one-line-docstrings
[4] https://www.python.org/dev/peps/pep-0257/#multi-line-docstrings

Lambda-Funktionen

```
lambda Parameterliste : Ausdruck
```

- Äquivalente ”normale” Funktion:

```
def anonyme_funktion(Parameterliste):
    return Ausdruck
```

- Anonyme Funktionen mit beliebiger Anzahl Parameter und einer Anweisung

`http://localhost:8888/notebooks/lambda.ipynb`

- Lambda-Funktionen werden häufig mit eingebauten Pythonfunktionen (`sort()`, `filter()`, `map()`) verwendet