Vertiefung der Python Konzepte

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Erweiterte Konzepte der Programmiersprache Python`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/09_PythonVertiefung.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Bernhard Jung

Fragen an die heutige Veranstaltung ...

- Welche Standarddatentypen exisiteren in Python über die Liste hinaus?
- In welchen Anwendungsfällen kommen diese zum Einsatz?
- Wie lassen sich Funktionen mit Python realsieren und welche Unterschiede existieren im Vergleich zu C++?

Weitere Datentypen

Sie haben bereits Listen (`list`), `range` Objekte und Text (`string`) als Datenstruktur kennengelernt - im Weiteren existieren daneben vier weitere Sequenzdatentypen: byte sequences (`bytes` objects), byte arrays (`bytearray` objects) und `tuples`. Dazu kommen `dictionaries` und `sets` als Containertypen.

Datentyp	Besonderheit	Syntax
`list`	*veränderbare* Sequenz von (beliebigen) Daten	`l = ["grün", 1, True]`
		`l[0] = 4` `l.append(1)`
`strings`	Darstellung von Zeichenketten, unveränderbar	`s = "Universität"`
`bytes`	Unveränderbare Folge von ASCII-Zeichen	`b = b"Universitaet"`
`bytearray`	Unveränderbare Folge von Bytes (Integer 0...255)	`b_array = bytearray([0, 1, 255])`
`tupel`	Unveränderbare Folge von Elementen	`t1 = (1, 2, 3)`
`range`	iterierbare, unveränderbare Folge von Elementen	`r = range(3, 20, 2)`

Tupel oder Liste?

- Listen sind veränderbar, d.h. Elemente können hinzugefügt, gelöscht oder modifiziert werden.
- Tupel sind dagegen unveränderbar.

Warum ein Tupel anstelle einer Liste verwenden?

- Die Programmausführung ist beim Iterieren eines Tupels etwas schneller als bei der entsprechenden Liste - Dies wird wahrscheinlich nicht auffallen, wenn die Liste oder das Tupel klein ist.
- Der Speicherbedarf eines Tupels fällt etwas in der Regel geringer aus, als bei einer Listendarstellung ein und des selben Inhaltes
  - weil die zusätzlichen Möglichkeiten zur Veränderung von Listen etwas zusätzlichen Speicherbedarf zur Verwealtung erfordern.
- Manchmal sollen Daten unveränderlich gehalten werden - Tupel schützen die Informationen vor versehentlichen Änderungen.
- Als Elemente von Sets und Schlüssel von Dictionaries (s.u.) sind nur unveränderliche Objekte erlaubt, also z.B. Tupel aber keine Listen.

size.py

```
1  import sys
2
3  a_list = []
4  a_tuple = ()
5  a_list = ["Hello", "TU", "Freiberg"]
6  a_tuple = ("Hello", "TU", "Freiberg")
7  print("List data size : " + str(sys.getsizeof(a_list)))
8  print("Tupel data size: " + str(sys.getsizeof(a_tuple)))
```

```
List data size : 88
Tupel data size: 64
List data size : 88
Tupel data size: 64
```

duration.py

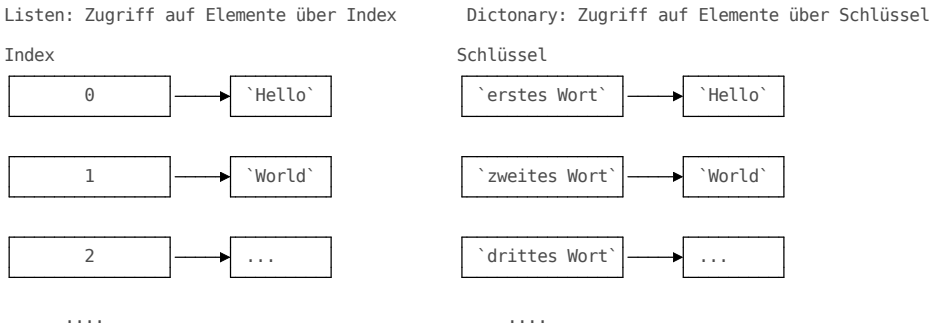
```python
import time

l=list(range(1_000_000))
t=tuple(range(1_000_000))

start = time.perf_counter()
for item in l:
    a = item
end = time.perf_counter()
print("List duration:  ", end - start, "s")

start = time.perf_counter()
for item in t:
    a = item
end = time.perf_counter()
print("Tuple duration: ", end - start, "s")
```

```
List duration:   0.04237871803343296 s
List duration:   0.03696513100294396 s
Tuple duration:  0.04299071303103119 s
Tuple duration:  0.05006967199733481 s
```

Dictionaries

Dictionaries werden zur Speicherung von Schlüssel Werte Paaren genutzt. Ein dictionary ist eine Sammlung von geordneten (entsprechend der Reihenfolge der "Einlagerung"), veränderlichen Einträgen, für die Schlüssel werden keine Duplikate zugelassen.

> Ein Telefonbuch ist das traditionelle Beispiel für eine Implementierung des Dictonaries. Anhand der Namen werden die Telefonnummern zugeordnet.



dictionary.py

```python
capitals = {"France": "Paris",
            "Italy": "Rome",
            "UK": "London"}

print(capitals)

print("UK" in capitals)

capitals["Germany"] = "Berlin"  # add a single value

more_capitals = {"Belgium": "Brussels",
                 "Poland": "Warsaw",}

capitals.update(more_capitals)  # add a single or multiple new entrie
                                # contained in a dictionary
print(capitals)

another_dict_with_capitals = {"France": "Paris",  # France is in this
  too!
                              "Spain": "Madrid",}
capitals = capitals | another_dict_with_capitals  # union of the two
print(capitals)
```

```
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London'}
True
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London', 'Germany':
'Berlin', 'Belgium': 'Brussels', 'Poland': 'Warsaw'}
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London', 'Germany':
'Berlin', 'Belgium': 'Brussels', 'Poland': 'Warsaw', 'Spain': 'Madrid'}
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London'}
True
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London', 'Germany':
'Berlin', 'Belgium': 'Brussels', 'Poland': 'Warsaw'}
{'France': 'Paris', 'Italy': 'Rome', 'UK': 'London', 'Germany':
'Berlin', 'Belgium': 'Brussels', 'Poland': 'Warsaw', 'Spain': 'Madrid'}
```

wrongdictonary.py

```
1  oldtimer =  {"brand": "VW",
2              "model": "Käfer",
3              "year": 1964,
4              "year": 2020
5  }
6  print(oldtimer)
```

```
{'brand': 'VW', 'model': 'Käfer', 'year': 2020}
{'brand': 'VW', 'model': 'Käfer', 'year': 2020}
```

Nehmen wir an, Sie entwerfen ein Verzeichnis der Studierenden aus Freiberg. Sie wollen die Paarung Studierendenname zu Matrikel als Dictonary umsetzen. Einer Ihrer Kommilitonen schlägt vor, dafür zwei Listen zu verwenden und die Verknüpfung über den Index zu realisieren. Was meinen Sie dazu?

goodSolution.py

```
1  students = {"von Cotta": 12,
2              "Humboldt": 17,
3              "Zeuner": 233}
4
5  student = "von Cotta"
6  print(f"Student  {student} ({students[student]})")
```

```
Student  von Cotta (12)
Student  von Cotta (12)
```

badSolution.py

```
1  names = ["von Cotta", "Humboldt", "Zeuner"]
2  matrikel = [12, 17, 233]
3
4  i = 1
5  print(f"Student  {names[i]} ({matrikel[i]})")
```

```
Student  Humboldt (17)
Student  Humboldt (17)
```

Sets

Ein Set ist eine Sammlung, die ungeordnet, unveränderlich (in Bezug auf exisiterende Einträge) und nicht indiziert ist. Eine Kernidee ist das Verbot von Duplikaten.

set.py

```
1  fruits = {"apple", "banana", "cherry", "lemon", "apple"}
2  print(fruits)
3
4  vegetables_list = ["carrot", "broccoli", "asparagus", "carrot"]
5  vegetables = set(vegetables_list)
6  print(vegetables)
```

```
{'banana', 'lemon', 'apple', 'cherry'}
{'asparagus', 'carrot', 'broccoli'}
{'apple', 'banana', 'lemon', 'cherry'}
{'carrot', 'broccoli', 'asparagus'}
```

Die Leistungsfähigkeit von Sets resultiert aus den zugehörigen Mengenoperationen.

set.py

```
1  a = {1,2,3,4,5,6}
2  b = {2,4,6,7,8,9}
3  even = {2,4,6,8,}
4
5  print(8 in a)      # ist 8 Element von a?
6  print(even < b)    # ist even eine Teilmenge von b?
7  print(a | b)       # Vereinigung von a und b
8  print(a & b)       # Schnittmenge von a und b
9  print(b - a)       # welche Einträge existieren in b die nicht in a prä
     sind
```

```
False
True
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{2, 4, 6}
{8, 9, 7}
False
True
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{2, 4, 6}
{8, 9, 7}
```

Über den Elementen eines Sets ist keine Reihenfolge festgelegt

- ... im Gegensatz zu Sequenz-Datentypen wie Listen, Zeichenketten oder Tupel, die eine Indizierung unterstützen, d.h. Zugriff auf Elemente mit ganzzahligem Index, z.B. `my_todo_list[3]`
- (nicht nur) bei `print`-Ausgaben ist es jedoch oft wünschenswert, die Elemente eines Sets in eine feste / vorhersagbare Reihenfolge zu bringen. Dies kann z.B. mit der builtin-Funktion `sorted()` erfolgen

```
1  fruits = {"banana", "cherry", "apple", "lemon", "apple"}
2  print("Set of fruits:", fruits)  # Reihenfolge der Elemente nicht defi
3  sorted_fruits = sorted(fruits)  # liefert sortierte Liste
4  print("List of fruits:", sorted_fruits)
5
6  vegetables = ["carrot", "broccoli", "asparagus", "broccoli", "carrot"]
7  unique_vegetables = set(vegetables)  # Duplikate werden entfernt
8  sorted_unique_vegetables = sorted(unique_vegetables)
9  print("\nVegetables (no duplicates, sorted):", sorted_unique_vegetable
```

```
Set of fruits: {'lemon', 'banana', 'cherry', 'apple'}
List of fruits: ['apple', 'banana', 'cherry', 'lemon']

Vegetables (no duplicates, sorted): ['asparagus', 'broccoli', 'carrot']
Set of fruits: {'banana', 'apple', 'cherry', 'lemon'}
List of fruits: ['apple', 'banana', 'cherry', 'lemon']

Vegetables (no duplicates, sorted): ['asparagus', 'broccoli', 'carrot']
```

Zusammenfassung

DataTypeExample.py

```python
for i in ['a','b','c']:      # Liste
  print(i, end=",")
print()

for i in "abc":              # String
  print(i, end=",")
print()

for i in ('a','b','c'):      # Tupel
  print(i, end=",")
print()

for i in {0:"a", 1:"b"}:     # Dictionary
  print(i, end=",")
print()

for i in {'a','b','c','c'}: # Set
  print(i, end=",")
```

```
a,b,c,
a,b,c,
a,b,c,
0,1,
c,b,a,a,b,c,
a,b,c,
a,b,c,
0,1,
b,a,c,
```

Gegeben sei eine Liste der Studiengangsbezeichnungen für die Studierenden dieser Vorlesung. Leiten Sie aus der Liste ab

1. wie viele Studierende eingeschrieben sind?
2. wie viele Studiengänge in der Veranstaltung präsent sind?
3. wie viele Studierende zu den Studiengängen gehören?

```python
# Angabe der Studiengänge der 2022 eingeschriebenen Teilnehmer in der
# Veranstaltung
study_programs = [
  "S-UWE", "S-WIW", "S-GÖ", "S-VT", "S-GÖ", "S-BAF", "S-VT",
  "S-WWT", "S-NT", "S-WIW", "S-ET", "S-WWT", "S-MB", "S-WIW",
  "S-FWK", "F1-INF", "S-WIW", "S-BWL", "S-WIW", "S-MAG",
  "F2-ANCH", "S-MAG", "S-WWT", "S-NT", "S-ACW", "S-GTB",
  "S-WIW", "F2-ANCH", "S-GTB", "S-GÖ", "S-GBG", "S-GM",
```

```
  "S-MAG", "S-GTB", "S-WIW", "S-WIW", "S-FWK", "S-WIW",
  "S-MAG", "S-GBG", "S-GÖ", "S-BAF", "S-BAF", "S-NT", "S-GÖ",
  "S-WWT", "S-GBG", "S-WWT", "S-GBG", "S-ERW", "S-WWT",
  "S-WIW", "S-NT", "S-WIW", "S-GÖ", "S-WIW", "S-GM",
  "S-GBG", "F1-INF", "S-WIW", "S-WWT", "S-ACW", "S-WIW",
  "S-WWT", "S-ACW", "S-INA", "S-FWK", "S-GTB", "S-WIW",
  "S-MORE", "S-WIW", "S-GÖ", "S-BWL", "S-CH", "S-WIW",
  "F2-ANCH", "S-WIW", "S-ACW", "S-ET", "S-ET", "S-GÖ",
  "S-GÖ"
]

# zu 1
print(len(study_programs))                    # Länge der Liste

# zu 2
print(len(set(study_programs)))               # Anzahl Studiengänge
                                              # als Größe des Sets

# zu 3
print({i:study_programs.count(i)              # (Studiengang:Häufigkeit
    for i in study_programs})                 #  als "Dictonary"
        Comprehension
```

```
82
22
{'S-UWE': 1, 'S-WIW': 18, 'S-GÖ': 9, 'S-VT': 2, 'S-BAF': 3, 'S-WWT': 8,
'S-NT': 4, 'S-ET': 3, 'S-MB': 1, 'S-FWK': 3, 'F1-INF': 2, 'S-BWL': 2,
'S-MAG': 4, 'F2-ANCH': 3, 'S-ACW': 4, 'S-GTB': 4, 'S-GBG': 5, 'S-GM':
2, 'S-ERW': 1, 'S-INA': 1, 'S-MORE': 1, 'S-CH': 1}
82
22
{'S-UWE': 1, 'S-WIW': 18, 'S-GÖ': 9, 'S-VT': 2, 'S-BAF': 3, 'S-WWT': 8,
'S-NT': 4, 'S-ET': 3, 'S-MB': 1, 'S-FWK': 3, 'F1-INF': 2, 'S-BWL': 2,
'S-MAG': 4, 'F2-ANCH': 3, 'S-ACW': 4, 'S-GTB': 4, 'S-GBG': 5, 'S-GM':
2, 'S-ERW': 1, 'S-INA': 1, 'S-MORE': 1, 'S-CH': 1}
```

Eigene Funktionen

> Das kennen wir schon ... aber noch mal zur Sicherheit

Funktionen sind Unterprogramme, die ein Ausgangsproblem in kleine, möglicherweise wiederverwendbare Codeelemente zerlegen.

Bessere Lesbarkeit

Der Quellcode eines Programms kann schnell mehrere tausend Zeilen umfassen. Beim Linux Kernel sind es sogar über 15 Millionen Zeilen und Windows 11, das ebenfalls zum Großteil in C geschrieben wurde, umfasst schätzungsweise über 60 Millionen Zeilen. Ein modernes Auto enthält sogar über 100 Millionen Zeilen Code. Um dennoch die Lesbarkeit des Programms zu gewährleisten, ist die Modularisierung unerlässlich.

Wiederverwendbarkeit

In fast jedem Programm tauchen die gleichen Problemstellungen mehrmals auf. Oft gilt dies auch für unterschiedliche Applikationen. Da nur Parameter und Rückgabetyp für die Benutzung einer Funktion bekannt sein müssen, erleichtert dies die Wiederverwendbarkeit. Um die Implementierungsdetails muss sich der Entwickler dann nicht mehr kümmern.

Wartbarkeit

Fehler lassen sich durch die Modularisierung leichter finden und beheben. Darüber hinaus ist es leichter, weitere Funktionalitäten hinzuzufügen oder zu ändern.

In allen 3 Aspekten ist der Vorteil in der Kapselung der Funktionalität zu suchen.

Syntax

> Funktionsdefinition starten immer mit dem Schlüsselwort **def**. Typen für Parameter oder Rückgabewerte müssen nicht notwendigerweise angegeben werden! Es gelten die üblichen Einrückungsregeln.

```
def funktionsname(arg1, arg2, ...):
  <anweisungen>
```

FunctionExample.py

```
1  import math
2
3  def print_pi():
4    print(math.pi)
```

Parameterübergabe

```python
def halbiere(zahl):
  zahl = zahl / 2
  print(zahl)

zahl = 5
print( halbiere(zahl) )
```

```
2.5
None
2.5
None
```

Python erlaubt analog zu C++ die Vergabe von Standardparametern beim Funktionsaufruf.

Returnwerte

Mit **return** kann **ein** Rückgabewert festgelegt und die Funktion beendet werden. Der Rückgabewert kann auch ein Tupel, eine Liste oder ein beliebiges anderes Objekt sein. Tupel können später wieder in einzelne Variablen augetrennt werden.

```python
def halbiere(zahl):
  zahl = zahl / 2
  return zahl

zahl = 5
print(halbiere(zahl))
print(zahl)
```

```
2.5
5
2.5
5
```

Der Parameter `zahl` wird als *Call-by-Assignment* übergeben. Die Zuweisung eines neuen Wertes ändert nicht die gleichnamige Variable außerhalb der Funktion!

`return` erlaubt formal lediglich einen Rückgabewert. Wie handhaben wir dann die Situation, wenn es mehrere sind?

- Python erlaubt nach einem `return` mehrere Rückgabewerte. Die Rückgabe erfolgt formal als *ein* Tupel

```python
def sumAndMultiplyTo(n):
  sum = 0
  prod = 1
  for i in range(1,n+1):
    sum = sum + i
    prod = prod * i
  return sum,prod

result = sumAndMultiplyTo(10)
print( type(result) )
print(result)

x,y = sumAndMultiplyTo(10)
print(x)
print(y)
```

```
<class 'tuple'>
(55, 3628800)
55
3628800
<class 'tuple'>
(55, 3628800)
55
3628800
```

Typ-Hinweise für Variablen

Der Interpreter ignoriert die Typhinweise und führt den Code aus ...

```python
def my_function(numbers: list[int]) -> int:
    return sum(numbers)

numbers = [1, 2, 3, 4, 5]
print(my_function(numbers))
```

```
15
15
```

Die Entwicklungsumgebung (oder ein externer Type Checker) meckert ...

```
1  def sum_number(a: int, b: int) -> int:
2      return a+b
3
4
5
6
7  sum_number(1, 1.01)
```

```
Argument of type "float" cannot be assigned to parameter "b" of type "int" in function "sum_number"
  "float" is incompatible with "int" Pylance(reportGeneralTypeIssues)
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
```

Beispiel der Woche

Wir nutzen das Newton-Verfahren zur näherungsweisen Berechnung einer Nullstelle. Gesucht wird die Quadratwurzel einer Zahl *a*.

Funktion $f(x) = x^2 - a$, so dass für die Nullstelle gilt: $x^2 = a$

1. In jedem Schritt berechnen wir $x_{n+1} = x_n - f(x_n)/f'(x_n)$.
2. Für unseren Fall: $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$
3. Wir beenden die Iteration, wenn $|x_{n+1} - x_n| < \varepsilon$

newton.py

```python
def square_root(start, a, verbose=False, eps=0.00000001):
  xn = a
  while True:
    if verbose: print(xn)

    # x = xn - (xn**2 - a) / (2*xn)
    # oder:
    x = (xn + a/xn) / 2

    if abs(x-xn) < eps:
      break
    xn = x

  return x

if __name__ == "__main__":
  a = float( input("Value for a (to compute sqrt(a)): ") )
  x = float( input("Inital value for x: ") )
  verbose_output = input("Show all iterations (y/n)? ")
  if verbose_output == 'y':
    verbose_output = True
  else:
    verbose_output = False
  result = square_root(x, a, verbose_output)
  print(f"The zero point of y=x^2-{a} is", result)
  print(f"I.e. the square root of {a} is {result}")
```

```
Value for a (to compute sqrt(a)): Value for a (to compute sqrt(a)):
```

Welches Problem sehen Sie?

Quiz

Weitere Datentypen

Tupel oder Liste?

Welche Vorteile hat der Datentyp `Tupel` gegenüber dem Datentyp `Liste`?

- [ ] `Tupel` sind einfacher zu erstellen.
- [ ] Iterationen über `Tupel` sind schneller als Iterationen über `Listen`
- [ ] `Tupel` können mehr Elemente enthalten
- [ ] `Tupel` benötigen weniger Speicherbedarf

Wie lautet die Ausgabe dieses Programms?

```
a = (9, 2)
print(a[0])
```

- [ ] 9
- [ ] 2
- [ ] Das Programm endet mit einem Error

Wie lautet die Ausgabe dieses Programms?

```
a = (9, 2)
a[0] = 7
print(a[0])
```

- ◯ 9
- ◯ 7
- ◯ 2
- ◯ Das Programm endet mit einem Error

Dictionaries

Wie lautet die Ausgabe dieses Programms?

```
# Hier werden Noten gespeichert
grades = {"Peter": 1.0,
          "Franz": 3.0,
          "Max": 1.7,
          "Jonas": 2.3}

examples = {"Kurt": 1.3,
            "Bernd": 3.3}

examples["Michi"] = 2.0

grades.update(examples)
print(grades["Michi"])
```

- ◯ 2.0
- ◯ Das Programm endet mit einem Error

Sets

Wie viele Elemente befinden sich im Set `st`?

```
st = {"Franz", "Peter", "Franz", "Michi", "Peter"}
print(len(st))
```

Wie lautet die Ausgabe dieser Funktion? (Bitte geben Sie die Antwort ohne geschweifte Klammern an)

```
a = {1,2,3,4}
b = {7,4,6,7}

print(sorted(b - a))
```

Wie lautet die Ausgabe dieser Funktion? (Bitte geben Sie die Antwort ohne geschweifte Klammern an)

```
a = {1,2,3,4}
b = {7,4,6,7}

print(b & a)
```

Wie lautet die Ausgabe dieser Funktion? (Bitte geben Sie die Antwort ohne geschweifte Klammern an)

```
a = {1, 2, 3, 4}
b = {7, 4, 6, 7}

print(b < a)
```

- ◯ True
- ◯ False
- ◯ {1, 2, 3}

Wie lautet die Ausgabe dieser Funktion? (Bitte geben Sie die Antwort ohne geschweifte Klammern oder Leerzeichen an)

```
a = {1,2,3,4}
b = {7,4,6,7}
```

```
print(sorted(b | a))
```

Eigene Funktionen

Syntax

Mit welchem Schlüsselwort starten Funktionsdefinitionen in Python?

Parameterübergabe

Wie lautet die Ausgabe dieses Programms auf 2 Nachkommastellen gerundet?

```
from math import pi

def to_rad(num):
  rad = num * (pi / 180)
  return rad

deg = 90
print(to_rad(deg))
```

Returnwerte

Wie lautet die Ausgabe dieses Programms? Bitte geben Sie die Antwort ohne Klammern an.

```
def get_min_max(a):
    return (min(a), max(a))

a = (10, 47, 18, 1, 33, 20)
result = get_min_max(a)
```

```
print(result)
```

Wie lautet die Ausgabe dieses Programms?

```
def modify_number(a):
    a = -1
    return

a = 42
modify_number(a)
print(a)
```