AUFBAUKURS PYTHON: TAG 1

DATENSTRUKTUREN, FUNKTIONEN, OOP

LERNZIELE

- Wiederholung: Datenstrukturen
- Wiederholung: Funktionen
- Objektorientierte Programmierung kennenlernen

DATENSTRUKTUREN

- Ermöglichen es Daten strukturiert abzulegen
- 4 eingebaute Datenstrukturen:

 Strings `str`  Listen `list`
 Dictionaries `dict`  Tupel `tuple`

STRINGS

```
lorem_ipsum = "Lorem ipsum dolor sit amet, consectetur!"
```

STRINGS: ÜBERBLICK

- Definition: Strings
- Strings anlegen
- wichtige String-Funktionen
- Escape-Sequenzen, String-Prefixes
- reguläre Ausdrücke*

STRINGS

- "Liste" von darstellbaren Zeichen ~ Text
- Beispiele:
  - `"xy3c!"`
  - `"Das ist ein Satz!"`
  - `"\n\r\tHallo\n\r\t"`

STRINGS ANLEGEN

EINZEILIG MIT EINFACHEN ' '

```
a = 'Lorem ipsum dolor sit amet, consectetur.'
```

EINZEILIG MIT " "

```
b = "Lorem ipsum dolor sit amet, consectetur."
```

MEHRZEILIG

```
c = """Lorem ipsum dolor sit amet, consectetur
adipisicing elit. Sunt molestiae corrupti ipsa
placeat sit ea maiores quam repellat corporis
asperiores expedita vero rem eveniet
impedit nulla adipisci, nihil quod, quas?"""
```

STRING-FUNKTIONEN

- Strings bringen Vielzahl an Methoden mit
- Strings manipulieren/bearbeiten

  `str.replace()` `str.format()`
  `str.upper()` `str.lower()`
- Strings auf Eigenschaften prüfen

  `isX()` `startswith()` `endswith()`
- (Teil)strings suchen/finden

Python Dokumentation (3.7.1): String-Methoden

Python Dokumentation (3.7.1): String-Operationen

ESCAPE-SEQUENZEN

- Zeichenfolge beginnend mit `\`
- Use Cases:
  - Darstellung von Zeilenümbrüchen, Tab-Stops etc.
  - Darstellung von Unicode-Zeichen
    - Emojis
    - Spezielle Zeichen

Übersicht: Escape-Sequenzen

STRING-PREFIXES

ROHSTRINGS: `r'lorem ipsum'`

```
user_home_path = r"C:/Users/"
```

- Escape-Sequenze werden ignoriert

FORMATSTRINGS: `f'lorem ipsum'`

```
name,age = "john",25
description = f"My name is {name} and i'm {age} years old."
#Ergebnis: "My name is john and i'm 25 years old."
```

- Spezielle Syntax zum "Befüllen" von Strings

PEP 498: format-Strings

Artikel f-Strings: realpython.com


LIVE-CODING

LISTEN

```
fruits = ["apple", "banana", "mango", "ananas"]
```

LISTEN: ÜBERBLICK

- Definition: Listen
- Listen anlegen
- Slicing
- wichtige Listen-Funktionen
- List Comprehensions

DEFINITION: LISTEN

- ermöglicht Listen von Daten zu speichern
- Kann Daten unterschiedlichen Typs enthalten
- `my_list = ["anna", 5.0, 3, True]`
- Zugriff auf Elemente über Indizes
  - `my_list[0] # "anna"`

INDIZES

0	1	2	3	4	5
[12,	1,	4,	-9,	8,	27]

- Indizes dienen dazu Elemente eines Sequenztyps anzusprechen
- Sowohl positiv

INDIZES

-6	-5	-4	-3	-2	-1
0	1	2	3	4	5

[12, 1, 4, -9, 8, 27]

- Indizes dienen dazu Elemente eines Sequenztyps anzusprechen
- Sowohl positiv als auch negativ möglich

SLICING

- Teilbereiche aus Datenstruktur entnehmen
- Notation
  - `[start:stop:step]`
- **`start` inklusive, `stop` exklusive**

BEISPIELE: SLICING

```
fruits = ["apple", "banana", "mango", "kiwi", "ananas", "grape
```

```
fruits[0:2] # liefert ["apple", "banana"] zurück
```

```
fruits[0:5:2] # liefert ["apple", "mango", "ananas"] zurück
```

```
fruits[3:0:-1] # liefert ['kiwi', 'mango', 'banana'] zurück
```

TIPPS: SLICING

- `start` / `stop` können ausgelassen werden
  - [:2] -> Python nimmt implizit **Start der Liste** an
  - [2:] -> Python nimmt implizit **Ende der Liste** an
- Shortcut um Liste zu kopieren:

```
fruits_copy = fruits[::]
```

AUSZUG: LISTENFUNKTIONEN

Dokumentation: Listen-Funktionen

LIST COMPREHENSION

- Reduzierte Schreibweise um Listen zu erzeugen

```
[<Ausdruck> for <Ausdruck> if <Bedingung> ]
```

- Use Cases:
  - ***Mapping*** von Werten (**`map()`**)
  - ***Filtern*** von Werten (**`filter()`**)

BEISPIEL: MAPPING

```
names = [("Anne","Meier"),
         ("Uli","Scholz"),
         ("Ingo","Müller"),
         ("Friedrich","Schmidt")]

initials = [first[0]+"."+last[0]+"." for first,last in names]
```

BEISPIEL: FILTER

```
satz = "LOREM ipsum dolor SIT amet"
worte = satz.split(" ")
uppercase = [wort for wort in worte if wort.isupper()]
```


LIVE-CODING


DRILL-TIME
MAKE GIFS AT GIFSOUP.COM

TUPEL

```
foo = ("a",5,True)
```

TUPEL: ÜBERBLICK

- Definition: Tupel
- Tupel anlegen
- Tupel Packing/Unpacking
- Unterschiede: Tupel vs. Listen

TUPEL: DEFINITION

- "Container" für heterogene Objekte
- Tupel ist unveränderbar

TUPEL ANLEGEN/BENUTZEN

```
person = ("Philipp", 25, "m")
```

- Tupel können geschachtelt werden..

```
names = [("Anne","Meier","w"),
         ("Uli","Scholz","m"),
         ("Inga","Müller","w")]
```

```
person = (("Philipp","Steinweg"),25,"m")
```

TUPLE PACKING / UNPACKING

TUPLE PACKING:

- Werte werden zu einem Tupel "gepackt"

```
foo = ("A","B","C")
```

TUPLE UNPACKING:

- Werte werden aus einem Tupel "ausgepackt"

```
person = (("Philipp","Steinweg"),25,"m")
name, alter, geschlecht = person
```

UNTERSCHIEDE: TUPEL VS. LISTEN

LISTEN

- für homogene Daten
- Liste ist veränderbar (*mutable*)

TUPEL

- für heterogene Daten
- Tupel ist unveränderbar (*immutable*)


LIVE-CODING

DICTIONARIES

```
person = {      "first_name" : "philipp",
                "last_name" : "steinweg",
                "age": 25}
```

DICTIONARIES: ÜBERBLICK

- Definition: Dictionary
- Dictionary anlegen
- wichtige Funktionen
- JSON*

DICTIONARY: BEISPIELE

- Struktur bestehend aus Paaren (***key*** : ***value***)
- Schlüssel(***key***) muss ..
  - eindeutig sein
  - nicht veränderbar sein
- Werte(***value***) können beliebig oft vorkommen

```
woerterbuch_de_en = {
        "weihnachten": "christmas",
        "apfel": "apple",
        "stift": ["pen","pencil"]
}
```

DICTIONARY: BEISPIELE

- Wörterbuch: "Wort" -> "Übersetzung"

```
country_codes = {"+49" : "germany",
                 "+31": "netherlands",
                 }
```

```
emojis = {":D" : "😄",
          ":O" : "😳",
          ":`D" : "😅"
          }
```

- X -> Häufigkeit

```
audience_answers = {
"Wie heißt die Hauptstadt von Slowenien?" :
                            {
                            "A: Ljubljana": 0.18,
                            "B: Vilnius": 0.22,
                            "C: Warschau": 0.30,
                            "D: Kiew": 0.30
                            }
}
```

DICTIONARY: BEISPIELE

- ID -> Benutzer

```
users = {1: {first_name:"Anna" ,
             last_name:"Meier",
             age:"38"},
         2: {first_name: "Ingo",
             last_name: "Müller",
             age: "24"}
}
```


LIVE-CODING


DRILL-TIME
MAKE GIFS AT GIFSOUP.COM

FUNKTIONEN

```
def function(parameter_1, parameter_2="",*args,*kwargs):
        #do_something()
        #return
```

FUNKTIONEN: ÜBERBLICK

- Definition: Funktionen
- Warum Funktionen?
- Bestandteile einer Funktion
  - Parameter, `*args`, `**kwargs`
  - Rückgabewerte `return`
  - `yield`

DEFINITION: FUNKTIONEN

- Gruppe von Anweisungen unter einem Namen
- Können wie Variablen verwendet werden

GRÜNDE FÜR FUNKTIONEN

- ✔ Erhöht die Lesbarkeit
- ✔ Code wird wiederverwendar
- ✔ Leichtere Verwendbarkeit
- Details können (mental) ausgeblendet werden

FUNKTIONEN DEFINIEREN

- Funktionssignatur:
  - **`def`**
  - Funktionsname
  - Parameterliste
- Funktionsrumpf
  - optional Rückgabewert: **`return`**
  - optional: **`yield`**

```
def greet(name):
        print(f"Hello, {name
```

```
def circumference(radius):
        return 2*3.14*radius
```

PARAMETER / ARGUMENTE

- Informationen, die in Funktion gegeben werden
- Bei Benutzung der Funktion müssen alle Parameter befüllt werden, außer ..
  - Default-Parameter: `end="\n"`
  - `*args`
  - `**kwargs`

KEYWORD ARGUMENTS: *ARGS

- Bei Benutzung von `*args`:
- variable Liste von Parametern
- Liste (args) kann durchlaufen werden

BEISPIEL: *args

```
def get_maximum(*args):
        maximum = args[0]
        for item in args:
                if item > maximum:
                        maximum = item

        return maximum
```

KEYWORD ARGUMENTS:

`**kwargs`

BEISPIEL: **kwargs

```
def print_kwargs(**kwargs):
        for key,value in kwargs.items():
                print(key,":",value)
```

```
print_kwargs(value_1=True,value_2="foo")
```


LIVE-CODING

OBJEKTORIENTIERTE PROGRAMMIERUNG (OOP)

OOP:ÜBERBLICK

- Definition: OOP
- Gründe für OOP
- Klassen, Objekte, Instanzen
- Klassen definieren
- Objekte instanziieren

Zusammenfassung: OOP (realpython.com)

DEFINITION: OOP

- Eine Art "Stil" (Paradigma) Programme zu schreiben
- "Alles ist ein Objekt"
- Idee: Programm besteht aus ***Objekten*** mit ***Eigenschaften*** und ***Verhalten***
- Objekte ***interagieren*** miteinander

FALLBEISPIEL: SMARTPHONE-APP

- Smartphone-App besteht aus ***Ansichten***
- Durch Aktionen navigiere ich zwischen diesen
- Eine Ansicht besteht aus ***Elementen***:
  - Buttons
  - Textfelder
  - Slider
  - ..
- **Was sind die Objekte in diesem Modell?**

DEFINITION: OBJEKT

- Objekt besteht aus:
  - Eigenschaften (Variablen)
  - Verhalten (Funktionen)

BUTTON:

EIGENSCHAFTEN

- Position
- Farbe
- Status

VERHALTEN

- ..bei Knopfdruck
- ..Knopf verschieben

KLASSE/INSTANZ

- Objekte werden durch ***Klassen*** beschrieben
- Objekte werden erzeugt = ***Instanz***

KLASSEN DEFINIEREN

- Klasse wird mit keyword `class` erstellt

```
class Person():
                pass
```

OBJEKTE INSTANZIEREN

- Um Instanzen einer Klasse zu erzeugen: `__init__()`

```
class Person:
        def __init__(self,first_name,last_name):
                self.first_name = first_name
                self.last_name = last_name
```

INSTANZVARIABLEN

- Variablen die sich für jede Instanz unterscheiden
- Stehen innerhalb der **`__init__()`**-Methode

```
class PhoneNumber():
country_codes = {'+49':'germany',
        '+31':'netherlands',
        #...
}
def __init__(self,prefix,number):
        self.prefix = prefix
        self.number = number
        self.country = self.country_codes.get(prefix,"unknown"
```

KLASSENVARIABLEN

- Variable die für alle Instanzen einer Klasse gleich ist
- Stehen außerhalb der `__init__()`-Methode

```
class PhoneNumber():
country_codes = {'+49':'germany',
        '+31':'netherlands',
        #...
}
def __init__(self,prefix,number):
        self.prefix = prefix
        self.number = number
        self.country = self.country_codes.get(prefix,"unknown"
```

VERERBUNG

- Klassen können von anderen Klassen erben
- Kindklasse erbt von der Elternklasse
  - Variablen
  - Methoden
- Kindklasse kann weitere Variablen/Methoden definieren

VERERBUNG

```
class Employee(Person):
        def __init__(self,salary,department):
                self.salary = salary
                self.department = department
```


LIVE-CODING


DRILL-TIME
MAKE GIFS AT GIFSOUP.COM

VIELEN DANK!

BIS MORGEN!