

Glossar

Felix Döring, Felix Wittwer, Anton Obersteiner

Python-Kurs

22. November 2021



Gliederung

1. Basic

- Variablen
- if-elif-else
- print und input
- Funktionen
- Datentypen
- Operatoren
- Listen
- for-Loop und range

- while-Loop
- continue und break

2. Intermediate

- Files
- Exception

3. Klassen

- Klassen, Objekte und Attribute
- Methoden
- Super- und Subklassen
- Spezielle Methoden

Variable

```
1 x = 15
2 y = x + 5
3 x = "Ein kleiner Text zum Thema"
4 print(x, y)
```

[resources/glossar/variables.py](#)

- ▶ Speichern Werte, bis sie neu gesetzt werden
- ▶ Variablen in einer Funktion sind nicht außerhalb sichtbar

if-elif-else

```
1 if role == "Enemy":  
2     print("You shall not pass!")  
3 elif role == "Friend":  
4     print("Welcome")  
5 else:  
6     input("Who are you?")  
7 print("Here we continue")
```

[resources/glossar/if_elif_else.py](#)

- ▶ if-Bedingung wird zuerst geprüft
- ▶ Wenn die nicht gilt (True ist), wird das erste elif überprüft
- ▶ Wenn keine if- oder elif-Bedingung gilt, kann else ausgeführt werden
- ▶ elif und else muss es nicht geben

print und input

```
1 age_as_str = input("How old are you? ")
2 #Text wie "22" in Zahl wie 22 umwandeln:
3 age = int(age_as_str)
4 if age > 16:
5     print("Ok, have a beer")
6 else:
7     print("Nope, you shall not drink")
```

[resources/glossar/print_input.py](#)

- ▶ print gibt alle Argumente aus
- ▶ input gibt seinen prompt aus und wartet auf Nutzerinput
- ▶ Vorsicht: input gibt immer einen String zurück

Funktionen

```
1 def funktion(argumente, argumente_mit_default=default_wert):  
2     #Sachen berechnen / tun  
3     return "ergebnis"  
4  
5 print("was man mit dem ergebnis tut, ..." +  
6       funktion(hier_argumente) +  
7       funktion(hier_argumente, anderes_als_default)  
8 )
```

<resources/glossar/function.py>

- ▶ mit funktion(argumente) aufrufen
- ▶ return-Wert kann weiter verwendet werden
- ▶ alles andere, was die Funktion macht, sind Seiteneffekte

Datentypen

`int` Ganzzahl: 5, -128, `int(17.5) == 17`

`float` Kommazahl: 12.7, -0.001

`str` Zeichenkette: "Text mit 123", "q", `str(5) == "5"`

`bool` Ja-Nein-Wert: `a == 5` → True oder False

`type` Datentyp: `type("q") == str`

`list` Liste mit irgendwelchen Elementen: ["Houille", 1, 12]

`dict` Zuordnung key → element: {1: "ja", 2: "nö"}

```
1 def is_primitive(something):      #list not complete
2     return type(something) in [int, float, str, bool]:
```

<resources/glossar/datatypes.py>

Operatoren

- + Addiert Zahlen, hängt anderes hintereinander
- * Multipliziert Zahlen, vervielfältigt anderes
- Subtrahiert Zahlen
- / Teilt Zahlen
- % Bildet den Rest $a \bmod b$
- ** Exponent: statt ^
- == Sind sie gleich? → bool als Ergebnis

```
1 2 + 2 == 2 * 2 == 2 ** 2
2 5 % 3 == 2 == 8 % 3
3 "=" * 5 + "mitte" + "=" * 5
```

<resources/glossar/operators.py>

Operatoren

`==` gleich?

`!=` ungleich?

`<`, `>` kleiner? größer?

`<=`, `>=` kleiner gleich? größer gleich

`and` gelten *a* und *b*?

`or` gelten *a* und/oder *b*?

```
1 inp = "ein text"
2 print(type(inp) == str or type(inp) == list)
3 inp = 15
4 print(type(inp) == int and inp % 5 == 0)
```

resources/glossar/operators_bool.py

Listen

- ▶ Mehrere Werte mit Reihenfolge
- ▶ Index 0, 1, 2, ..., $\text{len}(\text{liste}) - 1$

```
1 squares = []
2
3 #index:    0    1    2    3    4
4 for i in [2, 3, 5, 7, 11]:
5     squares += [i**2]
6
7 print("Quadrate von P[0:5]:", squares)
```

resources/glossar/list_squares.py

for-Loop und range

- ▶ Syntax: `for a in A: block`
- ▶ `a` durchläuft alle Elemente von `A` einmal
- ▶ jedes mal wird `block` (Eingerücktes darunter) ausgeführt
- ▶ `range(start = 0, stop, step = 1)` ist wie eine Liste von `start` bis `stop`
- ▶ Enden von Bereichen (hier: `stop`) werden nicht mit berechnet!
- ▶ eins größer wählen als letztes Element

```
1 for i in range(10, 101, 10):  
2     print(f"{i}% done!")
```

[resources/glossar/for_range.py](#)

while-Loop

- ▶ Syntax: `while b: block`
- ▶ prüft Bedingung *b* immer wieder
- ▶ führt jedes Mal, wenn *b* gilt, *block* aus

```
1 inp = ""
2 while inp != "quit":
3     inp = input("Enter Kommand (quit to stop):")
4     print("...")
```

<resources/glossar/while.py>

continue und break

`continue` Schleife ignoriert den Rest des Blocks
und geht zum nächsten Durchlauf über

`break` Schleife bricht komplett ab

```
1 for i in range(10, 101, 10):
2     if i == 30:
3         #program continues with 40 -> no 30 in print
4         continue
5     print(f"{i}% done!")
6     if i == 90:
7         #90% will be printed, then this message, then it
8         #will exit
9         print("At 90%, there's a problem, no next loop")
10        break
11    #no 100% :/
```

resources/glossar/continue_break.py

Files

```
1 content = []
2 file1 = open("Source.txt", "r") #r = read
3 for line in file1:
4     content += [line]
5 file1.close()
6
7 file2 = open("Result.txt", "w") #w = write
8 for l in range(len(content)):
9     file2.write(f"{l:03}: {content[l]}")
10 file2.close()
```

[resources/glossar/files.py](#)

```
1 with open("Source.txt", "r") as file:
2     l = 0
3     for line in file:
4         print(f"{l:03}: {line}")
5         l += 1
```

[resources/glossar/files_with.py](#)

Files

```
1 try:
2     #int(...) can fail
3     #example: int("text") raises TypeError
4     a = int(input("Give me an integer: "))
5     print(f"Nice, you gave me {a}")
6 except TypeError:
7     print("You didn't give an integer :(")
```

[resources/glossar/exception.py](#)

```
1 #find the next file number which doesn't exist yet
2 for i in range(100):
3     try:
4         file = open(f"file_{i}", "x") # "x" = new file
5         print(f"Opened File file_{i}")
6     except FileExistsError:
7         print(f"File file_{i} exists, trying next...")
```

[resources/glossar/exception_file.py](#)

Klassen und Attribute

- ▶ Klassen sind wie Schablonen

Klassen und Attribute

- ▶ Klassen sind wie Schablonen
- ▶ Objekte sind konkrete Instanzen

Klassen und Attribute

- ▶ Klassen sind wie Schablonen
- ▶ Objekte sind konkrete Instanzen

```
1 class Klasse:
2     def __init__(self, wert):
3         self.attribut = wert
4
5     def print_attribut(self):
6         print(self.attribut)
7
8 def main():
9     a = Klasse(3)
10    b = Klasse(12)
11    a.attribut = -a.attribut
12    b.print_attribut()
```

[resources/02_grundlagen/class.py](#) auf Deutsch: selbst festgelegt

Attribute

Attribute setzen: meist im **Initialisierer** `__init__`

```
1 class Mensch:
2     def __init__(self, vorname, nachname):
3         self.vorname = vorname
4         self.nachname = nachname
5     def vorstellen(self):
6         return f"Hi, ich bin {self.vorname} {self.nachname}"
7
8 def main():
9     # instanziiert zwei Objekte vom Typ 'Mensch'
10    maria = Mensch("Maria", "Stuhlbein")
11    john = Mensch("John", "Doe")
12    print(maria.vorstellen())
```

[resources/02_grundlagen/attributes.py](#) auf Deutsch: selbst festgelegt

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind
- ▶ erstes Argument `self` wird übergeben

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind
- ▶ erstes Argument `self` wird übergeben
- ▶ `def method(self, arg2): ...`
- ▶ `obj.method(arg2)`

Super- und Subklassen

Erben / Erweitern: mit `class subclass(superclass):` definieren

- ▶ neue Variablen und Methoden hinzufügen, auch alte überschreiben
- ▶ Superklasse mit `super()` aufrufen

```
1 class Mensch():
2     def __init__(self, vorname, nachname):
3         self.vorname = vorname
4         self.nachname = nachname
5
6 class Kind(Mensch):
7     def __init__(self, vorname, nachname, eltern):
8         super(Kind, self).__init__(vorname, nachname)
9         self.eltern = eltern
```

[resources/02_grundlagen/inheritance.py](#)

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

`__str__` Für Ausgabe als String

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

`__str__` Für Ausgabe als String

`__add__` Funktion für `+`-Operator

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

`__str__` Für Ausgabe als String

`__add__` Funktion für `+`-Operator

`__iadd__` Funktion für `+=`-Operator (muss self zurückgeben)

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

`__str__` Für Ausgabe als String

`__add__` Funktion für `+`-Operator

`__iadd__` Funktion für `+=`-Operator (muss self zurückgeben)

`__sub__` Funktion für `-`-Operator

Spezielle Methoden

von Python intern verwendet

`__init__` Bei Instanziierung aufgerufen

`__del__` Bei Löschung aufgerufen (selten)

`__str__` Für Ausgabe als String

`__add__` Funktion für `+`-Operator

`__iadd__` Funktion für `+=`-Operator (muss self zurückgeben)

`__sub__` Funktion für `-`-Operator

`__truediv__` Funktion für `/`-Operator