

Python



Marc Benesch

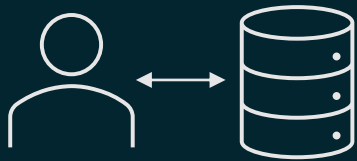
Wiegandweg 4,
80937 München

Mail: marc.benesch@pyucation.de

Web: www.pyucation.de

Tel: +49 152 58521568

Warum Python?



Webentwicklung



Desktop & GUI
Applikationen

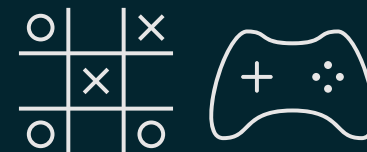


Embedded
Applications

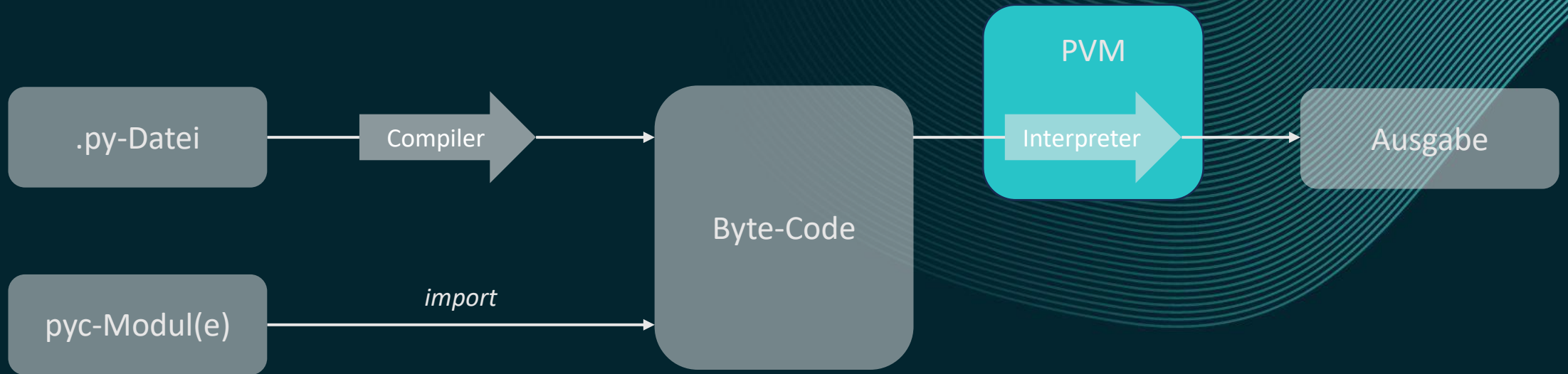
Data Science & ML



Spieleentwicklung*



Kompiliert oder Interpretiert?



Erste Schritte mit Python

- **Ausgabe auf dem Bildschirm**

```
print("Hello World")
```

- **Einrückungen sind wichtig**

```
if 5 > 2:  
    print("5 ist größer als 2")  
if 5 > 2:  
print("5 ist größer als 2")
```



- **Kommentare**

```
# das ist ein Kommentar
```

Datentypen

Datentyp	Bedeutung	Beispiel 1	Beispiel 2
int	Integer = Ganzzahl	3	-7
float	Floating Point Number = Fließkommazahl	5.8	-7.0
str	String = Textzeichen	"Teo"	"7"
bool	Boolean = Wahrheitswert	True	False

Variablen

- **Deklaration und Initialisierung in einer streng typisierten Sprache (Bsp. C)**

```
int x;      // Benennung und Bekanntgabe für den Compiler
x = 3;      // Zufallswert im Arbeitsspeicher mit neuem Wert
            // überschreiben
```

- **in Python ist keine Deklaration notwendig, die Variable wird erstellt, wenn ihr ein Wert zugewiesen wird**

```
x = 3
```

- **der Datentyp kann sich ändern (Python ist nicht streng typisiert)**

```
x = 3          # Typ int
x = "Teo"      # Typ str
```

Variablen

- **Regeln zur Benennung von Variablen**

- muss mit Buchstabe oder Unterstrich `_` beginnen
- darf nicht mit einer Zahl beginnen
- nur alphanumerische Zeichen (A-z, 0-9) und der Unterstrich `_`
- sind „case-sensitive“ (test, TEST, tEsT sind drei unterschiedliche Variablen)

- **Konvention**

- mehrere Wörter werden mit einem Unterstrich getrennt:
`my_var` statt `MyVar` (wie in anderen Sprachen üblich)

Variablenname	<code>client_name1</code>	<code>_internal_count</code>	<code>pay2win</code>	<code>6th_settings</code>	<code>AutoRequest</code>
---------------	---------------------------	------------------------------	----------------------	---------------------------	--------------------------

Gültig?

Typkonvertierung (*type casting*)

- **Typ abfragen**

```
x = 3  
print(type(x))
```

- **Typ konvertieren**

```
x = "3"  
y = 4.5  
  
# str in int  
a = int(x)  
# float in int  
b = int(y)
```


Strings

	G	u	t	e	n		T	a	g
Index	0	1	2	3	4	5	6	7	8
	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Strings sind Zeichenketten von Unicode Zeichen, einfach ausgedrückt: Text
- Strings beginnen und enden entweder mit doppelten " oder einfachen Anführungszeichen '
x = "Guten Tag"
- Strings sind geordnet und erlauben eine sogenannte Indizierung

```
print(x[1])      # Indizierung startet bei 0
print(x[-1])     # negative Indizierung zählt von hinten
```
- das sogenannte *slicing* gibt einen Bereich zurück, das definierte Ende ist in Python exklusiv

```
print(x[2:5])    # output: ten
```

Strings

- **Strings erlauben keine Neuzuweisung von Zeichen (*item assignment*)**

```
x[0] = "J"           # nicht erlaubt  
x = x.replace("G", "J") # es wird eine Kopie erzeugt und x  
                      # wird durch die Kopie ersetzt  
                      # output: Juten Tag
```

- **Länge von Strings**

```
print(len(x))        # output: 9
```

- **Strings addieren (*string concatenation*)**

```
a = "Hallo wie"  
b = "geht es dir?"  
print(a + " " + b)    # output Hallo wie geht es dir?
```










Strings

- **Escape Characters. werden genutzt, um**
 - unerlaubte Zeichen in einem String darzustellen
 - Formatierungshilfen (z. B. Tab, Enter, ...) einzusetzen
 - `\"` # Anführungszeichen, z. B. zum Zitieren)
 - `\\` # für einen einfachen Backslash
 - `\n` # neue Zeile
 - `\t` # neuer Tab
- **es gibt noch viele weitere Methoden für Strings → siehe Dokumentation**
<https://docs.python.org/2.5/lib/string-methods.html>
- **Wissen wo etwas steht > auswendig lernen**

Bools

- **Bools sind Wahrheitswerte** (`True` oder `False`)
- **eine Aussage kann wahr oder falsch sein**
Bsp.: eine mathematische Aussage $7 > 3$ (wahr) oder $7 < 3$ (falsch)
- **Python bietet sogenannte implizite Wahrheitswerte an; Dinge, die implizit `False` sind:**
 - `False`
 - `None`
 - `0`
 - leere Strings, Listen, Tuple, Dictionaries

Bools - Logikoperatoren

Lampe 1	Lampe 2	Lampe 1 UND Lampe 2	Lampe 1 ODER Lampe 2
		False	False
		False	True
		False	True
		True	True

Operatoren

Operator	Name	Beispiel
<code>+, -, *, /</code>	Addition, Subtraktion, Multiplikation, Division	<code>x + y, x - y, x * y, x / y</code>
<code>**</code>	Exponent	<code>10**3</code>
<code>%, //</code>	Modulo, Floor Division	<code>7 % 3 # = 1</code> <code>7 // 3 # = 2</code>
<code>=</code>	Zuweisung	<code>x = 3</code>
<code>+=, -=, *=, /=, %=, //=, **=</code>	gleich wie <code>x = x + 3</code> , Zuweisung und Rechnung in einem	<code>x += 3, x -= 1</code>
<code>>, <, >=, <=</code>	größer (gleich), kleiner (gleich)	<code>x >= y</code>
<code>==, !=</code>	gleich, ungleich (vergleichend)	<code>x == y, x != y</code>
<code>and, or, not</code>	Logikoperatoren, Invertierung	<code>not (3 < 7) # False</code>

Listen

- **Listen werden genutzt, um eine Sammlung von Daten in einer Variable zu speichern, Bsp.:**
`farben = ["rot", "gelb", "blau"]`
- **Listen sind geordnet, Elemente sind austauschbar und Mehrfachnennungen sind erlaubt**
 - **geordnet:** Elemente haben einen bestimmten Platz und sind indizierbar
 - **austauschbar:** Elemente können hinzugefügt, verändert, entfernt werden, nachdem die Liste erstellt wurde
 - **Mehrfachnennungen:** derselbe Wert kann mehrmals vorkommen
- **Listen dürfen aus verschiedenen Datentypen bestehen (wird kaum verwendet)**
`my_list = ["Felix", 21, 5, True, 7.0]`

Listen - Methoden

- **Zugreifen**

```
print(farben[1])           # output: gelb
print(farben[0:2])         # output: rot, gelb
```

- **Ändern**

```
farben[1] = lila           # ["rot", "lila", "blau"]
```

- **Hinzufügen**

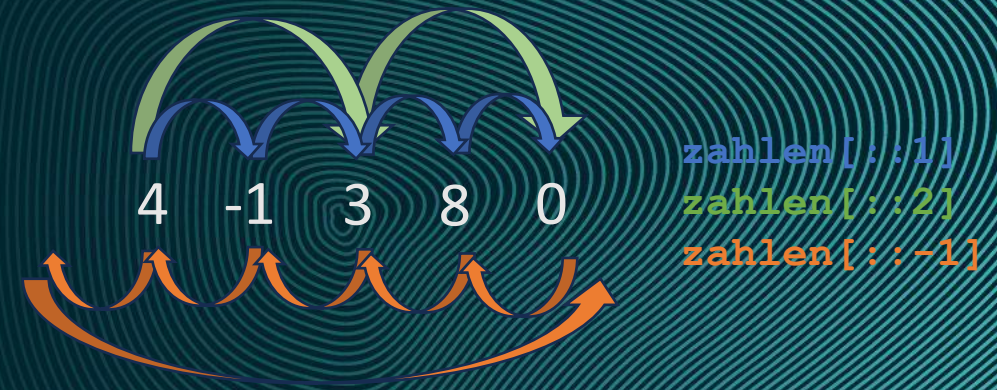
```
# hinten anhängen
farben.append(grün)         # ["rot", "lila", "blau", "grün"]
# an einer Stelle einfügen
farben.insert(1, braun)     # ["rot", "braun", "lila", "blau",
                           # "grün"]
```


Listen - Methoden

- **Entfernen**

```
farben.remove(rot)           # ["braun", "lila", "blau", "grün"]  
# an einer Stelle entfernen  
farben.pop(1)                # ["braun", "blau", "grün"]
```

Exkurs: Slicing



- *slicing* in Python ann vor allem am Anfang sehr verwirrend sein

- **prinzipiell** gilt folgendes:

listenname[start : stop (exklusiv) : schrittweite]

- Bsp.:

```
zahlen = [4, -1, 3, 8, 0]
# wir beginnen bei 2 und enden bei 3 (einen vor 4)
print(zahlen[2:4])                # output: [3, 8]
# wir gehen von Anfang bis Ende in 2er Schritten
print(zahlen[::2])                # output: [4, 3, 0]
```

- Was passiert hier?

```
zahlen[::-1]
```


Listen – Tuple – Sets

- **Tuple und Sets sind ebenfalls sequenzielle Datentypen in Python**

```
farben_tuple = ("rot", "gelb", "blau")  
farben_set = {"rot", "gelb", "blau"}
```

- **die Unterschiede sind:**

	geordnet	veränderbar	Mehrfachnennung
Listen	✓	✓	✓
Tuple	✓	✗	✓
Sets	✗	✗	✗

Dictionaries

- Dictionaries sind wie Wörterbücher aufgebaut
- sie folgen einem *Schlüssel-Werte-Paar-Aufbau* (*key-value pair*)
- Dictionaries waren mal ungeordnet (inzwischen entscheidet die Reihenfolge der Initialisierung), sie sind veränderbar und erlauben keine Mehrfachnennung

- **Bsp.:**

```
steckbrief = {  
    # key: value  
    "name": "Marc",  
    "age": 26,  
    "has_siblings": True  
}
```


Übersicht: *Collections* - Zugriff

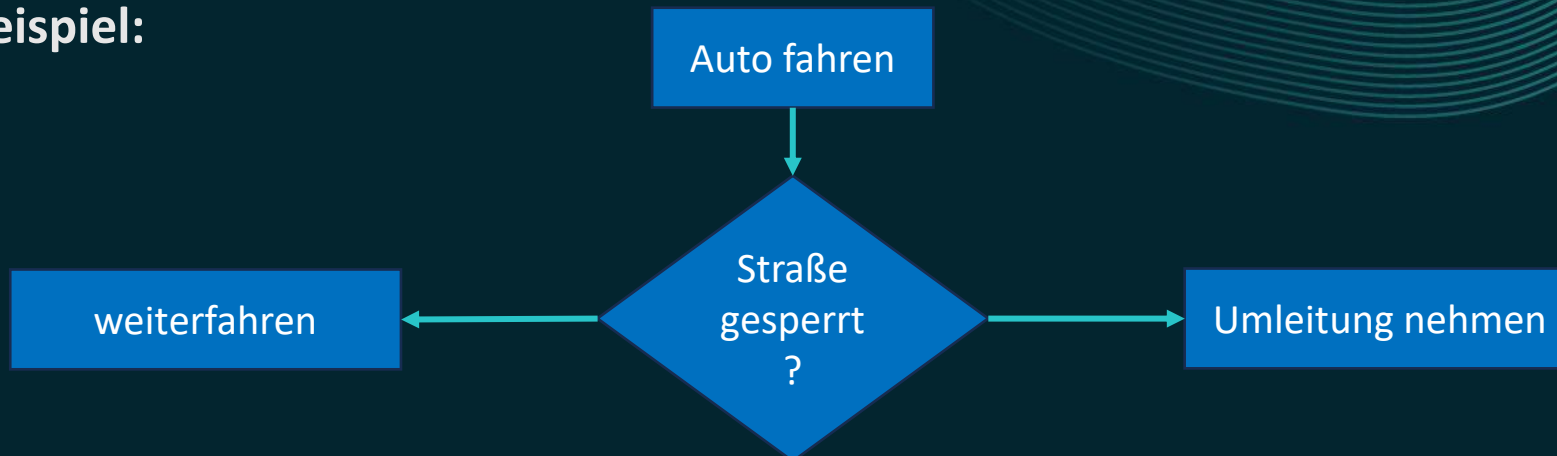
- **Zugreifen auf Listen-Element:**
`my_list[1]`
- **Zugreifen auf Tuple-Element:**
`my_tuple[1]`
- **Zugreifen auf Set-Element:**
`my_set[1]` `# nicht erlaubt`
- **Zugreifen auf Dict-Element (über den Schlüssel):**
`my_dict["age"]`

Zusammenfassung: Variablen & Datentypen

- ✓ Was sind Variablen?
- ✓ Was für verschiedene Datentypen gibt es und was sind die Unterschiede?
 - einfache Datentypen: int, float, str, bool
 - *collections*: list, tuple, set, dict
- ✓ Welche Beispiele für Methoden der einzelnen Datentypen haben wir kennengelernt?
- ✓ Welche Operatoren gibt es?
- ✓ Wie funktioniert Indizierung in Python?

Verzweigungen - Motivation

- bisher hatten wir ausschließlich Code, der von oben nach unten ausgeführt wurde; es wurde nichts ausgelassen oder übersprungen
- jetzt: Fallunterscheidungen, wir können unseren Code aufteilen und für unterschiedliche Fälle unterschiedlichen Code ausführen
- dabei prüfen wir, ob eine (oder mehrere) Bedingung zutrifft oder nicht (ob eine Aussage wahr oder falsch ist)
- Praxisbeispiel:



Verzweigungen - Beispiel

- obiges Beispiel als Python-Code:

```
straße_gesperrt = True
if straße_gesperrt:
    print(Umleitung nehmen)
else:
    print(weiterfahren)
```

Hinweis: es braucht nicht
zwingend einen `else`-Fall

- zu prüfende Bedingung muss immer ein boolscher Ausdruck sein
- Bedingung erfüllt → `if`-Fall wird ausgeführt
- andernfalls wird dieser abgelehnt und der `else`-Fall ausgeführt

Verzweigungen – Mehrere Fälle

- mehrere Fälle lassen sich mit einem `elif` (`else + if`) behandeln
- Bsp.:

```
zahl = 14

if zahl < 10:
    print(zahl * 2)
elif zahl < 20:
    print(zahl / 2)
else:
    print(zahl + 1)
```

Frage: Was wird hier die Ausgabe sein?

Schleifen - Motivation

- sich wiederholende, aufeinander aufbauende Aufgaben können in Schleifen geschrieben werden
- Bsp.: Schreibe ein Programm, welche dir die Zahlen von 1 bis 100 (ohne Gaußsche Summenformel) addiert.

Idee 1: „dummes“ Programm

```
1. summe = 1  
2. summe += 2  
3. summe += 3  
4. summe += 4  
...
```

Idee 1: „schlaues“ Programm

```
1. summe = 0  
2. i = 1  
3. addiere i zur Summe  
4. erhöhe i um 1  
5. wiederhole ab 2.
```


for loops

- ... sind Schleifen, die ein wohldefiniertes Ende haben
- ... nutzen eine Lauf-/Iterationsvariable, die nacheinander jeden Wert eines *listenähnlichen* Objekts annimmt

```
farben = ["rot", "lila", "blau"]  
for farbe in farben:  
    print(farbe)
```

Die Laufvariable wird in jedem Durchlauf der Schleife überschrieben.

- `farbe` ist in diesem Beispiel die Laufvariable, sie enthält in jedem Durchlauf den nächsten Wert der Liste `farben`
- wir gehen die Liste nach und nach durch und führen den Code innerhalb der Schleife aus bis die Schleife „leer“ ist (wenn die Liste durchlaufen wurde)

for loops

- wenn keine Liste vorhanden ist, können wir auch eine vom Programm erstellen lassen

```
# keine Liste gegeben
for i in range(5):
    print("Hallo")
    print(i)
```

- `i` ist diesmal die Laufvariable
- die `range`-Funktion erzeugt für uns ein listenähnliches Objekt, welches wir auch nutzen können, um bestimmten Code mehrmals auszuführen (hier z. B. „Hallo“ auf der Console ausgeben)

for loops

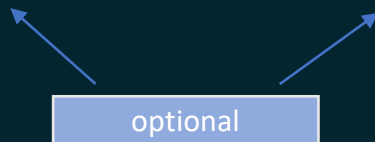
- beide Beispiele machen hier im Grunde das gleiche:

```
for i in range(5):  
    print("Hallo")  
    print(i)
```

```
zahlen = [0, 1, 2, 3, 4]  
for zahl in zahlen:  
    print("Hallo")  
    print(zahl)
```

- der range-Funktion kann (ähnlich zum slicing) auch ein Start-, Endwert und eine Schrittweite übergeben werden:

```
range(start, stop, schrittweite)
```



for loops - Verständnisübung

```
1. fruits = ["apple", "banana", "cherry"]  
   ____ fruit in fruits:  
       print(____)
```

```
2. for __ in ____ (1, 10):  
    print(x + 1)
```

```
3. for ____ in "Python " :  
    print(____)
```

```
4. numbers = [3, 8, 4, 1, 2]  
   for n in ____:  
       print(n == 2)
```


while loops

- mit einer `while` loop führen wir Code aus solange eine Bedingung zutrifft

```
straße_immernoach_gesperrt = True
while straße_immernoach_gesperrt:
    print("Umleitung nehmen")
```

- Was ist die Gefahr hierbei?

while loops – Endlosschleifen vermeiden

- Endlosschleifen sind Schleifen, die theoretisch für immer ausgeführt werden
- bei der Verwendung von `while` loops immer darauf achten, dass diese verlassen werden können

```
i = 0
while i < 10:
    print(i)
    i += 1
```

```
i = 0
while i < 10:
    if i == 5:
        break
    i += 1
```

- `break` kann genutzt werden, um vorzeitig aus einer Schleife auszubrechen (auch wenn die Bedingung noch erfüllt ist)

Funktionen

- eine Funktion ist ein logisch in sich geschlossener Code-Block, der nur ausgeführt wird, wenn die Funktion aufgerufen wird
- Funktionen können Daten in Form von Parametern übergeben werden
- Funktionen können (müssen nicht) Rückgabewerte liefern

```
def print_name(name):  
    print(f"Mein Name ist {name}.")
```

- `def` bedeutet *definition* oder *define*

Funktionen - Beispiel

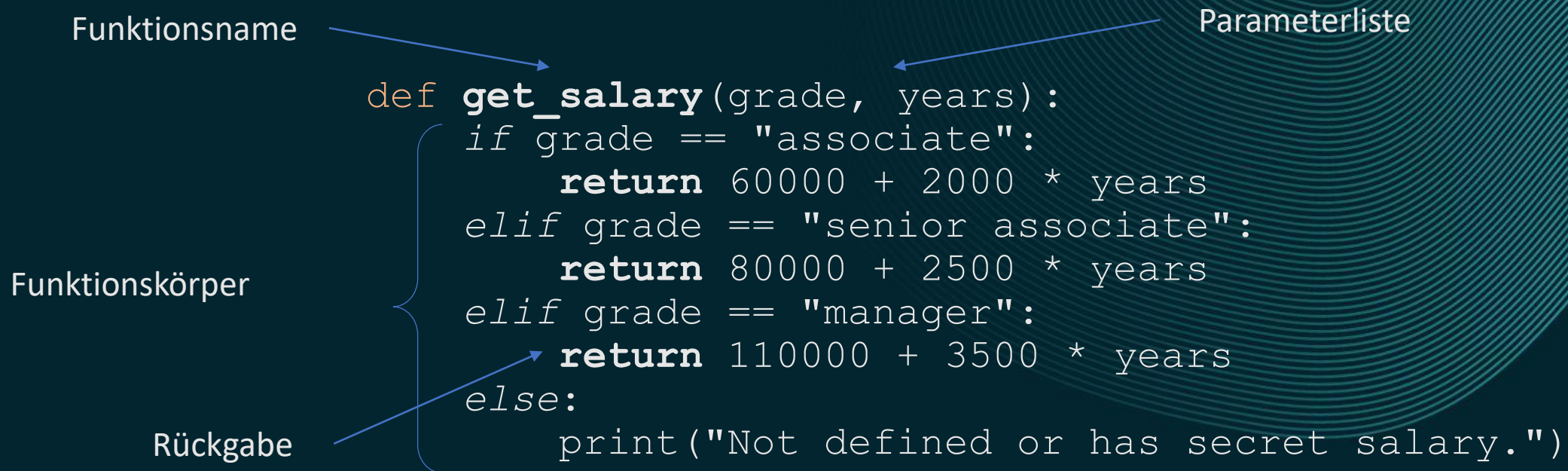
Funktionsname

Parameterliste

```
def get_salary(grade, years):  
    if grade == "associate":  
        return 60000 + 2000 * years  
    elif grade == "senior associate":  
        return 80000 + 2500 * years  
    elif grade == "manager":  
        return 110000 + 3500 * years  
    else:  
        print("Not defined or has secret salary.")
```

Funktionskörper

Rückgabe



Funktionen - Beispiel

- Erinnerung: die Funktion wird erst ausgeführt, wenn sie aufgerufen wird
`my_salary = get_salary("manager", 6)`
- wir können den Rückgabewert speichern
- Funktionen können von außen (aus Sicht des Benutzers) als Blackbox angesehen werden



File Handling – I/O Operationen

- Wofür benötigen wir Dateien in Python? → bisher haben wir ausschließlich im Arbeitsspeicher (RAM) gearbeitet
- Ressourcen (Dateien, Datenbanken, ...) erlauben eine Speicherung auch außerhalb des RAM
- die Reihenfolge ist immer gleich:
 - Öffnen der Ressource
 - Bearbeiten (bspw. lesen, schreiben)
 - Schließen der Ressource
- verschiedene Modi sind:

```
f = open("test.txt", mode="r")  
# irgendwas mit Datei machen  
f.close()
```

Modus	Beschreibung
r	read
w	write
a	append

with Ausdruck

- da es zwischen dem Öffnen und Schließen zu Fehlern kommen kann, ist die Verwendung eines context managers empfohlen:

```
with open("test.txt", mode="r") as f:  
    print(f.read())
```

- dieser garantiert ein ordentliches Öffnen und Schließen

Lesen und Schreiben

- Beispiel für das Lesen:

```
f.read()           # gesamter Inhalt der Datei
f.read(5)          # gibt die ersten 5 Zeichen wieder
f.readline()       # gibt die erste Zeile aus
for line in f:     # auch möglich
    print(line)
```

- Beispiel für das Schreiben:

```
f.write("meine erste Datei") # schreibt eine Zeile
f.write("das ist ziemlich cool")
```


csv-Datei einlesen

- csv steht für comma separated value
- zum Einlesen nutzen wir das Modul csv

```
import csv

with open("my_csv_file.csv") as csvfile:
    # csv.reader gibt ein Iterator Objekt zurück,
    # das über die Zeilen des csv files iteriert
    csv_reader = csv.reader(csvfile, delimiter=",")
    header = next(csv_reader)           # optional, falls
                                         # Kopfzeile

    for row in csv_reader:
        # do something
```

Objektorientierung (OOP)

- OOP = object oriented programming → Programmierparadigma
- Objekte sind quasi Abbildungen der Realität (zumindest kann man es sich so vorstellen)
- Bsp. Auto



mögliche Eigenschaften:

- Farbe
 - max. Geschwindigkeit
 - Hersteller
 - Fahrgestellnummer
-
- jedes Objekt hat Eigenschaften (Attribute) und spezifische Funktionen (= Methoden)
 - Bsp. für Methoden: starte_motor(), umlackieren(), beschleunigen()
 - Methoden beschreiben sozusagen das Verhalten, während Attribute das „Sein“ beschreiben

Klassen

- Klassen sind Blaupausen (engl.: *blueprints*) oder Schablonen/Vorlagen für Objekte
- jedes Objekt ist eine **Instanz** einer Klasse
- Klassen geben den Rahmen vor, Objekte/Instanzen hauchen dem Ganzen Leben ein → eine Klasse enthält meist keine konkreten Informationen zu einem Objekt

Klasse Auto:

- Attribute:
 - Farbe
 - max. Geschwindigkeit
- Methoden:
 - motor_starten()
 - beschleunigen()

(enthält keine
Informationen)



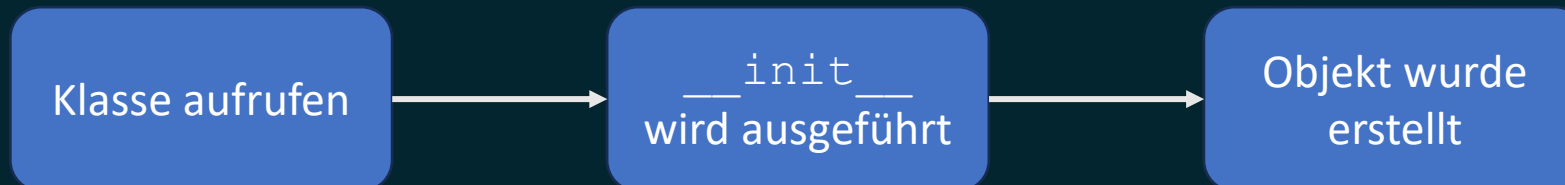
Objekt/Instanz Porsche:

- Attribute:
 - grün
 - 320
- Methoden:
 - motor_starten()
 - beschleunigen()

(enthält „echte“
Informationen)

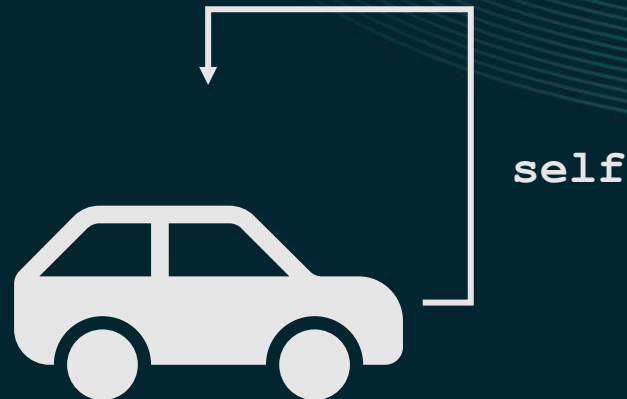
`__init__` () Methode

- *builtin method* von Python, die ausgeführt wird, sobald die Klasse initialisiert wird („ein Objekt geboren wird“)
- in anderen Sprachen: **Konstruktor**
- wir können diese Methode überschreiben und beliebig viele Parameter übergeben
- wichtig: der erste Parameter ist immer eine Referenz auf das Objekt selbst (nächste Folie): **self**
- wir nutzen die `__init__` Methode, um den Objektattributen Leben einzuhauchen



self

- ist ein Parameter/Variable, welche eine Referenz auf die aktuelle Instanz ist
- in anderen Sprachen wird oft **this** verwendet (tatsächlich kann man es in Python auch so nennen, der Name ist frei wählbar, aber die Konvention ist `self`)
- warum brauchen wir das? → um innerhalb der Klasse das Objekt selbst zu referenzieren, wenn sich bspw. ein Attribut ändert



Klassen- vs. Instanzvariablen

- Was ist der Unterschied?

```
# Fall 1
class Auto:
    def __init__(self, farbe, hersteller):
        self.farbe = farbe
        self.hersteller = hersteller
```

```
mein_auto = Auto("grün", "Porsche")
```

```
# Fall 2
class Auto:
    farbe = "grün"
    hersteller = "Porsche"
mein_auto = Auto()
```


Klassen- vs. Instanzvariablen

- Fall 1: **instance attributes**
 - sind Attribute des Objekts selbst, z. B. hat jeder Mensch einen anderen Namen
 - Änderung beeinflusst nur das Objekt selbst
- Fall 2: **class attributes**
 - sind Attribute der Klasse, die alle Objekte gemeinsam haben
 - Änderung beeinflusst alle Objekte der Klasse

OOP-Prinzipien

- **Encapsulation:**

- Kapselung von Daten in einer einzelnen Einheit
- Einschränkung von Zugriff auf bestimmte Objektkomponenten
- hilft bspw. *internal states* zu verbergen (*private, public, protected*)

- **Abstraction:**

- Verstecken von komplexen Implementierungsdetails
- nur essenzielle Komponenten werden gezeigt
- Fokus: Was macht das Objekt? statt Wie macht es das?

*Coding Prinzip: jede Klasse sollte nur eine Funktion/Verantwortlichkeit haben

OOP-Prinzipien

▪ **Inheritance:**

- erlaubt Klassen Attribute und Methoden von anderen Klassen zu erben (*parent child relationship*)
- erlaubt Wiederverwendung von Code und die Erweiterung von Klassen nach dem Prinzip der *single responsibility**

▪ **Polymorphism:**

- heißt: „viele Formen“
- verschiedene Klassen oder Methoden reagieren auf denselben Aufruf unterschiedlich
- Prinzip der Überladung
- Bsp.: + Operator können wir für `str` und `int` verwenden, aber verhält sich unterschiedlich
- z. B. bei Vererbung die Überladung einer Methode der *parent class*

Exception Handling

- in Python: try-except-Blöcke
- Bsp.:

```
try:  
    x = int(input("Gib eine Nummer ein: "))  
except ValueError:  
    print("Ups, das war keine valide Nummer")
```

- Ablauf:
 - Code innerhalb des `try`-Blocks wird versucht abzuarbeiten
 - wenn kein Fehler auftritt, läuft dieser bis zum Ende und `except` wird übersprungen
 - tritt ein Fehler ein, wird der Rest des `try`-Blocks ignoriert und stattdessen der `except`-Block ausgeführt; anschließend wird der Code nach dem `try-except` ausgeführt (keine Weiterführung des `try`-Blocks)
 - wird kein passender Fehler gefunden, geschieht eine *unhandled exception* (kennen wir bereits)

Exception Handling - finally

- optionales `finally` wird ganz am Ende ausgeführt, bevor der gesamte Block endet
- wird, wenn vorhanden, unabhängig von einem Fehler ausgeführt (auch wenn alles „gut“ läuft)
- auch wenn `return`, `break`, `continue` aufgerufen wird, `finally` wird nie übersprungen
- Nutzen in der Praxis? → Freigeben von Ressourcen (Netzwerkverbindungen, Dateien), unabhängig vom Erfolg der Nutzung

```
try:
    result = x / y
except ZeroDivisionError:
    print("Nicht möglich!")
else:
    print(f"Ergebnis: {result}")
finally:
    print("Eine letzte Sache noch ...")
```

