

# Programmierung mit Python für Programmierer

Kapitel 3 - Funktionen, Module, Pakete

Autor: Dr. Christian Heckler

# Vorbemerkungen

- Verwendete Literatur: siehe Referenzen
- Verwendete Symbole:
  - **!**: Beispielprogramm
  - **i**: Weitere Erläuterungen im Kurs
  - **?**: Übung

# Funktionen - der einfachste Fall

Definition einer Funktion:

```
def funktionsname():  
    anweisung(en)
```

- 1 *Funktionskopf*: Schlüsselwort `def` gefolgt von einem beliebigen *Funktionsnamen* und `()` und `:`.  
Für den Funktionsnamen gelten dieselben Regeln und Konventionen wie für Variablennamen.
- 2 *Funktionsrumpf*: Eingerückter *Codeblock* (vgl. Schleifen)  
Im Funktionsrumpf können beliebige Anweisungen stehen, also auch If-Anweisungen, Schleifen, Funktionsaufrufe.

# Aufruf einer Funktion

- Analog zu den bisher verwendeten Funktionen: Angabe des Funktionsnamens gefolgt von Klammern.
- Beim Aufruf werden die Anweisungen des Funktionsrumpfes durchlaufen. Danach kehrt der Programmfluß an die Aufrufstelle zurück.

# Eine einfache Funktion

```
def sage_hallo():  
    print("Hallo")  
  
sage_hallo()
```

# Funktionsparameter

- Wir haben gesehen, dass Funktionen *Argumente* haben können (z.B. `abs(5)` ).
- Diese werden beim Aufruf innerhalb der runden Klammern angegeben.
- Bei der **Funktionsdefinition** werden Variable für die erwarteten Parameter ebenfalls in den runden Klammern angegeben.
- Eine Funktion kann beliebig viele Parameter haben.
- Die Variablen für die Parameter sind nur in der Funktion bekannt (*lokale Variable*) und bekommen die beim Aufruf übergebenen Werte.

# Funktionsparameter - Beispiel

```
def berechne_umfang(laenge, breite): ❶  
    umfang = 2 * (laenge + breite)  
    print("Der Umfang ist", umfang)
```

```
x = 4  
berechne_umfang(x, 7) ❷  
berechne_umfang(2*5, 1+x) ❸
```

- ❶ Die Funktion `berechne_umfang` [\[Kle\]](#) erwartet zwei *Argumente*.
  - Das erste Argument wird während der Funktionsausführung in der (*lokalen*) Variablen `laenge` gespeichert.
  - Das zweite Argument wird während der Funktionsausführung in der (*lokalen*) Variablen `breite` gespeichert.
- ❷ Aufruf der Funktion mit den Werten `4` und `7`. Während der Funktionsausführung hat also die Variable `laenge` den Wert `4` und die Variable `breite` den Wert `7`.
- ❸ Analog: `laenge` erhält den Wert `10` und `breite` den Wert `5`.

# Return und Rückgabewert

- Eine Funktion wird nach der letzten Anweisung des Funktionsrumpfes verlassen.
- Eine Funktion kann explizit mit der Return-Anweisung verlassen werden.
- Mit der Return-Anweisung kann ein Rückgabewert an die Aufrufstelle zurückgegeben werden:

```
return Ausdruck
```

- Es können beliebige Werte zurückgegeben werden, also auch Listen, Dictionaries ...
- Die Auswertung des Funktionsaufrufes ergibt den Rückgabewert, der an der Aufrufstelle wie gehabt verwendet werden kann.
- Wird kein Wert explizit zurückgegeben, so ist der Rückgabewert das `None`-Objekt (Bsp: `print`-Funktion).



# Rückgabewert - Beispiel

```
def berechne_umfang(laenge, breite):  
    umfang = 2 * (laenge + breite)  
    return umfang ❶  
  
print(berechne_umfang(7,5)) ❷  
doppelter_umfang = 2 * berechne_umfang(3,2)
```

- ❶ Beim Erreichen dieser Anweisung wird die Funktion verlassen und der Wert der Variablen `umfang` zurückgegeben.
- ❷ Der Rückgabewert kann an der Aufrufstelle beliebig verwendet werden.

# Parameterübergabe

- Es können beliebige Objekte beliebigen Typs übergeben werden.
- **Es werden Referenzen auf die Objekte übergeben. ⓘ**
- Konsequenzen (siehe die folgenden Beispiele):
  - Weist man in der Funktion einem Parameter einen anderen Wert zu, bleibt der Wert an der Aufrufstelle unverändert.
  - Ändert man in der Funktion über den Parameter das übergebene Objekt (bei veränderlichen Objekten), ist es auch an der Aufrufstelle verändert.



Das nennt man *Seiteneffekt* des Funktionsaufrufs und sollte vermieden werden.

Die Idee einer Funktion ist, für Eingaben (Parameter) ein Ergebnis zu liefern - und nicht, sonstige Auswirkungen auf den Programmverlauf zu haben. Das vereinfacht das Verständnis und die Wartbarkeit (und vereinfacht das Erstellen „paralleler“ Programme).

# Beispiel: Änderung des Wertes eines Parameters

```
def meine_funktion(b):  
    print(b)  
    b = 2  
    print(b) ❶  
  
a = 1  
meine_funktion(a)  
print(a) ❷
```

- ❶ Die lokale Variable `b` hat nun den Wert `2`.
- ❷ Die Variable `a` im Hauptprogramm hat nach wie vor den Wert `1`

# Erläuterung

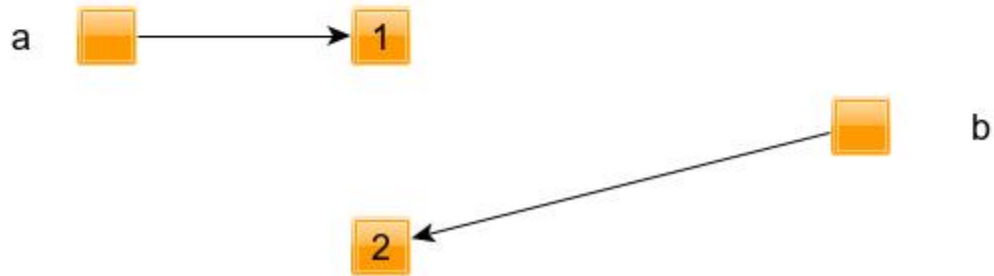
# Veranschaulichung

Hauptprogramm

meine\_funktion



`b=2`



# Beispiel: Änderung des übergebenen Objektes

```
def meine_funktion(l):  
    l[0] = 2 ❶  
  
t = [1]  
meine_funktion(t)  
print(t[0]) ❷
```

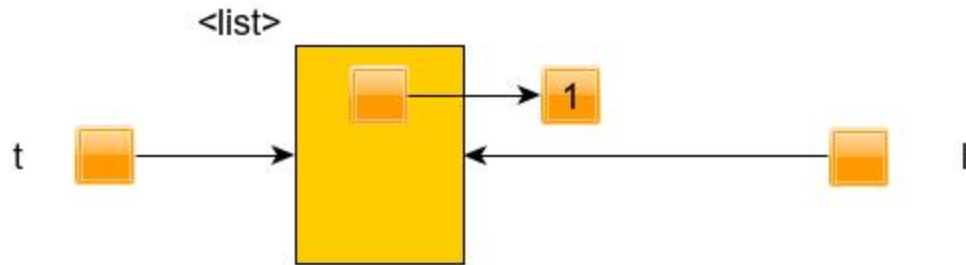
- ❶ Das 0. Element der übergebenen Liste wird verändert.
- ❷ Damit ist auch die Liste an der Aufrufstelle verändert. Ausgabe ist also 2.

# Erläuterung

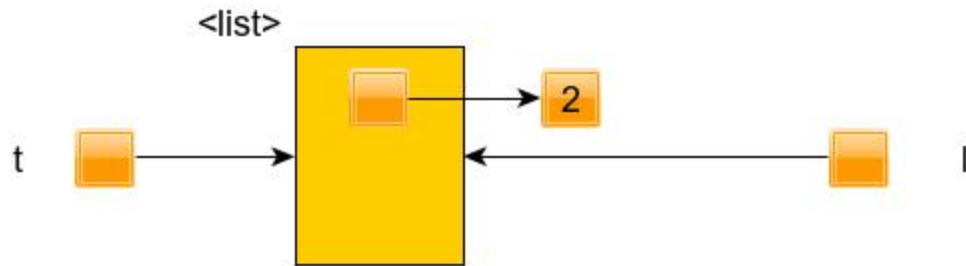
# Veranschaulichung

Hauptprogramm

meine\_funktion



$l[0]=2$





# Rückgabewert

- Analog zur Parameterübergabe wird bei der Rückgabe eine Referenz auf das Rückgabeobjekt übergeben, siehe `rueckgabe.py` !
- Es können beliebige Objekte zurückgegeben werden, also auch Tupel:

```
def berechne_umfang_flaeche(laenge, breite):  
    umfang = 2 * (laenge + breite)  
    flaeche = laenge * breite  
    return (umfang, flaeche)  
  
print(berechne_umfang_flaeche(2,3))  
umfang, flaeche = berechne_umfang_flaeche(4,8)  
print("Umfang:", umfang, ", Flaeche:", flaeche)
```

# Lokale Variable

- Variablen, die innerhalb einer Funktion „deklariert“ sind:
  - Die Funktionsparameter.
  - Variablen, denen innerhalb der Funktion ein Wert zugewiesen wird.
- Sind nur innerhalb der zugehörigen Funktion *sichtbar* (d.h. „zugreifbar“).

```
def erhoehe(x):  
    y = x + 1  
    return y
```

```
print(x) # Fehler: x ist hier nicht bekannt  
print(y) # Fehler: y ist hier nicht bekannt
```

# Globale Variable

- Sind außerhalb von Funktionen definiert.
- Sind in Funktionen *sichtbar* („zugreifbar“), wenn sie vor der Funktionsdefinition erzeugt wurden.

```
x = 1

def erhoehe():
    return x + 1

print(erhoehe())
```

# Überdeckung globaler Variablen

- Eine globale Variable kann durch eine lokale Variable *überdeckt* werden.
- In diesem Fall wird innerhalb der Funktion auf die lokale Variable zugegriffen und außerhalb auf die Globale.

```
s = "global"
```

```
def meine_funktion():  
    s = "lokal"  
    print(s)
```

```
meine_funktion() # lokal  
print(s) # global
```

# Ändernder Zugriff auf eine globale Variable

- Durch die Zuweisung eines Wertes an die Variable `s` innerhalb der Funktion wurde eine neue lokale Variable `s` angelegt.
- Wie kann man in einer Funktion den Wert einer globalen Variablen ändern? Deklaration als `global`!

```
s = "global"
```

```
def meine_funktion():  
    global s  
    s = "lokal"  
    print(s)
```

```
meine_funktion() # lokal  
print(s) # lokal
```

# Empfehlung zu globalen Variablen

- Vermeiden Sie den Zugriff auf globale Variable in Funktionen.
- Benennen Sie globale und lokale Variable unterschiedlich, so dass es gar nicht erst zu Namenskonflikten kommen kann.

# Standardwerte für Funktionsparameter

- Bei der Definition einer Funktion kann man einem Parameter einen Wert zuweisen (*Standardwert, Default*). Wird das Argument beim Aufruf nicht angegeben, so wird der Standardwert genommen. Der Parameter kann beim Aufruf also weggelassen werden (*optionaler Parameter*). Bsp. [\[Kle\]](#):

```
def umfang(laenge, breite=1):  
    return 2 * (laenge + breite)  
  
u = umfang(4) # breite hat in der Funktion den Wert 1
```

- Standardwerte können nur „von hinten“ vergeben werden. (Damit beim Aufruf die Zuordnung möglich ist.)

```
def f1(x,y=2,z=3): # moeglich  
    return  
  
def f2(x=1,y,z=3): # verboten  
    return
```

# Schlüsselwortparameter

- Bisher: Übergabe der Parameter an Hand der Position (*Positionsparameter*)

```
def umfang(laenge=1, breite=2):  
    return 2 * (laenge + breite)  
  
print(umfang(3,7)) # laenge ist 3, breite ist 7  
print(umfang(3))   # laenge ist 3, breite ist 2
```

- Es ist auch möglich, die Parameter per Schlüsselwort zu übergeben.  
(*Schlüsselwortparameter*)

```
u = umfang(laenge=3, breite=7)  
u = umfang(breite=7, laenge=3)  
u = umfang(laenge=3)  
u = umfang(breite=7) # das geht nicht über Positionsparameter
```



# Schlüsselwortparameter 2

```
def umfang(laenge, breite):  
    return 2 * (laenge + breite)  
  
laenge = 1  
print(umfang(1,2))  
print(umfang(laenge = 2 * 5, breite = 7))  
print(umfang(laenge = laenge, breite = laenge + 2)) ❶
```

- ❶ In der Funktion hat der Parameter `laenge` den Wert 1 und der Parameter `breite` den Wert 3.
- „Links“ vom Gleichheitszeichen steht der Parametername, wie er in der Funktion definiert ist ( `laenge` bzw. `breite` )
- „Rechts“ vom Gleichheitszeichen steht ein **Ausdruck** (z.B. die (globale) Variable `laenge` ). Dieser wird ausgewertet. Der Wert wird dem Parameter in der Funktion übergeben.

# Funktionsparameter - was noch

- Funktionen, die eine variable Anzahl von Parametern bekommen:
  - Positionsparameter: \*
  - Schlüsselwortparameter: \*\*
- Siehe beispielsweise: <https://www.python-kurs.eu/parameter.php>

# Kommandozeilenparameter

- Parameterübergabe an ein Pythonprogramm beim Programmaufruf.
- Z.B.: `python pythonprog.py parameter1 parameter2`
- Bsp ( `kommandozeile.py` ):

```
import sys
for argument in sys.argv:
    print(argument)
```

- Aufruf: `python kommandozeile.py hallo welt`
- `sys.argv` ist eine Liste der Parameter ( `argv[0]` = Python-Prg.)
- Bemerkung: Zu `"import sys"`: s. Kapitel „Module“

# docstring

- Dokumentation von Funktionen
- Steht in dreifachen Anführungszeichen direkt hinter der Funktionsdefinition
- Wird im Attribut `doc` des Funktionsobjektes gespeichert.
- Wird bei `help(funktion)` ausgegeben.
- Bsp: `docstring.py` !

# Funktionsreferenzen

- Funktionen sind Objekte.
- Der Funktionsname ist eine Referenz auf das Funktionsobjekt.
- Eine Funktion kann also auch einer Variablen zugewiesen werden:

```
def f(x):  
    print(x)  
  
def g(y):  
    print(y+1)  
  
meine_funktion = f  
meine_funktion(4)  
meine_funktion = g  
meine_funktion(4)
```

- Oder als Parameter einer anderen Funktion übergeben werden.

# Funktionsreferenzen 2

- Unterscheide:

```
meine_funktion = f      ①  
wert = f()             ②
```

- ① Die Funktion `f` wird der Variablen `meine_funktion` zugewiesen.
- ② Die Funktion `f` wird aufgerufen und der Rückgabewert der Variablen `wert` zugewiesen.

# Anwendungsbeispiel: Sortierung von Listen

- Erinnerung: Eine Liste ist eine Sequenz von Objekten

```
l = []  
l = [1, "Hallo"]  
l.append("Welt")
```

- `l.sort()` : Die Liste wird sortiert (`l` wird verändert)
- `s = sorted(l)` : `s` ist eine neue sortierte Liste. `l` ist unverändert.
- `l.sort(reverse=True)` : Umgekehrte Sortierreihenfolge

- Sortierreihenfolge?

Man kann einer Sortierfunktion eine Funktion mitgeben, die auf die Elemente angewendet wird. Verglichen werden dann nicht die Elemente selbst, sondern die Funktionswerte.

# Sortierung von Listen mit **key**-Funktion

- Beispiel:

```
namen = [("Anton", "Wunderlich"),  
          ("Berta", "Müller"),  
          ("Thomas", "Schmitz")]
```

- Sortierreihenfolge?

- nach Vornamen
- nach Nachnamen

- Sortierung nach Vornamen: **i**

```
def vorname(t):  
    return t[0]  
namen.sort(key=vorname)
```


- siehe `funktions_ref.py` **i**



# Anonyme Funktionen

- Die Vornamen bzw. Nachnamen-Funktion ist extrem simpel. Eine „normale“ Funktionsdefinition bläht den Quelltext auf.
- Alternative: Anonyme Funktion via `lambda`-Ausdruck: `key=lambda t: t[0]`
- Siehe: `lambda.py`
- Auf das Schlüsselwort `lambda` folgt:
  - Parameterliste
  - Doppelpunkt
  - Ausdruck (dessen Ergebnis zurückgegeben wird)
- Kontrollstrukturen sind also nicht möglich.

# Closures

- Für Anhänger der funktionalen Programmierung.
- Eine Funktion (ein Funktionsobjekt) kann den Zugriff auf den Erstellungskontext erhalten.
- Das nennt man Closure.
- Beispiel: `closure.py` 

# Generatoren

- Funktion, die bei jedem Aufruf einen weiteren Wert zurückgibt.
- Werte werden also nicht a priori berechnet und zurückgegeben.
- Vorteile:
  - Spart Platz, die Gesamtergebnismenge muss nicht auf einmal im Speicher gehalten werden.
  - Kann Zeit sparen, wenn zum Beispiel nach dem 100. Element festgestellt wird, dass die nächsten 10.000 Elemente nicht mehr benötigt werden; also auch nicht mehr berechnet werden müssen.
- Statt mit return wird das Teilergebnis mit yield zurückgegeben.
- Endgültige Beendigung der Funktion durch return.
- s. `fib_liste.py` und `fib_gen.py` !

# Generator-Ausdrücke

- Ausdruck für Generator
- Also Definition eines Generators, ohne eine Funktion zu schreiben
- Bsp.: `x*x for x in range(101)`
- Ausdruck für einen Generator für die ersten 100 Quadratzahlen
- Bsp.: `sum(x*x for x in range(101))`
- Damit lassen sich insbesondere Listen elegant erzeugen:
- Bsp.: `liste = [x*x for x in range(100)]`
- Dafür gibt es dann sogar einen eigenen Begriff: **Listen-Abstraktion** oder **List Comprehension**
- [https://www.python-kurs.eu/python3\\_list\\_comprehension.php](https://www.python-kurs.eu/python3_list_comprehension.php)

# Die Funktion `map`

- Alternative zur Listenabstraktion
- `map(f, s)` wendet die Funktion `f` auf jedes Element der Sequenz `s` an und gibt einen Iterator zurück
  - (den man in eine Liste umwandeln kann oder
  - über den man mit `for` schleifen kann).
- Beispiel: Summe der Quadratzahlen von oben:

```
liste = list(map(lambda x: x*x, range(101)))
```

# Die Funktion `reduce`

- Reduziert eine Sequenz auf einen Wert durch fortwährende Anwendung einer Funktion auf je zwei Sequenzelemente.
- Muss importiert werden:

```
from functools import reduce
```



- Beispiel: Summe der Quadratzahlen von oben

```
reduce(lambda x,y: x+y, liste)
```

# Einschub: Funktionale Programmierung statt Schleifen


- Funktionen sind also „normale“ Objekte, die man Variablen zuweisen kann und die man als Parameter anderen Funktionen übergeben kann.
- In der „funktionalen Programmierung“ macht man sich dies zu Nutze.
- Beispielsweise Verwendung von Funktionen statt Schleifen
- Motivation: In manchen Fällen sind Schleifen in Python zu langsam (z.B. Datenanalyse großer Datenmengen).
- Die entsprechenden Bibliotheken stellen dann andere Mechanismen bereit, z.B.
  - Anwendung einer Funktion auf alle Elemente einer Liste (ohne Schleife): `map`
  - Anwendung einer Operation auf je zwei Elemente einer Liste: `reduce`

# Ausnahmebehandlung - Motivation

- Im „Nachtbetrieb“ werden „viele“ Datensätze bearbeitet. Was passiert, wenn beim einem Datensatz ein Fehler auftritt (z.B. „Division durch Null“)?
- Das Programm bricht ab (siehe  `ex1.py` )!
- Wünschenswert wäre aber, dass der Datensatz ignoriert wird, und das Programm mit der Verarbeitung des nächsten Datensatzes weitermacht.
- Wie kann man das erreichen? Siehe  `ex2.py` .
- Nachteile:
  - Der Code wird aufgebläht. Die eigentliche Fachlichkeit ist nicht mehr zu erkennen.
  - Man muss an jeder Stelle im Code überlegen, was im Falle eines unerwarteten Ereignisses zu tun ist.




# Ausnahmen fangen

- Bessere Lösung: Siehe  `ex3.py`.
- Ausnahmen, die in einem *Codeblock* geworfen werden ( `try` ), können „gefangen“ werden.
- In einem `except` -Block wird definiert, was zu tun ist, wenn die Ausnahme auftritt.
- Ausführung:
  - Die Anweisungen zwischen `try` und `except` werden ausgeführt.
  - Tritt keine Ausnahme auf, wird der `except` -Block übersprungen.
  - Tritt eine Ausnahme auf, so wird sofort bei deren Auftreten in den `except` -Block gesprungen.


# Ausnahmetypen

- Mit `except Exception` (wie im Beispiel) werden alle Arten von Ausnahmen gefangen (was man aber nicht immer will).
- Es können auch „Unterarten“ geworfen und (verschieden) behandelt werden.
- Beispiel: `int("Hallo")` liefert einen `ValueError`.
- Man hätte also auch `except ValueError` schreiben können.
- Tritt eine Ausnahme auf, die nicht abgefangen wird, wird die Ausnahme weiter „nach oben“ geworfen.
- Beispiele:
  - **!** `ex4.py` : Abfangen von zwei Ausnahmen
  - **!** `ex5.py` : Division durch 0 wird nicht gefangen

# Ausnahmen - else

- Optional.
- Hinter den `except`-Blöcken.
- Wird nur ausgeführt, wenn keine Exception aufgetreten ist.
- Vorteil gegenüber dem Fall, dass die Anweisungen im `try`-Block stehen: Ausnahmen werden nicht „aus Versehen“ gefangen.
- Beispiel:  `else.py`

# Ausnahmen - finally

- Optional
- Der Block hinter `finally` wird auf jeden Fall ausgeführt, egal ob eine Ausnahme geworfen wurde oder nicht.
- Auch dann, wenn die Ausnahme nicht gefangen wird.
- Eignet sich daher gut für Aufräumarbeiten, z.B. das Schließen von Dateien.
- Ganz am Ende (hinter `else`, wenn vorhanden)
- Beispiel:  `finally.py`

# Ausnahmen werfen

- Wenn im eigenen Programm eine Situation auftritt, die an dieser Stelle nicht erwartet wurde, kann man auch selbst eine Ausnahme werfen (Bsp: als Entwickler der `int`-Funktion):


```
raise Exception("Fehler: ...")
```

- Beispiel: 🚫 `ex6.py`
- Statt eine Ausnahme vom Typ `Exception` zu werfen, kann man auch eigene Ausnahmetypen definieren (was i.a. auch besser ist). Das kommt aber später (neue Klasse im Sinne der Objektorientierten Programmierung)!

# Ausnahmen werfen - Anmerkungen

- **Empfehlung:** Ausnahmen nur werfen, wenn ein unerwarteter Fall auftritt, nicht, um (erwartete) Programmlogik zu implementieren.
- Es ist eine (manchmal schwierige) Entwurfsentscheidung, wann man Ausnahmen benutzt und wann eine „normale“ Behandlung (z.B. via if – Abfrage).

# Module - Motivation

- Motivation von Funktionen: Wiederverwendung
  - des eigenen Codes
  - aber auch von „Fremd-Code“.
- Wie kann man nun Funktionen, die in einem Programm definiert sind, in anderen Programmen (wieder-) verwenden?
- Lösung: Import der Python-Datei, in der die Funktionen definiert wurden. In diesem Fall nennt man die Python-Datei „Modul“.
- Beispiel:  `mod1.py` , `mod2.py`
- In einem Modul kann man Funktionen zu einem Thema bündeln (z.B. mathematische Funktionen, Funktionen zur String-Verarbeitung, Funktionen für Dateibehandlung).
- Module stellen also eine weitere Ebene der Strukturierung des Quellcodes dar.

# Module - Anwendung

- Import eines Moduls mit der Import-Anweisung (Dateiname der Python-Datei ohne Endung .py)

```
import mod1
```

- Die Funktionen und globalen Variablen, die in der importierten Python-Datei definiert wurden, können dann durch voranstellen des Modulnamens verwendet werden:

```
mod1.f1()
```



# Module - Suchpfad

- Wo werden Module gesucht, die per Import geladen werden sollen?
  - im aktuellen Verzeichnis
  - im Standardpfad der Python-Installation (z.B. math)
  - Umgebungsvariable PYTHONPATH

# Module Import

- Abkürzung:

```
import mod as m  
m.f1() # statt mod.f1()
```

- Direkter Import einer Funktion in den Namensraum:


```
from mod import f1  
f1() # statt mod.f1()
```

- Es gibt noch andere Möglichkeiten zum Import, die hier nicht besprochen werden sollen (s. [https://www.python-kurs.eu/python3\\_modularisierung.php](https://www.python-kurs.eu/python3_modularisierung.php))

# Module - Was passiert beim Import

- Was passiert eigentlich genau bei dem Import eines Moduls (also einer Python-Datei)?
- Der Interpreter führt die Datei aus, d.h.
  - Funktionsdefinitionen und globale Variablen sind danach an der Aufrufstelle bekannt – mit dem Modulnamen als Präfix (das ist der eigentliche Sinn des Imports).
  - „Normale Anweisungen“ werden aber auch ausgeführt.
- Letzteres ist normalerweise nicht gewünscht. Was kann man dagegen tun?
- Die Anweisungen, die nicht in Funktionen stehen und die nicht beim Import ausgeführt werden sollen, hinter if-Block:



```
if __name__ == "__main__":
```

- Beispiel:  `mod3.py` , `mod4.py` .

# Pakete

- Wie kann man zusammengehörige Module gruppieren?
  - Zusammenfassung zu einem Verzeichnis!
  - Das nennt man dann im Python-Sprech „Paket“
  - Ein Paket ist also zunächst nichts anderes als ein Verzeichnis.
- Eine Besonderheit gibt es dann doch: Datei `init.py` im Verzeichnis (s. später).
- Pakete werden dort gesucht, wo auch Module gesucht werden.
- Pakete sind also nach Funktionen und Modulen eine weitere Ebene der Strukturieren des Quellcodes

# Verwendung von Paketen

- Wie verwendet man Module aus einem Paket?
  - Import des Moduls über den vollständigen Pfad (vgl. Pfad im Betriebssystem, aber Trennung mit Punkt statt /). Beispiel:  `pak1.py`
  - Import des Moduls, wenn im Paket die Module via der Datei `init.py` eingeladen werden: Beispiel:  `pak2.py`
- Wie ein Verzeichnis Dateien und weitere Verzeichnisse enthalten kann, kann ein Paket Module und auch weitere Pakete enthalten.

# Pakete - Import

Was passiert beim Import eines Pakets?

- Beim Import eines Pakets wird die Datei `init.py` in dem entsprechenden Ordner ausgeführt.
- Dort kann man also die Importe der Module durchführen.

# Module und Pakete - Zusammenfassung

## Modul

- Ein Modul ist eine Python-Datei.
- Beim Import eines Moduls wird die Datei ausgeführt, wodurch Funktionen und globale Variable an der Aufrufstelle bekannt sind.

## Paket

- Ein Paket ist ein Verzeichnis mit der Datei `init.py`
- Beim Import wird diese Datei ausgeführt.
- Sie kann also dazu verwendet werden, die Module (=Python-Dateien) in dem Paket (=Verzeichnis) zu importieren. Muss man aber nicht.

# 🔍 Übungen Funktionen

- Binäre Suche
- Merge Sort
- Damenproblem
- Bestimmung der maximalen Wortlänge in einer Zeichenkette, die einen Satz darstellt, mit Hilfe von map / reduce.



# Referenzen

- [Ste] Ralph Steyer: Programmierung Grundlagen, Herdt-Verlag
- [Kle] Bernd Klein: Einführung in Python 3, Hanser-Verlag
- [Kof] Kofler: Python – Der Grundkurs, Rheinwerk Computing
- [EK] Johannes Ernesti, Peter Kaiser: Python 3 – Das umfassende Handbuch, Rheinwerk Computing
- [Mat] Eric Matthes: Python Crashkurs – Eine praktische, projektbasierte Programmierereinführung, dpunkt.verlag
- [Swe] Sweigart: Eigene Spiele programmieren: Python lernen