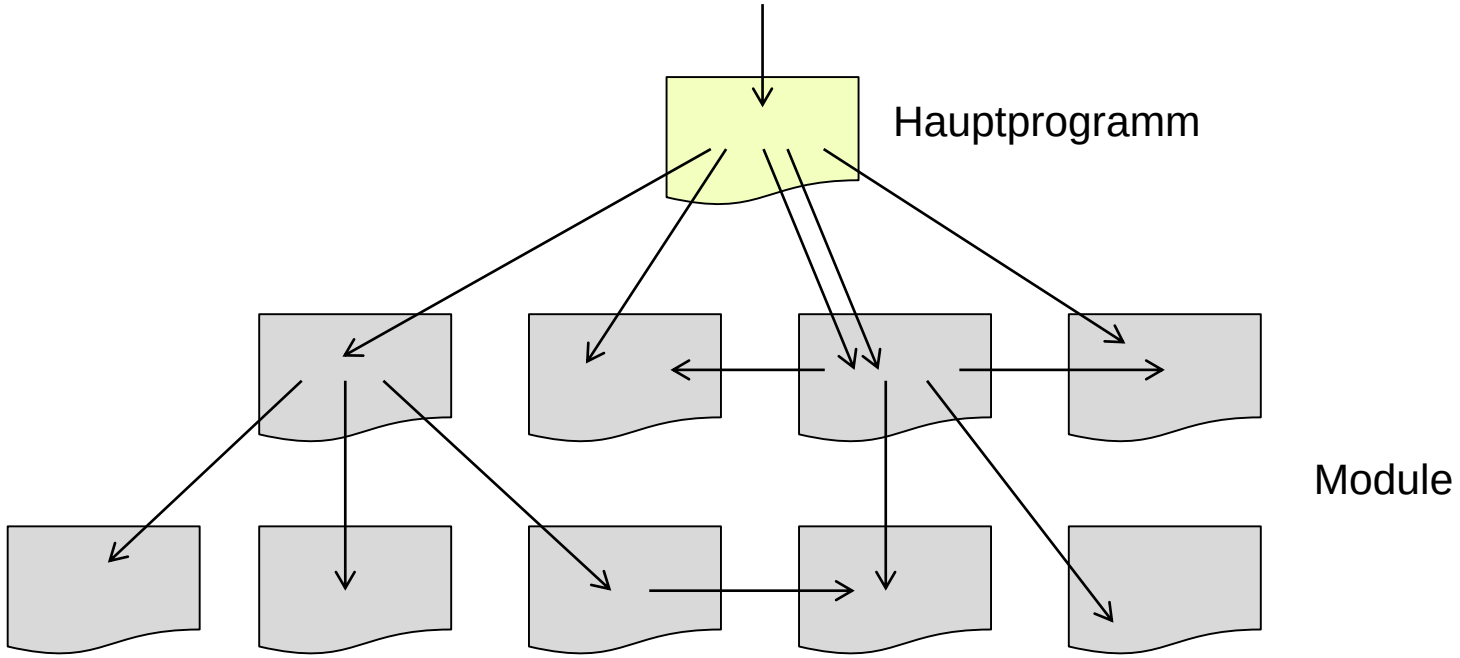
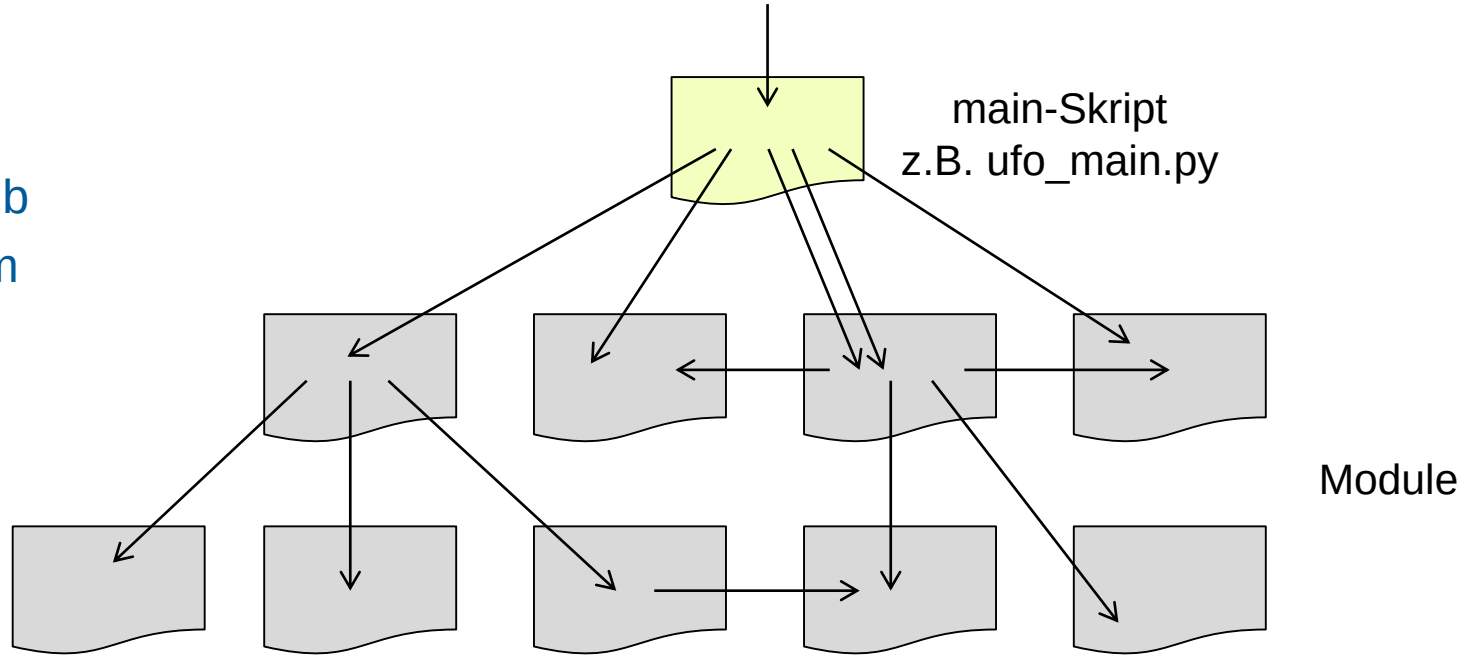
Module und testgetriebene Entwicklung (1)

- Größere Programme werden in mehrere (oder viele) Dateien aufgespalten. Einzelne Programmdateien oder Gruppen von inhaltlich zusammengehörenden Dateien werden oft auch **Module** genannt.
- In (mindestens) einer Datei muss aber das Hauptprogramm stehen. Das **Hauptprogramm** wird beim Programmstart ausgeführt und ruft i.A. weitere Funktionen auf.



Module und testgetriebene Entwicklung (2)

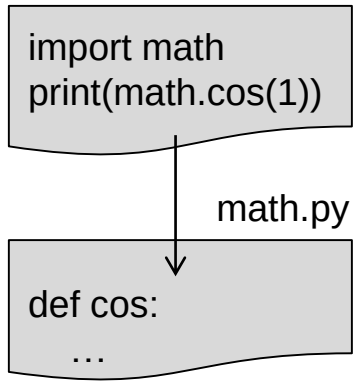
- Achtung: In vielen Programmiersprachen wird das Hauptprogramm durch einen speziellen Namen gekennzeichnet. In Java beispielsweise ist das Hauptprogramm eine Funktion mit dem Namen main.
- In Python wird ein Programm durch Aufrufen eines Skripts gestartet, z.B. python ufo_main.py. Dieses Skript könnte man auch main-Skript nennen. Das Hauptprogramm ist deshalb derjenige Code im main-Skript, der sich außerhalb von Funktionen befindet.
- Code in Modulen, der außerhalb von Funktionen steht, wird beim Import ebenfalls ausgeführt.



Module und testgetriebene Entwicklung (3)

- Um Funktionen aus einem anderen Modul verwenden zu können, muss das Modul importiert werden. Beispiel:

```
import math
print(math.cos(1))
```



- Man kann auch einzelne Funktionen importieren. Dadurch wird die Schreibweise beim Aufruf kürzer. Beispiel:

```
from math import cos
print(cos(1))
```

- Weiterhin ist es möglich, Module zu importieren, sie dann aber unter anderem Namen zu verwenden. Beispiel:

```
import math as m
print(m.cos(1))
```

Module und testgetriebene Entwicklung (4)

- Bei der Entwicklung von Funktionen ist der Test besonders wichtig.
- Beim Testen werden Argumente ausgewählt, die Funktion mit diesen Argumenten aufgerufen und der Rückgabewert mit dem erwarteten Ergebnis verglichen. Ein solcher Testaufruf wird **Testfall** genannt.
- Da jede nicht-triviale Funktion mehrere Abläufe hat, wird ein einziger Testfall kaum ausreichen.
- Zum Testen verwenden wir Testaufrufe im Hauptprogramm.
- Es gibt aber auch spezielle Module, die den Test von Funktionen unterstützen. Bekannt ist das Python-Modul unittest.

Module und testgetriebene Entwicklung (5)

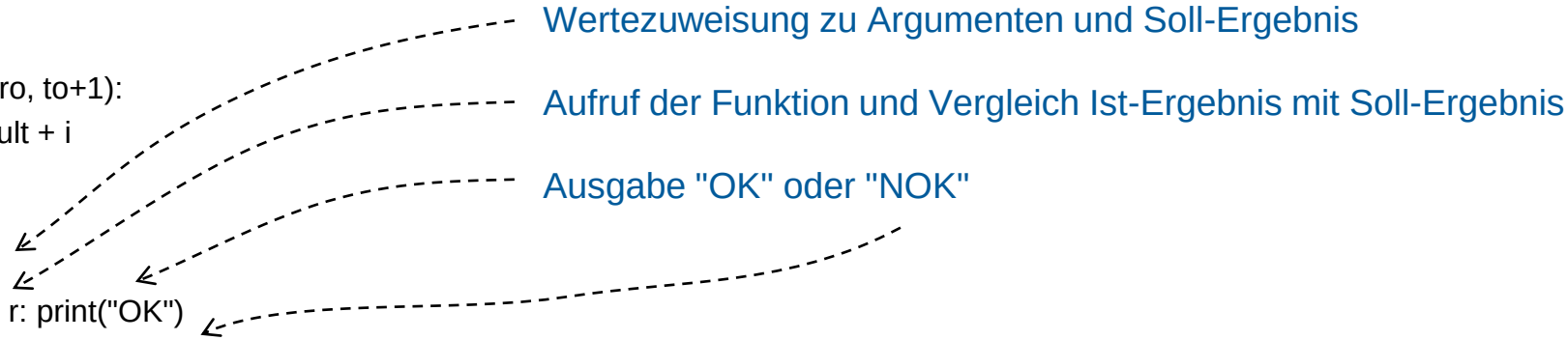
- Im folgenden Beispiel unterscheiden sich die drei Testfälle nur in den Variablenzuweisungen. Bei einem Fehler werden das Ist-Ergebnis und das Soll-Ergebnis ausgegeben. Die Texte können natürlich frei formuliert werden.

```
def sum(to, fro):
    result = 0
    for i in range(fro, to+1):
        result = result + i
    return result

to, fro, r = 1, 1, 1
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))

to, fro, r = 10, 1, 55
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))

to, fro, r = 10, 3, 52
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))
```



Module und testgetriebene Entwicklung (6)

- Wählen Sie als Testfälle auch Fehlerfälle. Beispiel:

```
to, fro, r = 3, 10, 0
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))
```

  Im Beispiel ist die obere Grenze kleiner als die untere Grenze.

- Man kann das Soll-Ergebnis in machen Fällen auch von Python berechnen lassen. Beispiel:

```
to, fro, r = 10, 3, 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))
```

Module und testgetriebene Entwicklung (7)

- In der **Testgetriebenen Entwicklung** geht man sogar soweit, dass man die Testfälle vor der Implementierung schreibt. Beispiel:

```
def sum(to, fro):
    # not yet implemented
    return 0

to, fro, r = 1, 1, 1
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))

to, fro, r = 10, 1, 55
if sum(to, fro) == r: print("OK")
else: print("NOK: sum(" + str(to) + ", " + str(fro) + ") is " + str(sum(to, fro)) + " but should be " + str(r))
```

Normalerweise schlagen alle (oder fast alle) Testfälle fehl.

Default-Parameter, Schlüsselwortparameter und Prozeduren (1)

- In einem Beispiel im letzten Kapitel haben wir die Funktion sum(to, fro) definiert. Damit kann man die Summe der Zahlen von fro bis to berechnen, z.B. 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10. Oft möchte man die Summe von 1 bis to ausrechnen. Zur Abkürzung kann man die Zahl 1 als **Default-Parameter** angeben und kann sie dann beim Aufruf weglassen.

```
def sum(to, fro=1):
    result = 0
    for i in range(fro, to+1):
        result = result + i
    return result
```

  Der Aufruf sum(10) ist gleichbedeutend mit sum(10, 1).

- Es können auch mehrere oder alle Parameter einer Funktion Default-Parameter haben.
- Vorteile: Weniger Schreibarbeit
  Einheitliche Default-Parameter

Default-Parameter, Schlüsselwortparameter und Prozeduren (2)

- Sie haben sich bestimmt gewundert, warum die Funktion sum als ersten Parameter to und als zweiten Parameter fro hat. Naheliegend wäre

  ```
  def sum(fro=1, to):
  ```

  Der Aufruf sum(10) mit dem Ziel die Summe der Zahlen von 1 bis 10 zu berechnen, würde dann aber nicht funktionieren, weil 10 als Argument für fro genommen wird und to kein Argument hat.

- Ist

  ```
  def sum(fro=1, to=1):
  ```

  und ein Aufruf sum(10) die Lösung? Leider nicht, da der Wert 10 wieder für den Parameter fro übergeben wird.

- In diesem Fall sind **Schlüsselwortparameter** beim Aufruf nützlich. Beispiel:

  ```
  sum(to=10)
  ```

Default-Parameter, Schlüsselwortparameter und Prozeduren (3)

- Schlüsselwortparameter legen fest, zu welchem Parameter der folgende Wert gehört. Beispiel:

```
def sum(fro=1, to=1):
    result = 0
    for i in range(fro, to+1):
        result = result + i
    return result


print(sum(to=10))              # Ausgabe: 55
print(sum(fro=3, to=10))       # Ausgabe: 52
print(sum(to=10, fro=3))       # Ausgabe: 52
print(sum(3, to=10))           # Ausgabe: 52
print(sum(3, fro=10))          # Geht leider nicht: „sum() got multiple values for argument 'fro‘”
print(sum(fro=3, 10))          # Geht leider nicht: „positional argument follows keyword argument”
print(sum())                   # Ausgabe: 1
```

- Beachten Sie die beiden Fälle, die Fehler ergeben.

Default-Parameter, Schlüsselwortparameter und Prozeduren (4)

- Wir haben schon gesehen, dass Funktionen immer einen Wert zurückgeben. Wenn nichts angegeben ist, ist der Rückgabewert None.
- Funktionen, die in keinem Zweig einen Rückgabewert definieren, werden manchmal auch **Prozeduren** genannt. Beispiel:

```
def say_hello2(name):
    print("Hello " + name)

n = "Michael"
say_hello2(n)
name = say_hello2(n)          # Geht zwar, macht aber wenig Sinn
print(name)                   # Ausgabe: None
```

- Eine return-Anweisung ist nicht erforderlich, kann aber in der Funktion enthalten sein, z.B. um die Funktion in einem Zweig abzubrechen.
- 🕮 Kapitel 14.4 – 14.6 in (Klein 2018)