



# Module

# Module



Längere Programme unübersichtlich

- Verteilung auf mehrere Dateien

Manche Teile allgemein, auch verwendbar in anderen Programmen

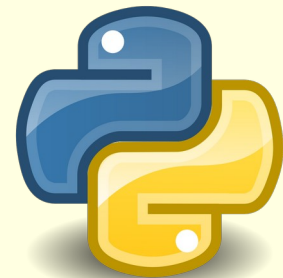
- Code in einer Datei in mehreren Programmen aufrufen

# Module



- externe Dateien
- deren Code wir verwenden
- heißen **Module**
- und werden **eingebunden**

# Module



unterscheiden drei Typen:

- Module der Standardbibliothek
- globale Module, auch Bibliotheken
- lokale Module

# Module



## Module der Standardbibliothek

- in jeder Installation dabei
- nicht verändert
- teilweise im Interpreter

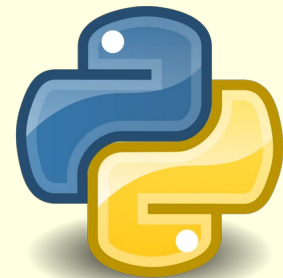
# Module



globale Module

- Unterverzeichnis *site-packages*
- vom Benutzer abgelegt / geschrieben
- für alle Programme zugänglich
- Veränderungen normal

# Module



## lokale Module

- meist im Verzeichnis des Programms
- vom Benutzer abgelegt / geschrieben
- nur für Programme im selben Verzeichnis zugänglich
- Veränderungen häufig

# Module

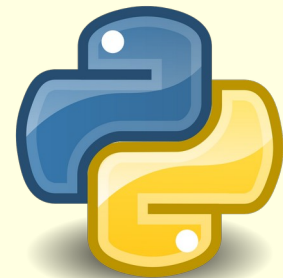


Einbindung mit `import`-Anweisung

```
1 import time
```



# Module



## Einbindung mit `import`-Anweisung

```
1 import time
2
3 start = time.process_time()
4
5 for i in range(10000):
6     print( "\r", i, end="" )
7
8 end = time.process_time()
9
10 print(end - start)
```

Ausgabe:

9999

0.17659425000000084

# Module



## Einbindung mit `import`-Anweisung

```
1 import time
2
3 start = time.process_time()
4
5 for i in range(100_000):
6     print( "\r", i, end="" )
7
8 end = time.process_time()
9
10 print(end - start)
```

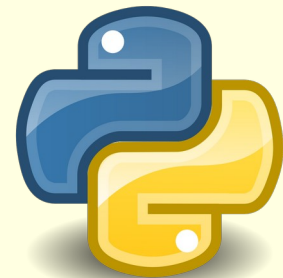
Ausgabe:

99999

1.7713348769999993

vorher:0. 17659425000000084

# Module

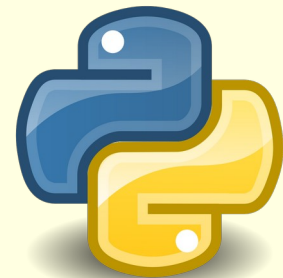


## Einbindung mehrerer Module

```
1 import time, math
2
3 import time
4 import math
```

- separate Anweisungen oder
- mit Komma getrennt

# Module

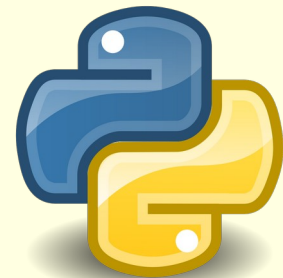


Nach Einbindung neuer Namensraum

- verwendet wie Instanz

```
1 import time
2
3 start = time.process_time()
4
5 for i in range(10000):
6     print( "\r", i, end="" )
7
8 end = time.process_time()
9
10 print(end - start)
```

# Module



Nach Einbindung neuer Namensraum

- kann umbenannt werden mit `as`

```
1 import time as zeitnahme
2
3 start = zeitnahme.process_time()
4
5 for i in range(10000):
6     print( "\r", i, end="" )
7
8 end = zeitnahme.process_time()
9
10 print(end - start)
```

# Module

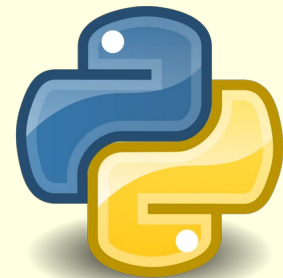


Einbindung ohne neuen Namensraum

- auch für einzelne Funktionen o.ä.

```
1 from time import process_time
2
3 start = process_time()
4
5 for i in range(10000):
6     print( "\r", i, end="" )
7
8 end = process_time()
9
10 print(end - start)
```

# Module

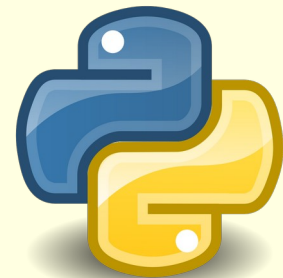


Einbindung ohne neuen Namensraum

- auch für einzelne Funktionen o.ä.

```
1 from time import process_time as zeitnahme, time_ns as zt
2
3 start = zeitnahme()
4
5 for i in range(10_000):
6     print( "\r", str(i), end="" )
7
8 end = zeitnahme()
9
10 print(end - start)
```

# Module



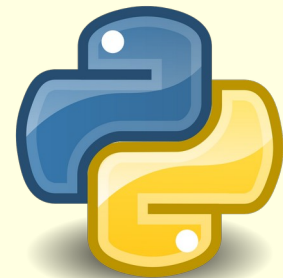
Einbindung ohne neuen Namensraum

- für alles: `from ... import *`

```
1 from time import *
2
3 start = process_time()
4
5 for i in range(10_000):
6     print( "\r", str(i), end="" )
7
8 end = process_time()
9
10 print(end - start)
```



# Module



Einbindung eines eigenen, lokalen Moduls

- ***eigenes*** oder ***blatt6.py*** liegt unter Abgaben/06/

```
1 def division(dividend, divisor):
2     erg, rest = divmod(dividend, divisor)
3     print(str(dividend) + " geteilt durch "
4           + str(divisor) + " ist ", end="")
5     print(str(erg) + " Rest " + str(rest) + ".")
6
7 def numerier(li):
8     for pos, wert in enumerate(li):
9         print("Element " + str(pos) + " der Liste ist: " + str(wert) )
10
11
12 def gerade(li):
13     return list(filter(lambda x: not x%2, li))
14
15 def vokale(string):
16     return list(filter(lambda x: x in "aeiou", string))
```

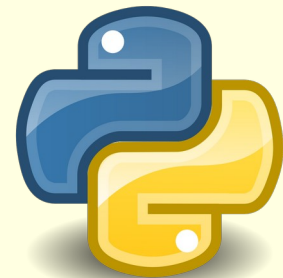
# Module



Einbindung eines eigenen, lokalen Moduls

```
1 import blatt6 as sammlung
2
3 print( sammlung.gerade( [1,2,3,4,5] ) )
```

# Module

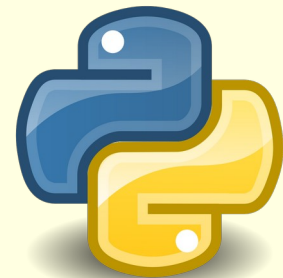


Einbindung eines eigenen, lokalen Moduls

- blatt6.py liegt unter Material/src/modul/

```
1 from blatt6 import gerade
2
3 print( gerade( [1,2,3,4,5] ) )
```

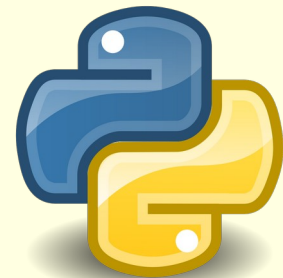
# Module



Suche nach Modul

- 1) eingebautes Modul
- 2) lokales Modul
- 3) globales Modul
- 4) `ModuleNotFoundError`

# Module



## Modulinterne Referenzen

- s. Tabelle 18.1

```
1 from blatt6 import gerade
2
3 gerade( [1,2,3,4,5] )
4
5 print(__name__)
6 print(__file__)

__main__
/Pfad/F02/import_blat6.py
```

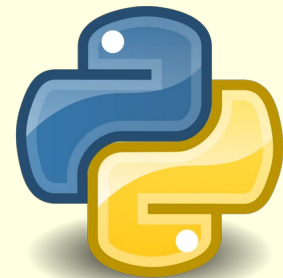
# Module



Code eines Moduls wird bei Einbindung ausgeführt

- z.B. Instanziierung von 'Konstanten'
- möglich: alles

# Module

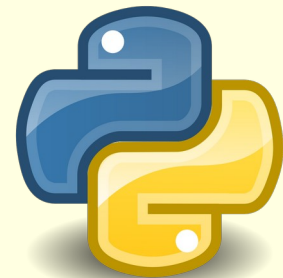


Ein *Paket (Package)* fasst mehrere Module zusammen

- eigener Unterordner, Name wie Paket
- optional Datei `__init__.py`, ausgeführt bei Einbindung
- Einbindung wie Modul

```
1 import paket
```

# Module



Ein *Paket* (*Package*) fasst mehrere Module zusammen

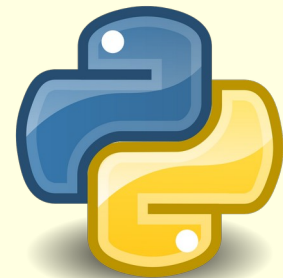
- eigener Unterordner, Name wie Paket
- optional Datei `__init__.py`, ausgeführt bei Einbindung
- Einbindung wie Modul

```
1 import paket
2
3 import paket.modul
```

- auch Einbindung einzelnen Moduls



# Module



Einbinden aller Module via

```
1 from paket import *  
2
```

geht nicht.

Wenn gewünscht, z.B. durch import-Anweisungen in  
\_\_init\_\_.py

# Module



Die Funktion `dir(obj)` liefert alle Methoden eines Objektes

```
1 import blatt7
2
3 print(dir(blatt7))

['Konst', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__',
 '__spec__', 'division', 'gerade', 'numerier', 'vokale']
```

ohne Argument: aktueller Namensraum

# Module / Pakete



- Modul: Datei mit Python-Code
- Paket: Verzeichnis, das Module enthält
- `__init__.py` wird bei Einbindung ausgeführt
- erst `__init__.py` macht Verzeichnis zu normalem Paket

# Module / Pakete



Was tut `__init__.py`?

- Beispiel

```
sound/
    __init__.py
    formats/
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        ...
    effects/
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
```

Top-level package  
Initialize sound package  
Subpackage for file  
format conversions

Subpackage for sound effects

Subpackage for filters

# Module / Pakete



\_\_init\_\_.py Verweise auf Teile des Pakets

- . und .. wie üblich in Dateisystem

```
from . import echo  
from .. import formats  
from ..filters import equalizer
```

# Module / Pakete



```
from . import echo
```

```
from .. import formats
```

```
from ..filters import equalizer
```

```
sound/
```

```
    __init__.py
```

```
    formats/
```

```
        __init__.py
```

```
        wavread.py
```

```
        wavwrite.py
```

```
        aiffread.py
```

```
        aiffwrite.py
```

```
        ...
```

```
effects/
```

```
    __init__.py
```

```
    echo.py
```

```
    surround.py
```

```
    reverse.py
```

```
    ...
```

```
filters/
```

```
    __init__.py
```

```
    equalizer.py
```

```
    vocoder.py
```

```
    karaoke.py
```

Top-level package

Initialize sound package

Subpackage for file

format conversions

Subpackage for sound effects

Subpackage for filters

# Module / Pakete



Was tut `__init__.py`?

- wichtig bei import \*
- `__all__` spezifiziert \*

```
__all__ = [  
    "echo",  
    "surround",  
    "reverse"  
]
```



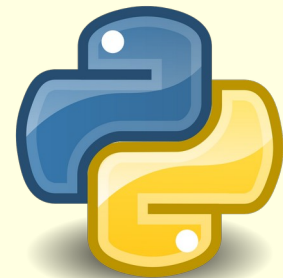
Given the following package layout, select all options containing valid relative imports called from `__init__.py`

```
package/  
  subpackage1/  
    __init__.py  
    moduleX.py  
    moduleY.py  
  subpackage2/  
    moduleZ.py  
  moduleA.py
```

- a) `from .moduleY import spam`
- b) `from .moduleY import spam as ham`
- c) `from ..subpackage1 import moduleY`
- d) `from ..subpackage2.moduleZ import eggs`
- e) `from ..moduleA import foo`



# Module



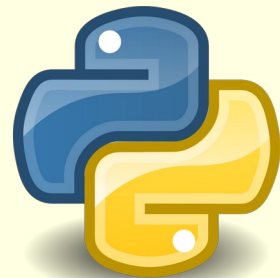
The module has been imported using the following line

```
from mod import func
```

How can you invoke the function?

- a) `mod.func()`
- b) `func()`
- c) `mod::func()`
- d) `mod:func()`

# Module



Assuming that all three files x.py, y.py, and z.py reside in the same folder, what will be the output produced by running the z.py file?

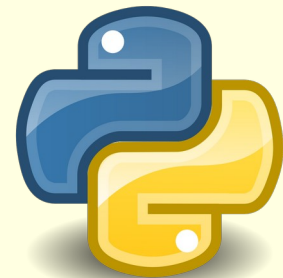
```
x.py:
    print('x', end="")
```

```
y.py:
    import x
    print('y', end="")
```

```
z.py:
    print('z', end="")
    import x
    import y
```

- a) zxxxy
- b) The code is erroneous.
- c) zxy
- d) zyx

# Module



Which of the following is true?

- a) packages can contain modules
- b) modules can contain packages
- c) modules can contain modules
- d) None of the above

# Module



Assuming that all three files reside in the same folder, what will be the output produced by running the spam.py file?

```
# spam.py
    print("spam", end=' ')
    import ham

# ham.py
    import eggs
    print("ham", end=' ')

# eggs.py
    print("eggs", end=' ')
```

- a) Syntax Error
- b) spam eggs ham
- c) spam ham
- d) eggs ham spam

# Module



Select all valid parameters to function dir()

- a) No parameter
- b) Object
- c) 0
- d) None