Python-Tutorial-Lernnotizen

Durch das bisherige Lernen haben wir bereits einiges über Python erfahren. Jetzt werden wir basierend auf der offiziellen Dokumentation unser Wissen über Python weiter ergänzen.

Steuerung des Codeflusses

Typ

```
print(type(1))
```

```
<class 'int'>
```

```
print(type('a'))
```

```
<class 'str'>
```

Die `type`-Funktion ist nützlich, um den Typ eines Objekts auszugeben.

range

Die `range`-Funktion in Python ist eine eingebaute Funktion, die eine Folge von Zahlen erzeugt. Sie wird häufig in Schleifen verwendet, um eine bestimmte Anzahl von Iterationen durchzuführen. Die `range`-Funktion kann bis zu drei Argumente annehmen:

1. **Start**: Die Startzahl der Sequenz (optional, Standardwert ist 0).
2. **Stop**: Die Endzahl der Sequenz (erforderlich, die Sequenz endet vor dieser Zahl).
3. **Step**: Der Abstand zwischen den Zahlen (optional, Standardwert ist 1).

Hier sind einige Beispiele:

```
# Erzeugt eine Sequenz von 0 bis 4
for i in range(5):
    print(i)

# Erzeugt eine Sequenz von 2 bis 8 mit einem Schritt von 2
for i in range(2, 9, 2):
    print(i)
```

```
# Erzeugt eine Sequenz von 10 bis 1 mit einem Schritt von -1
for i in range(10, 0, -1):
    print(i)
```

Die `range`-Funktion ist sehr effizient, da sie nicht die gesamte Sequenz im Speicher speichert, sondern die Zahlen bei Bedarf generiert. Dies macht sie ideal für die Verwendung in Schleifen, insbesondere bei großen Zahlenbereichen.

Die `range`-Funktion ist äußerst nützlich.

```
for i in range(5):
  print(i, end = ' ')
```

```
0 1 2 3 4
```

```
for i in range(2, 6, 2):
  print(i, end = ' ')
```

(Anmerkung: Der Code bleibt auf Englisch, da es sich um eine Programmiersprache handelt und die Syntax nicht übersetzt wird.)

```
2 4
```

Schauen Sie sich die Definition der `range`-Funktion an.

```
class range(Sequence[int]):
    start: int
    stop: int
    step: int
```

(Der Code bleibt unverändert, da es sich um eine Python-Klassendefinition handelt, die nicht übersetzt werden sollte.)

`Visible` ist eine Klasse.

```
print(range(5))
```

```
range(0, 5)
```

Anstatt:

```
[0,1,2,3,4]
```

Fortsetzung folgt.

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

Warum. Schauen Sie sich die Definition von `list` an.

```
class list(MutableSequence[_T], Generic[_T]):
```

(Die Codezeile bleibt unverändert, da es sich um eine Python-Klassendefinition handelt, die nicht übersetzt werden sollte.)

Die Definition von `list` ist `list(MutableSequence[_T], Generic[_T]):`. Die Definition von `range` ist `class range(Sequence[int])`. `list` erbt von `MutableSequence`, während `range` von `Sequence` erbt.

Weiter unten findet man Folgendes.

```
Sequence = _alias(collections.abc.Sequence, 1)
MutableSequence = _alias(collections.abc.MutableSequence, 1)
```

Hier verstehen wir die Beziehung zwischen den beiden nicht. Aber wir haben ungefähr verstanden, warum wir `list(range(5))` so schreiben können.

Funktionsparameter

Schauen wir uns zusätzliche Informationen zu Funktionen an.

```
def fn(a = 3):
  print(a)
```

fn()

```shell
3
```

Dies weist einem Parameter einen Standardwert zu.

```
def fn(end: int, start = 1):
  i = start
  s = 0
  while i < end:
    s += i
    i += 1
  return s
```

(Der Code bleibt unverändert, da es sich um eine Programmiersprache handelt und keine Übersetzung benötigt wird.)

```
print(fn(10))
```

```
45
```

`end` ist ein zwingend erforderlicher Parameter. Beachten Sie, dass die zwingend erforderlichen Parameter an erster Stelle stehen sollten.

```
def fn(start = 1, end: int):

    def fn(start = 1, end: int):
                                ^
SyntaxError: Nicht-Standard-Argument folgt auf Standard-Argument
```

Beachten Sie, dass `end` ein `non-default argument` ist. `start` ist ein `default argument`. Das bedeutet, dass ein nicht-standardmäßiges Argument auf ein standardmäßiges Argument folgt. Das heißt, Sie müssen das nicht-standardmäßige Argument vor allen standardmäßigen Argumenten platzieren. `start` ist ein Standardargument, was bedeutet, dass es bereits einen Wert hat, wenn es nicht übergeben wird.

```
def fn(a, /, b):
  print(a + b)
```

In diesem Python-Code wird eine Funktion `fn` definiert, die zwei Parameter `a` und `b` akzeptiert. Der Schrägstrich `/` vor dem Parameter `b` zeigt an, dass `a` ein positionsgebundener Parameter ist, was bedeutet, dass er nur durch seine Position im Funktionsaufruf übergeben werden kann und nicht als Schlüsselwortargument. Der Parameter `b` kann dagegen sowohl positionsgebunden

als auch als Schlüsselwortargument übergeben werden. Die Funktion gibt die Summe von `a` und `b` aus.

fn(1, 3)

```
Hier wird `/` verwendet, um die Parametertypen zu trennen. Es gibt zwei Arten, Parameter zu übergeben: 
```

```python
def fn(a, /, b):
  print(a + b)
```

In diesem Python-Code wird eine Funktion `fn` definiert, die zwei Parameter `a` und `b` akzeptiert. Der Schrägstrich `/` in der Parameterliste zeigt an, dass `a` ein positionsgebundener Parameter ist, was bedeutet, dass er nur durch seine Position übergeben werden kann und nicht als Schlüsselwortargument. Der Parameter `b` kann sowohl positionsgebunden als auch als Schlüsselwortargument übergeben werden. Die Funktion gibt die Summe von `a` und `b` aus.

fn(a=1, 3)

```shell
    fn(a=1, 3)
               ^
SyntaxError: Positionsargument folgt auf Schlüsselwortargument
```

So schreiben geht nicht. `a=1` bedeutet, dass dies ein Schlüsselwortparameter ist. Es wird als ein Schlüsselwortparameter behandelt. Während `b` ein Positionsparameter ist.

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional oder Keyword |
        |                                  - Nur Keyword
         -- Nur Positional
```

Beachten Sie, dass bei der Definition der Funktion die Verwendung von `/` und `*` bereits den Übergabetyp der Parameter impliziert. Daher müssen die Parameter entsprechend den Regeln übergeben werden.

```python
def fn(a, /, b):
  print(a + b)
```

In diesem Python-Code wird eine Funktion `fn` definiert, die zwei Parameter `a` und `b` akzeptiert. Der Schrägstrich `/` in der Parameterliste zeigt an, dass der Parameter `a` nur als Positionsparameter übergeben werden kann, d.h. er kann nicht als Schlüsselwortargument übergeben werden. Die Funktion gibt die Summe von `a` und `b` aus.

fn(1, b=3)

```
Der obige Code hat keinen Fehler verursacht.


```python
def fn(a, /, b, *, c):
  print(a + b + c)
```

In diesem Python-Code wird eine Funktion `fn` definiert, die drei Parameter `a`, `b` und `c` akzeptiert. Der Schrägstrich `/` vor `b` zeigt an, dass `a` ein positionsgebundener Parameter ist, während `b` ein normaler Parameter ist. Das Sternchen `*` vor `c` zeigt an, dass `c` ein schlüsselwortgebundener Parameter ist. Die Funktion gibt die Summe der drei Parameter aus.

fn(1, 3, 4)

```shell
    fn(1, 3, 4)
TypeError: fn() nimmt 2 Positionsargumente an, aber 3 wurden übergeben
```

`fn` kann nur 2 Positionsargumente empfangen, aber es wurden 3 übergeben.

```
def fn(a, /, b, *, c):
  print(a + b + c)
```

Hinweis: Der Code bleibt unverändert, da es sich um eine Python-Funktionsdefinition handelt, die spezielle Parameter-Syntax verwendet. Die Parameter `a`, `b` und `c` sind Teil der Funktionssignatur und sollten nicht übersetzt werden.

fn(a = 1, b=3, c=4)

```shell
    fn(a = 1, b=3, c=4)
TypeError: fn() erhielt einige positionsgebundene Argumente als Schlüsselwortargumente: 'a'
```

Einige Parameter, die früher nur über die Position übergeben werden konnten, werden jetzt mit Schlüsselwörtern übergeben.
```

Parameter in Form von Zuordnungen

```
def fn(**kwds):
  print(kwds)
```

fn(**{‘a’: 1})

```shell
{'a': 1}
```

```
def fn(**kwds):
  print(kwds['a'])
```

(Der Code bleibt auf Englisch, da es sich um eine Programmiersprache handelt und die Übersetzung den Code unbrauchbar machen würde.)

```
d = {'a': 1}
fn(**d)
```

```
1
```

Sichtbar ist, dass ** die Parameter entfaltet.

```
def fn(a, **kwds):
  print(kwds['a'])

d = {'a': 1}
fn(1, **d)
```

In diesem Codeausschnitt wird ein Wörterbuch `d` mit dem Schlüssel `'a'` und dem Wert `1` erstellt. Die Funktion `fn` wird mit dem Argument `1` und den Schlüsselwortargumenten aus dem Wörterbuch `d` aufgerufen. Der `**`-Operator wird verwendet, um das Wörterbuch in Schlüsselwortargumente zu entpacken. Das bedeutet, dass der Aufruf `fn(1, **d)` äquivalent zu `fn(1, a=1)` ist.

```
TypeError: fn() hat mehrere Werte für das Argument 'a' erhalten
```

Wenn eine Funktion wie `fn(1, **d)` aufgerufen wird, wird dies zu `fn(a=1, a=1)` erweitert. Daher tritt ein Fehler auf.

```
def fn(**kwds):
  print(kwds['a'])
```

Hinweis: Der Code wurde nicht übersetzt, da es sich um eine Programmiersprache handelt, die in der Regel nicht übersetzt wird. Der Code bleibt in der Originalsprache, um seine Funktionalität und Lesbarkeit zu erhalten.

```
d = {'a': 1}
fn(d)
```

```
TypeError: fn() nimmt 0 Positionsargumente an, aber 1 wurde gegeben
```

Wenn eine Funktion wie `fn(d)` aufgerufen wird, wird dies als Positionsargument behandelt und nicht als Schlüsselwortargument entpackt.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

In diesem Python-Code wird eine Funktion `fn` definiert, die ein Argument `a` akzeptiert, das nur als Positionsargument übergeben werden kann (gekennzeichnet durch den Schrägstrich `/`). Die Funktion nimmt auch beliebige Schlüsselwortargumente (`**kwds`) entgegen. Innerhalb der Funktion wird der Wert des Schlüsselwortarguments `'a'` aus dem `kwds`-Dictionary ausgegeben.

Beachte, dass der Parameter `a` in der Funktionsdefinition nicht direkt mit dem Schlüsselwortargument `'a'` in `kwds` verknüpft ist. Der Code würde einen Fehler auslösen, wenn `'a'` nicht in `kwds` enthalten ist.

```
d = {'a': 1}
fn(1, **d)
```

So geht es. Es zeigt, dass Positionsparameter und Parameter in Form von Zuordnungen denselben Namen haben können.

```
def fn(a, / , a):
  print(a)
```

Hinweis: Der obige Code enthält einen Syntaxfehler, da der Parameter `a` zweimal in der Parameterliste definiert wird. In Python ist es nicht erlaubt, denselben Parameternamen mehrfach in der Parameterliste zu verwenden. Der Code würde einen `SyntaxError` auslösen.

```
d = {'a': 1}
fn(1, **d)
```

In diesem Codeausschnitt wird ein Wörterbuch `d` mit einem Schlüssel `'a'` und dem Wert `1` erstellt. Anschließend wird die Funktion `fn` mit zwei Argumenten aufgerufen: dem Wert `1` und den Schlüssel-Wert-Paaren aus dem Wörterbuch `d`, die mit dem `**`-Operator entpackt werden. Dies bedeutet, dass die Funktion `fn` effektiv mit den Argumenten `1` und `a=1` aufgerufen wird.

```
SyntaxError: doppeltes Argument 'a' in der Funktionsdefinition
```

Auf diese Weise tritt ein Fehler auf. Beachten Sie die subtilen Beziehungen zwischen diesen verschiedenen Situationen.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

In diesem Python-Code wird eine Funktion `fn` definiert, die ein Argument `a` akzeptiert, das nur als Positionsargument übergeben werden kann (gekennzeichnet durch den Schrägstrich `/`). Die Funktion nimmt auch beliebige Schlüsselwortargumente (`**kwds`) entgegen. Innerhalb der Funktion wird der Wert des Schlüsselwortarguments `'a'` aus dem `kwds`-Dictionary ausgegeben.

Beachte, dass der Parameter `a` in der Funktionsdefinition nicht direkt mit dem Schlüsselwortargument `'a'` in `kwds` verknüpft ist. Das bedeutet, dass der Wert von `a` als Positionsargument übergeben wird, während das Schlüsselwortargument `'a'` separat im `kwds`-Dictionary gespeichert wird.

fn(1, **[1,2])

```shell
TypeError: __main__.fn() Argument nach ** muss eine Mapping sein, keine Liste
```

`**` muss von einer Zuordnung gefolgt sein.

Parameter von iterierbaren Typen

```
def fn(*kwds):
  print(kwds)

fn(*[1,2])
```

```shell
(1, 2)
```

```
def fn(*kwds):
  print(kwds)
```

fn(*1)

```shell
TypeError: __main__.fn() Argument nach * muss ein iterierbares Objekt sein, nicht int
```

`*` muss auf ein `iterable` folgen.

```
def fn(a, *kwds):
  print(type(kwds))
```

fn(1, *[1])

```shell
<class 'tuple'>
```

Gib den Typ aus. Das ist auch der Grund, warum oben `(1,2)` und nicht `[1,2]` ausgegeben wird.

```
def fn(*kwds):
  print(kwds)
```

fn(1, *[1])

```shell
(1, 1)
```

Beachten Sie, dass beim Aufruf von `fn(1, *[1])` die Argumente entpackt werden, sodass daraus `fn(1, 1)` wird. Wenn dann `fn(*kwds)` analysiert wird, verwandelt `kwds` die `1, 1` in das Tupel `(1, 1)`.

```
def concat(*args, sep='/'):
  return sep.join(args)
```

Übersetzung ins Deutsche:

```
def concat(*args, sep='/'):
  return sep.join(args)
```

Hinweis: Der Code bleibt auf Englisch, da es sich um eine Programmiersprache handelt und die Funktionen und Parameter in der Regel nicht übersetzt werden.

```
print(concat('a','b','c', sep=','))
```

```
a,b,c
```

Lambda-Ausdrücke

`lambda` ermöglicht es, Funktionen wie Variablen zu speichern. Erinnerst du dich an das, was im Artikel „Demystifying Computer Science“ gesagt wurde?

```
def incrementor(n):
  return lambda x: x + n
```

(Der Code bleibt auf Englisch, da es sich um eine Programmiersprache handelt und die Syntax nicht übersetzt wird.)

```
f = incrementor(2)
print(f(3))
```

```
5
```

Schauen wir uns ein weiteres Beispiel an.

```
pairs = [(1, 4), (2,1), (0, 3)]
```

pairs.sort(key = lambda pair: pair[1])

print(pairs)

```shell
[(2, 1), (0, 3), (1, 4)]
```

```
pairs = [(1, 4), (2,1), (0, 3)]
```

pairs.sort(key = lambda pair: pair[0])

```
print(pairs)
```

```
[(0, 3), (1, 4), (2, 1)]
```

`pair[0]` □□□□□□□□□□□□`pair[1]` □□□□□□□□□□□□

Dokumentationskommentare

```
def add():
  """Füge etwas hinzu
  """
  pass
```

```
print(add.__doc__)
```

```
etwas hinzufügen
```

Funktionssignatur

```
def add(a:int, b:int) -> int:
  print(add.__annotations__)
  return a+b
```

In diesem Python-Code wird eine Funktion `add` definiert, die zwei Integer-Werte `a` und `b` als Parameter entgegennimmt und deren Summe zurückgibt. Die Funktion gibt auch die Annotationen der Funktion aus, die in `add.__annotations__` gespeichert sind. In diesem Fall zeigt `add.__annotations__` die Typen der Parameter und des Rückgabewerts an.

```
add(1, 2)
```

```
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Datenstrukturen

Listen

```
a = [1,2,3,4]
```

a.append(5) print(a) # [1, 2, 3, 4, 5]

a[len(a):] = [6] print(a) # [1, 2, 3, 4, 5, 6]

a[3:] = [6] print(a) # [1, 2, 3, 6]

```
a.insert(0, -1)
print(a)   # [-1, 1, 2, 3, 6]
```

a.remove(1) print(a) # [-1, 2, 3, 6]

a.pop() print(a) # [-1, 2, 3]

```
a.clear()
print(a)  # []
```

```
a[:] = [1, 2]
print(a.count(1)) # 1
```

a.reverse() print(a) # [2, 1]

```
b = a.copy()
a[0] = 10
print(b)   # [2, 1]
print(a)   # [10, 1]
```

```
b = a
a[0] = 3
print(b)  # [3, 1]
print(a)  # [3, 1]
```

Listenkonstruktion

```
print(3 ** 2)   # 9
print(3 ** 3)   # 27
```

Zuerst lernen wir eine Operation kennen, `**`. Dies bedeutet `Potenz`.

```
sq = []
for x in range(10):
  sq.append(x ** 2)

print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Versuchen wir es nun mit `map`.

```
a = map(lambda x:x, range(10))
print(a)
# <map object at 0x103bb0550>
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sq = map(lambda x: x ** 2, range(10))
print(list(sq))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

sq = [x ** 2 for x in range(10)]
print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sichtbar ist, dass `for` sehr flexibel ist.

```
a = [i for i in range(5)]
print(a)
# [0, 1, 2, 3, 4]

a = [i+j for i in range(3) for j in range(3)]
print(a)
# [0, 1, 2, 1, 2, 3, 2, 3, 4]
```

```
a = [i for i in range(5) if i % 2 == 0]
print(a)
# [0, 2, 4]

a = [(i,i) for i in range(3)]
print(a)
# [(0, 0), (1, 1), (2, 2)]
```

Verschachtelte Listenkonstruktion

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]

t = []
for j in range(3):
  t.append([(i+j*4) for i in range(4)])
print(t)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

Beachte die Art und Weise dieser beiden Code-Abschnitte. Das heißt:

```
[[(i+j*4) for i in range(4)] for j in range(3)]
```

(Der Code bleibt unverändert, da es sich um eine Python-Liste handelt, die nicht übersetzt werden muss.)

Das entspricht:

```
for j in range(3):
   [(i+j*4) for i in range(4)]
```

(Der Code bleibt unverändert, da es sich um eine Programmiersprache handelt und keine Übersetzung erforderlich ist.)

Das entspricht also:

```
for j in range(3):
  for i in range(4):
      (i+j*4)
```

(Der Code bleibt unverändert, da es sich um eine Programmiersprache handelt und die Syntax nicht übersetzt wird.)

Daher eignet es sich gut für die Matrix-Transposition.

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
#  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

mt = [[row[j] for row in matrix] for j in range(4)] print(mt) # [[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]

```
print(list(zip(*matrix)))
[(0, 4, 8), (1, 5, 9), (2, 6, 10), (3, 7, 11)]
```

del

```
a = [1, 2, 3, 4]

del a[1]
print(a)  # [1, 3, 4]

del a[0:2]
print(a) # [4]

del a
print(a) # NameError: name 'a' is not defined
```

Wörterbuch

```
ages = {'li': 19, 'wang': 28, 'he' : 7}
for name, age in ages.items():
    print(name, age)
```

li 19

wang 28

he 7

```
for name in ages:
    print(name)

# li
# wang
# he

for name, age in ages:
    print(name)
```

ValueError: zu viele Werte zum Entpacken (erwartet wurden 2)

```
for i, name in enumerate(['li', 'wang', 'he']):
    print(i, name)
```

0 li

1 wang

2 he

print(reversed([1, 2, 3])) # <list_reverseiterator object at 0x10701ffd0>

print(list(reversed([1, 2, 3]))) # [3, 2, 1]

(Beachten Sie, dass der Text, der übersetzt werden soll, nicht im ursprünglichen Beitrag enthalten war.

Module

Skriptgesteuerter Aufruf von Modulen

```python
import sys

def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

if __name__ == "__main__":
    r = f(int(sys.argv[1]))
    print(r)

% python fib.py 3
2

% python -m fib 5
5
```

Verzeichnis (dir)

```
import fib

print(dir(fib))

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec_

import builtins
print(dir(builtins))
```

(Dieser Code bleibt unverändert, da es sich um eine Python-Syntax handelt, die nicht übersetzt wird.)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',

‘EOFError’, ‘Ellipsis’, ‘EnvironmentError’, ‘Exception’, ‘False’, ‘FileExistsError’, ‘FileNotFoundError’, ‘FloatingPointError’, ‘FutureWarning’, ‘GeneratorExit’, ‘IOError’, ‘ImportError’, ‘ImportWarning’, ‘IndentationError’, ‘IndexError’, ‘InterruptedError’, ‘IsADirectoryError’, ‘KeyError’, ‘KeyboardInterrupt’, ‘LookupError’, ‘MemoryError’, ‘ModuleNotFoundError’, ‘NameError’, ‘None’, ‘NotADirectoryError’, ‘NotImplemented’, ‘NotImplementedError’, ‘OSError’, ‘OverflowError’, ‘PendingDeprecationWarning’, ‘PermissionError’, ‘ProcessLookupError’, ‘RecursionError’, ‘ReferenceError’, ‘ResourceWarning’, ‘RuntimeError’, ‘RuntimeWarning’, ‘StopAsyncIteration’, ‘StopIteration’, ‘SyntaxError’, ‘SyntaxWarning’, ‘SystemError’, ‘SystemExit’, ‘TabError’, ‘TimeoutError’, ‘True’, ‘TypeError’, ‘UnboundLocalError’, ‘UnicodeDecodeError’, ‘UnicodeEncodeError’, ‘UnicodeError’, ‘UnicodeTranslateError’, ‘UnicodeWarning’, ‘UserWarning’, ‘ValueError’, ‘Warning’, ‘ZeroDivisionError’, ‘**build_class**’, ‘**debug**’, ‘**doc**’, ‘**import**’, ‘**loader**’, ‘**name**’, ‘**package**’, ‘**spec**’, ‘abs’, ‘all’, ‘any’, ‘ascii’, ‘bin’, ‘bool’, ‘breakpoint’, ‘bytearray’, ‘bytes’, ‘callable’, ‘chr’, ‘classmethod’, ‘compile’, ‘complex’, ‘copyright’, ‘credits’, ‘delattr’, ‘dict’, ‘dir’, ‘divmod’, ‘enumerate’, ‘eval’, ‘exec’, ‘exit’, ‘filter’, ‘float’, ‘format’, ‘frozenset’, ‘getattr’, ‘globals’, ‘hasattr’, ‘hash’, ‘help’, ‘hex’, ‘id’, ‘input’, ‘int’, ‘isinstance’, ‘issubclass’, ‘iter’, ‘len’, ‘license’, ‘list’, ‘locals’, ‘map’, ‘max’, ‘memoryview’, ‘min’, ‘next’, ‘object’, ‘oct’, ‘open’, ‘ord’, ‘pow’, ‘print’, ‘property’, ‘quit’, ‘range’, ‘repr’, ‘reversed’, ‘round’, ‘set’, ‘setattr’, ‘slice’, ‘sorted’, ‘staticmethod’, ‘str’, ‘sum’, ‘super’, ‘tuple’, ‘type’, ‘vars’, ‘zip’]

````
## Paket

Pakete, also `packages`.

```shell
pk.py
fibp
   cal
       cal.py
   pt
        pt.py
````

(Der Codeblock bleibt unverändert, da es sich um eine Dateistruktur handelt, die nicht übersetzt werden muss.)

cal.py:

```
def f(n):
    if n < 2:
```
```

```
        return n
    else:
        return f(n-1) + f(n-2)

def fl(n):
    return list(map(f, range(5)))
```

pt.py:

```
def p(l):
    print(l, end=' ')

def pln(l):
    print(l)
```

pk.py:

```
import fibp.cal.cal
import fibp.pt.pt
```

fibp.pt.pt.p(fibp.cal.cal.fl(10))

`pk.py` kann auch so geschrieben werden:

```python
from fibp.cal import cal
from fibp.pt import pt
```

pt.p(cal.fl(10)) “‘