

Carsten Knoll

Chair of Fundamentals of Electrical Engineering

Python for Engineers (2)

Pythonkurs für Ingenieur:innen (2)

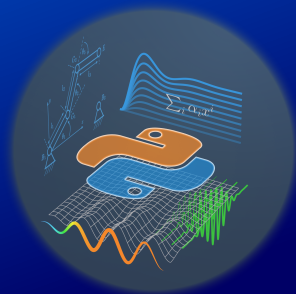
Introduction (part 2: files and functions)

Einführung (Teil 2: Dateien und Funktionen)

Dresden (Online), 2024-10-25

<https://tu-dresden.de/pythonkurs>

<https://python-fuer-ingenieure.de>



File access

function `open` (details see official docs: docs.python.org/3/...)

```
# Writing
```

```
myfile = open("my_file.txt", "w")  
myfile.write("Hello dear file.")  
myfile.writelines(["more\n", "content\n"])  
myfile.write(str(5))  
myfile.close()
```

```
# Reading
```

```
myfile = open("my_file.txt", "r")  
header = myfile.read(10) # read first 10 characters  
lines = myfile.readlines() # list of lines (starting at file cursor)
```

File access

function `open` (details see official docs: docs.python.org/3/...)

```
# Writing
```

```
myfile = open("my_file.txt", "w")  
myfile.write("Hello dear file.")  
myfile.writelines(["more\n", "content\n"])  
myfile.write(str(5))  
myfile.close()
```

```
# Reading
```

```
myfile = open("my_file.txt", "r")  
header = myfile.read(10) # read first 10 characters  
lines = myfile.readlines() # list of lines (starting at file cursor)
```

```
open("xy.pdf", "rb") Or open("xy.pdf", "wb")
```

File access

function `open` (details see official docs: docs.python.org/3/...)

```
# Writing
```

```
myfile = open("my_file.txt", "w")
myfile.write("Hello dear file.")
myfile.writelines(["more\n", "content\n"])
myfile.write(str(5))
myfile.close()
```

```
# Reading
```

```
myfile = open("my_file.txt", "r")
header = myfile.read(10) # read first 10 characters
lines = myfile.readlines() # list of lines (starting at file cursor)
```

```
open("xy.pdf", "rb") Or open("xy.pdf", "wb")
```

Strong recommendation: context manager syntax (using keyword `with`):

```
with open("my_file.txt") as myfile:
    lines = myfile.readlines()
# myfile.close() is called automatically at the end of this block
```

Functions in Python (1)

- encapsulate recurring subtasks
→ deal with complexity
- common mistakes:
 - forgot colon (syntax error)
 - `return` forgotten (→ `None` is returned)

```
def donothing():  
    pass # dummy statement (for Indentation)  
  
# Call a function without arguments  
print(donothing()) # -> None  
  
def square1(z):  
    """Squaring a number""" # Docstring (=built-in doc)  
    return z**2  
  
square1(7) # -> 49  
square1(7-12) # -> 25
```

Global and Local Variables

```
def square2(z):  
    x = z**2 # local variable (because write access)  
    print(x)  
    return x
```

```
x, a = 5, 3 # "unpacking" of tuple (5, 3)  
square2(a) # -> 9  
square2(x) # -> 25  
print(x) # -> 5
```

```
def square3(z):  
    print(x) # here x is a global variable (only read access)  
    return z**2
```

```
def square4(z):  
    print(x) # Error (local variable not yet known)  
    x = z**2 # write access -> x must be local variable  
    return x
```

∃ keywords `global` and `nonlocal` (Explanation; but in general not recommended)

Default Arguments

```
def square5(z=8):  
    return z**2
```

```
square5() # -> 64
```

```
square5(2.5) # -> 6.25
```

Default Arguments

```
def square5(z=8):  
    return z**2
```

```
square5() # -> 64
```

```
square5(2.5) # -> 6.25
```

```
def square_sum(a, b=0):  
    return a**2 + b
```

```
square_sum(10) # -> 100
```

```
square_sum(10, 3) # -> 103
```


Default Arguments

```
def square5(z=8):  
    return z**2
```

```
square5() # -> 64  
square5(2.5) # -> 6.25
```

```
def square_sum(a, b=0):  
    return a**2 + b
```

```
square_sum(10) # -> 100  
square_sum(10, 3) # -> 103
```

```
# explicit naming of the arguments ("keyword args")  
square_sum(b=-3, a=10) # -> 97 (here: order does not matter)
```

explicit naming helpful for functions with many arguments

Arbitrary Number of Arguments (*args, **kwargs)

positional arguments

```
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
```

```
mysum(5, 20, 3) # -> 28
```

Arbitrary Number of Arguments (*args, **kwargs)

positional arguments

```
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
```

```
mysum(5, 20, 3) # -> 28
```

```
numbers = [7, 4, -3, 15]
```

```
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

Arbitrary Number of Arguments (*args, **kwargs)

positional arguments

```
def mysum(*args):          # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
```

```
mysum(5, 20, 3) # -> 28
```

```
numbers = [7, 4, -3, 15]
```

```
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

keyword args

```
def sentence(**kwargs):    # the name `kwargs` is just a convention
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"The {key} is {value}." )
```

```
sentence(water="cold") # -> The water is cold.
```

```
sentence(u=8.2) # -> The u is 8.2.
```

Arbitrary Number of Arguments (*args, **kwargs)

positional arguments

```
def mysum(*args):          # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
```

```
mysum(5, 20, 3) # -> 28
```

```
numbers = [7, 4, -3, 15]
```

```
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

keyword args

```
def sentence(**kwargs):    # the name `kwargs` is just a convention
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"The {key} is {value}." )
```

```
sentence(water="cold") # -> The water is cold.
```

```
sentence(u=8.2) # -> The u is 8.2.
```

```
def complex_function(a, b, c, *args, **kwargs):
    ... # combination is possible
```

Functions as Objects → scoping

```
def plus(a, b):  
    return a + b  
  
print( type(plus) ) # -> function  
z = plus # assignment (dt: Zuweisung) (no function call!)  
print( type(z) ) # -> function  
z == plus # -> True  
z is plus # -> True  
z(2, 3) # -> 5
```

Functions as Objects → scoping

```
def plus(a, b):  
    return a + b
```

```
print( type(plus) ) # -> function
```

```
z = plus # assignment (dt: Zuweisung) (no function call!)
```

```
print( type(z) ) # -> function
```

```
z == plus # -> True
```

```
z is plus # -> True
```

```
z(2, 3) # -> 5
```

```
# factory functions ("closures")
```

```
def factory(q):
```

```
    b = q          # b is local here
```

```
    def add(a):
```

```
        return b + a # b is "global" here (from outer namespace)
```

```
    return add # an object of type `function` is returned
```

Functions as Objects → scoping

```
def plus(a, b):  
    return a + b
```

```
print( type(plus) ) # -> function
```

```
z = plus # assignment (dt: Zuweisung) (no function call!)
```

```
print( type(z) ) # -> function
```

```
z == plus # -> True
```

```
z is plus # -> True
```

```
z(2, 3) # -> 5
```

```
# factory functions ("closures")
```

```
def factory(q):
```

```
    b = q          # b is local here
```

```
    def add(a):
```

```
        return b + a # b is "global" here (from outer namespace)
```

```
    return add # an object of type `function` is returned
```

```
# add is unknown here
```

```
p1 = factory(7)
```

```
p2 = factory(1.5)
```

```
p1(3) # -> 10
```

```
p2(3) # -> 4.5
```

```
factory("The sun is ")("shining.") # evaluation without assignment
```


Argument- and Type-checking

- very flexible function calls
- frequent source of errors: wrong arguments (values or types)
- type checking often useful (effort-benefit consideration)

```
def some_function(sentence, number1, number2, a_list):  
  
    if not type(sentence) == str: # direct equality checking (not recommended)  
        print("Type error: sentence")  
        exit()  
  
    if not isinstance(number1, int):  
        print("Type error: number1")  
        exit()  
  
    if not isinstance(number2, (int, float)):  
        print("Type error: number2")  
        exit()
```

Argument- and Type-checking

- very flexible function calls
- frequent source of errors: wrong arguments (values or types)
- type checking often useful (effort-benefit consideration)

```
def some_function(sentence, number1, number2, a_list):  
  
    if not type(sentence) == str: # direct equality checking (not recommended)  
        print("Type error: sentence")  
        exit()  
  
    if not isinstance(number1, int):  
        print("Type error: number1")  
        exit()  
  
    if not isinstance(number2, (int, float)):  
        print("Type error: number2")  
        exit()
```

Recommended: compact variant in one line with keyword `assert` :

```
#  
assert isinstance(number1, int) and number1 > 0  
# -> raises an AssertionError if condition is not fulfilled  
...
```

Docstrings

- Strongly recommended:
 - Write documentation and code in the same file and (almost) simultaneously
- Docstrings $\hat{=}$ docs available from the program
- Allows for automatic creation of nice HTML doc with [Sphinx](#)

```
def calculate(x, mode="normal"):  
    """  
    Short description: A functions that does something.  
  
    :param x:      main argument (float)  
    :param mode:   mode of the algorithm (str)  
                   (e.g. "normal" or "reverse")  
  
    :rtype:        result of complicated formula (float)  
  
    After the description of the parameters, typically  
    there is more information on the documented function.  
    """  
  
    y = x**2*some_other_function(x, mode) + complicated_formula(x)  
    return y
```

Summary

- file access is simple ([docs](#))
- functions are important and powerful
- different variants to call a function ('polymorphism')
- safety queries (`assert ...`) might save much debugging time
- docstrings recommended
- documentation on defining functions: [Python Tutorial, Section 4.6](#)

Summary

- file access is simple ([docs](#))
- functions are important and powerful
- different variants to call a function ('polymorphism')
- safety queries (`assert ...`) might save much debugging time
- docstrings recommended
- documentation on defining functions: [Python Tutorial, Section 4.6](#)

→ gain your own experience