08 – Functions

Bálint Aradi

Scientific Programming in Python (2025)

https://atticlectures.net/scipro/python-2025/

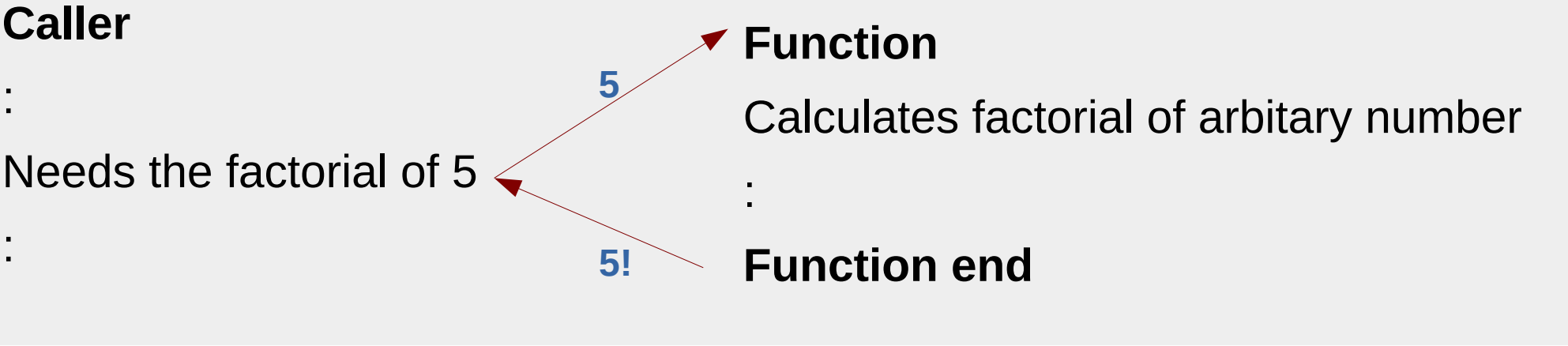
Functions

Function (procedure) = code container, which communicates with other code parts via well defined interfaces

Caller is suspended when function is called

Necessary information for function call is passed

Function execution starts



Function finishes

Result passed back to caller

Callers program execution continued

Advantages of functions

- **Partition** a problem/algorithm into small steps
- Each function can be developed, tested and improved **independently**
- Enable **code reusage**
- Help to write **descriptive code**
- **Internals** of a function are **not visible from outside**:
  - Clear programming structure
  - Implementation can be changed without affecting other code parts (provided interface remains the same)

Function definition

Arg. type
Return type
Type annotations/hints (optional)

Name
Argument(s)

```
def factorial(nn: int) -> int:
    """Calculates the factorial of a number.

    Args:
        nn: Number to calculate the factorial of.

    Returns:
        Factorial of the argument.
    """
    result = 1
    for ii in range(2, nn + 1):
        result *= ii
    return result
```

Doc-string

Use Google-docstring format
(or any established format)

Function code

Value to return
(return statement is optional)

```
aa = factorial(5)
aa
120
```

Local variables / global variables

Local variables

- Declared within the function
- Invisible outside the function
- Created each time (with no value) when function execution starts
- Destroyed each time after function finished

```
aa = factorial(5)
aa
120
result
Traceback ...
NameError: name 'result' is not defined
```

Global variables

- Declared outside of the function, but in the same module/notebook
- Visible inside the function
- Can not be modified within the function (by default)

```
BOHR_TO_ANGSTROM = 0.529177249

def convert_to_aa(val):
   return val * BOHR_TO_ANGSTROM
convert_to_aa(1.0)

0.529177249
```

Function return value

- Return statement (return value) is optional
- Without return, returned value is **None**

```
def print_greeting(name: str):
    print(f"Hello {name}!")
```

```
val = print_greeting("World")
Hello World!
print(val)
None
```

- Processing return value optional for the caller
- If not processed, return value will be discarded

```
print_greeting("World")
Hello World!
```

None, is operator

- Special (singleton) immutable object
- Signalizes missing or invalid entity usually

```
dd = {"a": 0, "b": 1}
item = dd.get("c")
print(item)
None
```

If no default argument is provided, None will be used

- The **is** operator can be used to check for None

```
if item is None:
    print("No item found")
```

- The **is not** operator can be used to check for not None

```
if item is not None:
    print("Item found")
```

- Be aware of unexpected results if checking None directly as boolean expression

```
item = dd.get("a")
if not item:
    print("No item found")
No item found
```

Note!
item = 0

- **== / !=** checks equality, **is** / **is not** identity

```
l1 = [1, 2]
l2 = l1
l1 == l2, l1 is l2
```
(True, True)

```
l2 = list(l1)
l1 == l2, l1 is l2
```
(True, False)

Function call

- Function calls always need parantheses (even for functions without parameters)

```
def hello_world():
    print("Hello world!")
hello_world()
```

- Evaluating function name without paranthesis returns the **function object** (without calling it!)

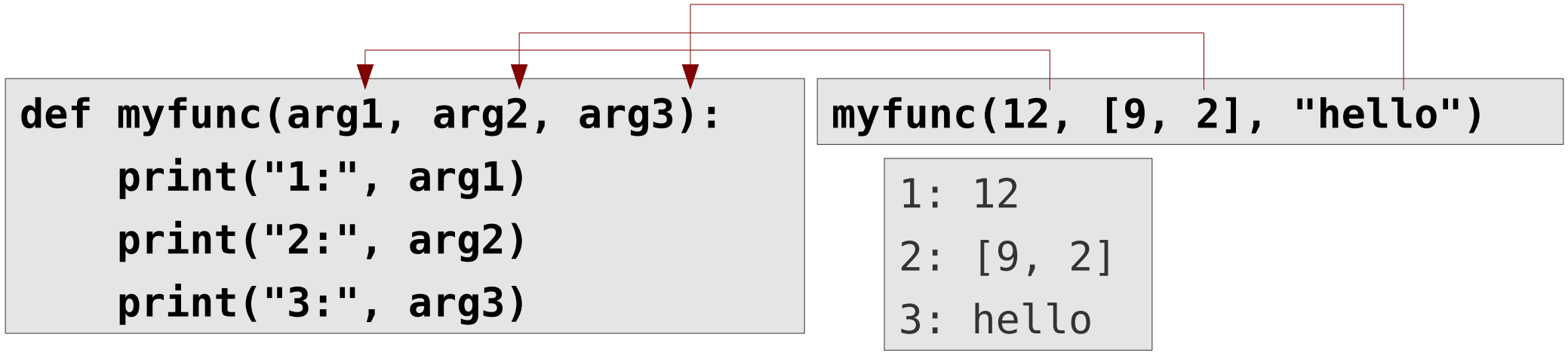
```
func = hello_world
func
<function __main__.hello_world>
```

- Actual call is can be made by evaluating the function object with appended parantheses

```
func()
```

Passing parameters

- Arguments are by default passed based on position (**positional arguments**)



```
def myfunc(arg1, arg2, arg3):
    print("1:", arg1)
    print("2:", arg2)
    print("3:", arg3)
```

```
myfunc(12, [9, 2], "hello")
```

```
1: 12
2: [9, 2]
3: hello
```

- Passing parameter is like **variable assigment**:
  argument variable in the function points to passed object instance
- Might lead to similar side effects for mutable types as during assignment

```
def with_side_effect(ll):
    ll[0] = -9
```

```
mylist = [1, 2, 3, 4]
with_side_effect(mylist)
mylist
```

```
[-9, 2, 3, 4]
```

Keyword arguments

- Arguments may have a default value (**keyword arguments**)
- If no explicit value is passed for a keyword argument, the specified **default value** is used

```
def myfunc(arg1, arg2=None, arg3="default"):
    print(f"1: {arg1}, 2: {arg2}, 3: {arg3}")
```

```
myfunc(12)
```

```
1: 12, 2: None, 3: default
```

```
myfunc(12, [1, 2])
```

```
1: 12, 2: [1, 2], 3: default
```

- Arguments can be passed in arbitrary order if their names are explicitely specified

```
myfunc(12, arg3="mystr")
```

```
1: 12, 2: None, 3: mystr
```

- Keyword arguments must be specified **after** positional arguments (in declarations & calls)

```
myfunc(12, arg2=[1, 2], "mystr")
```

```
def print_abc(a, b=2, c):
    print(a, b, c)
```

Hint: There are also keyword-only and postional-only arguments ...

Mutable type instances as argument defaults

- **Do not** use **mutables as default values in keyword arguments** as they cause **side effects!**
  (they are only created once before the first call, and not at every call)

```
def bad_example(arg=[]):
    return arg
res = bad_example()
res
```
```
[]
```

```
res.append(1)
res
```
```
[1]
```
```
res2 = bad_example()
res2
```
```
[1]
```

- Use **None** to signalize missing argument and create mutable as local variable

```
def good_example(arg=None):
    res = [] if arg is None else arg
    return res
res = good_example()
res
```
```
[]
```

```
res.append(1)
res
```
```
[1]
```
```
res2 = good_example()
res2
```
```
[]
```

Keyword-only and positional-only arguments

```
def myfunc(arg1, /, arg2, *, arg3=None):
    print(f"1: {arg1}, 2: {arg2}, 3: {arg3}")
```

Args before “/” are positional-only

Args after “*” are keyword-only

Positional-only arguments

- Can not be called with keyword

```
myfunc(arg1=9)
```

```
TypeError: ...
```

- Typical use-case: Arguments without semantic meaning

```
def mymin(x, y, /):
   return x if x <= y else y
mymin(12, 24)
```

Keyword-only arguments

- Can not be called without keyword

```
myfunc(1, 2, 3)
```

```
TypeError: ...
```

- Typical use-case: many optional arguments without a logical order

```
def plot(data, *, color=…, style=…):
   …
plot(mydata, color="blue")
```

Functions with arbitrary nr. of arguments

```
def myfunc(arg1, *args, kwarg1=None, **kwargs):
    print(f"arg1: {arg1}")
    print(f"args: {args}")
    print(f"kwarg1: {kwarg1}")
    print(f"kwargs: {kwargs}")
```

Collect further positional arguments into a tuple

Collect further keyword arguments into a dictionary

```
myfunc(1, 2, 3, kwarg1=11, kwarg2=12, kwarg3=13)
arg1: 1
args: (2, 3)
kwarg1: 11
kwargs: {'kwarg2': 12, 'kwarg3': 13}
```

- Arbitrary number of positional arguments collected in a tuple with “*”
- Arbitrary number of keywords arguments collected in a dict with “**”
- Collective arguments must be defined as last of their type (positional/keyword)

Type hints

- **Optional** information about expected type of a variable / function parameter
- Type hints are **not** used during **run-time**
- Python remains also with type hints **dynamically typed**

```
nn : int = 5
radius : float = 49.3
```

test.py

```
1  def myfunc(
2      arg1 : int,
3      kwarg1 : int = 5
4  ) -> int:
5      return (arg1 + kwarg1) * 42
6  print(myfunc(1, 2.8))
```

```
159.6
```

- **Type checker** tools and IDEs can use type hints to issue warnings **during developmen**

```
mypy test.py
test.py:6: error: Argument 2
to "myfunc" has incompatible
type "float"; expected "int"
[arg-type]
```

```
def myfunc(
    arg1 : int,
    kwarg1 : int = 5
) -> int:
    return (arg1 + kwarg1) * 42
print(myfunc(1, 2.8))
```

```
Argument of type "float" cannot be assigned to
parameter "kwarg1" of type "int" in function
"myfunc"
  "float" is incompatible with
"int" Pyright(reportArgumentType)
```

- See Support for type hints for more info