Functions & arrays

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- Functions (revisited)
- Arrays

You might need to install numpy in your Miniconda installation to try the examples

```
conda install numpy
```

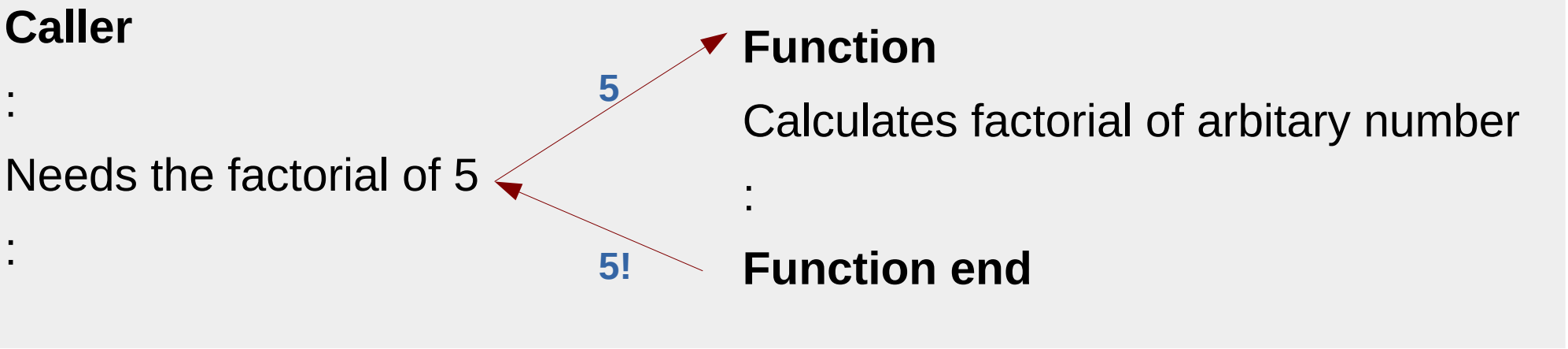
Functions

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 funtions

- **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

```
def factorial(nn):
    """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
```

Name

Argument(s)

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

```
NR_ITERS = 3
def make_iterations():
    for ii in range(NR_ITERS):
        print(ii)
make_iterations()
0
1
2
```

Function return value

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

```
def print_greeting(name):
    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 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!)

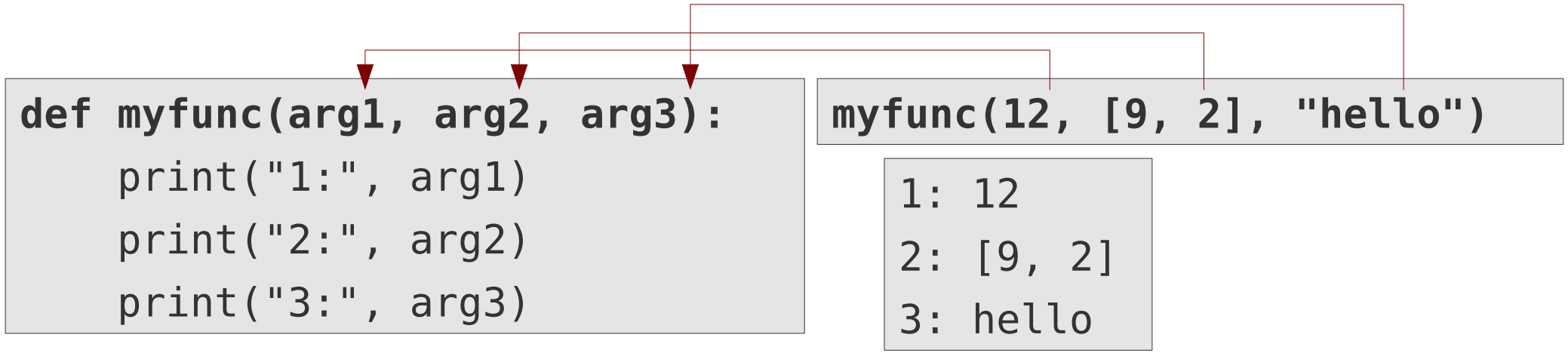
```
func = hello_world
func
<function __main__.hello_world>
```

- 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 postion-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                                []
```

Modules

- Declarations (e.g functions) in a separate file
- Better **code structuring**
- Accessible in many scripts/notebooks (reusability)
- Module must be imported before first use

```
import numpy
```

- Content is accessed by prefix notation

```
module.content
```

```
grid = numpy.linspace(-1, 1, 3)
grid
array([-1.,  0.,  1.])
```

- Module name prefix can be changed at import

```
import numpy as np
grid = np.linspace(-1, 1, 3)
```

- If only a few entities are needed, they can be **imported directly** and used without prefix

```
from numpy import linspace
linspace(-1, 1, 3)
```

- Importing all entities from a module directly is possible, but discouraged

```
from numpy import *
```

Python modules, SciPy stack

Python Standard Library

- Bundled with the Python interpreter
- **Python Standard Library documentation**

Third party modules

- Not part of the official Standard Library
- Must be installed additionally to the Python interpreter (Conda, PyPI, packages)

SciPy stack

- *De facto* standard mathematical / statistical / scientific (third party) modules

Numpy: arrays & basic numerical routines
https://docs.scipy.org/doc/numpy/

Scipy: additional mathematical routines
https://docs.scipy.org/doc/scipy/

Matplotlib: powerful graphical plotting routines
https://matplotlib.org/stable/users/index.html

Pandas: data analysis toolkit
http://pandas.pydata.org/pandas-docs/stable/

SymPy: symbolic mathematics
http://docs.sympy.org/latest/index.html

Arrays

Array = Multi-dimensional storage of elements with the same type (matrix)

```
import numpy as np
aa = np.array([1, 2, 3])
print(aa)
```

```
[1 2 3]
```

```
bb = np.array([[1, 2, 3], [4, 5, 6]])
print(bb)
```

```
[[1 2 3]
 [4 5 6]]
```

Advantages (compared to lists)

- Elements are stored sequentially in memory
  → Fast access
  (assuming the right access pattern)
- Optimized functions for array manipulation can speed up algorithm by orders of magnitude

Disadvantages (compared to lists)

- All elements must have the same type
- All strides in a multi-dimensional array must have the same length

Creating arrays

numpy.array(*expression*, dtype=*type*)

Creates an array of given type and storage order from an expression

```
np.array([1.0, 2.0], dtype=float)
```

→

```
[ 1.  2.]
```

numpy.empty(*shape*, dtype=*type*)

numpy.zeros(*shape*, dtype=*type*)

numpy.ones(*shape*, dtype=*type*)

Array with uninitialized elements / zeros / ones of given shape, type (and storage order)

```
np.ones((2, 3), dtype=float)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

- Data type: either intrinsic type (int, float, etc.) or numpy provided one (np.float32, np.float64, etc.)
- Storage order: C or Fortran (row-major / column-major)

Querying size and shape

- Object variable **size** contains the total **number of elements in the array**

```
bb = np.array([[1, 2, 3], [4, 5, 6]])
bb.size
```

```
6
```

- Object variable **shape** contains a tuple with **number of elements along each axis**

```
bb.shape
```

```
(2, 3)
```

- Array **shape can be changed**, provided size remains constant
- Order of elements in the memory remains unchanged

```
bb.shape = (3, 2)
bb
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Array element access

- Elements accessed as with nested lists
- First element indexed by 0
- Indices for multidimensional arrays collected to **index tuple**

```
bb = np.array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9])
```

`bb[1]`

```
[[1 2 3]
 [4 5 6]  → [4 5 6]
 [7 8 9]]
```

`bb[1, 1]`

```
[[1 2 3]
 [4 5 6]  → 5
 [7 8 9]]
```

- **Slices** analogously to lists as **[from:to:step]**
- Along arbitrary dimensions / axis

`bb[0,:]`

```
[[1 2 3]  → [1 2 3]
 [4 5 6]
 [7 8 9]]
```

`bb[:,0]`

```
[[1 2 3]  → [1 4 7]
 [4 5 6]
 [7 8 9]]
```

`bb[0:3:2, 0:2]`

```
[[1 2 3]  → [[1 2]
 [4 5 6]     [7 8]]
 [7 8 9]]
```

Modifying arrays

- Arrays are **mutable**, be aware of side effects

```
aa = np.array([1, 2, 3])
bb = aa
```

```
bb is aa
```
→ `True`

```
bb[0] = -1
```

```
aa
```
→ `[-1  2  3]`

- Arrays slices are pointers/views, no true copies

```
aa = np.array([1, 2, 3])
bb = aa[0:2]
```

```
bb
```
→ `[1 2]`

```
bb[0] = -1
```

```
aa
```
→ `[-1  2  3]`

- True **copy** can be enforced via **array()**

```
aa = np.array([1, 2, 3])
bb = aa
```

```
bb is aa
```
→ `False`

```
bb[0] = -1
```

```
aa
```
→ `[1  2  3]`

- True **copy** can be enforced via **array()**

```
aa = np.array([1, 2, 3])
bb = np.array(aa[0:2])
```

```
bb
```
→ `[1 2]`

```
bb[0] = -1
```

```
aa
```
→ `[1  2  3]`

Scalar operations with arrays

Arithmetic / logical operators

- Operations applied elementwise
- One of the operands must be either a scalar or operands must have identical shapes*
- Result has same shape as operand(s)

```
aa = np.array([1, 2, 3])
bb = np.array([5, 4, 3])
```

```
aa * bb
```
```
[5 8 9]
```

```
2.0 * aa
```
```
[2. 4. 6.]
```

```
aa**2
```
```
[1 4 9]
```

```
aa == bb
```
```
[False False True]
```

```
aa < bb
```
```
[True True False]
```

* or shapes which can be made identical via **broadcasting**

Universal functions (ufuncs)

- Scalar functions applicable on arrays
- Ufuncs are applied elementwise
- Result has same shape as argument

```
aa = np.array([0.0, np.pi / 2,
               -np.pi / 2])
np.sin(aa)
```

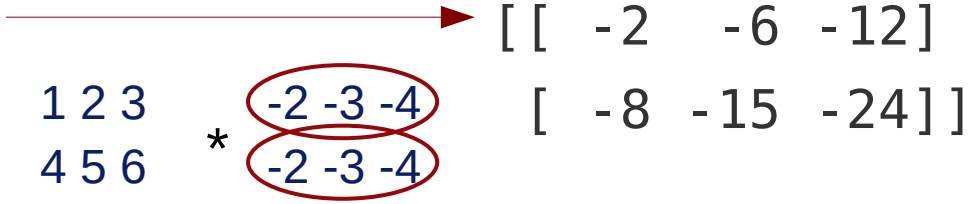
```
[0.0000e+00 1.0000e+00-1.0000e+00]
```

- Numpy module contains many useful ufuncs
  - Square root function (**sqrt**)
  - Trigonometric functions (**sin**, **cos**, …)
  - Rounding functions (**floor**, **ceil**, …)
  - etc.

Broadcasting

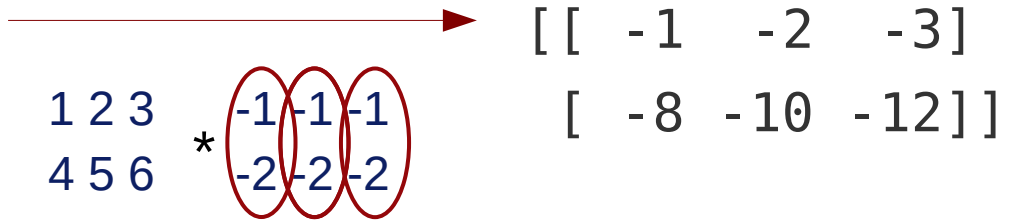
- If arrays in binary operators have different number of dimensions, numpy tries to **extend the one with less dimensions** by broadcasting:
  - **Extra dimensions inserted** before existing one(s), to equal number of dimensions
  - Array **repeated** along new dimensions to match correct shape
  - Shapes must be compatible!

```
aa = np.array([[1, 2, 3], [4, 5, 6]])
bb = np.array([-2, -3, -4])
aa * bb
```



- With **numpy.newaxis** **extra dimension(s)** can be **inserted at arbitrary position**

```
cc = np.array([-1, -2])
aa * cc[:,np.newaxis]
```



Matrix multiplication, array reduction

Matrix multiplication

- Arrays can be matrix-multiplied with the @ **operator** (eqivalent with **numpy.matmul()**)

```
aa = np.array([[1, 2], [3, 4]])
bb = np.array([5, 6])
```

```
aa @ bb
```
```
[17 39]
```

```
bb @ aa
```
```
[23 34]
```

```
bb @ bb
```
```
61
```

- **numpy.dot()** does also matrix multiplication
- For arrays with more than 2 dimensions, **numpy.dot()** and **numpy.matmul()** / @ behave differently

Array reduction

- Reduces array to scalar via given operation
- **numpy.sum(), numpy.product(), etc.**

```
vec = np.array([1.0, 2.0, 3.0])
np.sqrt(np.sum(vec**2))
```
→ `3.7416...`

- Reduction can be restricted to selected axis
- Resulting shape as before but without selected/reduced axis

```
aa = np.array([[1, 2], [3, 4]])
np.sum(aa, axis=0)
```

```
[[1, 3],
 [3, 4]]
```
→ `[4 6]`

Iterating over arrays

Arrays behave in iterations **as (nested) lists**:

- Iteration over 1D-array delivers each element
- Iteration over 2D-array delivers the rows of the 2D-array as 1D-arrays
- :

```
vec = np.array([1, 2, 3])
for ind, elem in enumerate(vec):
    print(f"{ind}: {elem}")
```

→

```
0: 1
1: 2
2: 3
```

```
aa = np.array([[1, 2, 3], [4, 5, 6]])
for ind, row in enumerate(aa):
    print(f"{ind}: {row}")
```

→

```
0: [1 2 3]
1: [4 5 6]
```

Have fun!