4 – Functions & modules, arrays

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Installing necessary packages

We need the NumPy package for this lecture. Please install it by issuing:

```
sudo apt install python3-numpy
```

In case you use Conda

```
conda install numpy
```

Functions

Function (procedure) = special container for code, which communicates with other parts of the code only via well defined interfaces

Main program is suspended

Necessary information for function call is provided

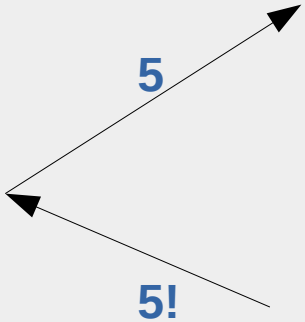
Function execution starts

Main program

:

Needs the factorial of 5

:



Function

Calculates factorial of arbitary number

:

Function end

Function finishes

Result passed back to main program

Execution of main program continues

Advantages of functions

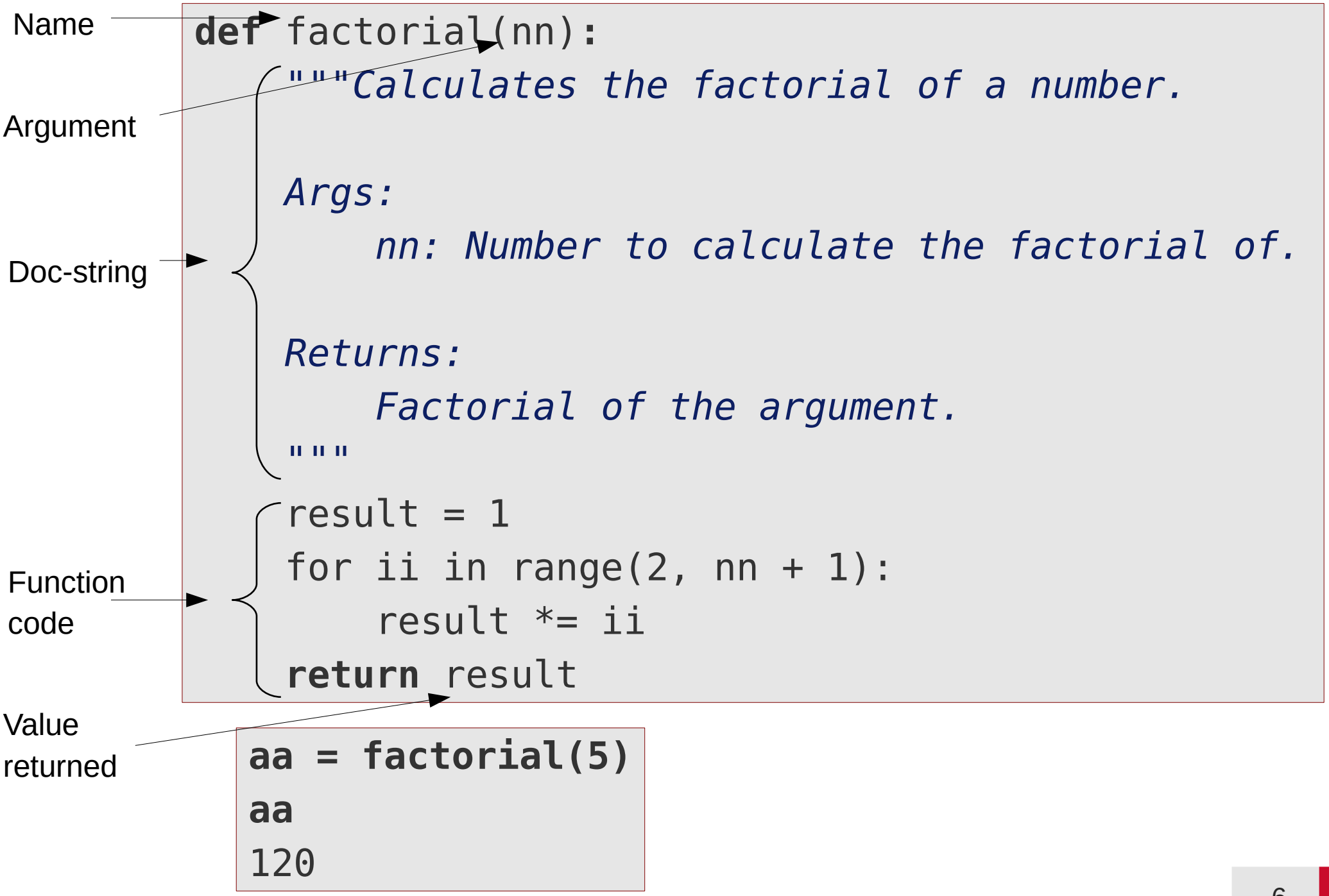
- **Partition** a problem/algorithm into small steps
- Each function can be developed, tested and improved **independently**
- Enable **code reusage**
- Help to write **descriptive code**
- **Internals** of a function are **not visible from outside**:
  - Clear programming structure
  - Implementation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

Functions

Typical function declaration:

```
def functionname(arg1, arg2, ...):
    "Documentation string"
    Subprogram statements
    ...
    return result
```

Functions


Name
Argument
Doc-string
Function code
Value returned
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
aa = factorial(5)
aa
120

Data hiding

- Variables declared within a function (**local variables**) are invisible outside the function
- The local variables are created each time (with no value) when the function execution starts
- Each time the function has finished, the local variables are destroyed

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

Function return value

- Functions do not necessary have to return any result
- If a function does not explicitely return a value, it will be set to **None**

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

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

- Storing / assigning the return value of the function is optional for the caller
- If the value is not assigned, it will be discarded

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

None

- None is a special (singleton) object in Python
- It is usually used to signalize missing or invalid entity

```
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 should be used to check whether a variable is associated with None or not:

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

- Using None directly evaluates to False, but be aware of unexpected results

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

Wrong!

Both 0 and None evaluate to False in Boolean expressions

is / is not operator

is operator

- Compares whether two variables point to the same object in memory

```
l1 = [1, 2, 3, 4]
l2 = l1
l1 == l2
True
l2 is l1
True
```

```
l1 = [1, 2, 3, 4]
l2 = [1, 2, 3, 4]
l1 == l2
True
l2 is l1
False
```

is not operator

- Negated form of the is operator
- Has been introduced to have more readable alternative for "not ... is"

```
item = dd.get("a")
if item is not None:
    print("Item found")
```

"Syntactic sugar" for:

```
if not item is None
```

Function calls

- Function calls always need parantheses, even if the function does expect any arguments

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

```
Hello world!
```

- Evaluating the function without paranthesis (without making an actual call) returns the **function object** itself

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

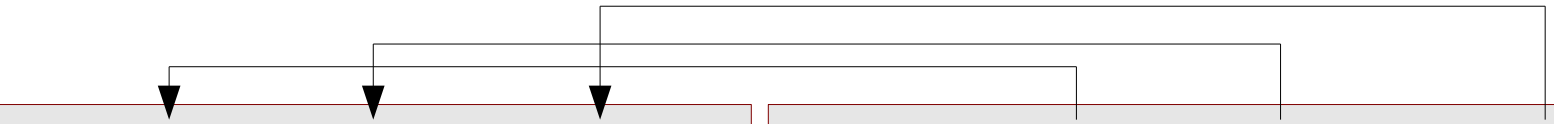
- Calling any variable containing the function object makes the actuall call

```
func()
```

```
Hello world!
```

Passing parameters to functions

- 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 a **variable assigment**: the argument variable in the function points to the passed object instance
- This can lead to similar side effects with mutable types as 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 value is passed for a keyword argument, the default value is used

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

```
myfunc(12)
```

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

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

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

- Keyword arguments can be passed in arbitrary order if their name is indicated in the call

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

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

Keyword arguments

- A keyword argument may only be followed by other keyword argument(s), both in the function declaration and the function call

```
myfunc(12, arg2=[1, 2], "mystr")
SyntaxError: positional argument follows
keyword argument
```

```
myfunc(12, arg2=[1, 2], arg3="mystr")
```

→

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

```
def print_abc(a, b=2, c):
    print(a, b, c)
SyntaxError: non-default argument
follows default argument
```

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

Mutable types as argument defaults

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

```
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):
    if arg is None:
        return []
    else:
        return arg

res = good_example()
```

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

Function documentation

- Functions should be documented in order to describe their functionality, expected arguments, returned value, etc.
- Documentation can be placed in a string immediately after the function head declaration (docstring)
- The doc-string is free-style, but it is recommended to use an established format (e.g. Google-docstring format)

```
def fibonacci(nterm):
    """Calculates the terms of the Fibonacci series.

    Args:
        nterm: Number of terms to calculate.

    Returns:
        List of generated terms.
    """
    ...
```

Function declaration

- Functions must be declared before they are called the first time
- In small scripts they are usually declared at the beginning

```
def func1(...):
    ...

def func2(...):
    ...

# Actual program
func1(...)
func2(...)
```

- In bigger projects they are defined in separate files (**modules**) and must be **imported** before being used

Modules

- Declarations (e.g. functions, classes, constants) collected in a separate file
- Enables better **code structuring**
- A given module can be used in various many script (reusability)
- Before it can be used, the module must be imported

```
import numpy
```

- Content of a module can be accessed by prefix notation

```
namespace.content
```

```
myrange = numpy.linspace(-1, 1, 5)
myrange
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

Modules

- Module name prefix can be changed at import

```
import numpy as np
np.linspace(-1, 1, 5)
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

- If only a few entities are used from a module, they can be **imported** explicitely **into the default namespace** and used without prefix

```
from numpy import linspace
linspace(-1, 1, 5)
```

- Importing everything from a module directly into the default name space is possible, but discouraged (bad programming praxis)

```
# Do NOT import everything into default namespace
from numpy import *
```

Python modules

Python Standard Library

- Most of Pythons functionality is split into several modules
- The Python Standard Libary contains those modules, which are bundled with the Python interpreter
- See the **Python Standard Library documentation** about the standard modules and their content

Third party modules

- Modules not part of the official Standard Library
- They must be installed additionally to the Python interpreter (many of them available as package in the distribution, though)

SciPy Stack projects

- Third party modules extending Python with very strong **mathematical, scientifical, plotting and statistical capabilities**
- De facto standard, almost all scientific/mathematical/statistical scripts make use of some of the SciPy modules

Numpy – array handling and basic numerical routines:
https://docs.scipy.org/doc/numpy/

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

Matplotlib – very powerful plotting routines:
https://matplotlib.org/contents.html

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

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

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)
- Highly optimized functions for array manipulation exist

Disadvantages (compared to lists)

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

Accessing array elements

- Array elements are accessed similarly to list elements
- The first element is indexed with 0
- Indices for multidimensional arrays should be collected to one index tuple

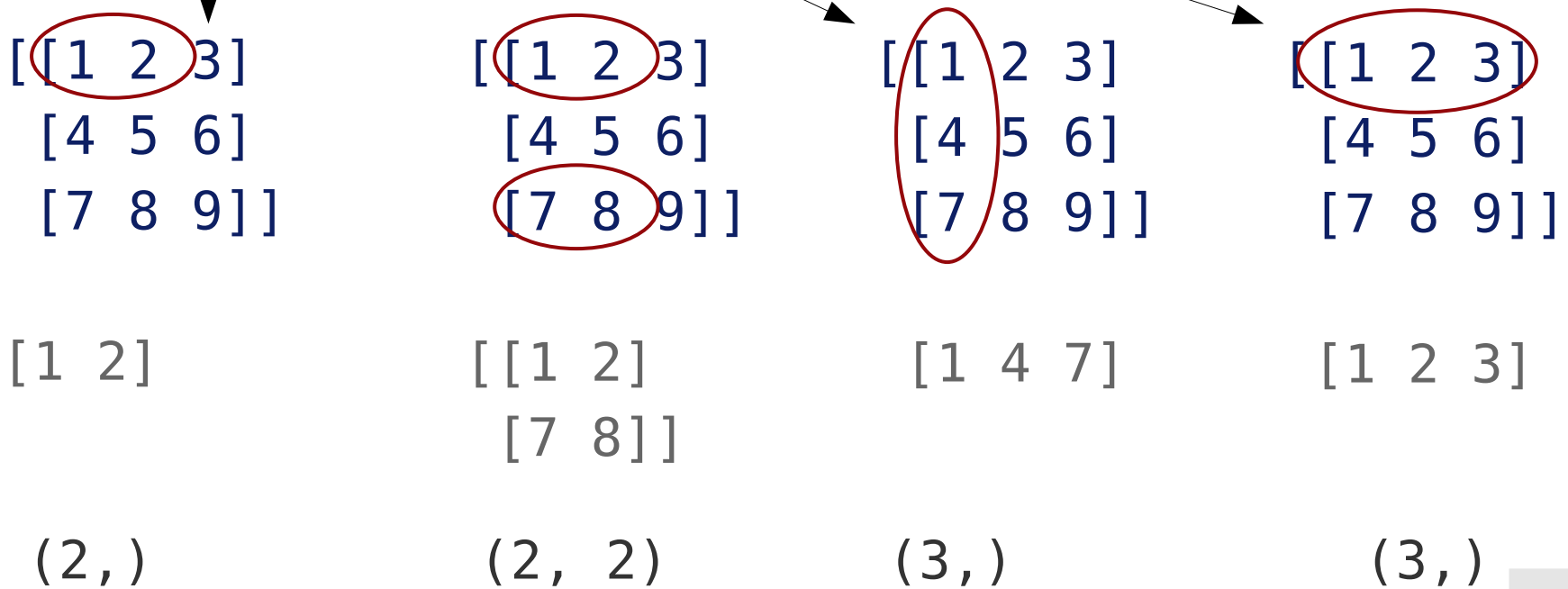
```
bb = np.array([[1, 2, 3], [4, 5, 6]])

print(bb)

print(bb[1])          Row number 1

print(bb[1, 1])       Row 1, Column 1
```

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

[4 5 6]

5
```

Slices

- **Arrays slices** can be built analogously to lists with **[from:to:step]**
- Slices are possible along arbitrary dimensions

```
bb = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(bb[0,:])
print(bb[:,0])
print(bb[0:3:2, 0:2])
print(bb[0, 0:2])
```



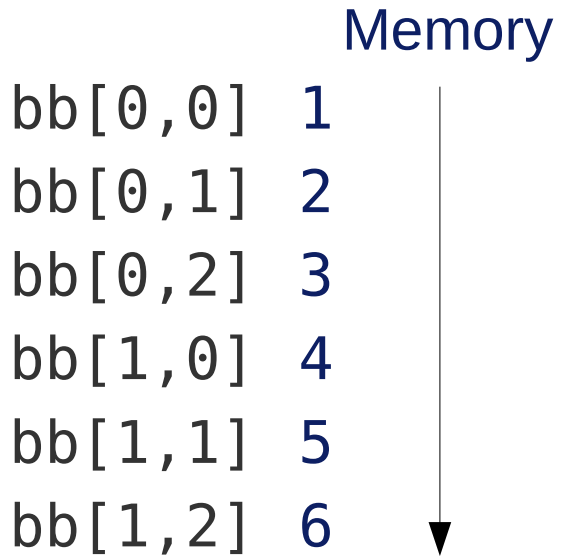
Shape:

Element storage ordering

- By default Numpy uses **row-major** array storage format (like in C): first index grows slowest, last index grows fastest:

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

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




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

- Array manipulation is usually fastest, if elements are accessed according their order in memory (cache!)
- Optionally, **column-major** storage format (like in Fortran) can be requested:

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

Querying size and shape of arrays

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

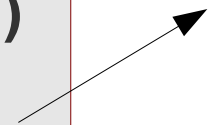
```
bb = np.array([[1, 2, 3], [4, 5, 6]])
bb.size
6
```

- Object variable **shape** contains the **number of elements along each dimension** as a tuple:

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

- The array **shape can be changed**, provided size remains constant
- The order of the elements in the memory remains unchanged

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



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

Array assignment

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

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

```
True
[-1  2  3]
[-1  2  3]
```

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

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

```
False
[1  2  3]
[-1  2  3]
```

Array assignment

- Array slices are only **pointers to the data region** of the given array, no true copies

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

```
[-1  2  3]
[-1  2]
```

- A **copy** can be enforced via the **array()** constructor

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

```
[1 2 3]
[-1  2]
```

Arithmetic operation with arrays

- All arithmetic and logical operators are declared for arrays
- The operations are always done element wise
- One of the operands must be either a scalar or both operands must be arrays with the same shape (except broadcasting…)

```
aa = np.array([1, 2, 3])
bb = np.array([5, 4, 3])
print(aa * bb)
print(2.0 * aa)
print(aa**2)
print(aa == bb)
print(aa < bb)
```

```
[5 8 9]
[ 2.  4.  6.]
[1, 4, 9]
[False False True]
[True True False]
```

Broadcasting

- If arrays in binary operators have different dimensions, numpy can **extend the one with less dimensions** in some cases by broadcasting.
  - The array with less dimensions obtains **extra dimensions inserted** before the existing one(s), to have equal number of dimensions in both
  - It will be **repeated** along the new dimensions to match the shape
  - The shape of the old dimensions must be identical with the shape of the corresponding dimensions of the bigger array

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

```
1 2 3     -2 -3 -4
4 5 6  *  -2 -3 -4
[[ -2  -6 -12]
 [ -8 -15 -24]]
```

- With **numpy.newaxis** the insertion of the **extra dimension(s)** can be **controlled manually**

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

```
1 2 3     -1 -1 -1
4 5 6  *  -2 -2 -2
[[ -1  -2  -3]
 [ -8 -10 -12]]
```

Universal functions (ufuncs)

- **Universal functions** are scalar mathematical functions, which can be appllied to the numpy arrays.
- Ufuncs are applied elementwise

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

```
[  0.00000000e+00   1.00000000e+00  -1.00000000e+00]
```

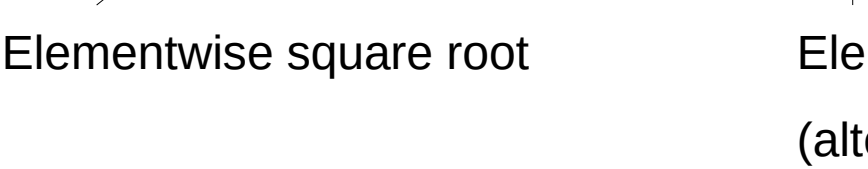
- The numpy module contains many useful ufuncs
  - Square root function (sqrt)
  - Trigonometric functions (sin, cos, …)
  - Rounding functions (floor, ceil, …)
  - Etc.

Array reduction

- Array reduction function **reduces an array to a single scalar**
- The numpy module contains useful reduction functions:
  - **numpy.sum()**: Sums all elements in an array
  - **numpy.product()**: Multiplies all elements in an array

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

```
3.7416573867739413
```



Elementwise square root

Elementwise square

(alternative: vec * vec)

Matrix multiplication

- Matrices and vectors can be multiplied with the **numpy.dot()** function

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

```
[17 39]
[23 34]
61
```

- From Python 3.5 there is a special matrix multiplication operator “@”

```
print(aa @ bb)
```

```
[17 39]
```

- Note: for arrays with more than 2 dimensions, **numpy.dot()** and the @ operator (equivalent to **numpy.matmul()**) behave differently

Creating arrays

numpy.array(*expression*, dtype=*type*, order=*'F'/'C'*)

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

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

→ `[ 1.  2.]`

numpy.empty(*shape*, dtype=*type*, order=*'F'/'C'*)
numpy.zeros(*shape*, dtype=*type*, order=*'F'/'C'*)
numpy.ones(*shape*, dtype=*type*, order=*'F'/'C'*)

Creates an array with uninitialized elements or zeros or ones of given shape, type and storage order

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

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

- Data type is either an intrinsic data type (int, float, etc.) or a numpy provided special one (np.float32, np.float64, etc.)

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 elem in vec:
    print(elem)
```

```
1
2
3
```

```
aa = np.array([[1, 2, 3], [4, 5, 6]])
for row in aa:
    print(row)
```

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