Part III

Functions, modules and packages

Example: Mathematical functions

Module `math`

- Constants `pi` and `e`
- Functions that operate on `int` and `float`
- All return values: type `float`
- import & usage: see below

```
ceil(x)
floor(x)
exp(x)
fabs(x)          # same as globally defined abs()
ldexp(x, i)      # x * 2**i
log(x [,base])
log10(x)         # == log(x, 10)
modf(x)          # (fractional, integer part)
pow(x, y)        # x**y
sqrt(x)
```

Module math (2)

- Trigonometric functions assume radians

```
cos(x); cosh(x); acos(x)
sin(x); ...
tan(x); ...

degrees(x)  # rad -> deg
radians(x)  # deg -> rad
```

- inf/nan

```
float("inf")
float("-inf")
float("nan")
```

- Use module `cmath` for complex numbers

Functions

- Functions are defined using `def`
- `return` a tuple via comma-separated items
- No `return` statement means returning `None`
- Call by object reference (see below)

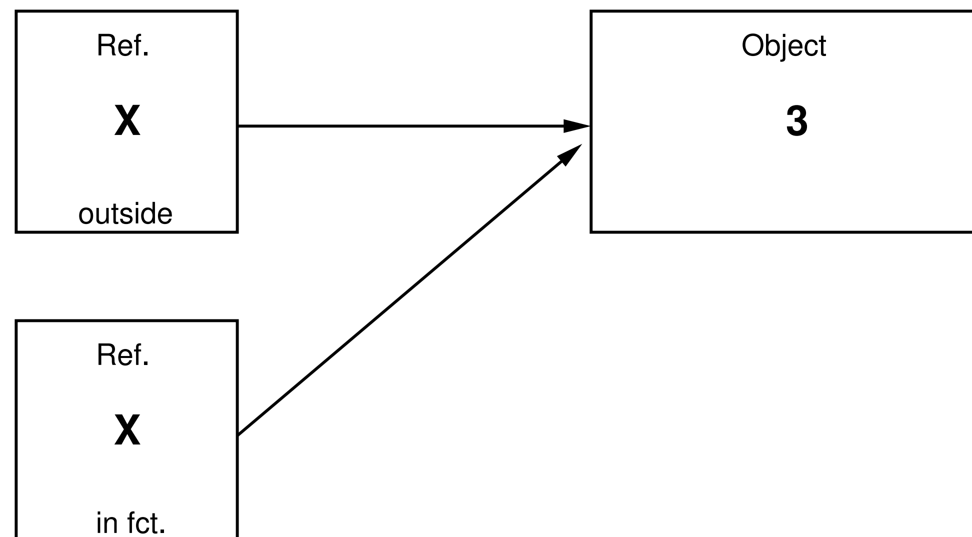
```
def hi():
    print("Hello␣World!")

def factorial(n):
    fac = 1
    for i in range(2, n+1):
        fac = fac*i
    return fac
```

- Functions are objects and can be assigned, too

```
f = factorial
f(3)
```

Call by Object Reference

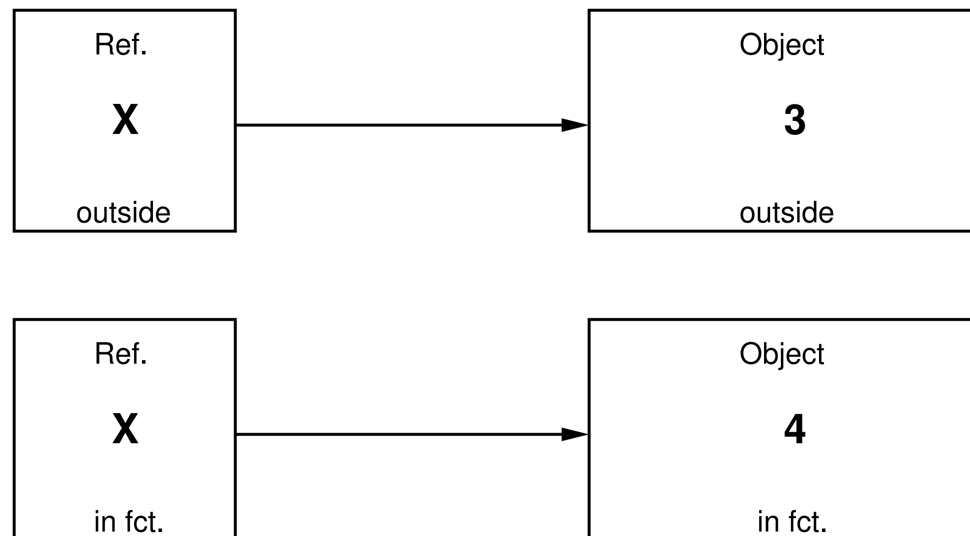
- At call of function: reference to parameter object (“address“ , id) handed over
- Reference (id) is copied (advantage: low amount of data)



Call by Object Reference (2)

- ATTENTION: Assignment changes referenced object in function!

```
>>> def add_one(x):
>>>     x = x + 1
>>> x = 3
>>> add_one(x)
>>> x               #still x==3
```



Call by Object Reference (3)

- After assignment: local variable refers to a new object
- Via methods: circumvent this effect and change actual objects outside

```
>>> def append1(lis):
>>>   lis = lis + [4]
>>>
>>> def append2(lis):
>>>   lis.append(4)
>>>
>>> lis = [1,2,3]
>>> append1(lis); lis
[1,2,3]
>>> append2(lis); lis
[1,2,3,4]
```

Functions (2)

- Default arguments

```
def printme(s="Frodo"):
    print(s)
printme()
printme("Bilbo")
```

- Keyword arguments

```
def exp(basis, exponent):
    return basis**exponent
exp(basis=2, exponent=10)
exp(exponent=10, basis=2)
```

Functions (3)

Docstrings

- Functions (and classes, modules, ...) can provide help text
- String after function header (can be multiline)

```
def answer():
    "Returns an answer to a question"
    print("answer")

help(answer)
```

Modules

- "Outsource" functionality in separate `.py` files
- Import them as library module
- Example (`tools.py`):

```python
"""This module provides some helper tools.
Try it."""

counter = 42

def readfile(fname):
    "Read text file. Returns list of lines"
    fd = open(fname, 'r')
    data = fd.readlines()
    fd.close()
    return data

def do_nothing():
    "Do really nothing"
    pass
```

Modules (2)

- Import module

```
import tools
tools.do_nothing()
print(tools.counter)
```

- Import module and change name

```
import tools as t
t.do_nothing()
```

- Import selected symbols to current namespace

```
from tools import do_nothing, readfile
from tools import counter as cntr
do_nothing()
print(cntr)
```

Modules (3)

- Import all symbols to current namespace

```
from tools import *
do_nothing()
print(counter)
```

- Modules can control which symbols are imported by `from module import *`:

```
# module tools.py
__all__ = ['readfile', 'counter']
```

Then `do_nothing()` is unknown after `import *`

Inspect namespace

- Inspect namespace of module with

```
dir(tools)
```

Modules (4)

Getting help

- Access docstrings

```
import tools
help(tools)
help(tools.do_nothing)
```

Execute module as main program

- `tools.py` should serve as program and module

```
# tools.py
...
if __name__ == '__main__':
    print("tools.py executed")
else:
    print("tools.py imported as module")
```

Modules (5)

Reload module

- Changes in a module: only active if Python is restarted
- Alternatively: `reload()` in module `importlib` (Python≥3.4).

Module search path

- Modules have to be in current directory or in directory in search path

```
import sys
sys.path.append("folder/to/module")
import ...
```

Automatically extend sys.path by setting environment variable `PYTHONPATH`

Packages

Group modules

- Modules can be grouped together
- Folder structure determines modules, e.g.:

```
tools/
    __init__.py     # contents for "import tools"
    files.py        # for "import tools.files"
    graphics.py     # for "import tools.graphics"
    stringtools/
        __init__.py  # for "import tools.stringtools"
        ... further nesting
```

- If `from tools import *` should import submodules, `tools/__init__.py` has to contain

```
__all__ = ["files", "graphics"]
```