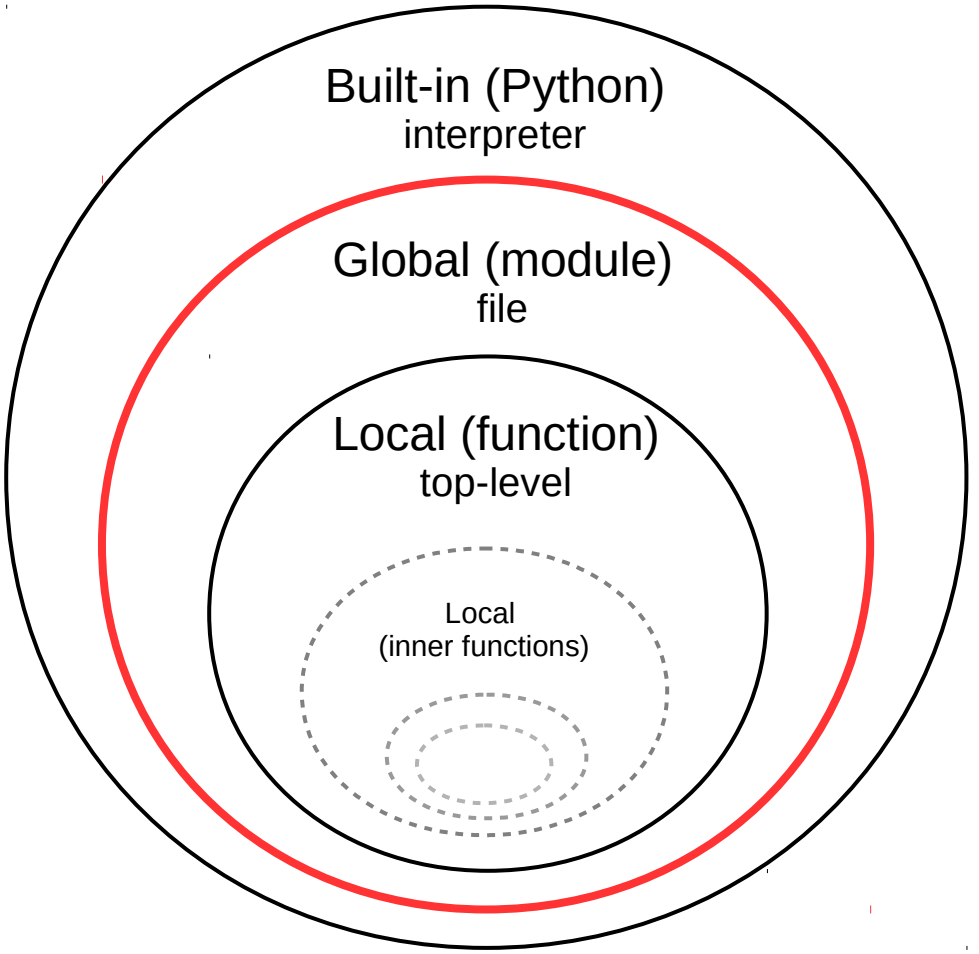
Scripting Languages

Python basics

- scope rules
- names and values
- parameters to functions
- importing modules
- module documentation
- exercise

Scope Lookup Rule

- Order of name resolution
  - local
  - global
  - built-in
- Name declaration (assignment '=')
  - new name is valid for current and lower scopes
  - new name shadows names from upper scopes
- When leaving current scope (function) all the local names are lost
- Values (object data) exist as long as they have at least one binding name
  - automatic memory management



Keywords

- Keywords are reserved names (v. 3.8)

```
import keyword

print(keyword.kwlist)
```

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

```
global = 10
       ^
SyntaxError: invalid syntax
```

Built-in functions

- Do not shadow built-in functions
  - possible, but not recommended !
    - Python 3.8 built-in functions

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

```
print = 10
print(0)

TypeError:
'int' object
is not callable
```

Example → global/local scope

```
x = "x-global"

def toplevel():
    x = "x-local"
    print(x)


print(x)
toplevel()
print(x)
```

x-global
x-local
x-global

```
x = "x-global"

def toplevel():
    print(x)
    x = "x-local"
    print(x)

print(x)
toplevel()
print(x)
```

?

Declaration → `global`

- with `global` it is possible to access (read/modify) names at global (module/file) scope

```
x = "old"

def func():
    global x
    print(x)
    x = "new"
    print(x)

print(x)
func()
print(x)
```



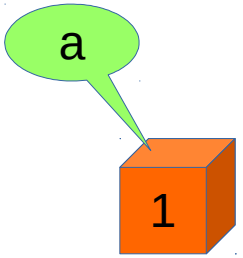
```
old
old
new
new
```

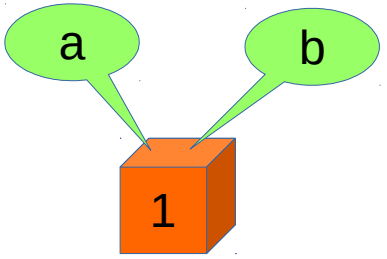
Data types modifications

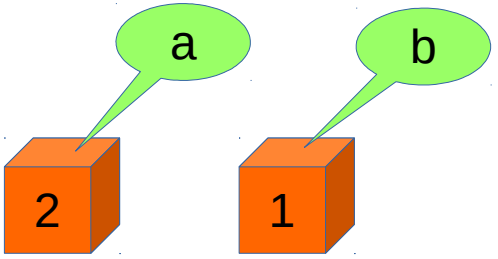
- Names and values/objects are separate items
  - names **are not** values/object data
  - any name can be bound to any data type objects
- Names are bound to values/objects
  - single value/object may have several names (bindings)
  - value/object exists as long as it has at least one binding
- **Numbers, strings, tuples** are **immutable**
  - any assignment creates a new value (object)
  - old value is lost (unless bound by other name)
- **Lists, dictionaries, sets** are **mutable**
  - adding/removing of items is done in place
  - sorting/reversing can be done in place

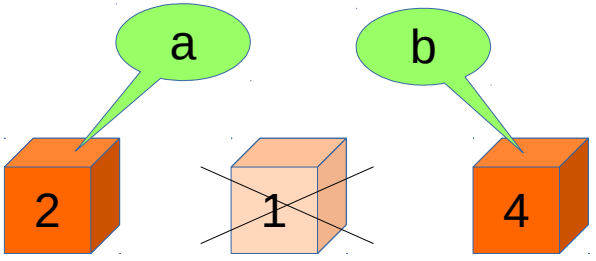
Names & numbers/strings

```
a = 1

b = a

a = 2

b = b+3
```



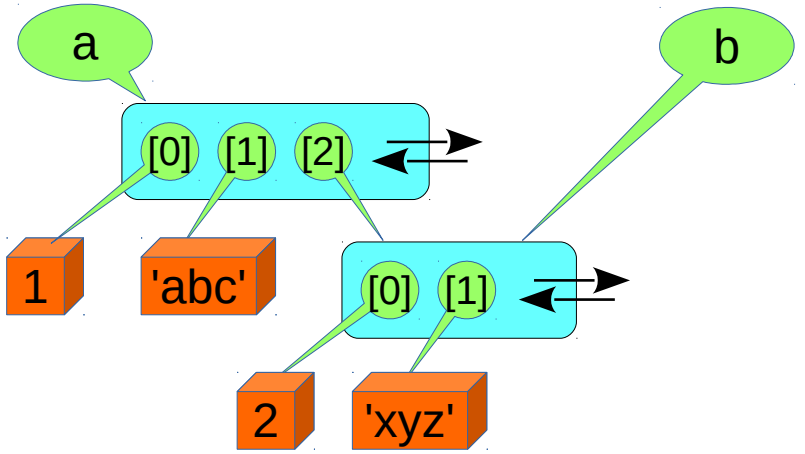


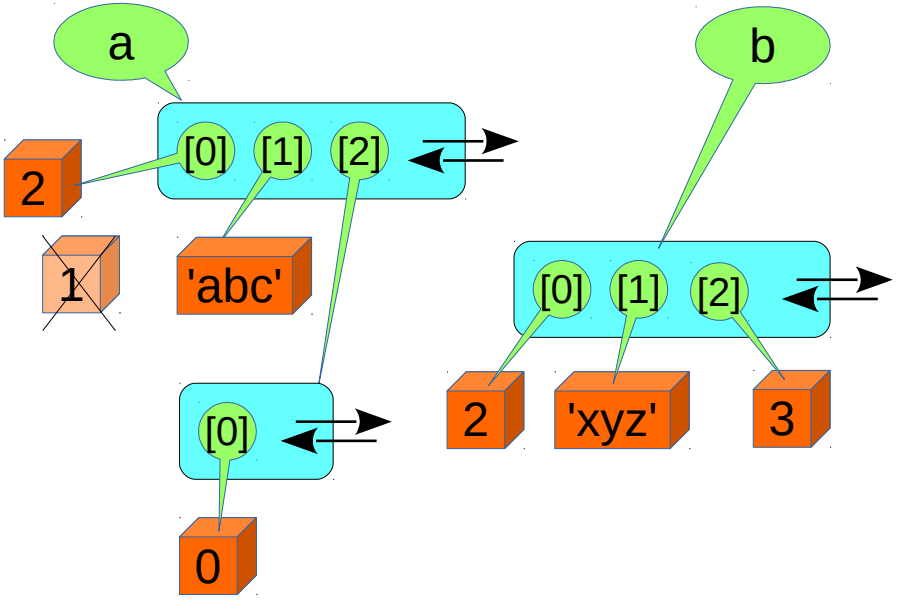




Names & lists/dictionaries

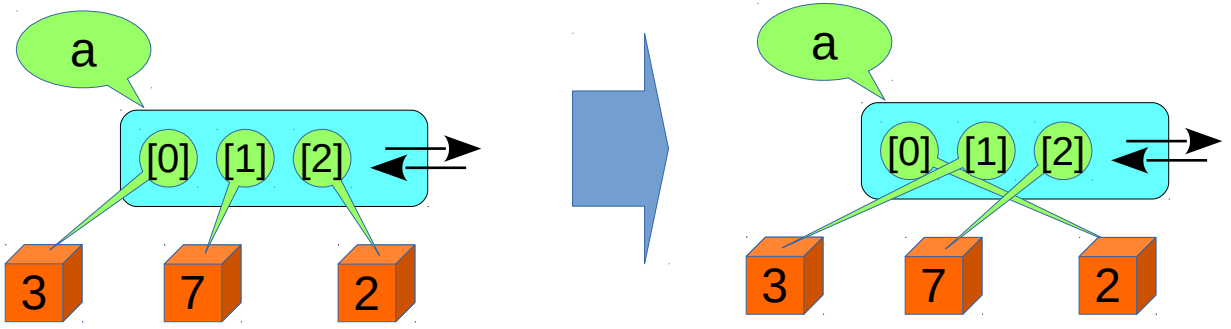
```
a = [1,'abc',[2,'xyz']]

b = a[2]




b.append(3)

a[2]=[0]

a[0]+=1
```



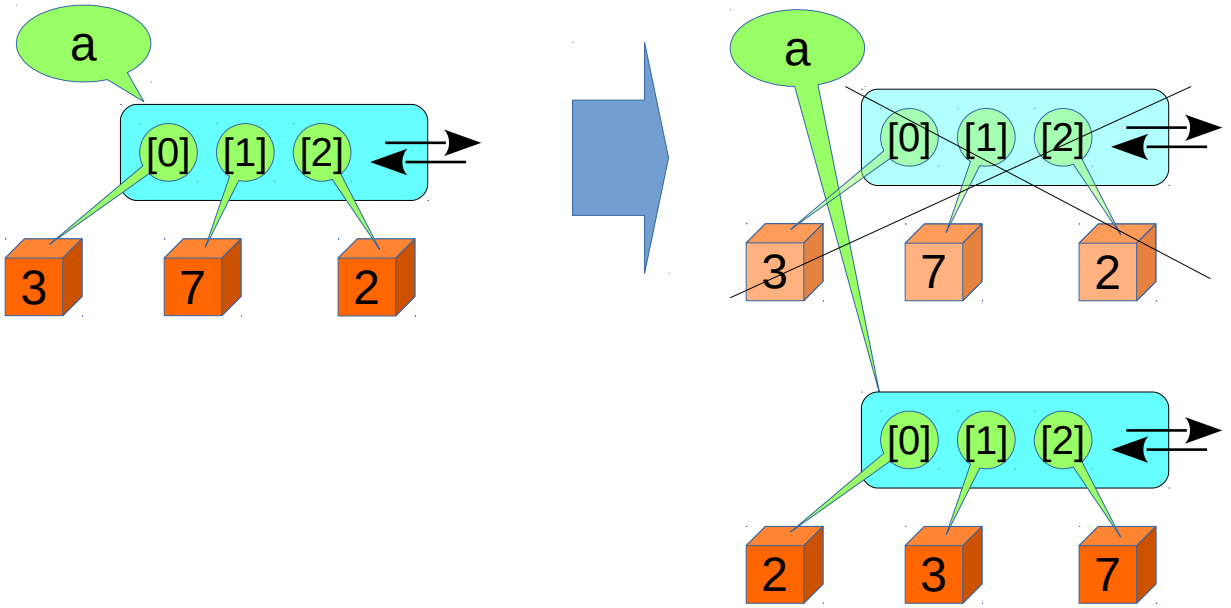


Names & lists/dictionaries

```
a = [3, 7, 2]

a.sort()
```



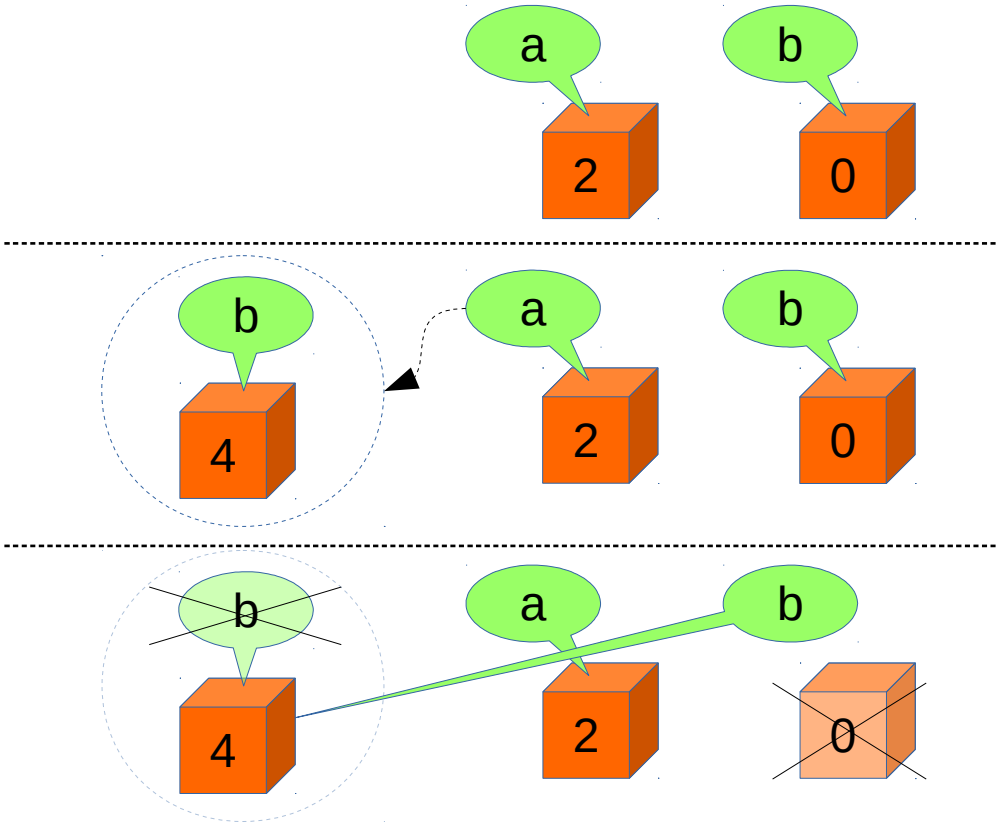
```
a = [3, 7 ,2]

a = sorted(a)
```



Names and scopes

- Inner scopes have access to outer scope names
- Local name declaration shadows outer name
- Scopes apply to names, but not to values/objects

```
def sqr():
    b = a*a
    return b

a = 2
b = 0
b = sqr()
```



Passing parameters to functions

- Parameters are passed as copies of names
- Assignments inside function binds names to new values (names become local)
  - in Python function cannot return results by modifying the parameters

```
def zero(x):
    x = 0

x = 100
zero(x)
```

Getting results from functions

- by return (recommended)
  - functions can return any collection of objects (not just single value)

```
def square(x)
    return x*x

y = square(5)
```

- by globals (limited use)
  - modify objects declared as global

```
PLN2USD = 3.85

def change_rate(x)
    global PLN2USD
    PLN2USD *= 1+x

change_rate(0.01)
```

Getting results from functions

- by elements of mutable data structures
  - lists, dictionaries

```
def zero(x):
    x[0] = 0

x = [100]
zero(x)
```

```
numbers = [1, 2]   # this a list

def change(x):
    x[0] = 2
    x[1] = 4
    x.append('Hello')

print(numbers)
change(numbers)
print(numbers)

[1, 2]
[2, 4, 'Hello']
```

Examples

1

```
def set_zero(x):
    x = 0

a = 100
set_zero(a)
print(a)


100
```

2

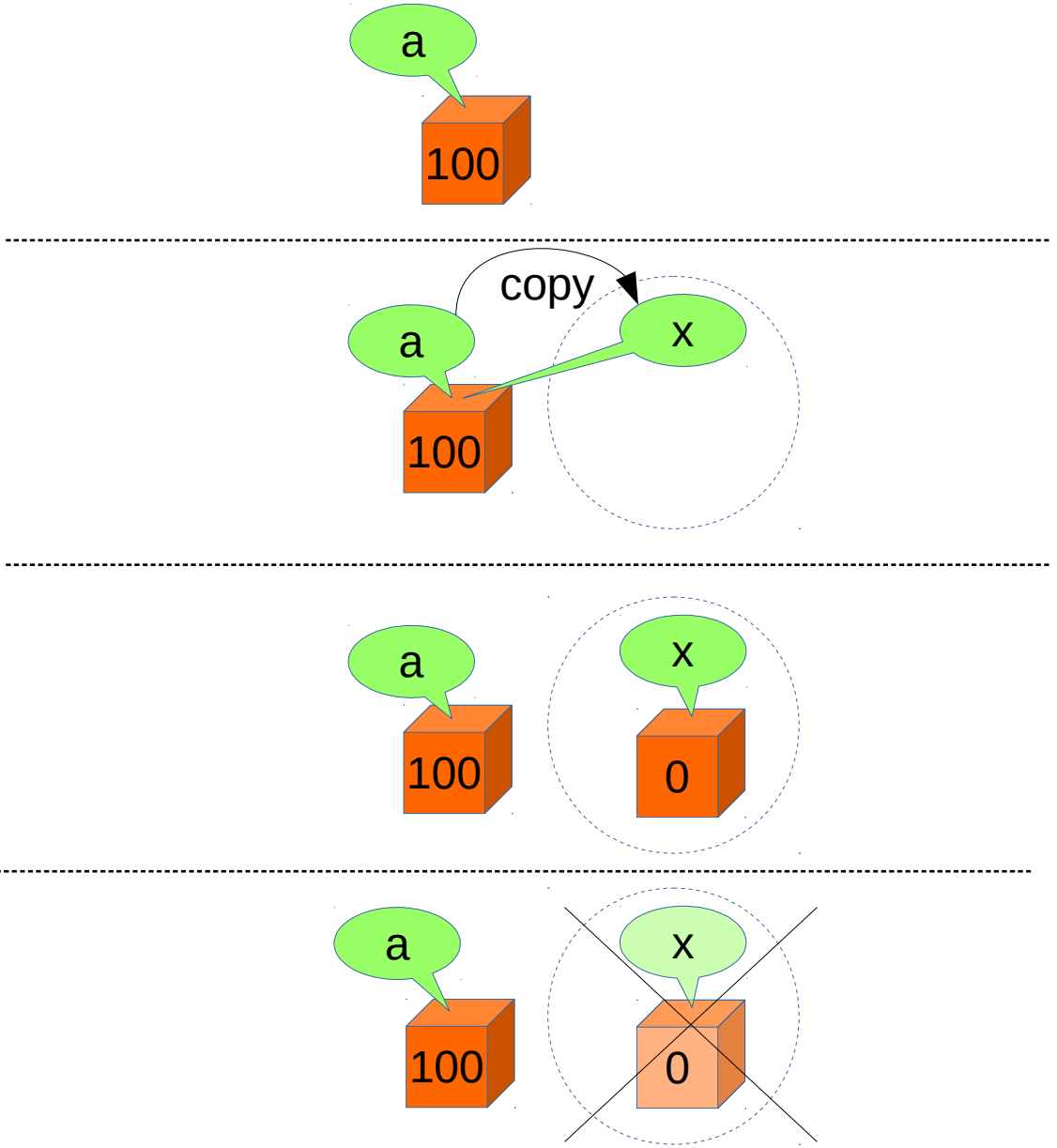
```
def set_zero(x):
    x = [0]

a = [100]
set_zero(a)
print(a)


[100]
```

3

```
def set_zero(x):
    x[0] = 0

a = [100]
set_zero(a)
print(a)


[0]
```

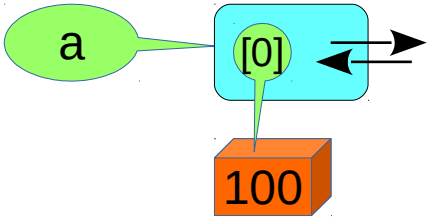
Parameters to functions (1)

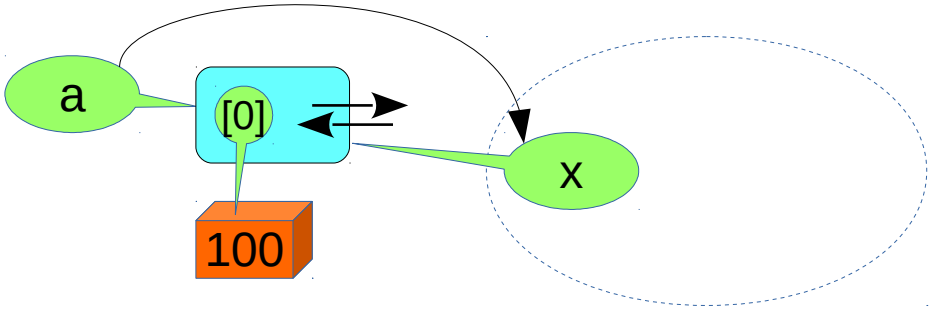
```
def set_zero(x):
    x = 0

a = 100
set_zero(a)
print(a)


100
```

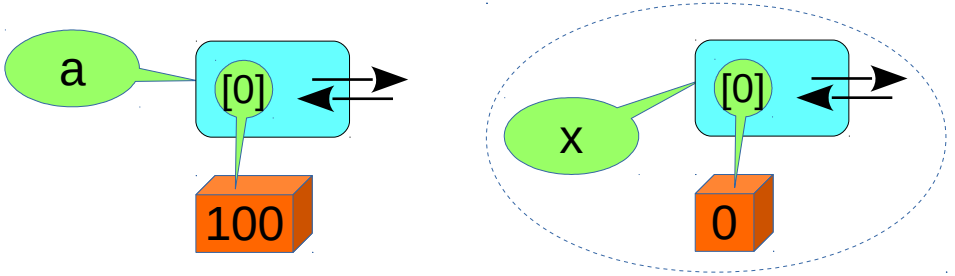


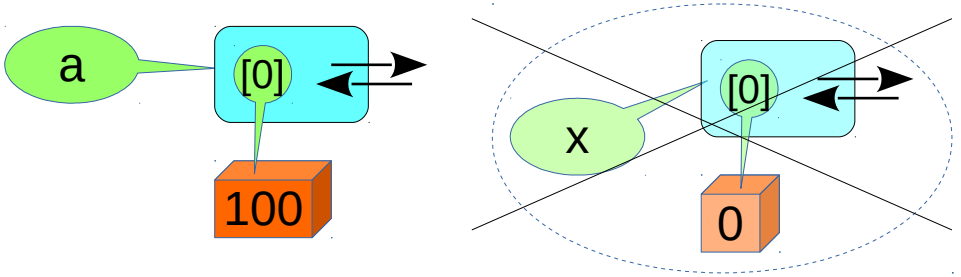
Parameters to functions (2)

```
def set_zero(x):
    x = [0]

a = [100]
set_zero(a)
print(a)


[100]
```



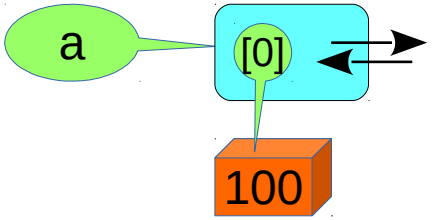


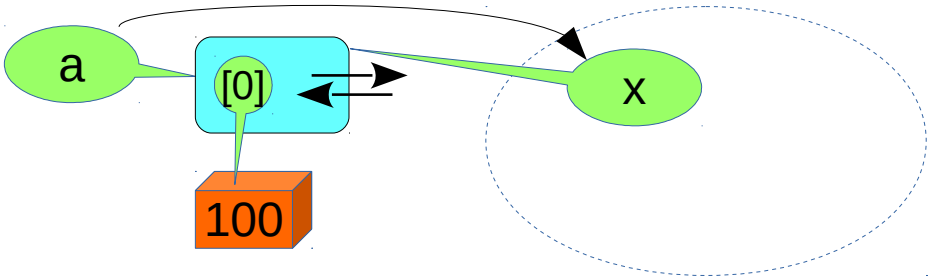


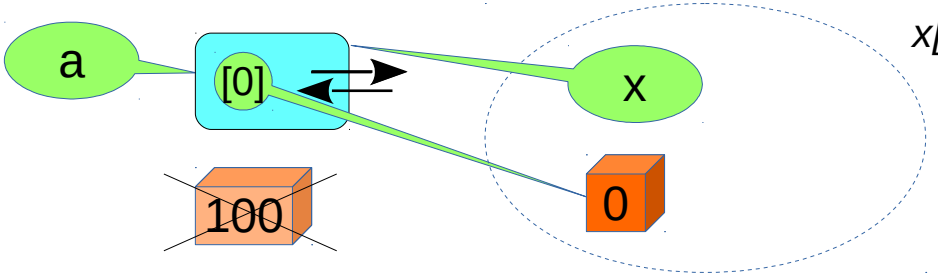


Parameters to functions (3)

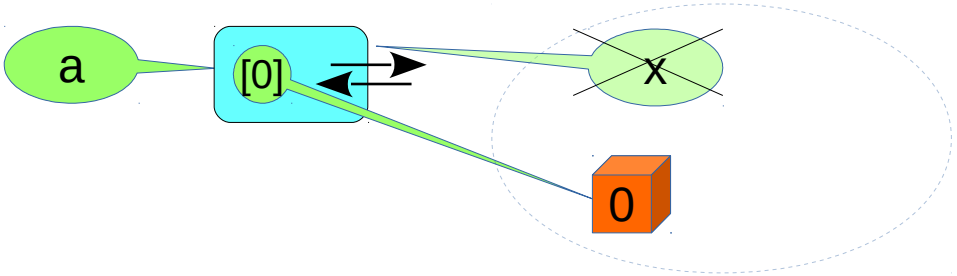
```
def set_zero(x):
    x[0] = 0

a = [100]
set_zero(a)
print(a)


[0]
```







x[0] is a[0]



Importing modules

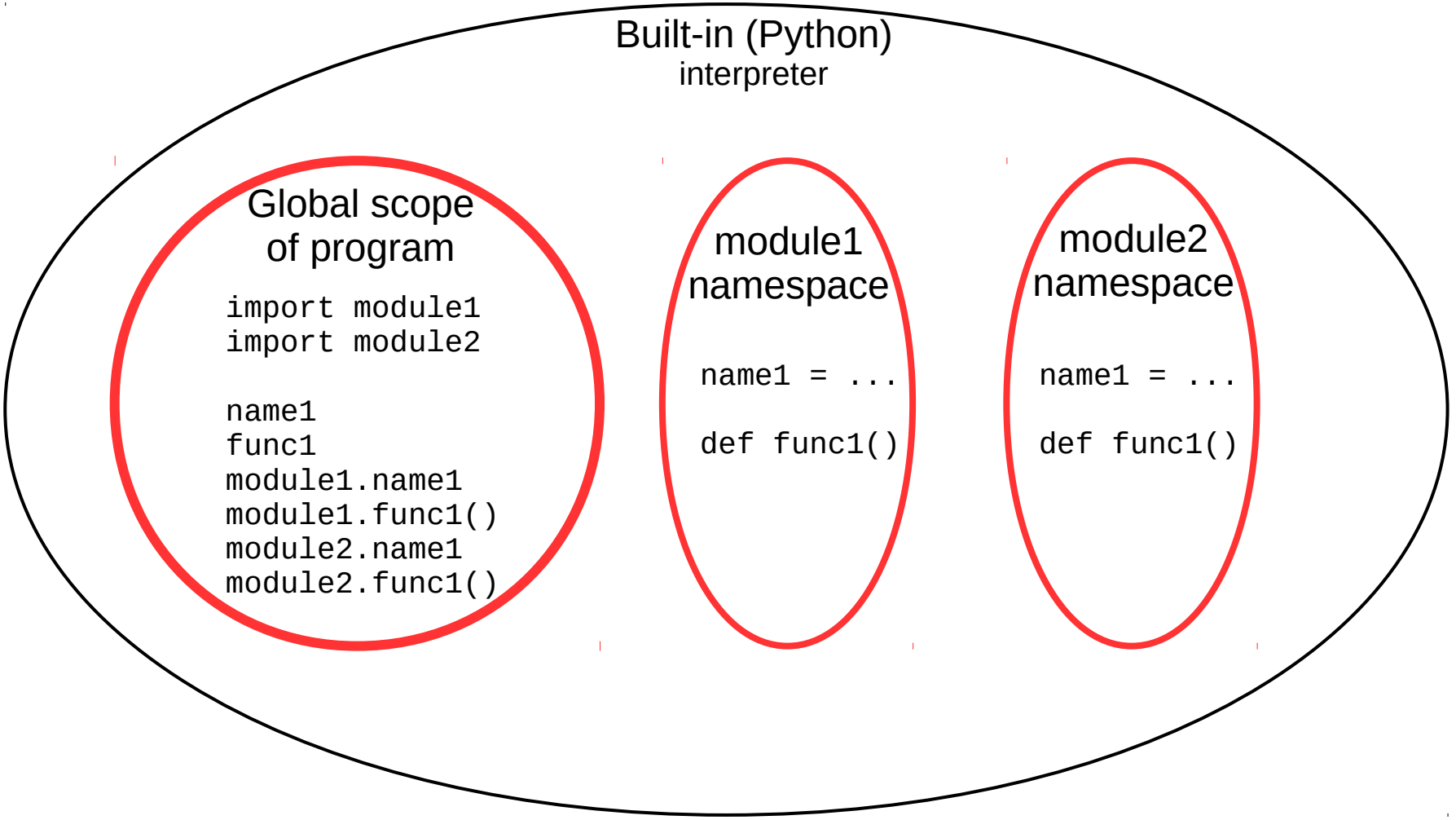
- Python code or functions from other files can be imported to the current program
  - file named 'module.py' will be included by:

```
import module
```

- when imported, all the global names and function are accessible within a namespace
  - name of the module creates a **namespace**
  - names of globals/functions are prefixed with **namespace**

```
import math

x = 2*math.pi
a = math.sin(x)
```

Module namespaces



Importing modules

- import can be selective

```
from module import func1, func2
```

- import can rename namespace

```
import long-named-module as short
```

- import can rename functions
  - `from module import func1 as otherfunc1`
- import can merge namespeces

  not recommended (but sometimes justified)
  - `from module import *`

```
import verylongname as short

a = short.function()
```

```
from math import *

x = 2*pi
a = sin(x)
```

Examining modules

- from Python interpreter:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']

>>> print(math.__name__)
math

>>> print(math.__doc__)
This module is always available.  It provides access to the
mathematical functions defined by the C standard.

>>> help(math.sin)
Help on built-in function sin in module math:
sin(...)
    sin(x)
    Return the sine of x (measured in radians).
```

Code merging

- Imported code is merged with current program
- Imported **global-scope code** is executed (!!!)

test.py

```
""" Module test"""

def test1():
    print "test1"

# global scope
print("Testing test1()")
test1()
```

```
import test


print("Hello")
```

```
Testing test1()
test1
Hello
```

Program structure

- The global-scope code should be moved to function
  - it is a good practice to use `main()` function for a global scope code
  - global declarations should be kept minimal/necessary
  - functions should exchange data explicitly (by parameters and returns, not by globals)

```
imports
declarations of globals
declarations of functions

def main():
    declarations of locals
    main program body

main() #global scope
```

Program/module structure

- Global-code should be executed conditionally:
  - when imported, its global-scope code must not be executed
  - when run as main program, its global code should be executed
- Python programs have built-in attribute `__name__`,
  - `if __name__ == "__main__":`
    - the code **is** executed as primary program
  - otherwise (`__name__` is the name of the module)
    - the code is imported as module
    - global-scope code **is not** executed

Program/module structure

- Global-scope code:
  - for modules: should contain functions and the demonstration code (to be executed when module is run as primary program)
  - for programs: the call to main() program function

program

```
imports
declarations of globals
declarations of functions

def main():
    main program body

if __name__ == "__main__":
    main()
```

module

```
imports
declarations of globals
declarations of functions

if __name__ == "__main__":
    demonstration code
```

Exercise: module & documentation

```
"""
General module documentation

Multi-line detailed
description on the module functionality

Description may contain empty lines
"""

def function():
    """
    Function documentation:

    what it does,
    what parameters it takes
    what it returns
    """
    function code


if __name__ == "__main__":
    print( function() )
```

Exercise → Fibonacci module

- Write Python module **`fib`** containing functions:
  - **`fib(n)`** → returns n-th Fibonacci number using iteration
  - **`fib_r(n)`** → returns n-th Fibonacci number using recursion
  - **`fib_series(n)`** → returns list of n-Fibonacci numbers
  - **`fib_upto(n)`** → returns list of Fibonacci numbers lower than n
- Write the missing code and documentation for functions
- Import the module from main program and run the code:
  - `fib(50)` → `12586269025`
  - `fib_r(35)` → `9227465` *(this may take many seconds!)*
  - `fib_series(10)` → `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]`
  - `fib_upto(60)` → `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`

`! Compare performance of` **`fib()`** `and` **`fib_r()`**

Making on-line documentation

- Python supports integration of code and docs
  - Documentation is kept together with the code
    - it is easy do modify and kept up-to-date
  - Documentation can be generated automatically
    - with external program "pydoc3" → to text, html, …

Run "pydoc3" from command line:

`pydoc3 module` → shows text help for a module

`pydoc3 -w module` → makes html file with documentation

`pydoc3 -b` → runs http-server with all available documentation

```
$ pydoc3 -w math
```

```
$ pydoc3 -b
Server ready at http://localhost:44236/
Server commands: [b]rowser, [q]uit
```