Python

Programming Techniques for Scientific Simulations II, 2015

Motivation

Let's break free from C-World

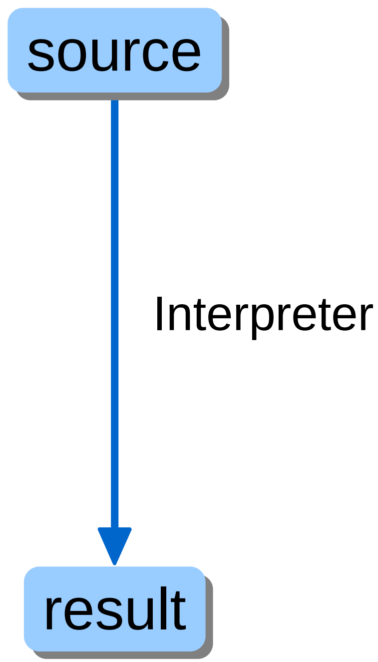
Disclaimer

- The code / examples in this lecture are written in Python 3.

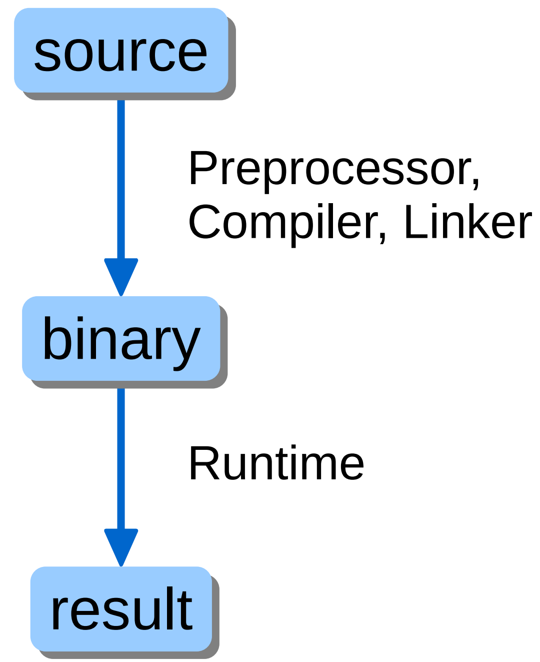
- Some differences between Python 2 and Python 3 will be discussed at the end.

Interpreted vs. Compiled

Python



C++



Typing

Python

- **Untyped** variable names:

  types not specified explicitly

- **dynamically** typed:

  variables can change type

C++

- **Typed** variable names:

  types specified explicitly

- **statically** typed:

  variables cannot change type

Built-in Types

- bool – same as C++, but with a capital letter

```
True, False
```

- int

```
x = 1
```

- float – marked by a dot

```
x = 1.
```

- str – marked by single- or double-quotes

```
x = 'string'
x = "string"
x = "I'm a string"
```

List

Python

```python
# list is built-in
x = [0, 1, 2, 3, 3]

x[2] == 2 # access via []

x.insert(0, 5) # index, value
# x == [5, 0, 1, 2, 3, 3]

# remove by index - returns value
x.pop(0) == 5
# x == [0, 1, 2, 3, 3]
```

C++

```cpp
#include <list>
std::list<int> x = {0, 1, 2, 3, 3};

std::list<int> x = {0, 1, 2, 3, 3};
auto it = x.begin();
std::advance(it, 2);
std::cout << *it << std::endl;

x.insert(x.begin(), 5);
// x == {5, 0, 1, 2, 3, 3}

// remove at iterator position
x.erase(x.begin());
// x == {0, 1, 2, 3, 3}
```

List

```
x = [0, 1, 2, 'three'] # can contain arbitrary types!

# access from the back with negative indices
x[-2] == 2

# adding lists concatenates them
x += [4, 5, 6] # x == [0, 1, 2, 'three', 4, 5, 6]

# slicing [start:end + 1]
x[1:4] == [1, 2, 'three']

# slicing with a stride [start:end + 1:step]
x[0:7:2] == x[::2] == [0, 2, 4, 6]

# reverse slicing
x[-1:0:-2] == [6, 4, 2]
x[::-2] == [6, 4, 2, 0]
```

Mutable vs. Immutable Types

- Mutable Types:
  - can change their contents / members
  - list, dict, user-defined types
- Immutable Types:
  - cannot change their contents / members
  - most built-in types (int, float, bool, str, tuple)

Tuple

- Tuples are immutable lists
- Support the same operations (slicing etc.)

list:

```
x = [1, 2, 3]

x[2] = 4 # x == [1, 2, 4]
```

tuple:

```
x = (1, 2, 3)

x[2] = 4 # TypeError!
```

Dictionary

Python

```python
x = dict(a=1, b=2, c='three')
x = {'a': 1, 'b': 2, 'c': 'three'}

# access via []
x['a'] == 1

# creating new entries
# any hashable type can be a key
x[1] = 4

# accessing keys, values or both
# order is not preserved
x.keys() # ['a', 'c', 1, 'b']
x.values() # [1, 'three', 4, 2]
x.items() # [('a', 1),
(c, 'three'), (1, 4), ('b', 2)]
```

C++

```cpp
#include <unordered_map>
std::unordered_map<char, int> x =
    {{'a', 1}, {'b', 2}, {'c', 3}};
// access via [] or at()
x.at('a') == 1;

// creating new entries
// key must be of the same type
x['d'] = 4;

// (key, value) is a pair
for(auto const & entry: x) {
    entry.first; // key
    entry.second; // value
}
```

For-Loops

Python

```python
a = [1, 2, 'b']

for x in a:
    print(x)
print(
    a[0]
)
```

C++

```cpp
#include<vector>
#include<iostream>

std::vector<int> a = {1, 2, 3};

for(auto x: a) {
    std::cout << x << std::endl;
}
std::cout << a[0]
          << std::endl;
```

For-Loops

Python

```python
a = [1, 2, 'b']

for x in a:
    print(x)
print(
    a[0]
)
```

Body:
- starts with a colon (:)
- is marked by indentation
- indentation can be tabs or
  spaces, but must be consistent

C++

```cpp
#include<vector>
#include<iostream>

std::vector<int> a = {1, 2, 3};

for(auto x: a) {
    std::cout << x << std::endl;
}
std::cout << a[0]
          << std::endl;
```

For-Loops

Python

```python
a = [1, 2, 'b']

for x in a:
    print(x)
print(
    a[0]
)
```

print:
built-in function to write to stdout

C++

```cpp
#include<vector>
#include<iostream>

std::vector<int> a = {1, 2, 3};

for(auto x: a) {
    std::cout << x << std::endl;
}
std::cout << a[0]
          << std::endl;
```

For-Loops

Python

```python
a = [1, 2, 'b']

for x in a:
    print(x)
print(
    a[0]
)
```

newline:
- marks the end of a statement
- open braces can span multiple lines

C++

```cpp
#include<vector>
#include<iostream>

std::vector<int> a = {1, 2, 3};

for(auto x: a) {
    std::cout << x << std::endl;
}
std::cout << a[0]
          << std::endl;
```

More Flow Control

Python

```python
x = 0
while True:
    if x == 10:
        break
    elif x == 1:
        x = 5
        continue
    x += 1
```

C++

```cpp
int x = 0;
while(true) {
    if(x == 10) {
        break;
    } else if (x == 1) {
        x = 5;
        continue;
    }
    x += 1;
}
```

More Flow Control

Python

```python
x = 0
while True:
    if x == 10:
        break
    elif x == 1:
        x = 5
        continue
    x += 1
```

C++

```cpp
int x = 0;
while(true) {
    if(x == 10) {
        break;
    } else if (x == 1) {
        x = 5;
        continue;
    }
    x += 1;
}
```

More Flow Control

Python

```python
x = 0
while True:
    if x == 10:
        break
    elif x == 1:
        x = 5
        continue
    x += 1
```

indentation required

C++

```cpp
int x = 0;
while(true) {
    if(x == 10) {
        break;
    } else if (x == 1) {
        x = 5;
        continue;
    }
    x += 1;
}
```

indentation only to improve readability

List- and Dict-Comprehension

- compact way of creating new lists / dicts

```
x = [1, 2, 3]
a = [val + 1 for val in x] # [2, 3, 4]

# can add a condition
b = [val + 1 for val in x if val != 2] # [2, 4]

# list comprehension: general form
[expression for variable(s) in iterable if condition]


# dict comprehension: general form
{key_expression: val_expression for variable(s) in iterable
 if condition}

y = dict(x=1, y=2, z=3)
a = {key + '_a': val + 1 for key, val in y.items() if val != 3}
# a == {'x_a': 2, 'y_a': 3}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

def keyword:
marks function definition

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

no types!

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

import keyword:
include library

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

scope operator: dot

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Functions

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

power operator: **

C++

```cpp
#include<cmath>

double d2(double dx,
          double dy,
          double dz){
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz;
    );
}
```

Named Arguments

Python

```python
def scale(p, sx=1., sy=1.):
    return p[0] * sx, p[1] * sy

# C++ - style call syntax
p = (1.2, 2.3)
scale(p, 2.)
scale(p, 1., 1.2)

# better version: named arguments
scale(p, sx=2.)
scale(p, sy=1.2)

# order becomes irrelevant
scale(p, sy=1.2, sx=2.)
scale(sy=1.2, sx=2., p=p)
```

C++

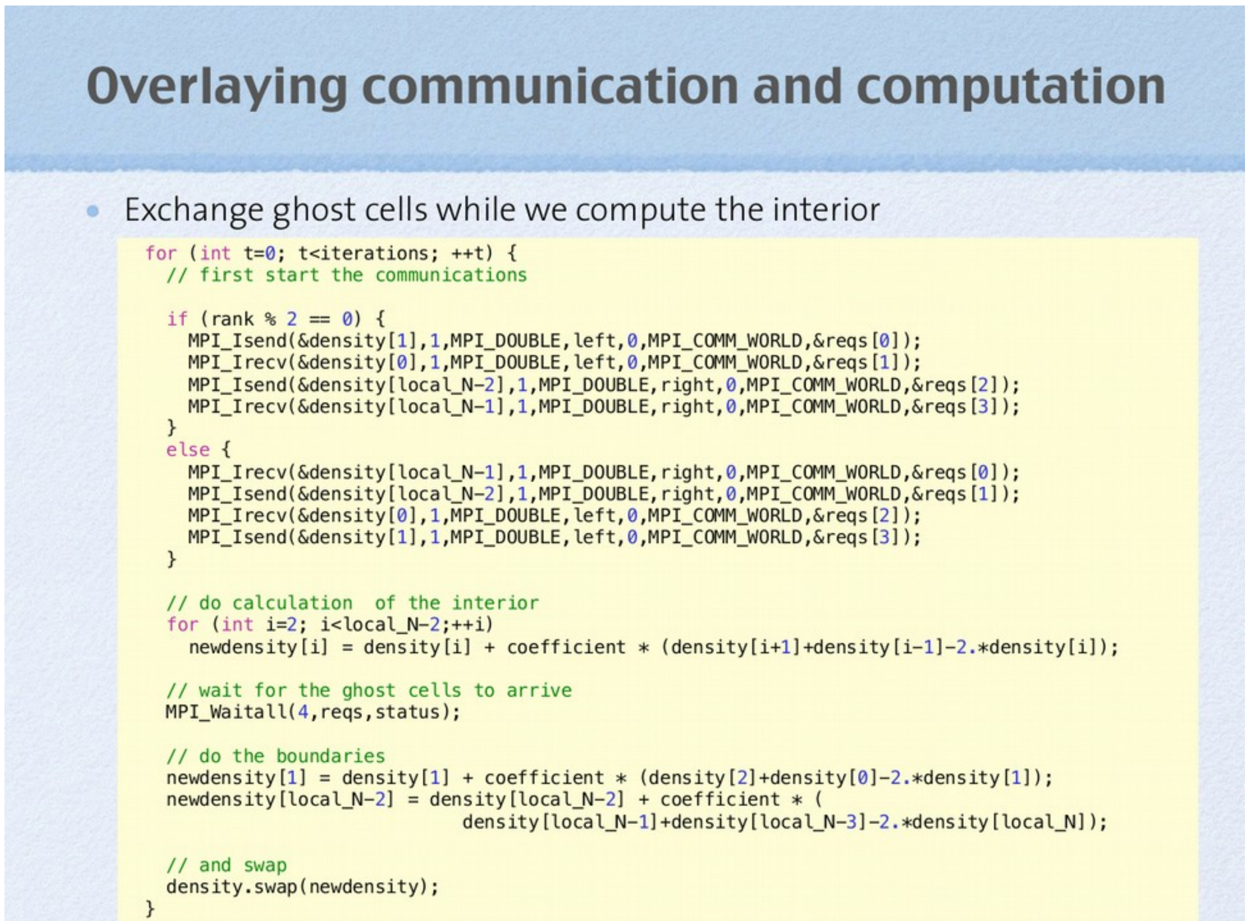
```cpp
typedef std::pair<double, double>
    point_t;

point_t scale (
    point_t p,
    double sx=1., double sy=1.
) {
    return point_t
    (p.first * sx, p.second * sy);
}

point_t p(1.2, 2.3);
scale(p, 2.) // ok
scale(p, 1., 1.2) // not so nice
```

Named Arguments

remember this?



Overlaying communication and computation

- Exchange ghost cells while we compute the interior

```
for (int t=0; t<iterations; ++t) {
  // first start the communications

  if (rank % 2 == 0) {
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[3]);
  }
  else {
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[3]);
  }

  // do calculation  of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // wait for the ghost cells to arrive
  MPI_Waitall(4,reqs,status);

  // do the boundaries
  newdensity[1] = density[1] + coefficient * (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient * (
                           density[local_N-1]+density[local_N-3]-2.*density[local_N]);

  // and swap
  density.swap(newdensity);
}
```

positional arguments can be horirlbe

*args

Python

```python
def accumulate(*args):
    res = 0
    # args is now a list
    for x in args:
        res += x
    return res

accumulate(1., 2, 2.3) # 5.3
# or, equivalently:
x = [1., 2, 2.3]
accumulate(*x)
sum(*x) # built-in function
```

C++

```cpp
double accumulate() {return 0;}

template<typename A, typename... Args>
double accumulate(
    A const & a,
    Args const & ... args
) {
    return a + accumulate(args...);
}

double x = 1.;
int y = 2;
double z = 2.3;

accumulate(x, y, z); // 5.3
```

Packing / Unpacking

∗ : **packs** positional parameters (in function **signature**):

```
def func1(x, *args):
    print(args, type(args))
func1(1, 2, 3) # (2, 3) <class 'tuple'>
```

∗ : **unpacks** iterable objects (in function **call**):

```
func1(1, 2, 3) == func1(*[1, 2, 3]) == func1(1, *(2, 3))
```

∗∗ : **packs** named arguments (in function **signature**)

```
def func2(x, y=1, **kwargs):
    print(args, type(args))
func2(1, y=2, z=3, a=4) # {'a': 4, 'z': 3} <class 'dict'>
```

∗∗ : **unpacks** mappings (in function **call**)

```
func2(1, 2, z=3, a=4) == func2(**{'x': 1, 'y': 2, 'z': 3, 'a': 4})
== func2(1, 2, z=3, **{'a': 4})
```

Function Signature

```
def func(x, y=1, *args, z, a=1, b, **kwargs)
```

`x`	**positional** arguments
`y`	positional arguments with **default**
`args`:	**variadic** positional arguments
`z, b`	**keyword-only** arguments
`a`	keyword-only arguments with **default**
`kwargs`	**variadic** keyword-only arguments

order can be mixed

```
func(positional_args, keyword_args) # call syntax
```

Packing / Unpacking

- **packing** in return statement

```
def func():
    return 1, 2, 3

a = func() # a == (1, 2, 3)
```

- **unpacking** in assignment

```
x, y, z = func() # x == 1, y == 2, z == 3

# unpack nested container
x, (y, z) = (1, (2, 3)) # x == 1, y == 2, z == 3
```

Lambda functions

```
def mul(x, y=1, z=2):
    return x * y * z
```

corresponds to

```
mul = lambda x, y=1, z=2: x * y * z
```

Lambda functions

```
def mul(x, y=1, z=2):
    return x * y * z
```

corresponds to

```
mul = lambda x, y=1, z=2: x * y * z
```

- function signature after 'lambda'

Lambda functions

```
def mul(x, y=1, z=2):
    return x * y * z
```

corresponds to

```
mul = lambda x, y=1, z=2: x * y * z
```

- function signature after 'lambda'
- single return statement

  (as in C++11 constexpr functions)

Lambda functions

```
def mul(x, y=1, z=2):
    return x * y * z
```

corresponds to

```
mul = lambda x, y=1, z=2: x * y * z
```

- function signature after 'lambda'
- single return statement

  (as in C++11 constexpr functions)
- function name not required

Passing Function Arguments

Python

```python
# pass by ?
def incr(x):
    x += 1



x = 1
incr(x)
# x == 1

# must be pass by copy!
```

C++

```cpp
// pass by copy
void incr(int x) {
    x += 1;
}

int x = 1;
incr(x);
// x == 1
```

Passing Function Arguments

Python

```python
# pass by ?
def incr_first(x):
    x[0] += 1


x = [0, 1, 2]
incr_first(x)
# x == [1, 1, 2]

# must be pass by ref!
```

C++

```cpp
// pass by reference
void incr_first(
        std::vector<int> & x
) {
    x[0] += 1;
}

std::vector<int> x = {0, 1, 2};
incr_first(x);
// x == {1, 1, 2}
```

Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

“object space”

```
x = 1
```

global scope

Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

“object space”

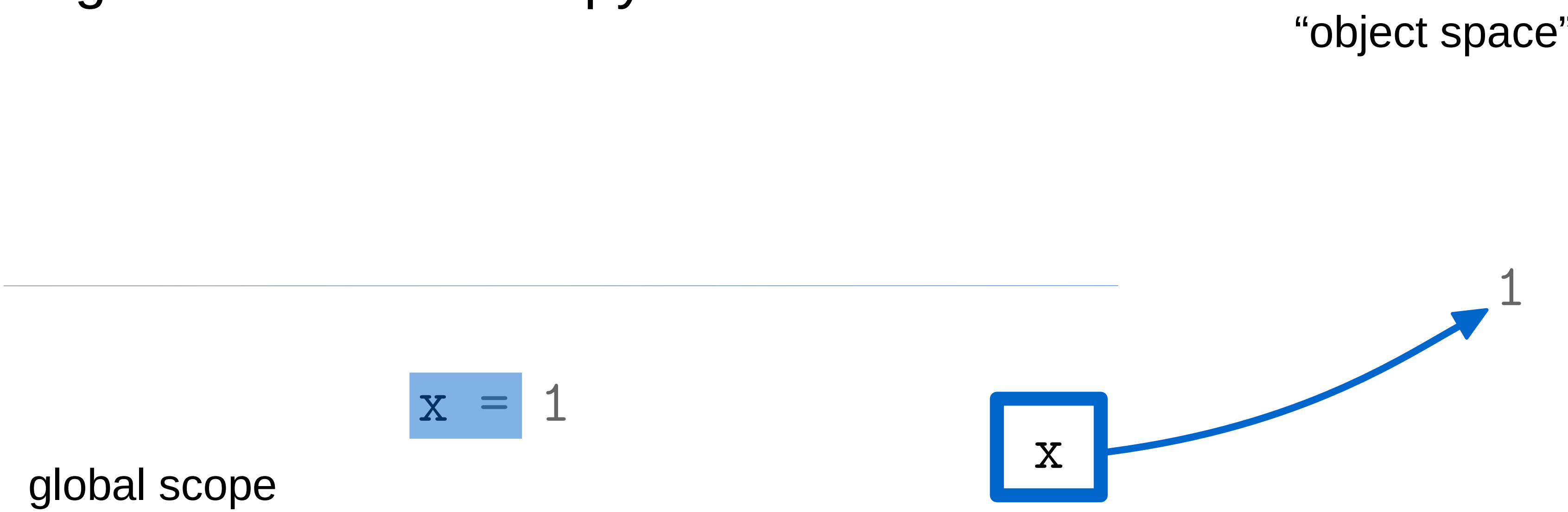
1

```
x = 1
```

global scope

Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data



Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

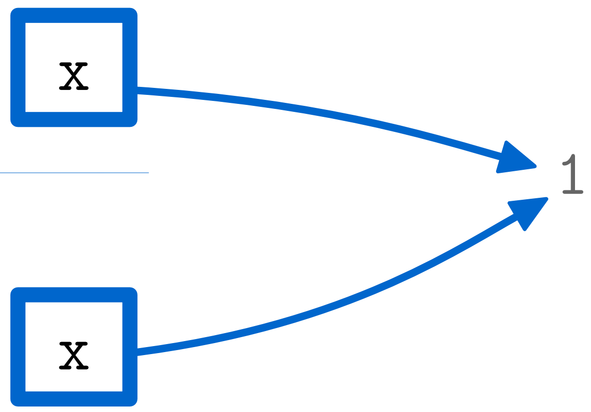
“object space”

local scope

```python
def incr(x):
```

```python
x = 1
incr(x)
```

global scope



Pass by Assignment

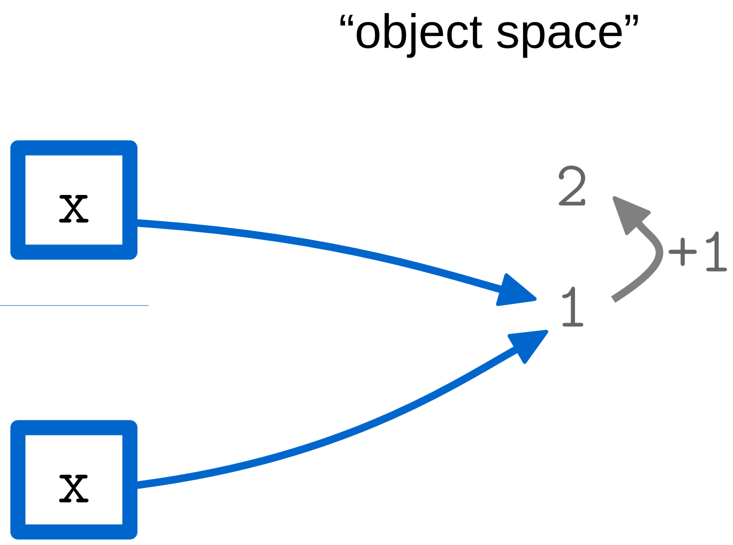
- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

local scope

```
def incr(x):
    x = x + 1
```

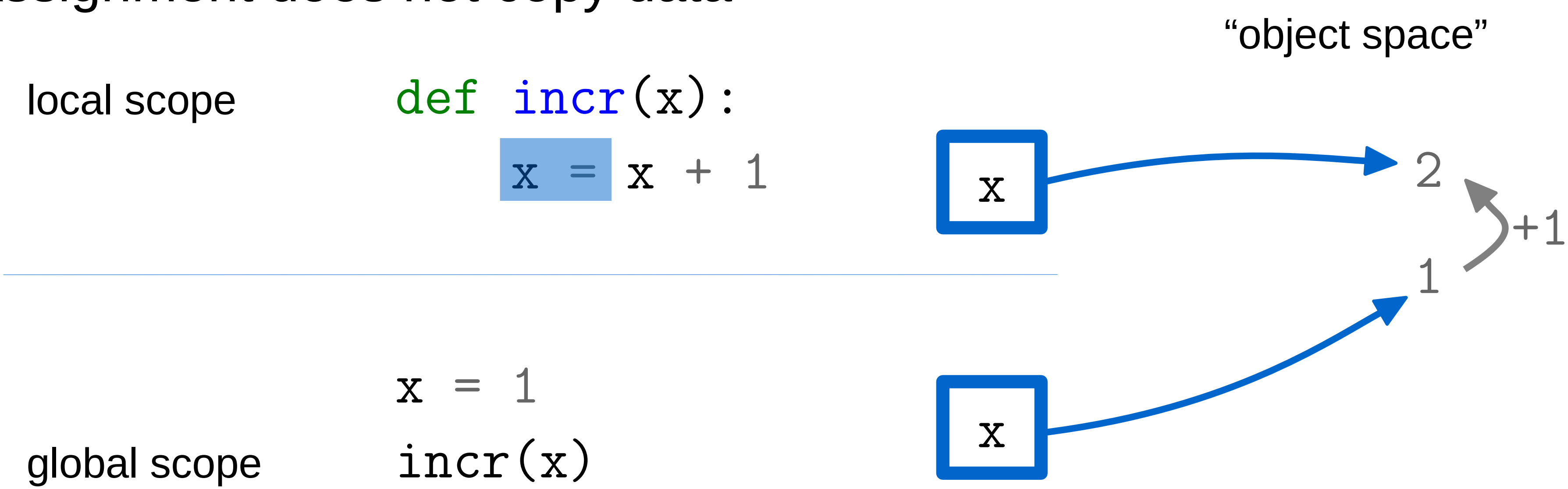
```
x = 1
incr(x)
```

global scope



Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data



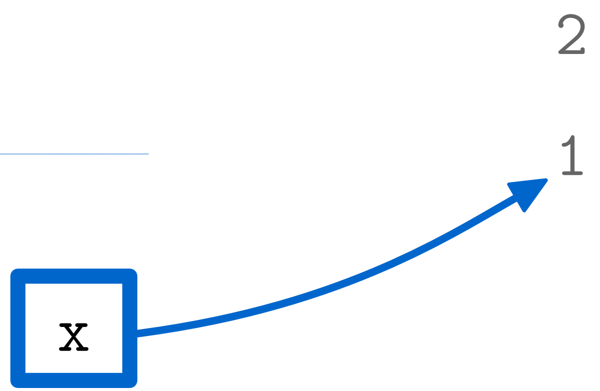
Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

“object space”

global scope

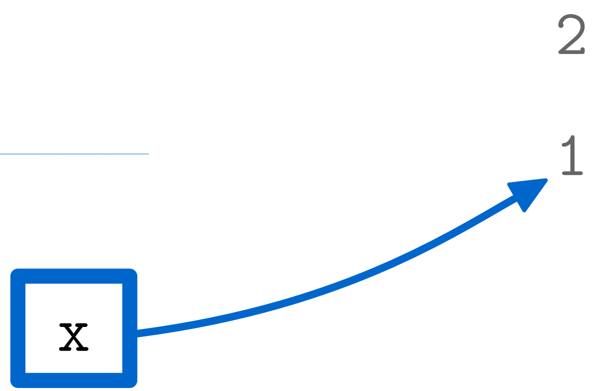
```
x = 1
incr(x)
```



Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data

“object space”

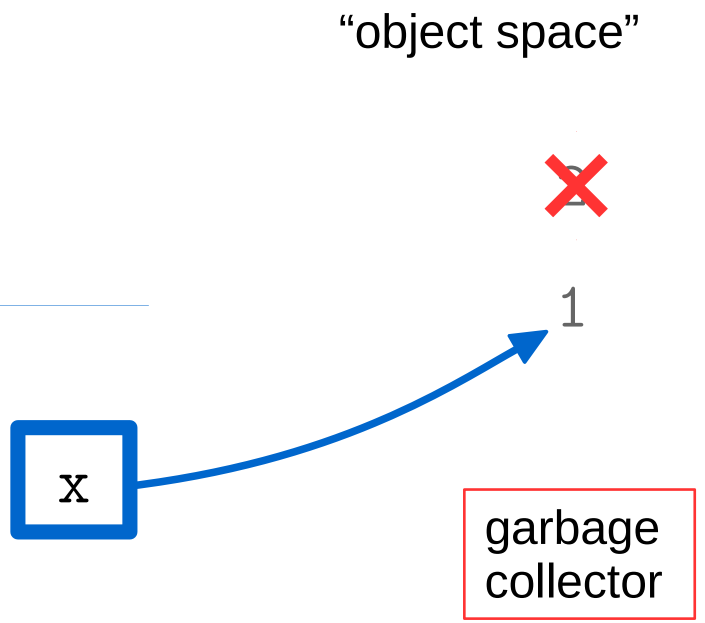


global scope

```python
x = 1
incr(x)
print(x) # 1
```

Pass by Assignment

- Variables in Python are just names (labels)
- Names “bind” to an object when assigned to
- Assignment does not copy data



global scope

```
x = 1
incr(x)
print(x) # 1
```

Pass by Assignment

“object space”

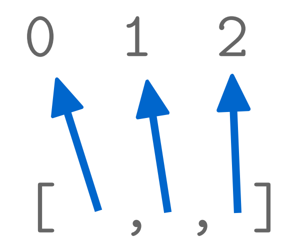
global scope

```
x = [0, 1, 2]
```

Pass by Assignment

“object space”

global scope   `x = [0, 1, 2]`

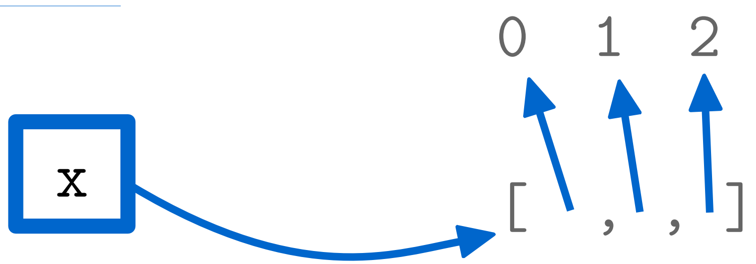


Pass by Assignment

“object space”

global scope

```
x = [0, 1, 2]
```



Pass by Assignment

"object space"

global scope

```
x = [0, 1, 2]
incr_first(x)
```

Pass by Assignment

“object space”

local scope

```
def incr_first(y):
```

global scope

```
x = [0, 1, 2]
incr_first(x)
```

Pass by Assignment

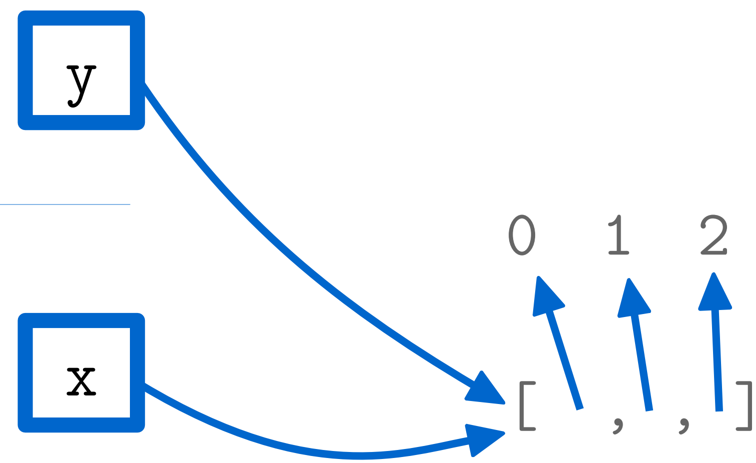
“object space”

local scope

```
def incr_first(y):
```

global scope

```
x = [0, 1, 2]
incr_first(x)
```



Pass by Assignment

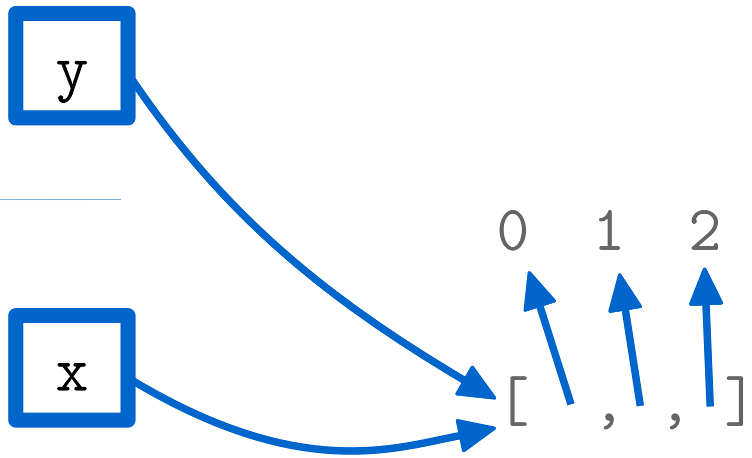
“object space”

local scope

```
def incr_first(y):
    y[0] = y[0] + 1
```

global scope

```
x = [0, 1, 2]
incr_first(x)
```



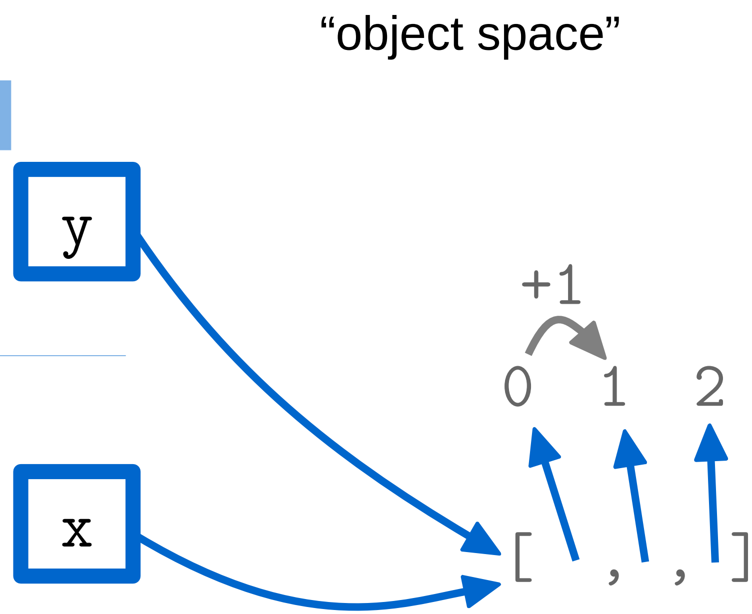
Pass by Assignment

local scope

```
def incr_first(y):
    y[0] = y[0] + 1
```

global scope

```
x = [0, 1, 2]
incr_first(x)
```

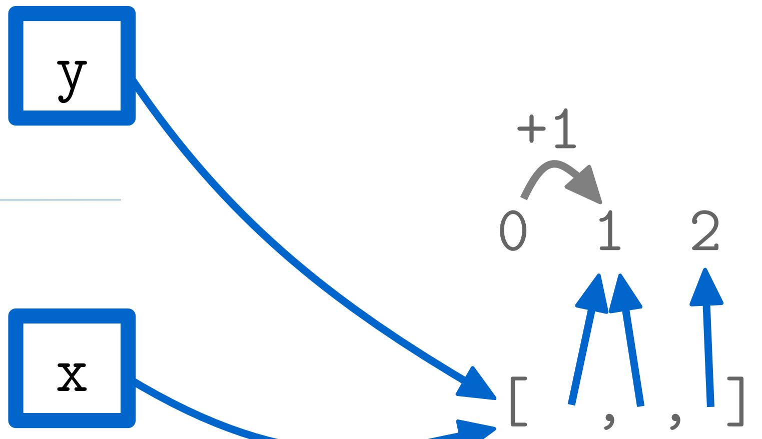


Pass by Assignment

“object space”

local scope

```python
def incr_first(y):
    y[0] = y[0] + 1
```



global scope

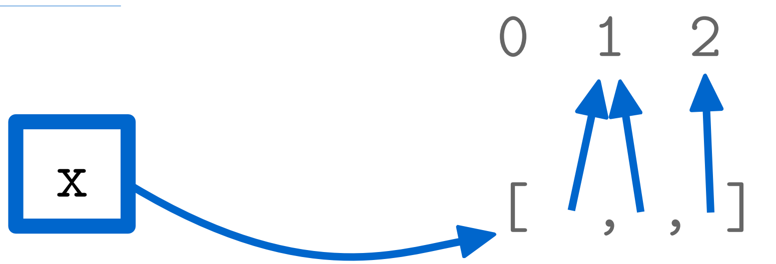
```python
x = [0, 1, 2]
incr_first(x)
```

Pass by Assignment

"object space"

global scope

```
x = [0, 1, 2]
incr_first(x)
```



Pass by Assignment

“object space”

global scope

```
x = [0, 1, 2]
incr_first(x)
print(x) # [1, 1, 2]
```

Scopes

```
x = 1
if True:
    x = x + 1
    y = x
print(x) # 2
print(y) # 2

a = [x for x in range(10)]
print(x) # 2
```

create scopes:

- class
- function
- list- & dict-comprehension

do **not** create scopes:

- for, while
- if, else, elif
- with
- try... except

Copy

- copy module (important!) – lets you copy objects **cleanly**

```
import copy

x = [0, 1, 2]
y = copy.copy(x) # shallow copy

x[0] = 3
print(x) # [3, 1, 2]
print(y) # [0, 1, 2]

a = [0, 1, [2, [3, 4]]]
b = copy.deepcopy(a) # this should be your default for copying
a[2][1][0] = 6
print(a) # [0, 1, [2, [6, 4]]]
print(b) # [0, 1, [2, [3, 4]]]
```

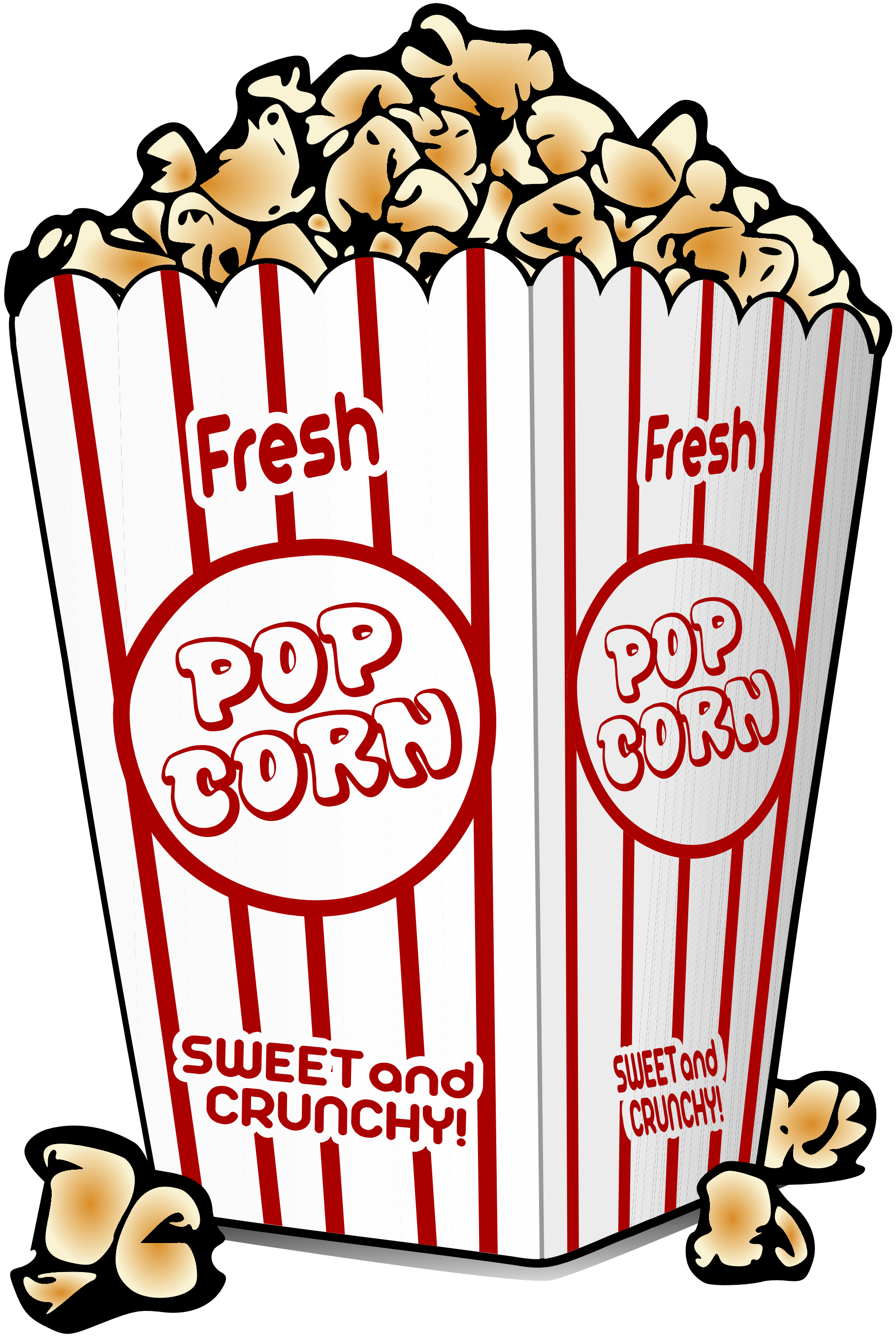


image source: openclipart.org

File I/O

Python

```python
x = 1
# 'r': read, 'w': write
# 'rw': read/write
f = open('filename', 'w')
f.write(x)
# need to manually close
f.close()

# better solution:
# context manager
with open('filename', 'w') as f:
    f.write(x)
```

C++

```cpp
#include <fstream>
int x = 1;
std::ofstream os("filename");
os << x;
// need to manually close
os.close();
```

Teaser:
You can easily define your own Context Manager, be it for resource control, timing, ...

Useful Built-in Functions

```
range # Creates a range of integers

range(5) # range(0, 5)
list(range(3, 5)) # [3, 4]
list(range(2, 7, 2)) # [2, 4, 6]

zip # Fuses two (or more) Iterables into tuples

list(zip([1, 2, 3], ['a', 'b', 'c', 'd'])) # [(1, 'a'), (2, 'b'), (3, 'c')]

enumerate # Takes an Iterable and creates tuples of (index, value)

enumerate(['a', 'b', 'c']) # <enumerate object at 0x7f18e66e4d20>
list(enumerate(['a', 'b', 'c'])) # [(0, 'a'), (1, 'b'), (2, 'c')]
enumerate(L) == zip(range(len(L)), L)

sorted

sorted([2, 1, 0, 4]) # [0, 1, 2, 4]
sorted([-1, 2, 0, 4], key=lambda x: x**2) # [0, -1, 2, 4]
```

full list: https://docs.python.org/3/library/functions.html

Useful Libraries

- built-in libraries
  - sys: system-specific parameters and functions
    - sys.argv: Command-line arguments
    - sys.path: List of paths where modules are searched
  - os: OS interface
  - shutil: high-level file operations
  - math, random, copy
  - pip: package manager
  - virtualenv: virtual environment
- external libraries
  - numpy: “BLAS” + math package, essential for any numerical computation
  - scipy: “LAPACK” + numerical algorithms (solvers etc.)
  - matplotlib: plotting library

Import

```
# import by module name
import math
math.sqrt(4)

# import by module name, with renaming
import numpy as np
a = np.array([1, 2, 3])

# import specific object from a module -> global namespace
from sys import argv
argv # ['filename.py']

# import all objects from a module -> global namespace
from os import *
linesep # '\n'
```

String Formatting

C++

```cpp
#import <iostream>
std::pair<int, int> x = {1, 2}
std::cout << "(" << x.first << " / " << x.second
          << ")" << std::endl; // (1 / 2)
```

Python

```python
x = (1, 2)
print('({} / {})'.format(x[0], x[1])) # (1 / 2)
print('({} / {})'.format(*x))         # (1 / 2)

# can change order, specify precision
print('({1} / {0:.1f})'.format(*x))   # (2 / 1.0)

# can access members directly
print('({0[0]} / {0[1]})'.format(x))  # (2 / 1)
```

Format mini-language:
https://docs.python.org/3.5/library/string.html#format-specification-mini-language

Classes

Python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

C++

```cpp
struct Point{
    Point(double x, double y):
        x(x), y(y) {}

    double x, y;
};
```

Classes

Python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

__init__ is the Constructor

C++

```cpp
struct Point{
    Point(double x, double y):
        x(x), y(y) {}

    double x, y;
};
```

Classes

Python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

self: reference to the object itself

C++

```cpp
struct Point{
    Point(double x, double y):
        x(x), y(y) {}

    double x, y;
};
```

Classes

Python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


a = Point(1, 2)
a.x # 1
```

C++

```cpp
struct Point{
    Point(double x, double y):
        x(x), y(y) {}

    double x, y;
};

Point a(1, 2);
a.x // 1
```

Classes

Python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y



a = Point(1, 2)
a.x # 1

# add members dynamically
a.z = 3 # works!
```

C++

```cpp
struct Point{
    Point(double x, double y):
        x(x), y(y) {}

    double x, y;
};

Point a(1, 2);
a.x // 1

// cannot add new members
a.z = 3 // Error!
```

Methods

Python

```python
class Point:
    ...
    def scale(self, sx=1, sy=1):
        self.x *= sx
        self.y *= sy



a = Point(1, 2)
a.scale(sy=1.2)
# a.x == 1, a.y == 2.4
```

C++

```cpp
class Point{
    ...
    void scale(
        double sx=1, double sy=1
    ):
        x *= sx;
        y *= sy;
};

Point a(1, 2);
a.scale(1, 1.2);
// a.x == 1., a.y == 2.4
```

Magic Methods

- Marked by leading & trailing double underscores
- Correspond to C++ operator overloading

```
class Point:
    ...
    def __add__(self, p):
        return Point(self.x + p.x, self.y + p.y)

    def __str__(self):
        return '({0.x}, {0.y})'.format(self)

a = Point(1, 2)
b = Point(1, -1)

print(a + b) # (2, 1)
```

Magic Methods – Overview

called by	magic method	C++ equivalent
	`__init__`	`constructor`
`del`	`__del__`	`destructor`
`+`	`__add__`	`operator+`
`*`	`__mul__`	`operator*`
`/`	`__truediv__`	`(operator/)`
`//`	`__floordiv__`	`(operator/)`
`str`	`__str__`	`operator std::string`
`len`	`__len__`	
		`operator=`

there are many more magic methods:
https://docs.python.org/3/reference/datamodel.html

Inheritance

Python

```python
class A:
    def __init__(self):
        self.x = 1

    def print_A(self):
        print(self.x)

class B(A):
    def __init__(self):
        super().__init__() # self.x == 1
        self.x = 2

    def print_B(self):
        print(self.x)
```

C++

```cpp
#include <iostream>

struct A {
    void print_A() const{
        std::cout << x << std::endl;
    }
    const double x = 1;
};

struct B: public A {
    void print_B() const{
        std::cout << x << std::endl;
    }
    const double x = 2;
};
```

Inheritance

Python

```python
class A:
    def __init__(self):
        self.x = 1

    def print_A(self):
        print(self.x)

class B(A):
    def __init__(self):
        super().__init__() # self.x == 1
        self.x = 2

    def print_B(self):
        print(self.x)
```

C++

```cpp
#include <iostream>

struct A {
    void print_A() const{
        std::cout << x << std::endl;
    }
    const double x = 1;
};

struct B: public A {
    void print_B() const{
        std::cout << x << std::endl;
    }
    const double x = 2;
};
```

Inheritance

Python

```python
class A:
    def __init__(self):
        self.x = 1

    def print_A(self):
        print(self.x)

class B(A):
    def __init__(self):
        super().__init__() # self.x == 1
        self.x = 2

    def print_B(self):
        print(self.x)
```

super(): reference to the base class

Base class constructor must be called explicitly

C++

```cpp
#include <iostream>

struct A {
    void print_A() const{
        std::cout << x << std::endl;
    }
    const double x = 1;
};

struct B: public A {
    void print_B() const{
        std::cout << x << std::endl;
    }
    const double x = 2;
};
```

Inheritance

Python

```python
class A:
    def __init__(self):
        self.x = 1

    def print_A(self):
        print(self.x)

class B(A):
    def __init__(self):
        super().__init__() # self.x == 1
        self.x = 2

    def print_B(self):
        print(self.x)


b = B()
b.print_A() # 2
b.print_B() # 2
```

C++

```cpp
#include <iostream>

struct A {
    void print_A() const{
        std::cout << x << std::endl;
    }
    const double x = 1;
};

struct B: public A {
    void print_B() const{
        std::cout << x << std::endl;
    }
    const double x = 2;
};

B b;
b.print_A(); // 1
b.print_B(); // 2
```

What about private variables?

- There are no private variables

  → no private / protected inheritance

- Convention:

  leading underscore → treated as private

- No const keyword

Python vs. C++

- transparent
- modifiable

- guts hidden
- rigid (const)

The open sea can be dangerous

image source: Wikipedia

How to avoid Sharks

- Conventions are very important in Python
- The Zen of Python: Design Guideline

```
import this
```

- Naming conventions (e.g. self, args, kwargs)
- PEP8(Python Enhancement Proposal): Style Guide
- PEP257: Docstring Conventions
- pep8: Tool for checking consistency with PEP8
- pylint: General-purpose code checker (scores your code)

Example – File I/O

It's easier to ask forgiveness than it is to get permission

Python

```python
try:
    with open('filename', 'w') as f:
        f.write(1)
except IOError:
    print("I'm sorry, I shouldn't"
          " have let you do that")
```

C++

```cpp
#include <fstream>
#include <iostream>
std::ofstream os("filename");
if(os.is_open()){
    os << 1;
    os.close();
} else {
    std::cout << "I'm sorry, I can't"
              << " let you do that"
              << std::endl;
    }
}
```

Example – File I/O

It's easier to ask forgiveness than it is to get permission

Python

```python
try:
    with open('filename', 'w') as f:
        f.write(1)
except IOError:
    print("I'm sorry, I shouldn't"
          " have let you do that")
```

try... except
is a valid control flow statement in Python.

C++

```cpp
#include <fstream>
#include <iostream>
std::ofstream os("filename");
if(os.is_open()){
    os << 1;
    os.close();
} else {
    std::cout << "I'm sorry, I can't"
              << " let you do that"
              << std::endl;
    }
}
```

Example – File I/O

It's easier to ask forgiveness than it is to get permission

Python

```python
try:
    with open('filename', 'w') as f:
        f.write(1)
except IOError:
    print("I'm sorry, I shouldn't"
          " have let you do that")
```

Errors should never pass silently: Capture only the type of error you are interested in.

C++

```cpp
#include <fstream>
#include <iostream>
std::ofstream os("filename");
if(os.is_open()){
    os << 1;
    os.close();
} else {
    std::cout << "I'm sorry, I can't"
              << " let you do that"
              << std::endl;
    }
}
```

Templates

- Everything is a “template” in Python

- **Duck typing:**

  “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” - J.W. Riley

- Explicit type of an object is unimportant

  → work with Concepts

- Typedefs are possible via name binding

```
P = Point
type(P(1, 1)) # <class 'Point'>
```

Duck Typing – Example

Python

```python
import math

def d2(dx, dy, dz):
    return math.sqrt(
        dx**2 + dy**2 + dz**2
    )
```

C++

```cpp
#include <cmath>

template< typename T1,
          typename T2,
          typename T3
        >
decltype(auto)
d2(T1 dx, T2 dy, T3 dz) {
    return std::sqrt(
        dx * dx + dy * dy
        + dz * dz
    );
}
```

Checking Concepts

- `isinstance, issubclass`

  Check if an object (class) models the concept of a given class.

```
import numpy as np
issubclass(np.int64, int) # True
```

- `collections.abc`

  Abstract Base Classes (Container, Iterable, Hashable,...)

```
from collections.abc import Iterable, Mapping
isinstance([1, 2, 3], Iterable) # True
isinstance([1, 2, 3], Mapping)  # False
```

- User-defined Abstract Base Classes are also possible

Introspection

- No private variables

  → anything can be inspected

- `__dict__` contains an object's attributes (by default, can be changed)

```
p = Point(1, 2)
p.__dict__ # {'x': 1, 'y': 2}
Point.__dict__ # holds methods, and much more
```

- `inspect` module: Advanced introspection tools

Serialization: pickle

- Introspection
  - code can analyze your classes
  - knows how to serialize them

```python
import pickle

p = Point(1, 2)
with open('filename', 'w') as f:
    pickle.dump(p, f)

with open('filename', 'r') as f:
    q = pickle.load(f) # q == p
```

Object

- **object** is a base for all classes

  “everything is an object”

  ```
  isinstance( , object) == True
  ```

- classes, functions etc. are just instances of another class

  ```python
  class A:
      pass

  A.__str__ = lambda self: 'A instance'
  print(A())       # A instance
  print(type(A)) # <class 'type'>
  ```

- **monkey patching:** changing existing objects to extend / modify their functionality (tread with caution!)
- **metaclasses:** Change how a class is created by modifying the class' class (advanced)

Decorators

```
def f():
    print('World')

def dec(f):
    def inner(*args, **kwargs):
        print('Hello', end=' ')
        return f(*args, **kwargs)
    return inner

g = dec(f)
g() # Hello World
```

Decorator: A function that takes a function and modifies it in a specific way

Decorators - @ Notation

```
def dec(f):
    def inner():
        print('Hello', end=' ')
        f()
    return inner

@dec # f = dec(f)
def f():
    print('World')

f() # Hello World
```

@ Notation: The decorator is applied to the function
Its return value binds to the function name

Built-in Decorators

```
class Point:
    ...
    @staticmethod       # doesn't exist in C++
    def dimensions():
        return 2

    @classmethod        # C++ static method
    def name(cls):
        return cls.__name__

a = Point(1, 2)
print(a.dimensions()) # 2
print(a.name())       # Point
```

staticmethod: no implicit first argument
classmethod: class as implicit first argument
method: self as implicit first argument

Built-in Decorators

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        print('getting x')
        return self.x

    def set_x(self, val):
        print('setting x')
        self.x = val

a = Point(1, 2)
a.get_x()                    # prints 'reading x'
a.set_x(3)                   # prints 'setting x'
```

Built-in Decorators

```python
class Point:
    def __init__(self, x, y):  # 'hide' x
        self._x = x
        self.y = y

    @property                # proxy_type in C++
    def x(self):
        print('getting x')
        return self._x
    @x.setter
    def x(self, val):
        print('setting x')
        self._x = val

a = Point(1, 2)
a.x                          # prints 'getting x'
a.x = 3                      # prints 'setting x'
```

Python 2 vs. Python 3

- Python 3 slightly breaks backwards-compatibility (for good reasons) changes include:
  - division: '/' now does **real division** instead of int division
  - **print**: used to be a keyword, and is now a function
  - not all classes derived from **object** in Python 2
- `from __future__ import division, print_function`
  as first statement in the file → new-style division and print
- Cheat sheet: http://python-future.org/compatible_idioms.html
- It's easy to write Python 2 and Python 3 **compatible** code!

Conclusion

+ Development speed: insane
  - also makes Python an excellent prototyping language

+ Flexibility

+ Portability
  - deployment is trivial (pip, PyPI)
  - virtualenv

- Safety
  - no encapsulation
  - needs good testing (unittest module)

- Speed
  - no premature optimization!
  - 80:20 rule – Python is perfect as “glue code”
  - next week: boost::python

Questions?