PY410 / 505
Computational Physics 1

Salvatore Rappoccio
• Code in CompPhys/ReviewPython
• type: “python” in the command line

• then:

• >>> import antigravity
• For all of the pain in C++, python fixes it
• But! Python is slower than cold molasses in winter. (technical term. Conversion is 3 molasseses / snail’s pace)

• Quick and dirty: Python wins
• Optimal performance: C++ wins

• BUT! This is not either/or, it’s both/and!
  – Best case is to have your “human” handling with python and your hardcore computer code in C++
  – Then call the C++ code from python
    • This is what scipy, numpy do, etc
python

• **C++:**
  – Fast
  – Compiled
  – Statically typed
    • int i = 0;
  – Access to pointers
  – Whitespace irrelevant

• **python:**
  – Slower
  – Interpreted
  – Dynamically typed:
    • i = 0
  – No pointers
  – Whitespace matters
You can learn Python in minutes once you learned another language

https://docs.python.org/3/tutorial/
• This is easy.

• Hello, word:
  >>> print “Hello, world”

• Add 2 + 3:
  >>> 2 + 3

• Make a vector (now called a “list”, and uses [ ]):
  >>> a = [0, 1, 2]

• Make a tuple:
  >>> t = ['smith', 'alice', 55.0, 'score1']
python: strings

• C++:
  – `std::string s = “apple”`;
  – `char c = ‘a’`;
  – `std::string b = “banana”`;
  – `std::string c = a + b`;

• python:
  • `s = ‘apple’`
  • `s = “apple”`
  • `s = “a”`
  • `s = ‘a’`
  • `s = ‘She said “ugh”!’`
  • `c = s + ‘blabla’`
python: lists

C++

```cpp
std::vector<int> a = {0,1,2,3};

for( auto i : a ) {
    std::cout << i << std::endl;
}

a.push_back( 4 );
a[2] = 5;
```

```python
a = [0,1,2]
print a

for i in a:
    print i

a.append(4)
a[2] = 5
```

```cpp
std::vector<int> b = {5,6,7};
for ( auto j : b ) {
    a.push_back( j );
}
```

```python
b = [5,6,7]
a = a + b
```
std::map<string, int> a;
a["mine"] = 0;
a["yours"] = 1;

for (auto i : a) {
    std::cout << i.first << "  "
               << i.second << std::endl;
}

a = {
    "mine": 0, "yours": 1
}
or

a = {}
a["mine"] = 0
a["yours"] = 0

or:
keys = ["mine", "yours"]
vals = [0, 1]
a = dict(zip(keys, vals))
if (i < 20) {
    std::cout << "Nuke" << std::endl;
} else if (i < 40) {
    std::cout << "Tweet" << std::endl;
} else {
    std::cout << "Fake news." << std::endl;
}

if i < 20:
    print 'Nuke'
elif i < 40:
    print 'Tweet'
else:
    print 'Fake news'
```cpp
for ( int i = 0; i < 4; ++i ) {
    std::cout << i << std::endl;
}
```

```python
for i in [0,1,2,3] :
    print i
```

```python
for i in range(4) :
    print i
```
int i = 0;
while ( i < 10 ){
    ++i;
}

i = 0
while i < 10:
    i += 1
def fib(n=0):
    a,b = 0,1
    while a < n:
        a,b = b,a+b
    return a
**python: functions**

**C++**

```c++
double f(double x=0.0, double y=0.0){
    return x + y;
}
```

**Python**

```python
def f(x=0.0,y=0.0) :
    return x + y
```

**C++**: (crickets)

Python can use **KEYWORD arguments** in any order you want!

- `f(y=1.0)`
- `f(x=0.0,y=1.0)`
- `f(0.0,1.0)`
python: Modules

Fibo.h:
```c
int fib(int n=0){
    int a = 0, b = 1;
    while(a < n){
        a = b;
        b = a + b;
    }
    return a;
}
```

main.cc:
```c
#include "Fibo.h"
...
int i = fib(10);
```

Fibo.py:
```python
def fib(n=0):
    a, b = 0, 1
    while a < n:
        a, b = b, a + b
    return a
```

main.py:
```python
from fibo import fib
...
```
int main (int argc, char ** argv){
    int n = argc;
    char * val = argv[n-1];
}

import sys
n = len(sys.argv)
val = sys.argv[n-1]
```cpp
#include <iostream>

int i;
std::cout << "enter val: ";
std::cin >> i;
std::cout << "i = " << i << std::endl;

#include <fstream>

std::ifstream f("input.txt");
std::string s;
f.getline( f, s );
```

```python
i = input("enter val: ")
print 'i = ', i
or
print 'i = ' + str(i)

f = open("input.txt")
# Read entire file:
value = f.read()
# Read one line:
value = f.readline()
```
class A{
  public:
    A(int i) { f_ = i; }
    int f() const { return f_; }
  protected:
    int f_;}

class A:
    f_ = 0
    def __init__(self, i):
        self.f_ = i

    def f(self):
        return self.f_

"self" is like the "this" pointer in C++, but ALWAYS needs to be in the class method argument list
python: Inheritance

class B : public A {
    (bla bla bla)
};

class B( A ):
    (bla bla bla)
• Division changed between python2 and python3

<table>
<thead>
<tr>
<th>python2</th>
<th>python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt;&gt; 2 / 7</td>
<td>&gt;&gt;&gt;&gt; 2 / 7</td>
</tr>
<tr>
<td>0</td>
<td>0.2857142857142857</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; 2 // 7</td>
<td>&gt;&gt;&gt;&gt; 2 // 7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; 2. / 7.</td>
<td>&gt;&gt;&gt;&gt; 2. / 7.</td>
</tr>
<tr>
<td>0.2857142857142857</td>
<td>0.2857142857142857</td>
</tr>
</tbody>
</table>

The “true division” operator (/) and the integer division operator (//) are now separate in python3

C++ has strong typing so it “knows” how to do this.
### python: operator overloading

#### C++

```cpp
class A{
    A operator+(A const & right);
    A operator-(A const & right);
    A operator*(A const & right);
    A operator/((A const & right);
};
```

#### python

```python
class B( A):
    def __add__(self, r):
        return self.r + r
    def __sub__(self, r):
        return self.r - r
    def __mul__(self, r):
        return self.r * r
    def __floordiv__(self, r):
        return self.r / r
```

[https://docs.python.org/2/reference/datamodel.html#emulating-numeric-types](https://docs.python.org/2/reference/datamodel.html#emulating-numeric-types)
python: Example

• Go “ReviewPython”
python

• Major difference to be careful of is that python DOES NOT pass “mutable” objects by value, it passes by reference
  – Called “passed by object reference”
  – “Mutable” or “Immutable” here refers effectively to the location in memory of the object
    • Ints, floats, tuples: immutable
    • Lists, maps: mutable

  – So passing an int to a function won’t modify it, but passing a list to a function may modify it

• Example: “mutabledemo.py”