

Why scripting

Lecture 17

Python: an accelerated introduction

Most parts of the course is centered around the C++ programming language.

We will switch to another, called **Python**. But we will first see why such languages are needed.

References:

- Web site of Python: <http://www.python.org/>
- We use version 1.5.2: <http://www.python.org/doc/1.5.2p2/>.
Read the tutorial there.

PMOOP(0396A)

PMOOP(0396A)-17.1

Common characteristics

- Very **short code** compared with traditional language code.
- More **dangerous feature**. E.g., variables declaration are usually unnecessary or optional, data type is know only at run-time.
- **Deallocate memory automatically** to reduce development time.
- **No separated compilation** needed. Programs are either compiled when executed, or interpreted, again to reduce development time.
- Highly **portable**. Same program runs on many types of computers.
- Strong focus on **gluing together existing utilities**.
- **Only built-in operations are fast**.
- **Highly modular**, enter new areas by making new modules.
- Built-in types and operations can be added by C/C++ **plugins**.

PMOOP(0396A)-17.2

PMOOP(0396A)-17.3

Characteristics of Python

- **Interpreted**. No compilation at all, so not very fast. But it is easy to peep inside the internals, e.g., see what variables are defined.
- Have very **clean syntax**, unlike sh or Perl. Good for algorithm studies.
- **Reasonable number of standard data types**, so reasonably fast.
- **Object-oriented, very modular**. Possible to develop large application on it (as long as speed is not a concern).
- Have a **large set of pre-defined modules** for basically every commonly needed application.
- Easily extendible using **plugins**, combine the benefits of C and C++.
- Python itself is a library. We **can embed** a Python interpreter in C/C++ programs by linking the library.
So our C++ program can call the Python interpreter when needed.

PMOOP(0396A)-17.4

Running Python interactively

Python is interpreted, so we can run python and type into it directly:

```
> python
Python 1.5.2 (#0, Apr 10 2001, 10:03:44) ...
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> 1+2*3**4+6%5 # comments looks like this
164
>>> (2+3j)*(1-1j) # complex numbers
(5+1j)
>>> a='Hello, World' # variable assignment
>>> a
'Hello, World'
>>> dir() # what is defined?
['_builtins_', '__doc__', '__name__', 'a']
>>> del(a) # remove the variable.
>>> dir()
['_builtins_', '__doc__', '__name__']
```

PMOOP(0396A)-17.5

Python scripts

In the above, we see the result of each Python statement. We call it the interactive mode of Python.

However, once the work become larger and we might make mistakes, we want programs (or **scripts**). Hello-world in Python:

```
print "Hello, World!"
```

If this is in a file hello.py, then **python hello.py** runs it. We normally use the Unix scripting mechanism to avoid typing "python" on execution:

```
#!/usr/bin/python
print "Hello, World!"
```

The first line tells the command to run this script, so when this program is made executable by "**chmod +x hello.py**", then we can run it by just **./hello.py**". The system then run **"/usr/bin/python J/hello.py"** for us.

PMOOP(0396A)-17.6

Simple data types in Python

All Python data are "**objects**", with a type associated:

- **Null**: like C++ void, means "nothing". The only instance is **None**.
- **Numeric types**: integer (e.g., 5), long integer (e.g., 1234567890L), floating point (e.g., 5.2), complex (e.g., 2+3j).
 - Supports +, -, *, /, %, **, &, ^, |, <<, >>, ~.
 - Complex numbers supports a.real, a.imag and a.conjugate().
 - Conversion done by int(a), long(a), float(a), complex(a,b) and abs(a).
- **Strings**: Written like 'abc' or "def". Convertable to and from ASCII: ord('a') is 97, chr(97) is 'a'. Throw exception on ord('abc').

They are **immutable**: you cannot change the value of the object.

E.g., you cannot insert a character into a string, you can only create a new one that contains one more character than the existing string.

PMOOP(0396A)-17.7

Sizes of numbers...

- An integer is similar to the C++ int: it can hold an **fix-sized integer**.
- You can find the **largest integer** by **sys.maxint** after **import sys**. (We will soon see what is import.)
- On the other hand, **long** type can hold integers of any size. It is implemented by an array to hold big integers.
- Floating point numbers are like C++ **double** (and are implemented by it). There is no constants about floating point numbers, though.
- Complex numbers are also implemented by C++ double.
- The modules **math** and **cmath** are used to support floating point and complex numbers respectively.
 - Providing functions like taking sine or cosine of a number or complex number, giving constants for pi and e, etc.

PMOOP(0396A)-17.8

Sequence data types in Python

There are two types of sequences (other than string):

- **List**. Mutable. Written like [], [2], [1, 'a'] and [[1, 'a'], 2].
- **Tuple**. Immutable. Written like (), (2,), (1, 'a') and ((1, 'a'), 2).

You can retrieve information from any sequence in the same way:

- length: e.g., len([1,3]) gives 2.
- element: e.g., 'abc'[1] gives 'b', 'abcde'[-1] = 'e'.
- slice: e.g., (1, 2, 3, 4, 5)[2:4] gives (3,4).

Lists supports additional operations that **modify** the list. They are all member functions, including **append**, **count**, **extend**, **index**, **insert**, **pop**, **remove**, **reverse** and **sort** (type dir([]) to list them).

Guessing what they do by trial and error is probably easier than reading docs.

PMOOP(0396A)-17.9

Map types in Python

Python directly support one type of maps, called dictionaries. It is mutable, and **maps any immutable type to any type**. I.e., you can map a tuple or a string, you can't map a list or map.

- Dictionaries are **written** like {'1':'abc', (2,):[4,5,6]}, mapping the integer 1 to the string 'abc', and the tuple (2,) to the list [4, 5, 6].
- **Lookup** is done using [], like C++ maps: {'1':'abc', (2,):[4,5,6]} [(2,)] gives you [4, 5, 6].
- The pairs are ordered in the dictionary is some specific ways, but it is best to treat them as unordered.
- Support the following operations:

```
>>> dir({})
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'update', 'values']
```

Again try to guess what they are by some experiments.

PMOOP(0396A)-17.10

File types in Python

Opening a file in Python results in a **file object** that can be read, written and manipulated otherwise. E.g., to print a file and quote it with '> ':

```
f = open("test.py", 'r')          # f is a file object
while (1):
    line = f.readline()
    if (line == ""):
        break
    line = line[:-1]              # Remove '\n'
    print '> ' + line
```

Other interesting operations:

- write(str), write str to the file.
- readlines(), read all lines in the file to form a list of string.
- read(n), read up to n bytes from the file and return the resulting string.

PMOOP(0396A)-17.11

Other types

There are some more standard types that are of interest...

- **Function** types, built-in and user-defined.
- **Class** types.

The idea is that functions and classes can get new names by assignments, e.g., we can say `myopen = open` and now `myopen` can be called like `open`.

With plugins, **new types** can be defined. So the **number of types is actually unlimited**. Some interesting non-standard ones:

- Array, provided by the array module, store simple values in a compact way.
- dbm, gdbm and bsddb database maps, provided by the dbm, gdbm and bsddb modules, allow databases to be updated using maps syntax.

PMOOP(0396A)-17.12

Defining functions

Python functions are written like this:

```
def sqr(n):  
    "Fine the square of n"  
    return n * n
```

Note that:

- **No type declaration**. Type is determined at **run-time**.
- A string at the beginning is the **documentation string**, which serve no purpose other than documentation.
- There is a colon right before the content of the **def** statement.
- There is no braces. Indentation is **mandatory**, and must be **consistent**, within a code block.

The last two bullets are true for all control structures.

PMOOP(0396A)-17.13

If-else

The if-else statements look like this:

```
def check_temperature(temp):  
    if temp > 25:  
        print "Hot"  
        print "Turn on air-conditioner"  
    else:  
        print "Not hot"
```

Python expects a statement after a colon. If nothing need to be done, put a **pass** statement there.

Right

```
if temp > 25:  
    pass  
else:  
    print "Not hot"
```

Wrong

```
if temp > 25:  
else:  
    print "Not hot"
```

PMOOP(0396A)-17.14

Conditions

- So we can have **conditions**, which are used in if-else statements (and other control constructs).
- Conditions like `temp > 25` is really **integers**: it evaluates to 1 if true, 0 if false. All C++ comparison operators work.
- The following values are **treated as false**:
 - Numeric values which is interpreted as 0, i.e., 0, 0.0 and 0+0j.
 - The empty string `""`.
 - The special value **None**.
- You can combine different values by using the keywords **and**, **or**, and **not**. The corresponding C++ operators are not defined.
- You can compare between **different types**, and the result is implementation dependent, but consistent.

PMOOP(0396A)-17.15

Repetitive structures

There are two repetitive structures in Python: while and for.

```
def print_factor(n):  
    # range(2,n) is the list  
    # [2, 3, ..., n-1]  
    for i in range(2, n):  
        if (n % i == 0):  
            print n, "=",  
            print i, "*", n/i  
            break  
    else:  
        print n, "is a prime."
```

```
def print_factors(n):  
    i = 2  
    while i < n:  
        if (n % i == 0):  
            print n, "=",  
            print i, "*", n/i  
            break  
        # ++ is supported in 2.0  
        i = i + 1  
    else:  
        print n, "is a prime."
```

The strange indentation of the **else** part is not a typo. It means "execute this if **break** is not executed".

Also, note that **for** is driven by a list instead of 3 expressions (in C++).

PMOOP(0396A)-17.16

Organization of Python: modules

The whole Python library is organized as a set of **modules**. Many commonly used operations have its module. For example,

- `string`: contain many string operations
- `os`: use the OS services, e.g., running a command.
- `ftplib`: use ftp from within your script.

To use something in a module, you have to **import** it. (Except that built-in module needs no import.) Examples:

```
import os  
os.system('clear')
```

```
from os import system  
system('clear')
```

You can even say "from os import *" to import everything from os. But that is not really recommended, since there is good chances of a name clash. But some packages require you to do so anyway, as we will see soon.

PMOOP(0396A)-17.17

Variables and scopes

Variables in Python are implemented with Python dictionaries. It maps a string to an object. So when Python runs

```
a = [ 2, 3, 4]
```

it actually map the string 'a' to the list [2, 3, 4].

There is a **dictionary per function, object, module and class.**

- `dir()` **return the list of keys** of the local dictionary. To get the whole dictionary, use `locals()`. (So `dir()` is similar to `locals().keys()`)
- Assignments and `def`'s are **normally done to the local dictionary**, unless a "global" statement is executed before (e.g., **global *gvar***). Then the variable will be read and written in global dictionary instead.
- Dictionary **uses** (getting variable, calling functions) is tried first in local, then current module, then built-in module dictionary.

PMOOP(0396A)-17.18

Assignments in Python

Assignment (=) in Python is somewhat magic. Assignment is a statement, not an operator: `print a = 5` is not valid. But...

- `a = b = c = [2, 3]` is valid.
- **Assignments do not copy objects.** They just manipulate the dictionary. In the above, a, b and c points to the same list.
- You can assign to tuples and lists of variables, e.g.

```
def print_fibs(n):
    (a, b) = (0, 1)
    for i in range(0, n):
        print a, b
        (a, b) = (b, a+b)
```

- You can assign to list slices, e.g. after `a = [1,2,3,4,5]`; `a[2:4] = [10]`, a becomes [1, 2, 10, 5].

PMOOP(0396A)-17.19

Classes in Python

How to define a class? Like this:

```
class MyClass:
    "An example class"
    s = 12345 # Behave like a static class data member
    def __init__(self, name): # a constructor is named __init__
        self.name = name # self is like this, but must be written
    def f(self): # Behave like a member function.
        return 'Hello, ' + self.name # remember to write self!
```

This shows how to create an object:

```
x = MyClass('Chan Tai Man')
```

... and how to call a member function...

```
print x.f()
```

As normal, we don't have a destructor.

PMOOP(0396A)-17.20

Inheritance and encapsulation

- What about encapsulation? How to make things private??
- The answer is... it **can't!** So there is no way in the language to enforce the contract between the user and writer of the class.
- How about **inheritance**? A language cannot be called OO without it!
- It has, so it can be called OO. :p E.g.,

```
class MyExtended(MyClass):
    s = 23456
    def __init__(self, name, title):
        MyClass.__init__(self, name) # Do chaining this way
        self.title = title
    def f(self): # All member functions are "virtual"
        return 'Hello, ' + self.title + ' ' + self.name
```

```
obj = MyExtended('Isaac To', 'Dr.')
```

PMOOP(0396A)-17.21

Exception handling

How about exceptions? The model is very similar with C++. We have an exception object to throw. E.g.,

```
import string
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror): # catching one type of exception
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError: # catching type, no associated value
    print "Could not convert data to an integer."
except: # Catch-all exception
    print "Unexpected error!"
    raise # Rethrow
finally: # do this to free resources
    close(f)
```

PMOOP(0396A)-17.22

Exercise

Write a Python script that reads the `/etc/passwd` file, and reports the shell of each user. E.g.,

```
root (uid 0) has /bin/bash as his/her shell.
daemon (uid 1) has /bin/sh as his/her shell.
...
```

Instructions:

- Try to do it using the string module. Read the module docs about it. Hint: use the `split()` function.
- Repeat the experiment, this time using the `pwd` module. Again, read the docs.

PMOOP(0396A)-17.23