

Why scripting

Lecture 20

Python: an accelerated introduction

So far we have been dealing with three languages: C++ for producing machine code, Makefile for automatic building, and Lisp in assignment.

We will introduce one more, 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)-20.1

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 known 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)-20.2

PMOOP(0396A)-20.3

Other uses

Some more uses of scripting languages:

- **System administration**: need small but cross-platform programs.
- **Experiments**: test if algorithms or systems are empirically good.
- **Extensions**: allow users to modify behaviour of your program flexibly.
- **GUI interfaces to commands**: usually good even if slow.
- **Interface to databases**: database operations are the bottleneck, flexibility is more important than speed for the interface.
- **Specific and well known areas**: everything is built-in. E.g., printing, typesetting, dialup ISP, etc.
- **Code in web clients**: real platform independence is needed.
- **Slow alternative to C++**: if speed is not needed at all.

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 library** 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. In this way, our own C++ program can call a Python script on some events.

PMOOP(0396A)-20.4

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)-20.5

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 ./hello.py**" for us.

PMOOP(0396A)-20.6

PMOOP(0396A)-20.7

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 is probably easier than reading docs.

PMOOP(0396A)-20.8

PMOOP(0396A)-20.9

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.

PMOOP(0396A)-20.10

PMOOP(0396A)-20.11

Simple data types in Python

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

- **Null**: similar to the C++ void, means "nothing". E.g., **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.)

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 {}: {'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.

Other types

There are some more standard types that are of interest, but we will only deal with them a bit later.

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

The idea is that functions and classes can get new names, just like fset in our Lisp assignment.

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)-20.10

PMOOP(0396A)-20.11

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 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)-20.12

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)-20.13

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, "x", 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, "x", n/i
            break
        i = i + 1      # No ++!
    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.

PMOOP(0396A)-20.14

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.

PMOOP(0396A)-20.15

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 (to be introduced later).

- `dir()` or `local()` returns the local dictionary. So you can see the variable a defined in the first example.
- Assignments and `def`'s are normally done to the local dictionary, unless a "global" statement is executed before.
- Dictionary uses (getting variable, calling functions) is done in the local, then current module, then built-in module dictionary.

PMOOP(0396A)-20.16

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)-20.17

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.