**安装paramiko模块**

paramiko安装要求：

（1）platforms supported: POSIX (Linux, Solaris, BSD, etc.); MacOS X; Windows

（2）python 2.3: python 2.2 is also supported, but not recommended. 目前系统中的python版本都支持

（3）pycrypto 2.1+下载最新的pycypto版本

一、下载软件包

wget http://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.3.tar.gz

wget http://www.lag.net/paramiko/download/paramiko-1.7.7.1.tar.gz

二、安装

tar -zxvf pycrypto-2.3.tar.gz

cd pycrypto-2.3

python setup.py install注释：(当初编译时报错：error: command 'gcc' failed with exit status 1；因为缺少python-dev的软件包，所以apt-get install python-dev；重新执行python sedup.py install通过）

tar -zxvf paramiko-1.7.7.1.tar.gz

cd paramiko-1.7.7.1

python setup.py install如果执行上述命令没有问题，则可以测试安装在当前目录下执行，python test.py

[root@localhost ~]# python  
Python 2.4.3 (#1, May 5 2011, 16:39:10)  
[GCC 4.1.2 20080704 (Red Hat 4.1.2-50)] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import paramiko  
>>>

# When installing paramiko, [from Crypto import Random -> ImportError: cannot import name Random](http://stackoverflow.com/questions/7210873/from-crypto-import-random-importerror-cannot-import-name-random)

The installed Crypto is in */usr/local/lib/python2.6/dist-packages/Crypto/*. But it seems that *Crypto* has been installed in */usr/lib/python2.6/dist-packages/Crypto/*. Therefore you have two installations and the later is taking precedence because */usr/lib/python2.6/dist-packages/* appears first in sys.path.

Fix it by renaming *Crypto* to something else like *Crypto\_bak* so you can rollback if something goes wrong. (Exploit # *sudo pip install pycrypto* to install the new version of *pycrypto*).

**安装pycassa模块**

pycassa是一个Cassandra的python客户端，见:<http://pycassa.github.com/pycassa/index.html>

When you failed to upload a file to the server, probably it’s because the attribute of the folder in the server is not enough. In this situation, login the ftp server and modify the attribute of this folder:

*# telnet 172.17.18.161 (Login this ftp server)*

*Trying 172.17.18.161…*

*Connected to 172.17.18.161.*

*…*

*login: lab (Need username and password to login this ftp server)*

*Password:*

*[lab@armor ~]$ su - (Switch to root to change the attribute)*

*Password:*

*[root@armor ~]# ls -al (This folder currently belongs to root)*

*drwxr-xr-x 4 root root 4096 Oct 12 2011 To\_SDC*

*[root@armor ~]# chmod -R lab.lab To\_SDC/ (Change the owner of this folder)*

*[root@armor ~]# ls -al (Now the owner of this folder is lab)*

*drwxr-xr-x 4 lab lab 4096 Oct 12 2011 To\_SDC*

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

There are two ways to format the string:

*>>> target\_ip = “172.17.21.104”*

*>>> port\_no = 8443*

*>>> cmd = “ping %s:%s –c1 –W1” % (target\_ip, port\_no)*

*>>> cmd = “ping {0}:{1} –c1 –W1”.format(get\_ip, port\_no)*

Two ways to set a =dictionary object:

*cfg = dict(username = ‘super’)*

*cfg = {‘username’: “admin”}*

If you want to initialize the member variables of its base class in the derived class, here is the example:

*class BmpController:*

*def \_\_init\_\_(self, cfg):*

*self.xcfg = dict(username = ‘BmpController’)*

*self.xcfg.update(cfg)*

*class BmpControllerEx(BmpController):*

*def \_\_init\_\_(self, cfg):*

*super(BmpControllerEx, self).\_\_init\_\_( cfg)*

*# You can’t initialize the member variables in the following way since this expression means   
# BmpControler::xcfg and BmpControlerEx::xcfg are different variables*

*# self.xcfg = dict(usernameEx = ‘BmpControllerEx’)*

*self.xcfg.update(cfg)*

How to convert data from *dict* to *list*:

*>>> x = {‘a’: 1, ‘b’: 2}*

*>>> y = list(x.values())*

You can use a text editor to create a py file and add *#!/usr/bin/python* in the first line. Like a shell script file, you have to modify the attribute of the file before running it: *chomod +x hello.py*.

In Linux script, the symbol *#!* Represents this file is executed by an interperator.

The data transformes from one type into another: *datatype(item)*. For example:

*>>> int(“45”)*

*>>> str(192)*

Python doesn’t use variable to store data but object reference. The expression is: *objectReference = value*. For example: *x = “blue”*. It’s not required to declare beforehand. When Python runs the first statement, a string object stored *“blue”* is created, and generate an object reference called *x* pointing to this object.

Unlike other languages which define *“=”* as an assignment operator to set the value to a variable. In Python, the *“=”* operator bind an object reference to a specified object in memory. If this object reference has already existed, Python just re-bind to the object on the right side of *“=”*. If this object ference doesn’t exist, the *=* operator allocates the memory first. If no object references refer to certain an object, this object will be released by garbage collector.

Python utilizes dynamic typing which means the object reference can refer to different types of object anytime.

There are some collection types in Python and the most common ones are tuple and list. Tuple is immutable so that can’t be modified after creation. List is mutable and can be inserted or removed the elements from it. The collection types ultilize comma to separate the elements.

To create empty tuple and list, exploit *()* and *[]*. Tuple and list store the object references, not objects.

Tuple, list and string can be sized, so that we can use *len()* to calculate the length of the data.

*>>> len((“one”,))*

*>>> len([3, 5, 1, 2, “pause”, 5])  
>>> len(“automatically”)*

The index in Python starts from 0.

*is* operator is a binary operator. If the object references on the left and right are equal, it returns *True*. *is* operator is usually used for checking if the object is *None*.

*>>> a is not None, b is None*

To compare the value of an object, use *==* operator.

*>>> a = “many paths”  
 >>> b = “many paths”*

*>>> a is b*

*False*

*>>> a == b*

*True*

*in* operator can be used to detect the membership of list, tuple and string. This operator ultilizes linear search to look for the membership and becomes ineffient when the data is huge.

*>>> p = (4, “frog”, 9, -33, 9, 2)*

*>>> 2 in p*

*True*

*>>> “dog” not in p*

*True*

Python contains three logical operators: *and*, *or* and *not*.

*if* statement:

*if boolean\_expression1:*

*suite1*

*elif boolean\_expression2:*

*suite2*

*…*

*elif boolean\_expressionN:*

*suiteN*

*else:*

*else\_suite1*

*if*, *else* and *elif* must be followed by colons. Python exploits indentation to represent block structure.

*while* loop

while boolean\_expression:

suite

*for…in* loop

*for variable in iterable:*

*suite*

*while* and *for* loops support *break* and *continue* and an optional *else* statement. *variable* points to each object in *iterable*. *iterable* is any iterable object including string (character by character), list and tuple.

*countries = [“Denmark”, “Finland”, “Norway”, “Sweden”]*

*for country in countries:*

*print(country)*

*for letter in “ABCDEFGHIJKLMNOPQRSTUVWXYZ”:*

*if letter in “AEIOU”:  
 print(letter, “is a vowel”)*

exception handler:

*try:*

*try\_suite*

*except exception1 as variable1:*

*exception\_suite1*

*…*

*except exception as variable:*

*exception\_suiteN*

*as variable* is optional.

The divide operator generates the floating point value rather integer. Use *int()* to transform it into the integer. After operation is complete, an object is created to store the result and then the target object re-binds to this result object.

The right side of *+=* operator must be an iterable object. For example:

*>>> seeds = [“sesame”, “sunflower”]*

*>>> seeds += [5]*

*>>> seeds*

*[“sesame”, “sunflower”, 5]*

To define a function:

*def functionName (arguments):*

*suite*

*arguments* is optional. If there is more than one argument, they are separated by colon. Every function in Python must have the return value. The default is *None*. Exploit *return value* to return the value. The return value can be one value or a set of values. Users can ignore the return value.

*def get\_int(msg):*

*while True:*

*try:*

*i = int(input(msg))*

*return i;*

*except ValueError as err:*

*print(err)*

The python modules are the *.py* files including Python codes. Before using the module, it’s required to import it. Exploit *import* statement and the file name of .py files. After importing the module, we can use any functions, classes and variables. For example:

*import sys*

*print(sys.argv)*

*sys* module provides the argv variable: it’s a list and the first item is so called the name of running program, the second and the rest of items are command argument. Generally, the way to use the function of module is:

*moduleName.functionName(arguments)*.

Python provides two builtin integer types: *int* and *bool*. These two types are immutable. Python provides two builtin Boolean objects: *True* and *False*.

Python exploits immutable *str* type to represent the string. The type preserves the string as the Unicode string sequence. To use the quotes in a string, it’s required to exploit delimiting quotes to disable some special character. For example: *a = “Signle ‘quotes’ are find; \”doubles\” must be escaped”*.

To avoid delimiting quotes, add “*r*(raw)” as the prefix so that all symbols are viewed as normal character.

*Phone1 = re.compile(“\\d”);*

*Phone2 = re.compile(r”\d”);*

The rule *[0-9A-Fa-f:]{17}* can be used to find mac address in regular expression.

*\w* is equivalent to the set *[a-zA-Z0-9\_]*, *\S* represents the set *[^ \t\n\r\f\v]*.

To check the string *‘syslog-wpa-price-tb / c0:8a:de:1f:24:48*’ through the regular expression:

*re.search(‘([\w\S]+)\s/\s([0-9A-Fa-f:]{17})’, ‘syslog-wpa-price-tb / c0:8a:de:1f:24:48’)*

The index of string in Python starts from 0 and defines -1 representing the last character of the string. If the index is out of range, the exception of “*IndexError*” is thrown. There are three ways to access the elements:

*seq[start]*, *seq[start:end]*, *seq[start:end:step]*. The default start and end index is 0 and *len(seq)* if *step* is positive. The default start and end index is -1 and –*len(seq)* if negative. The default value of *step* is 1.

*str.join()* can concatenate numerous strings.

*>>> treatises = [“Arithmetica”, “Conics”, “Elements”]*

*>>> “ ”.join(treatises)*

*Arithmetica Conics Elements*

*>>> “”.join(treatises)*

*ArithmeticaConicsElements*

\* operator provides the function of string duplication.

*>>> s = “=” \* 5*

*>>> print(s)*

*=====*

There are two ways to search for the substring in a string:

1. *str.index()*: Returns the index of the substring and throws *ValueError* exception when not found.

*line = “This is a test”*

*line\_found = “test”*

*try:*

*i = line.index(line, line\_found)*

*except ValueError:*

*return None*

1. *str.find()*: Returns the index of the substring and -1 if not found. Caution: *list* doesn’t have this function.

*line = “This is a test”*

*line\_found = “test”*

*i = line.index(line, line\_found)*

*if i == -1: return None*

*str.lstrip()*, *str.rstrip()* and *str.strip()* are used to remove the character (or space) on the left, right and both sides, for example:

*>>> s = “\t no parking “*

*>>> s.lstrip(), s.rstrip(), s.strip()*

*(‘no parking ’, ‘\t no parking’, ‘no parking’)*

*>>> “<[unbracketed]>”.strip(“[](){}<>”)*

*‘unbracketed’*

*str.replace()* replaces the sub string. This function requires two arguments. If the first arugment, which is a sub-string is found in the string, it is replaced by the second argument, which is another sub-string.

*str.split()* split a string into a string list recursively to the smallest unit:

*>>> record = “Leo Tolstoy\*1828-8-28\*1910-11-20”*

*>>> fields = record.split(“\*”)*

*>>> fields*

*[‘Leo Tolstoy’, ‘1828-8-28’, ‘1910-11-20’]*

*>>> born = fields[1].split(“-”)*

*[‘1828’, ‘8’, ‘28’]*

There are a few useful tips to convert a Python list (or any other iterable such as a tuple) to a string for display.

First, if it is a list of strings, you may simply use join this way:

*>>> mylist = ['spam', 'ham', 'eggs']*

*>>> print ', '.join(mylist)*

*spam, ham, eggs*

Using the same method, you might also do this:

*>>> print '\n'.join(mylist)*

*spam*

*ham*

*eggs*

This simple method does not work if the list contains non-string objects, such as integers. If you just want to obtain a comma-separated string, you may use this shortcut:

*>>> list\_of\_ints = [80, 443, 8080, 8081]*

*>>> print str(list\_of\_ints).strip('[]')*

*80, 443, 8080, 8081*

Or this one, if your objects contain square brackets:

*>>> print str(list\_of\_ints)[1:-1]*

*80, 443, 8080, 8081*

Finally, you may use *map()* to convert each item in the list to a string, and then join them:

*>>> print ', '.join(map(str, list\_of\_ints))*

*80, 443, 8080, 8081*

*>>> print '\n'.join(map(str, list\_of\_ints))*

*80*

*443*

*8080*

*808*

List is mutable. Use *[]* to create an empty list.

*\** can also be used as starred argument. For example:

*>>> product(2, 3, 5)*

*30*

*>>> L = [2, 3, 5]*

*>>> product(\*L)*

*30*

*>>> product(2, \*L[1:])*

*30*

*range()* is a builtin function which generates integer iterator. If the integer argument is *n*, *range()* returns an iterator which are 0, 1,… ,n-1.

If the reference points to the data which is not the collection, *del* operator breaks the connection between the reference and the data and the reference is deleted too. For example:

*>>> x = 8143*

*>>> x*

*8143*

*>>> del x*

*>>> x*

*Traceback (most recent call last):*

*…*

*NameError: name ‘x’ is not defined*

If an object reference is deleted, Python checks are there any refenece to this object. If not, the data is sent to garbage collector. If the reference points to collections, only the elements which the reference point to are being deleted.

To add/remove the item into/from the tail of the list, use *list.append()/list.pop()*.

If a list *x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*, we can use *x[1::2] = [0] \* len(x[1::2])* so that the content of the list is *x = [1, 0, 3, 0, 5, 0, 7, 0, 9, 0]*.

To insert an elment into list, use the command as: *woods[2:2] = [“Pine”]*. To replace a specified element in a list: *woods[2] = “Pine”*. To replace a set of element in a list, use the command as: *woods[2:5] = [“Pine”, “Banana”, “Apple”]*. To delete the elements inside the list, use the command: *woods[2:4] = []* or *del woods[2:4]*.

list compression contains an expression and a conditional loop. The loop is used to generate the elements of the list and the condition filters the unnecessary data. The two general forms to exress list comprehension:

*[expression* ***for*** *item* ***in*** *iterable]*

*[expression* ***for*** *item* ***in*** *iterable* ***if*** *condition]*

The second format is equal to:

*temp = []*

*for item in iterable:*

*if condition:*

*temp.append(expression)*

Here are three functions using sets to remove duplicate entries from a list, find the intersection of two lists, and find the union of two lists. sets were required in Python 2.4 or later. Also, the items in the list must be hashable and order of the lists is not preserved.

***def*** *unique(a):*

*""" return the list with duplicate elements removed """*

***return*** *list(set(a))*

***def*** *intersect(a, b):*

*""" return the intersection of two lists """*

***return*** *list(set(a) & set(b))*

***def*** *union(a, b):*

*""" return the union of two lists """*

***return*** *list(set(a) | set(b))*

***if*** *\_\_name\_\_ == "\_\_main\_\_":*

*a = [0,1,2,0,1,2,3,4,5,6,7,8,9]*

*b = [5,6,7,8,9,10,11,12,13,14]*

***print*** *unique(a)*

***print*** *intersect(a, b)*

***print*** *union(a, b)  
Results:*

*[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*

*[8, 9, 5, 6, 7]*

*[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]*

*class Point():*

*x = 0.0*

*y = 0.0*

*def \_\_init\_\_(self, x, y):*

*self.x = x*

*self.y = y*

*def ToString(self):*

*return "{X:" + str(self.x) + ",Y:" + str(self.y) + "}"*

*class Circle(Point):*

*radius = 0.0*

*def \_\_init\_\_(self, x, y, radius):*

*super(Circle, self).\_\_init\_\_(x,y)*

*self.radius = radius*

*def ToString(self):*

*return super().ToString() + ",{RADIUS=" + str(self.radius) + "}"*

Note that there is another place that calls *super()* incorrectly, inside of Circle's ToString() method:

*return super().ToString() + ",{RADIUS=" + str(self.radius) + "}"*

This is valid code on Python 3, but on Python 2 *super()* requires arguments, rewrite this as the following:

*return super(Circle, self).ToString() + ",{RADIUS=" + str(self.radius) + "}"*

**lambda: anonymous function**

*my\_profile = lambda user\_name, age: “%s is %d years old” % (username, age)*

*print my\_profile(“eddie”, 30)*

There are two main series in Python: 2.7.x and 3.2.x. These two versions are not compatible.

The variables in Python are non-typed, and just a reference to a certain memory address. “*=*” is the operator which assigns the variable to a memory address.

The keyword ‘*switch*’ is not supported in Python, exploit *if~elif~else* instead.

**Python function**

*def calc\_method(a, b=1):*

*print a \* b*

*def args\_method(\*args):*

*print args*

*def kwargs\_method(\*\*kwargs);*

*print kwargs*

*calc\_method(4) => 4*

*calc\_method(4, 10) => 40*

*calc\_method(b=10, a=2) => 20*

*args\_method(1, 2, 3) => (1, 2, 3)*

*kwargs\_method(name=’eddie’, age=20) => {‘age’: 20, ‘name’: ‘eddie’}*

*print 10 / 3 => 3*

*print 10.0 /3 => 3.333333333….*

*>>> my\_str = ‘abcdefg’*

*>>> print my\_str[1:4]*

*‘bcd’*

my\_string = ‘高見龍’

my\_string\_unicode = u‘高見龍’

len(my\_string) -> 9

len(my\_string\_unicode) -> 3

‘u’ represents Unicode.

**The differenece between *append()* and *extend()***

*my\_list = [1, 2, 3, 4, 5]*

*my\_sub\_list = [‘a’, ‘b’]*

*my\_list.extend(my\_sub\_list) -> [1, 2, 3, 4, 5, ‘a’, ‘b’]*

*my\_list.append(my\_sub\_list) -> [1, 2, 3, 4, 5, [‘a’, ‘b’]]*

**List Comprehension**

*>>> print [i for i in range(10)]*

*[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*

*>>> print [i for i in range(10) if i % 2 == 0]*

*[0, 2, 4, 6, 8]*

**Dictionary**

*>>> my\_dict = {‘one’: 1}*

*>>> my\_dict[‘two’] = 2*

*>>> print my\_dict*

*{‘one’: 1, ‘two’: 2}*

*>>> my\_dict.items() -> Divide key/value into tuple and return a list*

*[(‘two’, 2), (‘one’, 1)]*

*>>> my\_dict.keys()*

*[‘two’, ‘one’]*

*>>> my\_dict.values()*

*[2, 1]*

*>>> my\_dict.get(‘two’)*

*2*

*>>> my\_dict.get(‘three’, 3) -> Return a default value if this does NOT exist*

*3*

*>>> my\_dict.clear()* -> Clean all the contents in the dictionary

*>>> print my\_dict*

*{}*

*>>> del my\_dict* -> Remove this variable from the memory

**Tuple**

*my\_tuple = (1,)* -> The ‘comma’ is a must if the element is only one.

**try~except~finally**

*try:*

*…*

*except NameError, e: -> If exception occurs!*

*…*

*else: -> If no exception is raised*

*…*

*finally: -> This block must be executed no matter exception occurs*

*…*

**import**

*import decimal*

*decimal.Decimal(0.1) + decimal.Decimal(0.1) - decimal.Decimal(0.2)*

*form decimal import Decimal*

*Decimal(0.1) + Decimal(0.1) - Decimal(0.2)*

CheckConfigurationBase.py and SetupConfiguration.py are in the same folder.

*In CheckConfigurationBase.py:  
class CheckConfigurationBase(Testlet):*

*…*

*class CheckConfigurationCtrlInf(object):  
 …*

*In SetupConfiguration.py:*

*from CheckConfigurationBase import \**

*class SetupConfiguration (CheckConfigurationBase):*

*…*

*class SetupConfigurationCtrlInf(CheckConfigurationCtrlInf):  
 …*

# Caution: If you don’t import the correct module, you may get an exception:

# *TypeError:* [*Error when calling the metaclass bases*](http://stackoverflow.com/questions/2231427/error-when-calling-the-metaclass-bases-function-argument-1-must-be-code-not) *…*

# *Module.\_\_init\_\_() takes at most 2 arguments (3 given)*

# Python fails to recognize *CheckConfigurationBase* and *CheckConfigurationCtrlInf as* classes.

super() can be used only in the new-style classes, which means the root class needs to inherit from the 'object' class. For example, the top class needs to be like this:

*class SomeClass(object):*

*def \_\_init\_\_(self):*

*....*

not

*class SomeClass():*

*def \_\_init\_\_(self):*

*....*

The important point is that **the correct test** for "is this a new-style **instance** (i.e. object)?" is:

*>>> class OldStyle: pass*

*>>> instance = OldStyle()*

*>>> issubclass(instance.\_\_class\_\_, object)*

*False*

and not (as in the question):

*>>> isinstance(instance, object)*

*True*

For **classes**, the correct "is this a new-style class" test is:

*>>> issubclass(OldStyle, object) # OldStyle is not a new-style class*

*False*

*>>> issubclass(int, object) # int is a new-style class*

*True*

The **crucial point** is that with old-style classes, the *class* of an instance and its *type* are distinct. An old-style class just creates objects of type instance, and a new-style class creates objects whose type is the class itself.

**Module**

If you create an independent folder to define a module, the *\_\_init\_\_.py* file is a MUST! Otherwise, the module can not be imported successfully.

**Class**

*class MyClass():*

*def \_\_init\_\_(self): -> The first function when this instance is created, but not related to memory allocation.*

*So it is not like the constructor in C++.*

*…*

*def \_\_del\_\_(self): -> The last function when this instance is deleted, but not related to memory allocation.*

*So it is not like the destructor in C++.*

*…*

*def \_\_str\_\_(self): -> The return string is shown when printing this instance. If this function is un-defined, the address of this object is shown.*

*return “This is MyClass class”*

**Inheritance**

*class MyDerivedClass(MyClass):*

*…*

*easy\_install* is the dedicated setup tool for Python. To install this setup tool:

*# sudo sh setuptools-0.6c10-py2.6.egg*

To install a Python package called South:

*# easy\_install South*

**Threading in Python**

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

* ***threading.activeCount()*:** Returns the number of thread objects that are active.
* ***threading.currentThread()*:** Returns the number of thread objects in the caller's thread control.
* ***threading.enumerate()*:** Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

* ***run()*:** The entry point for a thread.
* ***start()*:** Starts a thread by calling the run method.
* ***join([time])*:** Waits for threads to terminate.
* ***isAlive()*:** Checks whether a thread is still executing.
* ***getName()*:** Returns the name of a thread.
* ***setName()*:** Sets the name of a thread.

To implement a new thread using the threading module, you have to do the following:

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which will in turn call *run()* method.

The threading module provided a simple-to-implement locking mechanism to allow you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object would be used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread will wait to acquire the lock.

If *blocking* is set to 0, the thread will return immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If *blocking* is set to 1, the thread will block and wait for the lock to be released.

The *release()* method of the the new lock object would be used to release the lock when it is no longer required.

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the *Queue*:

* ***get()*:** The *get()* removes and returns an item from the queue.
* ***put()*:** The *put()* adds item to a queue.
* ***qsize()* :** The *qsize()* returns the number of items that are currently in the queue.
* ***empty()*:** The *empty()* returns True if queue is empty; otherwise, False.
* ***full()*:** the *full()* returns True if queue is full; otherwise, False.

*#!/usr/bin/python*

*import Queue*

*import threading*

*import time*

*exitFlag = 0*

*class myThread (threading.Thread):*

*def \_\_init\_\_(self, threadID, name, q):*

*threading.Thread.\_\_init\_\_(self)*

*self.threadID = threadID*

*self.name = name*

*self.q = q*

*def run(self):*

*print "Starting " + self.name*

*process\_data(self.name, self.q)*

*print "Exiting " + self.name*

*def process\_data(threadName, q):*

*while not exitFlag:*

*queueLock.acquire()*

*if not workQueue.empty():*

*data = q.get()*

*queueLock.release()*

*print "%s processing %s" % (threadName, data)*

*else:*

*queueLock.release()*

*time.sleep(1)*

*threadList = ["Thread-1", "Thread-2", "Thread-3"]*

*nameList = ["One", "Two", "Three", "Four", "Five"]*

*queueLock = threading.Lock()*

*workQueue = Queue.Queue(10)*

*threads = []*

*threadID = 1*

*# Create new threads*

*for tName in threadList:*

*thread = myThread(threadID, tName, workQueue)*

*thread.start()*

*threads.append(thread)*

*threadID += 1*

*# Fill the queue*

*queueLock.acquire()*

*for word in nameList:*

*workQueue.put(word)*

*queueLock.release()*

*# Wait for queue to empty*

*while not workQueue.empty():*

*pass*

*# Notify threads it's time to exit*

*exitFlag = 1*

*# Wait for all threads to complete*

*for t in threads:*

*t.join()*

*print "Exiting Main Thread"*

When the above code is executed, it produces the following result:

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

To check if a file exists in the disk, you have the os.path.exists function:

*import os.path*

*os.path.exists(file\_path)*

I need to remove multiple items from a list according to their indexes. However, I can't use pop because it shifts the indexes. Is there a way to remove multiple items simultaneously?

Method 1: List comprehension:

*L = ['a', 'b', 'c', 'd']*

*L = [c for c in L if c not in ['a', 'c']]*

Method 2: If you really don't want to create a copy, go backwards:

*for i in reversed(range(len(L))):*

*if L[i] in ['a', 'c']:*

*L.pop(i) # del L[i] is more efficient*

Going backwards is space efficient but time inefficient; most of the time a list comprehension or generator (L = (...) instead of L = [...]) is best.

Method 3: An order-preserving, in-place filter that differs in speed from a list comprehension by a constant.

*write\_i = 0*

*for read\_i in range(len(L)):*

*L[write\_i] = L[read\_i]*

*if L[read\_i] not in ['a', 'c']:*

*write\_i += 1*

*del L[write\_i:]*

*print L*

*# output: ['b', 'd']*

<http://www.ibm.com/developerworks/cn/linux/l-pypt/part1/index.html>

**Python與設計模式**

Python和C++都支持對像模型運算符重載、多重繼承。但Python忽略數據隱藏。與C++和Java不同，Python沒public、protected和private等關鍵字，可直接修改對象屬性，語法簡潔。Python動態類型綁定，可在運行時動態改變。

**Python對像模型**

Python中的類也是一種用戶自定義的數據類型，其基本的語法格式是：

|  |
| --- |
| class <name>(superclass, ...): # 定義類  data = value # 共享的類變量  def method(self, ...): # 類中的方法  self.member = value # 實例的數據 |

類定義從關鍵字class開始，包含整個縮進代碼塊，類中定義的方法和屬性構成了類的名字空間(name space)。類通常有多個方法，以關鍵字def開頭，第一個參數是self，Python中的變量self相當於C++中的關鍵字this，其作用是傳遞一個對象的引用。Python中的類屬性位於類的名字空間中，可被所有的類實例所共享，這一點同C++和Java相同。除了自定義的類屬性外，Python每個類其實都具有一些特殊的類屬性，由Python的對像模型所提供的：

|  |  |
| --- | --- |
| \_\_dict\_\_ | 類名字空間的字典變量 |
| \_\_doc\_\_ | 類的文檔說明字符串 |
| \_\_name\_\_ | 類的名稱 |
| \_\_module\_\_ | 類的模塊名 |
| \_\_bases\_\_ | 該類所有父類組成的元組 |

定義類的目的是為了創建它的實例，從面向對象的角度看，類是對數據及其相關操作的封裝，而類實例則是對現實生活中某個實體的抽象。假設定義瞭如下一個類：

|  |
| --- |
| *class School:*  *def \_\_init\_\_(self, name):*  *self.name = name*  *self.students = []*  *def addStudent(self, student):*  *self.students.append(student)* |

要創建School類的一個實例，可以執行下面的語句：

|  |
| --- |
| *bit = School("Beijing Institute of Technology")* |

在C++和Java中創建類實例時，與類具有相同名稱的構造函數被調用，而在Python中創建一個類的實例時，將調用名為\_\_init\_\_的特殊方法。不像C++和Java使用前須先聲明，Python可動態創建。在針對接口的設計模式中，通常需知道對象所屬的類，以便能調用不同的實現方法，這些在C++和Java這些強類型語言的對像模型中不難實現，但對Python來講每個變量事實上都沒有固定的類型。Python的\_\_builtin\_\_模塊提供了兩個非常實用的內建函數：isinstance()和issubclass()。isinstance()用於測試一個對像是否是某個類的實例，如果是的話則返回1，否則返回0。其基本的語法格式是：

|  |
| --- |
| *isinstance (instance\_object, class\_object)* |

例如：

|  |
| --- |
| *>>> class Test:*  *pass*  *>>> inst = Test()*  *>>> isinstance(inst, Test)*  *1* |

而函數*issubclass()*則用於測試一個類是否是另一個類的子類，如果是的話則返回1，否則返回0。其基本的語法格式是：

|  |
| --- |
| *issubclass(classobj1, classobj2)* |

例如：

|  |
| --- |
| *>>> class TestA:*  *pass*  *>>> class TestB(TestA):*  *pass*  *>>> issubclass(TestA, TestB)*  *0*  *>>> issubclass(TestB, TestA)*  *1* |

和類一樣，Python中的每個類實例也具有一些特殊的屬性，它們都是由Python的對像模型所提供的。表2列出了這些屬性：

|  |  |
| --- | --- |
| \_\_dict\_\_ | 實例名字空間的字典變量 |
| \_\_class\_\_ | 生成該實例的類 |

表2 特殊的實例屬性

**繼承**

在面向對象的程序設計中，繼承（Inheritance）允許子類從父類那裡獲得屬性和方法，同時子類可以添加或者重載其父類中的任何方法。在Python中定義繼承類的語法格式是：

|  |
| --- |
| *class <name>(superclass, superclass, ...)*  *suit* |

例如，對於下面這個類：

|  |
| --- |
| *class Employee:*  *def \_\_init\_\_(self, name, salary = 0):*  *self.name = name*  *self.salary = salary*  *def raisesalary(self, percent):*  *self.salary = self.salary \* (1 + percent)*  *def work(self):*  *print self.name, "writes computer code"* |

可以為其定義如下的子類：

|  |
| --- |
| *class Designer(Employee):*  *def \_\_init\_\_(self, name):*  *super(Designer, self).\_\_init\_\_(self, name, 5000)*  *def work(self):*  *print self.name, "writes design document"* |

C++和Java子類的構造函數會自動調用父類的構造函數，但Python中卻不是這樣，必須在子類中顯示調用父類的構造函數。

C++允許多重繼承，而Java則是通過接口(Interface)來間接實現多重繼承。Python類似C++，允許多重繼承，例如：

|  |
| --- |
| *class A:*  *pass*  *class B(A):*  *pass*  *class C:*  *pass*  *class D(B, C):*  *pass* |

**多態**

嚴格說來，C++和Java強類型語言對像模型中的多態概念並不適用於Python，因為Python沒有類型聲明機制。但由於Python是一種動態類型語言，允許將任意值賦給任何一個變量，如果對多態的概念擴充，將其理解為具有能同時處理多種數據類型的函數或方法，那麼Python對像模型實際上也支持經過弱化後的多態。Python直到代碼運行時才去決定變量所屬的類型，這特性稱為運行時綁定(runtime binding)。Python解析器內部雖然也對變量進行類型分配，但只有在真正使用它們時才隱式地分配類型。

**可見性**

Python對像模型對可見性的處理與C++和Java完全不同。在C++和Java中，如果屬性或者方法被聲明為private，那就意味著它們只能在類中被訪問，而如果被聲明為protected，則只有該類或者其子類中的代碼能夠訪問這些屬性和方法。但在Python對像模型中，所有屬性和方法都是public的，也就是說數據沒有做相應的保護。

Python 1.5引入名字壓縮(name mangling)的概念，使類中的屬性得以局部化。在進行定義類時，如果一個屬性的名稱是以兩個下劃線開始，同時又不是以下劃線結束的，那麼它在編譯時將自動地被改寫為類名加上屬性名。例如：

|  |
| --- |
| *class Greeting:*  *\_\_data = "Hello World!"*  *def \_\_init\_\_(self, str):*  *Greeting.\_\_data = str*  *>>> g = Greeting("Hello Gary!")*  *>>> dir (g)*  *['\_Greeting\_\_data', '\_\_doc\_\_', '\_\_init\_\_', '\_\_module\_\_']* |

Greeting類的\_\_data屬性變成了\_Greeting\_\_data。雖然仍然無法阻止外界對它的訪問，但的確使得訪問變得不再那麼直接了，從而在一定程序上保護了類中的數據不被外界破壞。