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):*

*BmpController.\_\_init\_\_(self, 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 at any time.

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 which 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. In this situation, 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]*.

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

*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 return -1 when not found.

*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 be used as sequence unpacking operator and separated in any kind of iterable types. For example:

*>>> first, \*rest = [9, 2, -4, 8, 7]*

*>>> first, rest*

*(9, [2, -4, 8, 7])*

*>>> first, \*mid, last = “C P A G W”.split(“ ”)*

*>>> first, mid, last*

*(“C”, [“P”, “A”, “G”], “W”)*

*\** 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(Point,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\_metho(\*\*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)*

**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**

**Introduction**

If two or more threads were to attempt to manipulate the same object at the same time, problems would inevitably pop up. The global interpreter lock fixes this. Only one thread can perform an action at any given time. Python automatically switches between threads when it is needed. Its Thread class may be subclassed to create a thread or threads. The run method should contain the code you wish to be executed when the thread is executed:

*import threading*

*class MyThread ( threading.Thread ):*

*def run ( self ):*

*print 'Insert some thread stuff here.'*Executing the thread is to create an instance of our thread class and then call its start method:

*MyThread().start()*

Let's create a group of threads:

*for x in xrange ( 20 ):  
   MyThread().start()*

Servers often use threads to handle multiple clients. To send the client's data to the thread, we will need to override the Thread class's *\_\_init\_\_* method to accept parameters. The server will now send the thread on its way and then wait for new clients. Each thread will send a pickled object to the appropriate client and then print no more than ten strings received from the client. (A pickled object is basically an object that has been reduced to a few characters). Each thread will connect to the server and execute the code above:

*import pickle  
import socket  
import threading*

*# Here's our thread:  
class ConnectionThread ( threading.Thread ):*

*def run ( self ):*

*# Connect to the server:  
      client = socket.socket ( socket.AF\_INET, socket.SOCK\_STREAM )  
      client.connect ( ( 'localhost', 2727 ) )*

*# Retrieve and unpickle the list object:  
      print pickle.loads ( client.recv ( 1024 ) )*

*# Send some messages:  
      for x in xrange ( 10 ):  
         client.send ( 'Hey. ' + str ( x ) + '\n' )*

*# Close the connection  
      client.close()*

*# Let's spawn a few threads:  
for x in xrange ( 5 ):  
   ConnectionThread().start()*

It's important to remember that threads don't start up instantly. Creating too many of them can slow down your application. It takes time to spawn and later kill threads. Threads can also eat up valuable system resources in large applications. This problem is easily solved by creating a set number of threads (a thread pool) and assigning them new tasks, basically recycling them. Connections would be accepted and then pushed to a thread as soon as it finished with the previous client. Python's Queue module does this for us. Client information is stored in a Queue object, where threads can pull them out when needed.