

Quick and Dirty Python

13

INFORMATION IN THIS CHAPTER

- Introduction to programming
- Python intro
- Python components
- Examples and Samples
- Creating tools and transforms

INTRODUCTION

After covering many interesting topics related to utilizing different automated tools, in this chapter we will be learning to create some. Sometimes there is a need to perform some specific task for which we are not able to find any tools which suits the requirements, this is when we have some basic programming knowledge so that we can quickly create some code to perform the desired operation. This chapter will touch upon the basics of Python programming language. We will understand why and how to use Python, what are the basic entities and then we will move on to create some simple but useful code snippets. It is advised to have some programming knowledge before moving on with this chapter as we will be covering the basic essentials related to the language and jump straight into the code. Though the examples used would be simple yet having some programming experience would be helpful.

Anyone who has some interest in computer science is familiar with the concept of programming. In simple terms it is the process of creating a program to solve a problem. To create this program we require to have a language using which we can write instructions for computer to understand and perform the task. The simple objective of a computer program is to automate a series of instructions so that they need not to be provided one by one manually.

PROGRAMMING VERSUS SCRIPTING

The language we are going to be discussing in this chapter is Python, which is commonly termed as a scripting language, so before moving further let's understand what that means. Usually the code written in a programming language is compiled to machine code using a program called compiler to make it executable. For example, code written in C++ language is compiled to create an exe file which can be executed in a Windows platform.

There is another program called as an interpreter which allows running a language code without being compiled. So if the execution environment for a piece of code is an interpreter it is a script. Usually Python is executed in such environment and hence is commonly called a scripting language. This does not mean that a scripting language cannot be compiled, it simply is not usual. All scripting languages are programming languages.

INTRODUCTION TO PYTHON

Python is a high-level programming language created by Guido Van Rossum, which emphasizes on the readability of code. Python is very fast and allows solving problem with minimum amount of code and hence is very popular among people who need to create quick scripts on the go, such as pentesters. There are various versions of Python but we will be focusing on the 2.7 version in this chapter. Though the latest version as of now is 3.4, yet most of the Python tools and libraries available online are based on the 2.7 version and the 3.x version is not backward compatible and hence we will not be using it. There are some changes in 3.x version but once we get comfortable with 2.7 it won't require much effort to move to it, if required.

The main agenda behind this chapter is not to create a course on Python that would require a separate book in itself. Here we will be covering the basics quickly and then move on to creating small and useful scripts for general requirements. The aim is to understand Python, write quick snippets, customize existing tools, and create own tools as per requirements. This chapter strives to introduce the possibilities of creating efficient programs in a limited period of time, provide the means to achieve it, and then further extend it as required.

There are other alternatives to Python available, mainly Ruby and Perl. Perl is one of the oldest scripting languages and Ruby is being widely used for web development (Ruby on Rails) yet Python is one of the easiest and simplest language when it comes to rapidly creating something with efficiency. Python is also being used for web development (Django).

INSTALLATION

Installing Python in Windows is pretty straight forward, simply download the 2.7 version from <https://www.python.org/downloads/> and go forward with the installer. Linux and other similar environments mostly come preinstalled with Python.

Though, it is not mandatory yet highly recommended to install Setuptools and Pip for easy installation and management of Python packages. Details related to Setuptools and Pip can be found at <https://pypi.python.org/pypi/setuptools> and <https://pypi.python.org/pypi/pip> respectively.

MODES

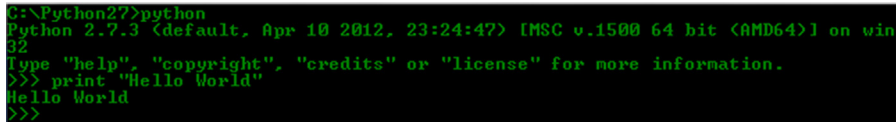
We can run Python basically in two ways, one is to directly interact with the interpreter, where we provide the commands through direct interaction and see the output of it (if any) and other one is through scripts, where we write the code into a file, save it as

filename.py and execute it using the interpreter. Though writing the script is a better way of writing a code which could be used and modified later yet the interactive mode is also very helpful. We can quickly check how a command works, what are its attributes, we can quickly try something that we want to write and see the results, and we can test and debug our code easily and also get help related to any command quickly. It is also a good practice to start with the interpreter to learn about the different aspects of the language and then utilize them to create the script by combining the blocks.

HELLO WORLD PROGRAM

So for the customary “Hello World” program, we can simply go ahead into the Python interpreter by typing “Python” and write the code.

```
print "Hello World"
```



```
C:\Python27>python
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>>
```

FIGURE 13.1

Hello World example.

This prints “Hello World” into the interpreter and it can’t get simpler than this. If we want this in a script form then we can write the same code in a text file and save it as helloworld.py. Now to execute it we need to call this file through Python.

```
python helloworld.py
```

In Windows we can also call the script file simply from the command prompt to run our script or by double clicking on the script file. In Linux environment we can execute this script directly using the dot slash notation, but for that first we need to make the file executable using the command “chmod.”

```
chmod 755 helloworld.py
./helloworld.py
```

Though it is not mandatory it is a good practice to specify Python environment into the script file itself through shebang notation. For this we simply need to include the following line at the starting of the script file.

```
#!/usr/bin/python
```

It simply specifies where is the interpreter required to execute this file. This is only supported in Linux environment but including it into the code does not have any change in Windows environment so it is better to include it so that the same code can execute in both the environments. If multiple interpreters are installed in Linux then we can simply change the environment path to the one suitable for our code, for example, if both Python 3.0 and 2.7 are installed we can write `#!/usr/bin/Python2.7` to use the 2.7 interpreter to execute our code.

IDENTIFIERS

In programming, identifiers are the names used to identify any variable, function, class, and other similar objects used in a program. In Python, they can start with an alphabet or an underscore followed by alphabets, digits, and underscore. They can contain a single character also. So we can create identifiers accordingly, except certain words which are reserved for special purposes, for example, “for,” “if,” “try,” etc. Python is also case sensitive which means “test” and “Test” are different.

DATA TYPES

Python has different variable types, but is decided by the value passed to it and does not require to be stated explicitly. Actually the data type is not associated with the variable name but the value object and the variable simply references to it. So a variable can be assigned to another data type after it already refers to a different data type.

```
C:\Python27>python
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> test=10
>>> test
10
>>> test="This is a test"
>>> test
'This is a test'
>>>
```

FIGURE 13.2

Value assignment.

Commonly used data types are:

- Numbers
- String
- Lists
- Tuples
- Dictionaries

To define a number simply assign a variable with a number value, for example,

```
>>>samplenum=10
```

Just to know there are various types of numerical such as float, long, etc.

To define a string we can use the help of quotes (both single and double), for example,

```
>>>samplestr="This is a string"
>>>samplestr2='This is another string'
```

We can also utilize both the types of quotes in a nested form. To create multiline strings we can use triple quotes.

```

>>> tripquot="""This is triple Quotes
... Another line
... Yet another
... And one more"""
>>> tripquot
'This is triple Quotes\n\nAnother line\n\nYet another\n\nAnd one more'

```

FIGURE 13.3

Triple quotes.

We can also utilize the `%` operator for strings to include different data types. The values are passed in a tuple (discussed later) and `%d` is for integers, `%s` is for strings, `%f` for float.

Example code

```

>>>sample_str="There are total %d number of floors in the %s
    building"%(4,'xyz')
>>>sample_str

```

There are total 4 number of floors in the xyz building

Python provides an interesting data type called list and according to its name it is a list of variables of different types. To create a list we can utilize square brackets and separate the variables with commas.

```

>>>samplelist=[123, "str", 'xyz', 321, 21.22]
>>>samplelist
[123, "str", 'xyz', 321, 21.22]
>>>samplelist[1]
'str'

```

Tuples are similar to lists but are immutable and are created using parentheses.

```

>>> samplelist=[123, "str", 'xyz', 321, 21.22]
>>> samplelist
[123, 'str', 'xyz', 321, 21.22]
>>> samplelist[1]
'str'
>>> sampletuple=(123, "str", 'xyz', 321, 21.22)
>>> sampletuple
(123, 'str', 'xyz', 321, 21.22)
>>> sampletuple[2]
'xyz'
>>> sampletuple[2]=21
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> samplelist[2]=21
>>> samplelist[2]
21

```

FIGURE 13.4

List and tuples.

Dictionary is another interesting data type which consists of items with values associated with them. In these key-value pairs the key needs to be unique whereas the value can change.

```

>>>sampldict={'test1':'123','test2':'234','test3':'345'}
>>>sampldict['test1']
'123'
>>>sampldict['test4']='456'
>>>sampldict['test3']='333'
>>>sampldict
{'test1': '123', 'test2': '234', 'test3': '333', 'test4': '456'}

```

There are also various functions provided by different object which can be of great help at times, instead of writing whole new set of code to perform it. To find out these we can get help from Python functions “dir” and “help”.

```

>>>dir(sampldict)
>>>help(sampldict)

```

```

>>> dir(sampldict)
['_class', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popi',
 'popitem', 'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
>>> help(sampldict)
Help on dict object:

class dict(object)
| dict() -> new empty dictionary
| dict(mapping) -> new dictionary initialized from a mapping object's
| (key, value) pairs
| dict(iterable) -> new dictionary initialized as if via:
|     d = {}
|     for k, v in iterable:
|         d[k] = v
| dict(**kwargs) -> new dictionary initialized with the name=value pairs
| in the keyword argument list.  For example:  dict(one=1, two=2)
|
| Methods defined here:
|
| __cmp__(...)
|     x.__cmp__(y) <=> cmp(x,y)
|
| __contains__(...)
|     D.__contains__(k) -> True if D has a key k, else False
|
| __delitem__(...)
|     x.__delitem__(y) <=> del x[y]
|
| __eq__(...)
|     x.__eq__(y) <=> x==y
|
| __ge__(...)
|     x.__ge__(y) <=> x>=y
|
| __getattr__(...)
|     x.__getattr__('name') <=> x.name
|
| __getitem__(...)
|     x.__getitem__(y) <=> x[y]
|
| __gt__(...)
|     x.__gt__(y) <=> x>y
|
| __init__(...)
|     x.__init__(...) initializes x; see help(type(x)) for signature
|
| __iter__(...)
|     x.__iter__() <=> iter(x)
|
| __le__(...)
|     x.__le__(y) <=> x<=y

```

FIGURE 13.5

Using Python help.

We have demonstrated some basics of data types but there is much more operations which can be performed on these data types. Some basic examples are shown below:

```
>>>a=12
>>>b=2
>>>a*b
24
>>>a="test"
>>>b="next"
>>>a+b
'test next'
>>>lt1=['1','2','3']
>>>lt2=['4','5','6']
>>>lt1+lt2
['1', '2', '3', '4', '5', '6']
```

We can perform various operations on these elements. Some examples are shown below.

```
>>>a=1
>>>b=2
>>>a+b
3
>>>a="test"
>>>b="string"
>>>a+b
'teststring'
>>>a.upper()
'TEST'
>>>c="This is a string"
>>>c.find('ring')
12
>>>c.find('xyz')
-1
>>>sample_list=['qw','er','ty',123]
>>>sample_list.append(456)
>>>sample_list
['qw', 'er', 'ty', 123, 456]
```

INDENTATION

Before moving further let's clear up on one import concept of Python. Python supports code readability. Unlike other languages such as C++ it does not use brackets to specify the code blocks, whereas uses indentation. So when creating a block of code we need to provide whitespaces to indicate the structure. One important point is that we can have variable number of spaces for indentation but within a block all the statements should have the same amount. Some people use spaces for indentation and some use the tab feature, it is better to stick with one and not mix up both in a

single code. The examples shown in the following chapter will work on this concept and we will be using spaces.

Basic terms (class, function, conditional statements, loops, etc.)

Now let's move forward with conditional statements.

The most basic conditional statement is “if.” The logic is simple, if the provided condition is it will execute the statement, else it will move on. Basic structure of “if” and associated conditions is shown below.

```
if condition:
    then_this_statement
elif condition:
    then_this_statement
else:
    this_condition
```

Example code

```
#!/usr/bin/python
a=10
b=12
c=15
if (a==b):
    print "a=b"
elif (b==c):
    print "b=c"
elif (c==a):
    print "c=a"
else:
    print "none"
```

Write this in a notepad file and save it as `if_con.py`. This code will result in the response “none,” when executed in Python. The “elif” and “else” conditions are not mandatory when using “if” statement and we can have multiple “elif” statements. Similarly we can also have nested “if” conditions where there will be if statements within another if statement, just proper indentation needs to be kept in mind.

```
if condition:
    then_this_statement
    if nested_condition:
        then_this_nested_statement
    else nested-else_condition:
        then_this_nested-else_statement
```

The “while” loop is next in line. Here we will provide the condition and the loop will run until that condition is true . Structure of “while” is shown below.

```
while this_condition_statement_is-true:
    run_this_statement
```


Example code

```
#!/usr/bin/python
a=10
c=15
while (a<c):
    print a
    a=a+1
```

Output

```
10
11
12
13
14
```

We can also utilize “break” and “continue” statement to control the flow of the loop. The “break” statement is used to break out of the current loop and the “continue” statement is used to pass the control back to the starting of the loop. There is one more interesting statement called “pass” which does nothing, in particular is used just as a placeholder.

Another useful conditional statement is “for” loop. Using it we can iterate through the items present within an object such as a tuple or list.

Example code

```
#!/usr/bin/python
sample_tup=('23','test',12,'w2')
for items in sample_tup:
    print items
```

Output

```
123
test
12
w2
```

We are simply passing the individual values in the tuple `sample_tup` and putting them inside the variable `items` one by one and printing them.

Example code

```
#!/usr/bin/python
str="String"
for items in str:
    print items
```

Output

```
S
t
r
i
n
g
```

We can also utilize the attributes of the objects (find through “dir” and “help”) for the iteration purpose. Similar to “while” we can also use “break” and “continue” statements in “for” loop as well.

Now we are done with the conditional statements and move forward with other structures.

MODULES

Sometimes there is a need to reuse the code or manage it depending upon our requirement, this is where modules come into picture. Say there are multiple components of an object and these components are also required in some other object, so instead of creating these components again and again we can simply create and store them separately and call them into the object as and when required. For example, creating a program for an entity car and another for truck, both will have common components such as brakes, accelerator, etc., so we will code these components once and simply call them into the program according to the requirement, instead of creating them again and again. This is very helpful in organizing and managing the code.

Modules can define variables, functions, and classes, we will discuss about these shortly. Once we create these and save them in separate files, we can import them into our code and use their functionalities.

Example code

```
#!/usr/bin/python
y="Module String"
```

Save this as x.py. Create another file called mod.py and save the following code into it:

```
#!/usr/bin/python
import x
print x.y
```

output

```
Module String
```

So we simply created a module with just a variable, called it into another code and used its variable. Utilizing modules we can create complex programs without

cluttering all the code into a single file. We can also import a module using the call “from module_name import desired_portion”.

Let’s learn about functions and classes.

FUNCTIONS

Functions help to group a set of code as a single functionality, which is useful in code with large number of lines of code. Function start with the keyword “def” followed by the function name and then the parenthesis inside which the arguments are placed and then the colon. Functions also contain a return statement to terminate it and pass back values (can be null). To call a function we can use its name along with the values to be passed (inside the parenthesis).

Example code

```
#!/usr/bin/python
def simplefunc(atr_arg):
    print "Print me first"
    print atr_arg
    return
str="Sample String"
simplefunc(str)
```

Output

```
Print me first
Sample String
```

CLASSES

Using classes we can group different operations together. To create a class we simply need to start with the keyword class followed by a name for the class and then a colon.

Example code

```
#!/usr/bin/python
class sample_class:
    def __init__(self, classarg):
        self.cla=classarg
    def firstfunc(self):
        print "First Function"
        return self.cla+" Return"
    def secfunc(self):
        print "Second Function"
        return self.cla+" Return"
classobj=sample_class("Argument")
print classobj.firstfunc()
print classobj.secfunc()
```

Output

```
First Function
Argument Return
Second Function
Argument Return
```

Here the function `__init__` is the constructor of the class and is the first function which runs in the class. The variable “classobj” is the object for the class “sample_class” and using it we can communicate with the objects inside the class. As discussed earlier we can also create this as a module and call it inside another program.

As discussed earlier, let’s take another example of importing modules.

Example code

```
#!/usr/bin/python
class sample_class:
    def __init__(self, classarg):
        self.cla=classarg
    def firstfunc(self):
        print “First Function”
        return self.cla+“ Return”
    def secfunc(self):
        print “Second Function”
        return self.cla+“ Return”
classobj=sample_class(“Argument”)
```

This file is being saved as `mod.py` and another file calls this as a module with the code:

```
#!/usr/bin/python
from mod import *
print classobj.firstfunc()
```

Output

```
First Function
Argument
```

In Python we can also create directory of modules for better organization through packages. They are hierarchical structures and can contain modules and subpackages.

WORKING WITH FILES

Sometimes there is a need to save or retrieve data from files for this we will learn how to deal with files in Python.

First of all, to open a file we need to create an object for it using the function `open` and provide the mode operation.

```
>>>sample_file=open(‘text.txt’,“w”)
```

Here the name `sample_file` is the object and using `open` function we are opening the file `text.txt`. If the file with this name does not already exists it will be created and if already exists it will be overwritten. The last portion inside the parenthesis describes

the mode, here it is w which means write mode. Some other commonly used modes are “r” for reading, “a” for append, “r+” for both read and write without overwriting, and “w+” for read and write with overwriting.

Now we have created an object so let’s go ahead and write some data to our file.

```
>>>sample_file("test data")
```

Once we are done with writing data to the file we can simply close it.

```
>>>sample_file.close()
```

Now to read a file we can do the following:

```
>>>sample_file=open('text.txt','r')
>>>sample_file.read()
'test data'
>>>sample_file.close()
```

Similarly we can also append data to files using “a” mode and write() function.

Python has various inbuilt as well as third party modules and packages which are very useful. In case we encounter a specific problem that we need to solve using Python code it is better to look for an existing module first. This saves a lot of time figuring out the steps and writing huge amount of code through simply importing the modules and utilizing the existing functions. Let’s check some of these.

Sys

As stated in its help file this module provides access to some objects used and maintained by interpreter and functions that strongly interact with it.

To use it we import it into our program.

```
import sys
```

Some of the useful features provided by it are argv, stdin, stdout, version, exit(), etc.

Re

Many times we need to perform pattern matching to extract relevant data from a large amount of it. This is when regular expressions are helpful. Python provides “re” module to perform such operations.

```
import re
```

Os

The “os” module in Python allows to perform operating system-dependent functionalities.

```
import os
```

Some sample usages are to create directories using mkdir function, rename a file using rename function, kill a process using kill function and display list of entries in a directory using listdir function.

Urllib2

This module allows to perform URL-related operations such as open a web page. It is very helpful when working with web applications.

```
import urllib2
```

There are many other useful modules such as Scapy (network), Scrapy (web scraping), nose (testing), mechanize (stateful web browsing), and others which provide huge amount of functionalities in their domain. Some modules are inbuilt and some need to be installed separately. There is still much more to explore in this topic but here we will be stopping with these points and move on to the next topic.

USER INPUT

Certain problems require to take user input. Here are two methods to do so:

Using Sys module we can take user input from command line argument.

Example code

```
#!/usr/bin/python
import sys
a=sys.argv[1]
print a
print a*4
a=int(a)
print a
print a*4
```

Save this as usrip.py and pass the command line argument.

```
C:\Python27>usrip.py 2
```

Output

```
2
2222
2
8
```

argv is a list that takes command line arguments where the index 0 is reserved for filename. We can also pass multiple values and iterate by changing the index value of argv. Here we have also demonstrated a simple type conversion (string to integer).

Another method is to get input at run time, this can be done using raw_input.

Example code

```
#!/usr/bin/python
import sys
a=raw_input("Enter something: ")
print a*4
```

When executing this code, it will prompt the message “Enter something”, once we input the value it will generate the response accordingly. For an input value “a” it will generate the output “aaaa”.

COMMON MISTAKES

Some common issues faced during the execution of Python code are as follows.

Indentation

As shown in examples above, Python uses indentations for grouping the code. Some people use spaces for this and some use tabs. When running the code written by some person or modifying it we sometimes face the indentation error. To resolve this error, check the code for proper indentation and correct the instances; also make sure to not mess up by using tabs as well as spaces in the same code as it creates confusion for the person looking at the code.

Libraries

Sometimes people have a completely correct code, yet it fails to execute with a library error. The reason is missing of a library that is being called in the code. Though it is a novice mistake, sometimes experienced people also don't read the exact error and start looking for errors in the code. The simple solution is to install the required library.

Interpreter version

Sometimes the code is written for a specific version of the language and when being executed in a different environment, it breaks. To correct this, install the required version and specify it in the code as shown earlier in this chapter or execute the code using the specific interpreter. Sometimes there are multiple codes which require different versions; to solve this problem we can use virtualenv, which allows us to create an isolated virtual environment where we can include all the dependencies to run our code.

Permission

Sometimes the file permissions are not set properly to execute the code so make the changes accordingly using chmod.

Quotes

When copying code from some resources such as documents and websites there is a conversion between single quote (‘) and grave accent (`) which causes errors. Identify such conversions and make the changes to the code accordingly.

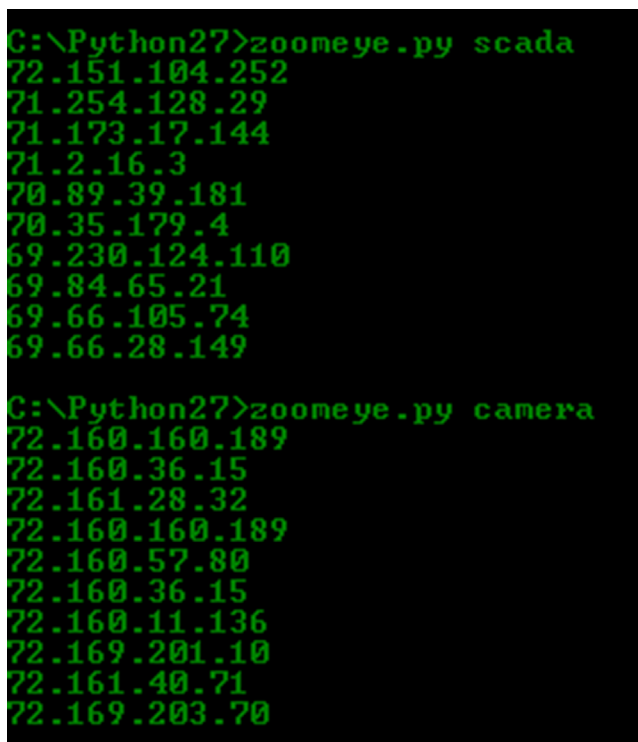
So we have covered basics about the language let's see some examples which can help us to understand the concepts and understand their practical usage and also get introduced to some topics not discussed above.

Similar to shodan, discussed in a previous chapter there is another service called zoomeye. In this example we will be creating a script using which will query

zoomeye and extract the IP address from the result page. We have to pass the query term from command line.

For this we will first create the URL for this by combining the base URL and the search term passed through command line. Then we will send the request to this URL using the function `urlopen` from the module `urllib2`. Further we will be parsing the response page and extract the IP addresses from it using `BeautifulSoup`.

```
#!/usr/bin/python
import sys
import urllib2
from bs4 import BeautifulSoup
url="http://www.zoomeye.org/search?q="
term=sys.argv[1]
comurl=url+term
response=urllib2.urlopen(comurl)
soup = BeautifulSoup(response)
for item in soup.findAll("a",{'class':'ip'}):
    print item.string
```



```
C:\Python27>zoomeye.py scada
72.151.104.252
71.254.128.29
71.173.17.144
71.2.16.3
70.89.39.181
70.35.179.4
69.230.124.110
69.84.65.21
69.66.105.74
69.66.28.149

C:\Python27>zoomeye.py camera
72.160.160.189
72.160.36.15
72.161.28.32
72.160.160.189
72.160.57.80
72.160.36.15
72.160.11.136
72.169.201.10
72.161.40.71
72.169.203.70
```

FIGURE 13.6

Zoomeye script result.

For our next example we will create an extension for Burp Suite. Burp Suite is an application proxy which is used for web application security assessment. It allows to create extensions through which we can extend its functionalities. For our extension we will simply extract the host name of the target.

```
#!/usr/bin/python

# A sample burp extension in python (needs jython) which extracts
hostname from the request (Target Tab).
from burp import IBurpExtender
from burp import IMenuItemHandler
import re
import urllib2

class BurpExtender(IBurpExtender):
    def registerExtenderCallbacks(self, callbacks):
        self.mCallbacks = callbacks
        self.mCallbacks.registerMenuItem("Sample Extension",
hostnamefunc())

class hostnamefunc(IMenuItemHandler):
    def menuItemClicked(self, menuItemCaption, messageInfo):
        print "--- Hostname Extract ---"

        if messageInfo:
            request1=HttpRequest(messageInfo[0].getRequest())
            req=request1.request
            host=req[1]
            print host
            print "DONE"

class HttpRequest:
    def __init__(self, request):
        self.request=request.toString().splitlines()
```

To make this extension run, first we need to install Jython and configure it under the options tab within extender. Once this is done we can add our extension under the Extensions tab within extender. To use our extension we simply need to right click on a target domain under the target tab and click on the "Sample Extension" on the right click menu, the result will be shown in the Extensions in extender tab. The example is simply to demonstrate an extension using Python we can further enhance it by performing other operations on the host name.

MALTEGO TRANSFORMS

In a previous chapter we discussed about Maltego, a simple and effective open source intelligence (OSINT) tool. We learned how to use it, what all features it provided, what are its elements, etc. Let's take this a step further and utilizing the knowledge of Python we have just acquired to extend this framework. As mentioned in a previous chapter, the power of Maltego lies in its transforms. For quick recall a transform is basically a piece of code which takes an entity (or a group of entities) as an input and

extracts data in the form of entity (or entities) based upon the relationship. Maltego has a lot of inbuilt transforms and keeps on updating the framework with new ones, but it also allows to create new ones and use them, this can be very helpful when we need something custom according to our needs.

Before we move any further we need the “MaltegoTransform” Python library by Andrew MacPherson, which is very helpful in local transform development. It can be downloaded from the page <https://www.paterva.com/web6/documentation/develop-local.php>. Some basic examples of local transforms created using the library are also present at the bottom of the page. Once we have the library in our directory we are ready to go and create our own first transforms.

To create any program first we need to have a problem statement. Here we need to create a transform so let’s first identify something that would be helpful during our OSINT exercise. There is a service called as HaveIBeenPwned (<https://haveibeenpwned.com>) created by Troy Hunt which allows users to check if their account has been compromised in a breach. It also provides an application programming interface (API) using which we can perform the same function. We will be using the v1 of the API (<https://haveibeenpwned.com/API/v1>) and provide an e-mail address to check if our supplied e-mail has any account associated.

To utilize the API we simply need to send a GET request to the service in the form shown below and it will provide a JSON response to show the website names.

<https://haveibeenpwned.com/api/breachedaccount/{account}>

Let’s first specify the path of the interpreter

```
#!/usr/bin/python
```

Now we need to import the library MaltegoTransform

```
from MaltegoTransform import *
```

Once we have the main library we need to import some other libraries that will be required. Library “sys” is to take user input and urllib2 to make the GET request.

```
import sys
import urllib2
```

Once we have imported all the required libraries, we need to assign the function MaltegoTransform() to a variable and pass the user input (e-mail address) from Maltego interface to it.

```
mt = MaltegoTransform()
mt.parseArguments(sys.argv)
```

Now we can pass the e-mail value to a variable so that we can use it to create the URL required to send the GET request.

```
email=mt.getValue()
```

Let’s create a variable and save the base URL in it.

```
hibp="https://haveibeenpwned.com/api/breachedaccount/"
```

As we have both the parts of the complete URL, now we can simply combine them to create the complete URL.

```
getrequrl=hibp+email
```

Let's send the GET request using the function `urlopen` in the library `urllib2` and store the response in a variable, but while handling the exception. Now we need to run a for loop to go through the values being stored in the variable (response) and add these values to the variable for the transform.

```
try:
    response = urllib2.urlopen(getrequrl)
    for rep in response:
        mt.addEntity("maltego.Phrase", "Pwned at " + rep)
except:
    print ""
```

In this last step we need to return the output of the variable.

```
mt.returnoutput()
```

Now simply save this as `emailhibp.py`.

Complete code

```
#!/usr/bin/python
from MaltegoTransform import *
import sys
import urllib2
mt = MaltegoTransform()
mt.parseArguments(sys.argv)
email=mt.getValue()
hibp="https://haveibeenpwned.com/api/breachedaccount/"
getrequrl=hibp+email
try:
    response = urllib2.urlopen(getrequrl)
    for rep in response:
        mt.addEntity("maltego.Phrase", "Pwned at " + rep)
except:
    print ""
mt.returnoutput()
```

Now to check if our code is running properly we simply need to execute this program in the terminal and pass an e-mail address as a command line argument.

Example

```
./emailhibp.py foo@bar.com
```

or

```
python ./emailhibp.py foo@bar.com
```

```

C:\Python27>emailhibp.py foo@bar.com
<MaltegoMessage>
<MaltegoTransformResponseMessage>
<Entities>
<Entity Type="maltego.Phrase">
<Value>Pwned at ["Adobe", "Gawker", "Stratfor"]</Value>
<Weight>100</Weight>
</Entity>
</Entities>
</UIMessages>
</UIMessages>
</MaltegoTransformResponseMessage>
</MaltegoMessage>

```

FIGURE 13.7

Transform output.

We can see that the response is a XML styled output and contains the string “Pwned at [“Adobe”, “Gawker”, “Stratfor”]”. This means our code is working properly and we can use this as a transform. Maltego takes this XML result and parses it to create an output. Now our next step is to configure this as a transform in Maltego.

Under the manage tab go to Local Transform button to start the Local Transform Setup Wizard. This wizard will help us to configure our transform and include it into our Maltego instance.

In the Display name field provide the name for the transform and press tab, it will generate a Transform ID automatically. Now write a small description for the transform in the Description field and the name of the Author in the Author field. Next we have to select what would be the entity type that this transform takes as input, in this case it would be Email Address. Once the input entity type is selected we can choose the transform set under which our transform would appear which can also be none.

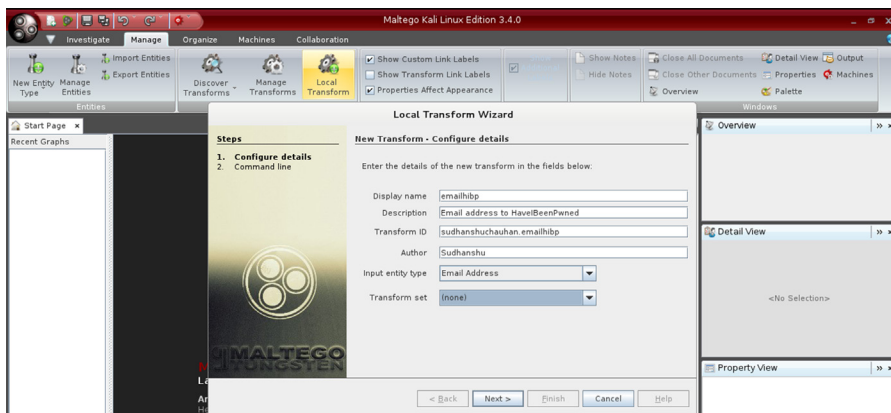


FIGURE 13.8

Transform setup wizard.

Now click on next and move to the second phase of the wizard. Here under the command field we need to provide the path to the programming environment we are going to use to run the transform code. In our case it would be

```
/usr/bin/python (for Linux)
```

```
C:\Python27\python.exe (for Windows)
```

Once the environment is set we can move to the parameters field, here we will provide the path to our transform script. For example,

```
/root/Desktop/transforms/emailhibp.py (for Linux)
```

```
C:\Python27\transforms\emailhibp.py (for Windows)
```

One point to keep in mind here is that if we select the transform file using the browse button provided in front of the “Parameters” field, then it will simply take the file name in the field, but we need absolute path of the transform to execute it so provide the path accordingly.



FIGURE 13.9

Transform setup wizard.

After all the information is filled into the place we simply need to finish the wizard and our transform is ready to run. To verify this, simply take an e-mail address entity and select the transform from the right click menu.

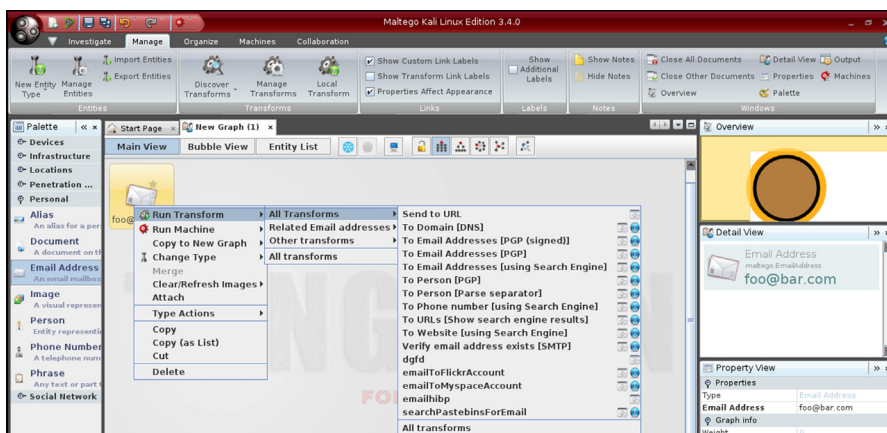


FIGURE 13.10

Select transform.

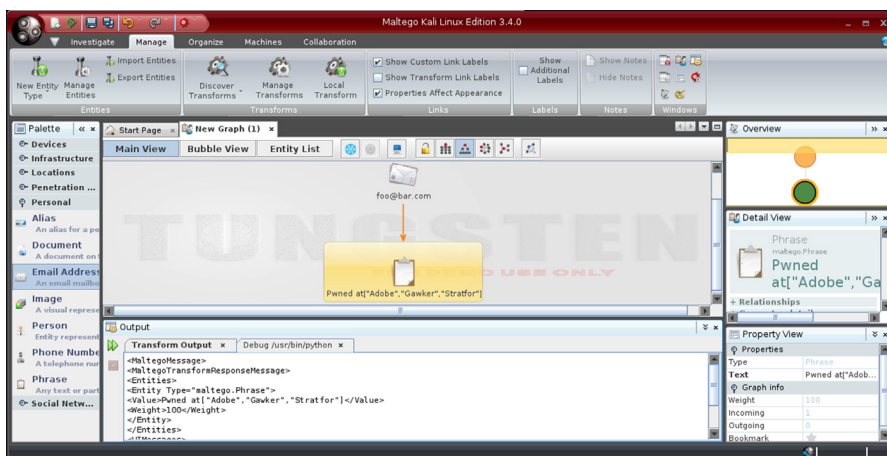


FIGURE 13.11

Transform execution.

Now we have created our first transform and also learned how to configure it in Maltego. Let's create another simple transform. For this example we will be using the website <http://www.my-ip-neighbors.com/>. It allows to perform a reverse IP domain lookup, simply said the domains sharing the same IP address as the one of the provided domain. As in the previous transform we provided an e-mail address as the input here we require a domain name, but this website provides no API service and hence we will have to send the raw GET request and extract the domains out of the web page using regular expressions through the library "re".

```
#!/usr/bin/python
from MaltegoTransform import *
import sys
import urllib2
import re
mt = MaltegoTransform()
mt.parseArguments(sys.argv)
url=mt.getValue()
mt = MaltegoTransform()
opencnam="http://www.my-ip-neighbors.com/?domain="
getrequrl=opencnam+url
header={'User-Agent': 'Mozilla'}
req=urllib2.Request(getrequrl,None,header)
response=urllib2.urlopen(req)
domains=re.findall("((?:[0-9]*[a-z][a-z\\.\\\\d\\-]+)\\.\\.(?:[0-9]*[a-z][a-z\\.\\\\d\\-]+))(![\\w\\.])",response.read())
for domain in domains:
    mt.addEntity("maltego.Domain", domain)
mt.returnoutput()
```

*<http://txt2re.com/> can be used to create regular expressions.

Similarly we can create lot of transforms which utilize online services, local tools (e.g., Nmap scan), and much more using Python. The examples shown above and some more can be found at <https://github.com/SudhanshuC/Maltego-Transforms>. Some other interesting transforms can be found at <https://github.com/cmlh>, else they are just a quick Github search away (<https://github.com/search?utf8=%E2%9C%93&q=maltego+transform>).

There is also a Python-based framework available, which allows creating Maltego tranforms easily called as Canari (<http://www.canariproject.com/>).

There are various topics which we have not covered but the scope is limited and topic is very vast. Some of these are exception handling, multiprocessing, and multithreading. Below there are some resources which can be helpful in this quest of learning Python.

RESOURCE

<https://github.com/dloss/python-pentest-tools>

A great resource to learn more about Python and its usage is the Python docs itself <https://docs.python.org/2/>. Another great list of Python-based tools with focus on pentesting is present at <https://github.com/dloss/python-pentest-tools>. It would be great to create something interesting and useful by modifying, combining, and adding to the mentioned resources. The list is divided into different sections based on the functionality provided by the tool mentioned.

So we have covered some basics of Python language and also learned how to extend Maltego framework through it. Through this chapter we have made an attempt to learn about creating own custom tools and modify existing ones in a quick fashion.

This chapter is just an introduction of how we can simply create tools with minimum amount of coding. There is certainly room for improvement in the snippets we have shown in functional as well as structural terms, but our aim is to perform the task as quickly as possible.

Though we have tried to cover as much ground as possible yet there is so much more to learn when it comes to Python scripting. Python comes with a large set of useful resources and is very powerful; and by using it one can create power tool-set, recon-ng (<https://bitbucket.org/LaNMaSteR53/recon-ng>) is great example of it. We have discussed about this Reconnaissance framework in a previous chapter. One great way to take this learning further would be to practice more and create such tools which could be helpful for the community and contribute to the existing ones such as recon-ng.

Slowly we are moving toward the end of this journey of learning. We have been through different aspects of intelligence gathering in different manners. Moving on we will be learning about some examples and scenarios related to our endeavor, where we can utilize the knowledge we have gained in a combined form.