FORWARD DEFENSE

Analyst Reference

Python Analyst Reference

Version 20190124

Contents

Credits and Formatting Notes

This document is extrapolated or quoted from Mark Baggett's SEC573: Automating Information Security with Python course offered by SANS. All information is used with the author's permission. This document was originally designed to provide a quick reference to students in that course but may also serve as a useful guide to others writing Python code that is both backward compatible with Python 2 and forward compatible with Python 3.

Throughout this document, examples will be provided using a Python interactive shell. These examples will begin with the Python prompt >>> and the output will follow on the next line or in some cases to the right on the same line. Code examples are listed in *italics*. Items indicating the type of information expected in a command are listed between <> such as *print(<thing to print>)*.

Python Variable Types

Name	Example	Notes
Integers (int)	1	Whole Number. Casting a float to an int will do a floor operation, not round. To round, use round() function first, e.g. int(round(100.9)) produces 101 . The int() function can take a string value and then a base, so int("11000101",2) would be 197 and int('0xc5',16) would also be 197
Longs (long)		Integer larger than CPU word size of 32 or 64 bit
Floats (float)	5.42	Real numbers
Strings (str)	"A" or 'A' or """A"""	
List (list)	["this", "is", "a", "list"]	Support a rich set of methods for manipulation, see the Lists section of this document.
Tuples (tuple)	("Here's", "a", "tuple")	Lighter weight than lists since they don't support as many menthods.
Dictionary (dict)	{"Key1":"Value1", "Key2":"Value2"}	Very fast retrieval based on the location of each key in memory.
Hexadecimal (hex)	0xff4d	Hex and bin are stored as int's or strings but can be displayed as hex or bin by casting to hex or bin. See int notes for converting binary or hex string to int.
Binary (bin)	0b11010001	Hex and bin are stored as int's or strings but can be displayed as hex or bin by casting to hex or bin. See int notes for converting binary or hex string to int.

Mathematical and Logic Operators

Mathematical Operators, when x = 5

Operation	Example	Result in x
Addition	x = x + 5	10
Subtraction	x = x - 10	-5
Multiplication	x = x * 5	25
Division (Python 3)	x = x / 2	2.5
Division (Python 2)	x = x / 2	2
Floor	x = x // 2	2
Modulo	x = x % 2	1
Exponent	x = x ** 2	25

In Python 2, if either dividend or divisor are floats, the result is a float, but if both are integers, the result will be an integer. To produce an integer result, Python 2 does a floor operation. In Python 3, even if both divisor and dividend are integers, the division result will be a float. To import Python 3 division into Python 2 use *from __future__ import division*

Floats are an approximation, so you need to specify the precision when doing comparisons: Examples:

```
>>> 0.1 + 0.2 == 0.3
```

False

Since the precision is not specified in the example above, Python attempts to carry the precision out to many decimal places, which at some point become non-zero since floats are an approximation. This makes the comparison not yield the expected result. Use format or round to define the precision:

```
>>> format(0.1 + 0.2, '3.1f') == format(0.3, '3.1f')
```

True

```
>>> format(0.1 + 0.2, '3.1f') == format(0.3, '3.1f')
```

True

```
>>> round (0.1 + 0.2, 3) == round(0.3, 3)
```

True

Mathematical Order of Operations

- Parenthesis, Exponents, Multiply and Divide, Add and Subtract
- Boolean AND before OR

Logical Operators

==	Equal
!=	Not Equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
and	And
or	Or
^	XOR

False and True are keywords and should not be quoted like strings. False, None, 0 and Empty values are False. Everything else is True.

>>> bool(1) is True

>>> bool([]) is False

>>> bool({}) is False

>>> bool(None) is False

>>> bool() is False

>>> bool(False) is False

>>> bool("False") is True since it is a string with a non-empty value, not the unquoted keyword False.

Python evaluates only enough of an expression to return a value with the same Boolean value as the whole expression. This is called shortcut processing, and behaves as follows:

OR expression, return 1st item if it is True, else return 2nd item (you can prove an OR in one item)

AND expression, return 1st item if it is False, else return 2nd item (you can disprove an AND in one item)

Python 2/3 Compatibility Imports

from __future__ import division (this imports the Python 3 division function which returns a real number result even if both the divisor and dividend are integers)

from __future__ import print_function (In Python 2, print was a built-in keyword, but in Python 3 it is a function. Since the keyword is not supported in Python 3, always use the function which must be imported into Python 2)

```
try:
    input = raw_input
except:
    pass
```

Python 2 has two functions that can accept input, *input()* and *raw_input()*. *raw_input()* always returns a string and is the safer one to use. *input()* in Python 2 evaluates the input before returning it, which can allow string injection attacks. In Python 3, the *input()* function is actually the same as *raw_input()* in Python 2 and Python 3 does not have the dangerous *input()* function at all.

Format Strings

("Some String") % (variable1, variable2)

String	Meaning
%d	Integer decimal
%10d	Integer decimal, 10 digits wide
%010d	Integer decimal, 10 digits wide, leading zeros
%x	Hex in lowercase
%X	Hex in uppercase
%f	Floating-point decimal (around six characters after the decimal by default)
%6.2f	Floating-point decimal, 6 wide (in total) with 2 after decimal. Note that the decimal point counts as one of the 6 characters, so it will be ###.## for a total of 6 characters, 2 after the decimal point. The result is rounded, not truncated.
%s	String
%%	Escapes the percent sign to print a single %

Examples:

print("I'd like %d %ss" % (5, 'parrot')) produces "I'd like 5 parrots"

newstring = ("I'm %d%% sure") % (100) produces "I'm 100% sure" and assigns it to newstring

String .format() Method

Syntax in the string: *{<Argument number>:<fill character><Alignment><Length><type>}*

Alignment options:

- < Left Align
- ^ Center
- > Right Align

Length

- Can be a single number or include a decimal point when a float is used to show the number of places after the decimal point (rounded, not truncated).
- Example *print("There are {0:6.2f} percent".format(44.365))* produces There are 44.37 percent

Type Options

- X for uppercase hexadecimal
- x for lowercase hexadecimal
- d for decimal
- f for float
- leave unspecified for string

Examples:

>>> "the number is {0:0>10d}".format(22)

'the number is 0000000022'

>>> '{0:a>6d} {1:X}'.format(22,22)

'aaaa22 16'

>>> """{} Flying {}""".format("Monty Python's","Circus")

"Monty Python's Flying Circus"

Built-In format() function

Takes a value and then a string describing how to format the value. Uses the same notation for the string as the Python 3 *.format()* method (the part after the colon)

Example:

>>> format(10.3,"0>8.2f")

'00010.30'

Python 2's byte strings vs. Python 3 UTF-8 strings

Python 2 strings are bytes, but Python 3 strings are UTF-8 encoded. In Python 3, you can convert from bytes to strings with the *decode()* method and from strings to bytes with *encode()* method.

Examples:

In Python 3:

b'ABC'.decode() produces a UTF-8 encoded string u'ABC'

"ABC".encode() produces a byte array of b'ABC'

In Python 3 interactive shell:

>>> b'ABC'.decode()

'ABC'

>>> "ABC".encode()

b'ABC'

also note:

>>> b'ABC'.encode()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'bytes' object has no attribute 'encode'

In Python 2.7

In Python 2.7 interactive shell:

>>> "test".encode()

'test'

>>> "test".decode()

u'test'

>>> "test"

'test'

String Methods

If x = “pyWars rocks!”

Method	Syntax	Result
Uppercase	x.upper()	PYWARS ROCKS!
Lowercase	x.lower()	pywars rocks!
Title Case	x.title()	Pywars Rocks!
Replace Substring (all occurrences)	x.replace(‘cks’,’x’)	pyWars rox!
Is substring in x? (Case sensitive)	“War” in x	True
	“war” in x	False
Display a list of the split string (original string remains unchanged)	x.split()	[‘pyWars’,’rocks!’]
	x.split('r')	['PyWa', 's ', 'ocks!']
Count substrings	x.count(‘r’)	2
Find first occurrence of substring	x.find(‘W’)	2
Length function (not a method)	len(x)	13
Join	“,”.join([“Make”,”a”,”csv”])	Takes a list and joins the items into a string, separated by the string provided at the beginning
Strip leading and trailing whitespace	.strip()	pyWars rocks!

Note that Title Case capitalizes the first letter of each word, regardless of length or significance.

Remember, strings are immutable, so these methods create a new string rather than modifying the original.

Note that the substring and main string must both be encoded in the same way (bytes or UTF-8).

Python 3 example:

>>> a = "Nasty big pointy teeth"

>>> a.replace('t','')*

'Nas*y big poin*y *ee*h'

But note that if the substring type is bytes when the Python 3 string is UTF-8:

>>> a.replace(b'e','*')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: replace() argument 1 must be str, not bytes

Slicing Strings

String[start (beginning at zero):end (up to but not including):step]

If x = “Python rocks”

Expression	Result
x[0]	P
x[2]	t
x[0:3] or x[:3]	Pyt
x[0:-1] or x[:-1]	Python rock
x[3:]	hon rocks
x[0::2] or x[::2]	Pto ok
x[::-1]	Skcor nohtyP
x[-1]+x[4]+x[7]*2+x[1]	sorry
x[:6][::-1] or x[5::-1]	nohtyP

codecs Module

To use the codecs module, you must first: *import codecs*

codecs.encode(object, <codec to be used passed as a string>)

Common Codecs

Codec	Description
bz2	Bzip2 encoding/decoding, requires bytes-like object
rot13	Rotates letters 13 ASCII places
base64	Base64 encodes/decodes bytes-like object (not UTF-8 strings)
zip	Creates a compressed Zip version, requires bytes-like object
hex	Produces a byte string of hex characters from a bytes-like object
utf-16	2-byte (16-bit) Unicode

Examples if x = “Python rocks” and using Python 2 (note: Python 3 strings are not bytes-like objects):

>>> codecs.encode(x,"rot13")

'Clguba ebpxf'

>>> codecs.encode(x,"utf-16")

b'\xff\xfeP\x00y\x00t\x00h\x00o\x00n\x00 \x00r\x00o\x00c\x00k\x00s\x00'

>>> codecs.encode(x,"base64")

'UHl0aG9uIHJvY2tz\n'

Lists

List Method	Description
.append(value)	Add an object to end of the list
.insert(position, value)	Insert the value at the given position, other items will shift right, position is a positive or negative number. The change is made to the list, so the function just returns None.
.remove(value)	Removes the first matching item by its value. The change is made to the list, so the function just returns None.
.sort(key=…, reverse=…)	Sort the elements of the list by changing the actual list's order. Can provide a key function to use for the sort. Both key and reverse are optional (so .sort(reverse=True) would sort the list backwards according to the default sort key (which is ASCII values). The change is made to the list, so the function just returns None.
.count(value)	Count number of occurrences of an item in the list. Entire list entry must match the provided value (does not look for substrings within items).
.index(value)	Look up where a value is in the list
.reverse()	Reorders the list, in reverse order. Changes the list itself, does not just display it backwards. The change is made to the list, so the function just returns None.

In addition to the list methods above, there are other useful functions that work on lists:

Function	Description
del list[index]	Delete an item by index number (del is a keyword, not a list method)
Sorted([])	Function that will display the list items in a sorted order but not change the list itself. Accepts an optional key=function() to produce an element on which to sort, e.g. sorted(customer_list, key=lowercase) You can optionally pass reverse=True as an argument to reverse the sort.
'a' in list	Looks for any items that match 'a' in the list and return True if present or false if not. Value being searched must match the whole list item, does not search substrings.
sum([])	Provides to sum of a list that contains only numbers (int or float). Traceback if contains strings, tuples, other lists, etc.
zip([],[])	Creates a new list of tuples from two or more lists. The first item from each list are placed in a tuple at in dex 0, the second items from each list are placed in a tuple at index 1, and so on. Stops once one list is exhausted.
map(function,[])	Run the specified function on each item in a list (or any iterable). Note that you use the name of the function without the parentheses. If two lists are provided, the specified function becomes a custom zipper. The result of map is a map object. You can use *list(map(function,[]))* to have it return a list with the results.
enumerate([])	Returns an enumerate object of tuples consisting of each list's items index and value. An example of use is *for index,value in enumerate(some_liist)* where each index would correspond to its associated value as the list is walked. To create a list of tuples with index and value use *list(enumerate(some_list))*

The default action when you copy a list is that it creates a pointer to the list rather than recreates a new list. So if a list called list1 exists, *list2 = list1* would make list2 a pointer to the same list, and changes to list1 would also result in changes to list2.

To copy the list items into a new list, use *list2 =list(list1)*

To copy a list of lists, use the deepcopy() function from the copy module:

Example: copy_of_list_of_lists = copy.deepcopy(list_of_lists)

Math operators work on lists, as does slicing. If a = [1,2] and b = [3,4]

c = a + b would set c to a new list [1, 2, 3, 4] and changes to a or b would not affect c.

*d = b * 2* would set d to a new list [3, 4, 3, 4]

and *c[1:3]* would be [2, 3] (same rules apply as with strings, with negative numbers and stepping)

List Comprehension

Newlist = [<expression> for <iterator> in <list> <filter>]

Example:

a = [x+1 for x in some_list if x < 6]

Note that in Python 2, the variables declared inside a list comprehension do not get their own scope, so if there is a local, global or built-in variable with the same name it will overwrite its values. In Python 3, the variable declared in a list comprehension gets its own scope.

Python 2 example:

```
>>> x = 3
>>> newlist = [x for x in range(10) if x<4]
>>> newlist
[0, 1, 2, 3]
>>> x
9
```

If there is not a variable with that name already declared, it declares it with a persistent scope.

```
>>> newlist = [var for var in range(10) if var<4]
>>> newlist
```

[0, 1, 2, 3]

>>> var

9

Python 3 example:

But with Python 3, the declared variable's scope is just the list comprehension:

>>> newlist = [var for var in range(10) if var<4]

>>> newlist

[0, 1, 2, 3]

>>> var

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'var' is not defined

Lambda functions

A small function, often used as a key or with *map()*.

Syntax is

optional_function_name = lambda <parameters>: <return expression>

Examples:

list(map(lambda x,y:int(x) + int(y), [1,2,'3'], [4,5,6]))

sorted(name_list, key=lambda x: (x.split()[1]+x.split()[0]).lower())

for and while Loops

for loops examples

for x in list:

for x in range(100):

for x in range(<start>, <stop>, <step>): (starts at start, stop is up to but not including the number provided, step is positive or negative as with slicing)

for index, value in enumerate(list):

while loops

- while loops can go on forever, whereas for loops have a defined end.
- while loops can have an else statement, that occurs only once when the test condition evaluates to False. If a break ends the loops, the else is not performed.

break and continue

- A *break* causes a for or while loop to exit the entire loop immediately, skipping any while loop else statement, and continue on to the next code after the loop. A *continue* causes the loop to end the current iteration and start the next iteration.

Tuples

- Tuples are immutable
- Can be declared with parentheses around the list, or just as comma separate values without parentheses.
- Can access individual elements by their index, e.g. tuple[2]

Dictionaries

Dictionary method	Description
.get(key, <value if not found>)	You provide the key, and it returns the value. Optionally, you can also provide a value that will be returned if the key specified is not found. You can also request a value by the key as if it were a list, i.e. dict[key] but if the key does not exist, this method causes a Traceback. Get() on the other hand just returns None if the key does not exist and no optional value to return is provided.
.copy()	Like lists, assigning a dictionary to a variable creates a link to the dictionary, not a new dictionary. To create a copy, you can cast the dictionary to a dict() like with lists, e.g. *b = dict(a)* or you can call the .copy() method, e.g. *b = a.copy()*
.keys()	Returns a list (or view in Python 3) of the keys (ordered based on the memory location of the key prior to Python 3.6 and in the order you put them in in 3.6 and later)
.values()	Returns a list (or view in Python 3) of the values (ordered based on the memory location of the key prior to Python 3.6 and in the order you put them in in 3.6 and later)
.items()	Returns a list (or view in Python 3) of the tuples containing (key,value). Items are ordered based on the memory location of the key prior to Python 3.6 and in the order you put them in in 3.6 and later.

In Python 3, .keys(), .values() and .items() do not return lists but instead return a View object pointing to the dictionary. A view object is iterable but it cannot be sliced or use any list methods. A view is more like a pointer in that if you assign a view to a variable, the elements of the dictionary will update as the dictionary updates.

For a dictionary named dict:

- *<key> in dict* syntax will search through the keys of the dictionary for a key and return True if present or False if not.
- *<value> in dict.values()* will do the same for values.
- *for x in dict* iterates through the keys (the same as *for x in dict.keys()* would)
- There is no efficient way to look up a key based on the value, but looking up a value based on key is very fast.
- *dict[<key>] = <value>* syntax will add a new key, value pair (overwriting the old value at that key if it previously existed)

Specialized Dictionaries

Special Dictionaries in collections Module	Description
defaultdict(default_function)	A dictionary that will create any key that you query and set it to the value returned by the default_function. You can access dict[key] safely for any value since nonexistent keys automatically call the default_function to have a value set.
Counter	Counter automatically counts the number of times a key is set. It is a customized defaultdict, similar to defaultdict(lambda:0) but it also adds additional methods.
	.most_common() counter method lists the keys with the greatest count first (takes an optional value to set the number of keys to display or else it displays all in frequency order).
	.update([]) method takes a list of keys and increments the count for each
	.subtract([]) method takes a list of keys and decrements the count for each

Python Debugger

Three ways to start:

- *import pdb; pdb.set_trace()* at the point where you want a breakpoint
- *python -m pdb <script.py>*
- *python -i <script.py>* then after a crash type *import pdb; pdb.pm()*

PDB Command	Meaning
?	Print help
n	Stop at next line in current function, stepping over any function calls
s	Step into functions and execute the next line
c	Continue to next break point
l start,end	List source code from start to end line. If lines not specified, it prints 11 lines around current line or continues from last listing
p <expression>	Print the value of an expression or variable
r	Finish the current subroutine and return to the calling function
break <options>	Create, list or modify breakpoints in the program. break plus a line number or function name sets a break point. break by itself lists break points.
clear <break_number>	Clears a breakpoint identified by the supplied breakpoint number (breakpoint numbers and their associated line number are displayed when you just type break). enable or disable can also be used on existing breakpoint numbers.
ignore <break_num> <# of times to ignore>	Ignore a breakpoint for a specified number of iterations. Example *ignore 1 3* would ignore breakpoint 1 for the next 3 occurences
condition <break_num> <logic test>	Set a conditional breakpoint by providing the breakpoint number and the logic test. If the test is false, the line will proceed but if the test is true, the line will be a break point. Example *condition 1 i==2* makes the line proceed unless i is equal to 2, in which case execution breaks at that line.
commands <break_num>	Allows you to enter a (com) prompt and specify pdg commands that will execute when the break occurs (common examples are p, args, end and cont).
display <expression>	Display variables as they change every time a breakpoint in reached (Python 3 only)
<ENTER>	Execute the last command again

Ternary Operator

Provides a shortcut way to do a conditional assignment. Example:

x = 10 if y==5 else 11

File Operations

Create a file object with the built-in *open()* function using:

file_handle = open(<complete file path as string>, <mode>)

OR

with open(<complete file path as string>, <mode>) as file_handle:
#Then a code block goes here that will use the file_handle. Any I/O errors encountered will be handled automatically and the file closes automatically once this code block ends.

open() can optionally take an *encoding=* argument if a specific encoding is used in the file (or if you want to read binary data as a string and encode it as Latin1 to avoid corruption). Example:

file_handle = open("/bin/bash", encoding="latin-1")

Where the path to the file is absolute or relative, and the mode is one of the following:

Mode (passed as a string)	Meaning
'r'	Read only - This is the default mode if one is not specified.
'w'	Overwrite (truncate) the file. If the file does not exist it will be created. If it does exist it overwrites the original.
'a'	Append - add data to the end. If the file does not exist it will be created. To change data in the middle, first read the contents in, make changes, and then write the file back.
'b'	Windows Only - Add a 'b' to the mode for binary files. If not, Windows will attempt to "fix" end of line markers which may corrupt binary data. Example: 'rb'
't'	Open in text mode (the default) which interprets unicoded strings and \n or \r\n as end of line. Example: 'rt'
'+'	Add a '+' to the mode to allow simultaneous reading and writing. 'w+' or 'r+' allow for reading and overwriting. 'a+' allows for reading and appending.

File Object Methods

Method	Description
.seek()	Sets the file pointer position
.tell()	Returns the file pointer's current position
.read()	Read the contents of a file as a string
.readlines()	Read the contents of a file as a list of lines
.readline()	Reads the next line of the file as a string
.write()	Writes a string to a file
.writelines()	Iterates over an object that produces strings and writes each to a file.
.close()	Closes the file

Iterating through a file:

```
filehandle = open('filename','r')
for oneline in filehandle:
   print(oneline, end="")
filehandle.close()
```

The os Module for File Operations

- You can use the os module to check if a file exists:

```
>>> import os

>>> os.path.exists("/bin/bash")
```

True

Note that False will be returned if either the file does not exist or the process is running as a user that does not have access to the file.

- You can list the contents of a single directory (not recursively) with *os.listdir("path/to/dir")*
- You can use os.walk("starting path") to recursively list files. Each iteration returns a tuple with three elements:
    - A string containing the current directory
    - A list of the directories in that directory
    - A list of the files in that directory
- Example: *for currentdir, list_of_dirs, list_of_files in os.walk("/"): #Some code block*

Python's gzip and zlib Modules

Python has default modules gzip and zlib to deal with gzipped files.

gzip.open() will open compressed files. In Python 3, it defaults to 'rb' mode but log files should be opened in text mode ('rt'). gzip objects support *the .read(), .readlines(), .write(), .writelines(), .seek(),* and *.tell()* methods the same as the built-in *open()* function does. Example: the following will read the first 40 characters of the gzipped log file:

```
file_handle = gzip.open("/var/log/syslog.2.gz", "rt")
first40 = file_handle.read(40)
```

zlib.decompress() will work on bytes, not files

zlib.compress() likewise works on bytes, not files

Regular Expressions

Python's re module implements regular expressions. It contains the following functions:

Function	Description
match()	Start at the beginning of data searching for pattern. Returns an object that stores the data. The object returned supports the .group() method to access capture group data. .group() returns the whole result, regardless of defined capture groups. .group(1) returns just the first capture group (numbering starts at 1). .group("name") will return a capture group with explicit group name of "name").
search()	Match pattern anywhere in the data. Returns an object that stores the data. The object returned supports the .group() method to access capture group data. .group() returns the whole result, regardless of defined capture groups. .group(1) returns just the first capture group (numbering starts at 1). .group("name") will return a capture group with explicit group name of "name").
findall('<expression', data,[optional_modifier])	Find all occurrences of the expression in the data. Returns a list containing each match as an item. The expression can be a regular string, a raw string (denoted by an r before the string) or a byte string (denoted by a b before the string). If desired, a third argument can be provided to modify the behavior. Examples:
	re.IGNORECASE will make the search case insensitive. Can alternatively just add (?i) to the beginning of the regular expression string.
	re.MULTILINE will make ^ and $ anchors apply to each new line character, not just the first line. Can alternatively just add (?m) to the beginning of the regular expression string.
	Re.DOTALL will make . match newlines also, since normally the . does not match newline character. This should always be used for searches within binary data. Can alternatively just add (?s) to the beginning of the regular expression string.

Python regular expression special characters:

Character	**Meaning**
.	Wildcard for any character
?	Previous character is optional
+	One of more of the previous character (greedy, match as much as possible)
+?	One or more of the previous character (stop as soon as the match is made)
*	Zero or more of the previous character (greedy, match as much as possible)
*?	Zero or more of the previous character (stop as soon as the match is made)
{x]	Match exactly x copies of the previous character
{x:y}	Match between x and y copies of the previous character

Character	Meaning
\w	Any text character (a-z, A-Z, 0-9, and _)
\W	Opposite of \w (only non-text characters)
\d	Matches digits (0-9)
\D	Opposite of \d
\s	Matches any white-space character (space, tab, newlines)
\S	Opposite of \s
[set of chars]	Define your own set of characters to match against one character. If the character is in the set you define, it matches.
[^set of char]	Caret in first position negates the set, so matches anything except was is listed.
\b	Border of a word character, the transition of \w to a \W or vice versa.
^	Match must be at the start of the string
$	Match must be at the end of the string
\	Escpapes special characters (\. means to search for a literal period)
|	A logical OR, Example a|b matches on an a or b
(<expression>)	Enclosing an expression in parentheses makes it a capture group. Only items inside a capture group are returned, but the rest of the expression must also match (it simply is not returned in the result). Capture groups are numbered beginning a 1, not zero.
(?P<groupname><expression>)	Creates a capture group with an explicit name, rather than the automatic numbering. Not that in this case, the first set of brackets is a literal syntactic requirement, example: *>>> a = re.search('(?P<test1>a..)de',"abcdefgabc")* *>>> a.group('test1')* *'abc'*
(?:<expression>)	If ?: is placed within a parenthetical group, it indicates that the grouping is for ordering only and is not a capture group. Example: to capture string dates from 0 to 31, use '(?:0[1-9]|[1-2][0-9]|3[0-1])'
(?i)	Make expression case insensitive.
(?s)	Make . match newlines also, since normally the . does not match newline character (always use for searching within binary data)
(?m)	Make ^ and $ anchors apply to each new line character, not just the first line (use for Multiline searches)
(?P<groupname>)	Makes a back reference to the capture group named groupname
\<number>	Makes a back reference to the capture group numbered <number>

- Since regular expressions are also python strings, the \ character is interpreted both by the Python string and by the regular expression, since it has spelling meaning for both. To indicate that a string is "raw" and should not process the string with the Python string engine by putting r at the beginning of the string. You can also do a b in front of the string to make it a byte string. Finally, in Python 3 only, you can do rb at the beginning of the string to denote a raw, byte string.

Sets

Sets are like lists, but each element is unique. Sets contain immutable objects, so they cannot contain lists or dictionaries. They are denoted with {} like dictionaries, so {1:2} is a dictionary and {1,2} is a set.

Set Method	Description
.add(<item>)	Adds one item
.update(<list or set>)	Add everything from another set or list
.remove(<item>)	Remove a single item from a list
.difference({set})	Given a second set as an argument, it returns a new set with the items in the original set but not in the set provided as an argument.
.difference_update({set})	Same as .difference({set}) but modifies original set instead of returning a new set (it removes any items in the second set from the original set)
.union({set})	Adds the original set and the set passed as an argument together into a new set
.issubset({set})	Returns True if all the items in original set are in the set you pass to the method as an argument
.issuperset({set})	Returns True if all the items in the set passed as an argument are contained in the original set
.isdisjoint({set})	Returns true if the original set and the set passed as an argument have no items in common
.intersection({set})	Returns items that are in both sets and puts them in a new set
.intersection_update({set})	Same as .intersection({set}) but modifies original set instead of returning a new set
.symmetric_difference({set})	Removes the intersection of the two sets and then returns all remaining items from both sets into a new set
.symmetric_difference_update({set})	Same as .symmetric_difference({set}) but modifies original set instead of returning a new set

Mathematical Operators with Sets

Operator	Behavior
^	Symmetric_differnce
&	Intersection
|	Union
-	Difference

Like lists and dictionaries, assigning a set to a variable creates a link to the original set, not a new set. To copy a set, use the set() function:

b = set(a)

Scapy

To import scapy, use

*from scapy.all import **

Scapy Functions

Function	Description
rdpcap(filename)	Reads a pcap file into a scapy.plist.PacketList data structure
wrpcap(filename, packetlist)	Writes a PacketList to a file
sniff()	Used to capture packets and return a PacketList object
sniff(offline="file.pcap")	Reads a pcap and returns a PaketList object
sniff(prn=function_name)	prn= argument specifies a callback function that is called for each packet returned by sniff. An example is lfilter, which provides the ability to filter packets based on specified criteria (sniff has a filter="BPF" argument as well but it has many OS dependencies that are often not met).

scapy.plist.PacketList Object Methods

Method	Description
.sessions()	Follows TCP streams, produces a dictionary with a key of "protocol srcip:srcport > dstip:dstport" and a value that is a scapy.plist.PacketList with all the associated packets in it.

PacketList objects are lists of packets. Packet objects have the .haslayer(<layer>) method, which returns True if that layer is present in that packet. Layer names are case sensitive and include Ether, IP, TCP, UDP, DNS, and Raw. Layer names are passed not in quotes, e.g. *packet_one.haslayer(TCP)*

Each layer has fields and the fields are addressed with a dot notation, for example *packet_one[TCP].sport* for the Source Port in the TCP layer. You can view the fields in any layer with *ls(<layer>)*. If a field name is unique, you can skip stating the layer and access just the field name, such as *packet_one.load* instead of *packet_one[Raw].load* since the field "load" only exists at the Raw layer. If the field name is not unique, scapy will return the first field it encounters with that name. Each packet also has a *.time* attribute that records the epoch time when the packet was captured.

struct Module

struct.unpack(<pattern_string>,<byte_data>) converts byte data into other types

struct.pack(<pattern_string>, <data>) converts int's and strings into binary data

the <pattern_string> describes how to pack or unpack the data, as shown in this chart:

Format	Type	Standard Size
x	N/A (means to ignore the byte)	Length specified before x
c	String of length 1	1 byte
b	Integer -128 to 127 (signed char)	1 byte
B	Integer 0 - 255 (unsigned char)	1 byte
?	Bool	1 byte
h	Integer (short)	2 bytes
H	Integer (unsigned short)	2 bytes
i	Integer (signed)	4 bytes
I (capital "eye")	Integer (unsigned)	4 bytes
l (lowercase "L")	Integer (signed long)	4 bytes
L	Integer (unsigned long)	4 bytes
f	Float	4 bytes
d	Float (double)	8 bytes
s	string	Length specified before s
!	Interpret data as big endian (used for network traffic data)	
>	Interpret data as big endian	
<	Interpret data as little endian	
=	Use sys.byteorder to determine endianess	
@	Use sys.byteorder to determine endianess	

Examples:

>>> struct.unpack("BB",b"\xff\x00")

(255, 0)

>>> struct.pack('<h', -5)

b'\xfb\xff' (Python 3 result)

'\xfb\xff' (Python 2 result)

struct.unpack("!6s6sH", data[:14]) to unpack Ethernet header

PIL Module

Originally called PIL (Python Image Library) but now called pillow; however, the package name pil is still used to avoid backward compatibility issues.

To install and import PIL:

pip install pil

from PIL import Image

Using an Image object:

imagedata=image.open("picture.jpg")

imagedata.show()

To read a picture carved out of a data stream, use BytesIO in Python 3 or StreamIO in Python 2

Python 3 Example:

from io import BytesIO

img = re.findall(r'\xff\xd8.\xff\xd9',raw_data, re.DOTALL) [0]*

Image.open(BytesIO(img)).show()

To get a dictionary of EXIF tag integers mapped to their meaning use:

from PIL.ExifTags import TAGS

Then use *TAGS.get(<integer>)* to look up an EXIF tag number and get its string meaning back. Can specify a second, optional argument to *TAGS.get()* that provides a string to return if the key is not found in the dictionary.

If you point a new variable to a variable holding an Image, it creates a pointer to the original, not a new object (like lists). To create a new image, use:

copy = Image.Image.copy(original)

PIL.Image methods

Method	Description
Image.open(<filename>)	Open a file and create an Image object (not a method of an Image object, but used to instantiate an ImageObject)
.show()	Use default viewe to display the image
.thumbnail((Width,Height),Method)	Reduces the size of the image to a maximum size specified by the tuple in the first argument, using the method specified in the next argument. Methods include Image.NEAREST, Image.BILINEAR, Image.BICUBIC, Image.ANTIALIAS and others. Preserves the image aspect ratio. Modifies the original image, does not produce a new copy.
.resize((Width,Height),Method)	Enlarge or reduce the size of image to the size provided in the tuple supplied as first argument, ignoring original aspect ratio. Does not modify original image, but returns a copy of it.
.size (attribute, not a method)	A tuple that provides the size of the image (width,height)
.crop((left,upper,right,lower))	Returns a cropped copy of the image. The argument is a tuple defining the area to be cropped.
.rotate(degrees)	Returns a copy of the image rotated the specified number of degrees, does not alter original image.
.save()	Saves the image to disk
_getexif()	Returns a dictionary describing the metadata about the image. The keys are integers designating the tag type as per the EXIF standard. Values are the associated data for that tag.

sqlite3 Module

To import this module, use: *import sqlite3*

To connect to a database file and access it through a variable named db:

db = sqlite3.connect("filename")

You can then make SQL queries to the database with the *.execute()* method as seen here:

list(db.execute("select name from sqlite_master where type='table';"))

.execute() method returns an iterable object that can be converted into a list to view all contents if desired or iterated with a *for* loop.

python-registry Module

pip install python-registry

from Registry import Registry

handle = Registry.Registry("path/to/file") to open a registry hive

regkey = handle.open("path\to\key") opens a specific key

Key Methods

Method Name	Description
.path()	Returns a string with the full path to the key starting with "ROOT\" and ending with the name of the key itself
.value(<value_name>)	Returns a single object of type Registry.value
.values()	Returns a list of all the values for the key
.subkey(<subkey_name>)	Returns a single key object for the specified subkey
.subkeys()	Returns a list of all the subkeys

Value objects also have methods

Method Name	Description
.name()	Returns the name of the value
.value()	Returns the value data associated with the value
.value_type_str()	Returns the type, such as REG_DWORD, REG_SZ, REG_BINARY, REG_QWORD, REG_NONE, etc.

To retrieve a list of subkey names for a key,

reg_key = reg_hive.open("Microsoft\Windows\CurrentVersion")

list(map(lambda x:x.name() , reg_key.subkeys()))

Registry Date formats

Format	
REG_BINARY	Eight values of 2-bytes each, representing year, month, day (a number representing the day of the week starting with Sunday), date, hr, min, sec, microsecond.
REG_DWORD	Linux timestamp integer recording the number of seconds since Epoch.

Generators

Placing a yield statement in a function makes it a generator, which pauses its execution, returns a value, and awaits a .__next__() call to resume execution and return the value indicated by the next yield statement.

requests Module

Import with: *import requests*

Can make get requests:

webdata = requests.get("http://www.sans.org")

Or post requests:

formdata = {'username':'admin','password':'ninja'}

webdata = requests.post("http://www.sans.org", formdata)

Both of these return a response object. Can access several different attributes of response objects:

response Object Attribute	Description
.content	The content of the web response
.headers	The headers returned
.status_code	The HTTP status code integer
.reason	The text description associated with the status code

Alternatively, you can create a session, which is like creating a browser that remembers setting, such as User-Agent, and maintains state via cookies

browser = requests.session()

browser.headers attribute displays a dictionary with the various header options like Accept-Encoding, User-Agent, etc. These can be changed as desired by simply changing this dictionary.

You can then call *browser.get(<url>)* and *browser.post(<url>,<postdata>))* to make requests from the customized browser object. The responses will still be request objects just as they were when using *requests.get(<url>)* and *requests.post(<url>,<postdata>)*

You can configure a proxy with *browser.proxies[<protocol>] = <url>:<port>* such as

browser.proxies['http'] = 'http://127.0.0.1:8080'

You can also use *browser.cookies* attribute to view the request.cookies.RequestCookieJar object, which is a special type of dictionary. Calling *browser.cookies.keys()* will provide a list of the cookies. You can use *browser.cookies[<cookie_name]* to view the value of a cookie.

browser.cookies.clear will clear all cookies. *browser.cookies.clear(domain=<domain_string>)* will clear cookies for the specified domain.

socket Module

To import, use: *import socket*

>>> socket.gethostbyname("www.sans.org")

'45.60.35.34'

>>> socket.gethostbyaddr("8.8.8.8")

('google-public-dns-a.google.com', [], ['8.8.8.8'])

(the above result is a tuple with hostname, list of aliases, and a list of addresses)

To create a socket, use:

<variable_name> = socket.socket(<IP type>, <Protocol>)

Where <IP type> is:

- socket.AF_INET for IPv4 (default if nothing specified)
- socket.AF_INET6 for IPv6

And <Protocol> is:

- socket.SOCK_DGRAM for UDP
- socket.SOCK_STREAM for TCP (default if nothing specified)

Example:

udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM) creates an IPv4, UDP socket

tcp_socket = socket.socket() creates an IPv4, TCP socket by default

After creating the socket object (TCP or UDP), if the object will be a server, bind it to a port with:

udp_socket.bind((“10.10.10.10”,9000))

(the argument is a tuple with a string for the IP and an integer for the port)

To send and receive data to a UDP socket, use *.sendto()* and *.recvfrom()* methods

udp_socket.sendto(“HELLO”,(“10.0.1.1”,3000)) returns number of bytes sent

udp_socket.recvfrom(<number_of_bytes>) returns (<data_received>,(<IP_addr>,port))

To create an outbound session to a TCP socket use *.connect()* method

tcp_socket.connect((<dest_ip>, <dest_port>)) this handles the 3-way handshake

To accept inbound connections to a TCP socket:

tcp_socket.bind((<ip>,<port>))

tcp_socket.listen(<max_num_of_connections>) (the port will show as listening at this point)

connection,remote = tcp_socket.accept()

.accept() will return a connection object and a tuple with the remote IP and port.

From that point, you can interact through the connection object with *.send()*, *.recv()* and *.close()*

try/except/else/finally Blocks

```
try:
   #block to try
except <specific_error_name>:
   #block for that error
except Exception as e:
   #block that can include the name of the exception as the variable e
else:
   #block to do if there is no exception
finally:
   #block to do at the end whether there was an exception or not (usually for clean up)
```

Try something until it works:

```
while True:
   try:
      #something to try
   except:
      continue
   else:
      break
```

Try various things until one of them works:

```
while not done:
  for thing_to_try in [list_of_options]:
    try:
      #try the first thing
    except:
      continue
    else:
      done = True
      break
```

subprocess Module

Can be used to start a new process, provide it input and capture the output:

```
processhandle = subprocess.Popen("some command",
      shell = True,
      stdout = subprocess.PIPE,
      stderr = subprocess.PIPE,
      stdin = subprocess.PIPE)
results = processhandle.stdout.read()
errors = processhandle.stderr.read()
```

Can use *processhandle.wait()* to cause your program to pause until the subprocess completes. It returns an integer exit code to show the status once the process terminates. However, if the subprocess generates a lot of output, the output buffer may fill and cause a hang.

Instead, you can use *processhandle.communicate()* which will read the subprocess.PIPE repeatedly until the subprocess is finished executing. It then returns a tuple with two, separate byte strings. The first contains all the stdout and the second contains all the stderr from the subprocess.

select Module

select.select([list_of_sockets], [list_of_sockets], [list_of_sockets])

The sockets in the list are each checked. The first list is checked to see if the sockets have data ready for you to receive. The second list is checked to see if they are ready for you to send data. The third list checks to see if they are in an error condition.