

# Python Manual

*A concise explanation of essential coding concepts*

by Windmaran Network

## *Level-1:*

Assignments, variables, expressions,  
constants, arithmetic operators

## *Level-2:*

If statements, if-else statements,  
comparison operators, logical operators

## *Level-3:*

Loops, lists, functions

Copyright © 2010-2021 Windmaran Network. All right reserved. No parts of this document, cover design, and icons, may be reproduced or translated in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

Link: [www.windmaran.com/wm/pdf/python-manual.pdf](http://www.windmaran.com/wm/pdf/python-manual.pdf)

This edition published May 2021  
Windmaran Network  
[www.windmaran.com](http://www.windmaran.com)

## Python Manual by Windmaran Network

<b>Level 1</b>	5-22	Assignments, variables, expressions, constants, arithmetic operators
<b>Level 2</b>	23-29	If statements, if-else statements, comparison and logical operators
<b>Level 3</b>	30-58	Loops, lists, functions
<b>Appendixes</b>	59-60	Appendix-1 Run Python programs using Thonny
	61	Appendix-2 Run Python online in browser

```
import random # Mainly used for random number
import time # Mainly used for time.sleep()
import math # More complex maths
import os # Working with files?
import pygame # Creating windows (games)
import turtle # Creating windows
import re # Regex
import numpy # Dealing with arrays
import socket # Netorking, communicating between computers
import this # Poem
import keyboard # Receiving keyboard inputs
import mouse # Detecting mouse clicks
import webbrowser # Opening websites
import pprint # Formating text
import pyautogui # Automation
```

## About this tutorial

This is a tutorial on the Python programming language. It is intended for students that have **completed the study of Python Primer!**

This manual is divided into three sections (Level-1, Level-2, and Level-3) – which corresponds to three levels of coding examples presented on the Windmaran Python webpage. The following concepts are covered:

Level-1: Assignments, variables, expressions, constants, arithmetic operators

Level-2: If statements, if-else statements, comparison & logical operators

Level-3: Loops, functions, lists

The text is divided into learning modules. Each module presents learning material in a logical, sequential order.

## Appendix 1 and 2

Appendix 1 introduces Thonny - an integrated development environment for Python that is designed for beginners

Appendix 2 explains how to run Python code online in your browser

## Follow the Seven Steps Guide

In order to study Python in an ordered fashion, it is necessary to follow the instructions as outlined in the Seven Steps Guide. Begin at the step no 4:

Link: [www.windmaran.com/wm/wm-aux-pages/seven-steps.php](http://www.windmaran.com/wm/wm-aux-pages/seven-steps.php)

## Other Python Manuals

### *Python Quick Guide*

A brief overview of Python features:

Link: [www.windmaran.com/wm/pdf/python-quick-reference.pdf](http://www.windmaran.com/wm/pdf/python-quick-reference.pdf)

## Recommended links to study Python

W3schools: W3schools.com is among the best online tutorials to learn Python

link: <https://www.w3schools.com/python/>

tutorialspoint: tutorial for Python beginners

link: <https://www.tutorialspoint.com/python/index.htm>

Python for kids: a playful introduction to programming

link: <https://epdf.pub/python-for-kids-a-playful-introduction-to-programming.html>

The official home of the Python Programming Language

link: <https://www.python.org/>

Learn to code with Thonny — a Python IDE for beginners

<https://thonny.org/>

Learn to code in python on repl.it

<https://repl.it/talk/learn/Learn-To-Code-In-Python/7484>

Learn Python online: Python tutorials for developers of all skill levels

<https://realpython.com>

Hands-on Python 3 tutorial by Dr. Andrew N. Harrington

<http://anh.cs.luc.edu/python/hands-on/3.1/>

# Level 1

## Module-1

*Variable names, Arithmetic expressions, Assignments, Modulo operator, print() function.*

### Variables in Python

Let's talk about variables. A Python variable name can be a single letter or a descriptive name such as: `accountNumber`, `totalAmount`, etc.

To give the names to variables so-called "camelCase style" is used. The camelCase is a naming convention in which each word within a compound word is capitalized except for the first word. Software developers often use camelCase when writing source code. Examples of variable names that follow this convention:

```
keepGoing  
acctNumber  
totalAmt  
creditCardNumber  
postalCode
```

Python rules for naming variables:

- Variable names cannot contain spaces.
- A variable starts with a letter A to Z or a to z or an underscore (`_`) followed by zero or more letters, underscores and digits (0 to 9)
- Python does not allow punctuation characters such as `@`, `$`, and `%` within variables
- Variable names are case-sensitive.

## Creating Variables

Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it:

```
a = 100
```

This means: variable (a) has been created and assigned value 100.

## End of Statements

To end a statement in Python, you do not have to type in a semicolon or other special character; you simply press Enter.

## Semicolon as a statement separator

Python allows more than one statement on a single line. A semicolon ';' is used to separate multiple statements on the same line. For instance, three statements can be written as:

```
a = 1; b = 2; print(a+b)
```

To the Python interpreter, this would be identical to this set of statements:

```
a = 1
b = 2
print(a+b)
```

## Multi-line statements

You cannot split a statement into multiple lines in Python by pressing Enter . Instead, use the backslash ( \ ) to indicate that a statement is continued on the next line. For example this statement:

```
a = 5 + 6 + 7
```

could be re-written as the following: (note the backslash characters at the end of lines):

```
a = 5  \
    + 6  \
    + 7
```

## Function print()

The print() function prints the specified message to the screen. The message can be a string, or any other variable, the variable will be converted into a string before written to the screen.

Let's explore a simple use of `print()` function. Note that the strings are enclosed in double quotes:

*Example 1:*

```
print ("Program begins")

a = 4
print(a)

b = 2
print(b)

print("The End")
```

The program will output:

```
Program begins
4
2
The End
```

Use comma to separate variables: if we want to print a combination of several variables and text string we must separate each element by a comma character.

*Example 2:*

```
print ("Program begins")

a = 4
b = 2
print ("value of a:", a, "value of b:", b);

print("The End")
```

The program will output:

```
Program begins
value of a: 4 value of b: 2
The End
```

### **How to print without line break**

In Python we can use the `'end='` parameter in the `print` function. By default the `end` parameter uses `'\n'` which is what creates the newline, but we can set a different value for the `end` parameter. This tells it to end the string with a character of our choosing rather than ending with a newline.

Let's compare the next two examples:

This example uses a default version of the `print()` function which automatically adds a newline character thus breaking the output line.

*Example 3:*

```
a = 1000
b = 2000
c = 3000
print (a)
print (b)
print (c)
```

The program will output:

```
1000
2000
3000
```

In contrast, this example ends each line with a comma character (except the last `print()` statement):

*Example 4:*

```
a = 1000
b = 2000
c = 3000
print (a, end=",")
print (b, end=",")
print (c)
```

The program will output:

```
1000,2000,3000
```

### **Arithmetic Operator(s)**

Python uses the following symbols (operators) in calculations:

```
+ is addition
- is subtraction
* is multiplication
/ is division
```

The order of operations (also called precedence rules) defines how an expression is to be evaluated. Multiplication is granted higher precedence than addition. Thus, the expression `2 + 3 * 4` is interpreted to have the value `2 + (3 * 4) = 14`



So the rule is that unless we use brackets the multiplication and division precedes addition and subtraction.

Read carefully the next 2 examples that demonstrate the syntax of Python arithmetic expressions:

*Example 5:*

```
a = 4
b = 2
x = (a + b) * (a - b)

print("value of x: ", x)
```

The program will output:    Value of x: 12

*Example 6:*

```
a = 4
b = a/2
c = (a + b) * (a - b)
d = a + b * a - b
x = c - d

print("value of x: ", x)
```

The program will output:    Value of x: 2

### **Modulo operator**

The modulo operator (%) returns the division remainder. For instance, when we divide 10/3 the remainder is 1. When we divide 10/5 the remainder is 0. As you can see the modulo operator is used to find if a number is divisible by another number.

*Example 7:*

```
a = 17
b = 5
c = a % b
print("value of c is ", c)
```

The program will output:    Value of x: 2

## Module-2

### *Comments, Constants*

#### **Python Comments**

Python comments are used to explain a code to make it more readable. In Python, we use the hash '#' symbol to start writing a comment.

```
# This is a comment line
```

Furthermore, comments can be placed after the end of the statement as the next line shows:

```
secDay = 24 * 3600 # Seconds in a day calculation
```

More examples of Python comments:

```
# This is a pretty good example  
# of how you can spread comments  
# over multiple lines in Python
```

#### **Python Constants**

Often we want to work with a variable that has a fixed value – a constant. But, you cannot declare a variable or value as constant in Python. Just don't change it! Considering this - the “constant definition” might look like the following:

```
PI = 3.141592653589
```

Many style guides suggest capitalizing constant names. It is a very good practice that is used in this manual.

Other Examples:

```
MOON_RADIUS = 1737.4      # in kilometers  
EARTH_RADIUS = 6378       # km  
GOLDEN_RATIO = 1.61803399  
RATIO_FEET_TO_METERS = 3.281
```

## Module-3

### *Boolean Data Type, Strings, String Concatenation*

#### **Boolean Data Type**

Boolean is a data type in Python. The Boolean data type can be one of two values, either True or False. Some languages use true and false all lowercase, but python uses the capitalized variation so you need to make sure that you always capitalize those words. We use Booleans in programming to make comparisons and to control the flow of the program. (More on the use of Boolean data types in Level-2.)

#### *Example 1*

```
a = True    # capitalized
b = False   # capitalized

print("Value of a = ", a)
print("Value of b = ", b)
```

Output :

```
Value of a =  True
Value of b =  False
```

#### **Strings**

A Python string stores a series of characters like "Los Angeles". A string can be any text inside double or single quotes.

In our manual we will use exclusively strings enclosed by double quotes:

```
myName = "Jonathan"
myCity = "New York"
```

#### **String Concatenation**

The process of combining strings is called "concatenation". In Python, you combine ("concatenate") strings with a (+) operator like this:

```
a = "alpha"
b = "bet"
# let's concatenate a and b
c = a + b          # value of c is "alphabet"
```

Here is another instance of concatenation. Note the use of a single character string " " used to create a space between the firstName and familyName

```
firstName = "John"
familyName = "Harris"
# value of fullName will be "John Harris"
# (see the extra space)
fullName = firstName + " " + familyName
```

## Empty String

Suppose we see in some Python code this assignment

```
txt = "" # assign an empty string
```

Yes, this is how to define and assign an empty string

## How to assign and output strings

### Example 2

```
city = "Los Angeles"
state = "California"

# the string is separated by a comma and a single space
myCity = city + ", " + state

print("I live in", myCity)
```

### Example 3

```
name1 = "John"
name2 = "David"

print("My brothers are", name1, "and", name2)
```

### Example 4

```
txt = "" # an empty string
txt = txt + "abc"
txt = txt + "def"

print("Value of txt is:", txt) # shows: "abcdef"
```

**Adding extra space**

You might have already notice that Python adds an **extra space** when printing several parameters

*Example 5*

```
a = 10
b = "alpha"
c = "epsilon"
d = 20
# extra space added to separate output elements:
print ("a:",a,"b:",b,"c:",c,"d:",d)
```

The program will output:    a: 10 b: alpha c: epsilon d: 20

## Module-4

*Round function, Type Conversion, Number-to-String Conversion, string padding*

**Round function**

The round() function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.

The default number of decimals is 0, meaning that the function will return the nearest integer. For example

```
x = round(5.76) # the value of x will be 6
```

Syntax:

```
round(number, digits)
```

*Example 1:*

```
pi = 3.14159265359
piRound = round(pi,2)
print("value of pi is ", pi)
print("value of piRound is ", piRound)
```

The program will output:

```
Value of pi is 3.14159265359
Value of piRound is 3.14
```

Example 2:

```
a = 7.543543543
b = round(a)
c = round(a,1)
d = round(a,2)
e = round(a,3)

print("a=", a)
print("b=", b)
print("c=", c)
print("d=", d)
print("e=", e)
```

The program will output:

```
a= 7.543543543
b= 8
c= 7.5
d= 7.54
e= 7.544
```

## Python Numbers and Type Conversion

There are two numeric types in Python (there are also complex numbers which we will not discuss):

int  
float

**int** - they are called just integers or ints, are positive or negative whole numbers with no decimal point.

**float** - they are called floats, they represent real numbers and are written with a decimal point dividing the integer and decimal parts.

Example:

```
x = -11      # integer
y = 3.14     # float
```

You can convert from one type to another with the `int()`, and `float()` functions.

Important:

`int(x)` will convert (x) to an integer and will get rid of the decimal part by truncating it (not rounding!). For example, `int(2.95)` will be converted to integer number 2

**Examples:**

```
x = 1    # assign int number
y = 2.8  # assign float number

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)
c = int(12.95)  # 12.95 will be converted to 12
d = int(5/3)    # 5/3   will be converted to  1

# convert from int to float
n = float(10)   # will be converted to 10.0
m = float(1+1+1) # will be converted to  3.0
```

**Converting Numbers to Strings**

Python converts numbers to strings through using the `str()` function. We'll pass either a number or a variable into the `str()` function, and, then that numeric value will be converted into a string value.

Let's first look at converting integers. To convert the integer 12 to a string value, you just pass 12 into the `str()` function:

```
str(12)  # converts to string "12"
```

We also can use `str()` function to convert float numbers or float variables into a string:

```
str(1/2)  # converts to string "0.5"
```

**Pad String with Spaces using function `rjust()`**

`rjust()` function in python returns the string with padding done on the left end of the string to the specified length.

```
xx = "xyz"
# return string 10 character long (padded with 7 leading spaces)
xx.rjust(10)

yy = 123
# return string 5 character long (padded with 2 leading spaces)
str(yy).rjust(5) # firstly, convert yy to a string
```

*Example 1:*

```
string = "my cat"
width = 9
print("0123456789")
print(string)
# print right justified string
print(string.rjust(width))
```

The program will output:

```
0123456789
my cat
   my cat
```

*Example 2:*

```
a = 6797.543543543
b = 48.987487
c = 7.387652

aStr = str(round(a,2))
bStr = str(round(b,2))
cStr = str(round(c,2))

print("aStr=", aStr)
print("bStr=", bStr)
print("cStr=", cStr)

print("aStr=", aStr.rjust(8))
print("bStr=", bStr.rjust(8))
print("cStr=", cStr.rjust(8))
```

The program will output (note the difference between before and after rjust application) the following:

```
aStr= 6797.54
bStr= 48.99
cStr= 7.39
aStr=  6797.54
bStr=   48.99
cStr=    7.39
```

aaa



## Module-5

*Import statement, random functions: random integer, random float*

### The import statement

Python gains access to the code in another module by the process of importing it. In a Python file, this will be declared at the top of the code. For example Python has a built-in module that you can use to make random numbers. In order to access random function declare this statement

```
import random
```

### Random Integer Function

Python randint() is a function of the random module in Python. The random module gives access to various useful functions and one of them being able to generate random numbers.

Syntax : random.randint(start, end)

The function returns a random integer within the given range as parameters. Note the "import random" statement at the top.

*Example 1:*

```
import random
i1 = random.randint(0,1)
i2 = random.randint(1,100)
print("Value of i1:", i1)
print("Value of i2:", i2)
```

The program will output:

```
Value of i1: 1
Value of i2: 48
```

### Random Float Function

Python random() function is used to return a floating-point random number between range <0,1> , 0 (inclusive) and 1 (exclusive).

Syntax : random.random()

The next example should be sufficient to explain how the function works.

### Example 2

```
import random
rand1 = random.random()
rand2 = random.random()
rand3 = random.random()
print("rand1=", rand1)
print("rand2=", rand2)
print("rand3=", rand3)
```

Output (it is different for each script execution – the following is just one of the particular runs):

```
rand1= 0.25173679172686325
rand2= 0.9133861116499714
rand3= 0.18862906774115207
```

## Module-6

### *Math functions list*

Here are the mathematical functions that we need to know:

Math Function	Action	Examples
pow()	pow(x, y) returns the value of x to the power of y	pow(8, 2) # returns 64
math.sqrt()	math.sqrt(x) returns the square root of x	math.sqrt(25) # returns 5
abs()	abs(x) returns the absolute (positive) value of x	abs(-9.7) # returns 9.7
math.ceil()	math.ceil(x) returns the value of x rounded up to its nearest integer	math.ceil(4.991) # returns 5 math.ceil(10.5) # returns 11 math.ceil(4.001) # returns 5
math.floor()	math.floor(x) returns the value of x rounded down to its nearest integer	math.floor(4.01) # returns 4 math.floor(4.99) # returns 4

The following example shows the math functions in action (note the "import math" statement at the top):

*Example 1*

```
import math

# pow() function
print("pow()")
res1 = math.pow(2, 4)
res2 = math.pow(5, 3)
res3 = math.pow(10, 3)
print("res1 = ", res1)
print("res2 = ", res2)
print("res3 = ", res3)
print()

# sqrt() function
print("sqrt()")
res1 = math.sqrt(2)
res2 = math.sqrt(3)
res3 = math.sqrt(1000)
print("res1 = ", res1)
print("res2 = ", res2)
print("res3 = ", res3)
print()

# ceil() function
print("ceil()")
res1 = math.ceil(100.001)
res2 = math.ceil(-1.001)
res3 = math.ceil(-1.99)
print("res1 = ", res1)
print("res2 = ", res2)
print("res3 = ", res3)
print()

# abs() function
print("abs()")
res1 = math.abs(-1.55)
res2 = math.abs(-987.123)
res3 = math.abs(1000.123)
print("res1 = ", res1)
print("res2 = ", res2)
print("res3 = ", res3)
print()
```

Output:

```
pow()
res1 = 16.0
res2 = 125.0
res3 = 1000.0

sqrt()
res1 = 1.4142135623730951
res2 = 1.7320508075688772
res3 = 31.622776601683793

ceil()
res1 = 101
res2 = -1
res3 = -1

abs()
res1 = 1.55
res2 = 987.123
res3 = 1000.123
```

### **A note on terminology (function vs. method)**

In this manual, we talked about `(rjust)` or `(randint)` as functions. But in fact, these are methods. However, since we are not dealing with object-oriented features we will stay with this rather 'imprecise' terminology. Keep this distinction in mind in case if you are going to study Python object-oriented features.

## Module-7

### *More examples to review*

We've covered a lot of ground; now it is time to review what we've learned. Read these examples carefully:

#### *Example 1*

```
a = 55
b = 7

c = a % b
d = int(a/b)
x = d * b + c

# value of x must be equal to value of a
print("x:", x, "c:", c, "d:", d)
```

Output :

```
x: 55 c: 6 d: 7
```

#### *Example 2*

```
MARS_RADIUS = 3396.2      # in kilometres
PI           = 3.1415927
MILES_RATIO  = 1.609344    # one mile is equal to 1.609344 kilometres

print("Mars Circumference Calculation")

marsCirc = 2 * MARS_RADIUS * PI
print("The equatorial circumference of Mars is", str(round(marsCirc,3)), "km")
print("The equatorial circumference of Mars is",
      str(round(marsCirc/MILES_RATIO,3)), "miles")

print("The End")
```

Output :

```
Mars Circumference Calculation
The equatorial circumference of Mars is 21338.954 km
The equatorial circumference of Mars is 13259.411 miles
The End
```

*Example 3*

```
cc = 1000
a = cc/3.1234
b = cc/47.754
c = cc/278.96705

aStr = str(round(a,3))
bStr = str(round(b,3))
cStr = str(round(c,3))

print("aStr=", aStr.rjust(8))
print("bStr=", bStr.rjust(8))
print("cStr=", cStr.rjust(8))
```

Output :

```
aStr=  320.164
bStr=   20.941
cStr=    3.585
```

The above example can be rewritten by combining (str) and (rjust) functions into one expression:

*Example 4*

```
cc = 1000
a = cc/3.1234
b = cc/47.754
c = cc/278.96705

# format by rounding to 3 decimals and right-justifying
print("a=", str(round(a,3)).rjust(8))
print("b=", str(round(b,3)).rjust(8))
print("c=", str(round(c,3)).rjust(8))
```

Output :

```
a=  320.164
b=   20.941
c=    3.585
```

This is the end of LEVEL-1; The next two activities are following:

- a) Explore the Python Level 1 Examples (Group A and B)  
Link: [www.windmaran.com/wm/python/py-main.php?px=L1A1](http://www.windmaran.com/wm/python/py-main.php?px=L1A1)
- b) Answer Python Online Level 1 Quiz questions  
Link: [www.windmaran.com/wm/python/py-quiz.php?px=L1Q1](http://www.windmaran.com/wm/python/py-quiz.php?px=L1Q1)

# Level 2

## Module-8

### *If statement, Indentation as a block marker*

We already know that the if statement is used to do something based on the condition. If the condition is true then the indented code is going to be processed. This means that Python relies on indentation (white-space at the beginning of a line) to define the scope in the code. In this manual, we'll use 4 spaces for indentation.

There is a mandatory colon after the end of the condition to give a visual indicator that a code block is following directly afterward.

To demonstrate the significance of proper indentation in Python let's compare the results of two examples:

#### *Example 1:*

```
a = 1
b = 1
if a == 0:
    a = a + 1
    b = b + 1
x = a + b
print("Value of x: ", x)
```

The program will output: Value of x: 2

Let's make a small "correction" – by removing the indentation of the statement:  
`b = b + 1` resulting in the different result!

*Example 2:*

```
a = 1
b = 1
if a == 0:
    a = a + 1
b = b + 1
x = a + b
print("Value of x: ", x)
```

The program will output: `Value of x: 2`

To summarize the rules about indentation in Python:

- a)  
Python strictly enforces the use of indentation to mark a block of code: all statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right
- b)  
Generally, four white-spaces are used for indentation.
- c)  
The block ends with the first un-indented line
- d)  
A colon is used to specify the start of a code block. In other words, the role of colon in Python is to give a visual indicator that a code block is following directly afterward.



## Module-9

### *If – else Statement, if-elif-else Statements*

The if-else statement executes a block of code if a specified condition is true. If the condition is false, another block of code is executed. Let's explore the following code:

*Example 1:*

```
a = 2
b = 2
if a > b:
    a = a * 2
    b = b * 2
else:
    a = a + 1
    b = b + 1

x = a + b
print("Value of x: ", x)
```

The program will output: Value of x: 6

Let's expand our understating of if-else syntax. Suppose we want to check for more than one condition. The syntax for such a case is following – note the new construct "elif" :

```
if condition1:
    code to be executed if this condition is true;
elif condition2:
    code to be executed if the first condition is false
    and this condition is true;
else:
    code to be executed if all conditions are false;
```

Here is a simple demonstration (note the use of mandatory colons):

*Example 2*

```
number = -1
msg = ""

if number > 0:
    msg = "Number is positive"
elif number < 0:
    msg = "Number is negative"
else:
    msg = "Number is 0"

print(msg)
```

The program will output: `Number is negative`

## Module-10

### *Nested if statements, comparison and logical operators*

When we put an 'if' statement inside another 'if' statement and possibly repeating that process multiple times we talk about nesting.

## Example 1

```
a = 10
b = 20
if b > a:
    if a == b:
        a = a - 1
        b = b - 1
    else:
        a = a + 1
        b = b + 1

x = a + b
print("Value of x: ", x)
```

## Python logical & comparison operators

Let's recall the comparison and logical operators that were described in Python Primer:

Expression	Python logical & comparison operator
Equals	==
Does not equal	!=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
And	and
Or	or

Explore the following example:

## Example 1

```
a = 0
b = a + 1

if a > 0 or b > 0:
    a = a + 1
    b = b + 1

if a > 0 and b > 0:
    a = a + 1
    b = b + 1

x = a + b
print("Value of x: ", x)
```

Output:

Value of x: 5

The previous example might look “tricky” – so, let’s take the previous Python code and insert tracking statement to explore how the execution proceeds:

*Example 3*

```
a = 0
b = a + 1

if a > 0 or b > 0:
    a = a + 1
    b = b + 1
    print("Trace1 a =", a, "b =", b)

if a > 0 and b > 0:
    a = a + 1
    b = b + 1
    print("Trace2 a =", a, "b =", b)

x = a + b
print("Value of x: ", x)
```

Output

```
Trace1 a = 1 b = 2
Trace2 a = 2 b = 3
Value of x: 5
```

## Module-11

### *Boolean variables and If-Else conditions*

We learned in the Level-1 section about Boolean data type. Boolean can have only two values, True or False (the first letter must be capitalized). Let’s explore how to use Boolean variables to control program flow using conditional statements. In the following example it should be obvious that the block after the ‘if’ statement will be executed as the condition is true.

*Example 1*

```
a = 0
b = True
if b:
    a = a + 1
x = a
print("value of x: ", x)
```

Output:

```
Value of x: 1
```

Let's slightly modify the previous code:

*Example 2*

```
a = 0
b = a >= 0 # the result of this comparison is True
if b:
    a = a + 1
x = a
print("value of x: ", x)
```

Output:

```
Value of x: 1
```

Let's continue with further modifications: notice in the next example the test of two Boolean values in the 'if' statement.

*Example 3*

```
a = 10
n = 10
i = 0

b1 = a == n
b2 = a > n

if b1 or b2:
    i = i + 1

x = i
print("value of x: ", x)
```

Output:

```
Value of x: 1
```

This is the end of LEVEL-2; The next two activities are following:

- a) Explore the Python Level 2 Examples (Group A and B)  
Link: [www.windmaran.com/wm/python/py-main.php?px=L2A1](http://www.windmaran.com/wm/python/py-main.php?px=L2A1)
- b) Answer Python Online Level 2 Quiz questions  
Link: [www.windmaran.com/wm/python/py-quiz.php?px=L2Q1](http://www.windmaran.com/wm/python/py-quiz.php?px=L2Q1)

# Level 3

## Module-12

### *While loop*

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. The syntax is very similar to the if statement - as seen below.

```
while condition:
    # execute code as long as condition is true
```

The while statement is the most basic loop to construct in Python. Let's compare two pieces of identical code in order to study 'while loop' behavior:

#### *Example 1*

```
n = 0
while n < 2:
    n = n + 1

x = n
print("value of x: ", x)
```

Let's modify the above Python code by adding a tracking statement. The output listing shows how the variables are assigned values during each iteration:

#### *Example 2*

```
n = 0
while n < 2:
    print("Track n: ", n)
    n = n + 1

x = n
print("value of x: ", x)
```

Output:

```
Track n:  0
Track n:  1
Value of x:  2
```

The next example prints all odd numbers between <1, 10>. Note that the 'if' statement that determines whether a number is odd or even (if the remainder of division by 2 is not zero then by definition the number is odd).

**Example 3**

```
n = 1
while n <= 10:
    if n%2 != 0:
        print(n, " is an odd number")
    n = n + 1
```

Output:

```
1  is an odd number
3  is an odd number
5  is an odd number
7  is an odd number
9  is an odd number
```

**Infinite Loops**

Working with while loops presents a challenge: a coding error can cause an infinite loop. Infinite loop, as the name suggests, is a loop that will keep running forever. If you accidentally make an infinite loop, it could crash your Python interpreter. Here is a very simple code that will run “forever”:

```
n = 1
while n > 0:
    print(n)
```

another instance of the infinite loop is presented below. Since the variable 'n' is not incremented during each cycle the while condition (n <= 10) will be permanently true!

```
n = 1
while n <= 10:
    if n%2 != 0:
        print(n, " is an odd number")
    === missing (n = n + 1) statement here ===
```

**Break Keyword**

The break keyword is used to break out the loop. In the next example the while loop ends when the condition (i > 5) is True

```
# end the loop if i is larger than 5
i = 1
while i < 10:
    print(i)
    if i > 5:
        break
    i = i + 1
```

## Module-13

### *For loop*

The for loop is more complex, but it's also the most commonly used loop. Frequently the for loops use "range" which is a Python function – see the example:

#### *Example 1*

```
n = 0
for i in range(0, 2, 1):
    n = n + 10

x = n
print("Value of x: ", x)
```

Looking at the above example you might think that (range) is part of the (for loop) syntax. It is not. Range() is a Python function which returns a sequence of integer numbers which in turn allows the for loop to iterate over. Since range creates a fixed length list the for loop runs for a fixed number of iteration.

Syntax of the range function is: range(start, stop, step)

start is an integer number specifying at which position to start

stop is an integer number specifying at which position to end.

step is an integer number specifying the incrementation.

It is very important to note that the sequence ends before the value (stop) is reached, in other words, range() function generate numbers up to, but not including (stop) number

#### examples

```
range (0, 5, 1) # generates numerical sequence: 0, 1, 2, 3, 4
range (2, 9, 3) # generates numerical sequence: 2, 5, 8
```

Furthermore, values (start) and (step) are optional. Default for (start) is 0 and the default for (step) is 1. This means that the following range definition:

```
range(0,10,1)
could be written as:
range(10) # generates numerical sequence 0,1,2,3,4,5,6,7,8,9
```

In the context of the for loop these two statements are identical

```
for i in range(0, 2, 1):
for i in range(2):
```



Here are examples that illustrate the for loops behavior:

*Example 2*

```
# Output odd numbers in interval <0,9>

limit = 9
for n in range (limit+1):
    if n%2 != 0:
        print(n, " is an odd number")
```

Output:

```
1  is an odd number
3  is an odd number
5  is an odd number
7  is an odd number
9  is an odd number
```

Let's modify the above code – just iterate using odd numbers sequence (it will produce an identical output):

*Example 3*

```
# Output odd numbers in interval <0,9>

for n in range (1,10,2):
    print(n, " is an odd number")
```

Yet another modification shows that the range sequence can be defined separately (producing an identical output):

*Example 4*

```
# Output odd numbers in interval <0,9>

a = range(1,10,2)
for n in a:
    print(n, " is an odd number")
```

## Break keyword used in the for loop

We've already learned that the while loop can be terminated by the break keyword. The same functionality applies for the "for loop" as the next example shows:

```
# end the loop if i is larger than 3:
for i in range(100):
    if i > 3:
        break
    print(i)
```

## while vs. for loop comparison

Let's write and compare two pieces of code. The assignment is to write a program that sums numbers from 1 to 100. Both versions (while and for) achieve the same result.

### Example 5

while loop	for loop
<pre>LIMIT = 100  sum = 0 i = 1 while i &lt;= LIMIT:     sum = sum + i     i = i + 1 print ("sum:", sum)</pre>	<pre>LIMIT = 100  sum = 0 for i in range(1,LIMIT+1,1):     sum = sum + i print ("sum:", sum)</pre>

## Module-14

### *Functions*

A Python function is a block of code designed to perform a particular task. A function is defined with the (def) keyword, followed by a name ('maxnum' in our example), followed by parentheses (). A colon is used to specify the start of the function code.

Function names follow the same convention as variable names.

#### *Important Rule:*

*Make sure that your function is declared before it is needed(called). Note that the body of a function isn't interpreted until the function is executed. However, the functions can be defined in any order, as long as all are defined before any executable code uses a function. So let's repeat:*

*The rule: All functions must be defined before any code that does real work*

*Therefore, put all the statements that do work last.*

#### *Example 1*

```
def maxnum(a, b):  
    r = a  
    if (r < b):  
        r = b  
    return r  
  
u = 10  
v = 20  
x = maxnum(u, v)  
print("value of x: ", x)
```

#### **Note about the distinction between parameters and arguments:**

When we talk about functions, the terms "parameters" and "arguments" are often interchangeably used as if it were one and the same thing but there is a very subtle difference:

- a)  
Parameters are variables listed as a part of the function definition.
- b)  
Arguments are values passed to the function when it is invoked.

Having explored the basics of for loops and function, let's present an example based on these concepts:

### Example 3

```
import random

def largestNumber(n1, n2, n3):
    largestNum = n1
    if (n2 > largestNum):
        largestNum = n2
    if (n3 > largestNum):
        largestNum = n3
    return largestNum

print("Iteration      n1      n2      n3 largestNum")
for i in range(5):
    # for each iteration generate 3 random float numbers
    # in interval <0, 1000>, and find/print the largest number
    a = random.random() * 1000
    b = random.random() * 1000
    c = random.random() * 1000
    largest = largestNumber(a, b, c)

    aStr      = str(round(a,2)).rjust(10)
    bStr      = str(round(b,2)).rjust(10)
    cStr      = str(round(c,2)).rjust(10)
    largestStr = str(round(largest,2)).rjust(10)

    print ("      ", (i+1), aStr, bStr, cStr, largestStr)
```

Output:

Iteration	n1	n2	n3	largestNum
1	683.15	133.59	870.13	870.13
2	54.36	931.9	380.46	931.9
3	215.9	195.45	890.17	890.17
4	908.97	802.3	296.81	908.97
5	291.17	226.68	577.85	577.85

## Module-15

### *The scope of a variable explained*

The Python code below defines function 'maxnum'

```
def maxnum (a, b):  
    r = a;  
    if r < b:  
        r = b;  
    return r
```

Note that the function defines variable `r`. This variable exists inside the function 'maxnum'. For other functions this variable is invisible. The technical term for this feature is called scope. It means that the variable `r` has only a local scope. Local scope means: within the scope of the function 'maxnum'. If we define variables outside of any function then the scope of such a variable is global. This means: anywhere in our program a global variable can be accessed and modified.

It is very important to grasp the distinction between local and global variables. Therefore let's re-state this again:

We know that Python defines a variable using an assignment statement:

```
x = 1 # this statement defines variable x
```

If this statement is defined *inside* a function body the scope (or the visibility) of this declaration/assignment is limited to the body of the function. However, if this statement is placed *outside* of any function then the scope of defined variables is *global*. It is not a good coding technique to use global variables as it leads to many problems. An experienced programmer can use under certain conditions global variables; Python certainly allows that. But for our purposes, we will avoid using global variables!

To repeat one more time: the scope refers to the visibility of variables. This means which parts of your program can see or use it. The coding technique implemented in this course limits a variable's scope to a single function.

How is this done? Let's explore the next example, which is going to be a template for all our examples:

```
# a Python Program Template
def main():

    some code here . . .
    # end of main()

def someFunction():
    some code here . . .
    # end of someFunction()

main()
```

What you see is a call to function main (at the very end of the program sequence) which passes the code execution to the body of the function main. In other words, all action happens *inside* the body of the function main. Other functions are called directly or indirectly from the main function. This style ensures that all variables have a local scope - which is our objective. This seemingly unnecessary complication of enclosing all Python code in one or several functions guarantees the reliability of our code.

Remember we've established a rule for ordering the functions stated like this:  
*the functions can be defined in any order, as long as all are defined before any executable code uses a function.*

The presented template puts the executable code in one line of code - main() which must be placed as the very last statement! Let's demonstrate this using the following example:

### Example 1

```
# Python program that does NOT use global variables
def main():
    a = 10
    b = 20
    x = maxnum(a, b)
    print("Value of x: ", x)
    # end of main()

def maxnum(a, b):
    r = a
    if r < b:
        r = b
    return r
    # end of maxnum()

# call to the 'main' function MUST BE THE LAST statement !!!
main()
```

### Note:

*All examples on Python webpage for Python – LEVEL-3, Section-B use this technique.*

## Module-16

### *Lists*

What is a list? A list is a special variable, which can hold more than one value at a time. Let's say we have a list of numbers. Storing them in single variables would look like this:

```
number1 = 200
number2 = 300
number3 = 400
```

But if we wanted to use a loop to find a specific number, that would be difficult – especially for large set of numbers. The solution is to use a list. The list can hold many values under a single name, and you can access the values by referring to an index number:

```
myList = [200, 300, 400]
```

Essential list characteristics are

- A list can hold many values under a single name
- List elements are accessed using their index number
- The indexing operator [ ] selects a single element from a sequence. The expression inside square brackets is called the index, and must be an integer value.
- List indexes start with 0. [0] is the first list element, [1] is the second, [2] is the third, etc.

Note1:

Python (as most modern languages) uses so-called zero-based indexing. This is extremely important! Let's remember and repeat:

- [0] the index of the first element
- [1] the index of the second element
- if the list length is defined as len then [len-1] is the index of the last element

Note2:

informally lists and arrays are treated as identical concepts. However, Python does make a distinction between lists and arrays (although they are very similar). However, we will not discuss the concept of Python arrays in this manual.

### **Creating a list**

Syntax:

```
listName = [item1, item2, ...]
```

Example :

```
numbers = [200, 300, 400] # this list has 3 elements
```

### Access the Elements of a list

You access a list element by referring to the index number. This statement accesses the value of the first element in 'numbers':

```
someVariable = numbers[0] # the first element
```

### The List length function len()

The length property of a list returns the number of its elements.

Example:

```
primes = [2, 3, 5, 7]
x = len(primes) # the length of primes is 4
```

### Empty List Definition

An empty list is defined as the following

```
myList = [] # define an empty list
x = len(myList) # the length is zero
```

So far we've covered the essentials of Python lists – it is time to show examples. The first example shows the basics:

#### Example 1

```
numberSet = [200, 300, 400]

print("First number: ", numberSet[0])
print("Second number: ", numberSet[1])
print("Last number: ". numberSet[2])
```

Output:

```
First number: 200
Second number: 300
Last number: 400
```

The next example defines and initiates a list. Consequently, list elements are assigned random values:

#### Example 2

```
import random

numberSet = [0, 0, 0]
setLen = len(numberSet)
for i in range(setLen):
    numberSet[i] = random.randint(1,1000)
    print("Index:", i, "Value:", numberSet[i])
```



Output:

```
Index: 0 Value: 276
Index: 1 Value: 822
Index: 2 Value: 562
```

### String Elements in List

A list can contain string elements

Example:

```
colors = ["red", "green", "orange", "blue"]
print(colors[0]) # outputs 'red'
print(colors[3]) # outputs 'blue'
```

### Boolean Elements in List

Here is an example of a list that has 3 elements, all of them set to False (capitalized):

```
boolVector = [False, False, False]
```

## Module-17

### *List Functions*

#### **Add a new item to a list – append():**

Function append adds an element to the end of the list

```
fruits = [] # let begin with an empty list
fruits.append("Apple") # sets fruits[0] to "Apple"
fruits.append("Mango") # sets fruits[1] to "Mango"
```

It should be self-evident that after 'append' is executed the list length is incremented by 1.

#### **Delete last element from list – pop()**

Function pop removes the last element of a list; the list length drops by 1

```
fruits = ["Orange", "Lemon", "Avocado"]
fruits.pop() # Removes the last element ("Avocado") from list
```

**Delete an element from a list by index using del operator**

The del operator removes an element at the specified index location from the list.  
Example:

```
values = [100, 200, 300, 400, 500, 600]
# Use del to remove by an index
del values[2]
print(values)
```

**Insert an element at the specified position using insert function**

The insert() function inserts an element at the specified index.

Syntax:

```
listName.insert(index, element)
```

Example:

```
values = [100, 200, 300, 400, 500, 600]

values.insert(2,999) # insert 999 into position values[2]
print(values)
values.insert(0,777) # insert 777 into position values[0]
print(values)
```

It is obvious that each insert increments the list length by 1.

**Remove all elements from a list – clear() function**

The clear() function removes all the elements from a list.

Syntax

```
listName.clear()
```

Example:

```
metals = ["iron", "copper", "silver", "zinc"]

metals.clear() # removes all elements - len(metals) returns 0
```

The following example shows all list functions in action:

*Example 1*

```
colors = [] # start with an empty list
# add 4 elements
colors.append("red")
colors.append("black")
colors.append("white")
colors.append("blue")
print("trace-1", colors)

# delete the last element
colors.pop()
print("trace-2", colors)

# insert into 3rd and 1st position (indexes as [2] and [0])
colors.insert(2,"green")
print("trace-3", colors)
colors.insert(0, "yellow")
print("trace-4", colors)

# delete the second (index is [1]) element
del colors[1]
print("trace-5", colors)
```

**Output:**

```
trace-1 ['red', 'black', 'white', 'blue']
trace-2 ['red', 'black', 'white']
trace-3 ['red', 'black', 'green', 'white']
trace-4 ['yellow', 'red', 'black', 'green', 'white']
trace-5 ['yellow', 'black', 'green', 'white']
```

## Module-18

*Looping List Elements, Copy List, Print List*

### Looping List Elements

Suppose you need to construct a loop that will iterate over all list elements. We know how to do it using the function `range()`:

*Example 1*

```
numbers = [100, 200, 300]
length = len(numbers)
for i in range(length):
    xx = numbers[i]
    print(xx)
```

Output:

```
100
200
300
```

However, Python allows to loop through all the elements of a list. The previous example could be rewritten (and simplified) as the following (producing identical output):

*Example 2*

```
numbers = [100, 200, 300]
for xx in numbers:
    print(xx)
```

### How to make a copy of the list

You cannot create a copy by the following assignment

```
a = [100, 200, 300]
b = a # this assignment does not make a copy of a
```

Make a copy of a list with the `list()` method:

*Example 3*

```
numberSet1 = [101, 102, 103, 104]
numberSet2 = list(numberSet1)
print(numberSet2)
```

**Print lists in different ways**

Let's explore the ways to print a list:

a)

we've already seen the simplest possible way:

```
a = [1, 2, 3, 4, 5]
print(a)
```

b)

print a list using a loop

```
a = [1, 2, 3, 4, 5]
for y in a:
    print(y)
```

c)

print a list using a loop and make the output a single line (no line breaks):

```
a = [1, 2, 3, 4, 5]
for y in a:
    print(y, end=" ")
```

## Module-19

*Misc. topics: comparing floating numbers, passing variable & list to function*

**Comparison of floating-point numbers**

Due to rounding errors, most floating-point numbers end up being slightly imprecise. This means that numbers expected to be equal often differ slightly, and a simple equality test fails. The next example illustrates the problem:

*Example 1*

```
a = 0.15 + 0.15
b = 0.1 + 0.2

if a == b:
    print("Values a and b are equal")
else:
    print("Values a and b are NOT equal")
```

Output:

```
Values a and b are NOT equal
```

This is not the result we want to see. Let's modify the above example by introducing a custom function called `nearlyEqual`

*Example 2*

```
def nearlyEqual (x, y, epsilon):
    ret = False
    if abs(a-b) < epsilon:
        ret = True
    return ret

epsilon = 0.001;
a = 0.15 + 0.15
b = 0.1 + 0.2

if nearlyEqual(a, b, epsilon):
    print("values a and b are equal")
else:
    print("values a and b are NOT equal")
```

Output:

```
Values and b are equal
```

It should be stressed that using the absolute difference (called epsilon) is not truly a valid method. But it is OK to use it in simple situations. A more complete answer is complicated and it is beyond the level of this document. For more details please refer to this article:

<https://floating-point-gui.de/errors/comparison/>

**Passing Variable to Function**

If you pass a variable to a function, its value is copied and handed to the function (pass by value). Therefore, the function can't change the variable. Here is the example of a function that demonstrates how this feature works:

*Example 3*

```
def calcNumbers(m, n):
    m = m + 1
    n = n + 1
    return (m + n)

a = 10
b = 20

x = calcNumbers(a, b)
# values of a, b are not changed in function calcNumbers()
print("value of a:", a)
print("value of b:", b)
print("value of x:", x)
```

Output:

```
value of a: 10
value of b: 20
value of x: 32
```

### Passing List to Function

Passing list to function is different – function can change/delete/append the list elements. Explore this example:

*Example 4*

```
def processList (list1, list2):
    # modify list1
    list1.append(40)
    for i in range(len(list1)):
        list1[i] = list1[i] + 1

    # modify list2
    list2.append(100)
    list2.append(200)
    list2.append(300)

numbers = [10, 20, 30]
myList = []

processList (numbers, myList)
print("numbers:", numbers)
print("myList:", myList)
```

Output:

```
numbers: [11, 21, 31, 41]
myList: [100, 200, 300]
```

## Module-20

### *Review of examples*

This module serves as a wrap-up of what we've learned about loops, lists and functions. The best way to verify what we've learned in our Python course is to carefully review the next examples. Please study the code, feel free to experiment by adding tracking output statements.

#### **A. Select randomly numbers from a list**

##### *Example 1*

```
import random

COUNT = 5
numbers = [12, 78, 458, 800, 2965, -568, 23450, -2, -1777]
print("numbers:", numbers)

print("Randomly selected numbers:")
for i in range(COUNT):
    randIdx = random.randint(0, COUNT-1)
    print("index:", randIdx, "number:", numbers[randIdx])
```

##### **Output:**

```
numbers: [12, 78, 458, 800, 2965, -568, 23450, -2, -1777]
Randomly selected numbers:
index: 0 number: 12
index: 3 number: 800
index: 4 number: 2965
index: 2 number: 458
index: 1 number: 78
```



**B. Find the smallest and the largest element**

In the following example we'll create and populate list with random numbers and then find the smallest and the largest element:

*Example 2*

```
import random

SET_LEN = 50
numberSet = []

# set the list elements to random numbers <0, 1000>
for i in range(SET_LEN):
    numberSet.append(random.random() * 1000)

# find/assign min and max values
minNum = numberSet[0]
maxNum = numberSet[0]

for i in range(SET_LEN):
    if minNum > numberSet[i]:
        minNum = numberSet[i]
    if maxNum < numberSet[i]:
        maxNum = numberSet[i]

print("min value is: ", round(minNum,3))
print("max value is: ", round(maxNum,3))
```

**Output:**

```
min value is: 8.186
max value is: 975.476
```

### C. Shuffle a list of consecutive numbers

#### Example 3

```
import random

NUMBERS_COUNT = 10
numbers = []

# create a list of consecutive numbers: 1,2,3, .. NUMBERS_COUNT
for i in range(NUMBERS_COUNT):
    numbers.append(i+1)

# shuffle the list
keepShuffling = True
pos = NUMBERS_COUNT - 1 # begin with the last element
while keepShuffling:
    randIdx = random.randint(0, pos-1)
    # swap numbers between [pos] and [randIdx] using a temporary variable tmp
    tmp = numbers[pos]
    numbers[pos] = numbers[randIdx]
    numbers[randIdx] = tmp

    # decrement [pos] until you reach [1] and then stop the loop
    pos = pos - 1
    if pos <= 1:
        keepShuffling = False # stop shuffling

print("Shuffled list:", numbers)
```

Output:

```
Shuffled list: [8, 7, 9, 1, 10, 3, 2, 6, 5, 4]
```

## Module-21

### *Binary Search*

Binary search is the most popular search algorithm. It is very efficient and commonly used. Binary search works only on a sorted set of elements!

#### **Binary Search Algorithm Explanation:**

Binary search compares the search element to the middle element of the list.

If the search element is greater than the middle element,  
then the left half or elements before the middle elements of the  
list is eliminated from the search space, and the search  
continues in the remaining right half.  
Else if the search element is less than the middle value,  
then the right half elements or all the elements after the middle  
element is eliminated from the search space, and the search  
continues in the left half.

This process is repeated until the middle element is equal to the search element, or if the algorithm finds that the searched element is not in the given list at all.

The following example show the binary search in action – note the presence of tracking print statement that show the changes in the range of searched elements.

*Example 1*

```
import math

# Binary Search
numbers = [11, 27, 32, 38, 42, 64, 87, 91]  # ascending order list
target = 87

print("Binary search on a list sorted in ascending order")
print("numbers=", numbers)
print("target=", target)

smallest = 0
biggest = len(numbers) - 1
found = False
count = 1

print("count smallest biggest middle midValue")

while smallest <= biggest and found == False:

    middle = math.floor((smallest + biggest) / 2)
    midValue = numbers[middle]
    s1 = str(count).rjust(5)
    s2 = str(smallest).rjust(8)
    s3 = str(biggest).rjust(7)
    s4 = str(middle).rjust(6)
    s5 = str(midValue).rjust(8)
    print(s1, s2, s3, s4, s5)
    if target == midValue:
        # Yes, target has been found
        print("target has been found")
        found = True
    else:
        if target < midValue:
            # target is smaller than the middle element
            biggest = middle - 1
        else:
            # target is higher than the middle element
            smallest = middle + 1
    count = count + 1

if found == False:
    print("target has NOT been found")
```

Output:

```

Binary search on a list sorted in ascending order
numbers= [11, 27, 32, 38, 42, 64, 87, 91]
target= 87
count smallest biggest middle midvalue
  1         0         7         3         38
  2         4         7         5         64
  3         6         7         6         87
target has been found

```

Note that this particular implementation of binary search work only for ascending order. Check the implementation in the Python-Level-3 Script "Binary Search" which handles both – ascending and descending order of elements.

## Module-22

### *Bubble Sort*

The bubble sort gets its name because elements tend to move up into the correct order like bubbles rising to the surface. This simple algorithm performs poorly in real world use and is used primarily as an educational tool.

#### **Bubble Sort Algorithm Explanation:**

Bubble sort is a sorting algorithm that works by repeatedly stepping through a list that need to be sorted, comparing each pair of adjacent elements and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted.

#### *Example 1*

```
# Bubble Sort
numbers = [876, 356, 700, 247, 111]

print("Bubble Sort")
print("numbers:", numbers)

count    = 0
swapped  = True
while swapped:
    swapped = False # reverse
    i = 1
    while i < len(numbers):

        # check if the pair is out of order
        if numbers[i-1] > numbers[i]:

            # swap the pair using variable temp
            temp = numbers[i-1]
            numbers[i-1] = numbers[i]
            numbers[i] = temp
            swapped = True # this will force to repeat the 'while' loop
        i = i + 1

    if swapped:
        count = count + 1
        print("count=", count, end=" ")
        print("numbers:", numbers)
    else:
        print("the list has been sorted - bubble sort ends")

print("The End")
```

#### Output:

```
Bubble Sort
numbers: [876, 356, 700, 247, 111]
count= 1 numbers: [356, 700, 247, 111, 876]
count= 2 numbers: [356, 247, 111, 700, 876]
count= 3 numbers: [247, 111, 356, 700, 876]
count= 4 numbers: [111, 247, 356, 700, 876]
the list has been sorted - bubble sort ends
The End
```

Explore the bubble sort implemented as a function in script Python-Level-3 Script "Bubble Sort"

## Module-23

### *Selection Sort*

Selection Sort is one of the most simple sorting algorithms. It works in the following way:

- Find the smallest element. Swap it with the first element.
- Find the second smallest element. Swap it with the second element.
- Find the third smallest element. Swap it with the third element.
- Repeat finding the next smallest element and swapping it into the corresponding correct position till the list is sorted.

Here is a practical demonstration. Note that the number of times the sort passes through the list is one less than the number of items in the list.

```
myList = [64, 25, 12, 22, 11]

# Find the minimum element in myList [0...4]
# and place it at beginning
11 25 12 22 64

# Find the minimum element in myList [1...4]
# and place it at beginning of myList [1...4]
11 12 25 22 64

# Find the minimum element in myList [2...4]
# and place it at beginning of myList [2...4]
11 12 22 25 64

# Find the minimum element in myList [3...4]
# and place it at beginning of myList [3...4]
11 12 22 25 64
```

Explore the selection sort implemented as a function in script Python-Level-3 Script "Selection Sort"

*Example 1*

```
# Selection Sort
numbers = [876, 356, 700, 247, 111]

print("Selection Sort")
print("numbers:", numbers)

# finding minimum element (numbersLen-1) times
count = 1
numbersLen = len(numbers)
for i in range(numbersLen-1):
    minIdx = i
    j0 = i + 1

    print("count=", count, "minIdx=", minIdx, "j0=", j0)

    # test against elements starting j0 to find the smallest
    j = j0
    while j < numbersLen:
        # if this element is less, then it is the new minimum
        if numbers[j] < numbers[minIdx]:
            minIdx = j # found new minimum - update its index
            j = j + 1

    if minIdx != i:
        # swap the elements - avoiding swapping an element with itself
        temp = numbers[i]
        numbers[i] = numbers[minIdx]
        numbers[minIdx] = temp

    print("numbers:", numbers)
    print(" ")
    count = count + 1
```

*Output:*

```
Selection Sort
numbers: [876, 356, 700, 247, 111]
count= 1 minIdx= 0 j0= 1
numbers: [111, 356, 700, 247, 876]

count= 2 minIdx= 1 j0= 2
numbers: [111, 247, 700, 356, 876]

count= 3 minIdx= 2 j0= 3
numbers: [111, 247, 356, 700, 876]

count= 4 minIdx= 3 j0= 4
numbers: [111, 247, 356, 700, 876]
```

The End

## Module-24

### *Merge Lists*

Merge algorithm takes two sorted lists as input and produce a single list as output, containing all the elements of both input lists in sorted order.

Let's show the process using real data:

```
Input Lists:
list1  = [1, 3, 4, 5]
list2  = [2, 4, 6, 8]

Output list (merged):
List3 = [1, 2, 3, 4, 4, 5, 6, 8]
```

The next example shows an implementation of two lists. Here is a brief review:

On a given iteration of the first loop, we take whichever list has the smaller-value element at their index and advance that index. This element will occupy the next position in the merged list.

Finally, once we've transferred all elements from one list, we'll copy the remaining from the other into the merged list.



## Example 1

```
# Merge Two Lists
numbersA = [100, 103, 141, 150, 500, 900]
numbersB = [101, 140, 180, 910, 999]
numbersX = []

print("numbersA:", numbersA)
print("numbersB:", numbersB)

# Step 1.
idxA = 0
idxB = 0
while idxA < len(numbersA) and idxB < len(numbersB):
    # Check if current element of first
    # list is smaller than current element
    # of second list. If yes, store first
    # list element and increment first list
    # index. Otherwise do same with second list
    if numbersA[idxA] < numbersB[idxB]:
        # append head(numbersA) to numbersX
        numbersX.append(numbersA[idxA])
        idxA = idxA + 1
    else:
        # append head(numbersB) to numbersX
        numbersX.append(numbersB[idxB])
        idxB = idxB + 1

print("Merged list(after step 1):", numbersX)

# Step 2.
# By now, only one of the lists had been fully copied to numbersX
# it remains to copy the other input list

while idxA < len(numbersA):
    # the rest of numbersA is to be copied
    numbersX.append(numbersA[idxA])
    idxA = idxA + 1

while idxB < len(numbersB):
    # the rest of numbersB is to be copied
    numbersX.append(numbersB[idxB])
    idxB = idxB + 1

print("Merged list(final):". numbersX)
```

## Output:

```
numbersA: [100, 103, 141, 150, 500, 900]
numbersB: [101, 140, 180, 910, 999]
Merged list(after step 1): [100, 101, 103, 140, 141, 150, 180,
500, 900]
Merged list(final): [100, 101, 103, 140, 141, 150, 180, 500, 900,
910, 999]
```

Explore the merge algorithm sort implemented as a function in script:  
Python Level-3 Script "Merge Two Lists"

This is the end of LEVEL-3; The next two activities are following:

- a) Explore the Python Level 3 Examples (Group A and B)  
Link: [www.windmaran.com/wm/python/py-main.php?px=L3A1](http://www.windmaran.com/wm/python/py-main.php?px=L3A1)
- b) Answer Python Online Level 3 Quiz questions  
Link: [www.windmaran.com/wm/python/py-quiz.php?px=L3Q1](http://www.windmaran.com/wm/python/py-quiz.php?px=L3Q1)

Having completed study of this manual and completing the online quiz questions you are ready take a Python Self-Assessment Test. There is a set of assignments to test your coding abilities:

Link: [www.windmaran.com/wm/python/py-skill-test.php](http://www.windmaran.com/wm/python/py-skill-test.php)

# Appendix-1

## Run Python programs using Thonny

Thonny is an integrated development environment (IDE) for Python that is designed for beginners. Explore and learn more about Thonny at:

<https://thonny.org/>

Check an online Thonny tutorial at:

<https://realpython.com/python-thonny/>

How to download and install Thonny on your computer check Thonny home page [thonny.org](https://thonny.org)

## Testing Thonny Environment

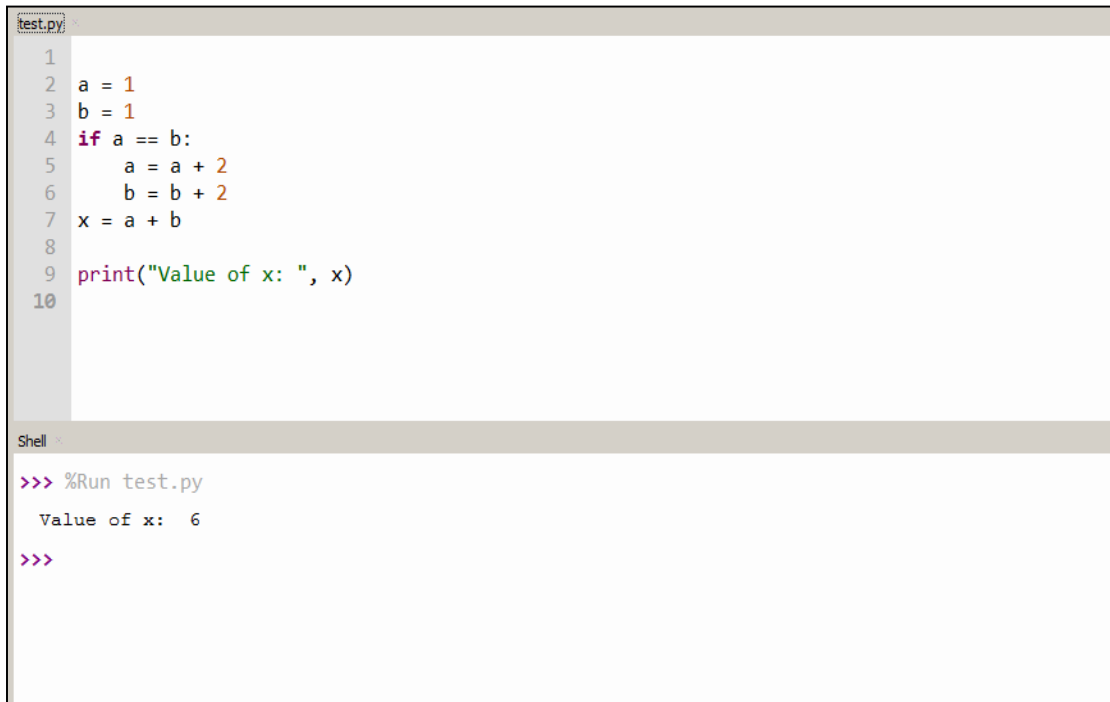
Assuming that you have successfully downloaded and installed Thonny software on your computer it is time to run the very first program. Here is a very simple program that can be used to verify that we can execute Python programs on our computer:

*Test Code*

```
a = 1
b = 1
if a == b:
    a = a + 2
    b = b + 2
x = a + b

print("Value of x: ", x)
```

When you copy/paste and execute the above program your Thonny screen should look like the following:



```
test.py
1
2 a = 1
3 b = 1
4 if a == b:
5     a = a + 2
6     b = b + 2
7 x = a + b
8
9 print("Value of x: ", x)
10
```

```
Shell
>>> %Run test.py
Value of x:  6
>>>
```

# Appendix-2

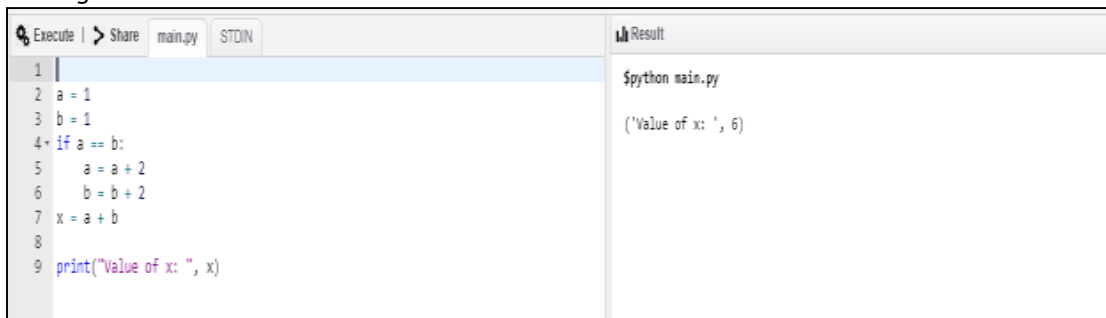
## Run Python programs online in browser

There are quite a few ways to run Python in your web browser. We recommend these web sites to explore

### 1. tutorialspoint.com

Open this link: [https://www.tutorialspoint.com/execute\\_python\\_online.php](https://www.tutorialspoint.com/execute_python_online.php)

Using the Python test script presented in the Appendix-1 the executed program looks like the following:



The screenshot shows the 'Execute' tab of the tutorialspoint.com online Python executor. The code editor on the left contains the following Python code:

```
1 |
2 a = 1
3 b = 1
4 if a == b:
5     a = a + 2
6     b = b + 2
7 x = a + b
8
9 print("Value of x: ", x)
```

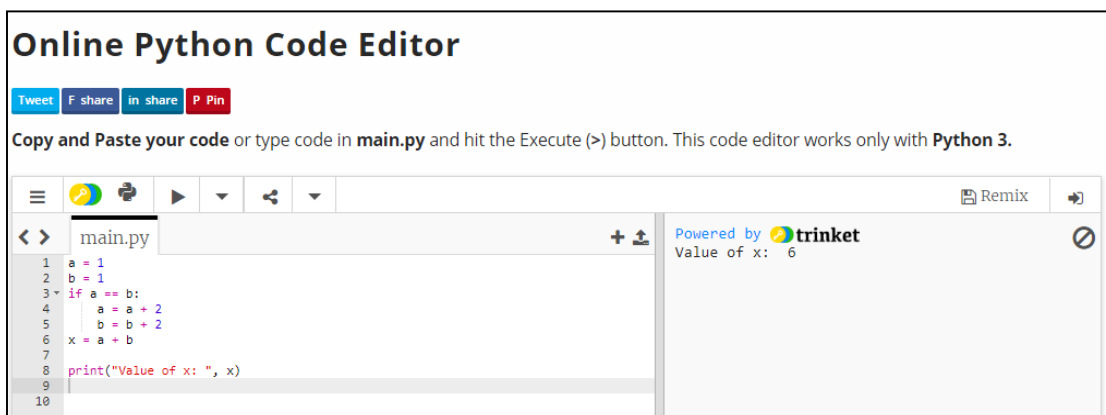
The 'Result' tab on the right shows the output of the program:

```
$python main.py
('Value of x: ', 6)
```

### 2. pynative.com

Open an online program editor and run the python test code:

<https://pynative.com/online-python-code-editor-to-execute-python-code/>



The screenshot shows the 'Online Python Code Editor' interface on pynative.com. The code editor on the left contains the following Python code:

```
1 a = 1
2 b = 1
3 if a == b:
4     a = a + 2
5     b = b + 2
6 x = a + b
7
8 print("Value of x: ", x)
```

The right panel shows the output of the program:

```
Powered by trinket
Value of x: 6
```

The End