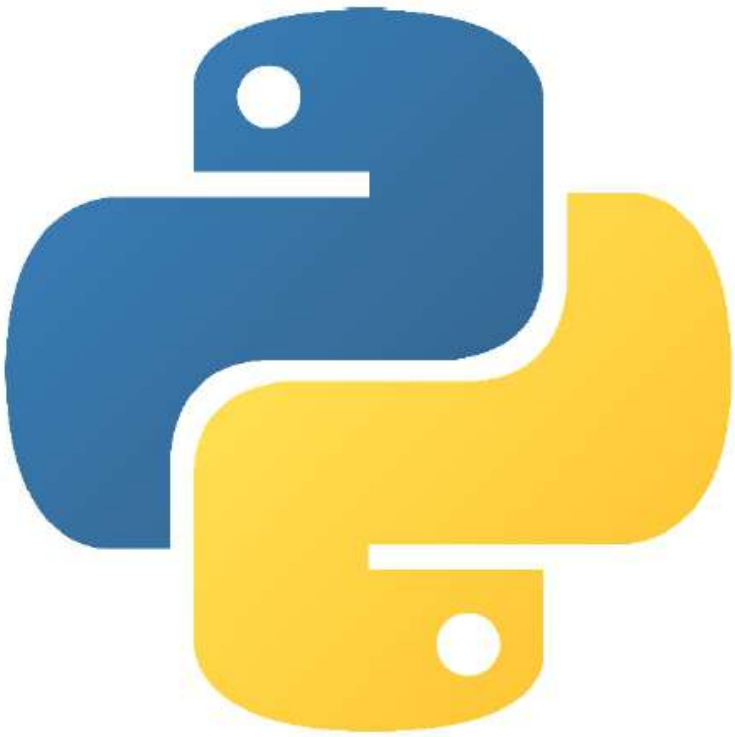
Elementary python programming

Basheer A. Hassoon
Mushtaq A. Hassoon
Abass Conteh
Yonas G. Mamecha



Elementary Python Programming

Basheer A. Hassoon

Mushtaq A. Hassoon

Abass Conteh

Yonas G. Mamecha

Deposit number in Iraqi National Library and Archive, Baghdad (2321) year 2021.

005, 13
H 344 A. Hassoon, Basheer
Elementary python programming / Basheer A.Hassoon, Mushtaq A. Hassoon and others : Basrah ; مطبعة بلال , 2021
76p; 24cm
Programming (python) – A –
A. Hassoon, Mushtaq (co.author) ,
Abass (co.author), Yonas (co.author)

٢٠٢١ / ٢٣٢١

المكتبة الوطنية / الفهرسة أثناء النشر

رقم الإيداع في دار الكتب والوثائق ببغداد (٢٣٢١) لسنة ٢٠٢١

Table of Contents

Chapter 1:

Introduction

Python is an open source, high level, powerful and flexible programing language. It is used to create applications, games, write GUI's, and develop web applications. It has several advantages over other high level programming languages (Readability, Higher productivity, less learning time and runs across different platforms).

This online free book is suitable for new programmers who would like to hit the ground and start running in different developments and career skill set of the current industrial needs. It's also good for experienced python programmers who just need a fast review and other experienced programmers from other programming languages who wish to acquire simple, fast, and easy way to exemplify the skills they need.

Chinese version of this book will be available online for free too as soon as translation done.

Chapter 2:

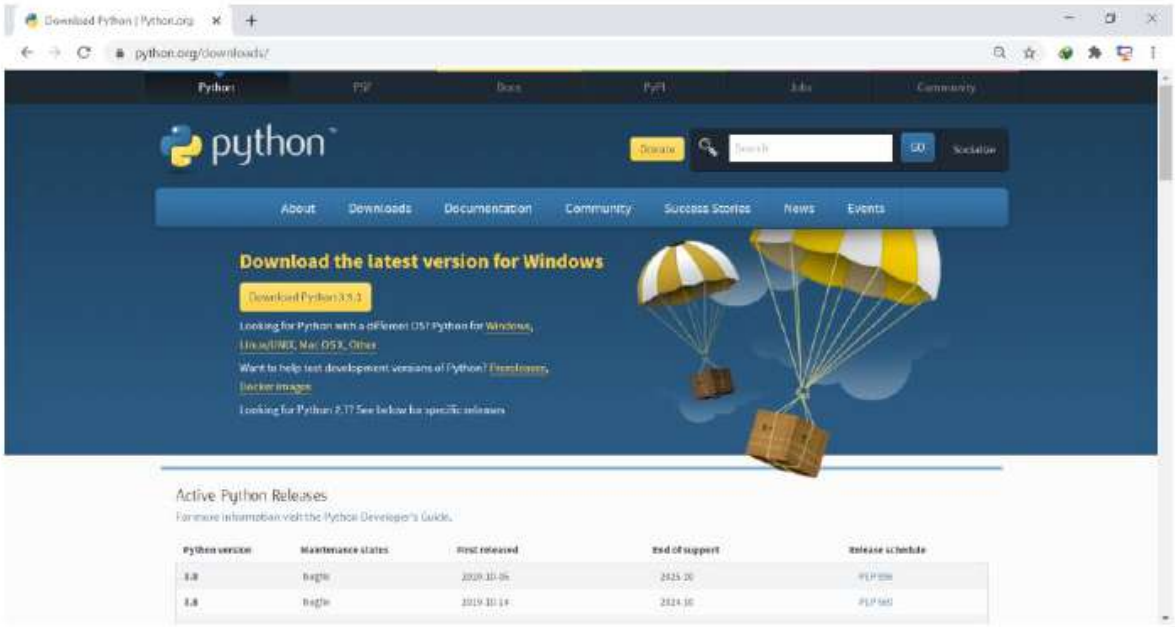
Installing Python

To install Python, you must first download the installation package from the following:

- For Windows OS: https://www.python.org/downloads/
- For MAC OS: https://www.python.org/downloads/mac-osx/
- For Linux/UNIX: https://www.python.org/downloads/source/
- Other: https://www.python.org/download/other/

On that page, you can download the latest version of Python. You will find two versions there, Python 2 and Python 3. In this book we will use Python 3.

After finished downloading, run the downloaded file and complete setup.



Python official download website

Chapter 3:

Launching and Using Python

Python is a flexible and dynamic language that you can use interactively when you want to test a statement or a code line-by-line, or you can use it in script mode when you want to interpret an entire file of statements or application program.

To use Python, you can use either the Command Line window or the IDLE Development Environment.

Starting Python

There are different ways to launch Python, depending on the operating system installed on your machine:

If you are using Windows, you can start the Python by clicking on its icon on your desktop or find it on the Start menu.

If you are using GNU/Linux, UNIX, and Mac OS systems, you can run Python from Terminal or browse all applications to find its icon or type IDE.

The Python Shell Window

You can type and enter statements or expressions for evaluation in the Python Shell Window, which has dropdown menus and a >>>prompt.

The Python Shell window has the following menu items: File, Edit, Shell, Debug, Options, Windows, and Help.

The File Window

The items on the file menu allows you to create a new file, open old file, open a module...etc. When you click on the 'New File' option, new window will pop up, simple and standard text editor where you can type or edit your code. Initially, its name is 'untitled' but it will change as soon as you save your code.

This window has menu items too (File, edit, Format, Run, Options, Window and Help). We will use File menu to save our file by clicking on (Save) Or using keyboard shortcut (Ctrl + S), and Run menu to run our module by clicking on (Run Module) or using keyboard shortcut (F5).

Chapter 4:

Python Syntax

Python syntax refers to the set of rules which defines how a Python program is written. As a result, you must become acquainted with these regulations.

Reserved keywords

There are reserved words in Python that you can't use them as variable, constant, identifier or function name in your code. In case you use them in your code and execute it then you will get an error.

and	as	assert	break
class	continue	def	del
elf	else	except	False
finally	for	from	global
if	import	in	is
lambda	None	nonlocal	not
or	pass	raise	return
True	try	while	with
yield			

Python Identifiers

A Python identifier is a name given to a variable, object, function, module and class that is used in a python code.

An identifier can be a combination of letters (A - Z or a - z) or digits (0 - 9) or an underscore (_). For example (my Class, x, X, Aa, aA, A9, A_9).

Python does not allow you to use Python keywords as identifiers and you can't use punctuation characters within identifiers such as @, $, and %. Python is a case sensitive programming language, hence the identifiers CountOddNumbers and countOddNumbers will convey different meanings.

Quotations

Python allows you to use single, double or triple quotation marks to indicate string literals. For single-line quotations, both single and double quotes can be used, while for multi-line strings, triple quotes are used. However, be consistent; it is required to have the same type of quote to start and end the string.

Statements

Statements are instructions that Python interpreter can execute whether it is single or multi-line statements. To break a long single line statement over multiple lines, you can wrap the expression inside braces, brackets or parentheses. Another way is to use backslash(\) at the end of every line to indicate line continuation.

Comments

Writing comments inside your program to describe what your code does is very good and helpful, especially when you must review or revisit your program later. It also helps other programmers to understand your code. In order to write a comment, you first need to use the hash (#) symbol before your comment. When running the code, the Python interpreter completely ignores your comment.

Chapter 5

Variables and Data types

Variables

A variable is a container that stores values that you can access, retrieve or change later.

In Python, you declare a variable by assigning value to it. For example:

My var = 10 (the statement means that "my var is set to 10" not stating that my_var is equal to 10.).

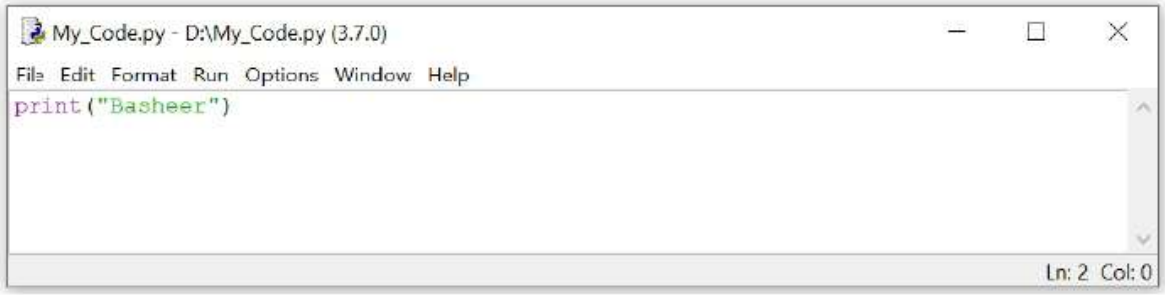
Note: before we dive in more, you should first know how to use Print Function because I will use it from now on to help you see and understand what I am doing.

print()

Print function will print out what is between the brackets

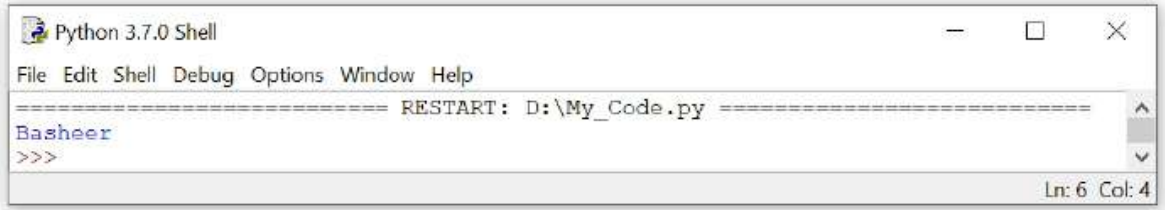
Example: Open new file and write down the following code then execute it.

```
print('Basheer')
```



Output:

Basheer



You can see that Python has print Basheer on the screen. Now write down your name instead of Basheer and run the code again. Ensure that your name printed on the screen.

Data Types

Python provides several data types (Binary, Hexadecimal, Integer, float, strings, Booleans, lists, date and time) to match the needs of programmers and application developer's usable data.

Binary

A binary number, also known as a base-2 number, is a number expressed in the binary numeral system. In order to assign a binary number to a variable you will need to use the prefix '0b' or '0B'.

Example:

```
B = 0b1000
print(B)
```

Output:

```
8
```

Hexadecimal

hexadecimal is a positional system that uses a base 16 depict numbers. In order to assign a hexadecimal number to a variable you will need to use the prefix '0x' or '0X'.

Example:

```
hex_var = 0xA0B
print (hex_var)
```

Output:

```
2571
```

Integer

Integers are whole numbers that do not contain a decimal point. For example (800, -900, 6)

Example:

```
inte = 800
print (inte)
```

Output:
800

Float

float is a real number written with a decimal point that segregates the integer from the fractional numbers.

Example:
```
fl_var = 6.22
print (fl_var)
```

Output:
6.22

Strings

A string is a sequence of characters. Characters can be combined with a single or double quote of letters, numbers, and symbols. In case your literal string contains a single quote then you must use double quote to avoid errors.

Example:
```
string1 = 'Am ok'
print (string1)
string2 = "I'm ok"
print (string2)
```

Output:
```
Am ok
I'm ok
```

To get the length of the string you can use len() function.

Example:
```
str1 = 'Python'
print(len (str1))
```

Output:
6

Example:

```
str2 = 'Hello World'
print (len (str2))
```

Output:

```
11
```

Strings can be indexed or subscripted. Indexing starts from 0 (zero). Hence, the first character in the string will have a zero index.

Example: Define the following string

```
Strg = 'HELLO WORLD'
```

This is how Python will index the string:

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	E	L	L	O		W	O	R	L	D
0	1	2	3	4	5	6	7	8	9	10

To access the first character, type and enter the variable name "Strg" and the index "0" within square brackets such as the following:

```
Strg[0]
```

You will get this output: 'H'

To access the last character, you can use either index -1 or use len() function.

```
Strg[-1]
```

OR

```
Strg[ len (Strg)-1]
```

You will get this output: 'D'

To concatenate two strings, you can use the plus (+) operator.

Example:

```
stringValue1 = 'Hello'
stringValue2 = 'World'
stringValue3 = stringValue1 + stringValue2
print (stringValue3)
```

Output:
```
HelloWorld
```

It's possible to repeat strings with the * operator.

Example:
```
str1 = 'Hi'
str2 = str1*3
print (str2)
```

Output:
```
HiHiHi
```

You can create substrings by placing two indices separated by a colon inside square brackets. The first index marks the start of the substring while the second index marks the end of the substring that will not be included.

Example:
```
strg = 'Python'
sub_strg = strg[2:4]
print(sub_strg)
```

Output:
```
th
```

If you want the substring to start from a specific index to the end of the original string, the second index is optional.

Example:
```
strg = 'Python'
sub_strg = strg[2:]
print(sub_strg)
```

Output:
```
thon
```

If you want your substring to start from the beginning of the original string to the end, you can omit the first index.

Example:

```
strg = 'Python'
sub_strg = strg[:2]
print(sub_strg)
```

Output:

```
Py
```

If you have a string like "Python Language" and you need it to be all in lower case, you can use the lower() function.

Example:

```
strg = 'Python Language'
lw_strg = strg.lower()
print(lw_strg)
```

Output:

```
python language
```

Supposing you need your string to be all capitalized, you can use the upper () function.

Example:

```
strg = 'Python Language'
ca_strg = strg.upper()
print(ca_strg)
```

Output:

PYTHON LANGUAGE

Sometimes you need to print an integer number with a string. The str() function will help do that by changing non-strings characters into string.

Example:
```
num = 8
print('My favorite number is:' + str(num))
```

Output:
```
My favorite number is:8
```

Booleans

Boolean is used to represent logic values, True and False, so it only takes these two values. In comparisons, Python it only generates one value (True or False).

Example:
```
bool_1 = 8 > 4
bool_2 = 8 < 4
print(bool_1)
print(bool_2)
```

Output:
```
True
False
```

Lists

A list is used to store any type, number, and information. Python allows you define empty lists or assign items directly.

Example:
```
my list = []
```

Example:
```
colors = ['Blue','Green','Yellow','Red']
```

Since this is an indexed list, the first item has zero as its index. Use the corresponding index to get to any item in the list.

Example: print the first item:
```
colors = ['Blue','Green','Yellow','Red']
print (colors[0])
```

Output:
```
Blue
```

Example: print the fourth item:
```
colors = ['Blue','Green','Yellow','Red']
print (colors[3])
```

Output:
```
Red
```

To see how many colors are on the list, you can use the len() function:

Example:
```
colors = ['Blue','Green','Yellow','Red']
print(len(colors))
```

Output:
```
4
```

To print all colors on the list

Example:
```
colors = ['Blue','Green','Yellow','Red']
print(colors)
```

Output:
```
['Blue', 'Green', 'Yellow', 'Red']
```

To add White color to your list, you can use append command:

Example:
```
colors = ['Blue','Green','Yellow','Red']
colors.append('White')
print(colors)
```

Output:
```
['Blue', 'Green', 'Yellow', 'Red', 'White']
```

To add Gray color to your list in a specific index, you can use insert command:

list.Insert (index, obj)

list (Your list object name)
index (the index where you want to insert your object)
obj (the object you want to insert)

Example:

```
colors = ['Blue', 'Green', 'Yellow', 'Red', 'White']
colors.insert (2,'Gray')
print(colors)
```

Output:

```
['Blue', 'Green', 'Gray', 'Yellow', 'Red', 'White']
```

You can also slice the lists in the same way you slice strings

Example:

```
colors = ['Blue', 'Green', 'Gray', 'Yellow', 'Red', 'White']
print(colors [2:4])
```

Output:

```
['Gray', 'Yellow']
```

Date and Time

Most of applications needs date and time information to work effectively and efficiently. To retrieve the current date and time use datetime.now() function.

Example:

```
from datetime import datetime
print(datetime.now())
```

Output:

```
2021-02-15 05:19:30.397325
```

In order to make the result more readable you can use strftime function.

Example:

```
from time import strftime
print(strftime("%Y-%m-%d %H:%M:%S"))
```

Output:

```
2021-02-15 05:19:30
```

Chapter 6:

Python Operators

There are several operators that python allows user to use them to manipulate data or operands.

1- Arithmetic Operators

+	Addition	adds the value of the left and right operands
-	Subtraction	subtract the value of the right operand from the value of the left operand
*	Multiplication	Multiplies values on the either side of the operator
/	Division	Divides left operand by the right one
**	Exponent	Exponent left operand raised to the power of right
%	Modulus	Divides one value by a second and gives the remainder as a result
//	Floor Division	division of operands where the solution is a quotient left after removing decimal numbers
=	Assign	assigns the value of the right operand to the left operand

Below example will explain the above arithmetic operation try it then make some changes to see the result

Example:

```
x = 15
y = 4
print('x + y =', x + y)
print('x - y =', x-y)
print('x * y =', x*y)
print('x / y =', x/y)
print('x ** y =', x**y)
print('x % y =', x % y)
print('x // y =', x//y)
```

Output:

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x ** y = 50625
x % y = 3
x // y = 3
```

2- Assignment Operators

Operators	Function	Example	Equivalent to
=	assigns the value of the right operand to the left operand	X = 5	X = 5
+=	adds the value of the right and left operand and assigns the total to the left operand	X+=5	X = X + 5
-=	subtract and deducts the value of the right operand from the value of the left operand and assigns the new value to the left operand	X-=5	X = X – 5
*=	multiply and multiplies the left and right operand and assigns the product to the left operand	X*=5	X = X * 5
/=	divides the left operand with the value of the right operand and assigns the quotient to the left operand	X/=5	X = X / 5
**=	performs exponential operation on the left operand and assigns the result to the left operand	X**=5	X = X ** 5

//=	floor division and performs floor division on the left operand and assigns the result to the left operand	X//=5	X = X // 5

Example:

```
x = 4
y = 7
z = 9
x+=2
y/=5
z**=3
print('x+=2 is', x)
print('y/=5 is', y)
print('z**=3 is', z)
```

Output:

```
x+=2 is 6
y/=5 is 1.4
z**=3 is 729
```

3- Relational or Comparison Operators

Relational operators evaluate values on the left and right side and return either True or False.

Operator	Meaning	Example
==	is equal to - True if both operands are equal	X == Y
<	is less than - True if left operand is less than the right	X < Y
>	is greater than - True if left operand is greater than the right	X > Y
<=	is less than or equal - True if left operand is less than or equal to the right	X <= Y

>=	is greater than or equal - True if left operand is greater than or equal to the right	X >= Y
!=	is not equal to - True if operands are not equal	X! = Y

Example:

```
x = 10
y = 12
print('x > y is', x>y)
print('x < y is', x<y)
print('x == y is', x==y)
print('x!= y is', x!=y)
print('x >= y is', x>=y)
print('x <= y is', x<=y)
```

Output:

```
x > y is False
x < y is True
x == y is False
x!= y is True
x >= y is False
x <= y is True
```

4- Logical Operators

Python supports Three logical operators (or), (and) (not).

Operator	Meaning	Example
and	True if both the operands are true	X and Y
or	True if either of the operands is true	X or Y
not	True if operand is false (complements the operand)	not X

Example:

```
x = True
y = False
print('x and y =', x and y)
print('x or y =', x or y)
print('not x =', not x)
```

Output:

```
x and y = False
x or y = True
not x = False
```

5- Bitwise Operands

Bitwise operands are treated as string of binary digits by bitwise operators. As the name implies, they function in a bit.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

6- Precedence of Python Operators

Python operators are evaluated according to their priority:

	Description	Operators
1	Exponentiation	**
2	Complement, unary plus, and minus	~, +, -
3	Multiplication, division, modulo, and floor division	*, /, %, //
4	addition and subtraction	+-
5	Right and left bitwise shift	>>, <<
6	Bitwise 'AND'	&
7	Regular `OR' and Bitwise exclusive 'OR'	|, ^

8	Comparison operators	<=, <>, >=
9	Equality operators	==, !=
10	Assignment operators	=, +=, -=, *-, /=, %=, **=
11	Identity operators	is, is not
12	Membership operators	in, not in
13	Logical operators	or, and, not

Chapter 7

Python's Built-in Functions

Python programming language has many useful built-in functions that make the programming simpler, more efficient and faster.

	Function	Description
1	abs()	Returns the absolute value of a number
2	all()	Returns True if all items in an iterable object are true
3	any()	Returns True if any unit is valid for each item in an iterable object
4	ascii()	Returns an item in a readable form. none-ascii characters are replaced using the escape character
5	bin()	Returns the binary version of a number
6	bool()	Returns the boolean value of the specified object
7	bytearray()	Returns an array of bytes
8	bytes()	Returns a bytes object
9	callable()	Returns True if the specified object is callable, otherwise False
10	chr()	Returns a character from the specified Unicode code.
11	classmethod()	Converts a method into a class method
12	compile()	Returns the specified source as an object, ready to be executed
13	complex()	Returns a complex number
14	delattr()	Deletes the specified attribute (property or method) from the specified object
15	dict()	Returns a dictionary (Array)

16	dir()	Returns a list of the specified object's properties and methods
17	divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
18	enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
19	eval()	Evaluates and executes an expression
20	exec()	Executes the specified code (or object)
21	filter()	Use a filter function to exclude items in an iterable object
22	float()	Returns a floating point number
23	format()	Formats a specified value
24	frozenset()	Returns a frozenset object
25	getattr()	Returns the value of the specified attribute (property or method)
26	globals()	Returns the current global symbol table as a dictionary
27	hasattr()	Returns True if the specified object has the specified attribute (property/method)
28	hash()	Returns the hash value of a specified object
29	help()	Executes the built-in help system
30	hex()	Converts a number into a hexadecimal value
31	id()	Returns the id of an object
32	input()	Allowing user input
33	int()	Returns an integer number
34	isinstance()	Returns True if a specified object is an instance of a specified object

35	issubclass()	Returns True if a specified class is a subclass of a specified object
36	iter()	Returns an iterator object
37	len()	Returns the length of an object
38	list()	Returns a list
39	locals()	Returns an updated dictionary of the current local symbol table
40	map()	Returns the specified iterator with the specified function applied to each item
41	max()	Returns the largest item in an iterable
42	memoryview()	Returns a memory view object
43	min()	Returns the smallest item in an iterable
44	next()	Returns the next item in an iterable
45	object()	Returns a new object
46	oct()	Converts a number into an octal
47	open()	Opens a file and returns a file object
48	ord()	Convert an integer representing the Unicode of the specified character
49	pow()	Returns the value of x to the power of y
50	print()	Prints to the standard output device
51	property()	Gets, sets, deletes a property
52	range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
53	repr()	Returns a readable version of an object

54	reverse()	Returns a reversed list
54	reversed()	Returns a reversed iterator
55	round()	Rounds a numbers
56	set()	Returns a new set object
57	setattr()	Sets an attribute (property/method) of an object
58	slice()	Returns a slice object
59	sorted()	Returns a sorted list
60	@staticmethod()	Converts a method into a static method
61	str()	Returns a string object
62	sum()	Sums the items of an iterator
63	super()	Returns an object that represents the parent class
64	tuple()	Returns a tuple
65	type()	Returns the type of an object
66	vars()	Returns the __dict__ property of an object
67	zip()	Returns an iterator, from two or more iterators

Example:

```
integer = -20
print('Absolute value of -20 is: ', abs(integer))
othertext = 'Python is interesting'
print('Ascii Result is:', ascii(othertext))
number = 5
print('The binary equivalent of 5 is:', bin(number))
print('The Character equivalent of 65 is:', chr(65))
print('The Float of 10 is:', float(10))
print('435 in hexadecimal is:', hex(435))
print("integer (123.23) is:", int(123.23))
testlist = [2, 5, 8]
print ('testlist length is:', len(testlist))
print ("The largest number is:", max(testlist))
print ("The smallest number is:", min(testlist))
lst = [10, 8, 12, 16, 14, 15]
lst.reverse()
print("The reversed list ",lst)
lst.sort()
print("The sorted list ",lst)
```

Output:

```
Absolute value of -20 is:  20
Ascii Result is: 'Python is interesting'
The binary equivalent of 5 is: 0b101
The Character equivalent of 65 is: A
The Float of 10 is: 10.0
435 in hexadecimal is: 0x1b3
integer (123.23) is: 123
testlist length is: 3
The largest number is: 8
The smallest number is: 2
The reversed list  [15, 14, 16, 12, 8, 10]
The sorted list  [8, 10, 12, 14, 15, 16]
```

Chapter 8

Python Flow Control

In the examples we have seen till now, there were always a series of statements made exactly top-down order executed by Python. What if you wanted to change the flow of how it works? For example, you want the program to take some decisions and do different things depending on different scenario, like printing the 'Good Morning' or the 'Good Evening' according to daytime? This is achieved with control flow statements – you might have guessed.

Python is made up of three control flow statements – if, for and while. These statements are common among programming languages and they are used to perform actions or calculations based on whether a condition is evaluated as true or false.

1- if...else Statement

The if statement is used to check a condition: if the condition is true, we run a block of statements (called the if-block), else we process another block of statements (called the else-block). Else the else clause executed.

Else clause is optional if we need use if without elif, or only if.

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

Example: If the number is positive, we print an appropriate message using if statement only.

```
num = -1
if num >= 0:
    print('Positive or Zero')
print ('END ')
```

Output:

```
END
```

Note: In the example above you see the Output is only (END) because the num = -1 and the *if* statement return false because the number is less than zero so that the *if* block will not executed.

Example: If the number is positive, we print an appropriate message using if statement only.

```
num = 2
if num > 0:
    print ('Positive or Zero')
print ('END ')
```

Output:

```
Positive or Zero
END
```

Note: here the number is larger than 0 so the *if* statement condition is fulfilled, the if condition gets executed.

Example: A program checks if the number is positive or negative and displays an appropriate message

```
num = 3
if num >= 0:
    print('Positive or Zero')
else:
    print('Negative number')
```

Output:

```
Positive or Zero
```

Note: In the above example, when num is equal to 3, the *if* condition is true, and the body of *if* is executed and the body of *else* is skipped.

Example: A program checks if the number is positive or negative and displays an appropriate message

```
num = -1
if num >= 0:
    print('Positive or Zero')
else:
    print('Negative number')
```

Output:

```
Negative number
```

Note: In the above example, num is equal to -1, the *if* condition is false so that the body of *if* is skipped and the body of *else* executed.

Example: Here we determine whether the number is positive, negative, or zero, and then print the corresponding message.

```
num = 3.4
if num == 0:
    print('Zero')
elif num > 0:
    print('Positive number')
else:
    print('Negative number')
```

Output:

```
    Positive number
```

Note: The *elif* is short for else if. It enables us to assess multiple terms. If False is the condition for if, the following *elif* block and so on forth will be checked.

Example: In this program, we check if the number is positive or negative or zero and display an appropriate message

```
num = -3.4
if num == 0:
        print('Zero')
elif num > 0:
        print('Positive number')
else:
        print('Negative number')
```

Output:

```
        Negative number
```

Note: If all the conditions are False, the else is carried out.

An *if...elif...else* statement can be contained within another *if...elif...else* statement. In computer programming, this is referred to as nesting. These statements can be nested inside of one another. The only way to determine the level of nesting is through indentation. They can get confusing, so they must be avoided unless necessary.

Example: In this program, we input a number then check if the number is positive or negative or zero and display an appropriate message. This time, we use nested *if* statement

```
num = float(input ('Enter a number: '))
if num >= 0:
        if num == 0:
                print('Zero')
        else:
                print('Positive number')
else:
        print('Negative number')
```

Output: Enter a number: 5

```
  Positive number
```

Note: Depending on the number you input an appropriate message will be display. As you can see, we input 5 and we got the output message (Positive number).

2- for Loop

The for loop is used to iterates over a sequence of (list, tuple, string) or other alterable objects. Iterating over a sequence is called transversal.

for loop syntax in Python:

```
for Val sequence:

        Body of for
```

Here, "Val" is the variable that takes into account the value of the item on each iteration in the sequence. It continues until in the sequence we reach the last item. The loop body is separated with indentation from the rest of the code.

Example: A Program that identifies the total of all number in list

```
# List of numbers
numbers = [6, 4, 3, 9, 5, 1, 7, 2, 12]
# variable to store the summation
sum = 0
# iterate over the list
for Val in numbers:
      sum = sum + Val
print("The sum is", sum)
```

Output:

```
The sum is 49
```

A for loop can have an optional else block as well if the items in the for loop's sequence run out, The else part is executed. If there is no break, the part continues.

Hence, a "for loop's *else* part runs if no break occurs.

```
for Val. sequence:
        body of for/statements
else:
        statements
```

Example:

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

Output:

```
0
1
5
No items left.
```

Note: Remember to keep it handy in case you need it.

3- while Loop

The while loop allows you to repeatedly execute a block of statements if the test expression (condition) is true. This loop is typically used when we don't know how many times we'll iterate ahead of time.

while loop syntax in Python

```
while test expression:
        body of while/statements
```

The test expression is checked first in the while loop. Only if the test expression is true, can the loop body be entered. The test expression is re-checked after one iteration. This process continues until the test expression evaluates to False.

In Python, the body of the while loop is determined through indentation. The body starts with indentation and the first un-indented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

Example: A Program to add natural numbers up to n numbers.

```
# sum = 1+2+3+...+n
n = 8
# initialize sum and counter
sum = 0
i = 1
while i <= n:
        sum = sum + i
        i = i+1    # update counter
# print the sum
print("The sum is", sum)
```

Output:

```
The sum is 36
```

As long as our counter variable i is less than or equal to n, the test expression in the above program will be True (8 in our program). In the loop's boy, we need to increase counter variable's value. This is vital information (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop). Finally, the result is displayed.

While loops, like for loops, can have an optional else block. If the condition becomes incorrect, it will be executed.

```
while test expression:
        body of while/statements
else:
        statements
```

Example: the use of else statement with the while loop counter = 0

```
counter = 0
while counter < 2:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output:

```
Inside loop
Inside loop
Inside else
```

Note: good to know it in case you need it.

4- break and continue

break and continue statements declarations may change normal loop flow. Loops run over a block of code until it's wrong, but occasionally we want to stop the current or even the entire iteration without checking the test expression. In these cases, break and continue declaration are used. The declaration of the break ends the loop that contains it. Program control flows directly after the loop corpus into the statements. If the break statement is in a nested loop, the break statement ends the loop on the inside.

Example: Use of break statement inside the loop

```
for val in "Python":
        if val=="t":
                break
        print(val)
print("The end")
```

Output:

```
P
y
The end
```

In this program, we iterate the "python" series. We break from the loop if the letter is t. as a result, we can see in our result that all of the letters up to t are written. The loop then ends for the current iteration only.

The continue statement is used to skip the rest of the code within a loop. Loop does not terminate but continues with the next iteration.

Example: A Program to show the use of continue statement inside loops

```
for val in "Python":
        if val == "t":
                continue
        print(val)
print("The end")
```

Output:

```
P
y
h
o
n
The end
```

Note: The program continues with the loop, if the string is t, not executing the rest of the block. Hence, we see in our output that all the letters except t gets printed.

Chapter 9

User Defined Functions

A function is a group of statements that perform a specific task, and it's a popular structuring feature that lets you use a piece of code repeatedly in different parts of a program.

The use of functions improves a program's clarity and comprehensibility and makes programming more efficient by reducing code duplication and breaking down complex tasks into more manageable pieces. Functions are also known as routines, subroutines, methods, procedures, or subprograms. They can be passed as arguments, assigned to variables, or stored in collections.

Syntax of Function

```
def function_name (parameters):
        """docstring"""
        body of function/statements
return statement
```

A function definition consists of the following components.

- Keyword **def** that marks the start of the function header.
- A function_name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python as follows.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body.
- An optional return statement to return a value from the function.

Note: Function bodies can have more than one return statement which may be placed anywhere within the function block. Return statements end the function call and return the value of the expression after the return keyword. A return statement with no expression returns the special value 'None'. In the absence of a return statement within the function body, the end of the function is indicated by the return of the value 'None'.

Example:

```
def say_hello():
# A function to print hello world
      print('hello world')
# End of function

say_hello() # call the function
say_hello() # call the function again
```

Output:

```
hello world
hello world
```

In the example above, we define a function called "say_hello" using the syntax as explained before. This function takes no parameters and hence there are no variables declared in the parentheses.

Notice that we can call the same function twice which means we do not have to write the same code again.

Example:

```
def print_max(a, b):
# A function to check and print the corresponding message
   if a > b:
         print(a, 'is maximum')
   elif a < b:
         print(b, 'is maximum')
   else:
         print(a, 'is equal to', b)
```

```
# directly pass literal values
print_max(3, 4)

x = 5
y = 7
# pass variables as arguments
print_max(x, y)
```

Output:

```
4 is maximum
7 is maximum
```

Here, we define a print_max function, which uses a and b parameters. This is used to detect the larger number by a simple ***if..else*** statement, followed by the larger The first time we call print_max, we pass the numbers directly as arguments. The function is called with variables as arguments in the second case. print_max(x, y) causes the value of argument x to be assigned to parameter a and the value of argument y to be assigned to parameter b. The print_max function works the same way in both cases.

Example:

```
def maximum(x, y):
#A function to find maximum number and print a message
    if x > y:
        return x
    elif x < y:
        return y
    else:
        return 'The numbers are equal'

print(maximum(2, 3))
```

Output:

```
3
```

The maximum function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple ***if..else*** statement to find the greater value and then returns that value.

Note that a return statement without a value is equivalent to return None. In python None is a special type which used to represent the variable has no value.

Variables' Scope and Lifetime

Scope is the area of a program where the variable is identified. Outside of the function, parameters and variables are not visible. As a result, they only have a local scope.

A variable's lifetime is the time it spends in memory. Variables within a function have a lifetime equal to the function's lifetime. A function therefore does not remember the variable value in its previous calls.

Example: to show the scope of a function variable.

```
def my_func():
    x = 10
    print("Value inside function : ", x)

x = 20
my_func()
print("Value outside function: ", x)
```

Output:

```
Value inside function : 10
Value outside function: 20
```

Here, we can see that the value of x is 20 initially. Even though the function my_func() changed the value of x to 10, it did not affect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

Within the function, we can read these values but not alter(write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global**.

Example:

```
x = 50
def func():
        global x
        print(' x is ', x)
        x = 2
        print('Changed global x to', x)

func()
print('Value of x is ', x)
```

Output:

```
x is 50
Changed global x to 2
Value of x is 2
```

Using the global statement to state that x is a global variable, the shift reflects when you assign a value to x within the feature when using the value of x in the main block. Using the same global statement, for example global x, y, z, you may set more than one world variable.

Chapter 10

Classes and Object-Oriented Programing

Python is a multi-paradigm programming language. Meaning, it supports different programming approach. One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

There are to features to an object: behavior and attribute. Let's take a case in point: It's an object of Parrot. The attributes are name, age, color. Dancing is behavior.

The OOP definition in Python is based on a few simple principles:

(a) Inheritance: Using data from a new class without having to change an existing class.
(b) Encapsulation: Hiding the private details of a class from other objects.
(c) Polymorphism: A concept of using common operation in different ways for different data input.

What are classes and objects in Python?

Object is merely data collection (variables) and methods (functions) which act on such data. And class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.

Defining a Class in Python

In Python, we define a class with the keyword class, just as we do functions. The first string, known as the docstring, contains a brief description of the class. Although it is not required, it is strongly advised.

Have a look for the below simple class definition.

```
class MyNewClass:
        '''This is a docstring. a new class created'''
        pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores __ . For instance, __doc__ return the class's docstring.
A new class object with the same name is generated as soon as a class is defined. We can access the various attributes using this class object.

Example:

```
class Kid:
        '''This is a kid class'''
        age = 8
        def greet(self):
                print('Hello')

# Output: 8
print(Kid.age)

# Output: 'This is a Kid class'
print(Kid.__doc__)
```

Output:

```
8
This is a kid class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes. It can also be used to create new object instances

(instantiation) of that class. The procedure to create an object is like a function call.

```
henry = Kid()
```

This will create a new instance object named henry. We can access attributes of objects using the object name prefix.

Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class. This means to say, since Kid.greet is a function object (attribute of class), Kid.greet will be a method object.

Example:

```
class Kid:
        '''This is a kid class'''
        age = 8
        def greet(self):
                print('Hello')

# create a new Kid
henry = Kid()

# Calling function greet()
# Output: Hello
henry.greet()
```

Output:

```
Hello
```

You may have noticed the self parameter in function definition inside the class but, we called the method simply as henry.greet() where there is no argument. It was still functional. That is why the object is passed as the first argument whenever it calls an object its method. still worked. Therefore henry.greet() translates into Kid.greet(henry).

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument

list that is created by inserting the object of the method before the first argument as a result, the first argument of the class function must be the object itself. This is commonly referred to as self. It can give a different name, but we strongly advise that you stick to the rules.

Constructors in Python

Class functions that begin with a double underscore __ are referred to as special functions because they have unique properties. One particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Example:

```
class ComplexNumber:
    def __init__(self, r = 0, i = 0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object
c1 = ComplexNumber(2,3)

# Call get_data( ) function
# Output: 2+3j
c1.get_data()

# Create another ComplexNumber object
# and create a new attribute 'attr.'
c2 = ComplexNumber(5)
c2.attr=10
```

```
# Output: (5, 0, 10)
print ((c2.real ,c2.imag, c2.attr))

# but c1 object doesn't have attribute 'attr'
# Attribute Error: 'ComplexNumber' object has no attribute 'attr'
print(c1.attr)
```

Output:

```
2+3j
(5, 0, 10)
Attribute Error: 'ComplexNumber' object has no attribute 'attr'
```

In the above example, a new class is defined to represent complex numbers. It contains __init__() and get_data() functions for initializing variables and for properly displaying the number respectively.

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute for object c2 and we read it as well. But this did not create that attribute for object c1.

Python Inheritance

In object-oriented programming, inheritance is a valuable tool. It helps us to create a class that inherits all of the features of the parent class while still allowing us to implement new functionality.

Inheritance is the process of creating a new class with few to no changes to current one. The new class is known as the derived (or child) class, whereas the one it inherits from is known as the base (or parent) class.

Python Inheritance Syntax

```
class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class
```

Example: Consider the following scenario: A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range (no_of_sides)]
    def inputSides (self):
        self.sides = [float (input ("Enter side "+str(i+1) +": "))
                      for i in range (self. n)]
    def dispSides(self):
        for i in range (self. n):
            print("Side ", i+1,"is", self.sides[i])
```

The number of sides, n, and magnitude of each side are stored as a list, sides, in this class's data attributes.

Method inputSides() takes the magnitude of each side, and dispSides() does the same.

A triangle is a three-sided polygon. As a result, we can construct a "Triangle" class that inherits from Polygon. As a result, all of the attributes available in the Polygon class are also available in Triangle. We don't need to define them again (code re-usability). Triangle is defined as follows.

```
class Triangle (Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a) *(s-b) *(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is the complete program code.

Complete Code:

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range (no_of_sides)]

    def inputSides(self):
        self.sides = [float (input ("Enter side "+str(i+1) +": ")) for i in
                      range (self. n)]

    def dispSides(self):
        for i in range (self. n):
            print("Side", i+1,"is", self.sides[i])

class Triangle (Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a) *(s-b) *(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)

t = Triangle()
t.inputSides()
t.dispSides()
t.findArea()
```

Output:

```
    Enter side 1: 3
    Enter side 2: 5
    Enter side 3: 4
```

```
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0
The area of the triangle is 6.00
```

Even though we did not define the methods for class Triangle such as inputSides() or dispSides(), we could use them.

If the class doesn't find an attribute, the base class search continues. Recursively, this occurs when the base class itself comes from other classes.

Method Overriding in Python

In the above example, Triangle and Polygon (__init__ () method was defined). When this occurs, the derived class method overrides the basic class. That means that __init__() is preferable to Triangle than Polygon.

In general, we tend to extend this definition instead of just replacing it when overriding a base method. The same happens by calling the basic class method from that of the derivative class (calling polygon). The integrated function super() would be a better choice(). Then, super().__init__(3) equates with and is preferred to Polygon.__init__(self,3)

Inheritance are checked with two build-in functions isinstance() and issubclass(). The isinstance() function is returned True when an object is an instance of the class or other classes it derives. Each Python class inherits from the object of the base class.

```
>>> isinstance(t,Triangle)
True
>>> isinstance(t,Polygon)
True
>>> isinstance(t,int)
False
>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon, Triangle)
False
>>> issubclass(Triangle, Polygon)
True
>>>issubclass(bool, int)
True
```

Chapter 11

Python Turtle Library

turtle is a python feature that provides a virtual canvas to let users create pictures and shapes. The turtle method is a pre-installed Python library that comes packed with the standard Python package. The onscreen pen which is used to draw is called a turtle, hence the library's name. The Python turtle library is a helpful tool for new programmers to get accustomed to Python in an engaging and enjoyable manner.

Beyond being used for drawing, designing and imaging: turtle is an effective way to teach the basics of coding to young kids since it's entertaining and can easily give new programmers the drift of Python and programming in general. In addition to these, turtle is used by game artists and animators to generate mini-games and animations.

Python library can be identified as a core module collection or an easy-to-use code chunk.

A program must first import a library module before using it. We use an **import** statement to make use of the functions in a module, in our case the turtle library. The import statement is made up of the import keyword and the name of the library/module.

```
import turtle
```

The next step in programming with turtle is to create a different window known as the screen which will be used to carry out the functions we need to perform. In order to create this screen we will need to initialize a variable for it.

```
s = turtle.getscreen()
```

Variables are storage locations which contain an unknown quantity or information which will later be used or referred to in our program. Initializing a variable is simply assigning a starting value to it. For this article let's call our screen t. Variables aren't constant and might be changed in the process of designing and executing our program.

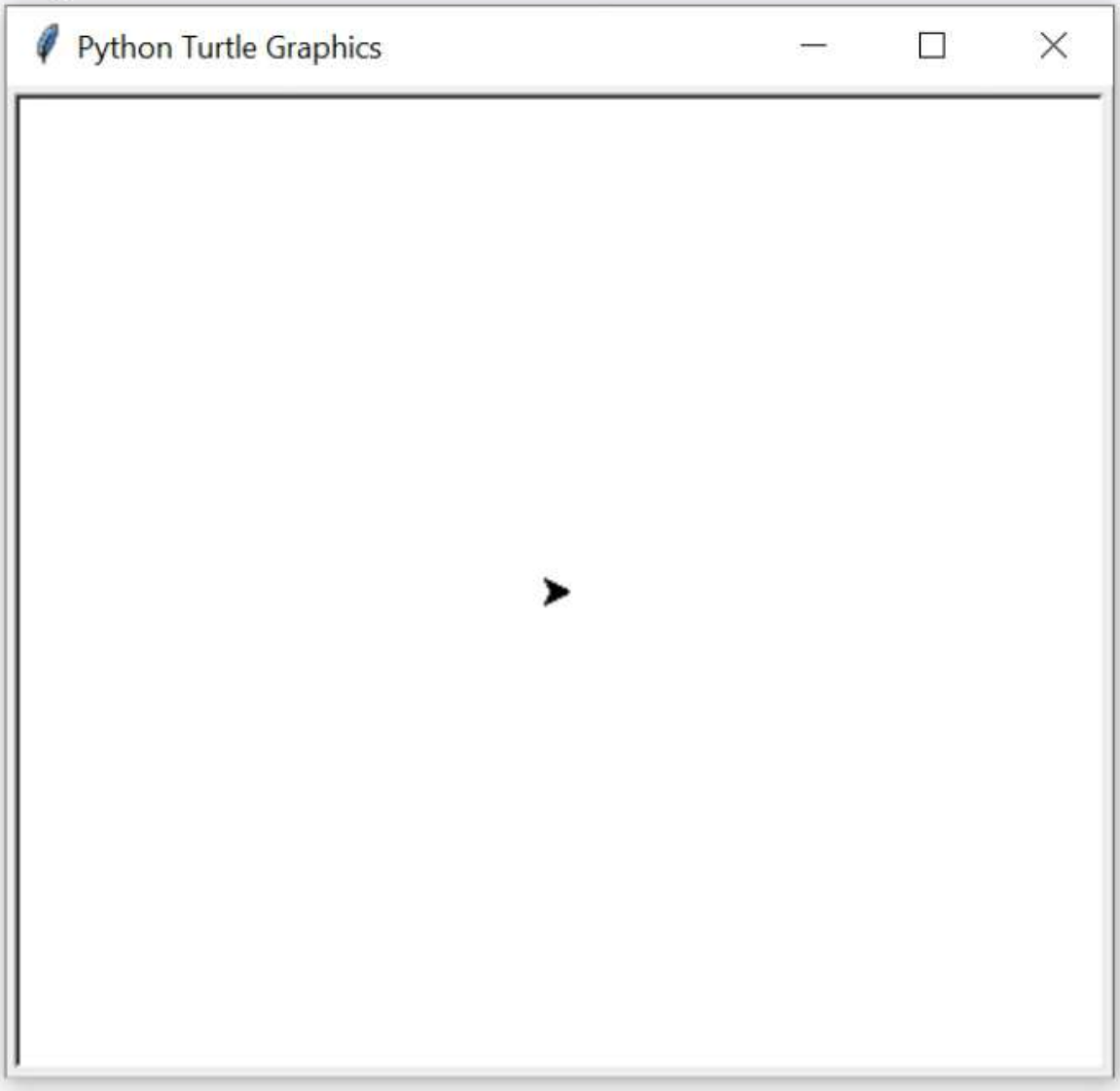
```
t = turtle.Turtle()
```

After you have finished initializing you can use the variable to refer to your screen throughout your entire program. Now that you've finished the prerequisites you can start programming and drawing on your canvas with your turtle that actually has certain adjustable attributes like color, size and speed. Unless you code otherwise, the turtle always faces one directions and moves in that direction as well.

Example:

```
import turtle
s = turtle.getscreen()
t = turtle.Turtle()
```

Output Screenshot:



Programming With turtle

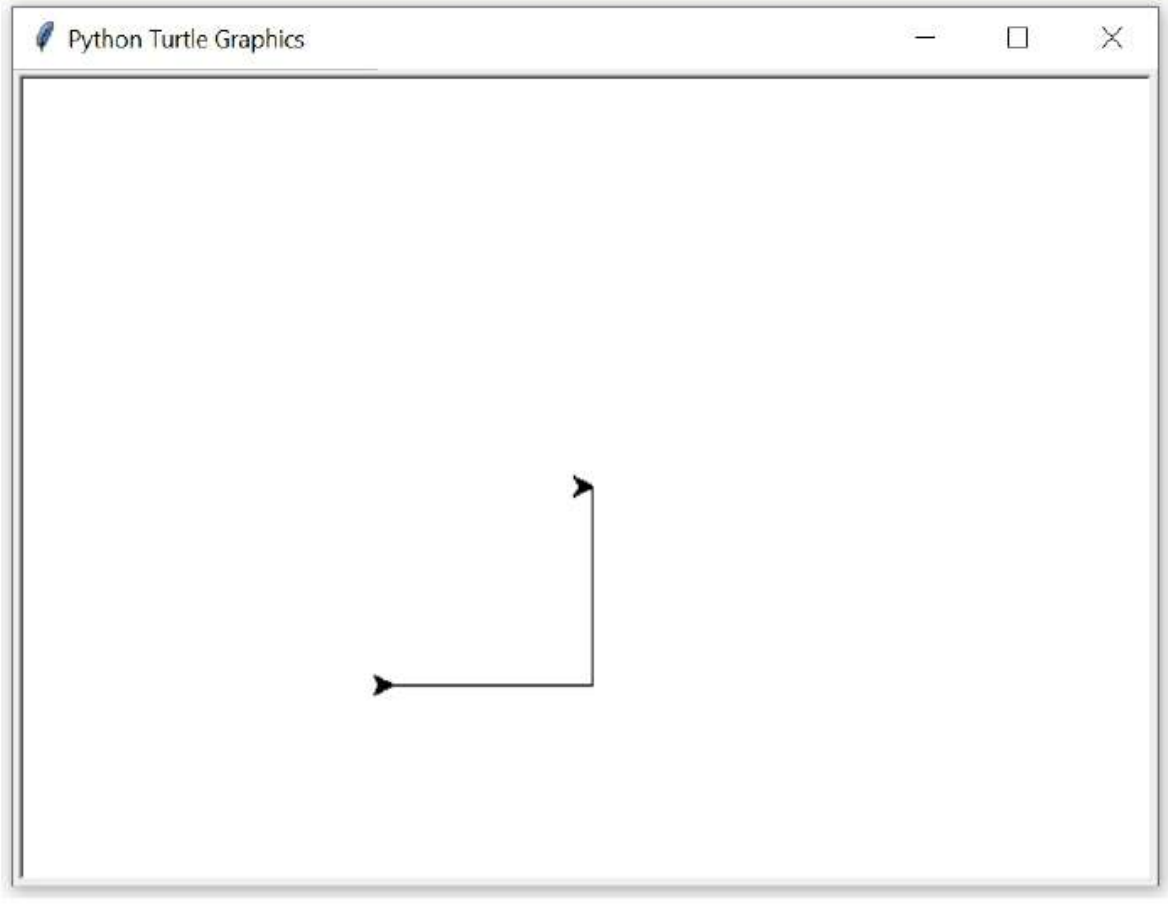
Here we will discuss how to make the turtle move in the direction we want it to go, how to customize our turtle's attributes and it's environment, and other extra commands which are able to perform different unique tasks.

The turtle can move in four directions. These are Forward, Backward, Left and Right. You can change the direction of the turtle by turning it to one of these four directions by a certain degree. For instance:

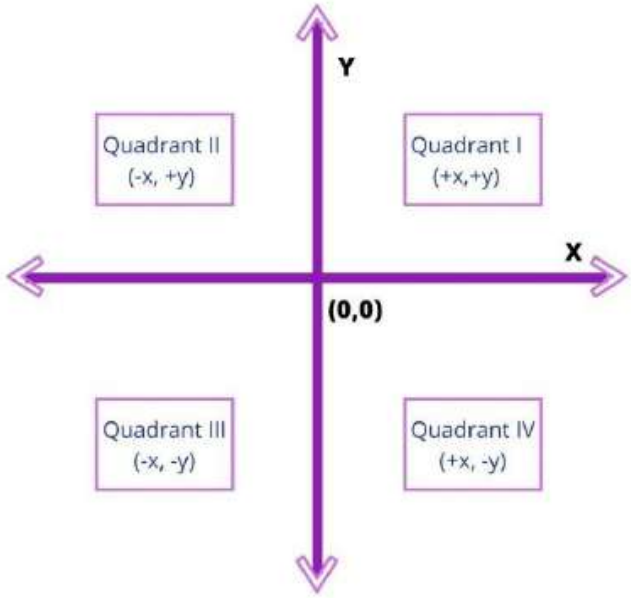
Example:

```
import turtle
s = turtle.getscreen()
t = turtle.Turtle()
t.right(90)
t.forward(100)
t.left(90)
t.backward(100)
```

Output Screenshot:



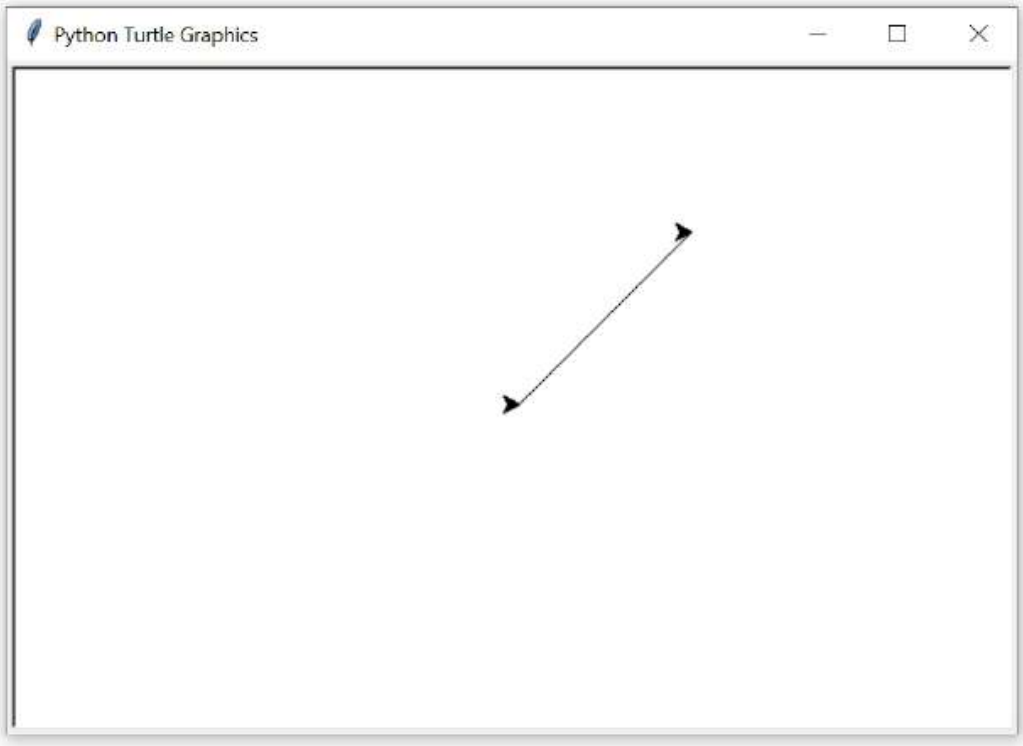
Drawing lines on the screen is also possible with the help of coordinates. By dividing the screen into four quadrants the turtle is initially at (0,0) and is called Home.



To move the turtle we use .goto() and enter the coordinates as such:

```
t.goto(100,100)
```

Your output will look like this:



To bring the turtle to its initial position we use:

```
t.home()
```

Turtle can be programmed to create different shapes and images.

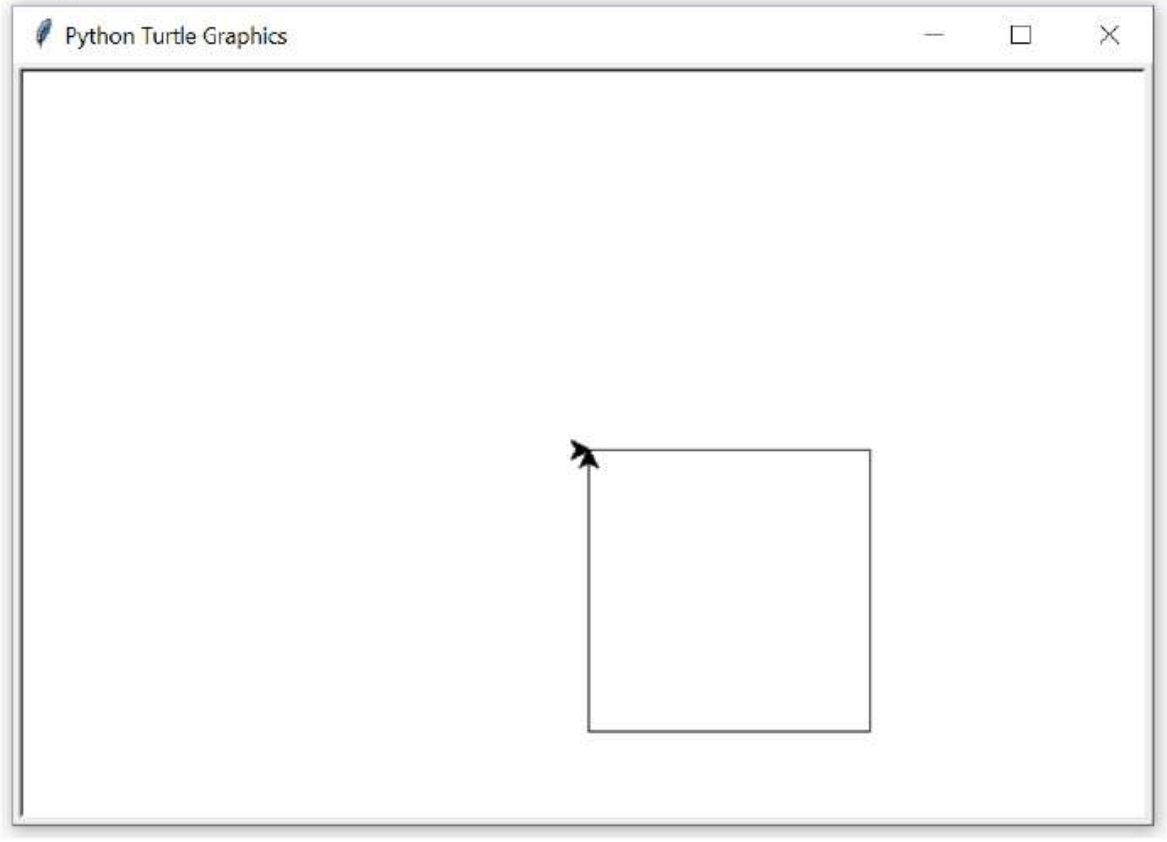
- To draw polygons, we use a combination of lines and angles. For instance:

Example:

```
import turtle
s = turtle.getscreen()
t = turtle.Turtle()
t.fd(150)
t.rt(90)
t.fd(150)
t.rt(90)
t.fd(150)
t.rt(90)
t.fd(150)
```

Note: fd means Forward, rt means right. (Using Abbreviation)

Output Screenshot:



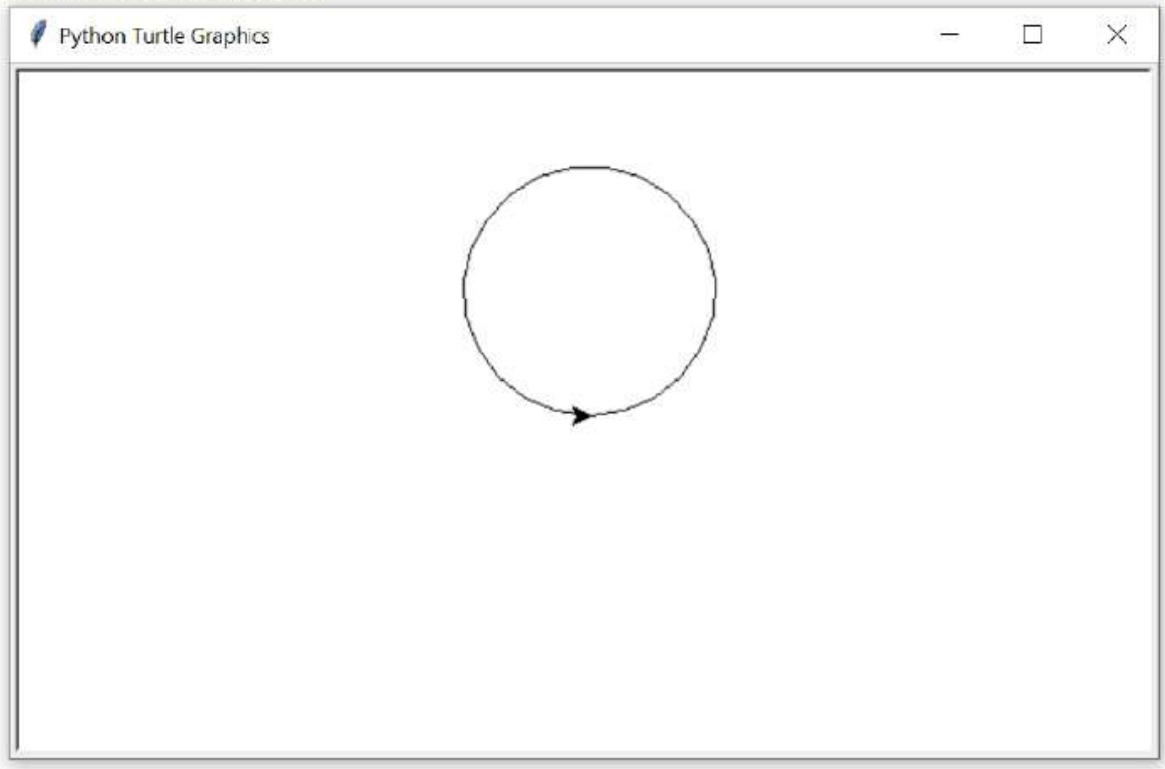
- To draw preset figures, such as a circles and dots we use different types of commands. For example:

Example:

```
import turtle
s = turtle.getscreen()
t = turtle.Turtle()
t.circle(70)
```

Note: number 70 here is the circle radius.

Output Screenshot:

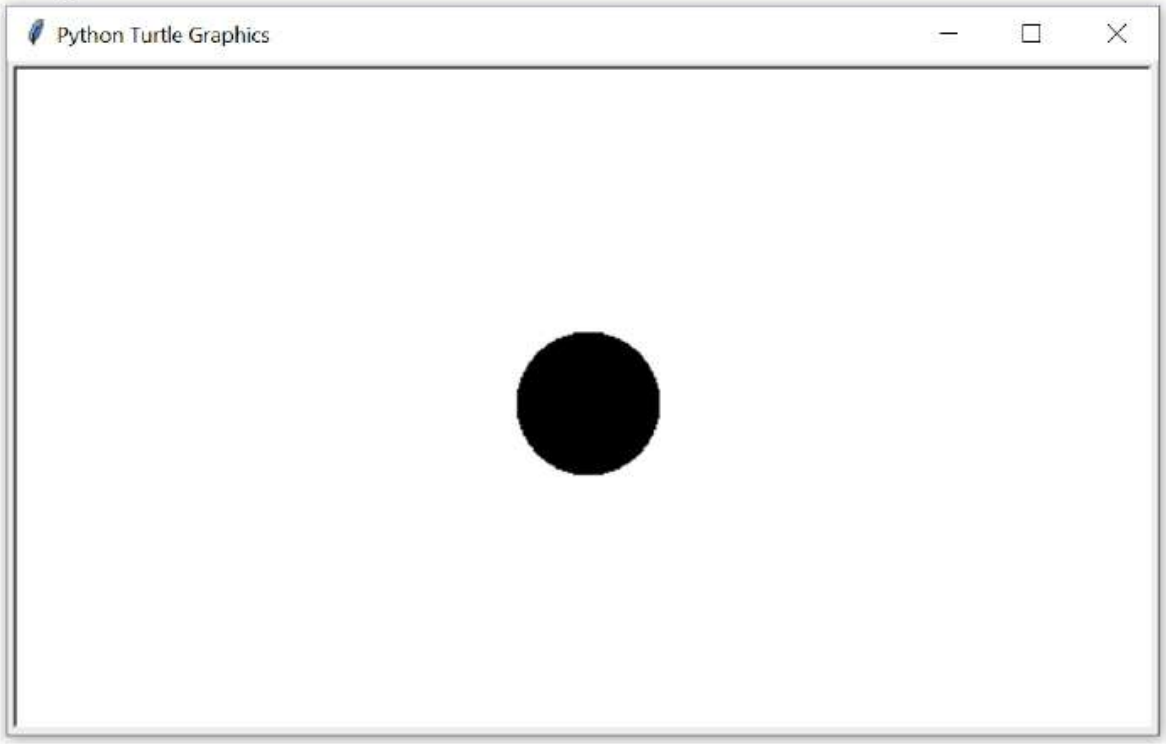


Example:

```
import turtle
s = turtle.getscreen()
t = turtle.Turtle()
t.dot(80)
```

Note: number 80 here is the dot diameter.

Output Screenshot:



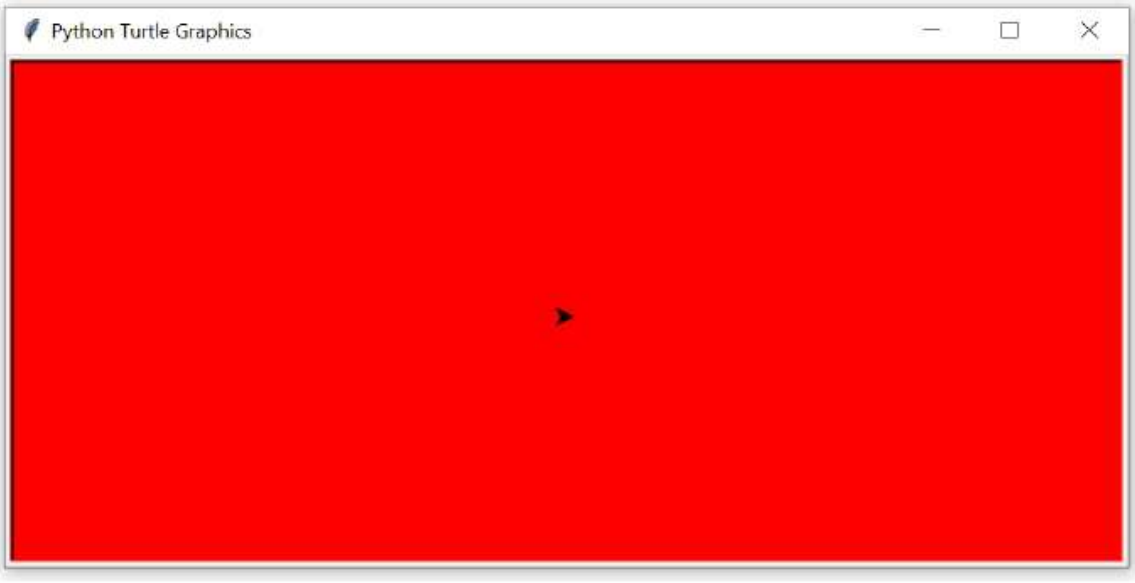
- To change the screen color, we use .bgcolor() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.bgcolor('red')
```

Note: You can replace the color red with names or hex codes of other colors.

Output Screenshot:



- To change the screen title, we use turtle.title() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
```

Output Screenshot:

- To change the size of our onscreen turtle we use .shapesize() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shapesize(1,5,10)
```

Note: The numbers passed as parameters with the function here represents as to how the turtle entailing the stretch length, stretch width and the outline width.

Output Screenshot:

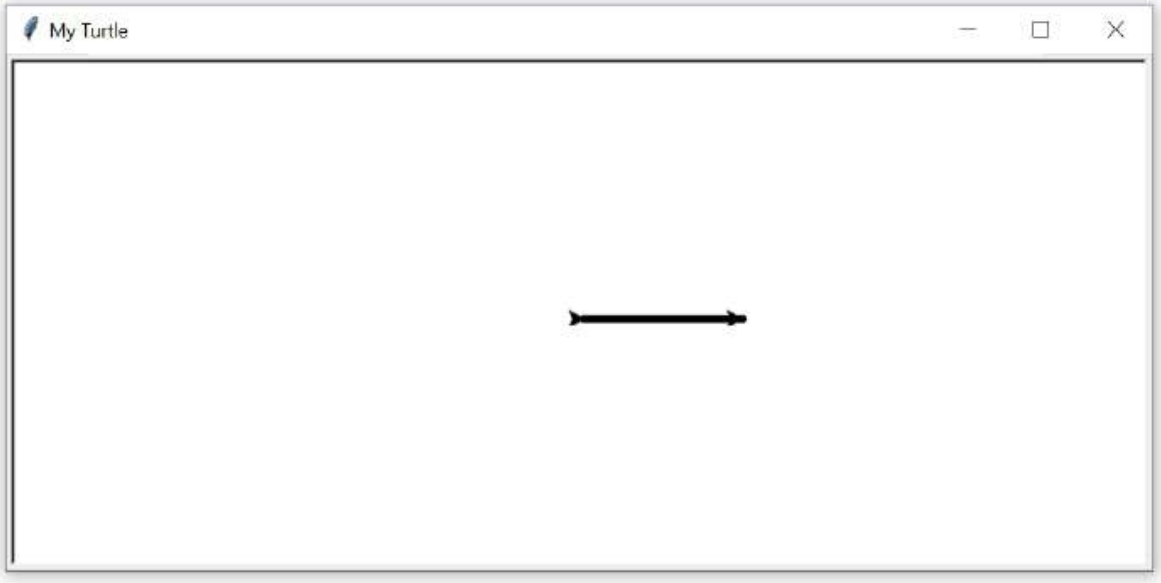
- To change the pen size or if we need to increase or decrease the thickness of the pen we use .pensize() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.pensize(5)
t.forward(100)
```

Note: The numbers in the bracket indicates how many times we want the pen increased in comparison with the original.

Output Screenshot:



- To change the color of the turtle and the color of the pen we use the following commands:

```
t.fillcolor("red")
t.pencolor("green")
```

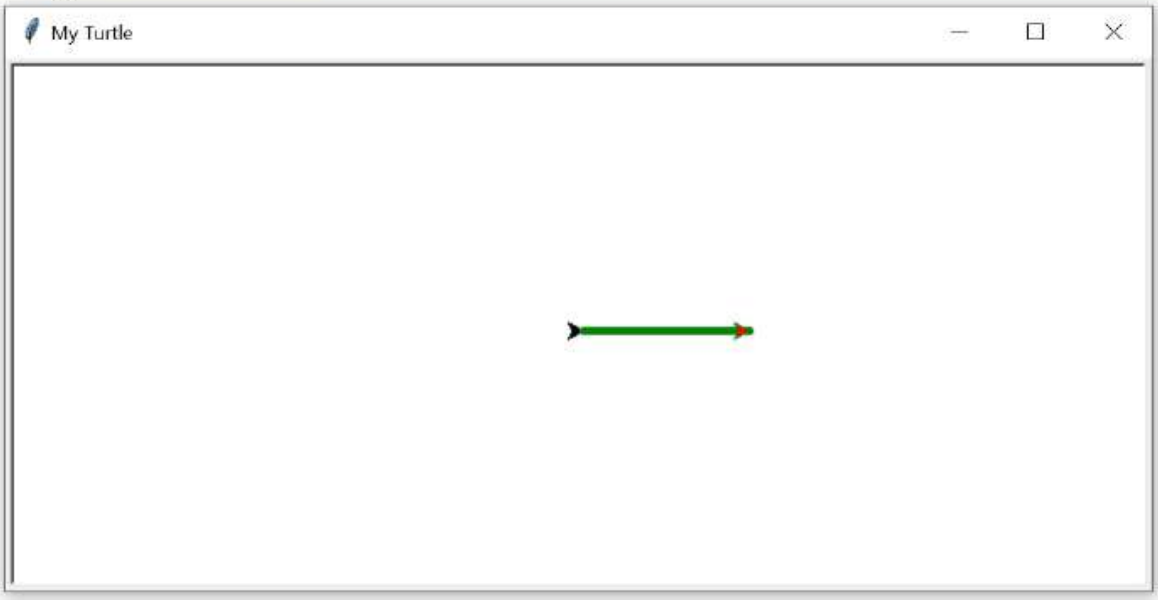
Note: To change the color of both we use the following code:

```
t.color("green", "red")
```

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.color("green", "red")
t.pensize(5)
t.forward(100)
```

Output Screenshot:

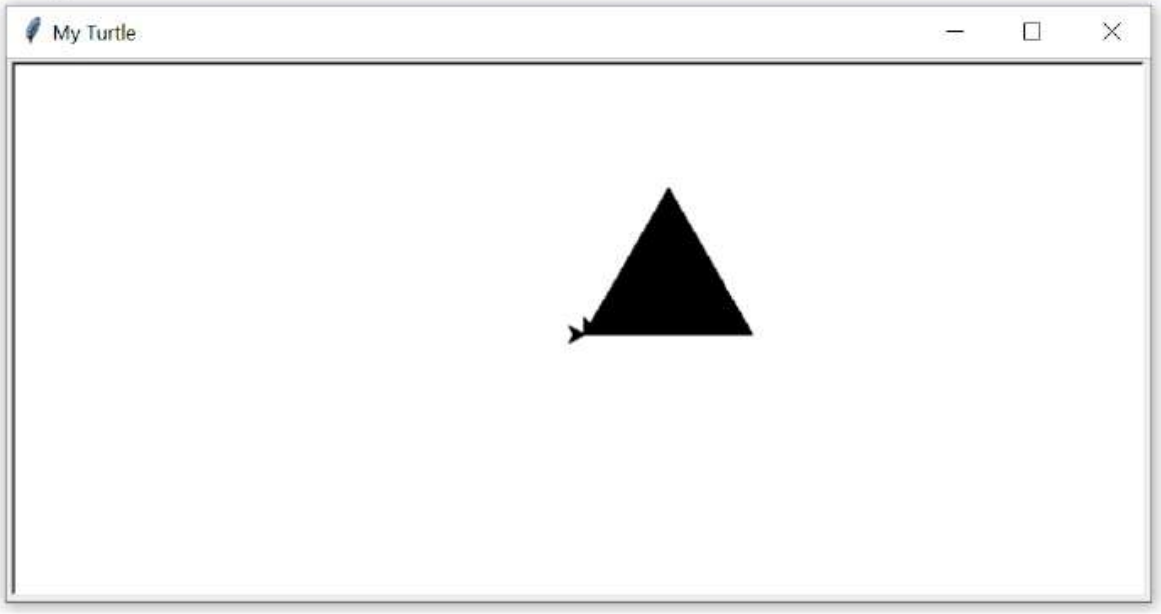


- We can use the Python turtle library to add color to our drawings. For instance, if we want to color a triangle with a solid color, we use the following code:

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.begin_fill()
t.fd(100)
t.lt(120)
t.fd(100)
t.lt(120)
t.fd(100)
t.end_fill()
```

Output Screenshot:

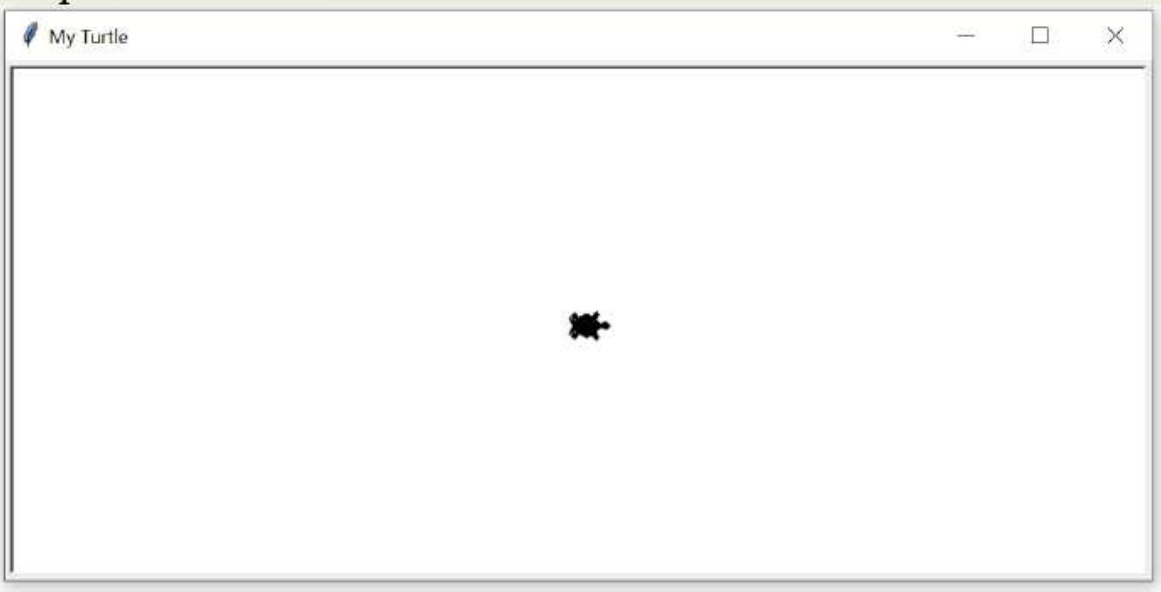


- To change the turtle shape we use .shape() command:

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shape("turtle")
```

Output Screenshot:



Note: We can change the turtle into a square, arrow, circle, turtle, triangle or we can use the classic option the turtle shape will change accordingly.

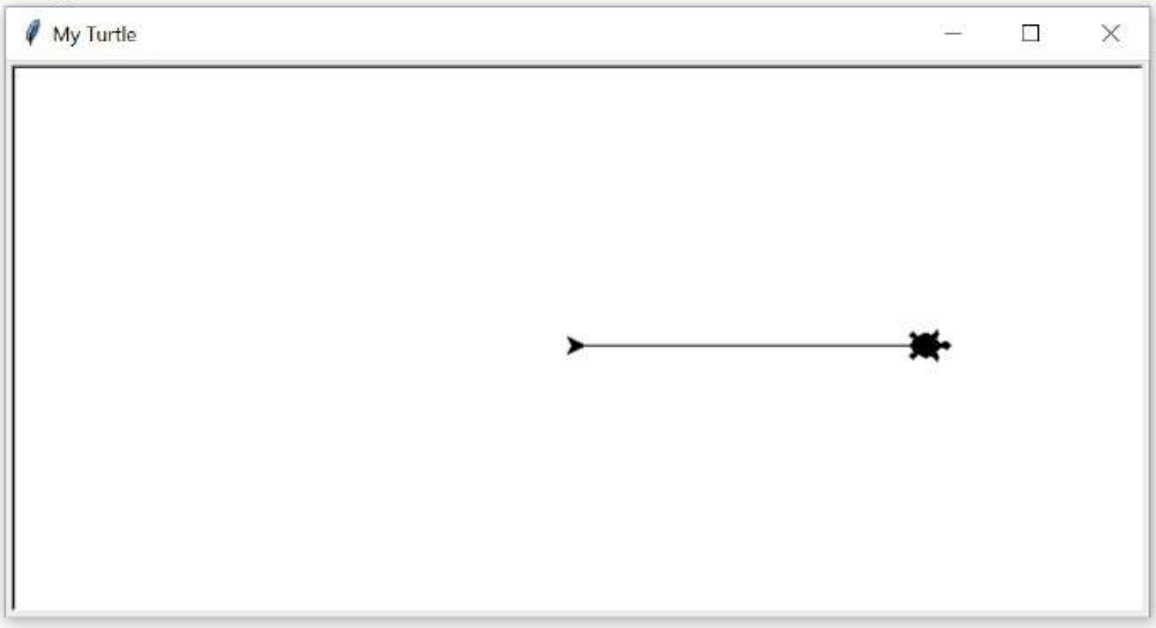
- To change the speed of the turtle we use .speed() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shape("turtle")
t.speed(1)
t.forward(100)
t.speed(10)
t.forward(100)
```

Note: Speed range is from 0 (slowest) to 10 (highest).

Output Screenshot:



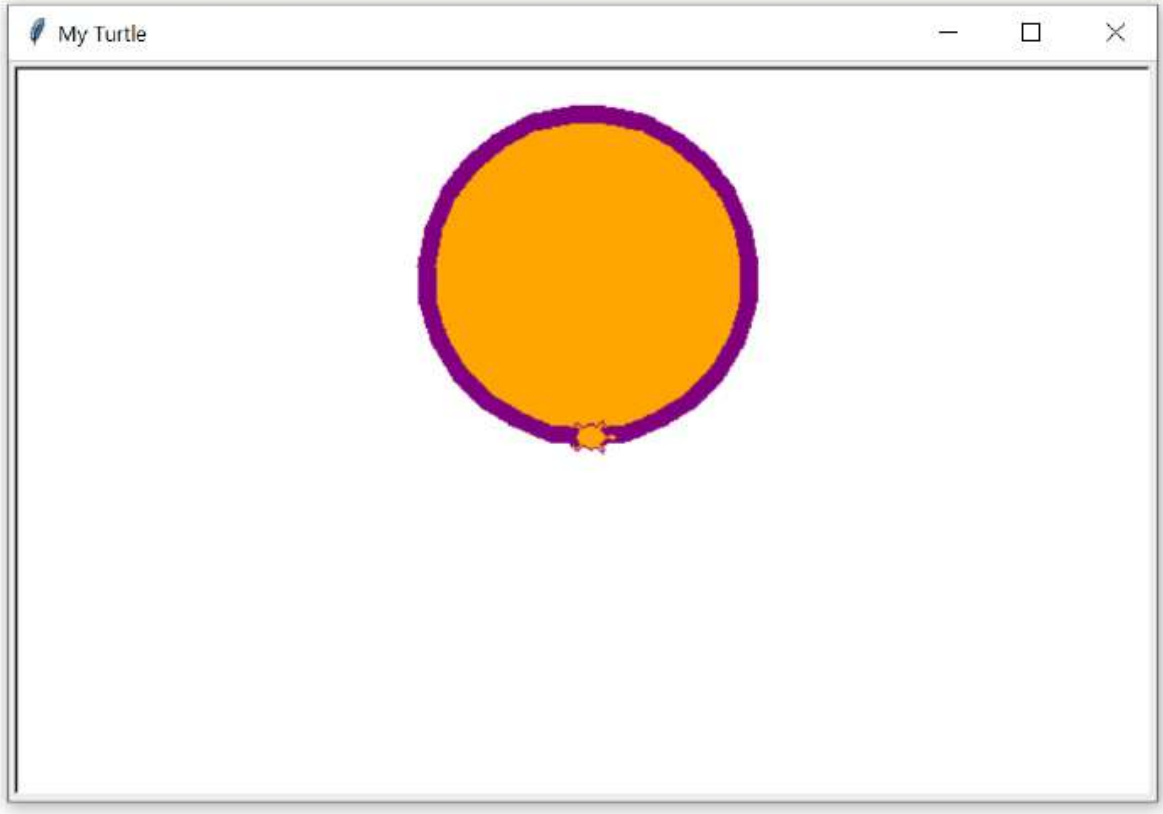
- To customize your turtle with all of the aforementioned properties we use the following code.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shape("turtle")
t.pen(pencolor="purple", fillcolor="orange", pensize=10, speed=9)
t.begin_fill()
t.circle(90)
t.end_fill()
```

Note: Speed range is from 0 (slowest) to 10 (highest).

Output Screenshot:



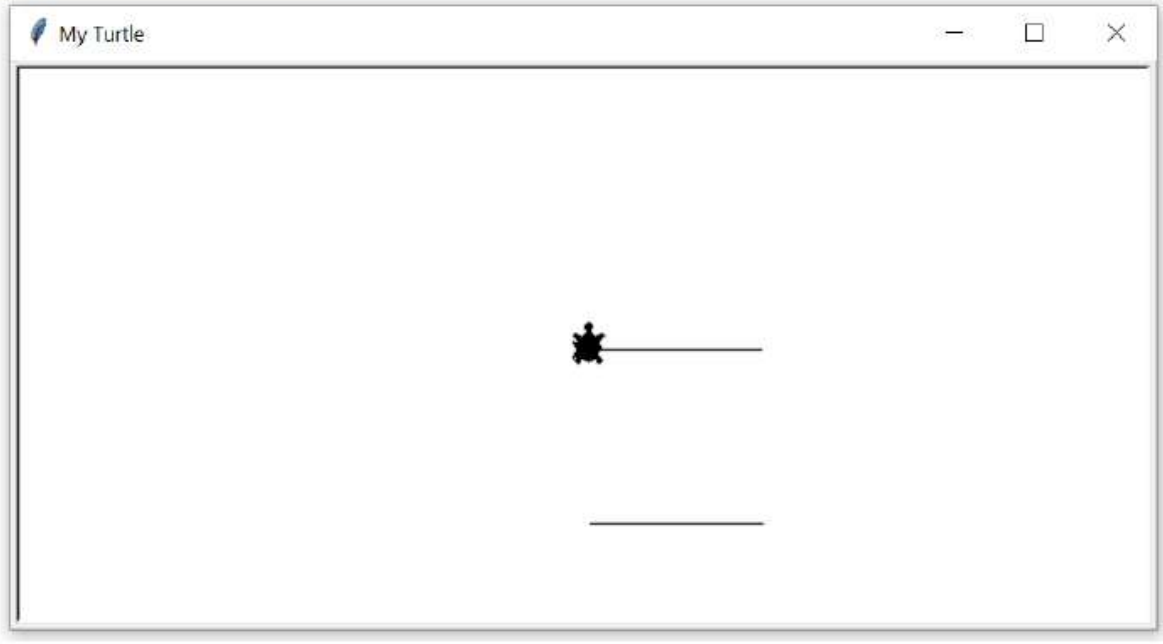
- The following commands are used to pass your pen up and down the screen without drawing anything:

```
t.penup()
t.pendown()
```

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shape("turtle")
t.fd(100)
t.rt(90)
t.penup()
t.fd(100)
t.rt(90)
t.pendown()
t.fd(100)
t.rt(90)
t.penup()
t.fd(100)
t.pendown()
```

Output Screenshot:



- To undo a previous task, we use the following command.

```
t.undo()
```

- To clear the screen, we use the following command

```
t.clear()
```

- To reset your turtle to its default settings we use the following command.

```
t.reset()
```

- To leave a stamp of your turtle on the screen we use the following command.

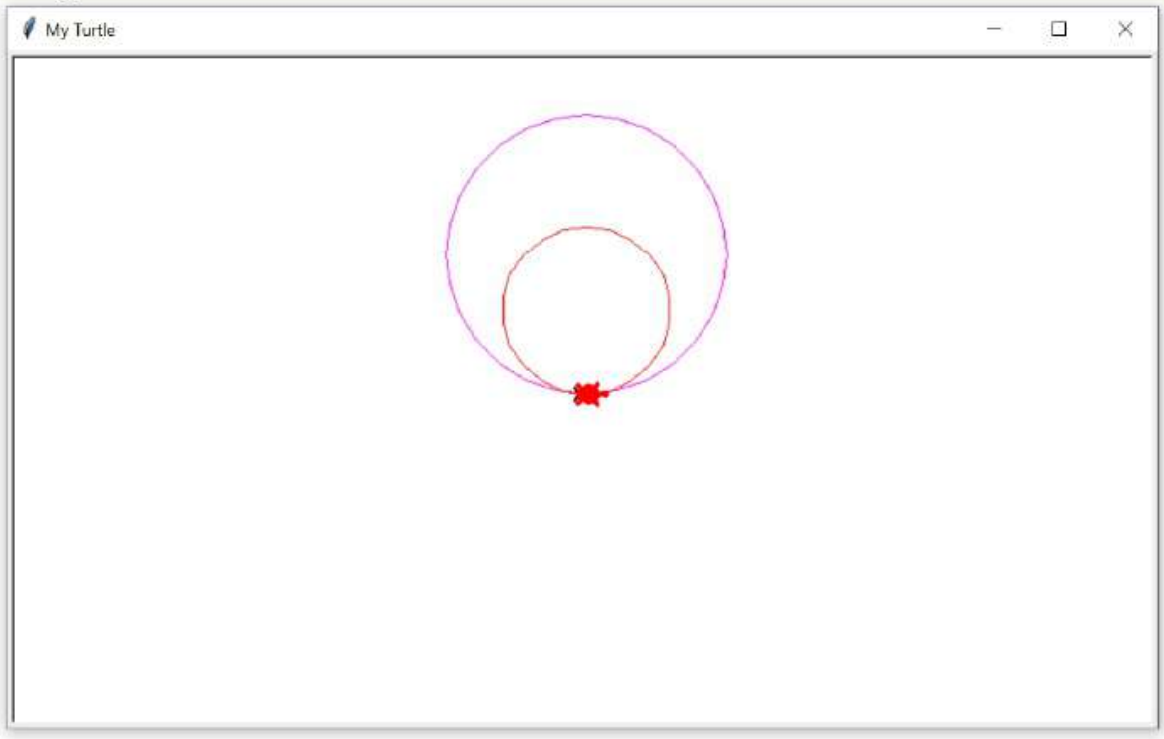
```
t.stamp()
```

- To create a clone "c" of your turtle then we use .clone() command.

Example:

```
import turtle
s = turtle.getscreen()
turtle.title("My Turtle")
t = turtle.Turtle()
t.shape("turtle")
c = t.clone()
t.color("magenta")
c.color("red")
t.circle(100)
c.circle(60)
```

Output Screenshot:



Chapter 12

Cover up previous knowledge

Example: Python program to find union and intersection of two array using set()

```
def printUnion(arr1, arr2):
    hs = set()

    # Insert the elements of arr1[] to set hs
    for i in arr1:
        hs.add(i)
    # Insert the elements of arr2[] to set hs
    for j in arr2:
        hs.add(j)

    # Print the Union array
    print("Union:")
    for i in hs:
        print(i, end=" ")
    print("\n")

def printIntersection(arr1, arr2):
    hs = set()

    # Insert the elements of arr1[] to set S
    for i in arr1:
        hs.add(i)

    # Print the intersection array
    print("Intersection:")
    for j in arr2:
        # If the element is in hs then print it
        if j in hs:
            print(j, end=" ")

# The two arrays
arr1 = [7, 1, 5, 2, 3, 6]
arr2 = [3, 8, 6, 20, 7]
```

```
# Function call
printUnion(arr1, arr2)
printIntersection(arr1, arr2)
```

Output:

```
Union:
1 2 3 5 6 7 8 20

Intersection:
3 6 7
```

Example: A program to find the nth Fibonacci number

```
def Fibonacci(n):
        if n<= 0:
                print("Incorrect input")
        # First Fibonacci number is 0
        elif n == 1:
                return 0
       # Second Fibonacci number is 1
        elif n == 2:
                return 1
        else:
                return Fibonacci(n-1)+Fibonacci(n-2)

num = int(input("Enter Number"))
print(Fibonacci(num))
```

Output: For Example we give number 5

```
3
```

Example: Bubble sort

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1] :
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
        print ("% d" % arr[i])
```

Output:

```
Sorted array is:
 11
 12
 22
 25
 34
 64
 90
```

Example: Find yesterday's, today's, tomorrow's date

```
# Import datetime and timedelta class
from datetime import datetime, timedelta

# Get today's date
presentday = datetime.today()

# Get Yesterday
yesterday = presentday - timedelta(1)

# Get Tomorrow
tomorrow = presentday + timedelta(1)

print("Yesterday = ", yesterday.strftime('%d-%m-%Y'))
print("Today = ", presentday.strftime('%d-%m-%Y'))
print("Tomorrow = ", tomorrow.strftime('%d-%m-%Y'))
```

Output:

```
Sorted array is:
Yesterday =  18-06-2021
Today =  19-06-2021
Tomorrow =  20-06-2021
```

Example: Find words greater than a given length

```
def greaterthan(k, str):

    # create the empty string
    string = []

    # split the string where space is comes
    text = str.split(" ")
```

```
        # iterate the loop till every substring
        for x in text:

                # check sub string length is greater than K
                if len(x) > k:
                        # append this sub string in string list
                        string.append(x)

        return string

# Given Length
k = 3
str ="Excuse me may I help you"
print("Words length greater than 3 is : ",greaterthan(k, str))
```

Output:

```
Words length greater than 3 is :  ['Excuse', 'help']
```

Example: Print duplicates from a list of integers

```
def Repeat(x):
        lsize = len(x)
        repeated = []
        for i in range(lsize):
                k = i + 1
                for j in range(k, lsize):
                        if x[i] == x[j] and x[i] not in repeated:
                                repeated.append(x[i])
        return repeated

list1 = [10, 20, 30, 20, 20, 30, 40, 50, -20, 60, 60, -20, -20]
print ("Duplicate: ",Repeat(list1))
```

Output:

```
Duplicate:  [20, 30, -20, 60]
```

Example: Multiple Inheritance

```
class GrandFather(object):

        # Constructor
        def __init__(self, name):
                self.name = name

        # To get name
        def getName(self):
                return self.name

# Inherited or Sub class (Note Person in bracket)
class Father(GrandFather):

        # Constructor
        def __init__(self, name, age):
                GrandFather.__init__(self, name)
                self.age = age

        # To get name
        def getAge(self):
                return self.age

# Inherited or Sub class
class Child(Father):

        # Constructor
        def __init__(self, name, age, address):
                Father.__init__(self, name, age)
                self.address = address

        # To get address
        def getAddress(self):
```

```
            return self.address

    # Running Code
    g = Child("Bash", 22, "Basra")
    print(g.getName(), g.getAge(), g.getAddress())
```

Output:

```
    Bash 23 Basra
```

Example: Turtle Race

```
    # Import needed library
    import turtle
    import random

    # Creating green and blue turtles
    green_player = turtle.Turtle()
    green_player.color("green")
    green_player.shape("turtle")
    green_player.penup()
    green_player.goto(-200,100)
    blue_player = green_player.clone()
    blue_player.color("blue")
    blue_player.penup()
    blue_player.goto(-200,-100)

    #Draw final point
    green_player.goto(300,60)
    green_player.pendown()
    green_player.circle(40)
    green_player.penup()
    green_player.goto(-200,100)
    blue_player.goto(300,-140)
    blue_player.pendown()
    blue_player.circle(40)
    blue_player.penup()
    blue_player.goto(-200,-100)
```

```
#Die points
die = [1,2,3,4,5,6]

#Start Playing
for i in range(20):
    if green_player.pos() >= (300,100):
        print("Green Turtle Wins!")
        break
    elif blue_player.pos() >= (300,-100):
        print("Blue Turtle Wins!")
        break
    else:
        green_player_turn = input("Press 'Enter' to roll the die ")
        die_outcome = random.choice(die)
        print("The result of the die roll is: ")
        print(die_outcome)
        print("The number of steps will be: ")
        print(20*die_outcome)
        green_player.fd(20*die_outcome)
        blue_player_turn = input("Press 'Enter' to roll the die ")
        die_outcome = random.choice(die)
        print("The result of the die roll is: ")
        print(die_outcome)
        print("The number of steps will be: ")
        print(20*die_outcome)
        blue_player.fd(20*die_outcome)
```

Output:

Run the code and play it with your friend to see who will win. Good Luck

With a close assessment of the current atmosphere; the current societal demand for industrial skill sets and to make the world a better place, a team of friends have decided to aid by providing this free Python Programming Hand Book for both beginners and experienced programmers.

This book provides a core and solid foundation with hands-on examples that will help beginners to easily, quickly and efficiently gain the industrial needed skill sets. For experienced programmers, it provides an easy grasp of relevance and more demanding Python Programming skills.