


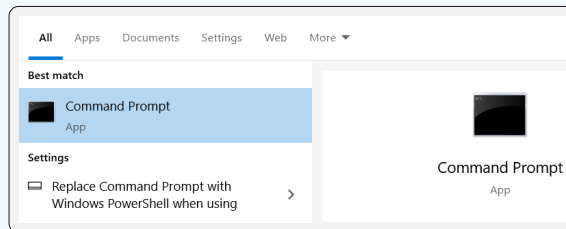
1. Getting Started

Python is available for many platforms including Microsoft Windows. The most up-to-date software can be downloaded from <https://www.python.org> to install on your operating system. (Appendix A explains how to install Python 3 on Windows 10.)

To check whether which Python is installed on your computer, issue the command `python --version`, at the command prompt.



To open a command prompt in Windows 10 or Windows 7, an easy way is to search by pressing the Windows-key . Press once on it on the keyboard and type the text `command`. A window will appear to show the search results:



Clicking on Command prompt will bring the command prompt shell to work on. ♦

Issue the command `python --version` at the Window command shell prompt `C:\Users\Siva>`, it will show the version of Python installed on your computer:

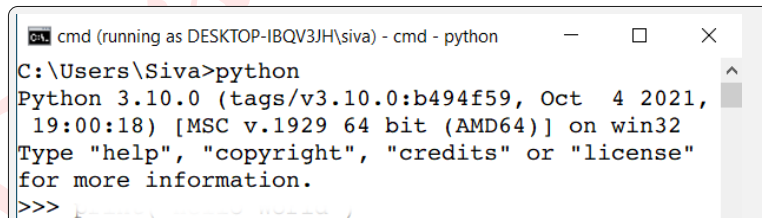
```
C:\Users\Siva>python --version
Python 3.10.0
C:\Users\Siva>
```

Python can be launched and program can be executed in more than one way in Windows. We shall see them one by one.

1.1 Python in Command Prompt

First, we shall see how Python can be used in Command prompt shell of Windows. To launch Python, issue the command: `C:\Users\Siva>python`

You will get something like:



This shows version number of Python, and some other details such as release date of this particular version. It also shows the commands (functions) `help`, `copyright`, *etc.* to type at the Python prompt `>>>` for more information.



The Python prompt, `>>>`, consists of three greater-than symbols followed by a space character. ♦

Let us try the traditional Hello world! at the Python prompt.



```
C:\Users\Siva>python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021,
19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license"
for more information.
>>> print("Hello World")
Hello World
>>>
```

The statement `print("Hello World!")` prints the text Hello World in the next line, below which Python prompt `>>>` appears again. We can continue this way working with Python interactively.

To get out of this Python session, we need to type `exit()` or `quit()` or CTRL-Z followed by pressing the ENTER-key. Linux users press CTRL-D instead of CTRL-Z with no need for pressing the ENTER-key.

Let us see another way of running a Python program.

We create a program using a text-editor like Notepad or Notepad++ with the name `hello.py` which has the following two lines:

```
print("Hello World!")
print("Done!")
```



Programs having one or two statements, like `hello.py`, can easily be created using `echo` command of Windows as follows:
At the command prompt, type as shown below
C:\Users\Siva>echo print("Hello World") > hello.py
C:\Users\Siva>echo print("Done!") >> hello.py
The use of single greater-than symbol, `>`, redirects the output of `echo` command to a new file whereas the use of double greater-than symbols, `>>`, will append the output of `echo` to the existing file. (If the file does not exist, it will create the file and add the output to it.) ♦

The file `hello.py` has a Python program consisting of two lines. To run the program, we *run* the program at the command prompt as follows:

```
C:\Users\Siva>python hello.py
```

The program gets executed and the output will be displayed on the command-prompt shell as follows:

```
C:\Users\Siva>python hello.py
Hello World!
Done!
C:\Users\Siva>
```

In this way, a program or a script in Python can be executed.



In general sense, programs are compiled to get an object code so that it is executed as an application in the operating system. Programs written in C are examples of this kind. On the other hand, scripts are executed by another application such interpreted languages as Bash, Visual Basic or JavaScript. For example, macros in Microsoft Excel are written as VisualBasic Scripts to do tasks in Excel. Bash scripts (sequences of Bash commands) are written to do tasks in a Linux-based System. ♦

Python is a kind of both compiled and interpreted language, like Java. Java has a compiler and an interpreter: `javac.exe` and `java.exe` (or with no `.exe` extension in other operating systems than Microsoft Windows). `javac` compiles the java source-code (the file has extension `.java`) into java bytecode (the file has extension `.class`). `javac` checks for grammatical errors (syntax errors), certain flaws in constructs and for certain kinds of variable oriented mishaps. On successful compilation, java is launched to run the bytecode file. Here, bytecodes are interpreted by java.

In Python, the python interpreter/compiler compiles a program into its bytecode, and it gets interpreted. The bytecode is not saved as a file, when a program is run at the command prompt. However, the bytecode can be saved in a file by using a python module `py_compile`:

```
>>> import py_compile
>>> py_compile.compile('hello.py')
'__pycache__/hello.cpython-310.pyc'
>>>
```

The bytecode of the program `hello.py` in this example is saved in the file named `hello.cpython-310.pyc` in the subfolder `__pycache__` in the current folder where the program is saved.

The bytecode can also be run at Windows Command prompt as `C:\Users\Siva>python '__pycache__/hello.cpython-310.pyc'`
In this case, python directly interprets the bytecode with no need for compilation.

There is yet another way to execute a single statement by Python.

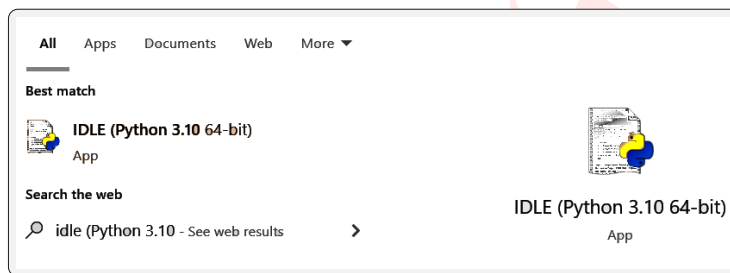
Suppose that we want to compute the value of an arithmetic expression like $(98.4 - 32) * 5 / 9$. This can be accomplished simply by using `-c` flag to the python command as shown below:

```
C:\Users\Siva>python -c "print((98.4 - 32)*5/9)"
36.8888888889
C:\Users\Siva>
```

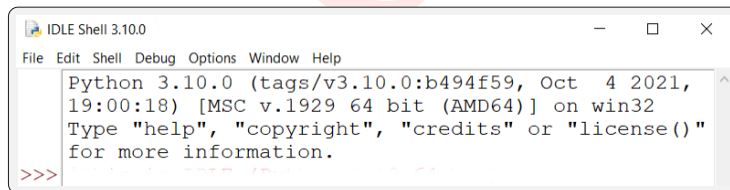
Python is installed with its own Python shell with a graphical user interface called *Integrated Development and Learning Environment* (IDLE), in which too we can interactively issue Python commands and run Python programs. It provides an editor as well as an interactive shell. The program or script written in the editor can be run from menu selection, it will run in the shell.

1.2 Python in IDLE

Python IDLE shell in Windows can be started by running `idle.pyw` which is saved in `Lib\IdleLib` subfolder in the Python folder. Easy way to start IDLE shell is by searching at Star-Menu: Press Windows-key once and type the text IDLE to get the search results as:



Clicking on IDLE (Python 3.10 64-bit) will open Python shell in IDLE that looks like:



This IDLE Python-shell looks like the Python-shell we have seen at the Command prompt shell. This graphical user interface is more convenient to work with. An interactive section can be saved and load for later tries. An editor can be invoked to write the scripts/programs and is used along with the shell seamlessly to run the script/program side by side. The programs can be easily tested while they are developed in a progressive way. The IDLE is simple and easy to use.

We can try the traditional *hello World* at the Python prompt by simply issuing the print function as `print("Welcome to Python3")` in this shell:

```

Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021,
19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()"
for more information.
>>> #This is IDLE (Python 3.10 64-bit)
>>> print("Welcome to IDLE Shell 3.10.0")
Welcome to IDLE Shell 3.10.0
>>>

```

The first line with the Python prompt, `>>>` shows a single-line comment. A text that begins with the hash-sign (#) character and ends with a newline (end-of-line) character is a *comment* in Python.



Multiline comments are enclosed in two triple single-quotes `'''`, or in two triple double-quotes `"""`. A multiline comment may consist of more than one line of text containing many newline characters as part of the comment. Multiline comments are used to provide document strings or doc-strings. ➤

Python will simply ignore the comments, and take the next command or statement to execute. The Python statement `print("Welcome to IDLE Shell 3.10.0")` prints the text (or string) `Welcome to IDLE Shell 3.10.0` as shown above, and the Python prompt, `>>>`, appears in the next line to receive the next command/statement. In this way, we can interactively work with Python in this shell.

Let us try commands and statements to get familiarised with basic features of Python.

1.3 An informal introduction to basic features

This section provides an overview of fundamental features to gain familiarity with them. Students are encouraged to try the examples interactively in an IDLE shell or a Python shell. A detailed coverage of complete features is presented in subsequent chapters.

Arithmetic operators, arithmetic expressions, numeric types like integers, float numbers, and complex numbers, as well as structures like lists, tuples, sets, dictionaries, and strings, assigning names to values/objects, and a few built-in functions are introduced through examples. Python commands or statements are shown at the Python prompt (`>>>`) followed by a comment beginning with a hash sign # (also known as number sign and pound sign), and the result or the output of the statement is given in the next line beneath the command/statement.

1.3.1 Interactive Session

Arithmetic Operators

Arithmetic operators include the usual plus (+), minus (−), multiplication (*), division (/), and modulus (%) operators as in other languages as well as integer division (//) and power or exponentiation (**) operators. These operators are known as binary operators as they take exactly two operands. When more than one operator are involved in an expression, they are applied in the order of precedence prescribed in arithmetic.

Demo 1.1

```
>>> print(1 + 2 * 3) # Operators are applied with usual precedence
7
>>> 1 + 2 * 3 #In the interactive shell the value of an expression is printed in the next line as if it were
printed by the print function.
7
>>> (1 + 2) * 3 # What is enclosed in parentheses has higher precedence
9
>>> 10 - 3 - 2 # Operators of the same precedence are applied left-to-right
5
>>> 10 - (3 - 2) # Enclosing in parentheses makes it right associative
9
>>> 0.21 + 0.37 # Operations on float type numbers may not give the expected result, due to
approximation in floating-point representation. Rounding to 15 decimals would give 0.58
0.5800000000000001
>>> 23 // 4 # Integer division (floor division) operator giving integer result
5
>>> 23 / 4 # Usual division operator giving a real number as the result
5.25
>>> 23.0 // 4 # If one of the operand is of float-type, so is the result though the value is an integer
5.0
>>> 10 // 2.25 # Floor division by a float-type divisor
4.0

>>> 23 % 4 # Modulus (remainder) operator as in C and Java
3
>>> -23 % 4 # Result takes the sign of the second operand
1
>>> 23 % -4 #  $23 = (-6) * (-4) + r$ ;  $-4$  and  $r$  have the same sign
-1
>>> -23 % -4 #  $-23 = (5) * (-4) + r$ ;  $-4$  and  $r$  have the same sign
-3
>>> 2**5 # In Python, double-star operator ** is the power operator as in Fortran. 2**5 means  $2^5$ 
32
>>> 2**213 # In Python 3, integer has no limit;  $2^{213}$  has 65 digits
13164036458569648337239753460458804039861886925068638906788872192
#  $2^{10^6}$  has 301030 digits
>>> 2**3**2 # The power operator, **, is right associative unlike the other operators
512
```

Let us see a few operations on complex numbers which are written in Python as $a + bj$ for literal number b (not an identifier label) or as `complex(a, b)` in general.

Demo 1.2

```
>>> 3 + 4j # Complex number in Python, 3 is real-part and 4 is imaginary-part, where  $1j = \sqrt{-1}$  &
4j =  $4\sqrt{-1}$ 
(3+4j)

>>> (-2)**0.5 # Real-part must be zero, but here a number less than  $1e-16$  is given.
(8.659560562354934e-17+1.4142135623730951j)

>>> (3 + 4j)**2 # same as  $(3 + 4j) * (3 + 4j)$ 
(-7+24j)

>>> (1+2j)/(3+4j) # Complex number division
(0.44+0.08j)

>>> (4+0j)//2 # Floor division is not defined for complex numbers
TypeError: can't take floor of complex number
```

Relational Operators

The relational operators, as in other languages, are ==, !=, <, <=, > and >=.

Demo 1.3

```
>>> 1 + 2 == 4 - 1 # The expression on the right-hand side is equal to that on the left-hand side
True

>>> 0.1 + 0.2 == 0.3 # It says False due to approximation in floating-point representation
False
```

Floating-point errors not only make equality check wrong, but also the inequality check.

Demo 1.4

```
>>> 0.3 < 0.1 + 0.2 # Due to possible error in addition of float numbers,  $0.1 + 0.2$  is greater than
0.3 by  $4 \times 10^{-17}$ 
True

>>> 'Python' != 'python' # True because 'P' is different from 'p'
True

>>> '23' > '142' # True because '23' and '142' are not numbers, but string of numeric characters;
lexically '2' comes after '1'
True

>>> 6 >= 3 # True because  $6 > 3$ 
True

>>> 6 >= 6 # True because  $6 == 6$ 
True

>>> 6 <= 6 # True because  $6 == 6$ 
True
```

Relational operators can be applied not only on numbers and strings but also on objects like list and tuples.

Demo 1.5

```
>>> (1, 3, 2) < (2, 1) # True, the first element of the tuple on the left is less than the first element
of the tuple on the right
True

>>> (1, 3, 2) < (1, 2, 3, 4) # The first elements are the same; the second element on the left
tuple is greater than the second element of the tuple on the right, and thus the less-than relation is false
False
```

Logical Operators

There are three logical operators: not, and, and or. They usually operate on boolean values: True and False. In Python, however, they operate on any objects including boolean values.

Demo 1.6

```
>>> False and True # Both must be True for it to be True
False

>>> False or True # Unless both are False, it is True
>>> False or False # What is it?
>>> True or False and False # Same as True or (False and False) because and has
higher priority than or
True
```

Use of parentheses would change the usual order of evaluation as per the priority order of the operators.

Demo 1.7

```
>>> (True or False) and False # What is enclosed in parentheses is evaluated first
False

>>> not False or True # Same as (not False) or True
True

>>> not (False or True) # What is enclosed in parentheses is evaluated first
False

>>> True or True and False # True and False is evaluated first
True
```

The operator not is a unary operator and has the highest priority when evaluating it to its operand.

Demo 1.8

```
>>> not (1 > 5) # Same as not False
True

>>> not 1 > 5 # Same as (not 1) > 5 same as False > 5 same as 0 > 5
False
```

Python allows more than one relational operators in an expression.

Demo 1.9

```
>>> 1 < 2 < 4 # Same as (1 < 2) and (2 < 4)
True
>>> 1 < 4 > 7 # Same as (1 < 4) and (4 > 7), and thus False
False
```

Logical operators in Python are applicable on any objects.

Demo 1.10

```
>>> not("Python") # Logical operators in Python would operate on any object
False
>>> False or 123 # When the first operand is false, or returns the second operand
123
>>> (2 > 1) and [4, 5, 6] # When the first operand is true, and returns the second operand
[4, 5, 6]
```

Bitwise Operators

There are six bitwise operators `&`, `!`, `^`, `<<`, `>>` and `~`. They are called *bitwise-AND*, *bitwise-OR*, *bitwise-XOR*, *bitwise left-shift*, *bitwise right-shift*, and *bitwise invert* respectively. These operators operate on integers only. Trying on other numbers will throw an error.

Let us try a few examples before see the details of in Chapter 5.

Demo 1.11

```
>>> 10 & 12 # Corresponding bits of binary representations should be considered; 1 & 1 is 1 ; for other
bits, it is 0. 1100 & 1010 = 1000
8
>>> 10 | 12 # 0 | 0 is 0; for other bits, it is 1. 1100 | 1010 = 1110
14
>>> 10 ^ 12 # For the same bits it is 0, and 1 for distinct bits. 1100 ^ 1010 = 0110
6
>>> 123 ^ 123 # The result 0; it is so for any integer
0
>>> 1234 ^ 0 # Exclusive-or of an integer, x, with 0 will be x itself
1234
>>> 123 ^ 456 ^ 678 # This is equivalent to (123 ^ 456) ^ 678 and 123 ^ (456 ^ 678)
789
>>> 123 ^ 456 ^ 456 # If two operands are the same, the result will be the third one
123
>>> 123 ^ 456 ^ 123 # If two operands are the same, the result will be the third one
456
>>> 12 ^ 12 ^ 345 # If two operands are the same, the result will be the third one
345
```

```
>>> ~123 # A unary operator, ~, defined as  $\sim x = -(x + 1)$ 
-124
>>> ~0 # Remember  $0 + \sim 0 = -1$ 
-1
```

Shift operators apply on integers with positive shift-counts only.

Demo 1.12

```
>>> 4<<3 # The binary representation of 4 shifted leftward 3 times padding with zeros on the right.
Equivalent to  $4 \times 2^3$ 
32
>>> 127>>3 # Imagine the binary representation of 123 shifted rightward 3 times padding the left with
zeros. Equivalent to  $123 \div 2^3$ 
15
>>> 123<<-1 # Negative shift-count raises a error named ValueError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: negative shift count
```

Assignment Delimiter

In Python, assignment statement takes different forms in addition to the usual simple form: `target = expression`, where `target` on the left side of the assignment delimiter (`=`) is a name or label given to the expression on the right. In Python, every expression is an *object* as detailed in Chapter,19, and the label refers to that object. Let us call this name or label a *variable* as in other languages. Traditional the symbol, "`=`", is called an assignment operator.

Demo 1.13

```
>>> x1 = 123 # Let us bind a name x1 to integer 123 so that x1 can be used to refer to 123 . We shall
also call such names or labels like x1 as 'variables'.
>>> x1 # The name will return the value it refers to.
123
```

Now the name or variable `x1` can be used in other expressions to refer to the value 123.

Demo 1.14

```
>>> x1 * 2 + 1 # This expression is evaluated using the value 123 referred to by x1.
247
>>> X1, X2 = 'Alice', "Bob" # X1 is bound to the string Alice, and X2 to Bob.
>>> print(x1, X1) # Small x1 and capital X1 are different names because Python is case sensitive.
123 'Alice'
>>> x1, X2 # It forms a tuple of two elements 123, and Bob.
(123, 'Bob')
>>> x1 = 2/7 # x1 now refers to the float type number 2/7 - Data type of x1 is now changed to float
from int. This feature makes Python a dynamically typed language.
>>> x1 # Returns the evaluated value correct to 16 decimals.
0.2857142857142857
```

```
>>> x1 = x1 * 7 # Right-hand side is evaluated by multiplying the value referred to by x1 by 7, and the
result is assigned to x1 again
>>> x1 # x1 now refers to a new value 2.0.
2.0
```

Assignment delimiter, =, can be augmented by prefixing with another delimiter giving an enhanced semantic.

Demo 1.15

```
>>> x1 *= 7 # A shorten form for x1 = x1 * 7.
>>> x1 # Similar shorten forms exist for other arithmetic operators as well.
14.0

>>> a = 100; a -= 60; print('a =', a) # More than one statement can be in the same line,
separated by semicolons; a is set to 100 - 60.
a = 40

>>> x = 10; x ^= 12; print('x =', x) # x is set to 10 ^ 12. Similar assignments are possible
for all bitwise-operators except for not-operator (~).
x = 6
```

In Python, multiple assignments are possible by means of a single statement.

Demo 1.16

```
>>> fname, sname, age = "Rajie", "Rajah", 43 # It assigns values in the given order.
>>> print(fname, age) # Prints the values referred to by fname and age in that order.
Rajie 43

>>> ls = [1, 2, 3, 4, 5] # ls is a list of five integers.
>>> first, *middle, last = ls # first and last take values 1 and 5 from ls and middle
takes the rest of ls, as the asterisk (*) does the magic.
>>> middle # middle (without asterisk) is now a list of three integers.
[2, 3, 4]

>>> ls[0] = "one" # ls[0] refers to the first element of ls, now is set to string "one".
>>> ls # The first element of ls is changed. Python lists are 'mutable' - means that such item assignment
is possible.
['one', 2, 3, 4, 5]
```

Assigning a string value to a variable is straight forward with the assignment symbol '='.

Demo 1.17

```
>>> my_lang = 'Python2.7.18' # Python string can be enclosed in double quotes or single quotes.
>>> my_lang[6] # It gives the seventh character, "2", of my_lang, as indexing begins with zero.
'2'

>>> my_lang = 'Python' "3.12" # A new value is assigned to my_lang. Now my_lang refers
to a new string object. Note that string literals can be written consecutively separated by spaces, and they will be
concatenated into a single string.
```

Item assignment (in-place assignment) is not allowed in strings, unlike in lists. Attempt to do so will cause TypeError.

Demo 1.18

```
>>> my_lang[6] = '3' # This causes an error as item assignment (in-place assignment) in a string is
not allowed. Strings are 'immutable' in Python
Tracebackback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Built-in functions and types

Let us explore a few built-in functions and a few objects of different types (or classes). The functions `print()`, `input()`, `type()`, `int()`, `bin()`, `range()`, `str()`, `list()` and `len()` are examples of built-in functions. (The pairs of parentheses are suffixed with the function names just to indicate they are functions rather than indicating as function calls.) The data types `int`, `float`, `str`, `list`, `tuple` and `dict` are examples of data types (also known as *classes* in Python) that we would come across often.

Demo 1.19

```
>>> print("Welcome to Python") # Prints its arguments on the standard output device.
Welcome to Python

>>> print(3**2 - 4*2) # Evaluates its argument and print.
1

>>> username = input("Type in your name: ") # Waits for a keyboard input from the user,
takes it as a string and assigns to username (and suppose you would type in Siva).
>>> print(username) # Prints the string Siva keyed in by you.
Siva

>>> age = input("Type in your age: ") # Waits for a keyboard input, takes it as a string and
assigns to age.
>>> type(age) # Gives the type of object referred to by age.
<class 'str'>

>>> age = int(age) # Need to change the string referred to by age as integer.
>>> type(age) # Now age is of type int.
<class 'int'>

>>> bin(25) # Gives the binary representation of integer 25, as a string.
'0b11001'

>>> hex(25) # Gives the hexadecimal representation of integer 25, as a string.
'0x19'

>>> oct(25) # Gives the octal representation of integer 25, as a string.
'0o31'
```

The `int()` function can be considered as a conversion function. It converts a numeric string to an integer, and it converts a float number to integer by removing the fractional part. Also, `int()` with no parameter produces zero. Similarly, `float()` function can also be thought of as a conversion: converting a string form of a float number or an integer to a float type number. With no parameter, it outputs zero as a float number, `0.0`.

Demo 1.20

```

>>> int('0b11001', 2) # It gives integer 25 by converting the binary string '0b11001'.
25
>>> int('0x19', 16) # It gives integer 25 by converting the hexadecimal string '0x19'.
>>> int('0o31', 8) # It gives integer 25 by converting the octal string '0o31'.
>>> int(12.9) # It converts the float number to integer by removing fractional part towards zero
12
>>> int(12.2) # It converts to 12
12
>>> int(-12.9) # It converts to -12
-12
>>> int() # If no argument is passed, it returns zero
0
>>> str(12) < str(113) # Integers are converted to strings and compared. Why False?
False
>>> int('12') < int('113') # Strings of numeric characters are converted to integers and
compared
True

```

The `range()` function is useful in various ways in Python. The functions `list()`, `tuple()`, and `set()` can also be thought of as conversion functions. The functions create objects of the respective types of the same names.

Demo 1.21

```

>>> range(5) # It returns a range object that will provide the sequence of integers from 0 to 4 one by one.
range(0, 5)
>>> list(_) # It creates a list of those integers provided by the range object resulted in by range(5);
The underscore,_, stands for the last result in the shell.
[0, 1, 2, 3, 4]
>>> r1 = range(4, 16, 3) # It assigns to r1 a range object that provides the sequence of integers
from 4 to 13 with the increment of 3.
>>> tuple(r1) # It creates a tuple from the range object r1.
(4, 7, 10, 13)
>>> list(range(10, 0, -2)) # It creates a list of integers provided by range(10, 0, -2)
[10, 8, 6, 4, 2]
>>> set(range(10, 0, -2)) # It creates a set of integers provided by range(10, 0, -2)
{10, 4, 8, 2, 6}
>>> list(range(10, 1)) # range(10, 1) can provide no integers since start > stop, but
step = 1 > 0
[]
>>> list(range(1, 10, -1)) # range(1, 10, -1) null object since start < stop, but
step < 0
[]

```

```
>>> len(range(3, 100, 13)) # It gives the number of integers in the sequence that range(3,
100, 13) will give
8
```

The functions `list()`, `tuple()` and `set()` can be used to create objects by converting other items.

Demo 1.22

```
>>> a_str = "Mississippi" # The string "Mississippi" is assigned to a_str.
>>> list(a_str) # It gives the list of all characters in the string in the order present, enclosed in square
brackets.
['M', 'i', 's', 's', 'i', 's', 's', 'i', 'p', 'p', 'i']
>>> set(a_str) # It gives a set of characters in "Mississippi"; four distinct elements enclosed in
curly braces in a random order.
{'i', 'p', 's', 'M'}
>>> tuple("Python") # It gives the tuple of characters in the string in that order, enclosed in parentheses.
('P', 'y', 't', 'h', 'o', 'n')
>>> list(_) # The last output tuple is converted as a list.
['P', 'y', 't', 'h', 'o', 'n']
>>> list() # With no argument, it creates an empty list.
[]
>>> tuple() # With no argument, it creates an empty tuple.
()
>>> a_str = "Mississippi"
>>> len(a_str) # It gives the number of characters in the string "Mississippi".
11
>>> mst = set(a_str) # It creates a set of characters from a_str, and assigns to mst.
>>> len(mst) # It gives the number of elements in the set mst.
4
>>> s1=str(576); type(s1) # It converts the integer 576 to a string '576'.
<class 'str'>
```

Dictionaries can be constructed in more than one way using `dict()` function:

Demo 1.23

```
>>> {"A": 1, "B": 2, "C": 3, "D": 4} # This is a literally written dictionary that maps
"A", "B", "C" and "D" respectively to 1, 2, 3, and 4.
>>> dc1 = dict(A=1, B=2, C=3, D=4) # dict() function creates the same dictionary created
above. Keys must be of immutable types such as number, string, tuple, but cannot be a list
>>> dc1 # key : value pairs are separated by commas and enclosed in curly braces.
{'A': 1, 'B': 2, 'C': 3, 'D': 4}
>>> dict([("A", 1), ("B", 2), ("C", 3), ("D", 4)]) # This returns the same
dictionary as above.
>>> z1 = zip((1, 2, 3, 4, 5), "ABCD") # zip() function makes tuples taking elements one
by one from each of its arguments until when no more element exists in any of the arguments. zip object returned
by this zip function will produce tuples of pairs (1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'). Since there
```

```
is no element after D in the second argument ABCD, the element 5 in the first argument is ignored.
>>> dc2 = dict(z1) # dict() function can take a single zip object and produce a dictionary
>>> dc2 # This dictionary maps 1 to "A", 2 to "B", 3 to "C" and 4 to "D".
{1: 'A', 2: 'B', 3: 'C', 4: 'D'}

>>> dict(zip((5, 15, 25), (25, 225, 625))) # This dictionary maps 5 to 25, 15 to 225,
and 25 to 625
{5: 25, 15: 225, 25: 625}
```

1.4 Summary

In this chapter, a quick guidance is given on how to install Python 3 on Windows operating systems assuming it helps majority of students as most of them use Personal computers or Laptops with Windows. Students are also guided how to set environment variable PATH in Windows, in case they need to do so.

Moreover, two different ways of running a Python script or program are introduced with a traditional hello world program: to run on terminal command prompt, and to run using integrated development and learning environment (IDLE), which is simple environment but good enough to interact and learn.

The IDLE is used to given an informal introduction to basic features of Python 3: core data types, operators on them and a few functions built-in to Python so as to encourage the students to get familiarise with Python by interactively learn with the help of easy-to-use IDLE.

Arithmetic operators, relational operators, logical operators, bitwise operators, assignment operators, and selected built-in functions and types are also introduced as interactive demo sessions.

Students are urged to try these demo sessions themselves. This would make you ready to explore further into the language, and to use it to solve problems as it would be the ultimate aim of learning a programming language.