Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

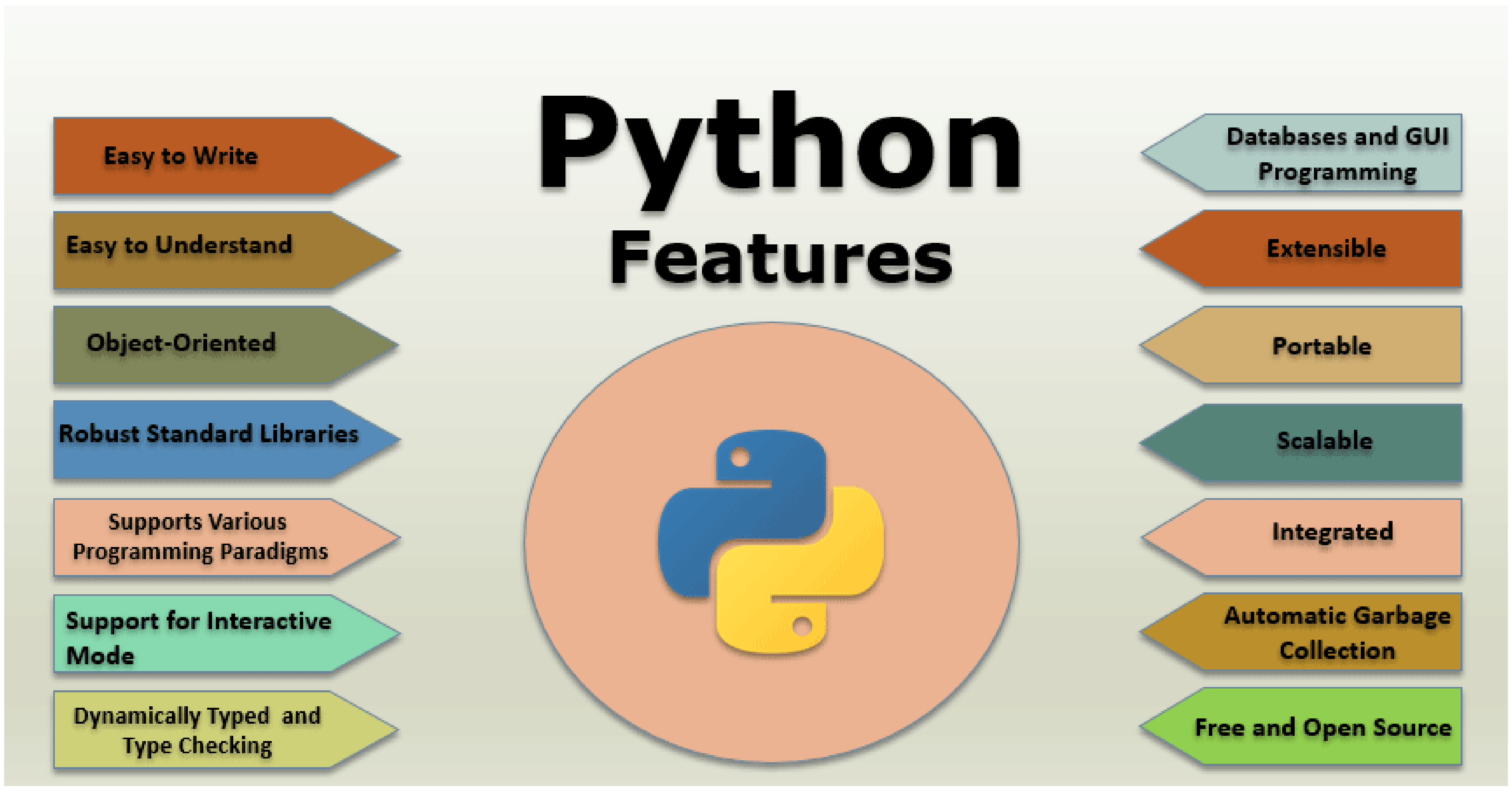
- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be **used on a server to create web applications.**
- Python can be used **alongside software to create workflows.**
- Python can connect to database systems. **It can also read and modify files.**
- Python can be used to handle **big data and perform complex mathematics.**

Why Python?

- Python **works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).**
- Python has a **simple syntax similar to the English language.**
- Python has syntax that allows developers to **write programs with fewer lines than some other programming languages.**
- Python runs on an interpreter system, **meaning that code can be executed as soon as it is written.** This means that prototyping can be very quick.
- Python can be treated in a **procedural way, an object-oriented way or a functional way.**


Python
Features
Easy to Write
Easy to Understand
Object-Oriented
Robust Standard Libraries
Supports Various Programming Paradigms
Support for Interactive Mode
Dynamically Typed and Type Checking
Databases and GUI Programming
Extensible
Portable
Scalable
Integrated
Automatic Garbage Collection
Free and Open Source

1. Easy to Write

These days with the increasing number of libraries in the languages, most of the developer's time goes in remembering them. This is one of the great features of python as python libraries use simple English phrases as its keywords. Thus it’s very easy to write code in python.

2. Easy to Understand

This is the most powerful feature of the python language, which makes it everyone’s choice. As the keywords used here are simple English phrases; thus, it is very easy to understand.

3. Object-Oriented

Python has all features of an object-oriented language **such as inheritance**, method overriding, objects, etc. Thus it supports all the paradigms and has corresponding functions in their libraries. It also supports the implementation of multiple inheritances, unlike java.

4. Robust Standard Libraries

The libraries of python are very vast that include various modules and functions that support various operations working in various data types **such as regular expressions** etc.

5. Supports Various Programming Paradigms

With support for all the features of an object-oriented language, Python also supports the procedure-oriented paradigm. It supports **multiple inheritances** as well. This is all possible due to its large and robust libraries that contain functions for everything.

6. Support for Interactive Mode

Python also has support for working in interactive mode where one can easily debug the code and unit test it line by line. This helps to reduce errors as much as possible.

7. Automatic Garbage Collection

Python also initiates automatic garbage collection for great memory and performance management. Due to this, memory can be utilised to its maximum, thus making the application more robust.

8. Dynamically Typed and Type Checking

This is one of the great features of python that one need not declare the data type of a variable before using it. Once the value is assigned to a variable, its datatype gets defined. Thus, type checking in python is done at a run time, unlike other programming languages.

For e.g.-

```
v=7;# here type or variable v is treated as an integer
```

```
v="great";#here type of the variable v is treated as a string<
```

9. Databases

The database of an application is one of the crucial parts that also need to be supported by the corresponding programming language being used. Python supports all the major databases that can be used in an application, such as MYSQL, ORACLE, etc. Corresponding functions for their database operations have already been defined in python libraries. one needs to include those files in code to use it.

10. GUI Programming

Python **being a scripting language** also supports many features and libraries that allow graphical development of the applications. In the vast libraries and functions, corresponding system calls and procedures are defined to call particular OS calls to develop an application's perfect GUI. Python also needs a framework to be used to create such a GUI. Examples of some of the frameworks are Django, Tkinter, etc.

11. Extensible

This feature makes use of other languages in python code possible. This means python code can be extended to other languages as well; thus, it can easily be embedded in existing code to make it more robust and enhance its features. Other languages can be used to compile our python code.

12. Portable

A programming language is portable if it allows us to code once and runs anywhere. This means the platform where it has been coded and where it is going to run need not be the same. This feature allows one of the most valuable features of object-oriented languages-reusability. As a developer, one needs to code the solution and generate its byte code and need not worry about the environment where it will run. EO-one can run code developed on the Windows operating system on any other operating system Linux, Unix, etc.

13. Scalable

This Language helps develop various systems or applications capable of handling a dynamically increasing amount of work. These types of applications help a lot in the organisation's growth as they are strong enough to handle the changes up to some extent.

14. Free and Open Source

Yes, u read it correctly u need not pay a single penny to use this language in your application. One needs just to download it from its official website, and it's all done to start. And as it is open-source, its source code has also been made public. One can easily download it and use it as required as well as share it with others. Thus it gets improved every day.

15. Integrated

Python can be easily integrated with other available programming languages such as C, C++, Java, etc. This allows everyone to use it to enhance the functionality of existing applications and make them more robust.

Difference between C++ / Java / Python

C++	JAVA	PYTHON

Compiled Programming language	Compiled Programming Language	Interpreted Programming Language
Supports Operator overloading	Does not support Operator Overloading	Supports Operator overloading
Provide both single and multiple inheritance	Provide partial multiple inheritance using interfaces	Provide both single and multiple inheritance
Platform dependent	Platform Independent	Platform Independent
Does Not support threads	Has in build multithreading support	Supports multithreading
Has limited number of library support	Has library support for many concepts like UI	Has a huge set of libraries that make it fit for AI, datascience, etc.
Code length is a bit lesser, 1.5 times less than java.	Java has quite huge code.	Smaller code length, 3-4 times less than java.
Functions and variables are used outside the class	Every bit of code is inside a class.	Functions and variables can be declared and used outside the class also.
C++ program is a fast compiling programming language.	Java Program compiler a bit slower than C++	Due to the use of interpreters, execution is slower.
Strictly uses syntax norms	Strictly uses syntax norms	Use of ; is not compulsory.
like ; and {}.	like punctuations , ; .	

Flavours of Python

1. CPython

It is the standard flavour of Python. It can be used to work with C language Applications.

2. Jython OR JPython

It is for Java Applications. It can run on JVM.

3. IronPython

It is for C#.Net platform

4. PyPy

The main advantage of PyPy is that performance will be improved because the JIT compiler is available inside PVM(Python Virtual Machine).

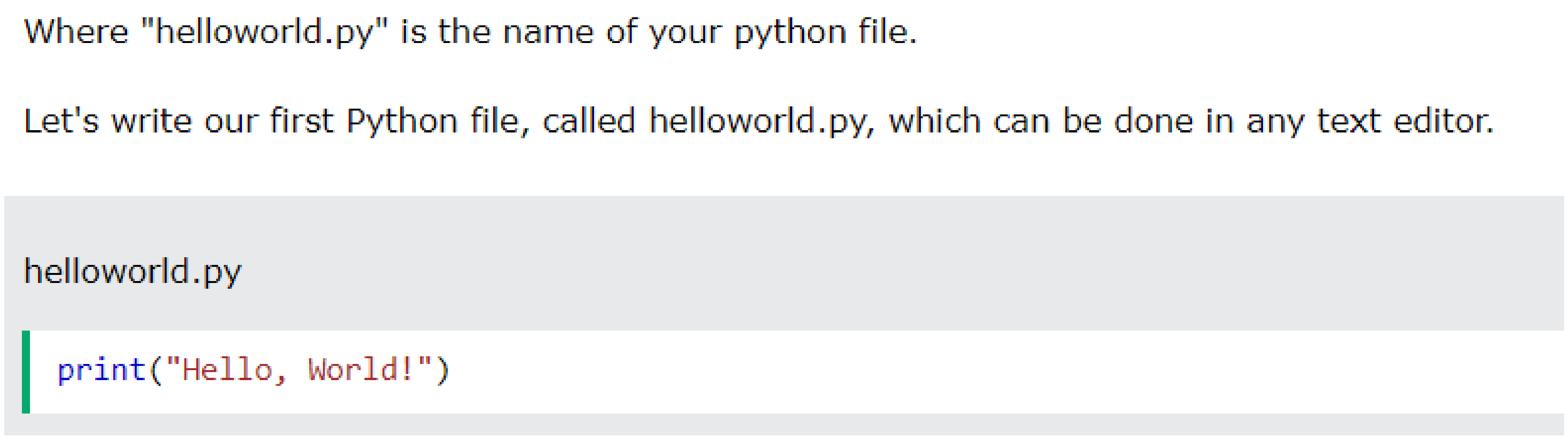
5. RubyPython

For Ruby Platforms

6. AnacondaPython

It is specially designed for handling large volumes of data processing.

My First Python Program



Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

helloworld.py

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

Python Virtual Machine (PVM)

We know that computers understand only machine code that **comprises 1s and 0s.**

Since a computer understands only machine code, **it is imperative that we should convert any program into machine code** before it is submitted to the computer for execution.

For this purpose, we should take the help of a compiler. **A compiler normally converts the program source code into machine code.**

A Python compiler does the same task but in a slightly different manner.

It converts the program source code into another code, called bytecode.

Each Python program statement is converted into a group of byte code instructions.

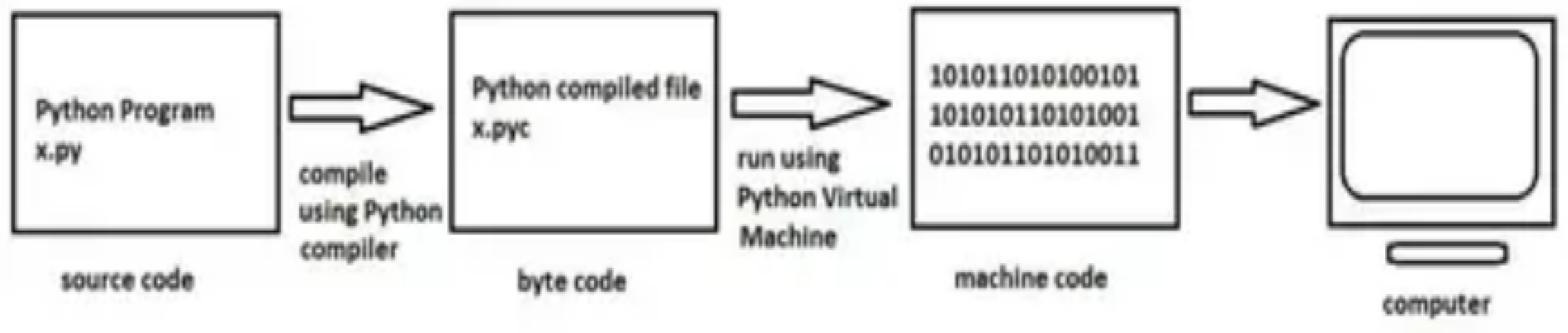
Byte code represents the fixed set of instructions created by Python developers representing all types of operations.

The size of each bytecode instruction is 1 byte (or 8 bits) and hence these are called byte code instructions.

The Python organisation says that there may be newer instructions added to the existing byte code instructions from time to time.

We can find byte code instructions in the .pyc file.

Following Figure shows the role of virtual machine in converting byte code instructions into machine code:



The **role of Python Virtual Machine (PVM) is to convert the byte code instructions into machine code** so that the computer can execute those machine code instructions and display the final output.

To carry out this conversion, **PVM is equipped with an interpreter.**

The interpreter converts the byte code into machine code and **sends that machine code to the computer processor for execution.**

Since interpreters are playing the main role, **often the Python Virtual Machine is also called an interpreter.**

Garbage Collection in Python

Python's memory **allocation and deallocation method is automatic.**

The user does not have to preallocate or deallocate memory similar to using dynamic memory allocation in languages such as C or C++.

Python uses two strategies for memory allocation:

1. Reference counting
2. Garbage collection

Reference counting

Python and various other programming languages employ reference counting, a memory management approach, to **automatically manage memory by tracking how many times an object is referenced.**

A reference count, or the number of references that point to an object, **is a property of each object in the Python language.**

When an object's reference count reaches zero, it becomes un-referenceable and its memory can be freed up.

Example 1: Simple Reference Counting

```python
# Create an object
x = [1, 2, 3]

# Increment reference count
y = x

# Decrement reference count
y = None
```

Example 2: Reference Counting with Cyclic Reference

```python
# Create two objects that refer to each other
x = [1, 2, 3]
y = [4, 5, 6]
x.append(y)
y.append(x)
```

Garbage collection

Garbage collection is a memory management technique used in programming languages to **automatically reclaim memory that is no longer accessible or in use by the application.**

It helps prevent memory leaks, optimise memory usage, and ensure efficient memory allocation for the program.

```python
# loading gc
import gc

# get the current collection
# thresholds as a tuple
print("Garbage collection thresholds:",
                    gc.get_threshold())
```

Output:

```
Garbage collection thresholds: (700, 10, 10)
```

Here, the default threshold on the above system is 700.

This means **when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.**

Thus any portion of your code **which frees up large blocks of memory is a good candidate for running manual garbage collection.**

Some Examples of Python Programs :

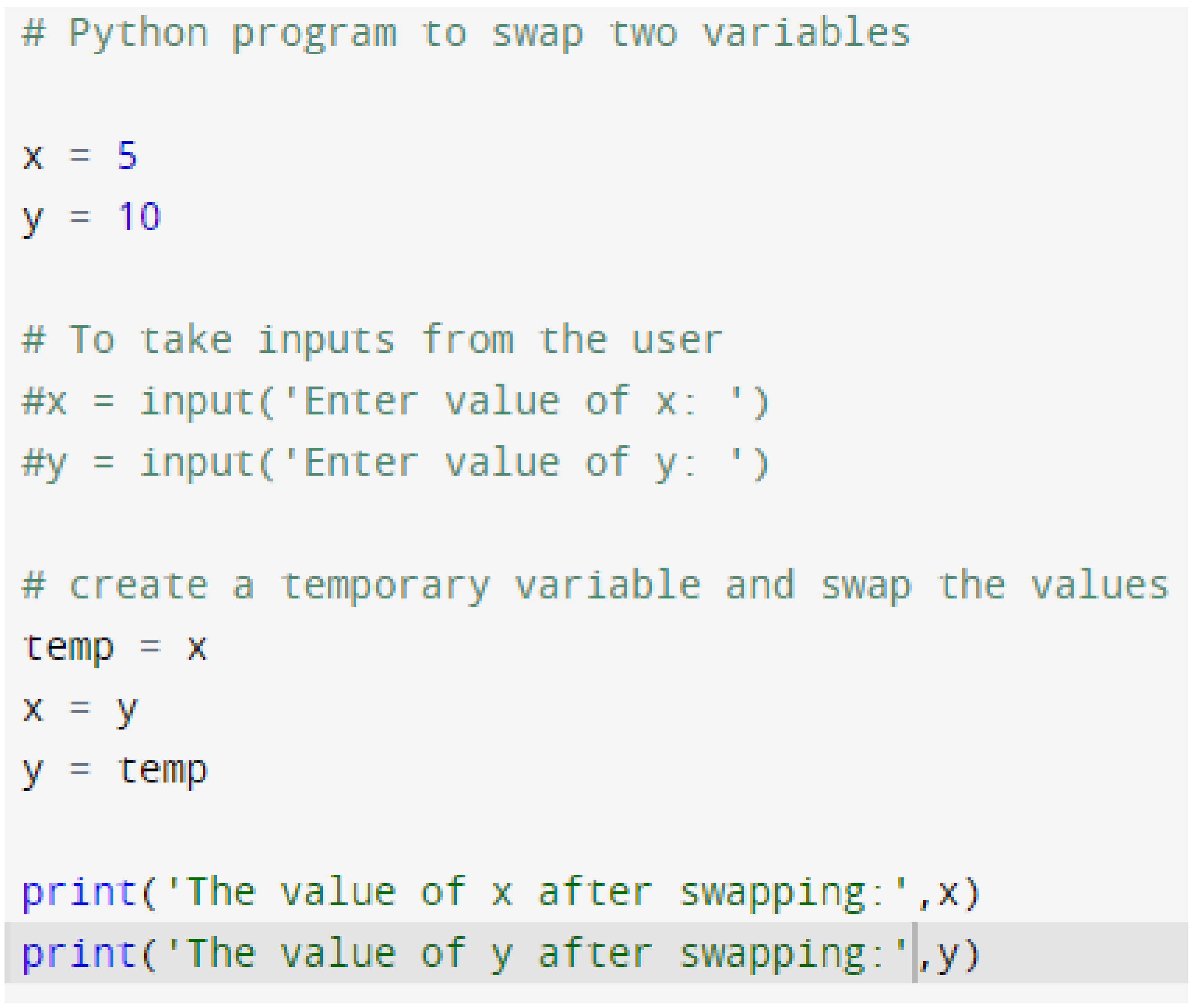
Example 1: Add Two Numbers

```
# This program adds two numbers

num1 = 1.5
num2 = 6.3

# Add two numbers
sum = num1 + num2

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Example 2: Add Two Numbers With User Input

```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

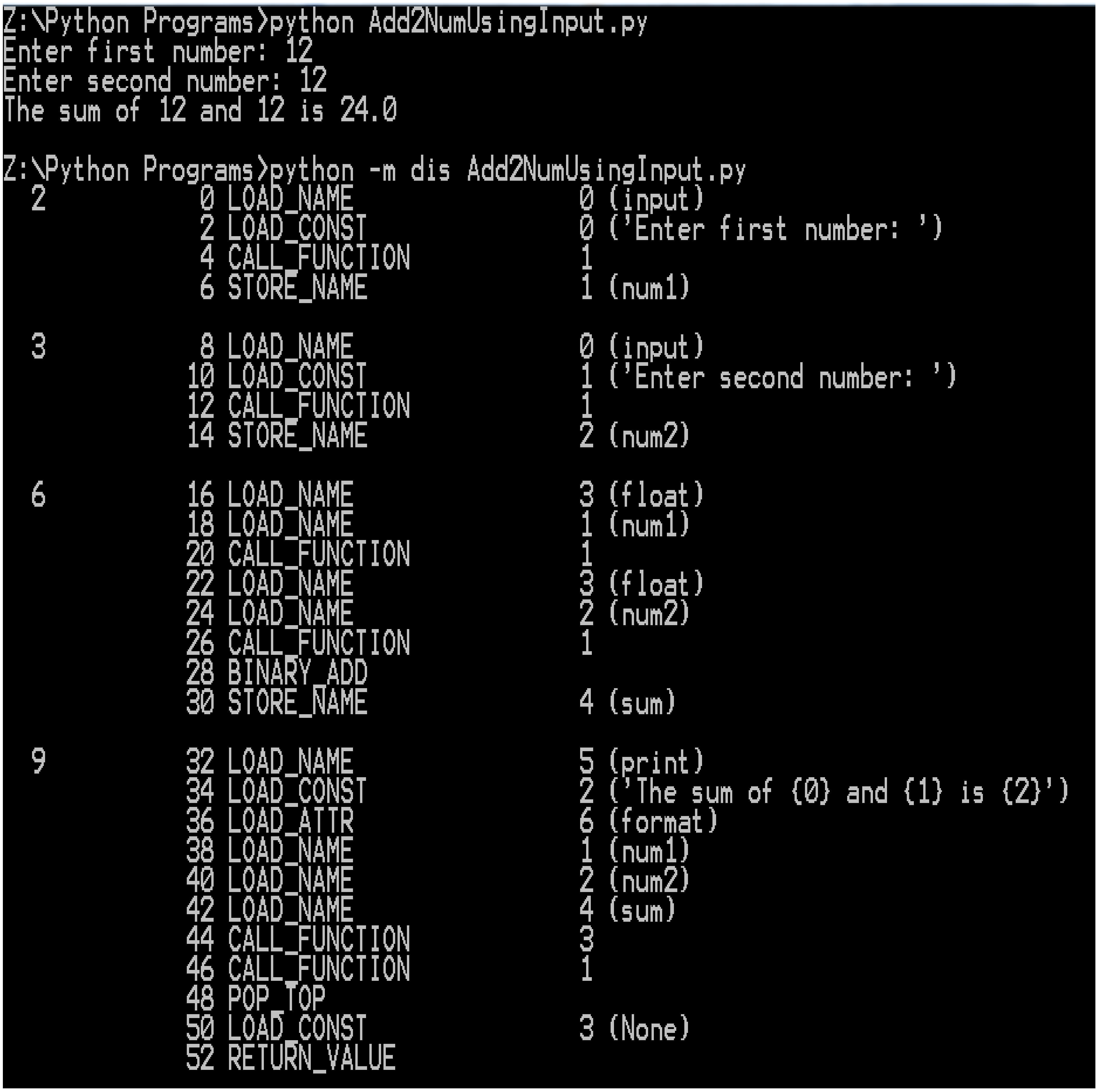
Example 3: Swap 2 Numbers Using Temporary Variable



```python
# Python program to swap two variables

x = 5
y = 10

# To take inputs from the user
#x = input('Enter value of x: ')
#y = input('Enter value of y: ')

# create a temporary variable and swap the values
temp = x
x = y
y = temp

print('The value of x after swapping:',x)
print('The value of y after swapping:',y)
```

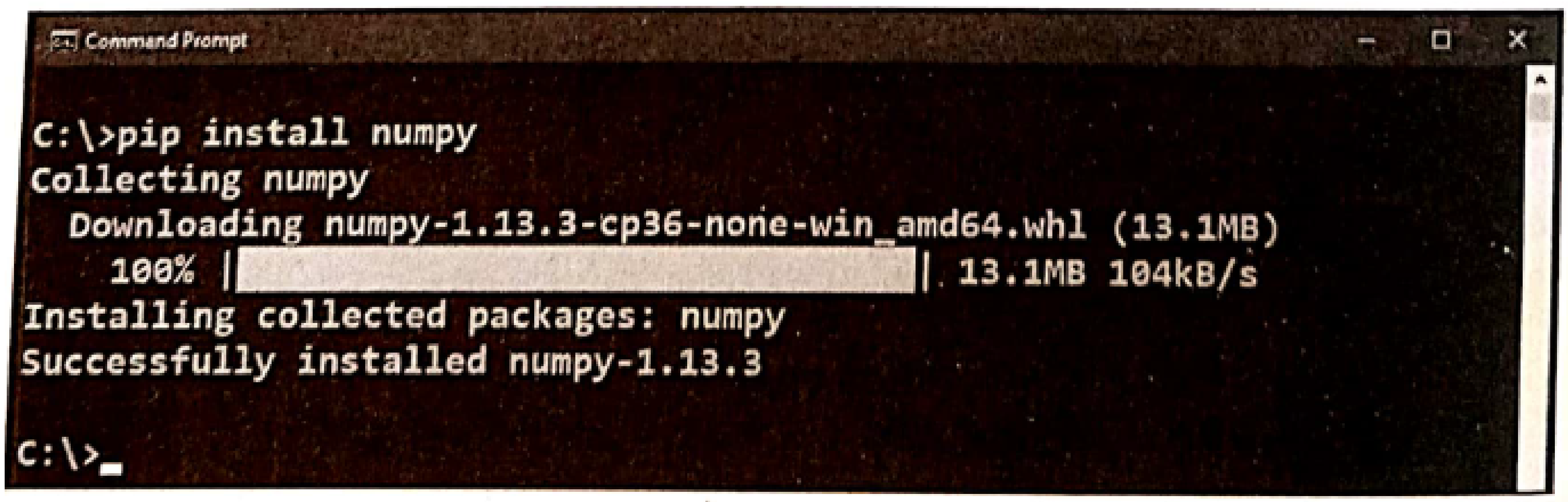
Creating, Executing and Viewing Byte Code of Python Programs

Viewing of Byte Code



```
Z:\Python Programs>python Add2NumUsingInput.py
Enter first number: 12
Enter second number: 12
The sum of 12 and 12 is 24.0

Z:\Python Programs>python -m dis Add2NumUsingInput.py
  2           0 LOAD_NAME                0 (input)
              2 LOAD_CONST               0 ('Enter first number: ')
              4 CALL_FUNCTION            1
              6 STORE_NAME               1 (num1)

  3           8 LOAD_NAME                0 (input)
             10 LOAD_CONST               1 ('Enter second number: ')
             12 CALL_FUNCTION            1
             14 STORE_NAME               2 (num2)

  6          16 LOAD_NAME                3 (float)
             18 LOAD_NAME                1 (num1)
             20 CALL_FUNCTION            1
             22 LOAD_NAME                3 (float)
             24 LOAD_NAME                2 (num2)
             26 CALL_FUNCTION            1
             28 BINARY_ADD
             30 STORE_NAME               4 (sum)

  9          32 LOAD_NAME                5 (print)
             34 LOAD_CONST               2 ('The sum of {0} and {1} is {2}')
             36 LOAD_ATTR                6 (format)
             38 LOAD_NAME                1 (num1)
             40 LOAD_NAME                2 (num2)
             42 LOAD_NAME                4 (sum)
             44 CALL_FUNCTION            3
             46 CALL_FUNCTION            1
             48 POP_TOP
             50 LOAD_CONST               3 (None)
             52 RETURN_VALUE
```

Installing Some Basic Packages



```
C:\>pip install numpy
Collecting numpy
  Downloading numpy-1.13.3-cp36-none-win_amd64.whl (13.1MB)
    100% |████████████████████████████████| 13.1MB 104kB/s
Installing collected packages: numpy
Successfully installed numpy-1.13.3

C:\>_
```

Figure 2.15: Installation of numpy package

- Similarly for Database Programs you need to install
- pip install mysql-connector-python

Verifying Installed Packages

We can verify whether the installed packages are added to our Python software properly or not by going into Python and typing the following command at Python prompt (triple greater than symbol) as:

```
>>> help('modules')
```

For this purpose, first click the Python IDLE Window pinned at the taskbar and then type the preceding command as shown in Figure 2.19:

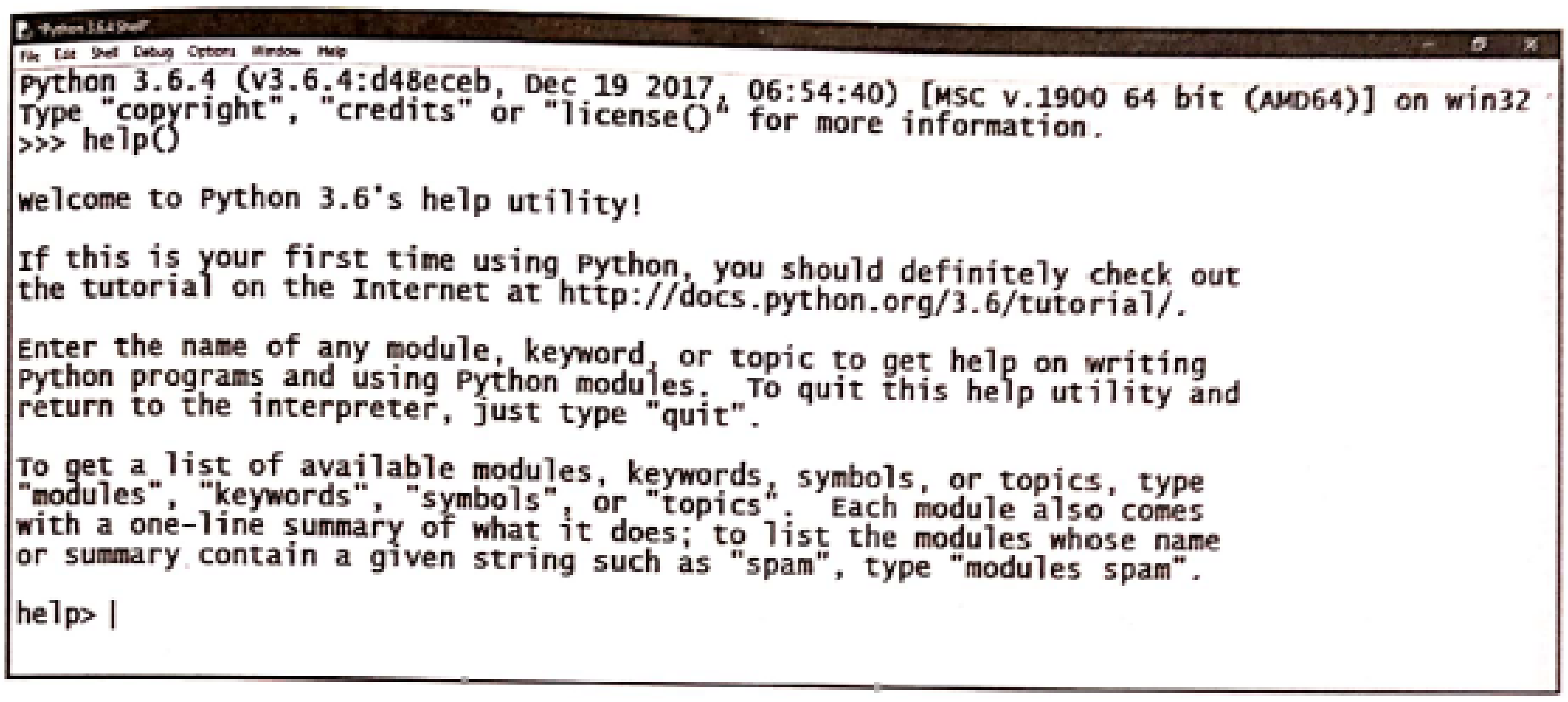
Execution of Python Program

Executing a Python Program

There are three ways of executing a Python program.

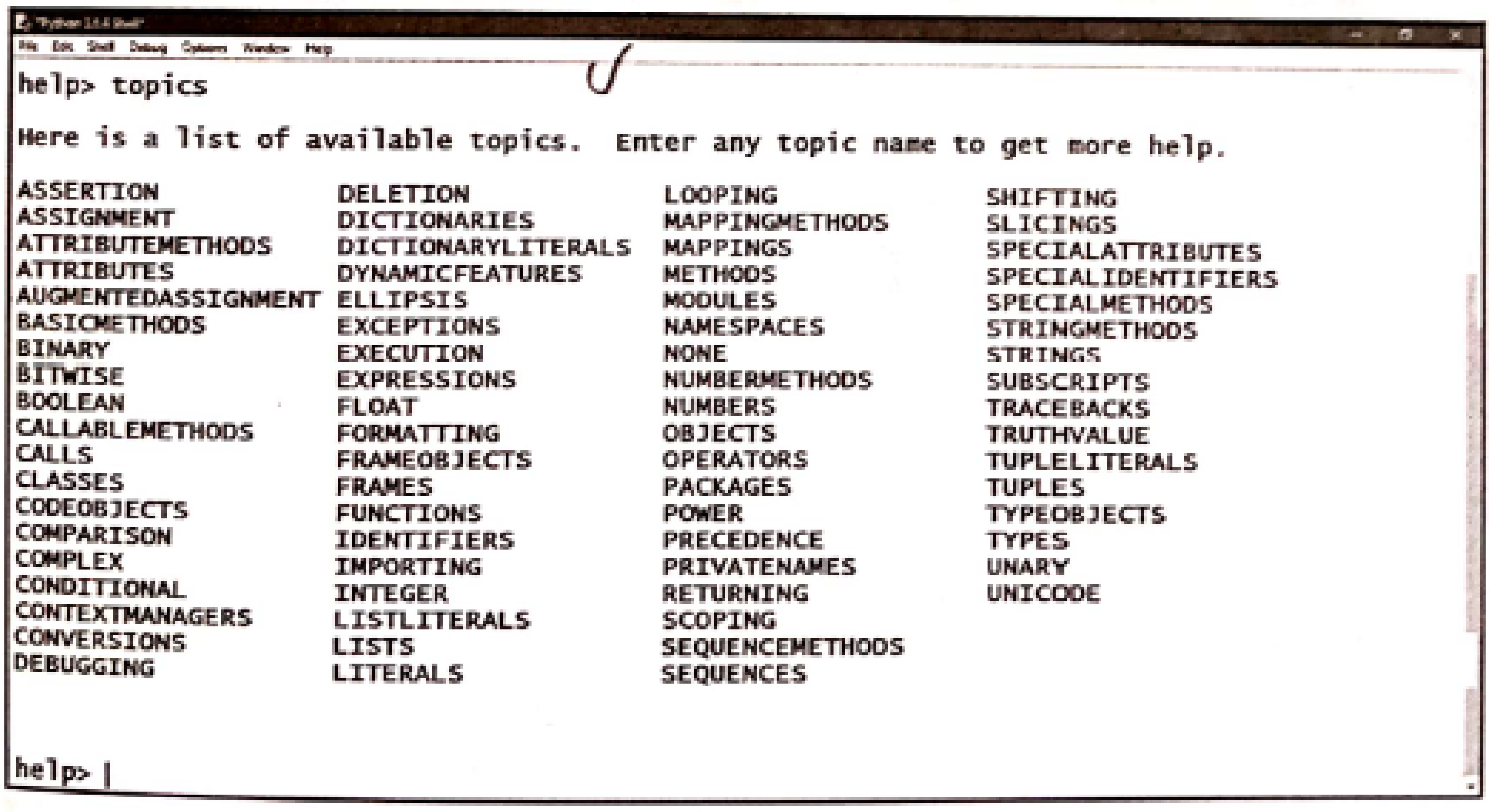
- Using Python's command line window
- Using Python's IDLE graphics window
- Directly from System prompt

Getting Help in Python



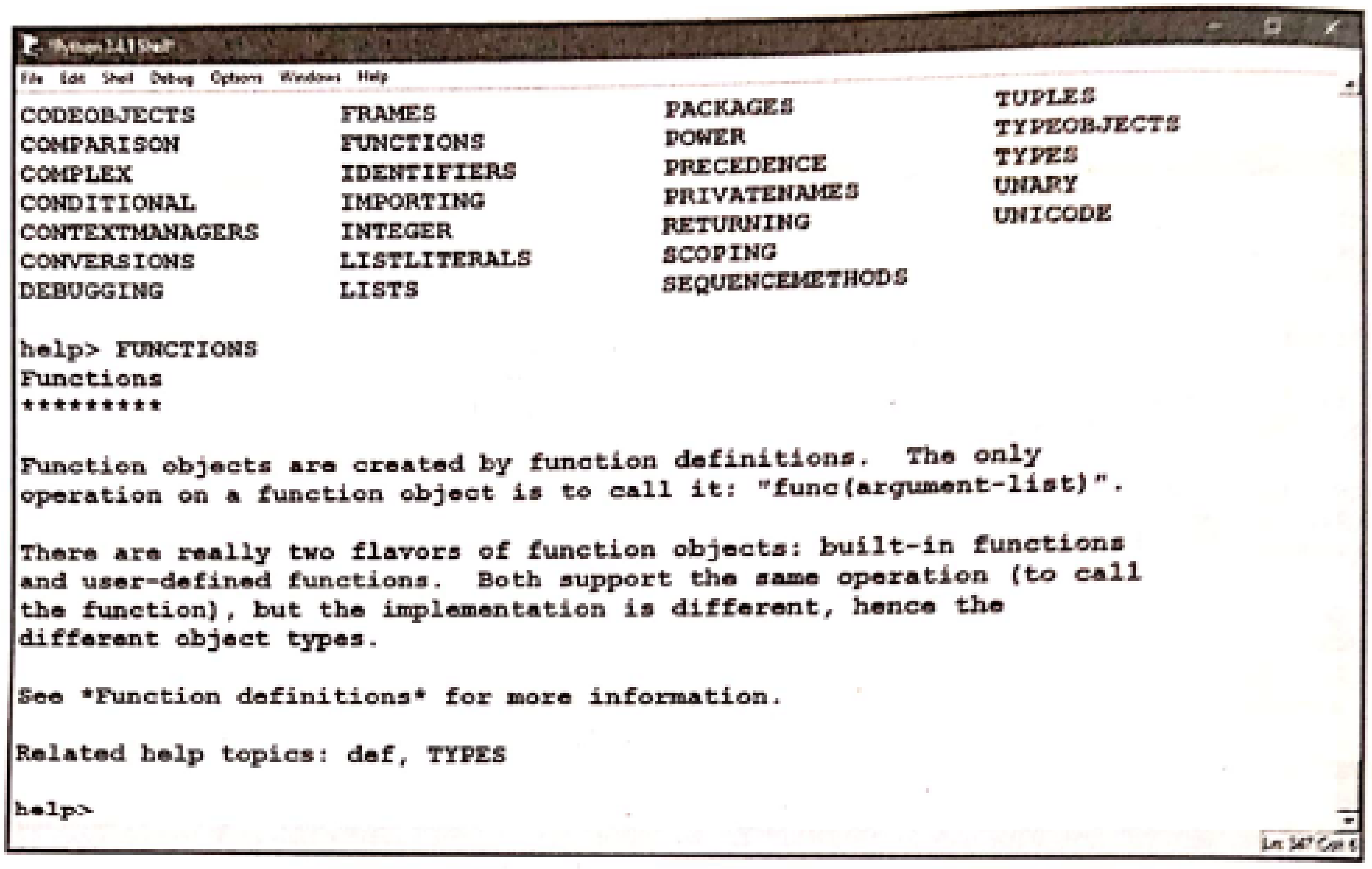
```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> |
```

Figure 2.32: Entering Help Mode in Python



```
help> topics

Here is a list of available topics.  Enter any topic name to get more help.

ASSERTION           DELETION            LOOPING             SHIFTING
ASSIGNMENT          DICTIONARIES        MAPPINGMETHODS      SLICINGS
ATTRIBUTEMETHODS    DICTIONARYLITERALS  MAPPINGS            SPECIALATTRIBUTES
ATTRIBUTES          DYNAMICFEATURES     METHODS             SPECIALIDENTIFIERS
AUGMENTEDASSIGNMENT ELLIPSIS            MODULES             SPECIALMETHODS
BASICMETHODS        EXCEPTIONS          NAMESPACES          STRINGMETHODS
BINARY              EXECUTION           NONE                STRINGS
BITWISE             EXPRESSIONS         NUMBERMETHODS       SUBSCRIPTS
BOOLEAN             FLOAT               NUMBERS             TRACEBACKS
CALLABLEMETHODS     FORMATTING          OBJECTS             TRUTHVALUE
CALLS               FRAMEOBJECTS        OPERATORS           TUPLELITERALS
CLASSES             FRAMES              PACKAGES            TUPLES
CODEOBJECTS         FUNCTIONS           POWER               TYPEOBJECTS
COMPARISON          IDENTIFIERS         PRECEDENCE          TYPES
COMPLEX             IMPORTING           PRIVATENAMES        UNARY
CONDITIONAL         INTEGER             RETURNING           UNICODE
CONTEXTMANAGERS     LISTLITERALS        SCOPING
CONVERSIONS         LISTS               SEQUENCEMETHODS
DEBUGGING           LITERALS            SEQUENCES

help> |
```

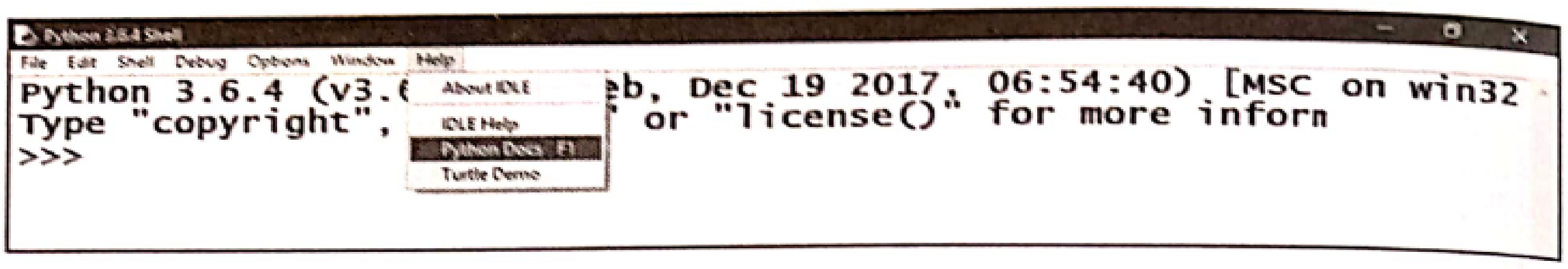
Figure 2.33: Getting Help on Topics



```
CODEOBJECTS          FRAMES            PACKAGES            TUPLES
COMPARISON           FUNCTIONS         POWER               TYPEOBJECTS
COMPLEX              IDENTIFIERS       PRECEDENCE          TYPES
CONDITIONAL          IMPORTING         PRIVATENAMES        UNARY
CONTEXTMANAGERS      INTEGER           RETURNING           UNICODE
CONVERSIONS          LISTLITERALS      SCOPING
DEBUGGING            LISTS             SEQUENCEMETHODS

help> FUNCTIONS
Functions
*********

Function objects are created by function definitions.  The only
operation on a function object is to call it: "func(argument-list)".

There are really two flavors of function objects: built-in functions
and user-defined functions.  Both support the same operation (to call
the function), but the implementation is different, hence the
different object types.

See *Function definitions* for more information.

Related help topics: def, TYPES

help>
```

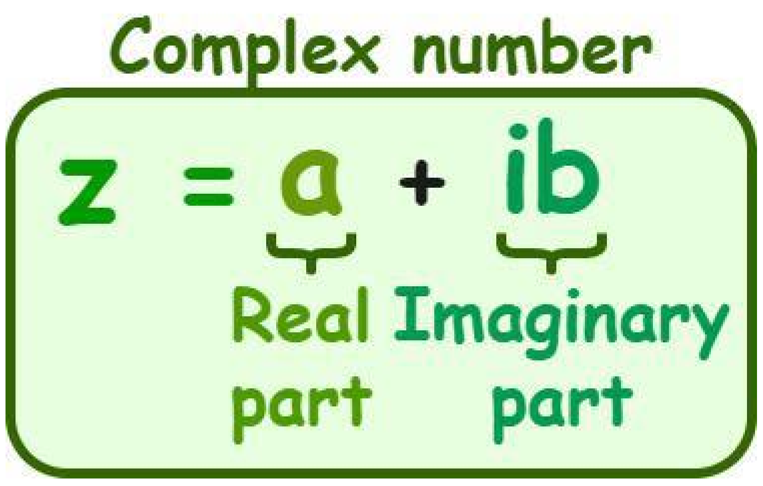
Figure 2.34: Getting Help on Functions

Python Docs



```
Python 3.6.4 (v3.6     eb, Dec 19 2017, 06:54:40) [MSC on win32
Type "copyright",      " or "license()" for more inform
>>>
```

Figure 2.38: Getting Help from Python Documentation



Comments in Python

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a `#`, and Python will ignore them:

```
#This is a comment

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

Multiline Comments

Python does not really have a syntax for multiline comments.

To add a multiline comment you could insert a `#` for each line:

```
#This is a comment

#written in

#more than just one line

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

you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""

This is a comment

written in

more than just one line

"""

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

Variables in Python

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

Casting

If you want to specify the data type of a variable, this can be done with casting.

Example

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Get the Type

You can get the data type of a variable with the `type()` function.

Example

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

Single or Double Quotes?

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John’
```

Case-Sensitive

Variable names are case-sensitive.

Example

This will create two variables:

```
a = 4
A = "Sally"
#A will not overwrite a
```

Naming Convention in Python

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with **a letter or the underscore character**
- A variable name **cannot start with a number**
- A variable name can only **contain alpha-numeric characters and underscores (A-z, 0-9, and _)**
- Variable names **are case-sensitive (age, Age and AGE are three different variables)**
- A variable name **cannot be any of the Python keywords.**

Multi Words Variable Names

Variable names **with more than one word** can be **difficult to read.**

There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "John"
```

Python Variables - Assign Multiple Values

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Note: Make sure the number of variables matches the number of values, or else you will get an error.

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example

Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Python - Output Variables

Output Variables

The Python `print()` function is often used to output variables.

Example

```
x = "Python is awesome"
print(x)
```

In the `print()` function, you output multiple variables, separated by a comma:

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

You can also use the `+` operator to output multiple variables:

Example

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Notice the space character after `"Python "` and `"is "`, without them the result would be "Pythonisawesome".

For numbers, the `+` character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

In the `print()` function, when you try to combine a string and a number with the `+` operator, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```

The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

Example

```
x = 5
y = "John"
print(x, y)
```

Data types in Python

Built in Data types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Getting the Data Type

You can get the data type of any object by using the `type()` function:

```
x = 5
print(type(x))
```

Output :

```
<class 'int'>
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

Python range() Function

Definition and Usage

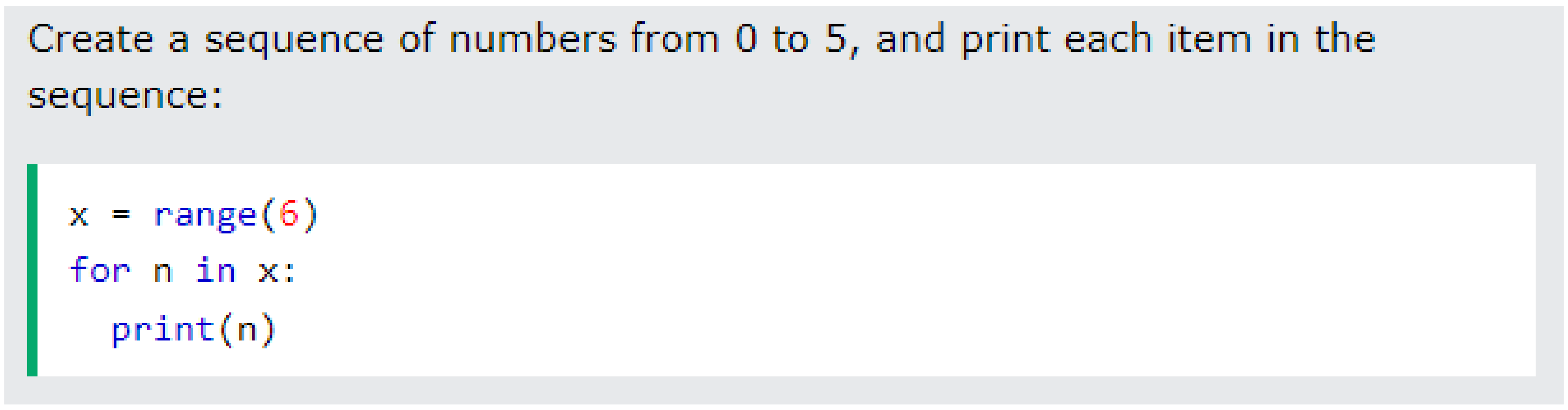
The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Syntax

range(*start*, *stop*, *step*)

Parameter Values

Parameter	**Description**
start	**Optional. An integer number specifying at which position to start. Default is 0**
stop	**Required. An integer number specifying at which position to stop (not included).**
step	**Optional. An integer number specifying the incrementation. Default is 1**



Create a sequence of numbers from 0 to 5, and print each item in the sequence:

```
x = range(6)
for n in x:
  print(n)
```

Example

Create a sequence of numbers from 3 to 5, and print each item in the sequence:

```
x = range(3, 6)
for n in x:
  print(n)
```

Create a sequence of numbers from 3 to 19, but increment by 2 instead of 1:

```
x = range(3, 20, 2)
for n in x:
  print(n)
```

```
Data types in Python
None Type in Python
```

None – None is an instance of the NoneType object type. And it is a particular variable that has no objective value. While new NoneType objects cannot be generated, None can be assigned to any variable.

Null – There is no null in Python, we can use None instead of using null values.

```
Example:
print(type(None))
print(type(Null))
```

Output:

```
<class 'NoneType'>
---------------------------------------------------------
-----------------
NameError                                 Traceback (most
recent call last)
Input In [9], in <cell line: 2>()
     1 print(type(None))
----> 2 print(type(Null))
NameError: name 'Null' is not defined
```

Declaring a variable as None.

```
var = None
# checking it's value
if var is None:
    print("var has a value of None")
else:
    print("var has a value")
```

Output:

```
var has a value of None
```

Numeric Data Types in Python

1) Int

```
x = 20
#display x:
print(x)
#display the data type of x:
print(type(x))
```

2) Float

```
x = 20.5
#display x:
print(x)
#display the data type of x:
print(type(x))
```

3) Complex

```
x = 1j
#display x:
print(x)
#display the data type of x:
print(type(x))
```

Explicit Type Conversion in Python

In programming, type conversion is the process of converting data of one type to another. For example: converting `int` data to `str`.

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion
- Explicit Conversion - manual type conversion

Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```

Output

```
Value: 124.23
Data Type: <class 'float'>
```

As we can see `new_number` has value 124.23 and is of the `float` data type.

It is because Python always converts smaller data types to larger data types to avoid the loss of data.

Note:

- We get `TypeError`, if we try to add `str` and `int`. For example, `'12' + 23`. Python is not able to use Implicit Conversion in such conditions.
- Python has a solution for these types of situations which is known as Explicit Conversion.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the built-in functions like `int()`, `float()`, `str()`, etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 2: Addition of string and integer Using Explicit Conversion

```
num_string = '12'
num_integer = 23

print("Data type of num_string before Type Casting:",type(num_string))

# explicit type conversion
num_string = int(num_string)

print("Data type of num_string after Type Casting:",type(num_string))

num_sum = num_integer + num_string

print("Sum:",num_sum)
print("Data type of num_sum:",type(num_sum))
```

Output

```
Data type of num_string before Type Casting: <class 'str'>
Data type of num_string after Type Casting: <class 'int'>
Sum: 35
Data type of num_sum: <class 'int'>
```

Sequences in Python

Python Sequences

In Python programming, sequences are a generic term for an ordered set which means that the order in which we input the items will be the same when we access them.

Python supports six different types of sequences. These are strings, lists, tuples, byte sequences, byte arrays, and range objects.

Types of Python Sequences

Python Strings

Strings are a group of characters written inside a single or double-quotes. Python does not have a character type so a single character inside quotes is also considered as a string.

Code:

```
name = “Somlalit”

type(name)
```

Output:

```
<class ‘str’>
```

Strings are immutable in nature so we can reassign a variable to a new string but we can’t make any changes in the string.

Code:

```
city= ‘China’

print(city[2])

city[2] = ‘a’
```

To declare an empty string, we may use the function str():

```
>>> name=str()

>>> name
```

Output

```
”
```

```
>>> name=str('Computer')

>>> name
```

Output

```
'Computer'

>>> name[3]
```

Output

```
‘p’
```

Python Lists

Python lists are similar to an array but they allow us to create a heterogeneous collection of items inside a list. A list can contain numbers, strings, lists, tuples, dictionaries, objects, etc.

Lists are declared by using square brackets around comma-separated items.

Syntax:

```
list1 = [1,2,3,4]

list2 = [‘red’, ‘green’, ‘blue’]

list3 = [‘hello’, 100, 3.14, [1,2,3] ]
```

Lists are mutable which makes it easier to change and we can quickly modify a list by directly accessing it.

Code:

```
list = [10,20,30,40]

list[1] = 100

print( list)
```

Output:

```
[10, 100, 30, 40]
```

```
>>> groceries=['milk','bread','eggs']

>>> groceries[1]
```

Output

```
‘bread’
```

```
>>> groceries[:2]
```

Output

```
[‘milk’, ‘bread’]
```

A Python list can hold all kinds of items; this is what makes it heterogenous.

```
>>> mylist=[1,'2',3.0,False]
```

Also, a list is mutable. This means we can change a value.

```
>>> groceries[0]='cheese'

>>> groceries
```

Output

```
[‘cheese’, ‘bread’, ‘eggs’]
```

Python Tuples

Tuples are also a sequence of Python objects. A tuple is created by separating items with a comma. They can be optionally put inside

the parenthesis () but it is necessary to put parenthesis in an empty tuple.

A single item tuple should use a comma in the end.

Code:

```
tup = ()
print( type(tup) )
tup = (1,2,3,4,5)
tup = ( “78 Street”, 3.8, 9826 )

print(tup)
```

Output:

```
<class ‘tuple’>

(’78 Street’, 3.8, 9826)
```

Tuples are also immutable like strings so we can only reassign the variable but we cannot change, add or remove elements from the tuple.

Code:

```
tup = (1,2,3,4,5)

tup[2] = 10
```

Output:

```
Traceback (most recent call last):

  File “<stdin>”, line 2, in <module>

TypeError: ‘tuple’ object does not support item assignment
```

```
>>> name=('Ayushi','Sharma')

>>> type(name)
```

Output

```
<class ‘tuple’>
```

We can also use the function tuple().

```
>>> name=tuple(['Ayushi','Sharma'])

>>> name
```

Output

```
('Ayushi', 'Sharma')
```

Like we said, a tuple is immutable. Let's try changing a value.

```
>>> name[0]='Avery'
```

Output

```
Traceback (most recent call last):File "<pyshell#594>",

line 1, in <module>

name[0]='Avery'

TypeError: 'tuple' object does not support item assignment
```

Bytes Sequences in Python

The bytes() function in Python is used to return an immutable bytes sequence. Since they are immutable, we cannot modify them.

If you want a mutable byte sequence, then it is better to use byte arrays. This is how we can create a byte of a given integer size.

Code:

```
size = 10

b = bytes(size)

print( b )
```

Output:

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Iterables can be converted into bytes.

Code:

```
print( bytes([4,2,1]) )
```

Output:

```
b’\x04\x02\x01′
```

For strings, we have to provide the encoding in the second parameter.

Code:

```
bytes(“Hey”, ‘utf=8’)
```

Output:

```
b’Hey’
```

Since it is immutable, if we try to change an item, it will raise a TypeError.

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

```
>>> a
```

Output

```
b’\x01\x02\x03\x04\x05′
```

```
>>> a[4]=3
```

Output

```
Traceback (most recent call last):File “<pyshell#46>”,

line 1, in <module>

a[4]=3

TypeError: ‘bytes’ object does not support item assignment
```

Byte Arrays in Python

Byte arrays are similar to bytes sequence. The only difference here is that byte arrays are mutable while bytes sequences are immutable. So, it also returns the bytes object the same way.

Code:

```
print( bytearray(4) )
print( bytearray([1, 2, 3, 4]) )

print( bytearray(‘Hola!’, ‘utf-8’))
```

Output:

```
bytearray(b’\x00\x00\x00\x00′)

bytearray(b’\x01\x02\x03\x04′)

bytearray(b’Hola!’)
```

Since byte arrays are mutable, let’s try changing a byte from the array.

Code:

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

print(a)
```

Output:

```
bytearray(b’\x01\x03\x04\x05′)

bytearray(b’\x01\x03\x02\x05′)
```

Python range() objects

range() is a built-in function in Python that returns us a range object. The range object is nothing but a sequence of integers. It generates the integers within the specified start and stop range.

Let’s see this with an example.

Code:

```
num = range(10)

print( type(num) )
```

Output:

```
<class ‘range’>
```

Since range object generates integers, we can access them by iterating using a for loop.

Code:

```
for i in range(5):

  print(i)
```

Output:

```
0

1

2

3

4
```

```
>>> for i in range(7,0,-1):

print(i)
```

Output

```
7
```

```
6

5

4

3

2

1
```

Python Sets

A set is a collection of unique data. That is, elements of a set cannot be duplicated. For example,

Suppose we want to store information about student IDs. Since student IDs cannot be duplicate, we can use a set.

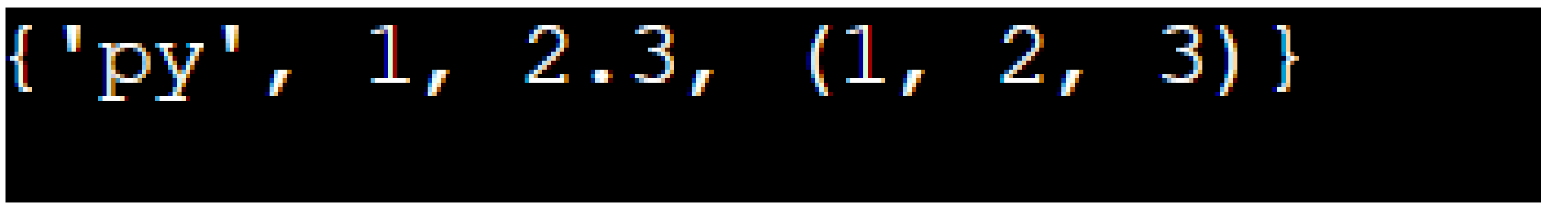
112
114
116
118
115

Set of Student ID

- ☐ Set is a collection of well-defined objects as per mathematical definition.
- ☐ Set in python is the same as in mathematics. Set is a Build in Data Type in python to store different unique, iterable data types in the unordered form.
- ☐ Set can have elements of many data types such as int, string, tuple, etc.
- ☐ Set is based on the hashing concept which makes it optimised in the search operation.
- ☐ So, it is used in case we have to perform many search operations on the collection of data.
- ☐ Set data structure is used when we want each element to be unique in a large collection of data.

```
x = {1,2.3, "py", (1,2,3)}
print(x)
```

Output:



```
{'py', 1, 2.3, (1, 2, 3)}
```

Properties of Set Data Type:

Set follow three properties:

1. **Unordered**
2. **Unique**
3. **Immutable**
1. **Set store the element in an unordered manner such that no direct access to the element is possible. We use an iterator to iterate over all elements of the set.**
2. **Set store the element uniquely. No duplicate element is allowed. If we try to insert an element that is already present in the set then the element will not insert and remain as it is. The frequency of each element is set as one.**
3. **Each element in the set is immutable which means elements can not be changed. Hence sets can contain elements of Datatype int, string, Tuple, etc but can not contain List, sets, dictionaries. set elements are immutable but the set is mutable which means we can insert and delete elements from the set.**

Program:

```
x = {4, 2, 2, 6, 1, 0, 6}
print(x)
y = {1, 2, [3, 4], 5}
```

Output:

```
{0, 1, 2, 4, 6}
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    y = {1, 2, [3, 4], 5}
TypeError: unhashable type: 'list'
```

Initialization of set:

Set can be initialized in two ways:

1. **By set() Constructor**
2. **By curly {} braces**

By set() Constructor

A set can be created with a set() constructor by passing an iterable element such as list, string, etc in the method.

Syntax:

```
variable = set(iterable element)
```

By Curly {} brackets:

A set can be created bypassing elements inside curly braces separated by a comma. These Elements can be any Immutable data type such as int, string, bool, tuple, etc but can not be List, sets, dictionaries.

Syntax:

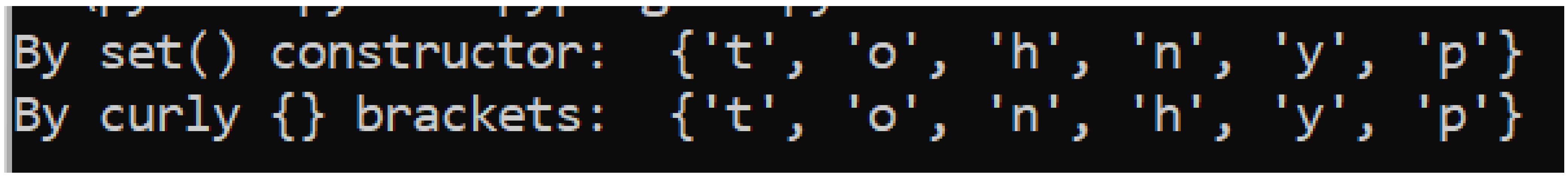
```
variable = {element1, element2,..}
```

Program:

```
# Initialization of set by set() Constructor
x = set("python")
print('By set() constructor: ', x)

# Initialization of set by curly {}brackets
y={'y', 'n', 't', 'p', 'h', 'o'}
```

```
print('curly {}brackets: ', y)
```

Output:



```
By set() constructor:  {'t', 'o', 'h', 'n', 'y', 'p'}
By curly {} brackets:  {'t', 'o', 'n', 'h', 'y', 'p'}
```

Traversal the set:

Set can be traversed with help of (for in) keywords. As no direct access to the element is possible therefore only linear traversal is possible in the set.

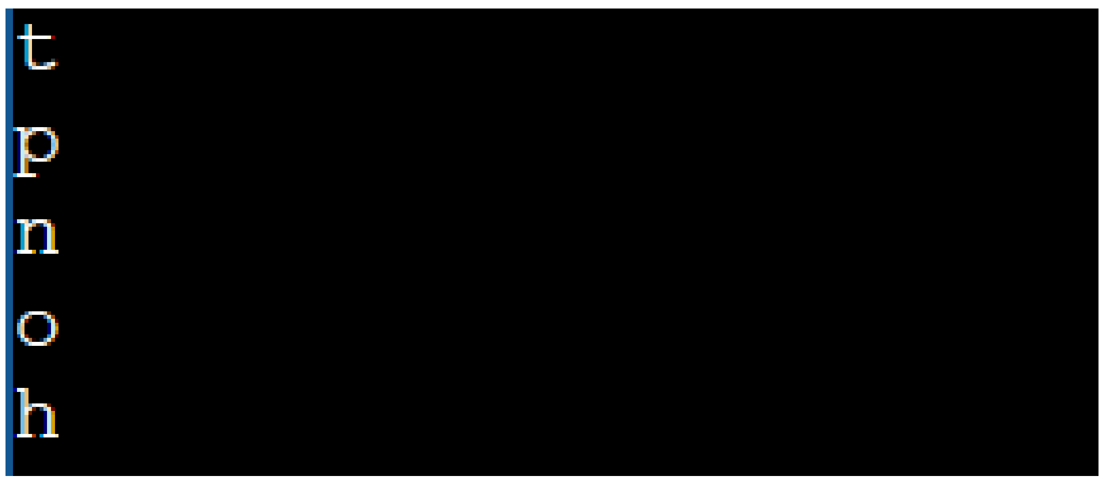
Syntax:

```
for element in set:
    print(element)
```

Program:

```
# Initialization of set x
x={'p','h','t','n','o'}

# Traversal the set with for loop
for y in x:
    print(y)
```

Output:



```
t
p
n
o
h
```

Insertion into the set:

Insertion into the set can be done by five method:

1. **add()**
2. **update()**

3. **intersection_update ()**
4. **difference_update()**
5. **symmetric_difference_update()**

Add()

Add() method is used to add a unique element into the set bypassing the element into the add method. The element can be any Immutable data type such as int, string, tuple, etc.

Syntax:

```
set.add(element)
```

Update()

Update() method is used to add multiple elements into the set. Multiple elements can be in form of iterable data types such as set, list, tuple, string, etc

Syntax:

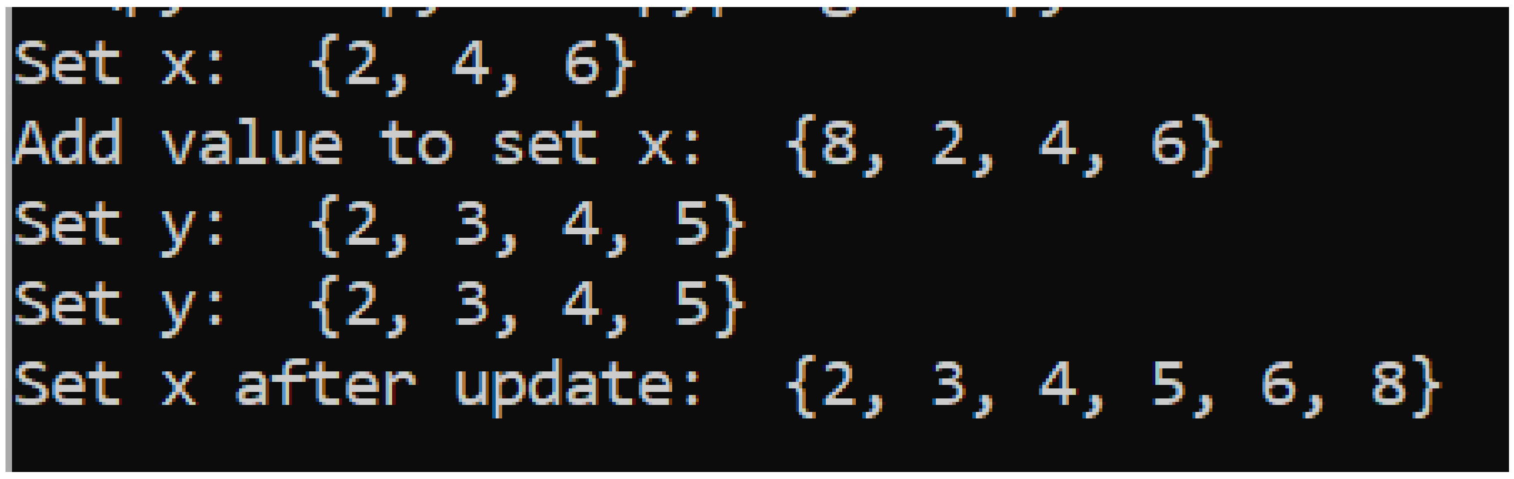
```
set1.update(list)
```

Program:

```
# Initialization of set x
x = {2,4,6}
print('Set x: ', x)

# add 8 to the set x
x.add(8)
print('Add value to set x: ', x)

# Initialization set y
y={2,3,4,5}
print('Set y: ', y)

# Update y to set x
x.update(y)
```

```
print('Set y: ', y)
print('Set x after update: ', x)
```

Output:



```
Set x:  {2, 4, 6}
Add value to set x:  {8, 2, 4, 6}
Set y:  {2, 3, 4, 5}
Set y:  {2, 3, 4, 5}
Set x after update:  {2, 3, 4, 5, 6, 8}
```

Intersection_update

Intersection_update updates the first set by tacking common elements from both sets.

Syntax:

```
set1.intersection_update(set2)
```

Difference_update:

Difference_update updates the first set by removing the common element of two sets from the first set.

Syntax:

```
set1.difference_update(set2)
```

Symmetric_difference

Symmetric_difference updates the first set by tacking elements that are not common in both sets.

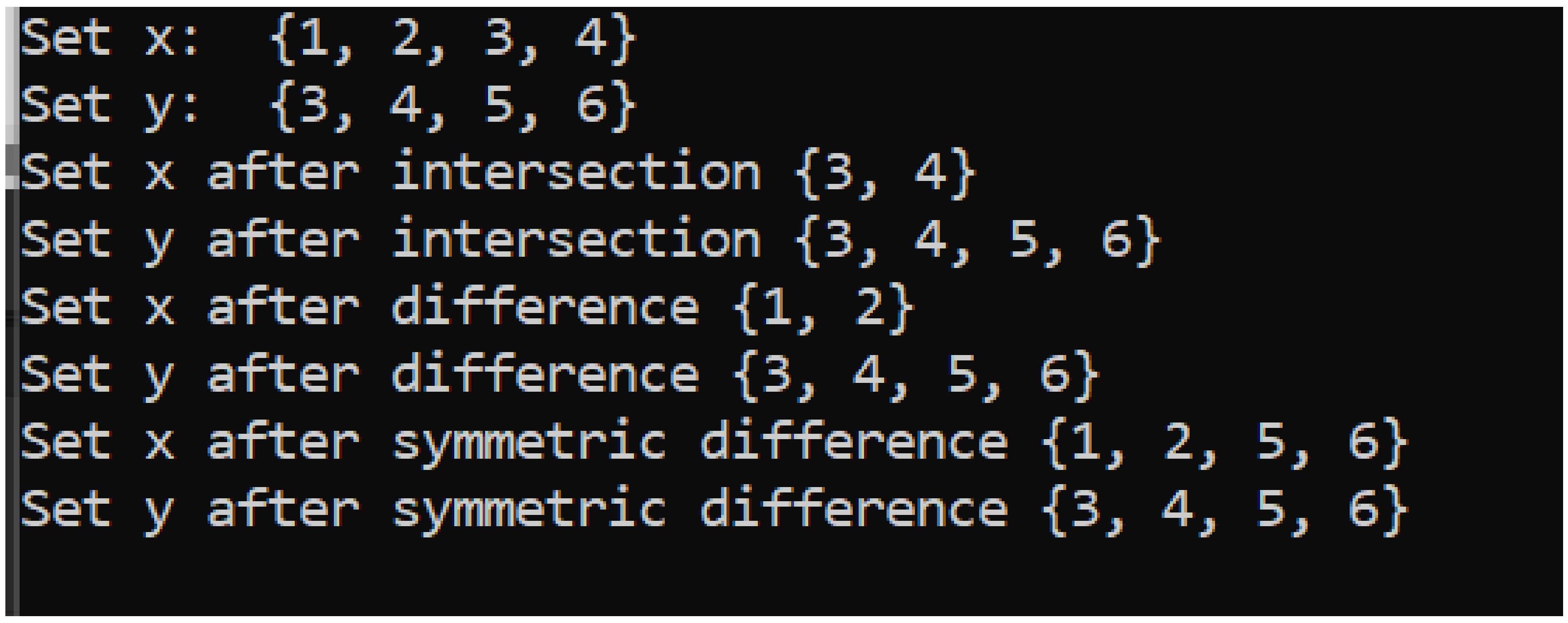
Syntax:

```
set1.symmetric_difference_update(set2)
```

Program:

```
# Initialization of set x & y
x={1,2,3,4}
y={3,4,5,6}
print('Set x: ', x)
print('Set y: ', y)

# Intersection of x & y
x.intersection_update(y)
print('Set x after intersection: ', x)
print('Set y after intersection: ', y)

# Difference of x and y
x={1,2,3,4}
x.difference_update(y)
print('Set x after difference: ', x)
print('Set y after difference: ', y)

# Symmetric difference of x and y
x={1,2,3,4}
x.symmetric_difference_update(y)
print('Set x after symmetric difference: ', x)
print('Set y after symmetric difference: ', y)
```

Output:



```
Set x:  {1, 2, 3, 4}
Set y:  {3, 4, 5, 6}
Set x after intersection {3, 4}
Set y after intersection {3, 4, 5, 6}
Set x after difference {1, 2}
Set y after difference {3, 4, 5, 6}
Set x after symmetric difference {1, 2, 5, 6}
Set y after symmetric difference {3, 4, 5, 6}
```

Deletion from the set

Deletion operation on the set can be performed by these four methods:

1. **remove**
2. **discard**

3. pop
4. clear

Remove

Remove method removes an element from the set which is passed in the method from the set. If the element is not present in the set then it raises an error message.

Syntax:

```
set.remove(element)
```

Discard

Discard method removes an element from the set which is passed in the method from the set. If the element is not present in the set then it gives no error message.

Syntax:

```
set.discard(element)
```

pop()

pop() gives the last element of the set by removing the last element from the set.

Syntax:

```
variable = set.pop()
```

clear()

clear() method removes all the elements of the set (set becomes null).

Syntax:

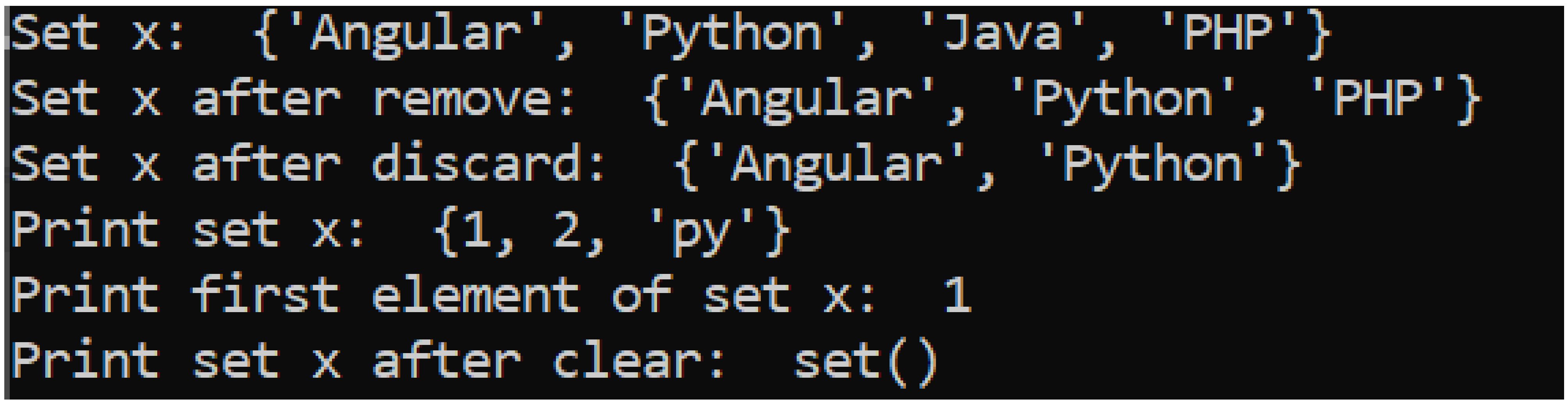
```
set.clear()
```

Program:

```
# Initialization of set x
x={'Python','Java','PHP','Angular'}
print('Set x:  ', x)
x.remove('Java')
```

```
print('Set x after remove: ', x)

x.discard('PHP')
print('Set x after discard: ', x)

# Initialization set x
x={1,2,"py"}
print('Print set x: ', x)
# pop last element from the set x
z=x.pop()
print('Print first element of set x: ', z)

# clear all element from set
x.clear()
print('Print set x after clear: ', x)
```

Output:



```
Set x:  {'Angular', 'Python', 'Java', 'PHP'}
Set x after remove:  {'Angular', 'Python', 'PHP'}
Set x after discard:  {'Angular', 'Python'}
Print set x:  {1, 2, 'py'}
Print first element of set x:  1
Print set x after clear:  set()
```

Set operation:

We can perform the following mathematical operation on the set:

1. **union**
2. **intersection**
3. **difference**
4. **symmetric_difference**
5. **issubset**
6. **isdisjoint**

Union()

Union() method gives the combination of the elements from both sets.

Syntax:

```
newSet=set1.union(set2)
```

Intersection

Intersection method gives the common elements from both sets.

Syntax:

```
newSet=set1.intersection (set2)
```

Difference

Difference method gives all elements which are in sets are not common between.

Syntax:

```
newSet=set1.difference (set2)
```

Symmetric difference

Symmetric method gives the elements which are in both sets but are not common.

Syntax:

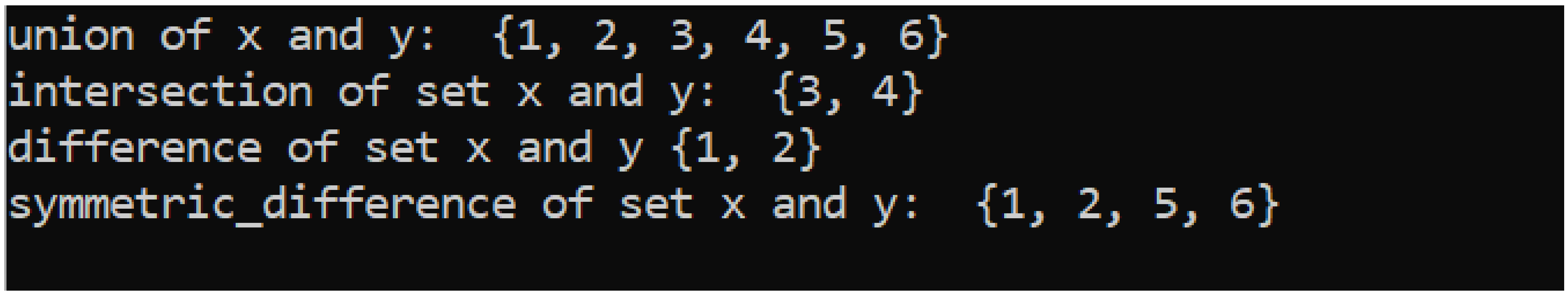
```
newSet=set1.symmetric_difference(set2)
```

Program:

```
# Initialization of set x and y
x={1,2,3,4}
y={3,4,5,6}

# Union of set x and y
z=x.union(y)
print('union of x and y: ', z)

# intersection of x and y
```

```
z=x.intersection(y)
print('intersection of set x and y: ', z)

# difference of set x and y
z=x.difference(y)
print('difference of set x and y', z)

# symmetric_difference of set x and y
z=x.symmetric_difference(y)
print('symmetric_difference of set x and y: ',z)
```

Output:



```
union of x and y:  {1, 2, 3, 4, 5, 6}
intersection of set x and y:  {3, 4}
difference of set x and y {1, 2}
symmetric_difference of set x and y:  {1, 2, 5, 6}
```

Duplicate Items in a Set

Let's see what will happen if we try to include duplicate items in a set.

```
numbers = {2, 4, 6, 6, 2, 8}
print(numbers)   # {8, 2, 4, 6}
```

Python frozenset()

Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

Due to this, frozen sets can be used as keys in a Dictionary or as elements of another set. But like sets, it is not ordered (the elements can be set at any index).

The syntax of `frozenset()` function is:

```
frozenset([iterable])
```

Example 1: Working of Python frozenset()

```
# tuple of vowels
vowels = ('a', 'e', 'i', 'o', 'u')

fSet = frozenset(vowels)
print('The frozen set is:', fSet)
print('The empty frozen set is:', frozenset())

# frozensets are immutable
fSet.add('v')
```

Output

```
The frozen set is: frozenset({'a', 'o', 'u', 'i', 'e'})
The empty frozen set is: frozenset()
Traceback (most recent call last):
  File "<string>, line 8, in <module>
    fSet.add('v')
AttributeError: 'frozenset' object has no attribute 'add'
```

Example 2: frozenset() for Dictionary

When you use a dictionary as an iterable for a frozen set, it only takes keys of the dictionary to create the set.

```
# random dictionary
person = {"name": "John", "age": 23, "sex": "male"}

fSet = frozenset(person)
print('The frozen set is:', fSet)
```

Output

```
The frozen set is: frozenset({'name', 'sex', 'age'})
```

Frozenset operations

Like normal sets, frozenset can also perform different operations like `copy`, `difference`, `intersection`, `symmetric_difference`, and `union`.

```
# Frozensets
# initialise A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])

# copying a frozenset
C = A.copy()  # Output: frozenset({1, 2, 3, 4})
print(C)

# union
print(A.union(B))  # Output: frozenset({1, 2, 3, 4, 5, 6})

# intersection
print(A.intersection(B))  # Output: frozenset({3, 4})

# difference
print(A.difference(B))  # Output: frozenset({1, 2})

# symmetric_difference
print(A.symmetric_difference(B))  # Output: frozenset({1,
2, 5, 6})
```

Output

```
frozenset({1, 2, 3, 4})
frozenset({1, 2, 3, 4, 5, 6})
frozenset({3, 4})
frozenset({1, 2})
frozenset({1, 2, 5, 6})
```

Python Dictionary

In Python, a dictionary is a collection that allows us to store data in `key: value` pairs.

Create a Dictionary

We create a dictionary by placing key-value pairs inside curly brackets `{}`, separated by commas. For example,

```
# creating a dictionary
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}

# printing the dictionary
print(country_capitals)
```

Output

```
{'United States': 'Washington D.C.', 'Italy': 'Rome',
'England': 'London'}
```

The `country_capitals` dictionary has three elements (key-value pairs).

Note: Dictionary keys must be immutable, such as tuples, strings, integers, etc. We cannot use mutable (changeable) objects such as lists as keys.

```
# Valid dictionary
my_dict = {
  1: "Hello",
  (1, 2): "Hello Hi",
  3: [1, 2, 3]
}
print(my_dict)
# Invalid dictionary
# Error: using a list as a key is not allowed
my_dict = {
  1: "Hello",
  [1, 2]: "Hello Hi",
}
print(my_dict)
```

Python Dictionary Length

We can get the size of a dictionary by using the len() function.

```
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}

# get dictionary's length
print(len(country_capitals)) # 3
```

Access Dictionary Items

We can access the value of a dictionary item by placing the key inside square brackets.

```
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}
print(country_capitals["United States"])  # Washington
D.C.
print(country_capitals["England"]) # London
```

Change Dictionary Items

Python dictionaries are mutable (changeable). We can change the value of a dictionary element by referring to its key. For example,

```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples",
  "England": "London"
}

# change the value of "Italy" key to "Rome"
country_capitals["Italy"] = "Rome"

print(country_capitals)
```

Output

```
{'United States': 'Washington D.C.', 'Italy': 'Rome',
'England': 'London'}
```

Add Items to a Dictionary

We can add an item to the dictionary by assigning a value to a new key (that does not exist in the dictionary). For example,

```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples"
}
# add an item with "Germany" as key and "Berlin" as its
value
country_capitals["Germany"] = "Berlin"
print(country_capitals)
```

Output

```
{'United States': 'Washington D.C.', 'Italy': 'Rome',
'Germany': 'Berlin'}
```

Remove Dictionary Items

We use the `del` statement to remove an element from the dictionary. For example,

```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples"
}

# delete item having "United States" key
del country_capitals["United States"]
print(country_capitals)
```

Output

```
{'Italy': 'Naples'}
```

If we need to remove all items from the dictionary at once, we can use the `clear()` method.

```
country_capitals = {
  "United States": "New York",
  "Italy": "Naples"
}

country_capitals.clear()

print(country_capitals) # {}
```

Python Keywords

Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler.

We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.

All the keywords except `True`, `False` and `None` are in lowercase and they must be written as they are. The list of all the keywords is given below.

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

Python Identifiers

Identifiers are the name given to variables, classes, methods, etc. For example,

```
language = 'Python'
```

Here, `language` is a variable (an identifier) which holds the value `'Python'`.

We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,

```
continue = 'Python'
```

The above code is wrong because we have used `continue` as a variable name.

Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

Python divides the operators in the following groups:

- **Arithmetic operators**
- **Assignment operators**
- **Comparison operators**
- **Logical operators**
- **Identity operators**
- **Membership operators**
- **Bitwise operators**

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Example

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as
y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and
"==": this comparison returns True because x is
equal to y
```

Output:

```
True
False
True
```

Example

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is not z)

# returns False because z is the same object as x

print(x is not y)

# returns True because x is not the same object as
y, even if they have the same content

print(x != y)

# to demonstrate the difference betweeen "is not"
and "!=": this comparison returns False because x
is equal to y
```

Output:

```
False
True
False
```

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example

```
x = ["apple", "banana"]

print("banana" in x)

# returns True because a sequence with the value
"banana" is in the list
```

Output:

```
True
```

```
x = ["apple", "banana"]

print("pineapple" not in x)

# returns True because a sequence with the value
"pineapple" is not in the list
```

Output:

```
True
```

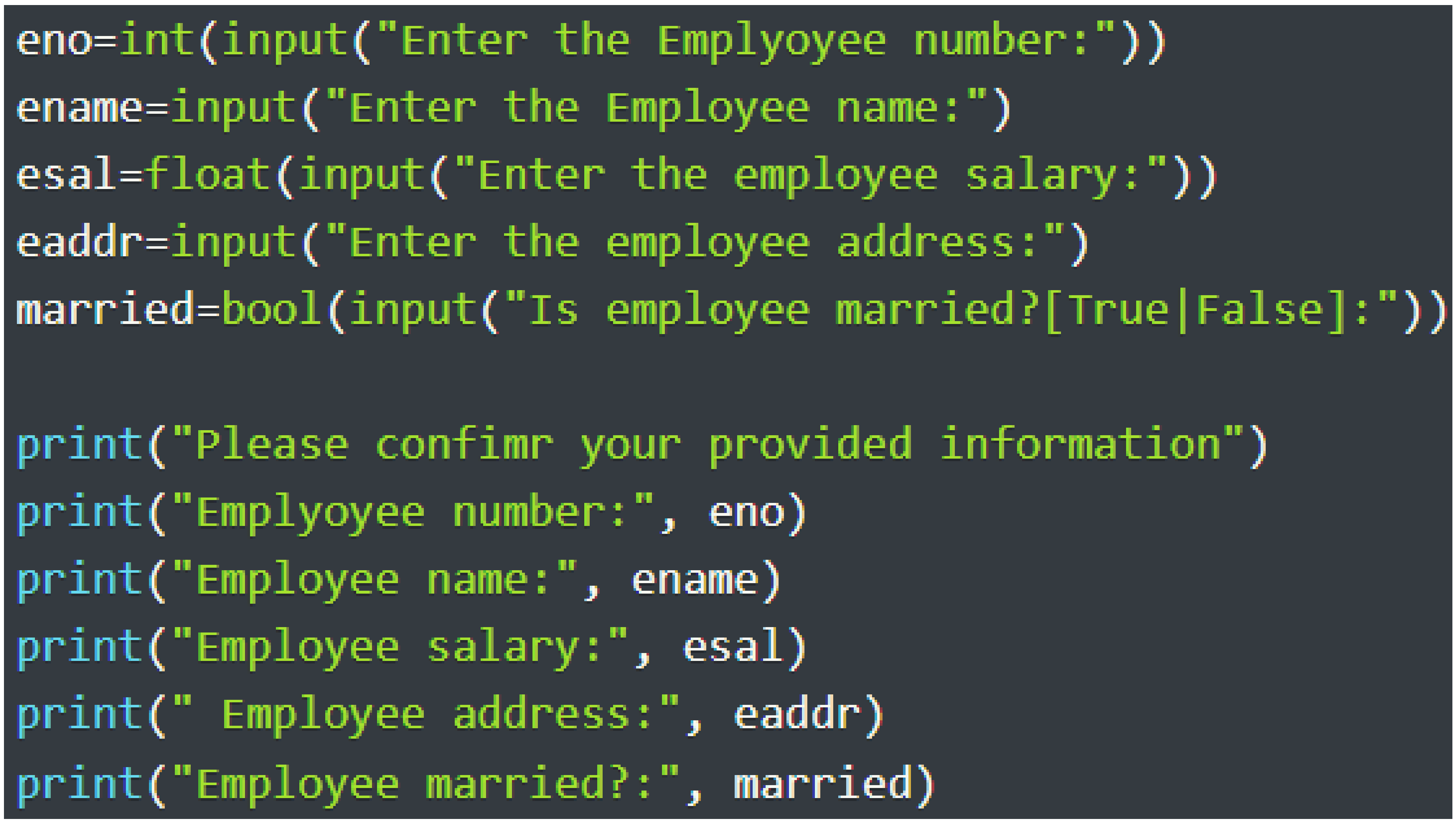
Input and Output Statements

Input : Any information or data sent to the computer from the user through the keyboard is called input.

Output: The information produced by the computer to the user is called output.

The syntax for input is input(prompt_message); the python will automatically identify whether the user entered a string, number, or list;

if the input entered from the user is not correct, then python will throw a syntax error.



```
eno=int(input("Enter the Emplyoyee number:"))
ename=input("Enter the Employee name:")
esal=float(input("Enter the employee salary:"))
eaddr=input("Enter the employee address:")
married=bool(input("Is employee married?[True|False]:"))

print("Please confimr your provided information")
print("Emplyoyee number:", eno)
print("Employee name:", ename)
print("Employee salary:", esal)
print(" Employee address:", eaddr)
print("Employee married?:", married)
```

Command Line Arguments

The arguments that are given after the name of the program in the command line shell of the operating system are known as Command Line Arguments.

Python provides various ways of dealing with these types of arguments. The most common is:

- Using sys.argv

Using sys.argv

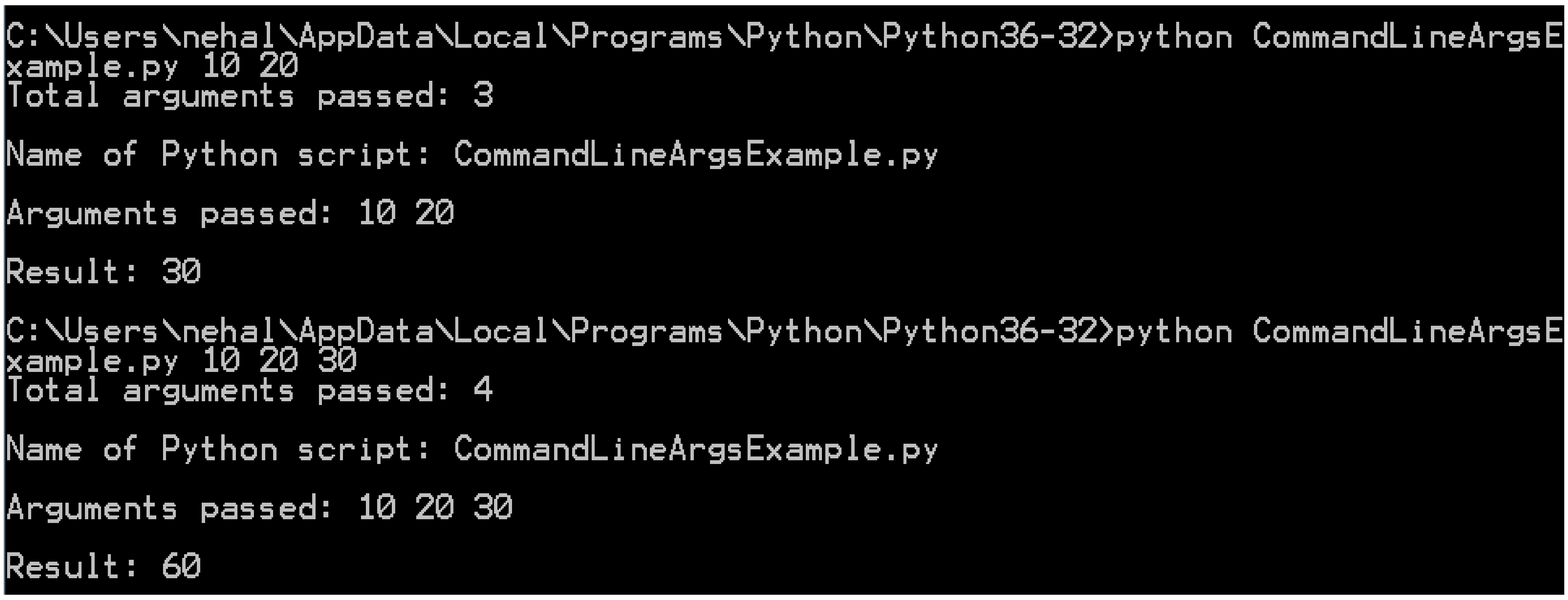
The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
One such variable is sys.argv which is a simple list structure. It’s main purpose are:

- It is a list of command line arguments.
- len(sys.argv) provides the number of command line arguments.
- sys.argv[0] is the name of the current Python script.

Example

```python
# Python program to demonstrate
# command line arguments


import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("\nName of Python script:", sys.argv[0])

print("\nArguments passed:", end = " ")
for i in range(1, n):
    print(sys.argv[i], end = " ")

# Addition of numbers
Sum = 0
# Using argparse module
for i in range(1, n):
    Sum += int(sys.argv[i])

print("\n\nResult:", Sum)
```

Output:



```
C:\Users\nehal\AppData\Local\Programs\Python\Python36-32>python CommandLineArgsE
xample.py 10 20
Total arguments passed: 3

Name of Python script: CommandLineArgsExample.py

Arguments passed: 10 20

Result: 30

C:\Users\nehal\AppData\Local\Programs\Python\Python36-32>python CommandLineArgsE
xample.py 10 20 30
Total arguments passed: 4

Name of Python script: CommandLineArgsExample.py

Arguments passed: 10 20 30

Result: 60
```

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:

  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:

print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, but it has to be at least one.

Example

```
if 5 > 2:

 print("Five is greater than two!")

if 5 > 2:

        print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example

Syntax Error:

```
if 5 > 2:

 print("Five is greater than two!")

        print("Five is greater than two!")
```

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- **Equals:** `a == b`
- **Not Equals:** `a != b`
- **Less than:** `a < b`
- **Less than or equal to:** `a <= b`
- **Greater than:** `a > b`
- **Greater than or equal to:** `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` **keyword.**

Example

If statement:

```
a = 33

b = 200

if b > a:

  print("b is greater than a")
```

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33

b = 200

if b > a:

print("b is greater than a") # you will get an error
```

Elif

The `elif` keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33

b = 33

if b > a:

  print("b is greater than a")

elif a == b:
  print("a and b are equal")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200

b = 33

if b > a:

  print("b is greater than a")

elif a == b:

  print("a and b are equal")

else:

  print("a is greater than b")
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

Example

```
a = 200
```

```
b = 33

if b > a:

  print("b is greater than a")

else:

  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2

b = 330

print("A") if a > b else print("B")
```

This technique is known as Ternary Operators, or Conditional Expressions.

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
a = 330

b = 330

print("A") if a > b else print("=") if a == b else
print("B")
```

And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
a = 200

b = 33

c = 500

if a > b and c > a:

  print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
a = 200

b = 33

c = 500

if a > b or a > c:

  print("At least one of the conditions is True")
```

Not

The `not` keyword is a logical operator, and is used to reverse the result of the conditional statement:

Example

Test if `a` is NOT greater than `b`:

```
a = 33

b = 200

if not a > b:

  print("a is NOT greater than b")
```

Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

Example

```
x = 41



if x > 10:

  print("Above ten,")

  if x > 20:

    print("and also above 20!")

  else:

    print("but not above 20.")
```

The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33

b = 200



if b > a:

  pass
```

Python Loops

Python has two primitive loop commands:

- `while` **loops**
- `for` **loops**

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1

while i < 6:

  print(i)

  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1

while i < 6:
  print(i)

  if i == 3:

    break

  i += 1
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0

while i < 6:

  i += 1

  if i == 3:

    continue

  print(i)
```

The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
```

```
while i < 6:

  print(i)

  i += 1

else:

  print("i is no longer less than 6")
```

Python For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-oriented programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)
```

The `for` loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":

  print(x)
```

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when `x` is "banana":

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)

  if x == "banana":

    break
```

Example

Exit the loop when `x` is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  if x == "banana":

    continue

  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):

  print(x)
```

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):

  print(x)
```

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):

  print(x)
```

Else in For Loop (Else Suite)

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):

  print(x)

else:

  print("Finally finished!")
```

Note: The `else` block will NOT be executed if the loop is stopped by a `break` statement.

Example

Break the loop when `x` is 3, and see what happens with the `else` block:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]

fruits = ["apple", "banana", "cherry"]

for x in adj:

  for y in fruits:
    print(x, y)
```

Example :

```
 x = [1, 2]
 y = [4, 5]

 for i in x:
   for j in y:
     print(i, j)
```

Output:

```
1 4

1 5

2 4

2 5
```

Example :

```
 for i in range(2, 4):

     # Printing inside the outer loop
     # Running inner loop from 1 to 10
     for j in range(1, 11):

         # Printing inside the inner loop
         print(i, "*", j, "=", i*j)
     # Printing inside the outer loop
     print()
```

Output :

```
2 * 1 = 2

2 * 2 = 4

2 * 3 = 6

2 * 4 = 8

2 * 5 = 10

2 * 6 = 12

2 * 7 = 14

2 * 8 = 16

2 * 9 = 18

2 * 10 = 20
```

The pass Statement

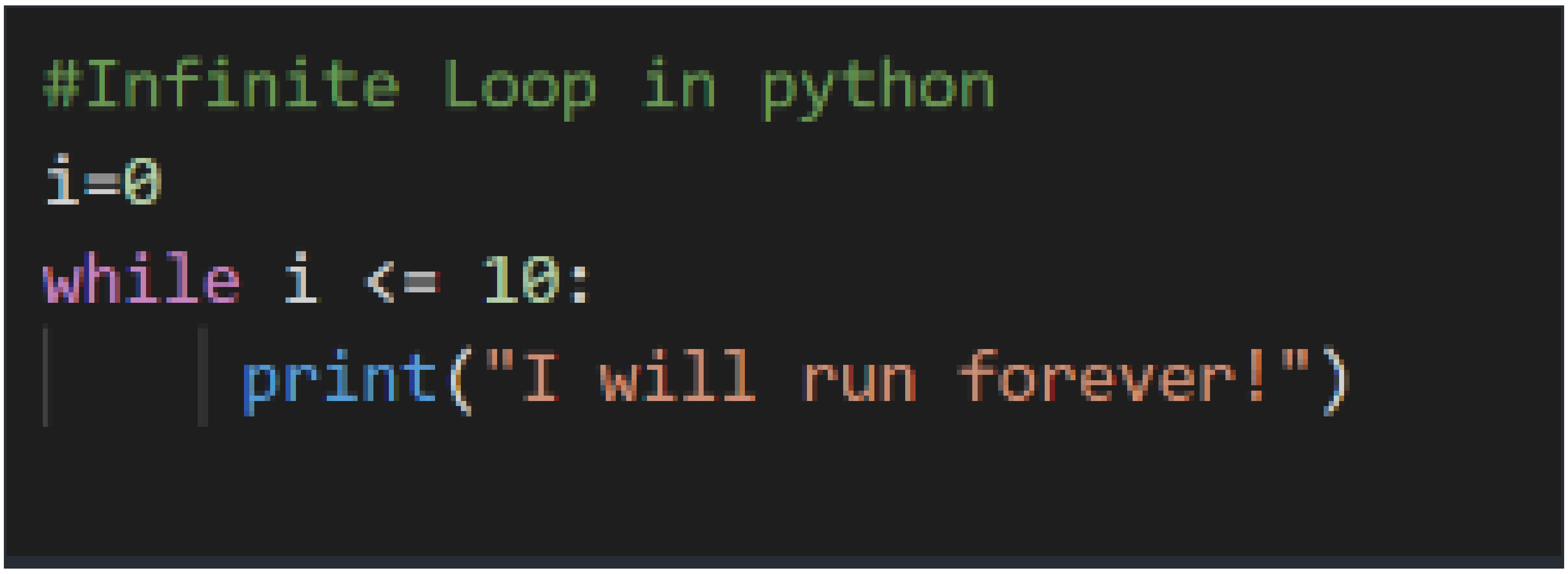
`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:

  pass
```

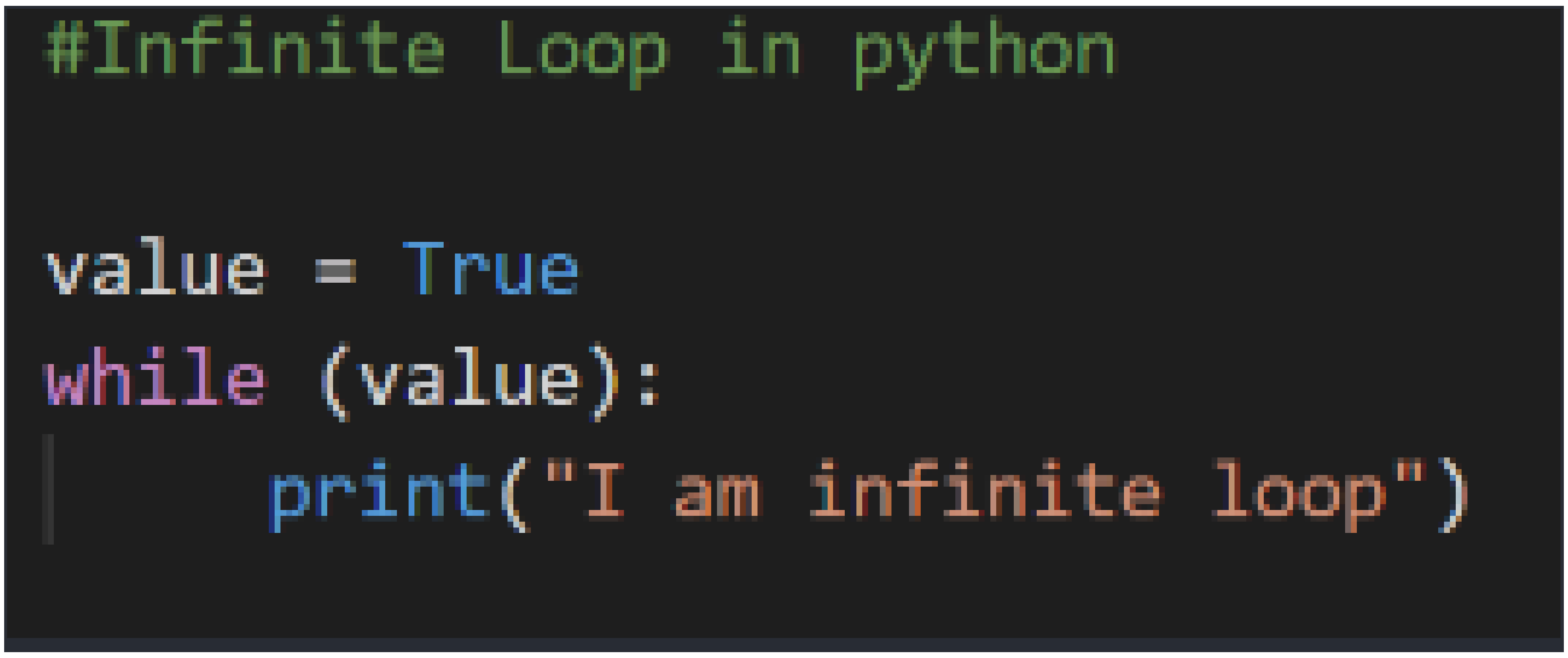
Infinite Loops

```
while True:
    print("Som Lalit")
```



```
#Infinite Loop in python
i=0
while i <= 10:
        print("I will run forever!")
```

While Statement in Python Infinite Loop



```
#Infinite Loop in python

value = True
while (value):
        print("I am infinite loop")
```

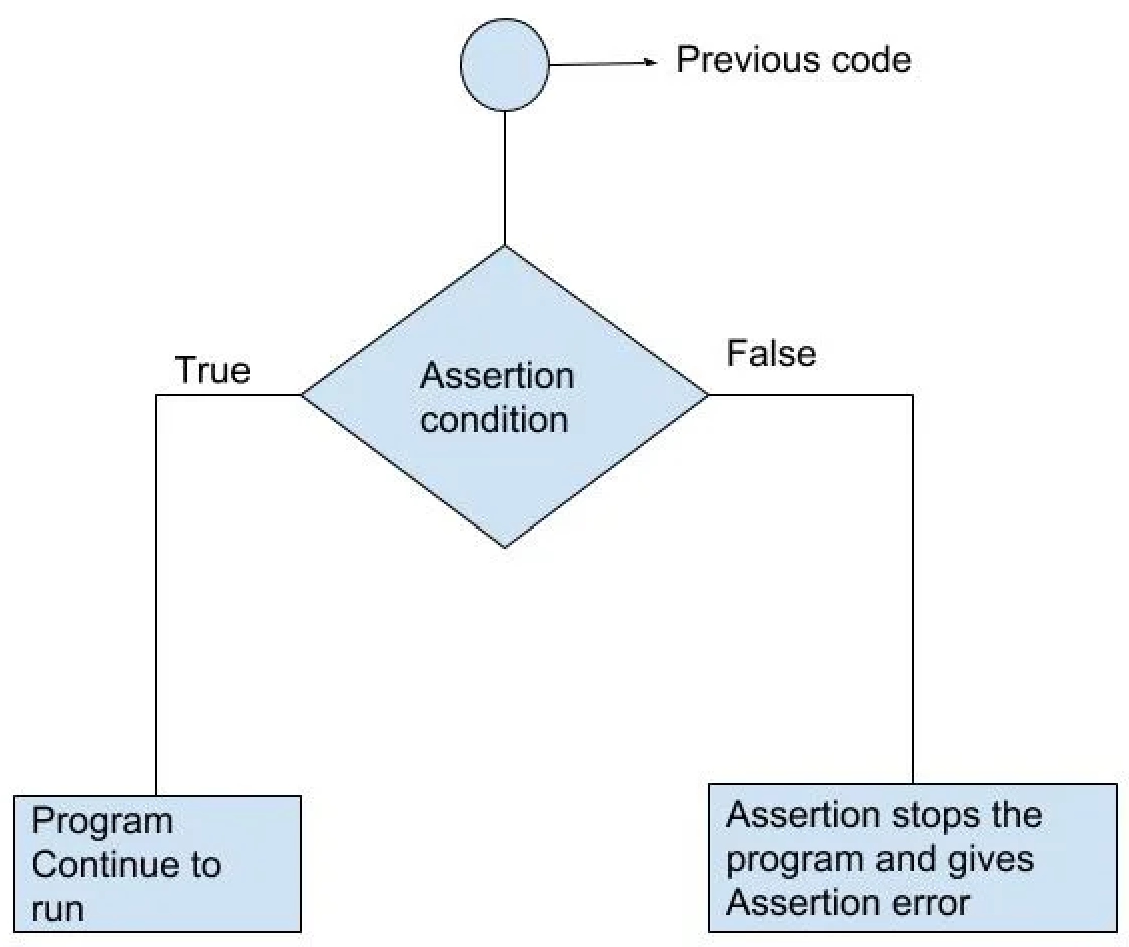
Assert Statements

The `assert` keyword is used when debugging code.

Assertion normally is an built in EXCEPTION class.

The `assert` keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

You can write a message to be written if the code returns False, check the example below.



Example :

```
x = "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be 'hello'"
Traceback (most recent call last):
  File "demo_ref_keyword_assert2.py", line 4, in <module>
    assert x == "goodbye", "x should be 'hello'"
AssertionError: x should be 'hello'
```

Example :

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

```
Traceback (most recent call last):
  File "demo_ref_keyword_assert.py", line 5, in <module>
    assert x == "goodbye"
AssertionError
```

Example :

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

```
Traceback (most recent call last):
  File "<string>", line 6, in <module>
  File "<string>", line 2, in avg
AssertionError
```

Example :

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = [10,20]
print("Average of mark1:",avg(mark1))
```

Output:

```
Average of marks: 15.0
```

Return Statement :

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A return statement is overall used to invoke a function so that the passed statements can be executed.

Note: Return statement can not be used outside the function.

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

Example:

```
def cube(x):
   r=x**3
   return r
```

```python
# Python program to
# demonstrate return statement

def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

Output:

```
Result of add function is 5


Result of is_true function is True
```