PYTHON PROGRAMMING LANGUAGE

OVERVIEW OF PYTHON

programming language is a formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behaviour of a machine or to express algorithms.

INTRODUCTION OF PYTHON

Python is an object-oriented, high level language, interpreted, dynamic and multipurpose programming language.
Python is easy to learn yet powerful and versatile scripting language which makes it attractive for Application Development.
Python's syntax and dynamic typing with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas.
Python supports multiple programming pattern, including object oriented programming, imperative and functional programming or procedural styles.
Python is not intended to work on special area such as web programming. That is why it is known as multipurpose because it can be used with web, enterprise, 3D CAD etc.
We don't need to use data types to declare variable because it is dynamically typed so we can write a=10 to declare an integer value in a variable.
Python makes the development and debugging fast because there is no compilation step included in python development and edit-test-debug cycle is very fast.
It is used for GUI and database programming, client- and server-side web programming, and application testing.
It is used by scientists writing applications for the world's fastest supercomputers and by children first learning to program.

HISTORY OF PYTHON

Python was conceptualized by Guido Van Rossum in the late 1980s. Rossum published the first version of Python code (0.9.0) in February 1991 at the CWI (Centrum Wiskunde & Informatica) in the Netherlands, Amsterdam. Python is derived from ABC programming language, which is a general-purpose programming language that had been developed at the CWI. Rossum chose the name "Python", since he was a big fan of Monty Python's Flying Circus. Python is now maintained by a core development team at the institute, although Rossum still holds a vital role in directing its progress.

COMPILER vs INTERPRETER

An interpreter is a program that reads and executes code. This includes source code, pre-compiled code, and scripts. Common interpreters include Perl, Python, and Ruby interpreters, which execute Perl, Python, and Ruby code respectively.

Interpreters and compilers are similar, since they both recognize and process source code.

However, a compiler does not execute the code like and interpreter does. Instead, a compiler simply converts the source code into machine code, which can be run directly by the operating system as an executable program.

Interpreters bypass the compilation process and execute the code directly.

Interpreters are commonly installed on Web servers, which allows developers to run executable scripts within their webpages. These scripts can be easily edited and saved without the need to recompile the code. Without an interpreter, the source code serves as a plain text file rather than an executable program.

#	COMPILER	INTERPRETER
1	Compiler works on the complete program at once. It takes the **entire program** as input.	Interpreter program works line-by-line. It takes **one statement at a time** as input.
2	Compiler generates intermediate code, called the o**bject code or machine code.**	Interpreter does not generate intermediate object code or machine code.
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs **faster than interpreter.**	Interpreter execute conditional control statements at a much **slower speed**.
4	**Compiled programs take more memory** because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, **interpreted programs are more memory efficient.**
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line-by-line every time they are run.
6	Errors are reported after the **entire program is checked** for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
7	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
8	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
9	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students.
10	**Examples** of programming languages that use compilers: C, C++, COBOL	**Examples** of programming languages that use interpreters: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

PYTHON VERSIONS

- Python 1.0
- Python 2.0
- Python 3.0

PYTHON FEATURES

- **Easy** to learn, easy to read and easy to maintain.
- **Portable:** It can run on various hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter.

- **Scalable:** Python provides a good structure and support for large programs. Python has support for an **interactive mode** of testing and debugging.
- Python has a broad standard **library cross-platform**.
- Everything in Python is an **object**: variables, functions, even code. Every object has an ID, a type, and a value.
- Python provides interfaces to all major **commercial databases**.
- Python supports functional and structured programming methods as well as **OOP**.
- Python provides very high-level **dynamic data types** and supports **dynamic type checking**.
- Python supports **GUI applications**
- Python supports **automatic garbage collection**.
- Python can be easily **integrated** with C, C++, and Java.

APPLICATIONS OF PYTHON

- Machine Learning
- GUI Applications (like Kivy, Tkinter, PyQt etc.)
- Web frameworks like Django (used by YouTube, Instagram, Dropbox)
- Image processing (like OpenCV, Pillow)
- Web scraping (like Scrapy, BeautifulSoup, Selenium)
- Test frameworks
- Multimedia
- Scientific computing
- Text processing

TYPES OF PROGRAM ERRORS

We distinguish between the following types of errors:

1. Syntax errors: errors due to the fact that the syntax of the language is not respected.
2. Semantic errors: errors due to an improper use of program statements.
3. Logical errors: errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, we distinguish:

1. Compile time errors: syntax errors and static semantic errors indicated by the compiler.
2. Runtime errors: dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

Syntax errors

Syntax errors are due to the fact that the syntax of the Java language is not respected.
Let us see some examples of syntax errors.

Example 1: Missing semicolon:

```
int a = 5 // semicolon is missing
```

Compiler message:

```
Example.java:20: ';' expected
int a = 5
```

Example 2: Errors in expressions:

```
x = ( 3 + 5; // missing closing parenthesis ')'
y = 3 + * 5;  // missing argument between '+' and '*'
```

Semantic errors

Semantic errors indicate an improper use of Java statements.
Let us see some examples of semantic errors.

Example 1: Use of a non-initialized variable:

```
int i;
i++;    // the variable i is not initialized
```

Example 2: Type incompatibility:

```
int a = "hello"; // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
String s = "...";
int a = 5 - s; // the - operator does not support arguments of type String
```

Example 4: Unknown references:

```
Strin x;                    // Strin is not defined
system.out.println("hello"); // system is not defined
String s;
s.println();        // println is not a method of the class String
```

Example 5: Array index out of range (dynamic semantic error)

```
int[] v = new int[10];
v[10] = 100;          // 10 is not a legal index for an array of 10 elements
```

The array v has been created with 10 elements (with indexes ranging from 0 to 9), and we are trying to access the element with index 10, which does not exist. This type of error is not caught during compilation, but causes an exception to be thrown at runtime.

Logical errors

Logical errors are caused by the fact that the software specification is not respected. The program is compiled and executed without errors, but does not generate the requested result.
Let us see some examples of logical errors:

Example 1: Errors in the performed computation:

```
public static int sum(int a, int b) {
return a - b ;
}
// this method returns the wrong value wrt the specification that requires
// to sum two integers
```

Example 2: Non termination:

```
String s = br.readLine();
while (s != null) {
System.out.println(s);
}  // this loop does not terminate
```

Errors detected by the compiler and runtime errors

All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler, which generates a message indicating the type of error and the position in the Java source file where the error occurred (notice that the actual error could have occurred before the position signaled by the compiler).
Other semantic errors (the dynamic semantic errors) and the logical errors cannot be detected by the compiler, and hence they are detected only when the program is executed.
Let us see some examples of errors detected at runtime:
Example 1: Division by zero:

```
int a, b, x;
a = 10;
b = Integer.parseInt(kb.readLine());
x = a / b;  //ERROR if b = 0
```

This error occurs only for a certain configuration of the input (b = 0).
Example 2: File does not exist:

```
FileReader f = new FileReader("pippo.txt");
```

The error occurs only if the file pippo.txt does not exist on the harddisk.
Example 3: Dereferencing of a null reference:

```
String s, t;
s = null;
t = s.concat("a");
```

The concat() method cannot be applied to a reference whose value is null. Note that the above code is syntactically correct, since the concat() method is correctly applied to a reference of type String, but it contains a dynamic semantic error due the fact that a method cannot be applied to a reference whose value is null.

I. STRUCTURE OF A PYTHON PROGRAM

Python Statements

In general, the interpreter reads and executes the statements line by line i.e. sequentially. Though, there are some statements that can alter this behaviour like conditional statements. Mostly, python statements are written in such a format that one statement is only written in a single line. The interpreter considers the 'new line character' as the terminator of one instruction.

Example 1:

```
print('Welcome to Geeks for Geeks')
```

Instructions that a Python interpreter can execute are called statements.
For example, a = 1 is an assignment statement.

```
if statement,
for statement,
while statement etc.  3wexc
```

Multi-line statement

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as,

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
          'green']
```

We could also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

Python Indentation

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent.

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing. But the former style is clearer.
Incorrect indentation will result into **IndentationError.**

Python Comments

Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.

In Python, we use the hash (#) symbol to start writing a comment.

It extends up to the newline character. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But theycan be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

Python Variables

A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data which can be changed later throughout programming.

Example 1: N1=10
N2=10.5
Example 2:

```
x = y = z = "same"  # assign the same value to multiple variables at once
print (x)
print (y)
print (z)
```

Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

Example 1:

```
PI = 3.14
GRAVITY = 9.8
```

Literals

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

Example 1:

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal

#Float    Literal
float_1  =  10.5
float_2 = 1.5e2

#Complex Literal
```

x = 3.14j

Python Operators

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Operators are used to perform operations on variables and values.

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Special operators
  - ✓ Identity operators
  - ✓ Membership operators

Example:

```
# Python Program to find the area of triangle

a = 5
b = 6
c = 7

# Uncomment below to take inputs from the user
# a = float(input('Enter first side: '))
# b = float(input('Enter second side: '))
# c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Output:

```
The area of the triangle is 14.70
```

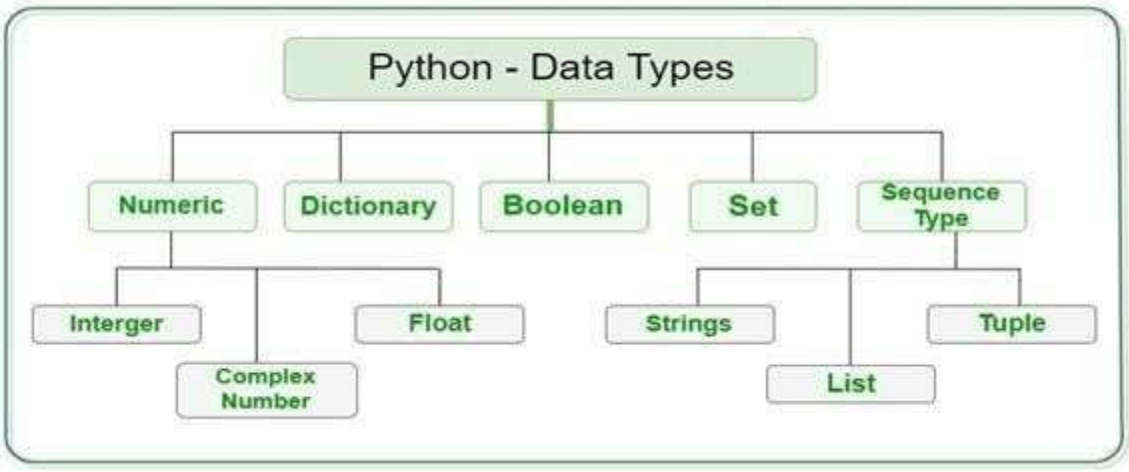
II. ELEMENTS OF PYTHON

- A Python program, sometimes called a script, is a sequence of definitions and commands.
- These definitions are evaluated and the commands are executed by the Python interpreter in something called the shell.
- Typically, a new shell is created whenever execution of a program begins. In most cases, a window is associated with the shell.

- A command, often called a statement, instructs the interpreter to do something.

The basic elements of Python are:

1. Keywords: and, assert, break, class, continue, def,del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield
2. Operators: + - * / % ** // > & | ^ ~ >= <> != ==
3. Delimiters: ( ) [ ] { }, :. ' = ; += -= *= /= //= %= &= |= ^= >>= <<= **=
4. Data types: **Numeric, Dictionary, Boolean, Set, Strings, List, Tuple**



Mutable and Immutable data types/objects:

- Example for mutable data types are: List, Set and Dictionary
- Example for Immutable data types are: Strings, tuple, int, float, bool,Unicode.

Numeric

In Python, numeric data type represent the data which has numeric value. Numeric value can be interger, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.
- **Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

Python program to demonstrate numeric value

```
a = 5
print("Type of a: ", type(a))
b = 5.0
```

```
print("\nType of b: ", type(b))
c = 2 + 4j
print("\nType of c: ", type(c))
```

OUTPUT:
Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>

Strings

In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned.

Program:

```
String1 = "IIIBSC STUDENTS"
print("Initial String: ")
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])

print("\n8th char: ")
print(String1[8])
#  Updation or deletion of characters from a String is not allowed
```

Output:
Initial String:
IIIBSC STUDENTS

First character of String is:
I

Last character of String is:
S

8th char:
T

Lists

Lists are just like the arrays, declared in other languages.
Lists need not be homogeneous always which makes it the most powerful tool in Python.
A single list may contain DataTypes like Integers, Strings, as well as Objects.

Lists are mutable, and hence, they can be altered even after their creation.
List in Python are ordered and have a definite count.
The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index.
Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.
It is represented by list class.

Creating a list

Lists in Python can be created by just placing the sequence inside the square brackets[].

```
#Python program to demonstrate Creation of List

# Creating a List
List = []
print("Intial blank List: ")
print(List)

# Creating a List with the use of a String
List = ['IIIBSCSTUDENTS']
print("\nList with the use of String: ")
print(List)

# Creating a List with the use of multiple values
List = ["III", "BSC", "STUDENTS"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List (By Nesting a list inside a List)
List = [['III', 'BSC'], ['STUDENTS']]
print("\nMulti-Dimensional List: ")
print(List)
```

Methods used in a List

```
#Append an element (ADD an element)
List.append(4)
print("\nList after Adding a number: ")
print(List)

# Addition of Element at specific Position (using Insert Method)
List.insert(2, 12)
print(List)

List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

```
# Addition of multiple elements to the List at the end (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)

# accessing a element from the list using index number
print("Accessing element from the list")
print(List[0])
print(List[2])

# accessing a element using negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])

List1=[1,2,3,4,5,6,7,8]
print("Original List")
print(List1)
# Removing elements from List using Remove() method
List1.remove(5)

print("\nList after Removal of element: ")
print(List1)

List1.pop()
print("\nList after popping an element: ")
print(List1)

# Removing element at a specific location from the set using the pop() method
List1.pop(2)
print("\nList after popping a specific element: ")
print(List1)
```

Output:

```
Intial blank List:
[]

List with the use of String:
['IIIBSCSTUDENTS']

List containing multiple values:
III
STUDENTS
```

Multi-Dimensional List:
[['III', 'BSC'], ['STUDENTS']]

List after Adding a number:
[['III', 'BSC'], ['STUDENTS'], 4]
[['III', 'BSC'], ['STUDENTS'], 12, 4]

List after performing Insert Operation:
['Geeks', ['III', 'BSC'], ['STUDENTS'], 12, 4]

List after performing Extend Operation:
['Geeks', ['III', 'BSC'], ['STUDENTS'], 12, 4, 8, 'Geeks', 'Always']
Accessing element from the list
Geeks
['STUDENTS']
Accessing element using negative indexing
Always
8
Original List
[1, 2, 3, 4, 5, 6, 7, 8]

List after Removal of element:
[1, 2, 3, 4, 6, 7, 8]

List after popping an element:
[1, 2, 3, 4, 6, 7]

List after popping a specific element:
[1, 2, 4, 6, 7]

Adding Elements to a List

Elements can be added to the List by using built-in append() function.
Only one element at a time can be added to the list by using append() method.
For addition of element at the desired position, insert() method is used.
Other than append() and insert() methods.
extend() method is used to add multiple elements at the same time at the end of the list.

Removing Elements from the List

Elements can be removed from the List by using built-in remove() function.
Pop() function can also be used to remove and return an element from the set, but by default it removes only the last element of the set, to remove element from a specific position of the List, index of the element is passed as an argument to the pop() method.
Remove method in List will only remove the first occurrence of the searched element.

Tuple

Tuple is an ordered collection of Python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers. **The important difference between a list and a tuple is that tuples are immutable.** Also, Tuples are hashable whereas lists are not. It is represented by tuple class.

Creating a Tuple

In Python, tuples are created by placing sequence of values separated by 'comma' with or without the use of parentheses for grouping of data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.). Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing 'comma' to make it a tuple.

In python, deletion or updation of a tuple is not allowed.

```
# Python program to demonstrate creation of Tuple

# Creating an empty tuple
Tuple1 = ()
print("Initial empty Tuple: ")
print (Tuple1)

# Creating a Tuple with the use of Strings
Tuple1 = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple1)

# Creating a Tuple with the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))

# Creating a Tuple with the use of built-in function
Tuple1 = tuple('Geeks')
print("\nTuple with the use of function: ")
print(Tuple1)

# Creating a Tuple with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'geek')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)

# Accessing element using indexing
print("Frist element of tuple")
print(Tuple1[0])

# Accessing element from last -- negative indexing
print("\nLast element of tuple")
```

```
print(Tuple1[-1])

print("\nThird last element of tuple")
print(Tuple1[-3])
```

OUTPUT:

```
Initial empty Tuple:
()

Tuple with the use of String:
('Geeks', 'For')

Tuple using List:
(1, 2, 4, 5, 6)

Tuple with the use of function:
('G', 'e', 'e', 'k', 's')

Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'geek'))
Frist element of tuple
0

Last element of tuple
3

Third last element of tuple
1
```

Boolean

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the class bool.

```
# Python program to demonstrate boolean type
print(type(True))
print(type(False))
print(type(true))   # Error, Small t for true is wrong
```

Set

In Python, Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Creating a set

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'. A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. The order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index.

```
# Python program to demonstrate Creation of Set in Python

# Creating a Set
set1 = set()
print("Intial blank Set: ")
print(set1)

# Creating a Set with the use of a String
set1 = set("GeeksForGeeks")
print("\nSet with the use of String: ")
print(set1)

# Creating a Set with the use of a List
set1 = set(["Geeks", "For", "Geeks"])
print("\nSet with the use of List: ")
print(set1)

# Creating a Set with a mixed type of values (Having numbers and strings)
set1 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
print(set1)
```

Methods used

```
set1.add(9)
print("\nSet after Addition of Three elements: ")
print(set1)

# Addition of elements to the Set using Update function
set1.update([10, 11])
print("\nSet after Addition of elements using Update: ")
print(set1)
# Removing elements from Set using Remove() method
set1.remove(9)
print("\nSet after Removal of two elements: ")
print(set1)

# Removing elements from Set using Discard() method
set1.discard(4)

print("\nSet after Discarding two elements: ")
```

```
print(set1)

# Removing element from the Set using the pop() method
set1.pop()
print("\nSet after popping an element: ")
print(set1)

# Removing all the elements from Set using clear() method
set1.clear()
print("\nSet after clearing all the elements: ")
print(set1)
```

Output:

```
Intial blank Set:
set()

Set with the use of String:
{'s', 'k', 'e', 'F', 'o', 'G', 'r'}

Set with the use of List:
{'Geeks', 'For'}

Set with the use of Mixed Values
{1, 2, 4, 'Geeks', 6, 'For'}

Set after Addition of Three elements:
{1, 2, 4, 'Geeks', 6, 'For', 9}

Set after Addition of elements using Update:
{1, 2, 4, 'Geeks', 6, 'For', 9, 10, 11}

Set after Removal of two elements:
{1, 2, 4, 'Geeks', 6, 'For', 10, 11}

Set after Discarding two elements:
{1, 2, 'Geeks', 6, 'For', 10, 11}

Set after popping an element:
{2, 'Geeks', 6, 'For', 10, 11}

Set after clearing all the elements:
set()
```

Removing elements from a set

Elements can be removed from the Set by using built-in remove() function but a KeyError arises if element doesn't exist in the set. To remove elements from a set without KeyError, use discard(). Pop() function can also be used to remove and return an element from the set, but it removes only the last element of the set. To remove all the elements from the set, clear() function is used.

Dictionary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a 'comma'.

Creating a dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its Key:value. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable.
Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.
Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

```
# Python Program
Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary  with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed  Keys: ")
print(Dict)

# Creating a Dictionary with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary  with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)

# Adding elements one at a time
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)
```

```
# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)
```

Methods used

```
# Python program to demonstrate for accessing elements from a Dictionary

# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using get() method
print("Accessing a element using get:")
print(Dict.get(3))

# Initial Dictionary
Dict = { 5 : 'Welcome', 6 : 'To', 7 : 'Geeks',
       'A' : {1 : 'Geeks', 2 : 'For', 3 : 'Geeks'},
       'B' : {1 : 'Geeks', 2 : 'Life'}}
print("Initial Dictionary: ")
print(Dict)

# Deleting a Key value
del Dict[6]
print("\nDeleting a specific key: ")
print(Dict)

# Deleting a Key using pop()
Dict.pop(5)
print("\nPopping specific element: ")
print(Dict)

# Deleting an arbitrary Key-value pair using popitem()
Dict.popitem()
print("\nPops an arbitrary key-value pair: ")
print(Dict)

# Deleting entire Dictionary
Dict.clear()
print("\nDeleting Entire Dictionary: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}

Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}

Dictionary after adding 3 elements:
{1: 'Geeks', 2: 'For', 0: 'Geeks', 3: 1}

Updated key value:
{1: 'Geeks', 2: 'Welcome', 0: 'Geeks', 3: 1}
Accessing a element using key:
For
Accessing a element using get:
Geeks
Initial Dictionary:
{5: 'Welcome', 6: 'To', 7: 'Geeks', 'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2:
'Life'}}

Deleting a specific key:
{5: 'Welcome', 7: 'Geeks', 'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}}

Popping specific element:
{7: 'Geeks', 'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}}

Pops an arbitrary key-value pair:
{7: 'Geeks', 'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}}

Deleting Entire Dictionary:
{}
```

References:

https://www.slideshare.net/rj143/python-by-rj

UNIT I – Question Bank

PART -A

1. Is Python a compiler or an interpreter?
2. Write the features of python.
3. List few Applications of Python.
4. Compare and Contrast: Syntax Error, Sematic Error.
5. Explain Run time Errors.
6. Find out the use of: type (10).
7. Find the output of:
   a. type("17")
   b. type("BSC Students")
8. Differentiate: Script mode and Interactive mode.
9. Give examples of Mutable and Immutable objects in Python.
10. Indentation gains much importance in Python. State True or False.
11. List out the various data types used in python.
12. Write few keywords employed in python.
13. Compare and contrast: Mutable versus Immutable.
14. What is a Boolean value?
15. Differentiate: Tuples and Lists in Python.
16. Create a dictionary with 4 Key-Value Pairs.
17. Mention few methods that are employed over Python Lists.
18. Comment on indexing and negative indexing used in List.
19. What are the built-in functions that are used in Tuple?
20. State the purpose of pop() method.

PART -B

1. Explain the Structure of a Python Program with an example.
2. List the Elements of Python and give a detailed note on it.
3. Write a menu-driven program, using user-defined functions to find the area of rectangle, square, circle and triangle by accepting suitable input parameters from user.
4. What is the difference between lists, tuples and dictionaries? Give an example for their usage.
5. What are the basic list operations that can be performed in Python?
6. Write a menu driven program to convert the given temperature from Fahrenheit to Celsius and vice versa depending upon user's choice.
7. Create a Simple Calculator using a python code.
8. How to create a dictionary? What are the methods employed over it.
9. Read the Student Name, ID, Department and marks of five subjects and find the percentage.
10. a. Write a python code to read an integer, float, complex values and display their data type.
    b. Find the Simple Interest and Compound Interest using a python code.

I. PYTHON INTERPRETER

FEATURES OF PYTHON INTERPRETER

Python interpreter offers some pretty cool features:

- Interactive editing
- History substitution
- Code completion on systems with support for readline

INVOKING THE PYTHON INTERPRETER

On your machine, you can find your interpreter at an address like:
C:\Python36
Or it may reside on the location you selected at the time of installation. Add path using this command:

```
set path=%path%;C:\python36
```

INTERACTIVE MODE

Python interpreter is in an interactive mode when it reads commands from a tty. The primary prompt is the following:

1. >>>

When it shows this prompt, it means it prompts the developer for the next command. This is the REPL. Before it prints the first prompt, Python interpreter prints a welcome message that also states its version number and a copyright notice.
This is the secondary prompt:

1. …

This prompt denotes continuation lines.

```
$ python3.7
Python 3.7 (default, Jul 16 2018, 04:38:07)
[GCC 4.8.2] on Windows
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You will find continuation lines when working with a multi-line construct:

```
>>> it_rains =True
>>> if it_rains:
>>> print("The produce will be good")
```

Output:

The produce will be good

We can also use the Python interpreter as a calculator:

```
>>> 2*7
14
>>> 4/2
2.0
```

HOW DOES PYTHON INTERPRETER WORKS?

Four things happen in a REPL:

i. Lexing- The lexer breaks the line of code into tokens.
ii. Parsing- The parser uses these tokens to generate a structure, here, an Abstract Syntax Tree, to depict the relationship between these tokens.
iii. Compiling- The compiler turns this AST into code object(s).
iv. Interpreting- The interpreter executes each code object.

USING PYTHON AS A CALCULATOR

Start with simple Python commands. Start the interpreter and wait for the primary prompt, >>>.

Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
```

```
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

With Python, it is possible to use the ** operator to calculate powers [1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2]. \ can be used to escape quotes:

```
>>>
```

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a newline
First line.
Second line.
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a newlist
[9, 16, 25]
```

All slice operations return a new list containing the requested elements.

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

```
#Program 1:

>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

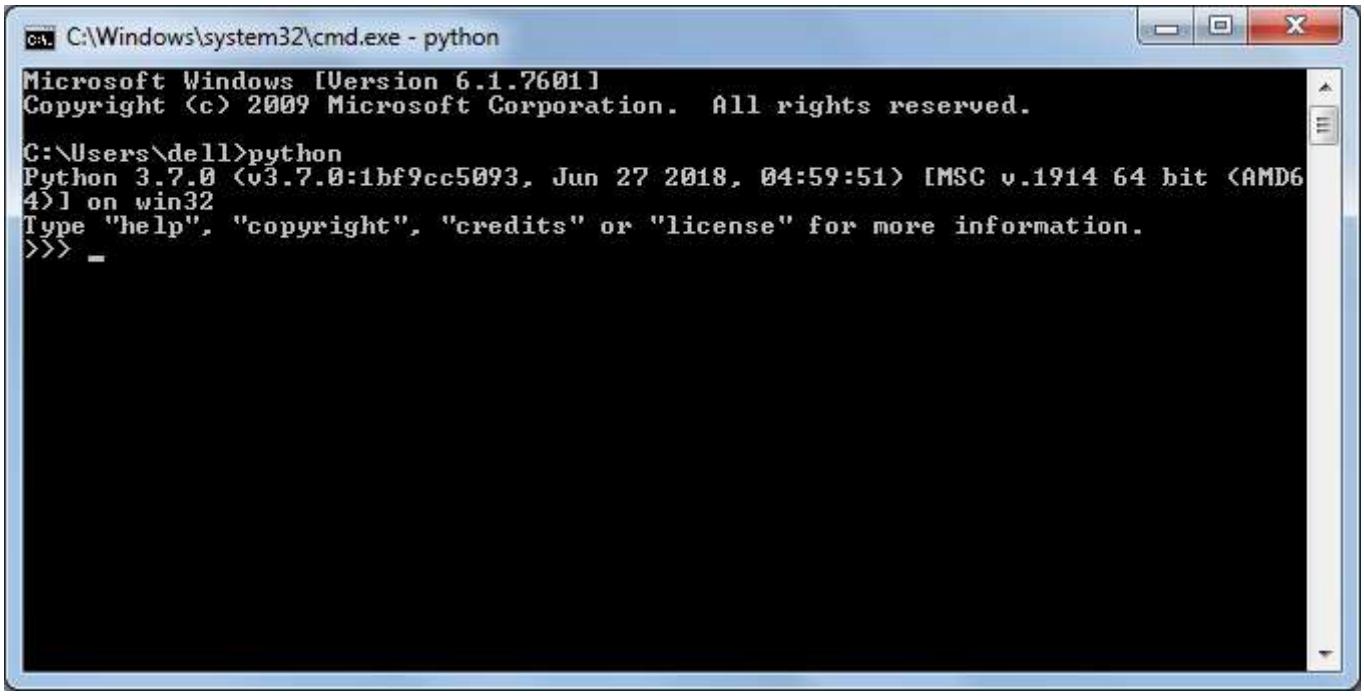
```
#Program 2:

>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

PYTHON - SHELL (INTERPRETER)

Python is an interpreter language. It means it executes the code line by line. Python provides a Python Shell (also known as Python Interactive Shell) which is used to execute a single Python command and get the result.

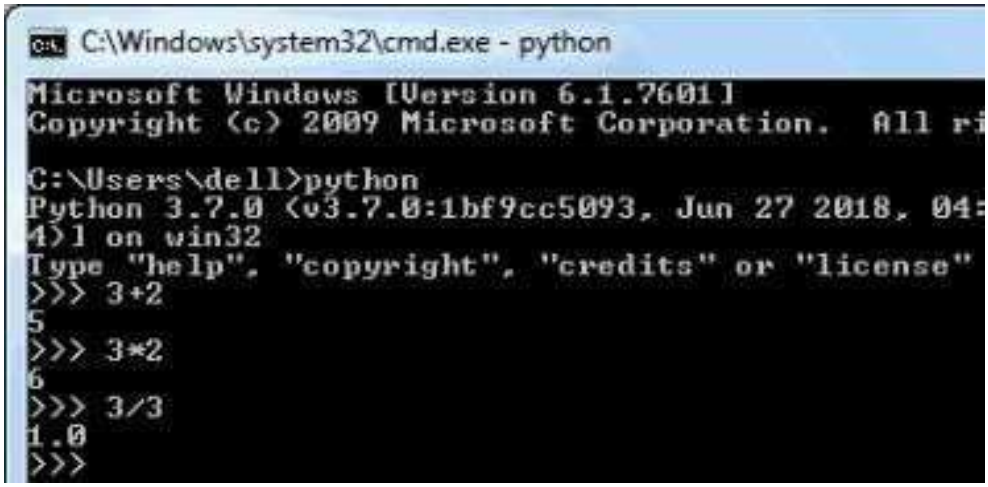
Python Shell waits for the input command from the user. As soon as the user enters the command, it executes it and displays the result.

To open the Python Shell on Windows, open the command prompt, write python and press enter.



Python Shell

As you can see, a Python Prompt comprising of three Greater Than symbols (>>>) appears. Now, you can enter a single statement and get the result. For example, enter a simple expression like 3 + 2, press enter and it will display the result in the next line.



Command Execution on Python Shell

Execute Python Script

Python Shell executes a single statement. To execute multiple statements, create a Python file with extension .py, and write Python scripts (multiple statements).

For example, enter the following statement in a text editor such as Notepad.

Example: myPythonScript.py

```
print ("This is Python Script.")
print ("Welcome to Python Tutorial by TutorialsTeacher.com")
```

Save it as myPythonScript.py, navigate command prompt to the folder where you have saved this file and execute the python myPythonScript.py command, as shown below.

Python Shell

Thus, you can execute Python expressions and commands using Python Shell.

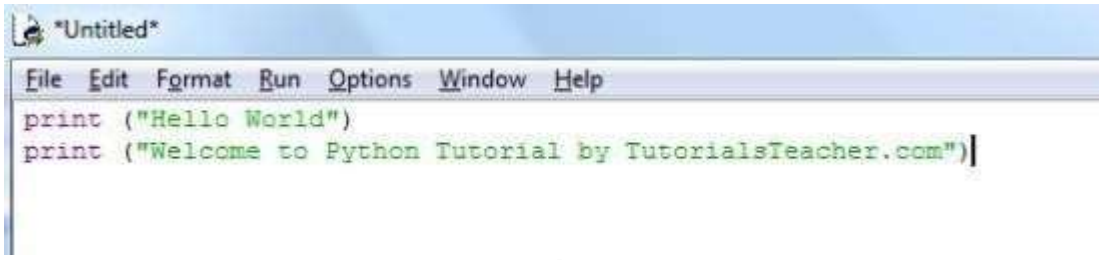
PYTHON – IDLE

IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

IDLE can be used to execute a single statement just like Python Shell and also to create, modify and execute Python scripts. IDLE provides a fully-featured text editor to create Python scripts that includes

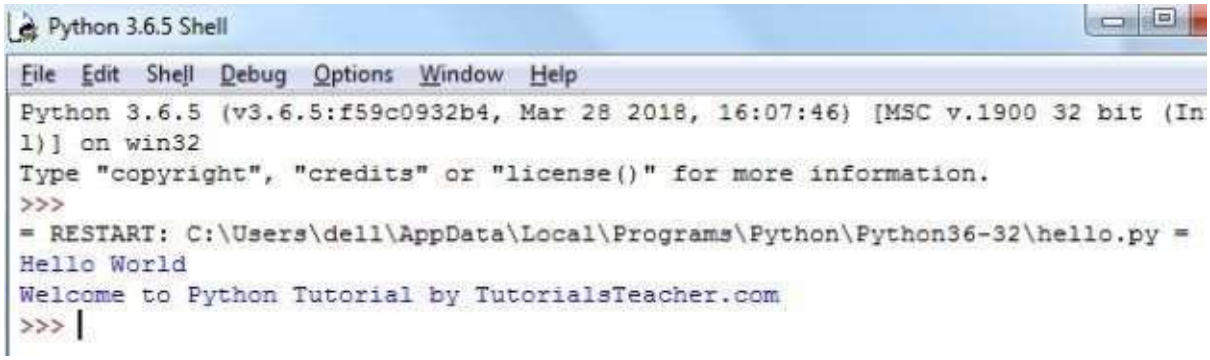
features like syntax highlighting, autocompletion and smart indent. It also has a debugger with stepping and breakpoints features.

Goto File->New File and open a new Script page and enter multiple statements and then save the file with extension .py using File -> Save. For example, save the following code as hello.py.



Python Script in IDLE

Now, press F5 to run the script in the editor window. The IDLE shell will show the output.



Python Script Execution Result in IDLE

Thus, it is easy to write, test and run Python scripts in IDLE.

PYTHON INDENTATION

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent.

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing. But the former style is clearer.
Incorrect indentation will result into **IndentationError.**

ATOMS

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom:      identifier | literal | enclosure
enclosure: parenth_form|list_display|dict_display|string_conversion
```

PYTHON IDENTIFIERS

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore _. Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
    SyntaxError: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, % etc. in the identifier.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
  SyntaxError: invalid syntax
```

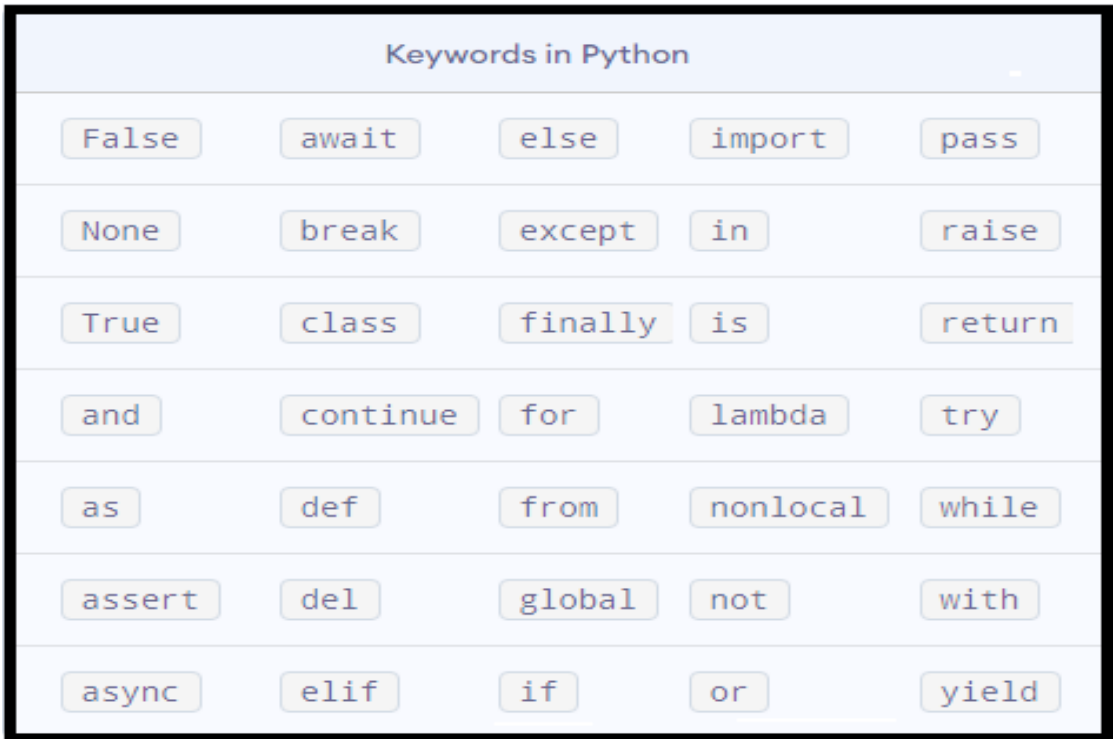
5. Identifier can be of any length.

PYTHON KEYWORDS

Keywords are the reserved words in Python.

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. In Python, keywords are case sensitive. There are 33 keywords in Python 3.7. This number can vary slightly in the course of time.

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



LITERALS

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

Example 1:

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal

#Float    Literal
float_1  =  10.5
float_2 = 1.5e2

#Complex Literal
x = 3.14j
```

STRINGS

In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned.

Program:

```
String1 = "IIIBSC STUDENTS"
print("Initial String: ")
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])

print("\n8th char: ")
print(String1[8])
#  Updation or deletion of characters from a String is not allowed
```

Output:

```
Initial String:
IIIBSC STUDENTS

First character of String is:
I

Last character of String is:
S

8th char:
T
```

II. OPERATORS IN PYTHON

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Operators are used to perform operations on variables and values.

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Special operators
  - ✓ Identity operators
  - ✓ Membership operators

1. **Arithmetic operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

OPERATOR	DESCRIPTION	SYNTAX
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y

Program:

```
# Examples of Arithmetic Operator
a = 9
b = 4

# Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
```

```
# Modulo of both number
mod = a % b

# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
```

Output:

```
13
5
36
2.25
2
1
```

2. **Relational Operators:** Relational operators compares the values. It either returns **True** or **False** according to the condition.

OPERATOR	DESCRIPTION	SYNTAX
>	Greater than: True if left operand is greater than the right	x > y
<	Less than: True if left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to: True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to: True if left operand is less than or equal to the right	x <= y

Program:

```
# Examples of Relational Operators
a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
```

```
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
print(a <= b)
```

Output:

```
False
True
False
True
False
True
```

3. **Logical operators:** Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

Program:

```
# Examples of Logical Operator
a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output:

```
False
True
```

False

4. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	DESCRIPTION	SYNTAX
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Program:

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c

c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c

c = a >> 2;       # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

Output:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

5. **Assignment operators:** Assignment operators are used to assign values to the variables.

OPERATOR	DESCRIPTION	SYNTAX
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

6. **Special operators:** There are some special type of operators like-
   - **Identity operators-**
     is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

```
is          True if the operands are identical
is not      True if the operands are not identical
```

Program:

```
# Examples of Identity operators
a1 = 3
b1 = 3
a2 = 'GeeksforGeeks'
b2 = 'GeeksforGeeks'
a3 = [1,2,3]
b3 = [1,2,3]

print(a1 is not b1)

print(a2 is b2)

# Output is False, since lists are mutable.
print(a3 is b3)
```

Output:

```
False
True
False
```

- **Membership operators-**

in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

in	True if value is found in the sequence
not in	True if value is not found in the sequence

Program:

```
# Examples of Membership operator
x = 'Geeks for Geeks'
y = {3:'a',4:'b'}

print('G' in x)

print('geeks' not in x)

print('Geeks' not in x)

print(3 in y)

print('b' in y)

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

```
print("banana" in x)

# returns True because a sequence with the value "banana" is in the list
print("pineapple" not in x)

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

Output:

```
True
True
False
True
False
True
True
```

Ternary operator

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5. It simply allows to test a condition in a **single line** replacing the multiline if-else making the code compact.

Syntax :

```
[on_true] if [expression] else [on_false]
```

Example 1: Simple Method to use ternary operator

```
# Program to demonstrate conditional operator
a, b = 10, 20

# Copy value of a in min if a < b else copy b
min = a if a < b else b

print(min)
```

Output:

```
10
```

Example 2: Python 3 program to find the factorial of given number

```
def factorial(n):
    # single line to find factorial
    return 1 if (n==1 or n==0) else n * factorial(n - 1)

# Driver Code
num = 5
print ("Factorial of",num,"is", factorial(num))
```

Output:

```
Factorial of 5 is 120
```

Increment and Decrement Operators in Python

Python is designed to be consistent and readable. One common error by a novice programmer in languages with ++ and -- operators is mixing up the differences (both in precedence and in return value) between pre and post increment/decrement operators. Simple increment and decrement operators aren't needed as much as in other languages.

```
for (int i = 0; i < 5; ++i)
```

In Python, instead we write it like,

```
# A Sample Python program to show loop (unlike many # other languages, it doesn't use ++)

for i in range(0, 5):
  print(i)
```

Output:

```
0
1
2
3
4
```

We can almost always avoid use of ++ and --. For example, **x++** can be written as **x += 1** and **x--** can be written as **x -= 1**.

UNIT II – Question Bank

PART -A

1. What operators does python support?
2. Mention the features of identity operators?
3. Give the characteristics of membership operator?
4. What is use of len() function.
5. Describe the function of **is** and **is not** operators.
6. List the relational operators used in python.
7. How Logical operators differs from Bitwise operators?
8. Explain about string slicing with examples.
9. How to split strings and what function is used to perform that operation?
10. Given A=60. Find A>>2.

PART -B

1. Give an example of how Python can be used as a calculator.
2. How Python Operates over Strings?
3. a. Explain string slicing with examples.
   b. How to split strings. Explain with examples.
4. List out the various operators used in python and explain them.
5. Give a detailed note on:
   a. Identity operators
   b. Membership operators
   c. Boolean operators
6. Given A=60, B=13. Using A and B, explain all the bitwise operators used in python.

I. INPUT AND OUTPUT STATEMENTS

Input Statement:

Input means the data entered by the user of the program. In python, we have input() and raw_input ( ) function available for Input.

1) input() function

Syntax:

input (expression)

If prompt is present, it is displayed on monitor, after which the user can provide data from keyboard. Input takes whatever is typed from the keyboard and evaluates it. As the input provided is evaluated, it expects valid python expression. If the input provided is not correct then either syntax error or exception is raised by python.

Example 1:

```
# python input operations

# user input
x = input("Enter any value: ")

# printing value
print("Entered value is: ", x)
```

Output

```
RUN 1:
Enter any value: 12345
Entered value is:  12345

RUN 2:
Enter any value: IncludeHelp
Entered value is:  IncludeHelp

RUN 3:
Enter any value: Python is a progamming language.
Entered value is:  Python is a progamming language.
```

Example 2:

```
# python input operations

# just provide a value and entered value prints
print(input())

# provide another value
x = input()
print("Your input is: ", x)
```

```
# prompting message for input
val1 = input("Enter a value: ")
val2 = input("Enter another value: ")
val3 = input("Enter another value: ")

# printing values
print("val1 =", val1)
print("val2 =", val2)
print("val3 =", val3)
```

Output

```
Hello
Hello
I'm Shivang!
Your input is:  I'm Shivang!
Enter a value: 100
Enter another value: 23.45
Enter another value: Hellloooooooo
val1 = 100
val2 = 23.45
val3 = Hellloooooooo
```

2) raw_input() function

This input method fairly works in older versions (like 2.x).

Syntax:

```
raw_input (expression)
```

If prompt is present, it is displayed on the monitor after which user can provide the data from keyboard. The function takes exactly what is typed from keyboard, convert it to string and then return it to the variable on LHS of '='.

Example: In interactive mode

```
>>>x=raw_input ('Enter your name: ')

Enter your name: ABC
```

x is a variable which will get the string (ABC), typed by user during the execution of program. Typing of data for the raw_input function is terminated by enter key.

We can use raw_input() to enter numeric data also. In that case we typecast, i.e., change the data type using function, the string data accepted from user to appropriate Numeric type.

Example:

```
>>>y=int(raw_input("Enter your roll no."))

Enter your roll no. 5
```

It will convert the accepted string i.e., 5 to integer before assigning it to 'y'.

Output Statement:

1). print() function/statement

print evaluates the expression before printing it on the monitor. Print statement outputs an entire (complete) line and then goes to next line for subsequent output (s). To print more than one item on a single line, comma (,) may be used.

Syntax:

```
print (expression/constant/variable)
```

Example 1:

```
# print() example in Python

# using single quotes
print('Hello!')
print('How are you?')

# using double quotes
print("Hello!")
print("How are you?")

# using triple single quotes
# those can be used to print multiple line string
print('''Hello!''')
print('''How are you?''')

# printing multiline string
print('''Hello... how are you?
Hey! I am good, what about you?
I am good also, thanks.''')
```

Output

```
Hello!
How are you?
Hello!
How are you?
Hello!
How are you?
```

```
Hello... how are you?
Hey! I am good, what about you?
I am good also, thanks.
```

Example 2:

```
# print() example in Python

# printing values
print("Printing direct values...")
print(10) # printing an integer
print(10.2345) # printing a float
print([10, 20, 30, 40, 50]) # printing a list
print({10, 20, 30, 40, 50}) # printing a set

# printing variables
a = 10
b = 10.2345
c = [10, 20, 30, 40, 50]
d = {10, 20, 30, 40, 50}

print("Printing variables...")
print(a)
print(b)
print(c)
print(d)

# printing message with variables
print("Printing message variables...")
print("a = ", a)
print("b = ", b)
print("c = ", c)
print("d = ", d)
```

Output

```
Printing direct values...
10
10.2345
[10, 20, 30, 40, 50]
{40, 10, 50, 20, 30}
Printing variables...
10
10.2345
[10, 20, 30, 40, 50]
{40, 10, 50, 20, 30}
Printing message variables...
a =  10
b =  10.2345
c =  [10, 20, 30, 40, 50]
d =  {40, 10, 50, 20, 30}
```

II. CONTROL STATEMENTS

(BRANCHING, LOOPING, CONDITIONAL STATEMENT, EXIT FUNCTION, DIFFERENCE BETWEEN BREAK, CONTINUE AND PASS)

The if statement

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if condition:
  # Statements to execute if
  # condition is true
```

Ex 1:

```
x = eval(input("Enter x: "))

if x>0:
   print("x is positive")
```

****

The if else statement with multiple statements

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
   # Executes this block if
   # condition is true
else:
   # Executes this block if
   # condition is false
```

Ex 1:

```
x = 'spam'
if x == 'spammy':
```

```
    print 'Hi spam\n'
    print "Nice weather we're having"
    print 'Have a nice day!'
else:
    print 'not spam'
    print 'Not having a good day?'
```

A nested if example
(an if statement within another if statement)

nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```

Ex 1:

```
score=raw_input("Enter score: ")
score=int(score)
if score>=80:
    grade='A'
else:
    if score>=70:
        grade='B'
    else:
        grade='C'
print "Grade is:" +grade
```

A nested if example - using if/else

Ex 1:

```
score = raw_input("Enter score: ")
score = int(score)
if score >= 80:
    grade = 'A'
else:
    if score >= 70:
        grade = 'B'
    else:
        if score >= 55:
            grade = 'C'
        else:
            if score >= 50:
```

```
            grade = 'Pass'
        else:
            grade = 'Fail'
print "\n\nGrade is: " + grade
```

A nested if example - using if/elif/else

if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

Ex 1:

```
score = raw_input("Enter score: ")
score = int(score)
if score >= 80:
    grade = 'A'
elif score >= 70:
    grade = 'B'
elif score >= 55:
    grade = 'C'
elif score >= 50:
    grade = 'Pass'
else:
    grade = 'Fail'
print "\n\nGrade is: " + grade
```

Examples of while loops

While Loop

Syntax :

```
while expression:
```

```
    statement(s)
```

Example:

```
x = 1
while x <= 5:
    print ('Hi spam')
    x = x + 1
    print ('I love spam')
print ('done')
print ('gone')
```

Output:

```
Hi spam
I love spam
Hi spam
I love spam
Hi spam
I love spam
Hi spam
I love spam
Hi spam
I love spam
done
gone
```

Examples : Using else Statement with Loops(while)

Python supports to have an else statement associated with a loop statement.

- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

Ex:

```
count = 0
while count < 5:
   print (count, " is less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

Output:

```
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is less than 5
6 is not less than 5
```

Examples of while loops - the infinite loop

```
x = 1
while x:
    print 'Hi spam'
    x = x + 1
    print 'I love spam'
    print 'Press the Ctrl key and the C key together'
    print 'to interupt this program...'
print 'done'
print 'gone'
```

Example: use of break to end an infinite loop

```
while 1:
    print 'Spam'
    answer = raw_input('Press y to end this loop')
    if answer == 'y':
        print 'Fries with that?'
        break
print 'Have a '
print 'nice day!'
```

Example: use of continue in a loop

```
while 1:
    print 'Spam'
    answer = raw_input('Press y for large fries ')
    if answer == 'y':
        print 'Large fries with spam, mmmm, yummy '
        continue
    answer = raw_input('Had enough yet? ')
    if answer == 'y':
        break
print 'Have a '
print 'nice day!'
```

Example: the counter-controlled for loop

For Loop

Syntax:

```
for iterator_var in sequence:
    statements(s)
```

Example:

```
for c in range (10):
    print c


# Note: range (10) is 0 through 9
```

Example: the counter-controlled for loop

```
for c in range (5,10):
    print c


# Note: range (5,10) is 5 through 9
```

Output:

```
5
6
7
8
9
```

Example: how to use a for loop in computing

```
n = [6,4,5,7,8,6,2,3,1]
s = 0
for val in n:
    s=s+val
print(s)
```

Output:

```
42
```

Example: For loop with strings

```
for letter in 'Python':     # First Example
   print ('Current Letter :', letter)

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print ('Current fruit :', fruit)

print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Example: 'break' with the for loop

```
for c in range (1,6):
    if c == 3:
        break
    print c
```

Output:

```
1
2
```

Example: how to use a loop within a loop a nested for loop

```
print ("This is the start of the program")
for i in range (1,3):
    for j in range (1,3):
        for k in range (1,3):
            print ("i: " + str(i) + " j: " + str(j) + " k: " + str(k))
    print ("Hi")
```

Output:

```
This is the start of the program
i: 1 j: 1 k: 1
i: 1 j: 1 k: 2
i: 1 j: 2 k: 1
i: 1 j: 2 k: 2
i: 2 j: 1 k: 1
i: 2 j: 1 k: 2
i: 2 j: 2 k: 1
i: 2 j: 2 k: 2
```

EXIT() FUNCTION

exit()

exit() is defined in site.py and it works only if the site module is imported so it should be used in the interpreter only. It is like a synonym of quit() to make the Python more user-friendly. It too gives a message when printed:

Example:

```
# Python program to demonstrate
# exit()

for i in range(10):

    # If the value of i becomes
    # 5 then the program is forced
    # to exit
    if i == 5:

        # prints the exit message
        print(exit)
        exit()
    print(i)
```

Output:

```
0
1
2
3
4
Use exit() or Ctrl-D (i.e. EOF) to exit
```

DIFFERENCE BETWEEN BREAK, CONTINUE AND PASS

Break statement

The break statement is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break statement, if available. If the break statement is present in the nested loop, then it terminates only those loops which contains break statement.

Syntax:

```
break
```

Continue statement

This statement is used to skip over the execution part of the loop on a certain condition. After that, it transfers the control to the beginning of the loop. Basically, it skips its following statements and continues with the next iteration of the loop.

Syntax:

```
continue
```

Pass statement

As the name suggests pass statement simply does nothing. We use pass statement to write empty loops. Pass is also used for empty control statements, functions and classes.

Syntax:

```
pass
```

Example:

```
# Python program to demonstrate
# difference between pass and
# continue statements

s = "geeks"

# Pass statement
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)

print()

# Continue statement
for i in s:
```

```
    if i == 'k':
        print('Continue executed')
        continue
        print(i)
```

OUTPUT:

```
g
e
e
Pass executed
k
s

g
e
e
Continue executed
s
```

Break Statement : It brings control out of the loop

Pass Statement:
We use pass statement to write empty loops. Pass is also used for empty control statement, function and classes.

Continue statement: forces the loop to continue or execute the next iteration.

III. FUNCTIONS

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. Python gives many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defininga Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Bydefault, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it bycalling it fromanother function or directly from the Python prompt.

Ex 1: The following function takes a string as input parameter and prints it on standard screen.

```
# Function definition is here
def fun1(str):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call fun1 function
fun1("III CSE")
fun1("M SEction-python class")
```

Output:

```
III CSE
M SEction-python class
```

Ex 2: Arithmetical Operations using functions

```
def add(x,y):
   return (x+y)
def sub(x,y):
    return (x-y)
  def mul(x,y):
   return (x*y)
def div(x,y):
   return (x/y)

n1=40; n2=20
print("Sum=",add(n1,n2))
print("Sub=",sub(n1,n2))
print("Multiplication=",mul(n1,n2))
mylist = [10,20,30]
changeme(mylist)
print("Values outside the function: ",mylist)print("Division=",div(n1,n2))
```

Output:

```
Sum= 60
Sub= 20
```

```
Multiplication= 800
Division= 2.0
```

Function Arguments

You can call a function by using the following types of formal arguments −

- **Required arguments**
- **Keyword arguments**
- **Default arguments**
- **Variable-length arguments**

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *fun1()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows:

Ex 1:

```
# Function definition is here
def fun1(str):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call fun1() function
fun1()
fun1("M SEction-python class")
```

Output:

```
TypeError                 Traceback (most recent call last)
<ipython-input-72-671f9452cf1c> in <module>
      6
      7 # Now you can call printme function
----> 8 fun1()
      9 fun1("M SEction-python class")
     10
TypeError: fun1() missing 1 required positional argument: 'str'
```

Keywordarguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways:

Ex 1:

```
# Function definition is here
def fun1(str):
   "This prints a passed string into this function"
   print (str)
   return;
```

```
fun1(str="III CSE")
fun1(str="M SEction-python class")
```

Output:

```
III CSE
M SEction-python class
```

Ex 2: Note that the order of parameters does not matter.

```
# Function definition is here
def printinfo( name, pyt ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Python Subject% ", pyt)
   return;

# Now you can call printinfo function
printinfo( pyt=100, name="M Section" )
```

Output:

```
Name:  M Section
Python Subject%  100
```

Defaultarguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

Ex 1:

```
# Function definition is here
def printinfo( name, pyt = 95 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Python Subject%  ", pyt)
   return;

# Now you can call printinfo function
printinfo( pyt=100, name="M Section" )
#printinfo( pyt, name="M Section")  Error stmt --as keyword argument should compulsorily
have a value--pyt=99
printinfo( name="M Section" , pyt=80)
printinfo( name="M Section")
```

Output:

```
Name:  M Section
Python Subject%  100
Name:  M Section
Python Subject%  80
Name:  M Section
Python Subject%  95
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

Ex 1:

```
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print ("Output is: ")
   print (arg1)
   for var in vartuple:
      print (var)
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

AnonymousFunctions–LambdaFunctions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.... argn]]:expression
```

Following is the example to show how *lambda* form of function works –

Ex 1:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print ("Value of total: ", sum( 10, 20 ))
print ("Value of total: ", sum( 20, 20 ))
```

Output:

```
Value of total : 30
Value of total :  40
```

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example –

Ex 1:

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2  # Here total is local variable.
   print ("Inside the function local total : ", total)

# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total)
```

Output:

```
Inside the function local total :  30
Outside the function global total :  0
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

Ex 1:

```
# Function definition is here
def printinfo( name, pyt = 95 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Python Subject%  ", pyt)
   return;

# Now you can call printinfo function
printinfo( pyt=100, name="M Section" )
#printinfo( pyt, name="M Section")  Error stmt --as keyword argument should compulsorily have a value--pyt=99
printinfo( name="M Section" , pyt=80)
printinfo( name="M Section")
```

Output:

```
Name:  M Section
Python Subject%  100
Name:  M Section
Python Subject%  80
Name:  M Section
Python Subject%  95
```

IV. Python Errors and Exceptions Handling

Try Except

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Example 1:

```
# The try block will generate an exception, because z is not defined
# Since the try block raises an error, the except block will be executed.
# Without the try block, the program will crash and raise an error

try:
  print(z)
except:
  print("An exception occurred")
```

Output:

```
runfile('C:/Users/others/.spyder-py3/ex2.py',    wdir='C:/Users/others/.spyder-py3')
```

An exception occurred

Suppose, if we only execute the following statement, then,

Example 2:

```
print(z)
```

Output:

NameError: name 'z' is not defined

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example 3:

```
# Print one message if the try block raises a NameError and another for other errors:
try:
  print(z)
except NameError:
print("Variable z is not defined")
except:
  print("Something else went wrong")
```

Output:

Variable z is not defined

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example 4:

```
# Here, the try block does not generate any err:
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Output:

Hello
Nothing went wrong

Example 5:

```
# since a is not defined, it generates exception
try:
  print(a)
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Output:

Something went wrong

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not. This can be useful to close objects and clean up resources

Example 6:

```
try:
  print(z)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Output:

```
Something went wrong
The 'try except' is finished
```

Try to open and write to a file that is not writable:

Example 7: # The try block will raise an error when trying to write to a read-only file:

```
try:
  f = open("cse.txt")
  f.write("M SECTION STUDENTS")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

Output:

```
Something went wrong when writing to the file
```

Example 8: Divide by zero

```
def divide(x,y):
 try:
     res=x/y
 except ZeroDivisionError:
    print ("Exception occured")
 else:
    print(res)
 finally:
    print("Program is complete")
```

Output:

```
>>> divide(2,1)
2.0
Program is complete
>>> divide(2,0)
Exception occured
Program is complete
```

List of Standard Exceptions

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError**

	Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

UNIT III – Question Bank

PART -A

1. What is a function?
2. Mention the types of function arguments in python.
3. What is meant by conditional if statement.
4. What is the difference between break and continue statement?
5. What is python pass statement?
6. What is an exception? Explain with an example.
7. List some few common Exception types and explain when they occur.
8. Give an example of lambda function.
9. State the use of Exit function.
10. Find the difference between break, continue and pass.
11. Give an example of Input and Output Statements.
12. Give the syntax of if else statement.
13. State the use of range function.
14. Explain for loop with an example.
15. Write the syntax of while loop with example.
16. Does Python support the concept of Default arguments?
17. Write a simple program which illustrates Handling Exceptions.

PART -B

1. Narrate the types of Function arguments with examples.
2. Narrate the importance of Default arguments with a python code.
3. Give a detailed note on the branching and looping control statements.
4. Using a python code, display the first n terms of Fibonacci series.
5. Using a python code, find the factorial of the given number.
6. Write a program to print the sum of the following series 1 + 1/2 + 1/3 +. …. + 1/n.
7. How Exceptions are handled using python. Give Examples.
8. Calculate the total marks, percentage and grade of a student. Marks obtained in each of the three subjects are to be input by the user. Assign grades according to the following criteria :

   Grade A: Percentage >=80,
   Grade B: Percentage >=70 and <80
   Grade C: Percentage >=60 and <70
   Grade D: Percentage >=50 and <60
   Fail: Percentage<50

I. ITERATION AND RECURSION

Iteration

An iteration is a single pass through a group/set of instructions. Most programs often contain loops of instructions that are executed over and over again. The computer repeatedly executes the loop, iterating through the loop.

Iteration is the act of repeating a process, either to generate an unbounded sequence of outcomes, or with the aim of approaching a desired goal, target or result. Each repetition of the process is also called an "iteration", and the results of one iteration are used as the starting point for the next iteration.

Example:

```
a = 0
for i from 1 to 3      // loop three times
{
a = a + i              // add the current value of i to a
}
print a      // the number 6 is printed (0 + 1; 1 + 2; 3 + 3)
```

Recursive Functions

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step.

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.
Examples of such problems are Towers of Hanoi (TOH), In order/Preorder/Post order Tree Traversals, DFS of Graph, etc.

1. Program for finding the factorial of a given no

```
def fact(n):
if n==0:
return 1
else:
return n*fact(n-1)
print("The factorial of a given no is",fact(5))
```

Output:

```
The factorial of a given no is:120
```

2. Finding nth Fibonacci Number

```
def fib(n):
if n<3:
return 1
else:
return fib(n-1)+fib(n-2)
print("The nth fibbonacci no is",fib(10))
```

Output:
The nth Fibonacci series is 55.

Disadvantages of Recursion over iteration

Note that both recursive and iterative programs have same problem solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. Recursive program has greater space requirements than iterative program as all functions will remain in stack unt il base case is reached. It also has greater time requirements because of function calls and return overhead.

Advantages of Recursion over iteration

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems it is preferred to write recursive code. We can write such codes also iteratively with the help of stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.

CONDITIONAL EXECUTION

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the if statement:

```
if x > 0:
  print "x is positive"
```

The boolean expression after the if statement is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.

Like other compound statements, the if statement is made up of a header and a block of statements:

```
HEADER:
  FIRST STATEMENT
  ...
  LAST STATEMENT
```

The header begins on a new line and ends with a colon (:). The indented statements that follow are called a **block**. The first unindented statement marks the end of the block. A statement block inside a compound statement is called the **body** of the statement.

There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one. Occasionally, it is useful to have a body with no statements. In that case, you can use the pass statement, which does nothing.

ALTERNATIVE EXECUTION

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
  print x, "is even"
else:
  print x, "is odd"
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might "wrap" this code in a function:

```
def printParity(x):
  if x%2 == 0:
    print x, "is even"
  else:
    print x, "is odd"
```

For any value of x, printParity displays an appropriate message. When you call it, you can provide any integer expression as an argument.

```
>>> printParity(17)
17 is odd
>>> y = 17
>>> printParity(y+1)
18 is even
```

CHAINED CONDITIONALS

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
  print x, "is less than", y
elif x > y:
  print x, "is greater than", y
else:
  print x, "and", y, "are equal"
```

elif is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit of the number of elif statements, but the last branch has to be an else statement:

```
if choice == 'A':
  functionA()
elif choice == 'B':
  functionB()
elif choice == 'C':
  functionC()
```

```
else:
  print "Invalid choice."
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

NESTED CONDITIONALS

One conditional can also be nested within another. We could have written the trichotomy example as follows:

```
if x == y:
  print x, "and", y, "are equal"
else:
  if x < y:
    print x, "is less than", y
  else:
    print x, "is greater than", y
```

The outer conditional contains two branches. The first branch contains a simple output statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
  if x < 10:
    print "x is a positive single digit."
```

The print statement is executed only if we make it past both the conditionals, so we can use the and operator:

```
if 0 < x and x < 10:
  print "x is a positive single digit."
```

These kinds of conditions are common, so Python provides an alternative syntax that is similar to mathematical notation:

```
if 0 < x < 10:
  print "x is a positive single digit."
```

This condition is semantically the same as the compound boolean expression and the nested conditional.

THE return STATEMENT

The return statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
import math

def printLogarithm(x):
  if x <= 0:
    print "Positive numbers only, please."
    return

  result = math.log(x)
  print "The log of x is", result
```

The function printLogarithm has a parameter named x. The first thing it does is check whether x is less than or equal to 0, in which case it displays an error message and then uses return to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

II. RECURSION

We mentioned that it is legal for one function to call another, and you have seen several examples of that. We neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do. For example, look at the following function:

```
def countdown(n):
  if n == 0:
    print "Blastoff!"
  else:
    print n
    countdown(n-1)
```

countdown expects the parameter, n, to be a positive integer. If n is 0, it outputs the word, "Blastoff!" Otherwise, it outputs n and then calls a function named countdown itself passing n-1 as an argument.

What happens if we call this function like this:

```
>>> countdown(3)
```

The execution of countdown begins with n=3, and since n is not 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is not 0, it outputs the value 2, and then calls itself...
The execution of countdown begins with n=1, and since n is not 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with n=0, and since n is 0, it outputs the word, "Blastoff!" and then returns.

The countdown that got n=1 returns.

The countdown that got n=2 returns.

The countdown that got n=3 returns.

And then you're back in __main___(what a trip). So, the total output looks like this:

```
3
2
1
Blastoff!
```

As a second example, look again at the functions newLine and threeLines:

```
def newline():
  print

def threeLines():
  newLine()
  newLine()
  newLine()
```

Although these work, they would not be much help if we wanted to output 2 newlines, or 106. A better alternative would be this:

```
def nLines(n):
  if n > 0:
    print
    nLines(n-1)
```

This program is similar to countdown; as long as n is greater than 0, it outputs one newline and then calls itself to output n-1 additional newlines. Thus, the total number of newlines is 1 + (n - 1) which, if you do your algebra right, comes out to n.

The process of a function calling itself is **recursion**, and such functions are said to be recursive.

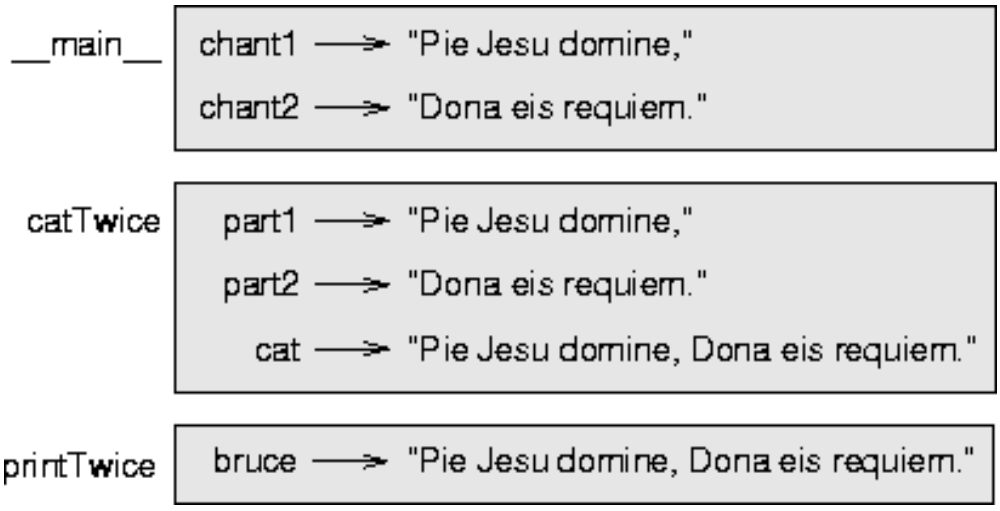
STACK DIAGRAM

Stack Diagram

A graphical representation of a stack of functions, their variables, and the values to which they refer.

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs.

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The order of the stack shows the flow of execution. printTwice was called by catTwice, and catTwice was called by _main_, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to _main_.

Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as chant1, part2 has the same value as chant2, and bruce has the same value as cat.

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to ___main___.

For example, if we try to access cat from within printTwice, we get a NameError:

```
Traceback (innermost last):
  File "test.py", line 13, in __main____
    catTwice(chant1, chant2)
  File "test.py", line 5, in catTwice
    printTwice(cat)
  File "test.py", line 9, in printTwice
    print cat
NameError: cat
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

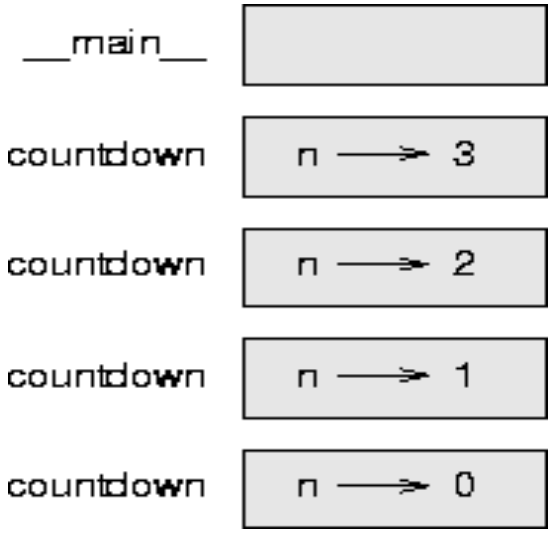
Notice the similarity between the traceback and the stack diagram. It's not a coincidence.

STACK DIAGRAMS FOR RECURSIVE FUNCTIONS

We can use a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

This figure shows a stack diagram for countdown called with n = 3:



As usual, the top of the stack is the frame for _main_. It is empty because we did not create any variables in __main__ or pass any arguments to it.

The four countdown frames have different values for the parameter n. The bottom of the stack, where n=0, is called the **base case**. It does not make a recursive call, so there are no more frames.

INFINITE RECURSION

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not considered a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
  recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  (98 repetitions omitted)
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 100 recurse frames on the stack!

ABBREVIATED ASSIGNMENT

Incrementing a variable is so common that Python provides an abbreviated syntax for it:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
```

`count += 1` is an abreviation for `count = count + 1` . We pronounce the operator as *"plus-equals"*. The increment value does not have to be 1:

```
>>> n = 2
>>> n += 5
>>> n
7
```

There are similar abbreviations for `-=`, `*=`, `/=`, `//=` and `%=`:

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
>>> n
6
>>> n //= 2
>>> n
3
>>> n %= 2
>>> n
1
```

III. THE while STATEMENT

Here is a fragment of code that demonstrates the use of the `while` statement:

```
def sum_to(n):
    """ Return the sum of 1+2+3 ... n """
    ss  = 0
    v = 1
    while v <= n:
        ss = ss + v
```

```
        v = v + 1
    return ss

# For your test suite
test(sum_to(4) == 10)
test(sum_to(1000) == 500500)
```

While `v` is less than or equal to `n`, continue executing the bodyof the loop. Within the body, each time, increment `v`. When `v` passes `n`, return your accumulated sum.

More formally, here is precise flow of execution for a `while` statement:

- Evaluate the condition at line 5, yielding a value which is either `False` or `True`.
- If the value is `False`, exit the `while` statement and continue execution at the next statement (line 8 in this case).
- If the value is `True`, execute each of the statements in the body (lines 6 and 7) and then go back to the `while` statement at line 5.

The body consists of all of the statements indented below the `while` keyword.

Notice that if the loop condition is `False` the first time we get loop, the statements in the body of the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “lather, rinse, repeat”, are an infinite loop.

In the case here, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `v` increments each time through the loop, so eventually it will have to exceed `n`. In other cases, it is not so easy, even impossible in some cases, to tell if the loop will ever terminate.

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop one has to manage the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates. By comparison, here is an equivalent function that uses `for` instead:

```
def sum_to(n):
    """ Return the sum of 1+2+3 ... n """
    ss  = 0
    for v in range(n+1):
        ss = ss + v
    return ss
```

IV. TABLES

One of the things loops are good for is generating tables. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, *"This is great! We can use the computers to generate the tables, so there will be no errors."* That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
for x in range(13):   # Generate numbers 0 to 12
    print(x, "\t", 2**x)
```

The string "\t" represents a **tab character**. The backslash character in "\t" indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a **newline**.

An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a print function, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

```
0       1
1       2
2       4
3       8
4       16
5       32
6       64
7       128
8       256
9       512
10      1024
11      2048
12      4096
```

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

TWO-DIMENSIONAL TABLES

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good wayto start is to write a loop that prints the multiples of 2, all on one line:

```
for i in range(1, 7):
    print(2 * i, end="  ")
print()
```

Here we've used the `range` function, but made it start its sequence at 1. As the loop executes, the value of `i` changes from 1 to 6. When all the elements of the range have been assigned to `i`, the loop terminates. Each time through the loop, it displays the value of `2 * i`, followed by three spaces.

Again, the extra `end="  "` argument in the `print` function suppresses the newline, and uses three spaces instead. After the loop completes, the call to `print` at line 3 finishes the current line, and starts a new line.

The output of the program is:

```
2     4     6     8     10    12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for. You have already seen some examples of encapsulation, including `is_divisible` in a previous chapter.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer.

This function encapsulates the previous loop and generalizes it to print multiples of `n`:

```
def print_multiples(n):
    for i in range(1, 7):
        print(n * i, end="  ")
    print()
```

To encapsulate, all we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter `n`.

If we call this function with the argument 2, we get the same output as before. With the argument 3, the output is:

```
3   6   9   12   15   18
```

With the argument 4, the output is:

```
4   8   12   16   20   24
```

By now you can probably guess how to print a multiplication table — by calling `print_multiples` repeatedly with different arguments. In fact, we can use another loop:

```
for i in range(1, 7):
    print_multiples(i)
```

Notice how similar this loop is to the one inside `print_multiples`. All we did was replace the `print` function with a function call.

The output of this program is a multiplication table:

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

More encapsulation

To demonstrate encapsulation again, let's take the code fromthe last section and wrap it up in a function:

```
def print_mult_table():
    for i in range(1, 7):
        print_multiples(i)
```

This process is a common **development plan**. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function.

This development plan is particularly useful if you don't know how to divide the program into functions when you start writing. This approach lets you design as you go along.

Python Program to print the following multiplication table.

Program:

```
def print_multiples(n):
    for i in range(1, 7):
        print(n * i, end="    ")
    print()



for i in range(1, 8):
    print_multiples(i)
```

Output:

```
1    2    3    4    5    6
2    4    6    8    10    12
3    6    9    12    15    18
4    8    12    16    20    24
5    10    15    20    25    30
6    12    18    24    30    36
7    14    21    28    35    42
```

References:

http://openbookproject.net/thinkcs/python/english3e/iteration.html
https://www.greenteapress.com/thinkpython/thinkCSpy/html/chap04.html
https://www.greenteapress.com/thinkpython/thinkCSpy/html/chap03.html#11
http://openbookproject.net/thinkcs/python/english3e/strings.html#cleaning-up-your-strings

UNIT IV – Question Bank

PART -A

1. Compare and Contrast: Iteration and Recursion.
2. Define Recursion.
3. State the purpose of a Stack Diagram.
4. Give an example of Conditional Execution.
5. Give an example of Alternative Execution.
6. Mention the disadvantages of Recursion over iteration.
7. Mention the advantages of Recursion over iteration.
8. Write an example of Nested Conditionals.
9. Print a 5*5 table.
10. Write a python program to print the multiplication table of 6.
11. Give few examples of Abbreviated Assignments.
12. Create a two dimensional table.

PART -B

1. Write a python program to create a Two Dimensional Table.
2. Find the factorial of a number using recursion.
3. Print the following multiplication table using a Python code.

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

4. Implement the concept of chained conditionals using a Python Program.
5. Illustrate Conditional Execution and Alternative Execution using a Python code.
6. Implement the concept of chained conditionals using a Python Program.
7. Illustrate the recursive function using a stack diagram.

I. STRINGS

A COMPOUND DATA TYPE

So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists and pairs. Strings, lists, and pairs are qualitatively different from the others because they are made up of smaller pieces. In the case of strings, they're made up of smaller strings each containing one **character**.

Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

WORKING WITH STRINGS AS SINGLE THINGS

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we could set the turtle's color, and we wrote `tess.turn(90)`.

Just like a turtle, a string is also an object. So each string instance has its own attributes and methods.

For example:

```
>>> ss = "Hello, World!"
>>> tt = ss.upper()
>>> tt
'HELLO, WORLD!'
```

`upper` is a method that can be invoked on any string object to create a new string, in which all the characters are in uppercase. (The original string `ss` remains unchanged.)

WORKING WITH THE PARTS OF A STRING

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
>>> fruit = "banana"
>>> m = fruit[1]
>>> print(m)
```

The expression `fruit[1]` selects character number 1 from `fruit`, and creates a new string containing just this one character. The variable `m` refers to the result. When we display `m`,

```
a
```

The letter at subscript position zero of `"banana"` is `b`. So at position `[1]` we have the letter `a`.

If we want to access the zero-eth letter of a string, we just place 0, or anyexpression that evaluates to 0, in between the brackets:

```
>>> m = fruit[0]
>>> print(m)
b
```

The expression in brackets is called an **index**. An index specifies a member of an ordered collection, in this case the collection of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

We can use `enumerate` to visualize the indices:

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

The same indexing notation works to extract elements from a list:

```
>>> prime_nums = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
>>> prime_nums[4]
11
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[3]
'Angelina'
```

LENGTH

The `len` function, when applied to a string, returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
sz = len(fruit)
last = fruit[sz]      # ERROR!
```

It causes the runtime error `IndexError: string index out of range`. The reason is that there is no character at index position 6 in `"banana"`. Because we start counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length of `fruit`:

```
sz = len(fruit)
last = fruit[sz-1]
```

Alternatively, we can use **negative indices**, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

TRAVERSAL AND THE FOR LOOP

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to encode a traversal is with a `while` statement:

```
ix = 0
while ix < len(fruit):
    letter = fruit[ix]
    print(letter)
    ix += 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `ix < len(fruit)`, so when `ix` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

But we've previously seen how the `for` loop can easily iterate over the elements in a list and it can do so for strings as well:

```
for c in fruit:
    print(c)
```

Each time through the loop, the next character in the string is assigned to the variable `c`. The loop continues until no characters are left. Here we can see the expressive power the `for` loop gives us compared to the while loop when traversing a string.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for p in prefixes:
    print(p + suffix)
```

The output of this program is:

```
Jack
Kack
Lack
```

```
Mack
Nack
Oack
Pack
Qack
```

SLICES

A *substring* of a string is obtained by taking a **slice**. Similarly, we can slice a list to refer to some sublist of the items in the list:

```
>>> s = "Pirates of the Caribbean"
>>> print(s[0:7])
Pirates
>>> print(s[11:14])
the
>>> print(s[15:24])
Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```

The operator `[n:m]` returns the part of the string from the n'th character to the m'th character, including the first but excluding the last. This behavior makes sense if you imagine the indices pointing *between* the characters, as in the following diagram:

If you imagine this as a piece of paper, the slice operator `[n:m]` copies out the part of the paper between the `n` and `m` positions. Provided `m` and `n` are both within the bounds of the string, your result will be of length (m-n).

Three tricks are added to this: if you omit the first index (before the colon), the slice starts at the beginning of the string (or list). If you omit the second index, the slice extends to the end of the string (or list). Similarly, if you provide value for `n` that is bigger than the length of the string (or list), the slice will take all the values up to the end. (It won't give an "out of range" error like the normal indexing operation does.) Thus:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
>>> fruit[3:999]
'ana'
```

What do you think `s[:]` means? What about `friends[4:]`?

STRING COMPARISON

The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
    print("Yes, we have no bananas!")
```

Other comparison operations are useful for putting words in *lexicographical* order:

```
if word < "banana":
    print("Your word, " + word + ", comes before banana.")
elif word > "banana":
    print("Your word, " + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")
```

This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

STRINGS ARE IMMUTABLE

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"
greeting[0] = 'J'           # ERROR!
print(greeting)
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object does not support item assignment`.

Strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
new_greeting = "J" + greeting[1:]
```

```
print(new_greeting)
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

THE in and not in OPERATORS

The `in` operator tests for membership. When both of the arguments to `in` are strings, `in` checks whether the left argument is a substring of the right argument.

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

Notethat a string is a substring of itself, and the emptystring is a substring of anyother string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```
>>> "a" in "a"
True
>>> "apple" in "apple"
True
>>> "" in "a"
True
>>> "" in "apple"
True
```

The `not in` operator returns the logical opposite results of `in`:

```
>>> "x" not in "apple"
True
```

Combining the `in` operator with string concatenation using `+`, we can write a function that removes all the vowels from a string:

```
1  def remove_vowels(s):
2      vowels = "aeiouAEIOU"
3      s_sans_vowels = ""
4      for x in s:
5          if x not in vowels:
6              s_sans_vowels += x
7      return s_sans_vowels
8
9  test(remove_vowels("compsci") == "cmpsc")
```

```
10 test(remove_vowels("aAbEefIijOopUus") == "bfjps")
```

A FIND FUNCTION

What does the following function do?

```
1  def find(strng, ch):
2      """
3        Find and return the index of ch in strng.
4      Return -1 if ch does not occur in strng.
5      """
6      ix = 0
7      while ix < len(strng):
8          if strng[ix] == ch:
9              return ix
10         ix += 1
11     return -1
12
13 test(find("Compsci", "p") == 3)
14 test(find("Compsci", "C") == 0)
15 test(find("Compsci", "i") == 6)
16 test(find("Compsci", "x") == -1)
```

In a sense, `find` is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

This is another example where we see a `return` statement inside a loop. If `strng[ix] == ch`, the function returns immediately, breaking out of the loop prematurely.

If the character doesn't appear in the string, then the program exits the loop normally and returns `-1`.

This pattern of computation is sometimes called a **eureka traversal** or **short-circuit evaluation**, because as soon as we find what we are looking for, we can cry "Eureka!", take the short-circuit, and stop looking.

LOOPING AND COUNTING

The following program counts the number of times the letter `a` appears in a string.

```
def count_a(text):
    count = 0
    for c in text:
        if c == "a":
            count += 1
    return(count)
```

```
test(count_a("banana") == 3)
```

OPTIONAL PARAMETERS

To find the locations of the second or third occurrence of a character in a string, we can modify the find function, adding a third parameter for the starting position in the search string:

```
def find2(strng, ch, start):
    ix = start
    while ix < len(strng):
        if strng[ix] == ch:
            return ix
        ix += 1
    return -1

test(find2("banana", "a", 2) == 3)
```

The call find2("banana", "a", 2) now returns 3, the index of the first occurrence of "a" in "banana" starting the search at index 2. What does find2("banana", "n", 3) return? If you said, 4, there is a good chance you understand how find2 works.

Better still, we can combine find and find2 using an **optional parameter**:

```
def find(strng, ch, start=0):
    ix = start
    while ix < len(strng):
        if strng[ix] == ch:
            return ix
        ix += 1
    return -1
```

When a function has an optional parameter, the caller *may* provide a matching argument. If the third argument is provided to find, it gets assigned to start. But if the caller leaves the argument out, then start is given a default value indicated by the assignment start=0 in the function definition.

So the call find("banana", "a", 2) to this version of find behaves just like find2, while in the call find("banana", "a"), start will be set to the **default value** of 0.

Adding another optional parameter to find makes it search from a starting position, up to but not including the end position:

```
def find(strng, ch, start=0, end=None):
    ix = start
    if end is None:
        end = len(strng)
    while ix < end:
        if strng[ix] == ch:
            return ix
```

```
        ix += 1
    return -1
```

The optional value for `end` is interesting: we give it a default value `None` if the caller does not supply any argument. In the body of the function we test what `end` is, and if the caller did not supply any argument, we reassign `end` to be the length of the string. If the caller has supplied an argument for `end`, however, the caller's value will be used in the loop.

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

Here are some test cases that should pass:

```
ss = "Python strings have some interesting methods."
test(find(ss, "s") == 7)
test(find(ss, "s", 7) == 7)
test(find(ss, "s", 8) == 13)
test(find(ss, "s", 8, 13) == -1)
test(find(ss, ".") == len(ss)-1)
```

THE BUILT-IN FIND METHOD

Now that we've done all this work to write a powerful `find` function, we can reveal that strings already have their own built-in `find` method. It can do everything that our code can do, and more!

```
test(ss.find("s") == 7)
test(ss.find("s", 7) == 7)
test(ss.find("s", 8) == 13)
test(ss.find("s", 8, 13) == -1)
test(ss.find(".") == len(ss)-1)
```

The built-in `find` method is more general than our version. It can find substrings, not just single characters:

```
>>> "banana".find("nan")
2
>>> "banana".find("na", 3)
4
```

THE SPLIT METHOD

One of the most useful methods on strings is the `split` method: it splits a single multi-word string into a list of individual words, removing all the whitespace between them. (Whitespace means any tabs, newlines, or spaces.) This allows us to read input as a single string, and split it into words.

```
>>> ss = "Well I never did said Alice"
>>> wds = ss.split()
```

```
>>> wds
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

THE STRING FORMAT METHOD

The easiest and most powerful way to format a string in Python 3 is to use the `format` method. To see how this works, let's start with a few examples:

```
1  s1 = "His name is {0}!".format("Arthur")
2  print(s1)
3
4  name = "Alice"
5  age = 10
6  s2 = "I am {1} and I am {0} years old.".format(age, name)
7  print(s2)
8
9  n1 = 4
10 n2 = 5
11 s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
12 print(s3)
```

Running the script produces:

```
His name is Arthur!
I am Alice and I am 10 years old.
2**10 = 1024 and 4 * 5 = 20.000000
```

The template string contains *place holders*, `... {0} ... {1} ... {2} ...` etc. The `format` method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted — make sure you understand line 6 above!

But there's more! Each of the replacement fields can also contain a **format specification** — it is always introduced by the `:` symbol (Line 11 above uses one.) This modifies how the substitutions are made into the template, and can control things like:

- whether the field is aligned to the left `<`, center `^`, or right `>`
- the width allocated to the field within the result string (a number like `10`)
- the type of conversion (we'll initially only force conversion to float, `f`, as we did in line 11 of the code above, or perhaps we'll ask integer numbers to be converted to hexadecimal using `x`)
- if the type conversion is a float, you can also specify how many decimal places are wanted (typically, `.2f` is useful for working with currencies to two decimal places.)

II. LISTS

A **list** is an ordered collection of values. The values that make up a list are called its **elements**, or its **items**. We will use the term *element* or *item* to mean the same thing. Lists are similar to strings, which are ordered collections of characters, except that theelements of a list can be of anytype. Lists and strings — and other collections that maintain the order of their items — are called **sequences**.

LIST VALUES

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
ps = [10, 20, 30, 40]
qs = ["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
zs = ["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested**.

Finally, a list with no elements is called an empty list, and is denoted [].

We have already seen that we can assign list values to variables or pass lists as parameters to functions:

```
>>> vocabulary = ["apple", "cheese", "dog"]
>>> numbers = [17, 123]
>>> an_empty_list = []
>>> print(vocabulary, numbers, an_empty_list)
["apple", "cheese", "dog"] [17, 123] []
```

ACCESSING ELEMENTS

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string — the index operator: [] (not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> numbers[0]
17
```

Any expression evaluating to an integer can be used as an index:

```
>>> numbers[9-8]
5
>>> numbers[1.0]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: list indices must be integers, not float
```

If you try to access or assign to an element that does not exist, you get a runtime error:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

It is common to use a loop variable as a list index.

```
horsemen = ["war", "famine", "pestilence", "death"]

for i in [0, 1, 2, 3]:
    print(horsemen[i])
```

Each time through the loop, the variable `i` is used as an index into the list, printing the `i`'th element. This pattern of computation is called a **list traversal**.

The above sample doesn't need or use the index `i` for anything besides getting the items from the list, so this more direct version — where the `for` loop gets the items — might be preferred:

```
horsemen = ["war", "famine", "pestilence", "death"]

for h in horsemen:
    print(h)
```

LIST LENGTH

The function `len` returns the length of a list, which is equal to the number of its elements. If you are going to use an integer index to access the list, it is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
horsemen = ["war", "famine", "pestilence", "death"]

for i in range(len(horsemen)):
    print(horsemen[i])
```

The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element.

Although a list can contain another list, the nested list still counts as a single element in its parent list. The length of this list is 4:

```
>>> len(["car makers", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
4
```

LIST MEMBERSHIP

`in` and `not in` are Boolean operators that test membership in a sequence. We used them previously with strings, but they also work with lists and other sequences:

```
>>> horsemen = ["war", "famine", "pestilence", "death"]
>>> "pestilence" in horsemen
True
>>> "debauchery" in horsemen
False
>>> "debauchery" not in horsemen
True
```

```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]

# Count how many students are taking CompSci
counter = 0
for (name, subjects) in students:
    if "CompSci" in subjects:
        counter += 1

print("The number of students taking CompSci is", counter)
```

LISTS AND FOR LOOPS

The `for` loop also works with lists, as we've already seen. The generalized syntax of a `for` loop is:

```
for VARIABLE in LIST:
    BODY
```

So, as we've seen

```
friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
for friend in friends:
    print(friend)

```

Any list expression can be used in a `for` loop:

```
for number in range(20):
    if number % 3 == 0:
        print(number)

for fruit in ["banana", "apple", "quince"]:
    print("I like to eat " + fruit + "s!")
```

The first example prints all the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits.

Since lists are mutable, we often want to traverse a list, changing each of its elements. The following squares all the numbers in the list `xs`:

```
xs = [1, 2, 3, 4, 5]

for i in range(len(xs)):
    xs[i] = xs[i]**2

```

Take a moment to think about `range(len(xs))` until you understand how it works.

In this example we are interested in both the *value* of an item, (we want to square that value), and its *index* (so that we can assign the new value to that position). This pattern is common enough that Python provides a nicer way to implement it:

```
xs = [1, 2, 3, 4, 5]

for (i, val) in enumerate(xs):
    xs[i] = val**2

```

`enumerate` generates pairs of both (index, value) during the list traversal. Try this next example to see more clearly how `enumerate` works:

```
for (i, v) in enumerate(["banana", "apple", "pear", "lemon"]):
    print(i, v)

```

```
1 banana
2 apple
3 pear
4 lemon
```

LIST OPERATIONS

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

LIST SLICES

The slice operations we saw previously with strings let us work with sublists:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

LISTS ARE MUTABLE

Unlike strings, lists are **mutable**, which means we can change their elements. Using the index operator on the left side of an assignment, we can update one of the elements:

```
>>> fruit = ["banana", "apple", "quince"]
```

```
>>> fruit[0] = "pear"
>>> fruit[2] = "orange"
>>> fruit
['pear', 'apple', 'orange']
```

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of `fruit` has been changed from `"banana"` to `"pear"`, and the last from `"quince"` to `"orange"`. An assignment to an element of a list is called **item assignment**. Item assignment does not work for strings:

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

but it does for lists:

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

With the slice operator we can update a whole sublist at once:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = ["x", "y"]
>>> a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning an empty list to them:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>> a_list
['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ["e"]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

LIST DELETION

Using slices to delete list elements can be error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list:

```
>>> a = ["one", "two", "three"]
>>> del a[1]
>>> a
['one', 'three']
```

As you might expect, `del`causes a runtime error if the index is out of range.

You can also use `del` with a slice to delete a sublist:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> del a_list[1:5]
>>> a_list
['a', 'f']
```

As usual, the sublist selected by slice contains all the elements up to, but not including, the second index.

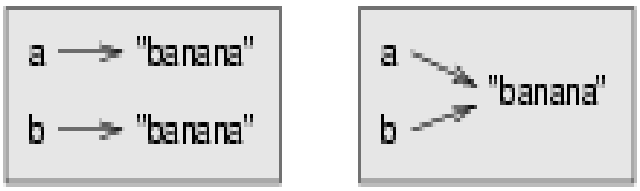
OBJECTS AND REFERENCES

After we execute these assignment statements

```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string object with the letters `"banana"`. But we don't know yet whether they point to the *same* string object.

There are two possible ways the Python interpreter could arrange its memory:



In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

We can test whether two names refer to the same object using the `is` operator:

```
>>> a is b
True
```

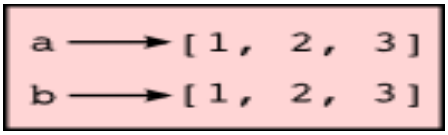
This tells us that both `a` and `b` refer to the same object, and that it is the second of the two state snapshots that accurately describes the relationship.

Since strings are *immutable*, Python optimizes resources by making two names that refer to the same string value refer to the same object.

This is not the case with lists:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

The state snapshot here looks like this:



`a` and `b` have the same value but do not refer to the same object.

ALIASING

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```

In this case, the state snapshot looks like this:

Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other:

```
>>> b[0] = 5
>>> a
[5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects (i.e. lists at this point in our textbook, but we'll meet more mutable objects as we cover classes and objects, dictionaries and sets). Of course, for

immutable objects (i.e. strings, tuples), there's no problem — it is just not possible to change something and get a surprise when you access an alias name. That's why Python is free to alias strings (and any other immutable kinds of data) when it sees an opportunity to economize.

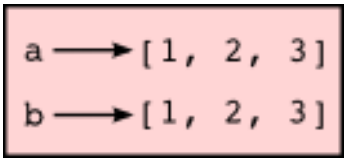
CLONING LISTS

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest wayto clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list. So now the relationship is like this:



Now we are free to make changes to `b` without worrying that we'll inadvertently be changing `a`:

```
>>> b[0] = 5
>>> a
[1, 2, 3]
```

LIST METHODS

The dot operator can also be used to access built-in methods of list objects. We'll start with the most useful method for adding something onto the end of an existing list:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
```

`append` is a list method which adds the argument passed to it to the end of the list.

```
>>> mylist.insert(1, 12) # Insert 12 at pos 1, shift other items up
>>> mylist
[5, 12, 27, 3, 12]
```

```
>>> mylist.count(12)       # How many times is 12 in mylist?
2
>>> mylist.extend([5, 9, 5, 11])   # Put whole list onto end of mylist
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)              # Find index of first 9 in mylist
6
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)           # Remove the first 12 in the list
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
```

III. STRINGS AND LISTS

Two of the most useful methods on strings involve conversion to and from lists of substrings. The `split` method (which we've already seen) breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

```
>>> song = "The rain in Spain..."
>>> wds = song.split()
>>> wds
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a **delimiter** can be used to specify which string to use as the boundary marker between substrings. The following example uses the string `ai` as the delimiter:

```
>>> song.split("ai")
['The r', 'n in Sp', 'n...']
```

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements:

```
>>> glue = ";"
>>> s = glue.join(wds)
>>> s
'The;rain;in;Spain...'
```

The list that you glue together (`wds` in this example) is not modified. Also, as these next examples show, you can use empty glue or multi-character strings as glue:

```
>>> '' --- ''.join(wds)
```

```
'The --- rain --- in --- Spain...'
>>> "".join(wds)
'TherainінSpain...'
```

LIST AND RANGE

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list.

```
>>> xs = list("Crunchy Frog")
>>> xs
["C", "r", "u", "n", "c", "h", "y", " ", "F", "r", "o", "g"]
>>> "".join(xs)
'Crunchy Frog'
```

One particular feature of `range` is that it doesn't instantly compute all its values: it "puts off" the computation, and does it on demand, or "lazily". We'll say that it gives a **promise** to produce the values when they are needed. This is very convenient if your computation short-circuits a search and returns early, as in this case:

```
def f(n):
    """ Find the first positive integer between 101 and less
        than n that is divisible by 21
    """
    for i in range(101, n):
        if (i % 21 == 0):
            return i


test(f(110) == 105)
test(f(1000000000) == 105)
```

In the second test, if range were to eagerly go about building a list with all those elements, you would soon exhaust your computer's available memory and crash the program. But it is cleverer than that! This computation works just fine, because the `range` object is just a promise to produce the elements if and when they are needed. Once the condition in the `if` becomes true, no further elements are generated, and the function returns.

```
>>> range(10)          # Create a lazy promise
range(0, 10)
>>> list(range(10))    # Call in the promise, to produce a list.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

NESTED LISTS

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

If we output the element at index 3, we get:

```
>>> print(nested[3])
[10, 20]
```

To extract an element from the nested list, we can proceed in two steps:

```
>>> elem = nested[3]
>>> elem[0]
10
```

Or we can combine them:

```
>>> nested[3][1]
20
```

Bracket operators evaluate from left to right, so this expression gets the 3'th element of `nested` and extracts the 1'th element from it.

UNIT V – Question Bank

PART -A

1. Given Str="BSC STUDENTS", Find Str[-4].
2. Find the output of the following:
   ```
   s = "We are BSC Students of Sathyabama"
   print(s[0:7])
   ```
3. State True or False: Strings are Immutable.
4. Identify the use of **in** and **not in** Operators.
5. How do you compare strings?
6. What is the output of the following:
   ```
   "Sathyabama".find("ab")
   ```
7. What is the use of split method in strings?
8. Define Lists.
9. Create a list with 5 integers, 2 floats, 3 strings.
10. State the functionality of len() function.
11. State how membership operators operates over lists?
12. What are List operations?
13. State True or False: Lists are Immutable.
14. Narrate how Slicing is carried out in Lists.

15. How to Clone the lists. Give an example.
16. Give examples of Nested Lists?

PART -B

1. How Python operates over Strings?
    a. Explain Slicing in Strings.
    b. How String Comparison is done?
    c. How in and not in Operators are employed over strings?
2. Create lists with 10 values and explain the following using them:
    a. Positive and negative Indexing
    b. List Traversal
    c. List Slicing
    d. Membership Operations
3. Create lists with 10 values and explain the following using them:
    a. Employ + and * and print the results.
    b. List Slicing
    c. List deletion
    d. Aliasing
4. List out few List methods. Explain them with examples.
5. Narratethe purpose of find function and Split method employed over strings?
6. a. Demonstrate with examples, how Cloning of lists can be done?
    b. Comment on: Lists and for loops.