

INTRODUCTION TO PYTHON

Python is a widely used high-level programming language for general-purpose programming, created by “Guido van Rossum” and first released in 1991. An interpreted language, Python has design philosophy which emphasizes code readability, and a syntax which allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale. Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library. Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems.

Python is used for:

- Web development (server-side)
- Software development
- Mathematics
- System scripting

Python can be used:

- On a server to create web applications
- Alongside software to create workflows
- To connect to database systems.
- It can also read and modify files
- To handle big data and perform complex mathematics
- For rapid prototyping, or for production-ready software development

Why Python?

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

Python was designed for readability, and has some similarities to the English language with influence from mathematics. Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses. Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes.

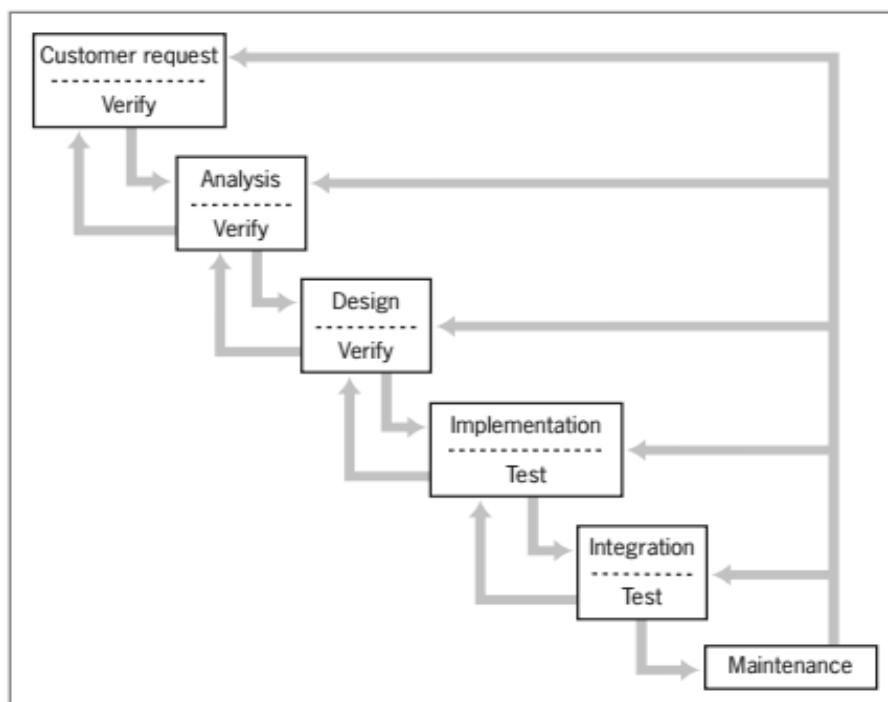
Other programming languages often use curly-brackets for this purpose. The most recent major version of Python is Python 3.

PROGRAM DEVELOPMENT CYCLE

Program Development Life Cycle (PDLC) is a systematic way of developing quality software. It provides an organized plan for breaking down the task of program development into manageable chunks, each of which must be successfully completed before moving on to the next phase. Python's development cycle is dramatically shorter than that of traditional tools.

There is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The “more” consists of organization and planning, and various conventions for diagramming those plans. Computer scientists refer to the process of planning and organizing a program as software development. There are several approaches to software development. One version is known as the waterfall model. The waterfall model consists of several phases:

1. **Customer request**—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the user requirements phase.
2. **Analysis**—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.
3. **Design**—The programmers determine how the program will do its task.
4. **Implementation**—The programmers write the program. This step is also called the coding phase.
5. **Integration**—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
6. **Maintenance**—Programs usually have a long life; a life span of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.



As you can see, the figure resembles a waterfall, in which the results of each phase flow down to the next. However, a mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases. Taken together, these phases are also called the **software development life cycle**.

Although the diagram depicts distinct phases, this does not mean that developers must analyze and design a complete system before coding it. Modern software development is usually **incremental** and **iterative**. This means that analysis and design may produce a rough draft, skeletal version, or **prototype** of a system for coding.

If you want to reduce the overall cost of software development, write programs that are easy to maintain. This requires thorough analysis, careful design, and a good coding style.

INPUT, PROCESSING, AND OUTPUT

Python Installation:

- Many PCs and Macs will have python already installed.
- To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

C:\Users\Your Name>python -version

- To check if you have python installed on a Linux or Mac, then on Linux open the command line or on Mac open the Terminal and type:

python -version

- If you find that you do not have python installed on your computer, then you can download it for free from the following website: <https://www.python.org/>

Python Quick start:

- Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.
- The way to run a python file is like this on the command line:

C:\Users\Your Name>python helloworld.py

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

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

print("Hello, World!")

Save your file with the file extension .py

- Open your command line, navigate to the directory where you saved your file, and run:

C:\Users\Your Name>python helloworld.py

The output should read: Hello, World!

- Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

exit ()

Python Indentation:

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code. Python will give you an error if you skip the indentation.

Taking the Input from user:

Let's say, we want to make a simple calculator in python. For that, we have to take one or maybe two input values from the user, then perform the user requested mathematical operation on the input numbers and return the output. For simplicity, let's just fix the user requested mathematical operation is addition for now.

We can add more operations to our simple calculator later, but let's stick to addition for now. In order to perform addition, you will have to take two numbers as input from the user (much like an ordinary calculator). The inputs must be assigned to or stored in variables (say x and y). Now, below is the code, to accept user input:

```
x = input()
```

```
y = input()
```

In order to make it more sense, let's try this:

```
x = input("Enter the first number:")
```

```
y = input("Enter the second number:")
```

The Output:

Now how can we display the input values on screen? You might think that all we have to do is just type the variable and press the Enter key. Well, it is true that we have been doing this the whole time, but this only works when you are working on IDLE. While creating real world python programs you have to write statements that output the strings or numbers explicitly. We use the print statement to do so.

```
x = input("Enter the first number:")
y = input("Enter the second number:")
print("The sum is:", x+y)
```

PYTHON COMMENTS

Comments can be used to explain Python code. Comments can be used to make the code more readable. Comments can be used to prevent execution when testing code. Comments starts with a #, and Python will ignore them:

Example:

```
#This is a comment
print("Hello, World!")
```

- Comments can also be placed at the end of a line, and Python will ignore the rest of the line:

```
print("Hello, World!") #This is a comment
```

- To add a multiline comment, we could insert# for each line:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

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

Example:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

PYTHON VARIABLES

Creating Variables:

Variables are containers for storing data values. Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Example:

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

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

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

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

Variable Names:

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

Rules for Python variables:

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

Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
```

Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Assign Value to Multiple Variables:

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

Example:

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

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

Example:

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

Output Variables:

- The Python print statement is often used to output variables. To combine both text and a variable, Python uses the + character:

Example:

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

- You can also use the + character to add a variable to another variable.

Example:

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

- If you try to combine a string and a number, Python will give you an error:

Example:

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

PYTHON OPERATORS

In every programming language, operators are used for performing various types of operations on any given operand(s). They are used to evaluate different types of expressions and to manipulate the values of operands or variables by performing different operations on them.

In python, we have 7 different types of operators, they are:

- Arithmetic Operators
- Comparison Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

While the first five are commonly used operators the last two i.e. Membership and Identity operators are exclusive to python.

Python: Arithmetic Operators:

Arithmetic operators are the operators used for performing arithmetic operations on numeric operands like division, subtraction, addition etc.

Following are different arithmetic operators:

Operator	Usage	Description
+	$a + b$	Adds values on either side of the operator
-	$a - b$	Subtracts the right hand operand from the left hand operand
*	$a * b$	To multiply values on either side of the operator
/	a / b	Divides left hand operand by right hand operand (returns float value)
%	$a \% b$	Returns the remainder from dividing left hand operand by right hand operand
**	$a ** b$	Returns Exponent – left operand raised to the power of right
//	$a // b$	Floor Division – The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)

Python: Comparison Operators:

These operators are used to compare the values of operands on either side of this type of operators. These operators return true or false Boolean values. They return true if the condition is satisfied otherwise the return false.

Following are different comparison operators:

Operator	Usage	Description
==	a == b	Returns True if the values on the either side of the operator is equal otherwise False
!=	a != b	Returns True if the values on either sides of the operator is not equal to each other otherwise False
>	a > b	Returns True if the value of the operand on the left of the operator is greater than the value on the right side of the operator
<	a < b	Returns True if the value of the operand on the left of the operator is less than the value on the right side of the operator
>=	a >= b	Returns True if the value of the operand on the left of the operator is greater than or equal to the value on the right side of the operator
<=	a <= b	Returns True if the value of the operand on the left of the operator is less than or equal to the value on the right side of the operator

Python: Assignment Operators:

The assignment operator is used to assign a specific value to a variable or an operand. The equal to (=) operator is used to assign a value to an operand directly, if we use any arithmetic operator (+, -, /, etc.) along with the equal to operator then it will perform the arithmetic operation on the given variable and then assign the resulting value to that variable itself.

Python: Logical Operators:

The logical operators are used to perform logical operations (like and, or, not) and to combine two or more conditions to provide a specific result i.e. true or false.

Following are different logical operators:

Operator	Usage	Description
and	x and y	True if both sides of the operator is True
or	x or y	True if either of the operand is True
not	not x	Complements the operand

Python: Bitwise Operators:

Bitwise operator acts on the operands bit by bit. These operators take one or two operands. Some of the bitwise operators appear to be similar to logical operators but they aren't.

Following are different bitwise operators:

Operator	Usage	Description
and	x and y	True if both sides of the operator is True
& Binary AND	(a & b)	Operator copies a bit to the result if it exists in both operands
Binary OR	(a b)	It copies a bit if it exists in either operand
^ Binary XOR	(a ^ b)	It copies the bit if it is set in one operand but not both
~ 1's complement	(~a)	It is unary and has the effect of 'flipping' bits
<< Binary Left Shift	a << 2	The left operands value is moved left by the number of bits specified by the right operand
>> Binary Right Shift	a >> 2	The left operands value is moved right by the number of bits specified by the right operand

Python: Membership Operators:

The membership operators are used to test whether a value is in a specific sequence or not like in lists, tuples, string, etc. It returns True or False as output.

Following are different membership operators:

Operator	Usage	Description
in	x in y	True if the value/operand in the left of the operator is present in the sequence in the right of the operator
not in	x not in y	True if the value/operand in the left of the operator is not present in the sequence in the right of the operator

Python: Identity Operators:

The identity operators are used to test whether a variable refers to same value/object or not. It returns True or False as output.

Following are different identity operators:

Operator	Usage	Description
is	x is True	True if both the operands refer to the same object
is not	x is not True	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise

PYTHON TYPE CONVERSIONS

The process of converting a Python data type into another data type is known as type conversion. There are mainly two types of type conversion methods in Python, namely, implicit type conversion and explicit type conversion.

1. Implicit Type Conversion in Python

In Python, when the data type conversion takes place during compilation or during the run time, then it's called an implicit data type conversion. Python handles the implicit data type conversion, so we don't have to explicitly convert the data type into another data type.

Example:

Let's add two Python variables of two different data types and store the result in a variable.

```
a = 5
b = 5.5
sum = a + b
print (sum)
print (type (sum)) #type() is used to display the datatype of a variable
```

Output:

```
10.5
<class 'float'>
```

In the above example, we have taken two variables of integer and float data types and added them. Further, we have declared another variable named "sum" and stored the result of the addition in it. When we checked the data type of the sum variable, we can see that the data type of the sum variable has been automatically converted into the float data type by the Python compiler. This is called implicit type conversion.

The reason that the sum variable was converted into the float data type and not the integer data type is that if the compiler had converted it into the integer data type, then it would have had to remove the fractional part and that would have resulted in data loss. So, Python always converts

smaller data type into larger data type to prevent the loss of data. In this context, Python cannot use implicit type conversion and that's where explicit type conversion comes into play.

Explicit Type Conversion in Python

Explicit type conversion is also known as type casting. Explicit type conversion takes place when the programmer clearly and explicitly defines the same in the program. For explicit type conversion, there are some in-built Python functions.

Following table contains some of the in-built functions for type conversion, along with their descriptions.

Function	Description
int(y [base])	It converts y to an integer, and Base specifies the number base. For example, if you want to convert the string in decimal number then you will use 10 as base
float(y)	It converts y to a floating-point number
complex(real [,imag])	It creates a complex number
str (y)	It converts y to a string
tuple(y)	It converts y to a tuple
list(y)	It converts y to a list
set(y)	It converts y to a set
dict(y)	It creates a dictionary and y should be a sequence of (key,value) tuples
ord(y)	It converts a character into an integer
hex(y)	It converts an integer to a hexadecimal string
oct(y)	It converts an integer to an octal string

Example:

```
# adding string and integer data types using explicit type conversion
```

```
a = 100
```

```
b = "200"
```

```
result1 = a + b
```

```
b = int(b)
```

```
result2 = a + b
```

```
print(result2)
```

Output:

Traceback (most recent call last):

File "", line 1, in

TypeError: unsupported operand type(s) for +: "int" and "str"

300

In the above example, the variable *a* is of the number data type and variable *b* is of the string data type. When we try to add these two integers and store the value in a variable named result1, a **TypeError** occurs as shown in the output. So, in order to perform this operation, we have to use explicit typecasting. As we can see in the above code block, we have converted the variable *b* into int type and then added variable *a* and *b*. The sum is stored in the variable named result2, and when printed it displays 300 as output, as we can see in the output block.

PYTHON EXPRESSIONS

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expression since it represents the value of the string as well. Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

Python expressions only contain identifiers, literals, and operators.

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python, you can implement the following operations using the corresponding tokens.

Operator	Token	Operator	Token
add	+	or	\
subtract	-	Binary Xor	^
multiply	*	Binary ones complement	~
power	**	Less than	<
Integer Division	/	Greater than	>
remainder	%	Less than or equal to	<=
decorator	@	Greater than or equal to	>=
Binary left shift	<<	Check equality	==
Binary right shift	>>	Check not equal	!=
and	&		

Following are a few types of python expressions:

1. List comprehension

The syntax for list comprehension is shown below:

[compute(var) for var in iterable]

For example, the following code will get all the number within 10 and put them in list.

```
>>> [x for x in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Dictionary comprehension

This is the same as list comprehension but will use curly braces:

{ k, v for k in iterable }

For example, the following code will get all the numbers within 5 as the keys and will keep the corresponding squares of those numbers as the values.

```
>>> {x:x**2 for x in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

3. Generator expression

The syntax for generator expression is shown below:

(compute(var) for var in iterable)

For example, the following code will initialize a generator object that returns the values within 10 when the object is called.

```
>>> (x for x in range(10))
```

```
<generator object <genexpr> at 0x7fec47aee870>
```

```
>>> list(x for x in range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. Conditional Expression

You can use the following construct for one-liner conditions:

true_value if Condition else false_value

Example:

```
>>> x = "1" if True else "2"
```

```
>>> x
```

```
'1'
```

PYTHON - DATA TYPES

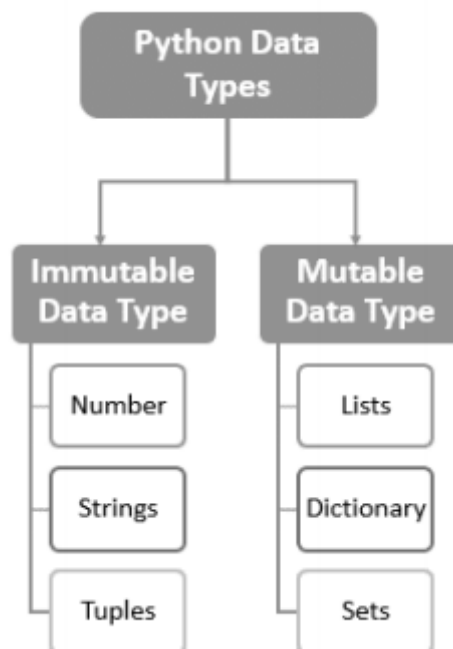
In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. One of the most crucial parts of learning any programming language is to understand how data is stored and manipulated in that language.

Users are often inclined toward Python because of its ease of use and the number of versatile features it provides. One of those features is dynamic typing. In Python, unlike statically typed languages like C or Java, there is no need to specifically declare the data type of the variable. In dynamically typed languages such as Python, the interpreter itself predicts the data type of the Python Variable based on the type of value assigned to that variable. As the name suggests, a data type is the classification of the type of values that can be assigned to variables.

In Python, we don't need to declare a variable with explicitly mentioning the data type, but it's still important to understand the different types of values that can be assigned to variables in Python. After all the data type of a variable is decided based on the value assigned. Python data types are categorized into two as follows:

- 1. Mutable Data Types:** Data types in python where the value assigned to a variable can be changed.
- 2. Immutable Data Types:** Data types in python where the value assigned to a variable cannot be changed.

Following diagram lists the data types that fall under the categories of the mutable data type and the immutable data type.



Let's discuss on the above-mentioned standard data types in Python.

- **Numbers:** The number data type in Python is used to store numerical values. It is used to carry out the normal mathematical operations.
- **Strings:** Strings in Python are used to store textual information. They are used to carry out operations that perform positional ordering among items.
- **Lists:** The list data type is the most generic Python data type. Lists can consist of a collection of mixed data types, stored by relative positions.
- **Tuples:** Tuples are one among the immutable Python data types that can store values of mixed data types. They are basically a list that cannot be changed.
- **Sets:** Sets in Python are a data type that can be considered as an unordered collection of data without any duplicate items.
- **Dictionaries:** Dictionaries in Python can store multiple objects, but unlike lists, in dictionaries, the objects are stored by keys and not by positions

PYTHON – STRINGS

Python string is an ordered collection of characters which is used to represent and store the text-based information. Strings are stored as individual characters in a contiguous memory location. It can be accessed from both directions: forward and backward. Characters are nothing but symbols. Strings are immutable Data Types in Python, which means that once a string is created, they cannot be changed.

Creating a String in Python:

In Python, strings are created using either single quotes or double quotes. We can also use triple quotes, but usually triple quotes are used to create doc strings or multi-line strings.

```
#creating a string with single quotes
String1 = „Satish“
print (String1)

#creating a string with double quotes
String2 = “Python tutorial”
Print (Strings2)
```

After creating strings, they can be displayed on the screen using the print () method as shown in the above example. The output of the above example will be as follows:

```
Satish
Python Tutorial
```


Accessing Python String Characters:

In Python, the characters of string can be individually accessed using a method called indexing. Characters can be accessed from both directions: forward and backward. Forward indexing starts from 0, 1, 2.... Whereas, backward indexing starts from -1, -2, - ..., where -1 is the last element in a string, -2 is the second last, and so on. We can only use the integer number type for indexing; otherwise, the TypeError will be raised.

Example:

```
String1 = „Satish“  
print (String1)  
print (String1[0])  
print (String1[1])  
print (String1[-1])
```

Output:

```
Satish  
S  
a  
h
```

Updating or Deleting a String in Python:

Strings in Python are immutable and thus updating or deleting an individual character in a string is not allowed, which means that changing a particular character in a string is not supported in Python. Although, the whole string can be updated and deleted, the whole string is deleted using a built-in 'del' keyword.

Example:

```
#Python code to update an entire string  
String1 = “ My Python Tutorial”  
print (“original string: “)  
print (String1)  
String1 = “Welcome to INDIA”  
print (“Updated String: “)  
print (String1)
```

Output:

```
Original String: My Python Tutorial  
Updated String: Welcome to INDIA
```

String special operators

Assume string variable **a** holds 'hello' and **b** holds 'python', then:

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a+b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
In	Membership - Returns true if a character exists in the given string	"h" in a will give True
not in	Membership - Returns true if a character does not exist in the given string	"m" not in a Will give True

String Backslash Characters or escape sequence characters:

Operators	Description	Operators	Description
\newline	Ignored (a continuation)	\"	Double quote (keeps ")
\n	Newline (ASCII line feed)	\r	Carriage return
\\	Backslash (keeps one \)	\a	ASCII bell
\v	Vertical tab	\f	Form feed
\'	Single quote (keeps ',')	\b	Backspace
\t	Horizontal tab	\0XX	Octal value XX
\newline	Ignored (a continuation)	\"	Double quote (keeps ")
\e	Escape (usually)	\000	Null (doesn't end string)
\xXX	Hex value XX		

Example: Program to concatenate two strings:

```
S1 = "Hello"
```

```
S2 = " Vignan"
```

```
print (S1 + S2)
```

Output:

Hello Vignan

Built-in Python String Methods and Functions:

Let's understand all the standard built-in methods/ string function in Python through the following table:

String Method/ Function	Description
capitalize()	It capitalizes the first letter of a string.
center(width, fillchar)	It returns a space-padded string with the original string centered to.
count(str, beg=0, end=len(string))	It counts how many times 'str' occurs in a string or in the substring of a string if the starting index "beg" and the ending index "end" are given.
encode(encoding="UTF-8", errors="strict")	It returns an encoded string version of a string; on error, the default is to raise a ValueError unless errors are given with „ignore“ or „replace“.
endswith(suffix, beg=0, end=len(string))	It determines if a string or the substring of a string (if the starting index "beg" and the ending index "end" are given) ends with a suffix; it returns true if so, and false otherwise.
expandtabs(tabsize=8)	It expands tabs in a string to multiple spaces; defaults to 8 spaces per tab if the tab size is not provided.
find(str, beg=0, end=len(string))	It determines if "str" occurs in a string or in the substring of a string if starting index "beg" and ending index "end" are given and returns the index if found, and -1 otherwise.
index(str, beg=0, end=len(string))	It works just like find() but raises an exception if "str" not found.
isalnum()	It returns true if a string has at least one character and all characters are alpha numeric and false otherwise.
isalpha()	It returns true if a string has at least one character and all characters are alphabetic, and false otherwise.
isdigit()	It returns true if a string contains only digits, and false otherwise.
islower()	It returns true if a string has at least one cased character and all other characters are in lowercase, and false otherwise.
isupper()	It returns true if a string has at least one cased character,

	and all other characters are in uppercase, and false otherwise.
<code>len(string)</code>	It returns the length of a string.
<code>max(str)</code>	It returns max alphabetical character from the string str.
<code>min(str)</code>	It returns min alphabetical character from the string str.
<code>upper()</code>	It converts lowercase letters in a string to uppercase.
<code>rstrip()</code>	It removes all trailing whitespace of a string.
<code>split(str= " ")</code>	It is used to split strings in Python according to the delimiter str (space if not provided any) and returns the list of substrings in Python
<code>splitlines(num=string.count("\n"))</code>	It splits a string at the newlines and returns a list of each line with newlines removed.

DECISION STRUCTURES & BOOLEAN LOGIC

Python - if, if-else, if-elif-else Statements

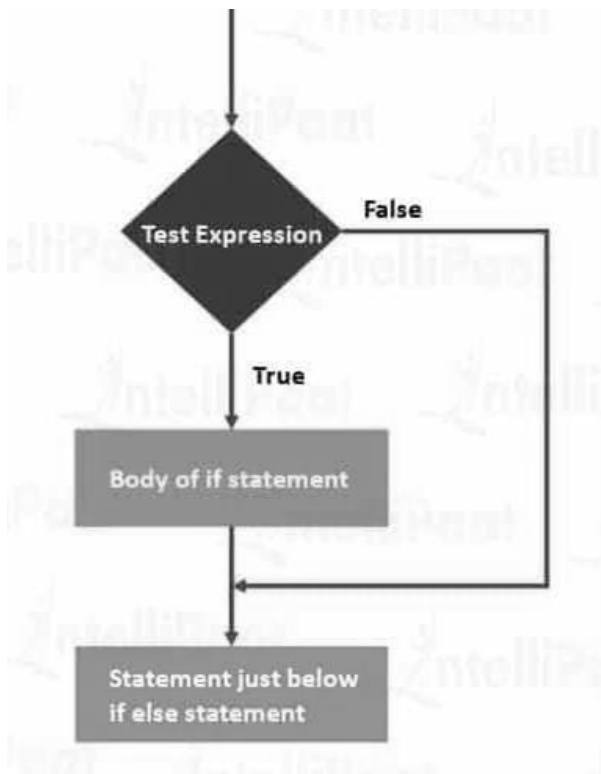
Control flow refers to the order in which the program should be executed. Generally, the control flow of a program runs from top to bottom. However, the control flow statements break the general top-to-bottom order of execution by including decision-making, looping, and more. This enables the program to first execute a specific block of code based on the conditions used.

Python Conditional Statements:

IF statement:

It is the most basic decision-making statement. It simply decides whether a particular code block will be executed or not on the basis of the condition provided in IF statement. If the condition provided in IF statement is true, then the code block is executed, and if it is false then the code block is not executed.

Following flowchart explains the working of if statement in Python:



Syntax of IF statement in Python:

if test expression:

statement(s)

As shown in the flowchart above, the Python program first evaluates the test expression. It is basically the condition in IF statement in Python. If the condition is met or if the condition is true, then only the statement(s) in the body of the if statement is(are) executed.

Note: The body of IF statement in Python starts after an indentation, unlike other languages that use brackets to write the body of if statements.

Let's see an example of the implementation of IF statement.

```
a = 5
if (a < 10):
    print ("5 is less than 10")
    print ("Statement after if statement")
```

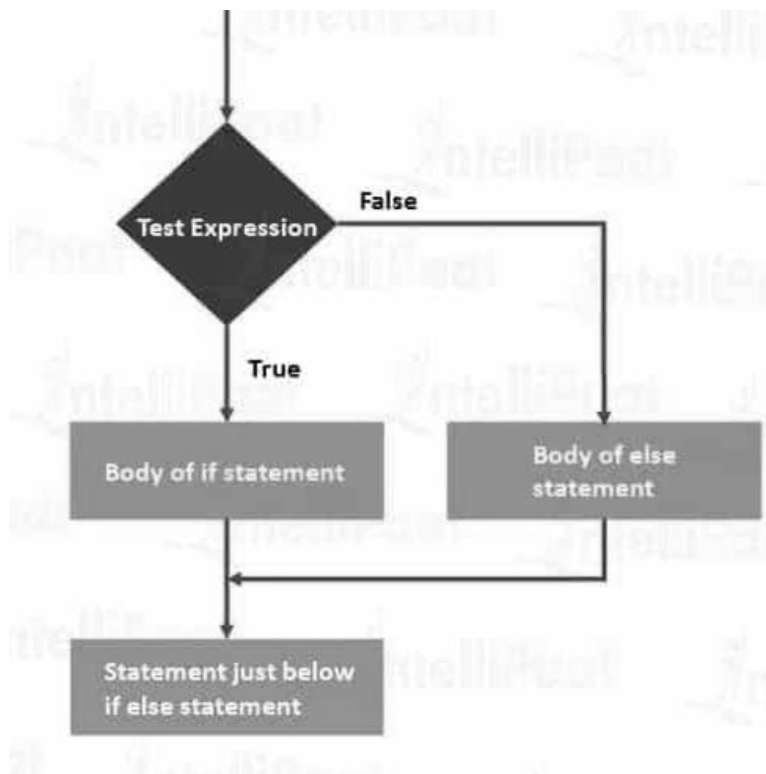
Output:

```
5 is less than 10
Statement after if statement
```

IF ELSE Statement:

IF statement in Python tells the program what to do if the condition is true. In case the condition is false, the program just goes on to execute what comes after if statements. In situations where we want the program to execute some statement if the condition is true and some other statement only if the condition is false, then we use if else in Python.

Following flowchart explains the working of if else in Python:



Syntax of the if else in Python:

if test expression:

Body of if

else:

Body of else

As depicted by the flowchart above, the Python program first evaluates the test expression. It is basically the condition in the if statement. If the condition is met or if the condition is true, then only the statement(s) in the body of the if statement is(are) executed. If the condition is not true, then the statement in the body of the else statement is executed. The body of if and else statements start with indentation.

Let's see an example of the implementation of the if...else statement.

```

i = 20;
if (i < 15):
    print ("i is smaller than 15")
else:
    print ("i is greater than 15")
print ("statement after if statement")

```

Output:

```

i is greater than 15
statement after if statement

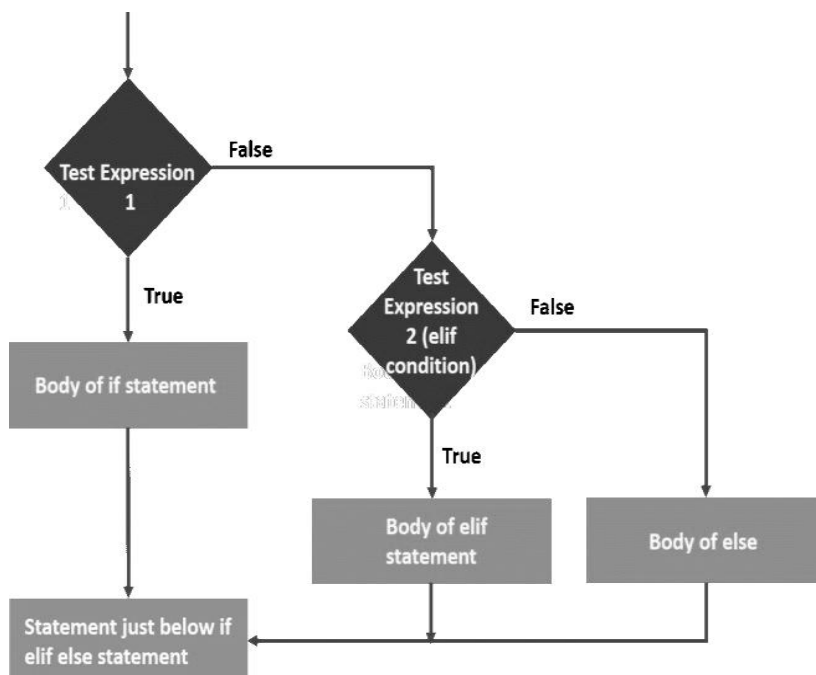
```

IF ELIF ELSE Statement:

Here, the ELIF stands for else if in Python. This conditional statement in Python allows us to check multiple statements rather than just one or two like we saw in if and if else statements. If the first IF condition is true, then same as in the previous if and if else statements, the program will execute the body of the IF statement. Otherwise, the program will go to the ELIF block (else if in Python) which basically checks for another IF statement. Again, if the condition is true, the program will execute the body of the ELIF statement, and if the condition is found to be false, the program will go to the next ELSE block and execute the body of the else block.

If all the IF conditions turn out to be false, then the body of the last else block is executed.

Following flowchart depicts the working of IF ELIF ELSE statements:



Syntax:**if test expression:****Body of if****elif test expression:****Body of elif****else:****Body of else**

We can put as many elif statements as our program requires before the last else statements, making it an if elif else ladder.

Let's see the following example of the if elif else statement to get a better understanding.

```
a = 50
```

```
if (a == 20):
```

```
    print ("value of variable a is 20")
```

```
elif (a == 30):
```

```
    print ("value of variable a is 30")
```

```
elif (a == 40):
```

```
    print ("value of variable a is 40")
```

```
else:
```

```
    print ("value of variable a is greater than 40")
```

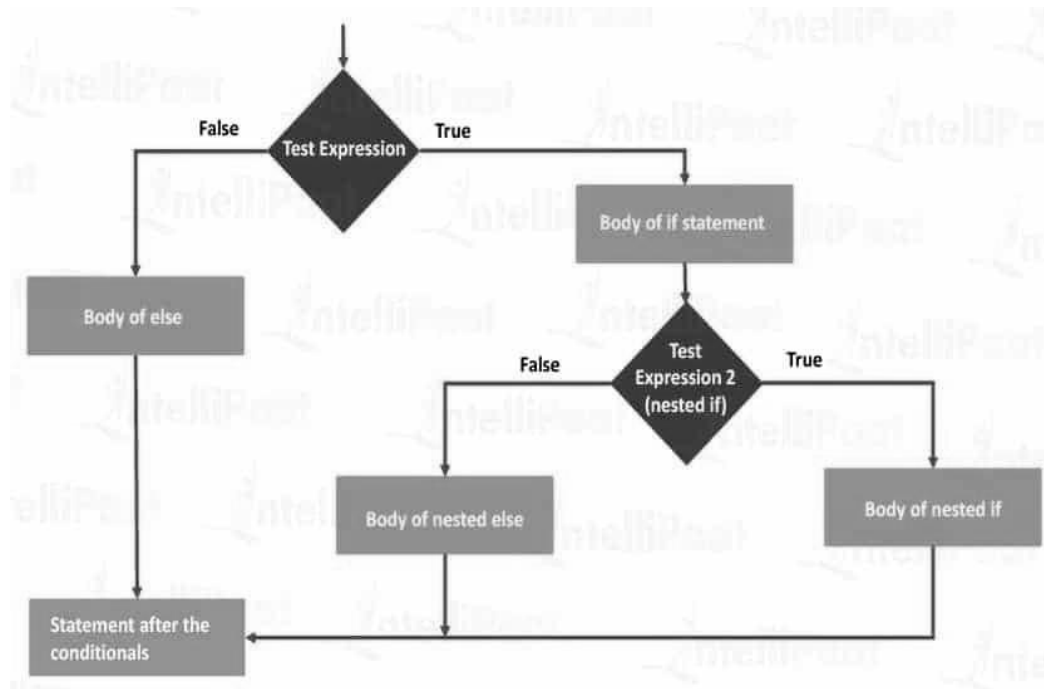
Output:

Value of variable a is greater than 40

Nested If Statement:

As the name suggests, nested if statements are nested inside other if statements. That is, a nested if statement is the body of another if statement. We use nested if statements when we need to check secondary conditions only if the first condition executes as true.

Following flow chart depicts the working of nested if statements.



Syntax of nested if in Python:

if test expression 1:

#body of if statement

if test expression 2:

else:

else:

LOOPING STATEMENTS

Loops are used when we want to repeat a block of code a number of times which can be achieved through the backward flow of program control.

WHILE LOOP:

While loop statements in Python are used to repeatedly execute a certain statement as long as the condition provided in the while loop statement stays true. While loop let the program control to iterate over a block of code.

Syntax of While Loop in Python:

while test_expression:

body of while

The program first evaluates the while loop condition. If it's true, then the program enters the loop and executes the body of the while loop. It continues to execute the body of the while loop as long as the condition is true. When it is false, the program comes out of the loop and stops repeating the body of the while loop.

Let's see the following example to understand it better.

```
a = 1
while( a<4):
    print(" loop entered", a, "times")
    a = a+1
    print("loop ends here")
```

Output:

```
loop entered 1 times
loop entered 2 times
loop entered 3 times
loop entered 4 times
loop ends here
```

Else with the While Loop in Python:

In Python, we can also use the else statement with loops. When the else statement is used with the while loop, it is executed only if the condition becomes false.

Example:

```
a = 1
while a<3:
    print("condition is true")
    a=a+1
else:
    print("condition is false now")
```

Output:

condition is true

condition is true

condition is false now

In the above example, the program keeps executing the body of the while loop till the condition is true. Since the initial value of a is 1 and every time the program entered the loop the value of a is increased by 1, the condition becomes false after the program enters the loop for the third time when the value of a is increased from 2 to 3. When the program checks the condition for the third time, it executes it as false and goes to the else block and executes the body of else, displaying, "condition is false now".

Control Transfer Statements or Loop Interruptions:

Python offers following two keywords which we can use to terminate a loop iteration forcefully.

1. Break: The break keyword terminates the loop and transfers the control to the end of the loop.

Example:

```
a = 1
while a < 5:
    a += 1
    if a == 3:
        break
    print(a)
```

Output:

2. Continue: The continue keyword terminates the ongoing iteration and transfers the control to the beginning of the loop and the loop condition is evaluated again. If the condition is true, then the next iteration takes place.

Example:

```
a = 1
while a <5:
    a += 1
    if a == 3:
        continue
    print(a)
```

Output:

```
2
4
5
```

FOR LOOP

FOR loop is used to repeat a block of code for a fixed number of times. FOR loop is yet another control flow statement since the control of the program is continuously transferred to the beginning of the FOR loop to execute the body of for loop for a fixed number of times.

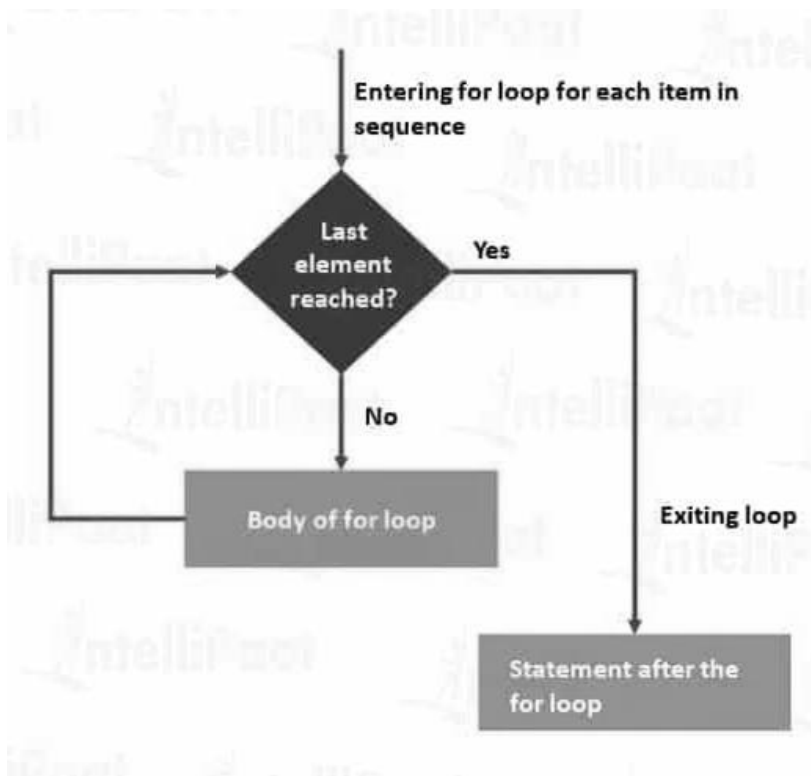
For loops in Python are used for sequential iterations for a certain number of times, that is, the length of the sequence. Iterations on the sequences in Python are called traversals.

Syntax of for loop in Python:

for a in sequence:

body of FOR loop

The following flowchart explains the working of FOR Loop:



As depicted by the flowchart, the loop will continue to execute until the last item in the sequence is reached. The body of FOR loop, like the body of the Python while loop, is indented from the rest of the code in the program.

Let us take a look at FOR loop example for better understanding:

```
square = 1
numbers_list = [1,2,3,4,5]
for i in numbers_list:
    square = i*i
print(" The square of ", i, " is ", square)
```

Output:

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

Using Python for loops with the range() function:

Example:

We can simply use Python for loop with the range() function as shown in the example below.

```
for i in range(2,10):  
    print(i)
```

Output:

2 3 4 5 6 7 8 9

Another Example:

```
for i in range(2,10,2):  
    print(i)
```

Output:

2 4 6 8

Else in For Loop:

For loop in Python can have an optional else block. The else block will be executed only when all iterations are completed. When break is used in for loop to terminate the loop before all the iterations are completed, the else block is ignored.

Example:

```
for i in range(1,6):  
    print(i)  
  
else:  
    print(" All iterations completed")
```

Output:

1
2
3
4
5
All iterations completed

Nested For Loop in Python:

As the name suggests, nested loops are the loops that are nested inside an existing loop, that is, one loop statement is the body of another loop.

Example:

```
for i in range(1,9,2):
    for j in range(i):
        print( i, end = " ")
    print()
```

Output:

```
1
3 3 3
5 5 5 5
7 7 7 7 7 7
```

In the above example, we have used nested loops to print a number pyramid pattern. The outer loop is used to handle the number of rows in the pattern, and the inner loop or the nested loop is used to handle the number of columns in the pattern. Note that we have also used the third parameter in the range() function and have set the increment to 2, that is why the number in every column is incremented by 2.

BOOLEANS

Booleans represent one of two values: True or False. In programming we often need to know if an expression is True or False. We can evaluate any expression in Python, and get one of two answers, True or False. When we compare two values, the expression is evaluated and Python returns the Boolean answer.

Example:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Output:

```
True
False
False
```

Example:

Print a message based on whether the condition is True or False:

```
a = 20
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

The bool() function allows you to evaluate any value, and give you True or False in return.

Example:

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Output:

```
True
True
```

Evaluate two variables:

```
x = "Hello"
y = 15
print(bool(x))
print(bool(y))
```

Output:

```
True
True
```

Most Values are True:

- Almost any value is evaluated to True, if it has some sort of content.
- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.

UNIT – II

ACCESSING CHARACTER AND SUBSTRING IN STRING

A string is a sequence of Unicode character/s. String consists of a series or sequence of characters that may contain alphanumeric and special characters.

Subscript Operator []

Subscript Operator [] is used to access the characters of a string at specific index position.

The simplest form of the subscript operation is the following:

`<string>[<index>]`

```
>>> name = "Alan Turing"
```

```
>>> name[0]
'A'
```

```
>>> name[3]
'n'
```

```
>>> name[-1] # using negative index one can access the characters in the string
'g'
```

In python substring is a sequence of characters within another string. In python, it is also referred to as the slicing of string.

Slicing Strings in Python:

- **Slicing using start-index and end-index:**

```
s= s[ startIndex : endIndex]
```

```
# index start from 0 in a string
```

- **Slicing using start-index without end-index:**

Example: `s = 'This is to demonstrate substring functionality in python.'`

```
s= s[ startIndex :]
```

Example:

```
s[5:]
```

Returns a sliced string from the string which starts from index position 5 till the end of the string

- **Slicing using end-index without start-index:**

```
s= s[: endIndex]
```

Example:

```
s[:7]
```

Returns a sliced string from start to seventh character

- **Slice complete string**

```
s= s[:]
```

Output:

Returns the whole string

- **Slice single character from a string:**

```
s= s[index]
```

Example:

```
s[9]
```

Output: 'o'

- **Reverse a string using a negative step:**

```
s= 'This is to demonstrated substring functionality in python.'
```

```
s[::-1]
```

Output: “.nohtyp ni ytilanoitcnuf gnirtsbus etartnsomed ot si sihT”

- The index in string starts from 0 to 5, alternatively we can use a negative index as well:

```
s= 'PYTHON'
```

```
s[0:-3]
```

Output: 'PYT'

Testing for a Substring with the in Operator

For example, you might want to pick out filenames with a **.txt** extension. A slice would work for this, but using Python's **in** operator is much simpler. When used with strings, the left operand of **in** is a target substring, and the right operand is the string to be searched. The operator **in** returns **True** if the target string is somewhere in the search string, or **False** otherwise. The next code segment traverses a list of filenames and prints just the filenames that have a **.txt** extension:

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
```

```
>>> for fileName in fileList:
```

```
    if ".txt" in fileName:
```

```
        print(fileName)
```

Output:

Myfile.txt

Yourfile.txt

DATA ENCRYPTION

Data transmitting on the Internet is vulnerable to spies and potential thieves. It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions.

Cryptography is a process which is mainly used for safe and secure communication. It works on different mathematical concepts and algorithms to transfer the encoded data into a secret code which is difficult to decode. It involves the process of encrypting and decrypting the data.

Encryption techniques are as old as the practice of sending and receiving messages. The sender **encrypts** a message by translating it to a secret code, called a **cipher text**. At the other end, the receiver **decrypts** the cipher text back to its original **plaintext** form. Both parties to this transaction must have at their disposal one or more **keys** that allow them to encrypt and decrypt messages.

A very simple encryption method that has been in use for thousands of years is called a **Caesar cipher**. This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence.

For example, if the distance value of a Caesar cipher equals three characters, the string "invaders" would be encrypted as "lqydghuv". To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement.

Below figure shows the first five and the last five plaintext characters of the lowercase alphabet and the corresponding cipher text characters, for a Caesar cipher with a distance of +3. The numeric ASCII values are listed above and below the characters.

ASCII values	97	98	99	100	101		118	119	120	121	122
Plaintext	a	b	c	d	e	...	v	w	x	y	z
Cipher text	d	e	f	g	h	...	y	z	a	b	c
ASCII values	100	101	102	103	104		121	122	97	98	99

"""

File: encrypt.py

Encrypts an input string of lowercase letters and prints the result. The other input is the distance value.

"""

```

plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - (ord('z') - ordvalue + 1)
    code += chr(cipherValue)
print(code)
"""
```

File: decrypt.py

Decrypts an input string of lowercase letters and prints the result. The other input is the distance value.

"""

```
code = input("Enter the coded text: ")
```

```

distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
    ordvalue = ord(ch)
    cipherValue = ordvalue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - (distance - (ord('a') - ordvalue - 1))
    plainText += chr(cipherValue)
print(plainText)

```

Sample Output:

Enter a one-word, lowercase message: **invaders**

Enter the distance value: 3

Lqydghuv

Enter the coded text: **lqydghuv**

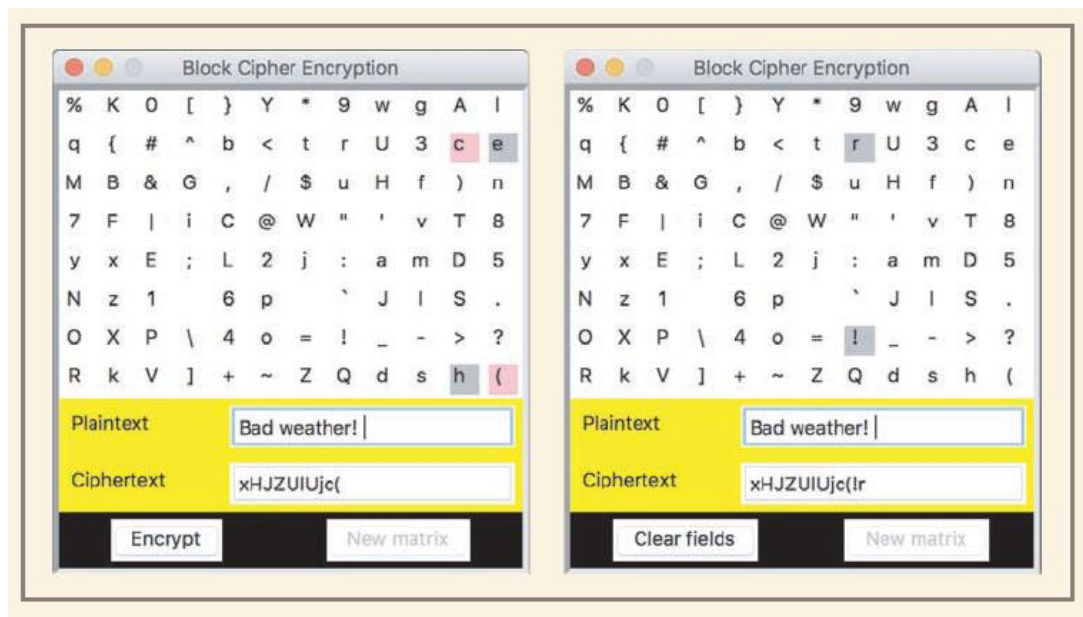
Enter the distance value: 3

Invaders

The main shortcoming of this encryption strategy is that the plaintext is encrypted one character at a time, and each encrypted character depends on that single character and a fixed distance value. In a sense, the structure of the original text is preserved in the cipher text, so it might not be hard to discover a key by visual inspection.

Block Cipher: A more sophisticated encryption scheme is called a **block cipher**. A block cipher uses plaintext characters to compute two or more encrypted characters. This is accomplished by using a mathematical structure known as an **invertible matrix** to determine the values of the encrypted characters. The matrix provides the key in this method. The receiver uses the same matrix to decrypt the cipher text. Here the information used to determine each character comes from a block of data makes it more difficult to determine the key.

A block cipher encryption method uses a two-dimensional grid of the characters, also called a **matrix**, to convert the plaintext to the cipher text. The algorithm converts consecutive pairs of characters in the plaintext to pairs of characters in the cipher text. For each pair of characters, it locates the positions of those characters in the matrix. If the two characters are in the same row or column, it simply swaps the positions of these characters and adds them to the cipher text. Otherwise, it locates the two characters at the opposite corners of a rectangle in the matrix, and adds these two characters to the cipher text.



Note that the characters are in random order in the matrix, and that the program allows the user to reset the grid to a new randomly ordered set of characters when the encryption is finished.

Python has the following modules/libraries which are used for cryptography namely:

- Cryptography
- Simple-Crypt
- Hashlib:MD5 & SHA1(Most secured)

Simple-Crypt: It is a python module which is fast and converts the plaintext to cipher text and cipher text to plain text in seconds and with just a single line of code.

Implementation:

We first need to install the library using

```
pip install simple-crypt
```

1. Loading the Library:

```
from simplecrypt import encrypt, decrypt
```

2. Encrypting and Decrypting:

Simple-crypt has two pre-defined functions encrypt and decrypt which controls the process of encryption and decryption. For encryption, we need to call the encrypt function and pass the key and message to be encrypted.

```
message = "Hello!! Welcome to INDIA!!"
ciphercode = encrypt('AIM', message)
print(ciphercode)
```

Similarly, we can call the decrypt function and decode the original message from this cipher text.

```
original = decrypt('AIM', ciphercode)
print(original)
```

Here you can see that we have used “AIM” as the password and it is the same for encryption and decryption. In simple-crypt, we should keep in mind that the same key should be provided for encryption and decryption otherwise messages will not be decoded back to original.

TEXT FILES

A text file is a computer file that only contains text only. Using a text editor such as notepad, one can create, view and save data in a text file. The data in the text file can be viewed as characters, words or numbers but all data output to or input from a text file must be strings. Thus, numbers must be converted to strings before output and these strings must be converted back to numbers after input.

Writing Text to a FILE:

Data can be output to a text file using a file object. To perform write operation on a text file, it is required to open the file in before. To open a file python's open function is used. Python's **open** function, which expects a file name and a **mode string** as arguments, opens a connection to the file on disk and returns a file object. The mode string is **'r'** for input files and **'w'** for output files.

Below example opens a file object on a file named **myfile.txt** for output:

```
>>> f = open('myfile.txt', 'w')
```

- If the file does not exist, it is created with the given filename.
- If the file already exists, Python opens it.
- When an existing file is opened for output, any data already in it are erased.

String data are written to a file using the method **write ()** with the file object. The **write** method expects a single string argument.

```
>>> f.write('First line.\nSecond line.\n')
```

When the write operation is finished, the file should be closed using the method **close**, as follows:

```
>>> f.close()
```

Failure to close an output file can result in data being lost.

Writing Numbers to a File

The file method **write** is used to write only strings. Therefore, other types of data, such as integers or floating-point numbers, must first be converted to strings before being written to an output file. In Python, the values of most data types can be converted to strings by using the **str** function

Example Program: below code writes 1 to 499 integers to a text file **integers.txt**

```
f = open('integers.txt', 'w')
for i in range(1,500):
    f.write(str(i) + '\n')
f.close()
```

Reading Text from a FILE:

Data can be input from a text file using file object. To perform the read operation, the file should be opened using the `open()` function. While opening the file, the mode should be given as `"r"`.

Example:

```
>>> f = open("myfile.txt", 'r')
```

read() method:

The file method **read** to input the entire contents of the file as a single string.

Example:

```
>>> text = f.read()
```

```
Print(text)
```

After read is finished, another call to **read** would return an empty string, to indicate that the end of the file has been reached.

readline () method:

The **readline** method consumes a line of input and returns this string, including the newline. If **readline** encounters the end of the file, it returns the empty string.

Example:

```
f = open("myfile.txt", 'r')
while True:
    line = f.readline()
    if line == "":
        break
    print(line)
```

Reading Numbers from a File

All of the file input (or read) operations return data to the program as strings. If these strings represent other types of data, such as integers or floating-point numbers, the programmer must convert them to the appropriate types before use. This can be achieved by using the functions **int** and **float**, respectively.

Example: Below is the program that reads integers from `integers.txt` and prints their sum.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    line = line.strip()
    number = int(line)
    theSum += number
print("The sum is", theSum)
```

Accessing and Manipulating Files and Directories on Disk:

When the Python is started, the shell is connected to a **current working directory**. At any point during the execution of a program, you can open a file in this directory just by using the file's name.

However, you can also access any other file or directory within the computer's file system by using a **pathname**. A file's pathname specifies the chain of directories needed to access a file or directory.

- When the chain starts with the root directory, it's called an **absolute pathname**.
- When the chain starts from the current working directory, it's called a **relative pathname**.

While accessing files in the file system, following functions are useful provided in the **os** module.

Function	Description
chdir(path)	Changes the current working directory to path.
getcwd()	Returns the path of the current working directory
listdir(path)	Returns a list of the names in directory named path.
makedirs(path)	Creates a new directory named path and places it in the current working directory.
remove(path)	Removes the file named path from the current working directory
rename(old, new)	Renames the file or directory named old to new
rmdir(path)	Removes the directory named path from the current working directory

os.path Module Functions:

- **exists(path):** It returns **True** if **path** exists and **False** otherwise.
- **isdir(path):** It returns **True** if **path** names a directory and **False** otherwise.
- **isfile(path):** It returns **True** if **path** names a file and **False** otherwise.
- **getsize(path):** It returns the size of the object names by **path** in bytes.

STRINGS AND NUMBER SYSTEMS

According to the mathematics we have four types of number systems which are representing the numbers in computer architecture.

1. Binary Number System
2. Decimal Number System
3. Octal Number System
4. Hexadecimal Number Systems

When you perform arithmetic operations, you use the **decimal number system**. This system uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits. The **binary number system** is used to represent all information in a digital computer. The two digits are 0 and 1. Because binary numbers can be long strings of 0s and 1s, computer scientists often use other number systems, such as **octal** (base eight) and **hexadecimal** (base 16) to represent numbers.

Converting Binary to Decimal

Here a binary number is represented as a string of bits or a **bit string**. Using this bit string, its equivalent decimal number is calculated as follows:

$$\begin{aligned} 1100111_2 &= \\ 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 &= \\ 1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 &= \\ 64 + 32 + 4 + 2 + 1 &= 103 \end{aligned}$$

Below is the python script for the converting the binary number into decimal

```
"""
```

File: binarytodecimal.py

Converts a string of bits to a decimal integer.

```
"""
```

```
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

Output

Enter a string of bits: 1111

The integer value is 15

Enter a string of bits: 101

The integer value is 5

Converting Decimal to Binary

This conversion of decimal to binary involves repeatedly divides the decimal number by 2. After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits. The quotient becomes the next dividend in the process. The string of bits is initially empty, and the process continues while the decimal number is greater than 0.

Below is the python script for converting the decimal to binary:

"""

File: decimaltobinary.py

Converts a decimal integer to a string of bits.

"""

```
decimal = int(input("Enter a decimal integer: "))
```

```
if decimal == 0:
```

```
    print(0)
```

```
else:
```

```
    print("Quotient Remainder Binary")
```

```
bitString = ""
```

```
while decimal > 0:
```

```
    remainder = decimal % 2
```

```
    decimal = decimal // 2
```

```
    bitString = str(remainder) + bitString
```

```
print("The binary representation is", bitString)
```

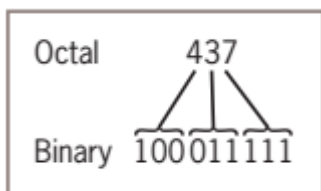
Output:

Enter a decimal integer: 34

The binary representation is 100010

Octal and Hexadecimal Number Systems

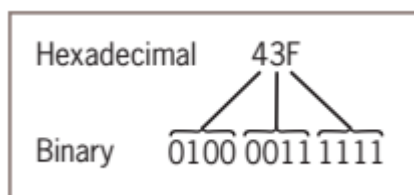
- ❖ To convert from octal to binary, you start by assuming that each digit in the octal number represents three digits binary number. You then start with the leftmost octal digit and write down the corresponding binary digits, padding these to the left with 0s to the count of 3, if necessary. You proceed in this manner until you have converted all of the octal digits. Below figure shows such a conversion:



To convert binary to octal, you begin at the right and factor the bits into groups of three bits each. You then convert each group of three bits to the octal digit they represent.

For a binary number $(100\ 011\ 111)_2$, its octal equivalent is $(437)_8$.

- ❖ To convert from hexadecimal to binary, you start by assuming that each digit in the hexadecimal number represents four digits binary number. You then start with the leftmost digit in hexadecimal number and write down the corresponding binary digits, padding these to the left with 0s to the count of 4, if necessary. You proceed in this manner until you have converted all of the hexadecimal digits. Below figure shows such a conversion:



To convert from binary to hexadecimal, you factor the bits into groups of four and look up the corresponding hex digits.

For a binary number $(0100\ 0011\ 1111)_2$, its hexadecimal equivalent is $(43F)_{16}$.

UNIT III

List and Dictionaries: Lists, Defining Simple Functions, Dictionaries

Design with Function: Functions as Abstraction Mechanisms, Problem Solving with Top down Design, Design with Recursive Functions, Case Study Gathering Information from a File System, Managing a Program's Namespace, Higher Order Function.

Modules: Modules, Standard Modules, Packages.

LISTS

A list is a sequence of data values called items or elements. An item can be of any type.

Here are some real-world examples of lists:

- A shopping list for the grocery store
- A to-do list
- A roster for an athletic team
- A guest list for a wedding
- A recipe, which is a list of instructions
- A text document, which is a list of lines
- The words in a dictionary
- The names in a phone book

Each of the items in a list is ordered by position. Each item in a list has a unique index that specifies its position. The index of the first item is 0, and the index of the last item is the length of the list minus 1.

In Python, a list is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets ([and]). Here are some example lists:

```
[1951, 1969, 1984]      # A list of integers
["apples", "oranges", "cherries"] # A list of strings
[]                      # An empty list
```

- ❖ It is also possible to create a list using list () function as shown under:

```
>>> sample = list(range(1, 5))
```

This creates a list with the set of values generated by the range function i.e. from 1 to 5. And the elements in the list can be displayed using the print () function as follows:

```
>>> print(sample)
```

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

- ❖ To print the contents of a list without the brackets and commas, you can use a **for** loop, as follows:

```
>>> for number in [1, 2, 3, 4]:
    print(number, end = " ")
1 2 3 4
```

- ❖ The function **len ()** can be used to count the number of elements in the list

Example:

```
>>> len (sample)
4
```

As the list named sample contains 4 elements it returns 4.

- ❖ Subscript Operator [] is used to refer individual elements in the list.

Example:

```
>>> sample = [1, 2, 3, 4]
>>> print(sample[2])
3
```

- ❖ You can use the **in** operator to detect the presence or absence of a given element:

```
>>> 3 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
```

- ❖ Concatenation (+) is used to combine two lists into a single list

```
>>> sample + [5, 6]
[1, 2, 3, 4, 5, 6]
```

REPLACING AN ELEMENT IN A LIST

A list is changeable—that means it is mutable. At any point in a list’s lifetime, elements can be inserted, removed, or replaced. The subscript operator is used to replace an element at an index position 3, as shown below:

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

Below is the example that shows the replace each element in the list with its square:

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> for index in range(len(numbers)):
        numbers[index] = numbers[index] ** 2
>>> numbers
[4, 9, 16, 25]
```

LIST METHODS FOR INSERTING AND REMOVING ELEMENTS

Following is the list of methods that are used to perform the inserting and removing of elements in the list:

Say **L** is the name of the List:

List Method	What It Does
L.append(element)	Adds element to the end of L .
L.extend(aList)	Adds the elements of aList to the end of L .
L.insert(index, element)	Inserts element at index if index is less than the length of L . Otherwise, inserts element at the end of L .
L.pop()	Removes and returns the element at the end of L .
L.pop(index)	Removes and returns the element at index .

insert() :

This method is used to insert a new element in the list at the specified index location. Once a new element is inserted all the elements are shifted one position to the right in the list. The following example illustrates how to use `insert()` method:

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
```

append() :

This method is used to add a new element at the end of the list. The following example illustrates how to use `append()` method:

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(3)
>>> example
[1, 2, 3]
```

extend():

This method is used to add the elements of one list to the other list. Below is the example:

```
>>> first=[1,2,3,4]
>>> print(first)
[1, 2, 3, 4]
>>> second=[5,6,7,8]
>>> first.extend(second)
>>> print(first)
[1, 2, 3, 4, 5, 6, 7, 8]
```

This clearly shows that the elements in the list “second” are added to the list “first”

pop():

The method **pop** is used to remove an element at a given position. If the position is not specified, **pop** removes and returns the last element. If the position is specified, **pop** removes the element at that position and returns it. In that case, the elements that followed the removed element are shifted one position to the left.

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)
1
>>> example
[2, 10, 11, 12]
```

SEARCHING A LIST

After elements have been added to a list, a program can search for a given element. The **in** operator determines an element’s presence or absence, but some people are more interested in the position of an element if it is found (for replacement, removal, or other use).

To locate an element position in the list **index ()** method is used but it raises an exception if the search element is not found.

Example:

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

SORTING A LIST

Sorting a list means arranging the elements of the list in the numerical order or alphabetical order. To sort the elements in the list, **sort ()** method is used, which arranges the elements of the list in the ascending order.

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

MUTATOR METHODS

Methods which can modify the internal state of the list object are called Mutator methods. Examples are **insert ()**, **append ()**, **extend ()**, **sort ()**, these methods change the state of the list, which is also desirable. And all these methods return no value but python automatically returns special value **None** even though those methods does not return any value explicitly.

Example:

```
>>>aList=[2,3,1,4]
>>> x = aList.sort()
>>> print(x)
None
```

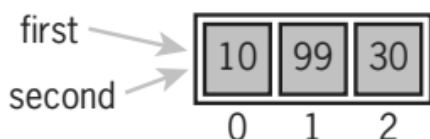
ALIASING AND SIDE EFFECTS

Aliasing means giving more than one name to single list of items.

Example:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
```

The above example shows that the variable name “second” is given to the list variable “first”. Here the variable second acts as alias to the variable “first”. This phenomenon is called Aliasing, which is shown below:



- ❖ Side effect with this aliasing is that any modifications such as insert, append, extend, replace made to the list “first” is also affected the list “second”.

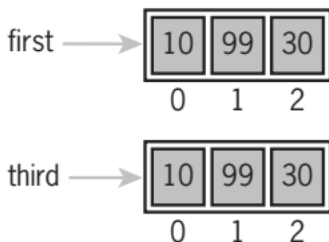
Aliasing is useful when the data structure is immutable.

To prevent the side effect of aliasing, it is good enough to create a new object and copy the contents of one list into the other.

Below is the example that shows the copying the contents of one list into other:

```
>>> first = [1,2,3]
>>> third = list(first)
>>> first
[1, 2, 3]
>>> third
[1, 2, 3]
```

The variables **first** and **third** refer to two different list objects, although their contents are initially the same, as shown in Figure, they may change without having any effect on the other.



EQUALITY: OBJECT IDENTITY AND STRUCTURAL EQUIVALENCE

Sometimes, programmers need to see whether two variables refer to the exact same object or to different objects.

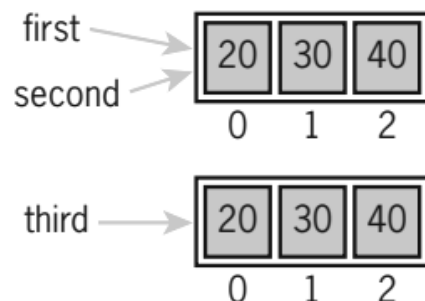
For example, you might want to determine whether one variable is an alias for another. The `==` operator returns **True** if the variables are aliases for the same object. Unfortunately, `==` also returns **True** if the contents of two different objects are the same.

The first relation is called **object identity**

The second relation is called **structural equivalence**.

The `==` operator has no way of distinguishing between these two types of relations. Python's **is** operator can be used to test for object identity. It returns **True** if the two operands refer to the exact same object, and it returns **False** if the operands refer to distinct objects (even if they are structurally equivalent).

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first)
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



TUPLES

A tuple is a type of sequence that resembles a list except that a tuple is immutable. A tuple is indicated by enclosing its elements in parentheses instead of square brackets. Below example shows how to create several tuples:

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
```

```
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
```

```
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
```

❖ Below is the example that creates a tuple from a list of data items:

```
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

Since, tuples are quite similar to lists; both of them are used in similar situations as well. Most of the operators and functions used with lists also apply to tuples. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected

DICTIONARIES

Lists organize their elements by position. This mode of organization is useful when we want to locate the first element, the last element, or visit each element in a sequence. However, in some situations, the position of a data element in a structure is irrelevant; we're interested in its association with some other element in the structure. For example, you might want to look up someone's phone number you don't care where that number is in the phone book rather you can search based on the name.

A dictionary is the one that organizes information by association, not position.

A Python dictionary is written as a sequence of key/value pairs separated by commas. These pairs are sometimes called entries. The entire sequence of entries is enclosed in curly braces ({ and }). A colon (:) separates a key and its value.

Examples:

A phone book: {"Savannah":"476-3321", "Nathaniel":"351-7743"}

Personal information: {"Name":"Molly", "Age":18}

You can even create an empty dictionary—that is, a dictionary that contains no entries.

An example of an empty dictionary:

```
d = {}
```

The keys in a dictionary can be data of any types, such as strings or integers. The associated values can be of any types.

ADDING KEYS AND REPLACING VALUES

- ❖ You add a new key/value pair to a dictionary by using the subscript operator []. The form of this operation is the following:

Dictionary_name[key] = value

The below code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name':'Sandy', 'occupation':'hacker'}
```

- ❖ The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

Adding of new values and replacing of existing values can be done in the same manner with a difference in operation that, if the key is already existed in the dictionary then its associated value is replaced, if the key is not present in the dictionary then a new key value pair is added to the dictionary.

ACCESSING VALUES

The subscript to obtain the value associated with a key. However, if the key is not present in the dictionary, Python raises an exception.

Example:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name': 'Sandy', 'occupation': 'hacker'}

>>> info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
info["job"]
KeyError: 'job'
```

Here the exception has been raised as the key “job” is not present in the dictionary.

If the existence of a key is uncertain, the programmer can test for it using the operator `in`, as follows to prevent rising of an exception:

```
>>> if "job" in info:
print(info["job"])
```

Another way is to use the method `get`. This method expects two arguments, a possible key and a default value. If the key is in the dictionary, the associated value is returned. However, if the key is absent, the default value passed to `get` is returned.

Here is an example of the use of `get` with a default value of **None**:

```
>>> print(info.get("job", None))
None
```

REMOVING KEYS

To delete an entry from a dictionary, one removes its key using the method `pop`. This method expects a key and an optional default value as arguments. If the key is in the dictionary, it is removed, and its associated value is returned. Otherwise, the default value is returned.

Example:

```
>>> info = {'name': 'Sandy', 'occupation': 'hacker'}
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
Manager
>>> info
{'name': 'Sandy'}
```

TRAVERSING A DICTIONARY

When **for** loop is used with a dictionary, the loop's variable is bound to each key in an unspecified order. The below code segment prints all of the keys and their values in **info** dictionary:

```
for key in info:  
    print(key, info[key])
```

Alternatively, you could use the dictionary method **items()** to access the dictionary's entries. Below example shows the use of this method with a dictionary of grades:

```
>>> grades = {90:'A', 80:'B', 70:'C'}  
>>> list(grades.items())  
[(80,'B'), (90,'A'), (70,'C')]
```

Note that the entries are represented as tuples within the list. A tuple of variables can then access the key and value of each entry in this list within **for** loop:

```
for (key, value) in grades.items():  
    print(key, value)
```

One can obtain a list of keys using the **keys** method and process this list to rearrange the keys. For example, you can sort the list and then traverse it to print the entries of the dictionary in alphabetical order:

```
theKeys = list(info.keys()) # this statement generates a list only with the keys in dictionary  
theKeys.sort()           # this statement sorts the keys in the list  
for key in theKeys:      # this loop prints key value pairs in the sorted order of keys  
    print(key, info[key])
```

Below is the list of commonly used dictionary operations, let **d** be the name of the dictionary:

- **len(d)** Returns the number of entries in **d**.
- **d[key]** Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
- **d.get(key [, default])** Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
- **d.pop(key [, default])** Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
- **list(d.keys())** Returns a list of the keys
- **list(d.values())** Returns a list of the values
- **list(d.items())** Returns a list of tuples containing the keys and values for each entry.
- **d.clear()** Removes all the keys.

FUNCTIONS

Defining our own functions allows us to organize our code in existing scripts more effectively. Most of the functions used thus far expect one or more arguments and return a value.

Let's define a function that expects a number as an argument and returns the square of that number.

```
def square(x):  
    return x * x
```

The definition of this function consists of a **header** and a **body**. The header includes the keyword **def** as well as the function name and list of parameters. The function's body contains one or more statements. Here is the syntax:

```
def <function name>(<parameter-1>, ..., <parameter-n>):
```

```
    -----  
    -----  
    -----
```

The function's body contains the statements that execute when the function is called. A function may contain a single **return** statement which returns the result of the computations done in the function.

Now consider how the function will be used. Its name is square, so we can call it like this:

```
>>> square(2)  
4  
>>> square(6)  
36  
>>> square(2.5)  
6.25
```

PARAMETERS AND ARGUMENTS

Parameter is the name used in the function definition for an argument that is passed to the function when it is called. The number and positions of the arguments of a function call should match the number and positions of the parameters in that function's definition. Some functions expect no arguments, so they are defined with no parameters.

RETURN STATEMENT

The programmer places a **return** statement at each exit point of a function when that function should explicitly return a value. The syntax of the **return** statement is as follows:

```
return <expression>
```

Upon encountering a **return** statement, immediately transfers control back to the caller of the function. The return value is also sent back to the caller. If a function contains no **return** statement, Python transfers control to the caller after the last statement in the function's body is executed, and the special value **None** is automatically returned.

BOOLEAN FUNCTIONS

A **Boolean function** usually tests its argument for the presence or absence of some property. The function returns **True** if the property is present, or **False** otherwise.

The below example shows the use and definition of the Boolean function **odd**, which tests a number to see whether it is odd.

```
def odd(x):
    if x % 2 == 1:
        return True
    else:
        return False
```

Calling the function odd:

```
>>> odd(5)
True
>>> odd(6)
False
```

DEFINING A MAIN FUNCTION:

In python scripts that include the definitions of several functions, it is often useful to define a special function named **main** that serves as the entry point for the script. This function usually expects no arguments and returns no value. Its sole purpose is to take inputs, process them by calling other functions, and print the results.

The definition of the **main** function and the other function definitions can appear in no particular order in the script, as long as **main** is called at the very end of the script.

The following example shows a complete script that is organized in the manner just described. The **main** function prompts the user for a number, calls the **square** function to compute its square, and prints the result. We can define the **main** and the **square** functions in any order.

When Python loads this module, the code for both function definitions is loaded and compiled, but not executed. Note that **main** is then called as the last step in the script. This has the effect of transferring control to the first instruction in the **main** function's definition. When **square** is called from **main**, control is transferred from **main** to the first instruction in **square**. When a function completes execution, control returns to the next instruction in the caller's code.

```
def main():
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)
def square(x):
    return x * x
# The entry point for program execution
if __name__ == "__main__":
    main()
```

TYPES OF ARGUMENTS

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

- Required arguments
- Default arguments
- Keyword 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.

```
def prod(a,b):  
    print("product of two numbers is:")  
    prod=a*b  
    print(prod)  
prod(10,15) #function call with required no.of parameters  
  
output:  
150
```

If required number of arguments are not passed either more in number or less in number, then it shows an error.

Keyword Arguments

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. If we want to pass actual parameters in different order to function definition i.e same name or keyword should be used in formal parameters.

Example:

```
>>>def printinfo(name,age):  
    print("name:",name)  
    print("age:",age)  
  
>>>printinfo(name="john",age=50) #two keyword arguments  
name:john  
age:50  
>>>printinfo(age=50,name="john") #two keyword arguments(out of order)  
name:john  
age:50  
>>>printinfo("john",age=50) #1 positional and 1 keyword argument
```

One can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

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.

Example:

```
def printinfo(name,age=35):
    print("name:",name)
    print("age:",age)
printinfo(age=50,name='john')
printinfo(name='john')# takes age as default

output:
name: john
age: 50
name: john
age: 35
```

In the above example observe second call of **printinfo()** function. We did not provide 'age' as an argument. But in formal parameters we are having an argument as 'age'. So the function call takes age=35 as default argument.

In function definition, default arguments are placed after the non – default arguments, otherwise it shows an error.

FUNCTIONS AS ABSTRACTION MECHANISMS

The problem with the human brain is that we can wrap ourselves around just a few things at once. People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an abstraction. Put most plainly, an abstraction hides detail and thus allows a person to view many things as just one thing. Likewise, effective designers must invent useful abstractions to control complexity with the use of functions.

Functions Eliminate Redundancy

The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code. Consider the below example:

```
def summation(lower, upper):
    result = 0
    while lower <= upper:
        result += lower
        lower += 1
    return result
```

```
>>> summation(1,4)
10
>>> summation(50,100)
3825
```

If the **summation** function didn't exist, the programmer would have to write the entire algorithm every time a summation is computed. In a program that must calculate multiple summations, the same code would appear multiple times. In other words, redundant code would be included in the program. Code redundancy is bad for several reasons. Therefore, it is clear that the functions can eliminate redundancy.

Functions Hide Complexity

Another way that functions serve as abstraction mechanisms is by hiding complicated details. If the same summation function is considered, just making a call to summation to add a range of values is very simple though writing the logic for this summation is little complex. It ensures simplicity by defining an independent function block of code from its surrounding code.

Functions Support General Methods

An algorithm is a **general method** for solving a class of problems. The individual problems that make up a class of problems are known as **problem instances**. The problem instances for our summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed. The problem instances of a given algorithm can vary from program to program. When you design an algorithm, it should be general enough to provide a solution to many problem instances, not just one or a few of them.

Functions Support the Division of Labor

In a computer program, functions can enforce a division of labor. Ideally, each function performs a single task, such as computing a summation or formatting a table of data for output. Each function is responsible for using certain data, computing certain results, and returning those results. Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions.

DESIGN WITH RECURSIVE FUNCTIONS

While designing, one can decompose a complex problem into a set of simpler problems and solve these with different functions. In some cases, you can decompose a complex problem into smaller problems of the same form. In these cases, the sub problems can all be solved by using the same function. This design strategy is called recursive design, and the resulting functions are called recursive functions.

Defining a Recursive Function

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 that stops recursive call. This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step.

Tracing a Recursive Function

To get a better understanding of how a recursive function works, it is helpful to trace its calls. How calls are made is shown under with an example of recursive function to find the factorial of an integer. Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

```
def calc_factorial(x):
    """this is the recursive function

    to find the factorial of an integer"""
    if x== 1:
        return 1
    else:
        return(x*calc_factorial(x-1))
num=4
print("the factorial of",num,"is",calc_factorial(num))

output:

the factorial of 4 is 24
```

In the above example, `calc_factorial()` is a recursive functions as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number. This recursive call can be explained in the following steps:

<code>calc_factorial(4)</code>	# 1st call with 4
<code>4 * calc_factorial(3)</code>	# 2nd call with 3
<code>4 * 3 * calc_factorial(2)</code>	# 3rd call with 2
<code>4 * 3 * 2 * calc_factorial(1)</code>	# 4th call with 1
<code>4 * 3 * 2 * 1</code>	# return from 4th call as number=1
<code>4 * 3 * 2</code>	# return from 3rd call
<code>4 * 6</code>	# return from 2nd call
 24	# return from 1st call

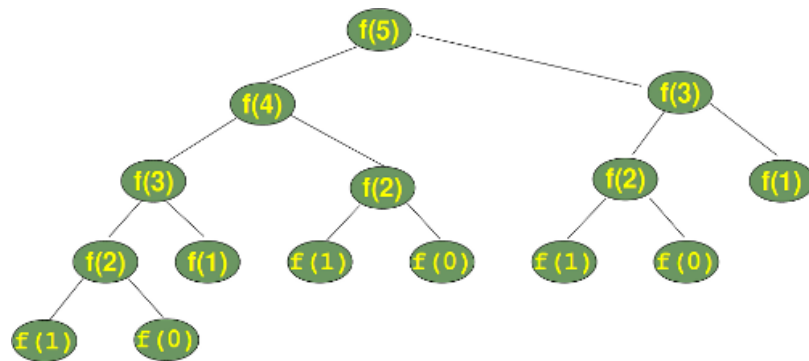
Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Printing Fibonacci series using recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
n=int(input("enter number of terms in sequence:"))
print("fibonacci sequence:")
for i in range(n):
    print(fib(i))

output:
enter number of terms in sequence:5
fibonacci sequence:
0
1
1
2
3
```

Let's have a look at the calculation tree, i.e. the order in which the `fib ()` function is called recursively.



Infinite Recursion

Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process. In fact, the Python virtual machine eventually runs out of memory resources to manage the process, so it halts execution with a message indicating a stack overflow error.

Recursive Function Definition

```
>>> def runForever(n):
    if n > 0:
        runForever(n)
    else:
        runForever(n - 1)
```

Recursive Function Call

```
>>> runForever(1)
```

The PVM keeps calling **runForever(1)** until there is no memory left to support another recursive call. Unlike an infinite loop, an infinite recursion eventually halts execution with an error message.

Costs and Benefits of Recursion:

Advantages of recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug

HIGHER-ORDER FUNCTIONS

A higher-order function is the one that expects a function and a set of data values as arguments. The argument function is applied to each data value, and a set of results or a single data value is returned.

FUNCTIONS AS FIRST-CLASS DATA OBJECTS

In Python, functions can be treated as first-class data objects. This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries

```
>>> f = abs
>>> f(-4)
4
>>> funcs = [abs, math.sqrt]
>>> funcs[1](2)
1.4142135623730951
```

MAPPING: The first type of useful higher-order function to consider is called a **mapping**. This process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results. Python includes a **map** function for this purpose.

Syntax:

map(function, list)

Example:

```
def addition(n):
    return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

FILTERING

A second type of higher-order function is called a **filtering**. In this process, a function called a **predicate** is applied to each value in a list. If the predicate returns **True**, the value passes the test and is added to a filter object (similar to a map object). Otherwise, the value is dropped from consideration.

Python has a **filter** function for this purpose whose syntax is as follows:

syntax:

filter(function, sequence)

Example:

```
seq = [0, 1, 2, 3, 5, 8, 13]
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

REDUCING

Another higher-order function is called a **reducing**. It takes a list of values and repeatedly applies the function that accumulates a single data value. A summation is a good example of this process. The first value is added to the second value, and then the sum is added to the third value, and so on, until the sum of all the values is produced.

The Python **functools** module includes a **reduce** function that expects a function and a list of values as its arguments. The **reduce** function returns the result of applying the function.

Syntax:

reduce(function, list)

Example:

```
import functools

lis = [ 1 , 3, 5, 6, 2, ]
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))
```

CREATING ANONYMOUS FUNCTIONS USING LAMBDA:

Python has a mechanism called **lambda** that allows the programmer to create functions in this manner. A **lambda** is an **anonymous function**. It has no name of its own, but it contains the names of its arguments as well as a single expression. When the **lambda** is applied to its arguments, its expression is evaluated, and its value is returned.

The syntax of a **lambda** is very tight and restrictive:

lambda <argname-1, ..., argname-n>: <expression>

Example:

```
>>> f=lambda x,y:x+y
>>> f(1,1)
2
```

The following piece of code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
>>> def f(x):return x**2

>>> print(f(8))
64
>>> g=lambda x:x**2
>>> print(g(8))
64
```

It is observed that f() and g() do exactly the same and can be used in the same ways. Note that the lambda definition does not include a "return" statement -- it always contains an expression which is returned.

Key points

- Lambda functions has no name
- Lambda functions can take any number of arguments.
- Lambda functions do not have an explicit return statement. But it always contains an expression which is return.
- Lambda functions cannot even access global variables
- We can pass lambda functions as arguments to other functions

MODULES

A module is a file containing Python definitions and statements. A module can define functions, classes, and variables and all these can be used in any application.

Modules are of two types:

1. Predefined modules
2. user-defined modules

PREDEFINED MODULES

Predefined modules are the built-in modules developed by the programmers and are updated in the python library. In order to use these modules, one has to use import statement or from import or from import *.

Example:

```
import math
| x1=5
  y1=4
  x2=3
  y2=2
  d= math.sqrt((x2-x1)**2+(y2-y1)**2)
  print(d)
```

output:
2.8284271247461903

Example:

```
from math import pi
print(pi)
```

output:
3.141592653589793

USER DEFINED MODULES

Modules that are created to meet the application needs are User Defined Modules. Every python program is a module that is every file that you save as **.py** extension is a module.

Creating a Module:

A file containing Python code is called a module and its module name would be filename. To create a module just save the code you want in a file with the file extension **.py**

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our commonly used functions in a module and import it, instead of copying their definitions into different programs.

Example

Sample.py

```
x=10
y=20
def show( ):
    print("hello")
```

basic.py

```
import sample
sample.show( )
print(sample.x)
print(sample.y)
```

output:

```
hello
10
20
```

Here, we are just accessing variables and functions from one program file to another program file. To work out with this type of accessing is that the both the files must be in of same folder or same directory otherwise an error generates

MODULE LOADING AND EXECUTION

A module imported in a program must be located and loaded into memory before it can be used.

- Python first searches for the module in the current working directory. If the module is not found there, it then looks for the module in the directories specified in the **PYTHONPATH** environment variable.
- If the module is still not found or if the **PYTHONPATH** variable is not defined, then a python installation-specific path (like c:\python34\lib) is searched. If the module is not located even there, then an error **IMPORT ERROR** exception is generated.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in `sys.path`. The search is in this order.

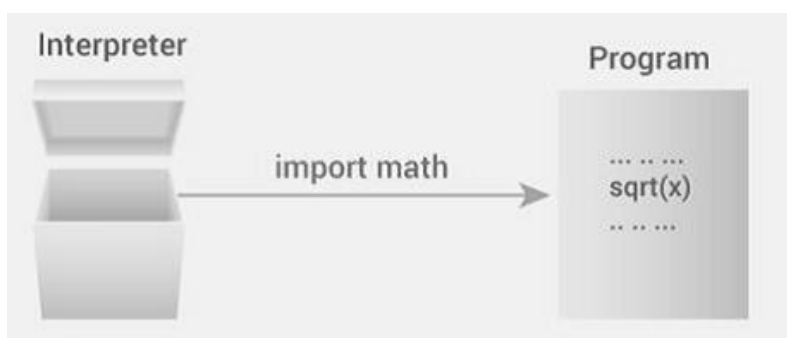
1. The current directory.
2. `PYTHONPATH` (an environment variable with a list of directories).
3. The installation-dependent default directory

IMPORT STATEMENT

One can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax:

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.



For example, to import the module `support.py`, you need to put the following command at the top of the script –

```
#importing module support
```

```
import support
```

```
#now you can call defined function that module as follows
```

```
support.print_func("zara")
```

When the above code is generated it produces the following the result:

```
Hello:zara
```

A module is loaded only once, regardless of the number of times it is imported.

RELOADING A MODULE

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works. Suppose we have the following code in a module named **my_module**.

```
#This module shows the effect of multiple imports and reload  
  
print("This code got executed")
```

Now we see the effect of multiple imports.

```
>>>import my_module  
This code got executed  
>>>import my_module  
>>>import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it. One way to do this is to restart the interpreter. But this does not help much. Python provides a neat way of doing this. We can use the `reload ()` function inside the `imp` module to reload a module.

```
>>>import imp  
>>>import my_module  
This code got executed  
>>>import my_module  
>>>imp.reload(my_module)  
This code executed  
<module 'my_module' from './my_module.py'>
```

The from.....import statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function `fibonacci` from the module `fib`, use the following statement –

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib`.

The from....import * statement

It is also possible to import all names from a module into the current namespace by using the following import statement –

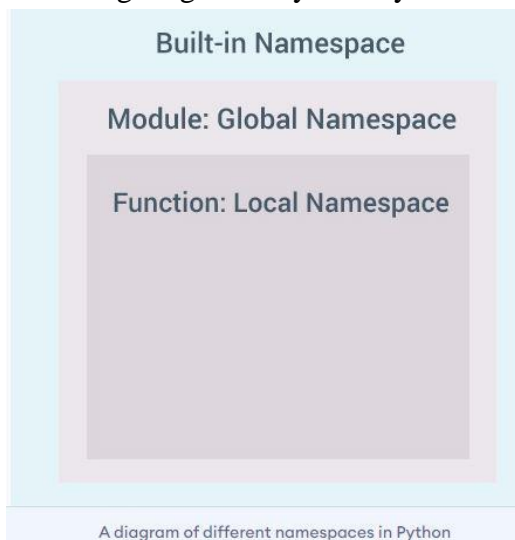
```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace

MANAGING NAMESPACES IN PYTHON

- A namespace is a collection of names. In Python, we can imagine a namespace as a mapping of every name you have defined to corresponding objects. Different namespaces can exist at a given time but are completely isolated.
- A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs. This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program.
- Each module creates its own global namespace. These different namespaces are isolated. Hence, the same name may exist in different modules those do not collide.
- Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class.

Following diagram may show you how the different name space isolated from one another:



Python Variable Scope:

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play. A scope is the portion of a program from where a namespace can be accessed directly without any prefix. At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace. If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python

```
def outer_function():  
    b = 20  
    def inner_func():  
        c = 30  
  
a = 10
```

Here, the variable `a` is in the global namespace. Variable `b` is in the local namespace of `outer_function()` and `c` is in the nested local namespace of `inner_function()`.

1. When we are in `inner_function()`, `c` is local to us, `b` is nonlocal and `a` is global.
2. We can read as well as assign new values to `c` from `inner_function()`.
3. If we try to assign a value to `b`, a new variable `b` is created in the local namespace which is different than the nonlocal `b`.
4. The same thing happens when we assign a value to `a`.

Another Example:

```
def outer_function():  
    a = 20  
    def inner_function():  
        a = 30  
        print('a =', a)  
    inner_function()  
    print('a =', a)  
  
a = 10  
outer_function()  
print('a =', a)
```

In this program, three different variables `a` are defined in separate namespaces and accessed accordingly.

Output:

```
a = 30  
a = 20  
a = 10
```

Another Example:

In the following program, all references and assignments are to the global `a` due to the use of keyword `global`.

```
def outer_function():
    global a
    a = 20
    def inner_function():
        global a
        a = 30
        print('a =', a)
    inner_function()
    print('a =', a)

a = 10
outer_function()
print('a =', a)
```

Output:

```
a = 30
a = 30
a = 30
```

PYTHON STANDARD MODULES

Python's standard library is very extensive, offering a wide range of functionalities. In this context, we will see how to use some of Python 3's standard modules such as the statistics module, the math module and the random module.

STATISTICS MODULE

This module, as mentioned in the Python 3 documentation, provides functions for calculating mathematical statistics of numeric (Real-valued) data. Some of the most commonly used functions of the module are:

1. `mean()`: Arithmetic mean ('average') of data.

```
>>>import statistics
>>>A = [5,10,6,21,5,17,14,8,3,9]
>>>mean = statistics.mean(A)
>>> print('The mean of A is ', mean)
```

2. `median()`: Median (middle value) of data.

```
>>>import statistics
>>>B = [1,7,13,24,35,37,42,44,53,55]
>>>median = statistics.median(B)
>>>print('Median of B is:', median)
```

3. median_low(): Low median of the data is the lower of two middle values when the number of data elements is even, else it is the middle value.

```
>>> import statistics
>>> B = [1,7,13,24,35,37,42,44,53,55]
>>> low_median = statistics.median_low(B)
>>> print('Low Median of B is:', low_median)
```

4. median_high(): High median of the data is the larger of two middle values when the number of data elements is even, else it is the middle value.

```
>>> import statistics
>>> B = [1,7,13,24,35,37,42,44,53,55]
>>> high_median = statistics.median_high(B)
>>> print('High Median of B is:', high_median)
```

5. mode(): Mode (most common value) of discrete data

```
>>> import statistics
>>> C = [1,1,2,2,2,2,4,4]
>>> mode_value = statistics.mode(C)
>>> print('The most common item of C is:', mode_value)
```

6. variance(): Return the sample variance of data

```
>>> import statistics
>>> A = [5,10,6,21,13,17,14,8,3,9]
>>> variance = statistics.variance(A)
>>> print('The variance of A is:', variance)
```

7. stdev(): Return the sample standard deviation (the square root of the sample variance).

```
>>> import statistics
>>> A = [5,10,6,21,13,17,14,8,3,9]
>>> std = statistics.stdev(A)
>>> print('The std of A is:', std)
```

MATH MODULE

This module provides access to the mathematical functions. Some of the most commonly used functions of the module are:

1. sqrt(x): Return the square root of x.

```
import math
x = 81
print('The square root of', x, 'is', math.sqrt(x))
```

2. ceil(x): Return the ceiling of x, the smallest integer greater than or equal to x

```
import math
x = 5.6
print('The ceil of', x, 'is', math.ceil(x))
```

3. floor(x): Return the floor of x, the largest integer less than or equal to x

```
import math
x = 5.6
print('The floor of', x, 'is', math.floor(x))
```

4. factorial(x): Return x factorial

```
import math
x = 6
print('The factorial of', x, 'is:', math.factorial(x))
```

5. exp(x): Return e raised to the power x, where $e = 2.718281\dots$ is the base of natural logarithms.

```
import math
x = 2
print('e ^', x, '=', math.exp(x))
```

6. pow(x,y): Return x raised to the power y.

```
import math
base = 2
x = 4
print(math.pow(base, x))
```

7. log(x, base): With one argument, return the natural logarithm of x (to base e). With two arguments, return the logarithm of x to the given base.

```
import math
base = 2
x = 16
print(math.log(x, base))
```

RANDOM MODULE

This module, as mentioned in the Python 3's documentation, implements pseudo random number generators for various distributions.

1. randint(a, b): Return a random integer x, which belongs to the range [a,b].

```
import random
result = random.randint(1, 11)
print('The random integer number in range [1,11] is:', result)
```

2. choice(seq): Return a random element from the non-empty sequence.

```
import random
coin_side = random.choice(['H','T'])
print('Flip a coin simulation using choice function. Result:', coin_side)
```

3. random(): Returns a random number between 0 and 1

```
import random
print(random.random())
```

4. randrange(): Returns a randomly selected element from the specified range.

```
import random
print(random.randrange(3, 9))
```

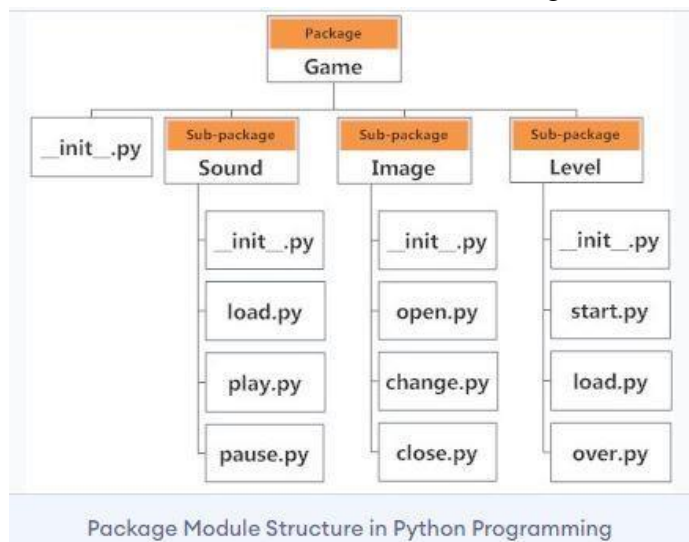
PACKAGES

We don't usually store all of our files on our computer in the same location. We use a well organized hierarchy of directories for easier access. Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory.

Analogous to this, Python has packages for directories and modules for files. As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear. Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



IMPORTING MODULE FROM A PACKAGE

- We can import modules from packages using the dot (.) operator. For example, if we want to import the start module in the above example, it can be done as follows:

`import Game.Level.start`

- Now, if this module contains a function named `select_difficulty()`, we must use the full name to reference it.

`Game.Level.start.select_difficulty(2)`

- If this construct seems lengthy, we can import the module without the package prefix as follows:

`from Game.Level import start`

- We can now call the function simply as follows:

start.select_difficulty(2)

- Another way of importing just the required function (or class or variable) from a module within a package would be as follows:

from Game.Level.start import select_difficulty

- Now we can directly call this function.

select_difficulty(2)

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding. While importing packages, Python looks in the list of directories defined in **sys.path**, similar as for module search path.

Installing packages via PIP

- Pip is a tool for installing python packages from the python package index.
- Python virtual environments allow python to be installed in an isolated location for a particular application, rather than being installed globally.
- In python, many third-parties are stored in **python package index** (PyPI). To install them, you can use a program called `_pip`. However, new versions of python have these modules installed by default. Once you have these modules, installing libraries from PyPI becomes very easy. Simply, goto the command line (for windows it will be the command prompt), enter the following code:

pip install package_name

- Once the library is installed, import it into your program and use it in your code

UNIT - IV

READING CONFIG FILES IN PYTHON

- A **configuration** is the way a system is set up, or the assortment of components that make up the system.
- Configuration **files** (commonly known simply as **config files**) are files used to configure the parameters and initial settings for some computer programs. They are used for user applications, server processes and operating system settings.
- Configuration files written in the common .ini configuration file format.
- **In python, configparser** module can be used to read configuration files.

Consider the below sample configuration file:

```
#Sample configuration file

[installation]
prefix = /usr/local
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin

# Setting related to debug configuration
[debug]
pid-file = /tmp/spam.pid
show_warnings = False
log_errors = true

[server]
nworkers: 32
port: 8080
root = /www/root
signature:
```

Code for reading the configuration file is as follows:

```
from configparser import ConfigParser

configur = ConfigParser() # creating the object of type ConfigParser
print (configur.read('config.ini')) #reading the configuration file config.ini

print ("Sections : ", configur.sections())
print ("Installation Library : ", configur.get('installation','library'))
print ("Log Errors debugged ? : ", configur.getboolean('debug','log_errors'))
print ("Port Server : ", configur.getint('server','port'))
print ("Worker Server : ", configur.getint('server','nworkers'))
```

Output:

```
Sections : ['installation', 'debug', 'server']
Installation Library : '/usr/local/lib'
Log Errors debugged ? : True
Port Server : 8080
Worker Server : 32
```

WRITING LOG FILES IN PYTHON

Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging and running. If you don't have any logging record and your program crashes, there are very little chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time.

With logging, you can have the trace of events so that if something goes wrong, we can determine the cause of the problem.

Some developers use the concept of printing the statements to validate if the statements are executed correctly or some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but for complex scripts, printing approach will fail.

Python has a built-in module logging which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

Levels of Log Message:

Following are the five built – in levels

- **Debug:** These are used to give detailed information when diagnosing problems.
- **Info :** These are used to Confirm that things are working as expected
- **Warning :** These are used an indication that something unexpected happened, or indicative of some problem in the near future
- **Error :** This tells that due to a more serious problem, the software has not been able to perform some function
- **Critical :** This tells serious error, indicating that the program itself may be unable to continue running

Each built-in level has been assigned its numeric value.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

In Python, **Logging** module is packed with several features. It has several constants, classes, and methods.

- `Logger.debug(msg)`: This will log a message with level DEBUG on this logger
- `Logger.info(msg)` : This will log a message with level INFO on this logger.

- `Logger.warning(msg)` : This will log a message with level **WARNING** on this logger.
- `Logger.error(msg)` : This will log a message with level **ERROR** on this logger.
- `Logger.critical(msg)` : This will log a message with level **CRITICAL** on this logger.
- `Logger.setLevel(level)` : This function sets the threshold of this logger to **level**. This means that all the messages below this level will be ignored.

To write log files:

1. Import the logging module from the library

```
import logging
```

2. Create and configure the logger

```
logging.basicConfig(filename="newfile.log", filemode='w')
```

The above statement specifies the name of the file to which the logged events are to be written, file opening mode.

```
logger=logging.getLogger()
```

The above statement creates the logger object.

3. Set the level of the logger

```
logger.setLevel(logging.WARNING)
```

The above statement sets the level of logger messages to the **WARNING** level. Since **WARNING** Level is set, then any messages that are below this level are not logged.

4. Finally. write the events tracked to the file

```
import logging
```

```
logging.basicConfig(filename="newfile.log", filemode='w')
```

```
logger=logging.getLogger()
```

```
logger.setLevel(logging.DEBUG)
```

```
logger.debug("Harmless debug Message")
```

```
logger.info("Just an information")
```

```
logger.warning("Its a Warning")
```

```
logger.error("Did you try to divide by zero")
```

```
logger.critical("Internet is down")
```

FILE HANDLING IN PYTHON

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. File handling operations are easy and short.

Python treats file differently as text or binary.

Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like newline character. It ends the current line and tells the interpreter a new one has begun.

open() Function

We use **open ()** function in Python to open a file in read or write mode. **open ()** will return a file object, which is called file handle. To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write.

Syntax:

open(filename, mode).

There are three kinds of mode, that Python provides and how files can be opened:

- “ **r** “, for reading.
- “ **w** “, for writing.
- “ **a** “, for appending.
- “ **r+** “, for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “ **r** ” by default.

Example:

```
file = open('sample.txt', 'r')
for line in file:
    print (line)
```

The open command will open the file in the read mode and the **for** loop will print each line present in the file.

read()

- This function is used to read all the characters of a file as a single string.

Example:

```
file = open("file.text", "r")
print (file.read())
```

- The same read() function is used to read certain number of characters in a given file

Example:

```
file = open("file.txt", "r")
print (file.read(5))
```

Above is the code that the interpreter will read the first five characters of stored data and return it as a string.

readline()

readline() function reads a line of the file and return it in the form of the string. It takes a parameter **n**, which specifies the maximum number of bytes that will be read. However, it does not read more than one line, even if **n** exceeds the length of the line.

It will be efficient when reading a large file because instead of fetching all the data in one go, it fetches line by line. This function returns the line of the file which contains a newline character in the end. If the end of the file is reached, it will return an empty string.

Example:

```
file1 = open('myfile.txt', 'r')
while True:
    line = file1.readline()    # reads next line from file
    if not line:              # if end of file is reached
        break
    print(line)
file1.close()
```

Another Example:

```
file1 = open('myfile.txt', 'r')
for line in file1:
    print(line)
file1.close()
```

An **iterable** object is returned by `open()` function while opening a file. This way of reading in a file line-by-line includes iterating over a file object in **for** loop. Doing this we are taking advantage of a built-in Python function that allows us to iterate over the file object using **for** loop.

readlines()

readlines() is used to read all the lines at a single go and then return them as each line a string element in a list. This function can be used for small files, as it reads the whole file content, then split it into separate lines. We can iterate over the list and strip the newline ‘\n’ character using **strip()** function.

Example:

```
file1 = open('bdays.py', 'r')
Lines = file1.readlines()
for line in Lines:
    print(line.strip())    #printing each line
```

write ()

The function **write ()** is used to perform the write operation on the files.

Example:

```
file = open('sample.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The above example code opens the file `sample.txt` and performs write operation on the file. This write operation will remove all the contents of that file. If the given file name is not there then it creates a new file and write the contents into the file.

Below is the python code block that appends a line of text to the file **sample.txt**:

```
file = open ('sample.txt','a')
file.write ("This will add this line")
file.close ( )
```

writelines()

The **writelines ()** method writes the items of a list or multiple lines to the file. Where the texts will be inserted depends on the file mode and current file pointer position.

"a": The texts will be inserted at the current file pointer position i.e. at the end of the file.

"w": The file will be emptied before the texts will be inserted at the current file pointer position, i.e. 0.

Example:

```
L = ["Good Morning\n", "Hope You are doing Good\n", "Get Well Soon\n", "See You Soon..!\n"]
```

writing to file

```
file1 = open('myfile.txt', 'w')
```

```
file1.writelines(L)
```

```
file1.close()
```

Manipulating File Pointer using seek():

In Python, seek() function is used to **change the position of the File Handle** to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax:

f.seek(offset, from_what), where f is file pointer

Parameters:

Offset: Number of positions to move forward

from_what: It defines point of reference.

The reference point is selected by the **from_what** argument. It accepts three values:

- **0:** sets the reference point at the beginning of the file
- **1:** sets the reference point at the current file position
- **2:** sets the reference point at the end of the file

By default **from_what** argument is set to 0.

Example: Let's suppose we have to read a file named "myfile.txt" which contains the following text:

"Code is like humor. When you have to explain it, it's bad."

Python program to demonstrate seek() method

```
f = open("myfile.txt", "r")
```

```
# Sets the position of the file handle to index position 20 from the beginning of the file
```

```
f.seek(20,0)
```

```
print(f.tell()) # prints current position
```

```
print(f.readline())
```

```
f.close()
```

This block of code reads the characters from the file starting from the index position 20 to the end of the file

Example 2: Seek() function with negative offset

Let's suppose the binary file contains the following text.

'Code is like humor. When you have to explain it, its bad.'


```
f = open("data.txt", "rb")
# sets Reference point to tenth position from end of the file
f.seek(-10, 2)
# prints current position
print(f.tell())
print(f.readline())
f.close()
```

The above given block of code reads the last ten characters of the file

Output:

, its bad

OBJECT ORIENTED PROGRAMMING

CLASSES AND OBJECTS

- A class is a user-defined data type from which objects are created. Classes provide a means of encapsulating data and functionality together. Each class instance can have attributes attached to it for maintaining its state. It can also have methods (defined by their class) for modifying their state.
- An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

The syntax of a simple class definition is the following:

```
class <class name>(<parent class name>):  
    <method definition-1>  
    ...  
    <method definition-n>
```

The class definition syntax has two parts: a class header and a set of method definitions that follow the class header. The class header consists of the class name and the parent class name.

The code for the **Student** class follows:

```
lass Student(object):
```

```
    def __init__(self, name, number):  
        self.name = name  
        self.scores = [ ]  
        for count in range(number):  
            self.scores.append(0)  
    def getName(self):  
        return self.name  
    def setScore(self, i, score):  
        self.scores[i - 1] = score
```

```

def getScore(self, i):
    return self.scores[i - 1]
def getAverage(self):
    return sum(self.scores) / len(self.scores)
def getHighScore(self):
    return max(self.scores)

```

Method Definitions

All of the method definitions are indented below the class header. Each method definition must include a first parameter named **self**, even if that method seems to expect no arguments when called. Otherwise, methods behave just like functions.

- Methods that allow a user to access but not change the state of an object are called **accessors**.
- Methods that allow a user to modify an object's state are called **mutators**.

Example for Mutator Method:

```

def setScore(self, i, score):
    self.scores[i - 1] = score

```

Example for Accessor Method:

```

def getScore(self, i):
    return self.scores[i - 1]

```

Declaring an Object:

For the class Student, an object can be declared as follows:

```
s = Student("Juan", 5)
```

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of objects.

Class and Instance Variables

- Instance variables are for data unique to each instance
- Class variables are for attributes and methods shared by all instances of the class.
- Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Example Program:

```
class Dog:
    animal = 'dog'          # Class Variable
    def __init__(self, breed, color):
        self.breed = breed
        self.color = color

Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class name also

print("\nAccessing class variable using class name")
print(Dog.animal)
```

CONSTRUCTOR:

Most classes include a special method named `__init__`. This method must begin and end with two consecutive underscores. This method is also called the class's **constructor**, because it is run automatically when a user instantiates the class. The purpose of the constructor is to initialize an individual object's attributes.

Here is the example code for this method in the **Student** class:

class Student(object):

```
def __init__(self, name, number):  
    self.name = name  
    self.scores = [ ]  
    for count in range(number):  
        self.scores.append(0)
```

Thus, when the code segment

```
s = Student("Juan", 5)
```

is run, Python automatically runs the constructor or `__init__` method of the **Student** class.

In the given example, in addition to **self**, the **Student** constructor expects two arguments that provide the initial values for these attributes. The attributes of an object are represented as **instance variables**. Each individual object has its own set of instance variables. These variables serve as storage for its state. All methods of the class can access the instance variables.

DESTRUCTORS

Destructors are called when an object gets destroyed. In Python, destructors are not needed because Python has a garbage collector that handles memory management automatically.

The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e. when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):  
    # body of destructor
```

A reference to objects is deleted when the object goes out of reference or when the program ends.

Example Program:

Class Employee:

```
def __init__(self):
    print('Employee created.')
def __del__(self):
    print('Destructor called, Employee deleted.')
```

```
obj = Employee()
```

```
del obj
```

Output:

Employee created.

Destructor called, Employee deleted.

INHERITANCE

Inheritance is the capability of one class to derive or inherit the properties from another class. It provides **reusability** of a code and there is no need to write the same code again and again. Moreover, it allows us to add more features to a class without modifying it.

In python, object is the root class of all classes in hierarchy:

Syntax to inherit a class:

```
Class Sub_Class_Name ( Parent_Class_Name):
```

```
<method definitons>
```

```
-----
```

```
-----
```

Below is the example program to illustrate the inheritance:

```
class Person(object):
```

```
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def isEmployee(self):
        return False
```

Inherited or Subclass (Note Person in bracket)

```
class Employee(Person):  
  
    def isEmployee(self):  
        return True
```

Driver code – main module of code

```
emp = Person("Vikram") # An Object of Person  
print(emp.getName())  
print(emp.isEmployee())
```

```
emp = Employee("Vinod") # An Object of Employee  
print(emp.getName())  
print(emp.isEmployee())
```

Output

```
Vikram  
False  
Vinod  
True
```

Calling Parent Class Constructor from the Sub class:

Constructor is the special method that can be used to initialize the data members of a class. In case of inheritance, at the time of creating object to the sub class, there is a need to call the constructor of the parent class to initialize the data members which are inherited by the sub class. Below is the example that illustrates how to call the parent class constructor from the sub class:

parent class

```
class Person( object ):  
    def __init__(self, name, idnumber):  
        self.name = name  
        self.idnumber = idnumber  
    def display(self):  
        print(self.name)  
        print(self.idnumber)
```

child class

```
class Employee( Person ):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post
        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)
```

Driver Code

```
a = Employee('Rahul', 886012, 200000, "Intern")
a.display()
```

Private members of parent class

When we want to restrict any of the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won't be available to the child class.

We can make an instance variable by adding double underscores before its name.

For example:

```
class C(object):
    def __init__(self):
        self.c = 21
        self.__d = 42 # d is private instance variable

class D(C):
    def __init__(self):
        self.e = 84
        C.__init__(self)
```

Driver Code

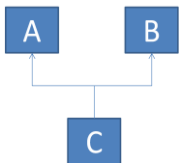
```
object1 = D( )
print(object1.d)      # produces an error as d is private instance variable
```


Different forms of Inheritance:

1. Single Inheritance: When a child class inherits from only one parent class, it is called single inheritance.



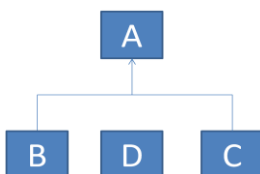
2. Multiple Inheritance: When a child class inherits from multiple parent classes, it is called multiple inheritance.



3. Multilevel Inheritance: When we have a grand parent and child relationship among the classes, then it is called multilevel inheritance.



4. Hierarchical Inheritance When there is more than one derived classes are created from a single base, and then it is called Hierarchical Inheritance.



5. Hybrid Inheritance: This form combines more than one form of inheritance. Basically, it is a mix of more than one type of inheritance.

POLYMORPHISM

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being used for different types and for different number of arguments.

Example:

```
#len() being used for a string
print(len("geeks"))
```

```
# len() being used for a list
print(len([10, 20, 30]))
```

Here, the same **len ()** is used in the context of a string and list

Another Example:

```
def add(x, y, z = 0):
    return x + y + z
```

Driver code

```
print(add(2, 3)) # here two arguments are passed
print(add(2, 3, 4)) # here three arguments are passed
```

Here **add ()** is the function that can accept different number of arguments in two different function calls.

Polymorphism with Inheritance:

Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

METHOD OVERRIDING:

Method overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

```
# Defining parent class
class Parent():

    def __init__(self):
        self.value = "Inside Parent"

    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    def __init__(self):
        self.value = "Inside Child"

    # Child's show method – overridden method
    def show(self):
        print(self.value)

# Driver's code
obj1 = Parent()
obj2 = Child()
obj1.show()
obj2.show()
```

Output:

```
Inside Parent
Inside Child
```

super () :

The **super ()** built-in returns a proxy object (temporary object of the super class) that allows us to access methods of the base class. It allows us to avoid using the base class name explicitly.

Example Program:

Parent Class

```
class Mammal(object):  
    def __init__(self, mammalName):  
        print(mammalName, 'is a warm-blooded animal.')  
    def show(self):  
        print("This is parent class show")
```

#Child Class

```
class Dog(Mammal):  
    def __init__(self):  
        print('Dog has four legs.')  
        super().__init__('Dog')  
        super().show()  
    def show(self):  
        print("This is child class show")  
        super().show()
```

Driver Code

```
d1 = Dog()  
d1.show()
```

Output:

```
Dog has four legs.  
Dog is a warm-blooded animal.  
This is parent class show  
This is child class show  
This is parent class show
```

OVERLOADING OPERATORS:

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

Operator overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments. A class can implement overloading operators by defining methods with special names called “Magic Methods” or “Special Methods”. These are the methods have name that begins and ends with double underscore.

Magic Methods for Different Operators

Operator	Method	Description
+	<code>__add__(self, other)</code>	Invoked for Addition Operations
-	<code>__sub__(self, other)</code>	Invoked for Subtraction Operations
*	<code>__mul__(self, other)</code>	Invoked for Multiplication Operations
/	<code>__truediv__(self, other)</code>	Invoked for Division Operations
//	<code>__floordiv__(self, other)</code>	Invoked for Floor Division Operations
%	<code>__mod__(self, other)</code>	Invoked for Modulus Operations
**	<code>__pow__(self, other[, modulo])</code>	Invoked for Power Operations
<<	<code>__lshift__(self, other)</code>	Invoked for Left-Shift Operations
>>	<code>__rshift__(self, other)</code>	Invoked for Right-Shift Operations
&	<code>__and__(self, other)</code>	Invoked for Binary AND Operations
^	<code>__xor__(self, other)</code>	Invoked for Binary Exclusive-OR Operations
	<code>__or__(self, other)</code>	Invoked for Binary OR Operations

Example Program:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

```
p1 = Point(1, 2)
p2 = Point(2, 3)
```

```
print(p1+p2)
```

Output:

```
(3, 5)
```

DYNAMIC ATTRIBUTES

- ❖ Dynamic attributes in Python are those attributes defined at runtime, after creating the objects or instances.

Consider the below example:

```
class Person:
    def __init__(self):
        self.employee = True

# Driver Code
e1 = Person()
e2 = Person()

e1.employee = False
e2.name = "Nikhil" # Dynamic Attribute

print(e1.employee) # Prints False
print(e2.employee) # Prints True
print(e2.name) # Prints Nikhil

print(e1.name) # this will raise an error as name is a dynamic attribute
               # created only for the e2 object
```

Dynamic Attributes are those attributes which are defined for the object but not as an attribute of the class.

❖ Another way to access and manipulate attributes is using the **getattr**, **setattr** and **delattr** functions.

- **setattr** () has three required parameters, the object, attribute name, and value.

Usage: **setattr**(c, "description", "My description")

c is the object

description is the name of the dynamic attribute

My description is its value

- **getattr** () has two required parameters, an object and the attribute name, and an optional default value. If the default value isn't provided unknown attributes will raise an **AttributeError**

Usage: **getattr** (c, "description", "No description")

c is the object

description is the name of the dynamic attribute

No description is error message if exception is raised

- When you want to delete any dynamic attribute, use the **delattr** () function. If the attribute doesn't exist an **AttributeError** will be raised.

Usage: **delattr** (object_name, attribute_name)

❖ In python, functions and methods are also treated as objects. So you can define dynamic attributes to functions just like in case of objects.

Example:

```
def value( ):
    return 10

# Dynamic attribute of a function
value.d1 = "This is Dynamic Attribute"
print(value.d1)
```

UNIT V

Errors and Exceptions: Syntax Errors, Exceptions, Handling Exceptions, Raising Exceptions, User defined Exceptions, Defining Clean-up Actions, Predefined Clean-up Actions

Graphical User Interfaces: The Behavior of Terminal Based Programs and GUI – Based, Programs, Coding Simple GUI-Based Programs, Other Useful GUI Resources

Programming: Introduction to Programming Concepts with Scratch

ERROS AND EXCEPTIONS

An Exception is a signal that an error or other unusual condition has occurred. Examples of exceptions are reading end of the file, divide by zero.

There are two kinds of errors:

1. Syntax Errors

Syntax errors occurred when the syntax rules to write the programming instructions are followed correctly. Due to these errors, program execution stops.

Example:

```
balance = 10000
```

```
if(balance<500)
```

```
    print("Your Account has no minimum balance")
```

This block of code generates a syntax error as a colon (:) character is not place for if statement

Example:

```
>>> Print("Hello, How are you?")
```

For the above statement Python raises a syntax error – Name Error: ‘Print’ is not defined

2. Exceptions

These are runtime errors. Exceptions occurred when exceptional situations occur in your program. For example, you tried to read a file but file not existed or deleted something accidentally while running programs. Such situations are handled using Exceptions.

Example:

```
S = input("Enter something")
```

When this statement is executed – instead of giving some input if you presses control + d, then it fails to read the input from keyboard. At this case, Python raises EOFError – means End of the File Error which it don’t expected.

Hence, exceptions are raised when unexpected events are occurred.

Examples:

```
>>> x = 5/0
```

The above statement raises **ZeroDivisionError**


```
>>>list = [1,2,3]
```

```
>>>print(list[3])
```

This statement raises **IndexError**: List index out of range

```
>>> list+2
```

This statement raises **TypeError**: One list is concatenated with another list only

```
>>> info = {"name": "Vijay", "place": "Chikmangalur"}
```

```
>>>print(info["job"])
```

This statement raises **KeyError**: 'job'

❖ EXCEPTION HANDLING

To handle exceptions, Python provides exception handling mechanism using **try...except** blocks.

Exceptions are common in Python Programming. Unknowingly, every module in Python Standard Library uses the exception handling mechanism to handle exceptions.

1. Accessing not existed element in the dictionary raises **KeyError** Exception
2. Searching for an non – existent element in the list generates **ValueError** Exception
3. Calling a non – existent method will raise an **AttributeError** Exception
4. Mixing data types will give a possibility of **TypeError** Exception
5. And many

In each of these cases, when an error occurred, exception message was printed and the program execution is stopped. These are unhandled exceptions. It is required to handle the exceptions so that the python program cannot stop in the middle of the program execution. If the exception is handled and resolved, the program continues its execution without any loss due to exception.

Below is the example that illustrates handling exceptions using **try...except** blocks:

```
try:
```

```
    n = int(input("Enter Numerator"))
```

```
    m = int(input("Enter Denominator"))
```

```
    k= n/m
```

```
    print(k)
```

```
except:
```

```
    print("An error occurs")
```

```
print("Program Continues execution")
```

Output: Run 1

Enter Numerator 10
Enter Denominator 2
5.0
Program continues execution

Output: Run 2

Enter Numerator ^c
An error occurs
Program continues execution

Some of the common built-in exceptions are:

Exception	When it is occurred
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program run out of memory.
TypeError	It occurs when a function and operation is applied in an incorrect type.

WRITING YOUR OWN EXCEPTIONS or USER DEFINED EXCEPTIONS

In addition to built in exceptions, Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class.

Writing own exception is illustrated with an example:

```
class ShortInputException(Exception):
    def __init__(self, value):
        self.value = value
    try:
        s=input("Enter Some thing")
        k=len(s)
        if(k<5):
            raise(ShortInputException(k))
    except ShortInputException as error:
        print('A New Exception Occurred : Message is Short of only', error.value, ' characters')
        print('Input Atleast 5 Characters')
    else:
        print("No Exception is Raised ")
```

Output Run 1:

Enter Something

Hai

A New Exception Occurred: Message is Short of Only 3 Characters

DEFINING CLEAN UP ACTIONS

The “try” statement provides very useful optional clause i.e. finally clause which is meant for defining ‘clean-up actions’ that must be executed under any circumstances.

A finally clause statement is always executed whether an exception is occurred or not.

Example:

```
def divide(x, y):  
    try:  
        result = x/y  
    except ZeroDivisionError:  
        print("Division By Zero Error!.....So Program Halted")  
    else:  
        print("Result is", result)  
    finally:  
        print("Executing Finally Clause")
```

```
>>> divide (2,1)
```

Result is 2.0

Executing Finally Clause

```
>>> divide (2,0)
```

Division By Zero Error!.....So Program Halted

Executing Finally Clause

Another Example:

Below is the example to raise an exception by making a file read-only and try to write onto it, thus causing it to raise an exception.

```
file = open('finally.txt', 'r')  
try:  
    file.write("Testing1 2 3.")  
    print("Writing to file.")  
except IOError:  
    print("Could not write to file.")  
else:  
    print("Write successful.")  
finally:  
    file.close()  
    print("File closed.")
```

Above program will give an output, something like –

Could not write to file.

File closed

PREDEFINED CLEAN UP ACTIONS

Some Objects define the standard clean up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed.

Example:

```
for line in open("myfile.txt"):
    print(line)
```

This is the code that prints the contents of the file. The problem with this code is that it leaves the file open after this part of the code has finished its executing. This is not issue in small applications but it can be a problem for larger applications.

To avoid this kind of problems, **with** statement is used such that after completion of the using object then it cleaned up promptly and correctly

Example:

```
with open(myfile.txt) as f:
    for line in f:
        print(line)
```

GRAPHICAL USER INTERFACE (OR) GUI PROGRAMMING

A Graphical User Interface (GUI) displays text as well as small images (called icons) that represent objects such as folders, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select some of these icons with a pointing device, such as a mouse, and move them around on the display. Commands can be activated by pressing the enter key or control keys, by pressing a command button, by selecting a drop-down menu item, or by double-clicking on some icons with the mouse. Put more simply, a GUI displays all information, including text, graphically to its users and allows them to manipulate this information directly with a pointing device.

BEHAVIOR OF TERMINAL-BASED PROGRAMS AND GUI-BASED PROGRAMS

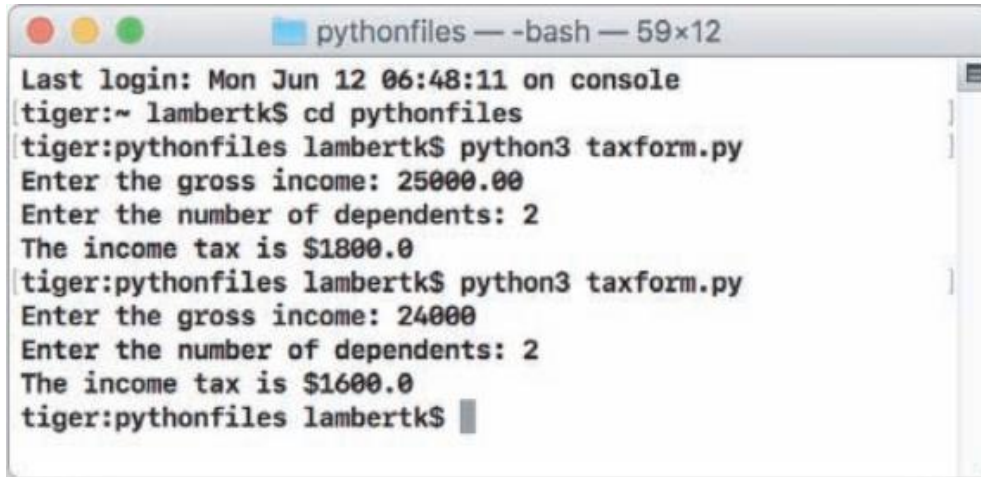
- A GUI program is event driven, meaning that it is inactive until the user clicks a button or selects a menu option. In contrast, a terminal-based program maintains constant control over the interactions with the user.
- A terminal-based program prompts users to enter successive inputs, whereas a GUI program puts users in charge, allowing them to enter inputs in any order and waiting for them to press a command button or select a menu option.

To have a clear difference on look and behavior between these modes of programming, an example program is given below:

This program computes the income tax of a person for the given two inputs – gross income and the number of dependents.

Terminal based Version:

The terminal-based version of the program prompts the user for his gross income and number of dependents. After he enters his inputs, the program responds by computing and displaying his income tax. The program then terminates execution. A sample session with this program is shown in Figure:



```
pythonfiles — -bash — 59x12
Last login: Mon Jun 12 06:48:11 on console
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 25000.00
Enter the number of dependents: 2
The income tax is $1800.0
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 24000
Enter the number of dependents: 2
The income tax is $1600.0
tiger:pythonfiles lambertk$
```

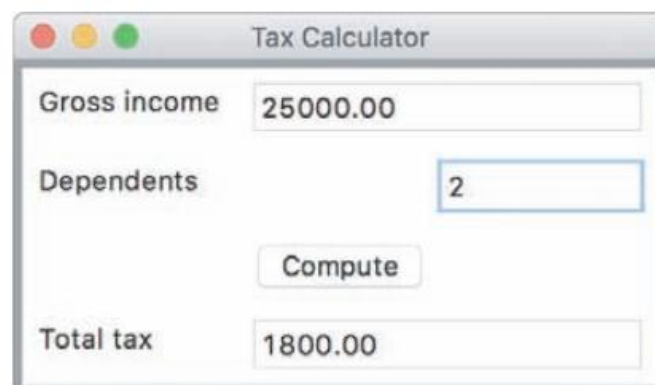
In this terminal based interface, once the user has given the input there is no way to go back to change it and enter again. To get the results for the different inputs, the program has to be executed again and the inputs are re – entered.

These can be solved by converting the interface to GUI.

GUI Based Version

The GUI-based version of the program displays a **window** that contains various components, also called **widgets**. These components includes text fields, command buttons, radio buttons, labels, menus, pop – up message boxes, prompt boxes, frames, colors, etc

For the same tax calculation program one can develop GUI interface may be as follows:



In this interface, user can input in any order and can also change, after clicking the **compute** button, the window is responded with **tax** amount. And the user can try different set of inputs, without running the program again.

This effective mode of interactive programming, GUI based programming is preferred.

EVENT – DRIVEN PROGRAMMING

In case of terminal based programs, user is prompted with to enter the inputs, after entering the inputs, program process them and gives the outputs.

But in case of GUI based programs, a window is opened and waits for the user to manipulate the window components like mouse clicks, keyboard strokes. These user events trigger the program to respond by taking the given inputs, process them and display the results. And this type of programming is called **Event Driven Programming**.

An event-driven program is developed in several steps. Initially, the types of window components and their arrangement in the window are determined. And then determine how one component is interacted with another component in the window. Once the interactions among these resources have been determined, their coding can begin. It consists of several steps:

1. Define a new class to represent the main application window.
2. Instantiate the classes of window components needed for this application, such as labels, fields, and command buttons.
3. Position these components in the window.
4. Register a method with each window component in which an event relevant to the application might occur.
5. Define these methods to handle the events.
6. Define a **main** function that instantiates the window class and runs the appropriate method to launch the GUI.

CODING SIMPLE GUI BASED PROGRAMS

- Below is the small program defines a class for a main window that displays a greeting “Hello World!”. This program uses open source module called **breezypythongui** available at <http://home.wlu.edu/~lambertk/breezypythongui/>

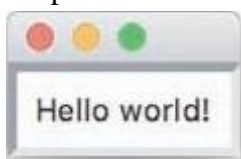
```
from breezypythongui import EasyFrame
```

```
class LabelDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self)
        self.addLabel(text = "Hello world!", row = 0, column = 0)

def main():
    LabelDemo().mainloop()

# Driver Code
if __name__ == "__main__":
    main()
```

Output:



❖ **This program performs the following steps:**

1. Import the **EasyFrame** class from the **breezypythongui** module. This class is a subclass of **tkinter**'s **Frame** class, which represents a top-level window.
2. Define the **LabelDemo** class as a subclass of **EasyFrame**. The **LabelDemo** class describes the window's layout and functionality for this application.
3. Define an **__init__** method in the **LabelDemo** class. This method is automatically run when the window is created. The **__init__** method runs a method with the same name on the **EasyFrame** class.

In this case, the **addLabel** method is run on the window itself. The **addLabel** method creates a window component, a **label object** with the text "Hello world!," and adds it to the window at the grid position (0, 0).

4. The last few lines of code define a **main** function and check to see if the Python code file is being run as a program. If this is true, the **main** function is called to create an instance of the **LabelDemo** class. The **mainloop** method is then run on this object. At this point, the window pops up for viewing.

The **mainloop**, as the name implies, enters a loop. The Python Virtual Machine runs this loop behind the scenes. Its purpose is to wait for user events. The loop terminates when the user clicks the window's close box.

❖ **Template for All GUI Programs**

```
from breezypythongui import EasyFrame
```

Other imports if any

```
class ApplicationName(EasyFrame):
```

The __init__ method definition

Definitions of event handling methods

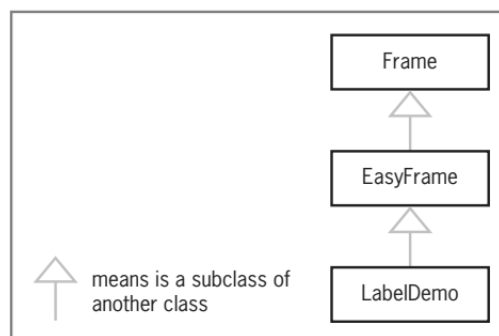
```
def main():
```

ApplicationName().mainloop()

```
if __name__ == "__main__":
```

```
    main()
```

❖ **Below is the class diagram for above program:**



WINDOWS AND WINDOW COMPONENTS

Window is root component of all graphical components such as input text fields, radio buttons, check boxes, menus, numerical fields, etc. A window has a layout which determines how the graphical components are arranged in the window.

Windows and Their Attributes

A window has several attributes. The most important ones are its

- title (an empty string by default)
- width and height in pixels
- resizable (true by default)
- background color (white by default)

The window's initial dimensions are automatically established. We can override the window's default title, an empty string, by supplying another string as an optional **title** argument to the **EasyFrame** method **__init__**. Other options are to provide a custom initial width and height in pixels.

Example: **EasyFrame.__init__(self, width = 300, height = 200, title = "Label Demo")**

A window component is created using class **EasyFrame** which is subclass of **tkinter's Frame** class. This class includes the following methods:

setBackground(color)	Sets the window's background color to color .
setResizable(aBoolean)	Makes the window resizable (True) or not (False).
setSize(width, height)	Sets the window's width and height in pixels.
setTitle(title)	Sets the window's title to title .

Window Layout

Window components are laid out in the window's two-dimensional **grid layout**. The grid's rows and columns are numbered from the position (0, 0) in the upper left corner of the window. A window component's row and column position in the grid is specified when the component is added to the window.

Example Program:

```
class LayoutDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self)
        self.addLabel(text = "(0, 0)", row = 0, column = 0)
        self.addLabel(text = "(0, 1)", row = 0, column = 1)
        self.addLabel(text = "(1, 0)", row = 1, column = 0)
        self.addLabel(text = "(1, 1)", row = 1, column = 1)

def main():
    LayoutDemo().mainloop()

if __name__ == "__main__":
    main()
```

Output Screen:



As the window is shrink wrapped window around the components placed in the window, if it is maximized then the all the components are stick to upper left or north-west corners and the alignment of the components in the window's grid layout is also disturbed. To avoid these effects, following attributes are provided:

- **sticky**
This attribute has values N, S, E, W. Each represents the position of component in the window layout. One can use the combination of these values in a desire manner. Then the window component is stick to the specific position though the window size is stretched.
- **rowspan**
This attribute specifies number of grid positions a component has to be aligned horizontally.
- **columnspan**
This attribute specifies number of grid positions a component has to be aligned vertically.

Types of Window Components and Their Attributes

GUI programs use several types of window components. These include labels, entry fields, text areas, command buttons, drop-down menus, sliding scales, scrolling list boxes, canvases, and many others.

The **breezypythongui** module includes methods for adding each type of window component to a window. Each such method uses the form

`self.addComponentType(<arguments>)`

When this method is called, **breezypythongui**

- Creates an instance of the requested type of window component
- Initializes the component's attributes with default values or any values provided by the programmer
- Places the component in its grid position (the row and column are required arguments)
- Returns a reference to the component

The window components supported by **breezypythongui** such as **Label**, **Button**, **Scale**, **FloatField**, **TextArea**, and **EasyCanvas**. A complete list is shown below. Parent classes are shown in parentheses.

Label	Displays text or an image in the window.
IntegerField(Entry)	A box for input or output of integers.
FloatField(Entry)	A box for input or output of floating-point numbers.
TextField(Entry)	A box for input or output of a single line of text.

TextArea(Text)	A scrollable box for input or output of multiple lines of text.
EasyListbox(Listbox)	A scrollable box for the display and selection of a list of items.
Button	A clickable command area.
EasyCheckbutton(Checkbutton)	A labeled checkbox.
Radiobutton	A labeled disc that, when selected, deselects related radio buttons.
EasyRadiobuttonGroup(Frame)	Organizes a set of radio buttons, allowing only one at a time to be selected.
EasyMenuBar(Frame)	Organizes a set of menus.
EasyMenubutton(Menubutton)	A menu of drop-down command options.
EasyMenuItem	An option in a drop-down menu.
Scale	A labeled slider bar for selecting a value from arrange of values.
EasyCanvas(Canvas)	A rectangular area for drawing shapes or images.
EasyPanel(Frame)	A rectangular area with its own grid for organizing window components.
EasyDialog(simpleDialog.Dialog)	A resource for defining special-purpose popup windows.

DISPLAYING IMAGES

Below is the program that displays an image with a caption:

```
from breezypythongui import EasyFrame
from tkinter import PhotoImage
from tkinter.font import Font

#Class that displays the image and caption
class ImageDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Image Demo")
        self.setResizable(False);
        imageLabel = self.addLabel(text = "", row = 0, column = 0, sticky = "NSEW")
        textLabel = self.addLabel(text = "Smokey the cat", row = 1, column = 0, sticky =
                                "NSEW")

#Load the image and associate with the image label
self.image = PhotoImage(file = "smokey.gif")
imageLabel["image"] = self.image
```

This program adds two labels to the window. One label displays the image and the other label displays the caption. The image label is first added to the window with an empty text string. The program then creates a **PhotoImage** object from an image file and sets the **image** attribute of the image label to this object.

Below is the list that summarizes the **tkinter.Label** attributes

image	A PhotoImage object and it must be loaded from a GIF file.
text	A string.
background	Background color of the label text
foreground	A label's foreground is the color of its text.
font	A Font object (imported from tkinter.font).

COMMAND BUTTONS AND RESPONDING TO EVENTS

- A command button is added to a window by specifying its text and position in the grid. A button is centered in its grid position by default.
- The method **addButton** accomplishes all this and returns an object of type **tkinter.Button**.
- A button can also display an image, usually a small icon, instead of a string.
- A button also has a **state** attribute, which can be set to "normal" to enable the button (its default state) or "disabled" to disable it.
- To allow a program to respond to a button click, the programmer must set the button's **command** attribute. There are two ways to do this:
 - By supplying a keyword argument when the button is added to the window (or)
 - By assignment to the button's attribute

Example Program:

```
from breezypythongui import EasyFrame
```

```
class ButtonDemo(EasyFrame):
```

```
    def __init__(self):
        EasyFrame.__init__(self)
        self.label = self.addLabel(text = "Hello world!", row = 0, column = 0, columnspan = 2,
                                    sticky = "NSEW")
        self.clearBtn = self.addButton(text = "Clear", row = 1, column = 0,
                                        command=self.clear)
        self.restoreBtn = self.addButton(text = "Restore", row = 1, column = 1,
                                        state="disabled", command=self.restore)
```

#methods for handling events

```
    def clear(self):
        self.label["text"] = ""
        self.clearBtn["state"] = "disabled"
        self.restoreBtn["state"] = "normal"

    def restore(self):
        self.label["text"] = "Hello world!"
        self.clearBtn["state"] = "normal"
        self.restoreBtn["state"] = "disabled"
```

#main method

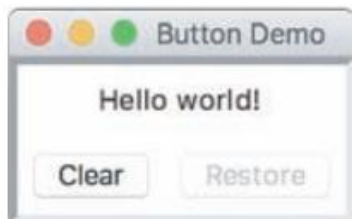
```
def main():
    ButtonDemo().mainloop()
```

#driver code

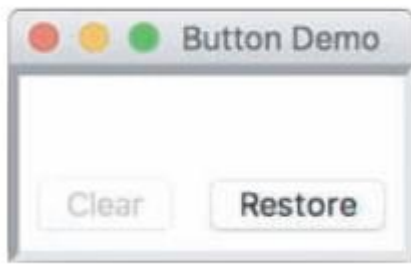
```
if __name__=="__main__":
    main()
```

OUTPUT:

On executing the above program, window will appears as follows:



After clicking the button **clear**, the program responds to mouse click event and invokes the method **clear()** because this method is assigned to the attribute **command**. Then the window will be displayed as follows:



Once **Restore** button is clicked, then “**Hello World!**” label is displayed again.

INPUT AND OUTPUT WITH ENTRY FIELDS

An entry field is a box in which the user can position the mouse cursor and enters line of text.

TEXT FIELD

- A text field is appropriate for entering or displaying a single-line string of characters.
- The programmer uses the method **addTextField** to add a text field to a window. The method returns an object of type **TextField**, which is a subclass of **tkinter.Entry**.
- Required arguments to **addTextField** are **text** (the string to be initially displayed), **row**, and **column**. Optional arguments are **rowspan**, **columnspan**, **sticky**, **width**, and **state**.
- A text field has a default width of 20 characters. This represents the maximum number of characters viewable in the box, but the user can continue typing or viewing them by moving the cursor key to the right.
- The programmer can set a text field’s **state** attribute to “readonly” to prevent the user from editing an output field.
- The method **getText** returns the string currently contained in a text field.
- The method **setText** outputs its string argument to a text field.

Example Program that converts the given input text to upper case letters:
from breezypythongui import EasyFrame

```
class TextFieldDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Text Field Demo")
        self.addLabel(text = "Input", row = 0, column = 0)
        self.inputField = self.addTextField(text = "",row = 0,column = 1)
        self.addLabel(text = "Output", row = 1, column = 0)
        self.outputField = self.addTextField(text = "",row = 1,column = 1, state = "readonly")
        self.addButton(text = "Convert", row = 2, column = 0, columnspan = 2, command =
                                                                    self.convert)

    def convert(self):
        text = self.inputField.getText()
        result = text.upper()
        self.outputField.setText(result)
```

```
def main():
    TextFieldDemo().mainloop()

if __name__=="__main__":
    main()
```

OUTPUT:



INTEGER AND FLOAT FIELDS FOR NUMERIC DATA:

- Text field is used for input and output of characters. If numbers are needed then it is required to convert from string format to number format. To avoid this and make it simple, Integer and Float Fields are provided
- **breezypythongui** includes two types of data fields, called **IntegerField** and **FloatField**, for the input and output of integers and floating-point numbers
- The methods **addIntegerField** and **addFloatField** to add these components to the window
- It has an attribute **value** used to supply the value to this field
- This value must be an integer for an integer field, but can be either an integer or a floating-point number for a float field.
- The default width of an integer field is 10 characters, whereas the default width of a float field is 20 characters.
- The method **addFloatField** allows an optional **precision** argument. It specifies the precision of the number displayed in the field.
- The methods **getNumber** and **setNumber** are used for the input and output of numbers with integer and float fields.

Example Program: Below is the program that inputs an integer and displays its square root

```
from breezypythongui import EasyFrame
```

```
import math
```

```
class NumberFieldDemo(EasyFrame):
```

```
    def __init__(self):
```

```
        EasyFrame.__init__(self, title = "Number Field Demo")
```

```
        self.addLabel(text = "An integer",row = 0, column = 0)
```

```
        self.inputField = self.addIntegerField(value = 0,row = 0, column = 1,width = 10)
```

```
        self.addLabel(text = "Square root",row = 1, column = 0)
```

```
        self.outputField = self.addFloatField(value = 0.0,row = 1,column = 1,width = 8,  
                                                precision = 2,state = "readonly")
```

```
        self.addButton(text = "Compute", row = 2, column = 0, columnspan = 2,  
                        command = self.computeSqrt)
```

```
    def computeSqrt(self):
```

```
        number = self.inputField.getNumber()
```

```
        result = math.sqrt(number)
```

```
        self.outputField.setNumber(result)
```

```
# Main method
```

```
def main():
```

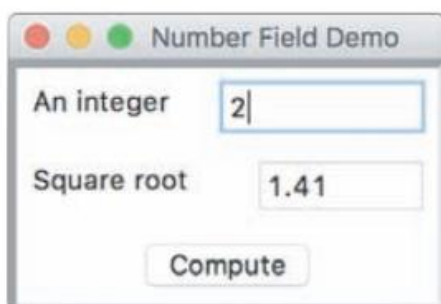
```
    NumberFieldDemo().mainloop()
```

```
#Driver code
```

```
if __name__=="__main__":
```

```
    main()
```

Output:



POP – UP MESSAGE BOXES

When errors arise in a GUI-based program, the program often responds by popping up a dialog window with an error message. Such errors are usually the result of invalid input data. The program detects the error, pops up the dialog to inform the user, and, when the user closes the dialog, continues to accept and check input data. In a terminal-based program, this process usually requires an explicit loop structure. In a GUI-based program, Python's implicit event-driven loop continues the process automatically.

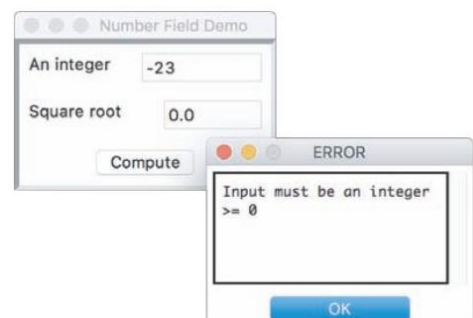
A message box can be added using the method `messageBox()` with two arguments: one is title and other is error message.

Example Program: Below is the program that inputs an integer and displays its square root and shows an error message box if an exception is raised.

```
from breezypythongui import EasyFrame
import math
class NumberFieldDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Number Field Demo")
        self.addLabel(text = "An integer",row = 0, column = 0)
        self.inputField = self.addIntegerField(value = 0,row = 0, column = 1,width = 10)
        self.addLabel(text = "Square root",row = 1, column = 0)
        self.outputField = self.addFloatField(value = 0.0,row = 1,column = 1,width = 8,
                                                precision = 2,state = "readonly")
        self.addButton(text = "Compute", row = 2, column = 0, columnspan = 2,
                        command = self.computeSqrt)

    def computeSqrt(self):
        try:
            number = self.inputField.getNumber()
            result = math.sqrt(number)
            self.outputField.setNumber(result)
        except:
            self.messageBox(title = "ERROR",message = "Input must be an integer >= 0")

def main():
    NumberFieldDemo().mainloop()
if __name__ == "__main__":
    main()
```



DEFINING AND USING INSTANCE VARIABLES

An **instance variable** is used to store data belonging to an individual object. Together, the values of an object's instance variables make up its **state**.

The state of a given window includes its title, background color, dimensions and other things. You A dictionary maintains these data within the window object.

The window class's **__init__** method establishes the initial state of a window object when it is created, and other methods within that class are run to access or modify this state.

Example: A simple counter application is shown defining and accessing the instance variable

```
from breezypythongui import EasyFrame
class CounterDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Counter Demo")
        self.setSize(200, 75)
        self.count = 0
        self.label = self.addLabel(text = "0", row = 0, column = 0, sticky = "NSEW",
                                    columnspan = 2)
        self.addButton(text = "Next",row = 1, column = 0,command = self.next)
        self.addButton(text = "Reset",row = 1, column = 1,command = self.reset)

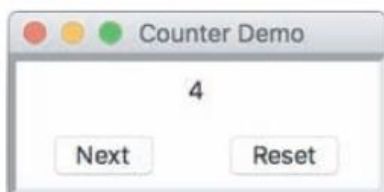
    def next(self):
        self.count += 1
        self.label["text"] = str(self.count)

    def reset(self):
        self.count = 0
        self.label["text"] = str(self.count)

def main():
    CounterDemo().mainloop()

if __name__=="__main__":
    main()
```

Output:



With every click on **Next** button the value is increment, if **Reset** button is clicked, then it becomes 0.

MULTI-LINE TEXT AREAS

- Text fields are useful for entering and displaying single lines of text, some applications need to display larger chunks of text with multiple lines. A text area widget allows the program to output and allows the user to input and edit multiple lines of text.
- The method **addTextArea** adds a text area to the window.
- The required arguments are the initial text to display, the row, and the column.
- Optional arguments include a width and height in columns (characters) and rows (lines), with defaults of 80 and 5, respectively.
- The optional argument is called **wrap**. This argument tells the text area what to do with a line of text when it reaches the right border of the viewable area. The default value of wrap is “none,” which causes a line of text to continue invisibly beyond the right border. The other values are “word” and “char,” which break a line at a word or a character, and continue the text on the next line.
- The **addTextArea** method returns an object of type **TextArea**, a subclass of **tkinter.Text**.
- This object has three important methods: **getText**, **setText**, and **appendText**.

Example:

```
#Add the text area to the window (in __init__() method)
```

```
self.outputArea = self.addTextArea("", row = 4, column = 0, columnspan = 2, width = 50, height=15)
```

CHECK BOXES

A **check button** consists of a label and a box that a user can select or deselect with the mouse. Check buttons often represent a group of several options, any number of which may be selected at the same time. The application program can either respond immediately when a check button is manipulated, or examine the state of the button at a later point in time.

- A check button is added to the window using **addCheckButton()** method and return an object of type **EasyCheckButton**
- **addCheckbutton** expects a **text** argument (the button’s label) and an optional **command** argument (a method to be triggered when the user checks or unchecks the button)
- The method **isChecked** returns **True** if the button is checked, or **False** otherwise.

Example Program:

```
from breezypythongui import EasyFrame
```

```
class CheckbuttonDemo(EasyFrame):
```

```
    def __init__(self):
```

```
        EasyFrame.__init__(self, "Check Button Demo")
```

```
        self.chickCB = self.addCheckbutton(text = "Chicken",row = 0, column = 0)
```

```
        self.taterCB = self.addCheckbutton(text = "French fries",row = 0, column = 1)
```

```
        self.beanCB = self.addCheckbutton(text = "Green beans",row = 1, column = 0)
```

```
        self.sauceCB = self.addCheckbutton(text = "Applesauce",row = 1, column = 1)
```

```
        self.addButton(text = "Place order", row = 2, column = 0,columnspan = 2,  
                        command = self.placeOrder)
```

```

def placeOrder(self):
    message = ""
    if self.chickCB.isChecked():
        message += "Chicken\n\n"
    if self.taterCB.isChecked():
        message += "French fries\n\n"
    if self.beanCB.isChecked():
        message += "Green beans\n\n"
    if self.sauceCB.isChecked():
        message += "Applesauce\n"
    if message == "":
        message = "No food ordered!"
    self.messageBox(title = "Customer Order",message = message)

def main():
    CheckbuttonDemo().mainloop()

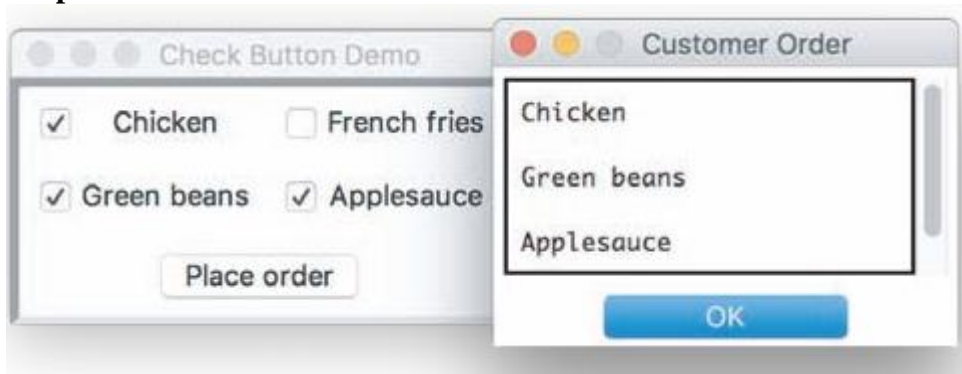
```

```

if __name__=="__main__":
    main()

```

Output:



RADIO BUTTONS

When the user must be restricted to one selection only in a group of options, then the set of options can be presented as a group of **radio buttons**. A radio button consists of a label and a control widget. One of the buttons is normally selected by default at program start-up. When the user selects a different button in the same group, the previously selected button automatically deselects.

- To add radio buttons to a window, the programmer first adds the radio button group to which these buttons will belong.
- The method **addRadiobuttonGroup** expects the grid coordinates as required arguments and text argument to give name of the group.
- The method **addRadioButton** method adds a radio button to the group
- The method **getSelectedButton** returns the currently selected radio button in a radio button group.
- The method **setSelectedButton** selects a radio button under program control.

To illustrate the use of radio buttons, consider a restaurant scenario, where a customer has two choices of meats, potatoes, and vegetables, and must choose exactly one of each food type

```
from breezypythongui import EasyFrame
```

```
class RadiobuttonDemo(EasyFrame):
```

```
    def __init__(self):
```

```
        EasyFrame.__init__(self, "Radio Button Demo")
```

```
        self.addLabel(text = "Meat", row = 0, column = 0)
```

```
        self.meatGroup = self.addRadiobuttonGroup(row = 1, column = 0, rowspan = 2)
```

```
        defaultRB = self.meatGroup.addRadiobutton(text = "Chicken")
```

```
        self.meatGroup.setSelectedButton(defaultRB)
```

```
        self.meatGroup.addRadiobutton(text = "Beef")
```

```
        self.addLabel(text = "Potato", row = 0, column = 1)
```

```
        self.taterGroup = self.addRadiobuttonGroup(row = 1, column = 1, rowspan = 2)
```

```
        defaultRB = self.taterGroup.addRadiobutton(text = "French fries")
```

```
        self.taterGroup.setSelectedButton(defaultRB)
```

```
        self.taterGroup.addRadiobutton(text = "Baked potato")
```

```
        self.addLabel(text = "Vegetable", row = 0, column = 2)
```

```
        self.vegGroup = self.addRadiobuttonGroup(row = 1, column = 2, rowspan = 2)
```

```
        defaultRB = self.vegGroup.addRadiobutton(text = "Applesauce")
```

```
        self.vegGroup.setSelectedButton(defaultRB)
```

```
        self.vegGroup.addRadiobutton(text = "Green beans")
```

```
        self.addButton(text = "Place order", row = 3, column = 0, columnspan = 3,  
                        command = self.placeOrder)
```

```
    def placeOrder(self):
```

```
        message = ""
```

```
        message += self.meatGroup.getSelectedButton()["text"] + "\n\n"
```

```
        message += self.taterGroup.getSelectedButton()["text"] + "\n\n"
```

```
        message += self.vegGroup.getSelectedButton()["text"]
```

```
        self.messageBox(title = "Customer Order", message = message)
```

```
def main():
```

```
    RadiobuttonDemo().mainloop()
```

```
if __name__ == "__main__":
```

```
    main()
```



KEYBOARD EVENTS

GUI-based programs can also respond to various keyboard events. The most common event is pressing the enter or return key when the mouse cursor has become the insertion point in an entry field. This event might signal the end of an input and a request for processing.

One can associate a keyboard event and an event-handling method with a widget by calling the **bind** method. This method expects a string containing a key event as its first argument, and the method to be triggered as its second argument. The string for the return key event is "**<Return>**". The event-handling method should have a single parameter named **event**. This parameter will automatically be bound to the event object that triggered the method.

Consider the square root program to allow the user to compute a result by pressing the return key while the insertion point is in the input field. You bind the keyboard **return** event to a handler for the **inputField** widget as follows:

```
self.inputField.bind("<Return>", lambda event: self.computeSqrt())
```

example:

```
from breezypythongui import EasyFrame
import math
```

```
class NumberFieldDemo(EasyFrame):
```

```
    def __init__(self):
        EasyFrame.__init__(self, title = "Number Field Demo")
        self.addLabel(text = "An integer",row = 0, column = 0)
        self.inputField = self.addIntegerField(value = 0,row = 0, column = 1,width = 10)
        self.addLabel(text = "Square root",row = 1, column = 0)
        self.outputField = self.addFloatField(value = 0.0,row = 1,column = 1, width = 8,
                                                precision = 2, state = "readonly")
        self.inputField.bind("<Return>", lambda event: self.computeSqrt())
```

```

def computeSqrt(self):
    try:
        number = self.inputField.getNumber()
        result = math.sqrt(number)
        self.outputField.setNumber(result)
    except:
        self.messageBox(title = "ERROR",message = "Input must be an integer >= 0")

def main():
    NumberFieldDemo().mainloop()

if __name__=="__main__":
    main()

```

INPUT FROM PROMPTER BOXES

The Prompter Box is used to input the data from the user by using a Popup Dialog Box. The prompter box displays a title, a message for the prompt, an entry field for the user's input, and a button to submit the input.

Below is the example program illustrates the usage of Prompter Boxes:

```

from breezypythongui import EasyFrame
class PrompterBoxDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Prompter Box Demo", width = 300, height = 100)
        self.label = self.addLabel(text = "Ajay Kumar", row = 0, column = 0, sticky="NSEW")
        self.addButton(text = "Username", row = 1, column = 0,
                        command = self.getUserName)

    def getUserName(self):
        name = self.prompterBox(title = "Input Dialog",promptString = "Your username:")
        self.label["text"] = "Hi " + name + "!"

def main():
    PrompterBoxDemo().mainloop()

if __name__=="__main__":
    main()

```



On execution of above program, if the button **Username** is clicked then the prompter box is appeared and prompts to enter the input and the input text is displayed onto the screen.

FILE DIALOGS

GUI-based programs allow the user to browse the computer's file system with **file dialogs**. Python's **tkinter.filedialog** module includes two functions, **askopenfilename** and **asksaveasfilename**, to support file access in a GUI-based program. Each function pops up the standard file dialog for the user's particular computer system.

Syntax for these two functions is as follows:

```
fList = [("Python files", "*.py"), ("Text files", "*.txt")]
filename = tkinter.filedialog.askopenfilename(parent = self, filetypes = fList)
filename = tkinter.filedialog.asksaveasfilename(parent = self)
```

If the user selects the file dialog's **Cancel** button, the function returns the empty string. Otherwise, when the user selects the **Open**, the function returns the full pathname of the file that the user has selected or if the user selects the **Save** button, then it asks to enter the name of the file to save. The program can then use the filename to open the file for input or output in the usual manner.

Following is the list of the optional arguments one can supply to the two file dialog functions:

- **defaultextension** The extension to add to the filename, if not given by the user (ignored by the open dialog).
- **filetypes** A sequence of (label, pattern) tuples. Specifies the file types available for input.
- **initialdir** A string representing the directory in which to open the dialog.
- **initialfile** A string representing the filename to display in the save dialog name field.
- **parent** The dialog's parent window.
- **title** A string to display in the dialog's title bar.

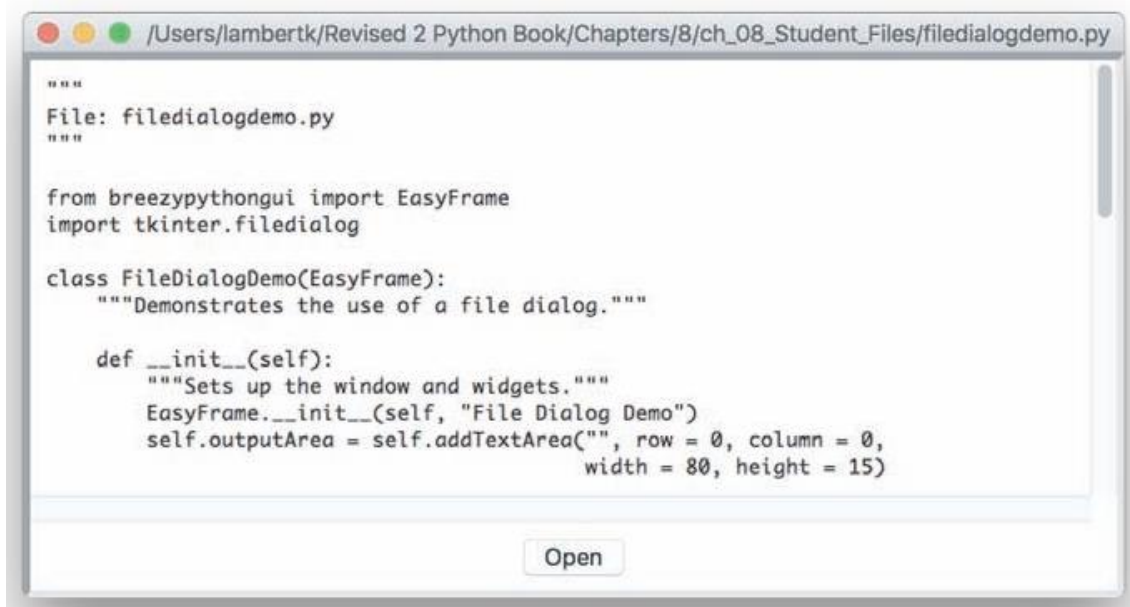
Below is the program that opens a selected Python File

```
from breezypythongui import EasyFrame
import tkinter.filedialog
class FileDialogDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, "File Dialog Demo")
        self.outputArea = self.addTextArea("", row = 0, column = 0, width = 80, height = 15)
        self.addButton(text = "Open", row = 1, column = 0, command = self.openFile)
    def openFile(self):
        fList = [("Python Files", "*.py")]
        fileName = tkinter.filedialog.askopenfilename(parent = self, filetypes = fList)
        if fileName != "":
            file = open(fileName, 'r')
            text = file.read()
            file.close()
            self.outputArea.setText(text)
            self.setTitle(fileName)
```

```
def main():
    FileDialogDemo().mainloop()

if __name__=="__main__":
    main()
```

Output:



USING NESTED FRAMES TO ORGANIZE COMPONENTS

Nested Frames are used in those situations when the design of the window is complex. And to make arrangement of window component that can be fit into grid layout of window, nested frames are preferred to use. While designing the window with the use of nested frames, window is assumed to be partitioned into panels, where each panel consists of some components which are aligned in the form of grid.

- First create a window, by creating a class that extends **EasyFrame** class of the **breezypythongui** module, which is sub class of the **tkinter.Frame** class.
- In order to add panel to the window, the method **addPanel** with two arguments row and column position in the window's grid.
- And then individual components are added to the panel using the method **addComponent**

To create the following window, the code should be written as follows:



Program Block of Code:

```
from breezypythongui import EasyFrame
class PanelDemo(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, "Panel Demo - v2")

        dataPanel = self.addPanel(row = 0, column = 0)
        dataPanel.addLabel(text = "Label 1", row = 0, column = 0)
        dataPanel.addTextField(text = "Text1", row = 0, column = 1)
        dataPanel.addLabel(text = "Label 2", row = 1, column = 0)
        dataPanel.addTextField(text = "Text2", row = 1, column = 1)

        buttonPanel = self.addPanel(row = 1, column = 0)
        buttonPanel.addButton(text = "B1", row = 0, column = 0)
        buttonPanel.addButton(text = "B2", row = 0, column = 1)
        buttonPanel.addButton(text = "B3", row = 0, column = 2)

def main():
    PanelDemo().mainloop()

if __name__ == "__main__":
    main()
```

USING A COLOR CHOOSER

Most graphics software packages allow the user to pick a color with a standard color chooser. This is a dialog that presents a color wheel from which the user can choose a color with the mouse. Python's **tkinter.colorchooser** module includes an **askcolor** function for this purpose.

The **tkinter.colorchooser.askcolor** function returns a tuple of two elements. If the user has clicked **OK** in the dialog, the first element in the tuple is a nested tuple containing the three RGB values, and the second element is the hex string value of the color. If the user has clicked **Cancel** in the dialog, both elements in the tuple are **None**.

Below is the program that uses the **askcolor** function and this program used the **addCanvas** function which is used for drawing shapes:

```
from breezypythongui import EasyFrame
import tkinter.colorchooser
class ColorPicker(EasyFrame):
    def __init__(self):
        EasyFrame.__init__(self, title = "Color Chooser Demo")
        self.addLabel('R', row = 0, column = 0)
        self.addLabel('G', row = 1, column = 0)
        self.addLabel('B', row = 2, column = 0)
        self.addLabel("Color", row = 3, column = 0)
```

```

self.r = self.addIntegerField(value = 0,row = 0, column = 1)
self.g = self.addIntegerField(value = 0,row = 1, column = 1)
self.b = self.addIntegerField(value = 0,row = 2, column = 1)
self.hex = self.addTextField(text = "#000000",row = 3, column = 1,width = 10)
self.canvas = self.addCanvas(row = 0, column = 2, rowspan = 4, width = 50,
                             background = "#000000")
self.addButton(text = "Choose color", row = 4,column = 0, columnspan = 3,
               command = self.chooseColor)

```

def chooseColor(self):

```

    colorTuple = tkinter.colorchooser.askcolor()
    if not colorTuple[0]: return
    ((r, g, b), hexString) = colorTuple
    self.r.setNumber(int(r))
    self.g.setNumber(int(g))
    self.b.setNumber(int(b))
    self.hex.setText(hexString)
    self.canvas["background"] = hexString

```

def main():

```

    ColorPicker().mainloop()

```

if __name__=="__main__":

```

    main( )

```

Output:

On execution of the above program, the following window is displayed:



After clicking the **choose color** button, then color chooser window is appeared:



Select the desired color and press OK, then the canvas area of the window is drawn with the selected colors as follows:

