MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject Name : PYTHON PROGRAMMING

CLASS : I-MCA

SUBJECT CODE : 23PCA13

Unit III

Design with Classes: Getting inside Objects and Classes Data-Modeling Examples Building a New Data Structure The Two Dimensional Grid - Structuring Classes with Inheritance and Polymorphism - GraphicalUser Interfaces - The Behavior of terminal-Based programs and GUI-Based programs - Coding Simple GUI-Based programs - Windows and Window Components - Command Buttons and responding to events.

Python Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Syntax: Class Definition

```
class ClassName:

    # Statement
```

Syntax: Object Definition

```
obj = ClassName()

print(obj.atrr)
```

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Creating a Python Class

Here, the class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

- Python3

```
class Dog:
```

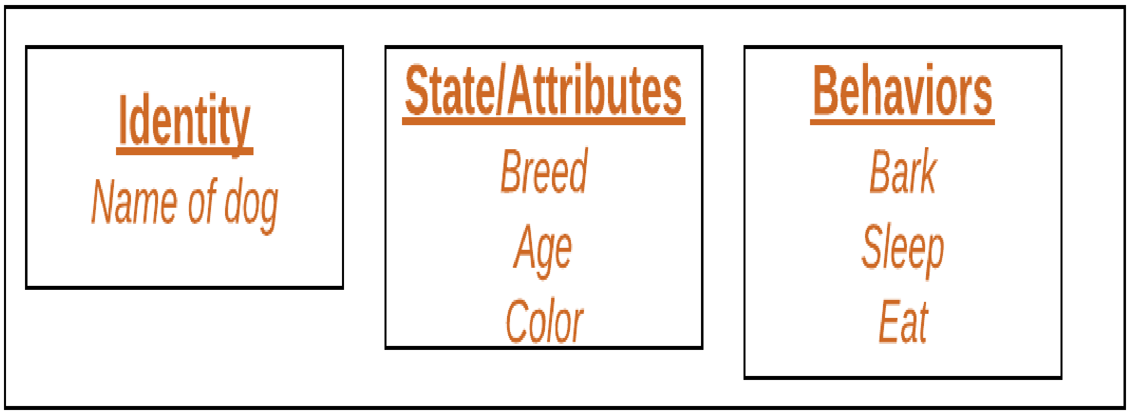
```
sound = "bark"
```

Object of Python Class

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. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

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.



Explanation :

In this example, we are creating a Dog class and we have created two class variables **attr1 and attr2**. We have created a method named **fun()** which returns the string "I'm a, {attr1}" and I'm a, {attr2}. We have created an object of the Dog class and we are printing at the **attr1** of the object. Finally, we are calling the **fun()** function.

Self Parameter

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

- Python3

```
class GFG:

    def __init__(self, name, company):

        self.name = name

        self.company = company



    def show(self):

        print("Hello my name is " + self.name+" and I" +

              " work in "+self.company+".")
```

```
obj = GFG("John", "GeeksForGeeks")

obj.show()
```

The Self Parameter does not call it to be Self, You can use any other name instead of it. Here we change the self to the word someone and the output will be the same.

- Python3

```
class GFG:

    def __init__(somename, name, company):

        somename.name = name

        somename.company = company



    def show(somename):

        print("Hello my name is " + somename.name +

              " and I work in "+somename.company+".")




obj = GFG("John", "GeeksForGeeks")
```

```
obj.show()
```

Output: Output for both of the codes will be the same.

Hello my name is John and I work in GeeksForGeeks.

Explanation:

In this example, we are creating a GFG class and we have created the **name, and company** instance variables in the constructor. We have created a method named **say_hi()** which returns the string "Hello my name is " + {name} +" and I work in "+{company}+".".We have created a person class object and we passing the name **John and Company** GeeksForGeeks to the instance variable. Finally, we are calling the **show()** of the class.

Pass Statement

The program's execution is unaffected by the **pass** statement's inaction. It merely permits the program to skip past that section of the code without doing anything. It is frequently employed when the syntactic constraints of Python demand a valid statement but no useful code must be executed.

- Python3

```
class MyClass:

    pass
```

__init__() method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

- Python3

```
# Sample class with init method

class Person:



    # init method or constructor

    def __init__(self, name):

        self.name = name



    # Sample Method

    def say_hi(self):

        print('Hello, my name is', self.name)




p = Person('Nikhil')
```

```
p.say_hi()
```

Output:

```
Hello, my name is Nikhil
```

Explanation:

In this example, we are creating a Person class and we have created a **name** instance variable in the constructor. We have created a method named as say_hi() which returns the string "Hello, my name is {name}".We have created a person class object and we pass the name Nikhil to the instance variable. Finally, we are calling the say_hi() of the class.

__str__() method

Python has a particular method called **__str__()**. that is used to define how a **class** object should be represented as a string. It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information. When a class object is used to create a string using the built-in functions print() and str(), the **__str__()** function is automatically used. You can alter how objects of a **class** are represented in strings by defining the **__str__()** method.

- Python3

```
class GFG:

    def __init__(self, name, company):

        self.name = name

        self.company = company
    def __str__(self):

        return f"My name is {self.name} and I work in {self.company}."




my_obj = GFG("John", "GeeksForGeeks")

print(my_obj)
```

Output:

```
My name is John and I work in GeeksForGeeks.
```

Explanation:

In this example, We are creating a class named GFG.In the class, we are creating two instance variables **name and company**. In the __str__() method we are returning the **name** instance variable and **company** instance variable. Finally, we are creating the object of GFG class and we are calling the __str__() method.

Class and Instance Variables

Instance variables are for data, unique to each instance and 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.

Defining instance variables using a constructor.

- Python3

```
# Python3 program to show that the variables with a value
# assigned in the class declaration, are class variables and

# variables inside methods and constructors are instance

# variables.



# Class for Dog




class Dog:



    # Class Variable

    animal = 'dog'



    # The init method or constructor

    def __init__(self, breed, color):
        # Instance Variable

        self.breed = breed

        self.color = color




# Objects of Dog class

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)
```

Output:

```
Rodger details:

Rodger is a dog

Breed:  Pug

Color:  brown

Buzo details:

Buzo is a dog

Breed:  Bulldog

Color:  black

Accessing class variable using class name

dog
```

Explanation:

A class named Dog is defined with a class variable animal set to the string "dog". Class variables are shared by all objects of a class and can be accessed using the class name. Dog class has two instance variables **breed and color.** Later we are creating two objects of the **Dog** class and we are printing the value of both objects with a class variable named animal.

Defining instance variables using the normal method:

- Python3

```
# Python3 program to show that we can create

# instance variables inside methods



# Class for Dog





class Dog:



    # Class Variable

    animal = 'dog'
```

```
# The init method or constructor

def __init__(self, breed):



    # Instance Variable

    self.breed = breed



# Adds an instance variable

def setColor(self, color):

    self.color = color



# Retrieves instance variable

def getColor(self):

    return self.color
```

```
# Driver Code

Rodger = Dog("pug")

Rodger.setColor("brown")

print(Rodger.getColor())
```

Output:

```
brown
```

Explanation:

In this example, We have defined a class named **Dog** and we have created a class variable animal. We have created an instance variable breed in the **constructor**. The class Dog consists of two methods **setColor** and **getColo**r, they are used for creating and initializing an instance variable and retrieving the value of the instance variable. We have made an object of the **Dog** class and we have set the instance variable value to brown and we are printing the value in the terminal.

Python | Using 2D arrays/lists

Python provides powerful data structures called lists, which can store and manipulate collections of elements. Also provides many ways to create 2-dimensional lists/arrays. However one must know the differences between these ways because they can create complications in code that can be very difficult to trace out. In this article, we will explore the right way to use 2D arrays/lists in Python.

Using 2D arrays/lists the right way

Using 2D arrays/lists the right way involves understanding the structure, accessing elements, and efficiently manipulating data in a two-dimensional grid. When working with structured data or grids, 2D arrays or lists can be useful. A 2D array is essentially a list of lists, which represents a table-like structure with rows and columns.

Creating a 1-D list

In Python, Initializing a collection of elements in a linear sequence requires creating a 1D array, which is a fundamental process. Although Python does not have a built-in data structure called a ‘1D array’, we can use a list that can achieve the same functionality. Python lists are dynamic and versatile, making them an excellent choice for representing 1D arrays. Let’s start by looking at common ways of creating a 1d array of size N initialized with 0s.

Creating 1D List using Naive Methods

Manually initializing and populating a list without using any advanced features or constructs in Python is known as creating a 1D list using “Naive Methods”.

Declaring Claas Objects (Also called instantiating a class)

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 instances.

Example:

```
N = 5

ar = [0]*N

print(ar)
```

Output

```
[0,                    0,                    0,                    0,                    0]
```

Creating 1D List using List Comprehension

Here we are multiplying the number of rows by the empty list and hence the entire list is created with every element zero.

- Python3

```
N = 5

arr = [0 for i in range(N)]

print(arr)
```

Output

```
[0,                    0,                    0,                    0,                    0]
```

Creating a 2-D list

Using 2D arrays/lists the right way involves understanding the structure, accessing elements, and efficiently manipulating data in a two-dimensional grid. By mastering the use of 2D arrays, you can significantly improve your ability to handle complex data and efficiently perform various operations.

Creating 2D List using Naive Method

Here we are multiplying the number of columns and hence we are getting the 1-D list of size equal to the number of columns and then multiplying it with the number of rows which results in the creation of a 2-D list.

- Python3

```
rows, cols = (5, 5)

arr = [[0]*cols]*rows

print(arr)
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Note: Using this method can sometimes cause unexpected behaviors. In this method, each row will be referencing the same column. This means, even if we update only one element of the array, it will update the same column in our array.

- Python

```
rows, cols = (5, 5)

arr = [[0]*cols]*rows

print(arr, "before")



arr[0][0] = 1 # update only one element

print(arr, "after")
```

Output

```
([[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]], 'before')
([[1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0, 0, 0]], 'after')
```

Creating 1D List using List Comprehension

Here we are basically using the concept of list comprehension and applying a loop for a list inside a list and hence creating a 2-D list.

- Python3

```
rows, cols = (5, 5)

arr = [[0 for i in range(cols)] for j in range(rows)]

print(arr)
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Creating 1D List using Empty List

Here we are appending zeros as elements for a number of columns times and then appending this 1-D list into the empty row list and hence creating the 2-D list.

- Python3

```
arr=[]

rows, cols=5,5

for i in range(rows):

    col = []

    for j in range(cols):

        col.append(0)

    arr.append(col)
```

```
print(arr)
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Initializing 2D Array

The provided code demonstrates two different approaches to initializing a 2D array in Python. First, the array arr is initialized using a 2D list comprehension, where each row is created as [0, 0, 0, 0, 0]. The entire array is created as a list of references to the same inner list, resulting in aliasing. Any change made to an element in one row will be reflected in all rows. The code then shows another approach using a nested list comprehension to create the 2D array arr. This method avoids aliasing by creating a new list for each row, resulting in a proper 2D array.

- Python3

```
# Python 3 program to demonstrate working

# of method 1 and method 2.

rows, cols = (5, 5)

# method 2 1st approach

arr = [[0]*cols]*rows

# lets change the first element of the

# first row to 1 and print the array
```

```
arr[0][0] = 1



for row in arr:

    print(row)



# method 2 2nd approach

arr = [[0 for i in range(cols)] for j in range(rows)]



# again in this new array lets change

# the first element of the first row

# to 1 and print the array

arr[0][0] = 1

for row in arr:

    print(row)
```

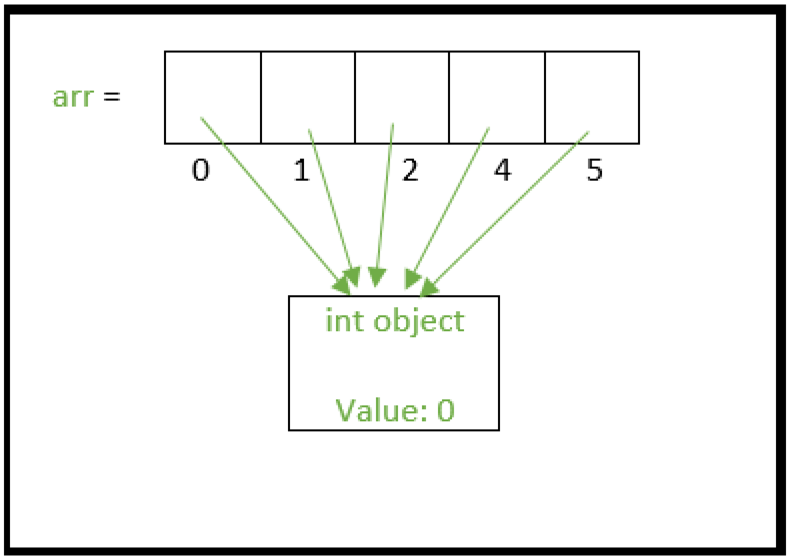
Output

```
[1,                 0,                 0,                 0,                 0]
[1,                 0,                 0,                 0,                 0]
```

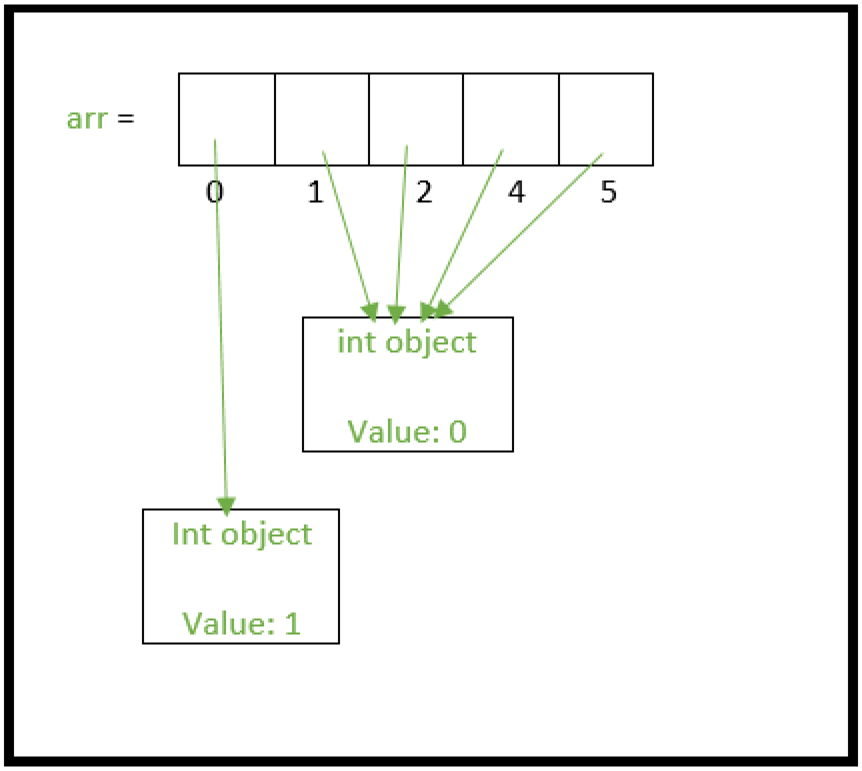
```
[1,    0,    0,    0,    0]
[1,    0,    0,    0,    0]
[1,    0,    0,    0,    0]
[1,    0,    0,    0,    0]
[0,    0,    0,    0,    0]
[0,    0,    0,    0,    0]
[0,    0,    0,    0,    0]
[0,    0,    0,    0,    0]
```

Explanation:

We expect only the first element of the first row to change to 1 but the first element of every row gets changed to 1 in method 2a. This peculiar functioning is because Python uses shallow lists which we will try to understand.
In method 1a, Python doesn’t create 5 integer objects but creates only one integer object, and all the indices of the array arr point to the same int object as shown.



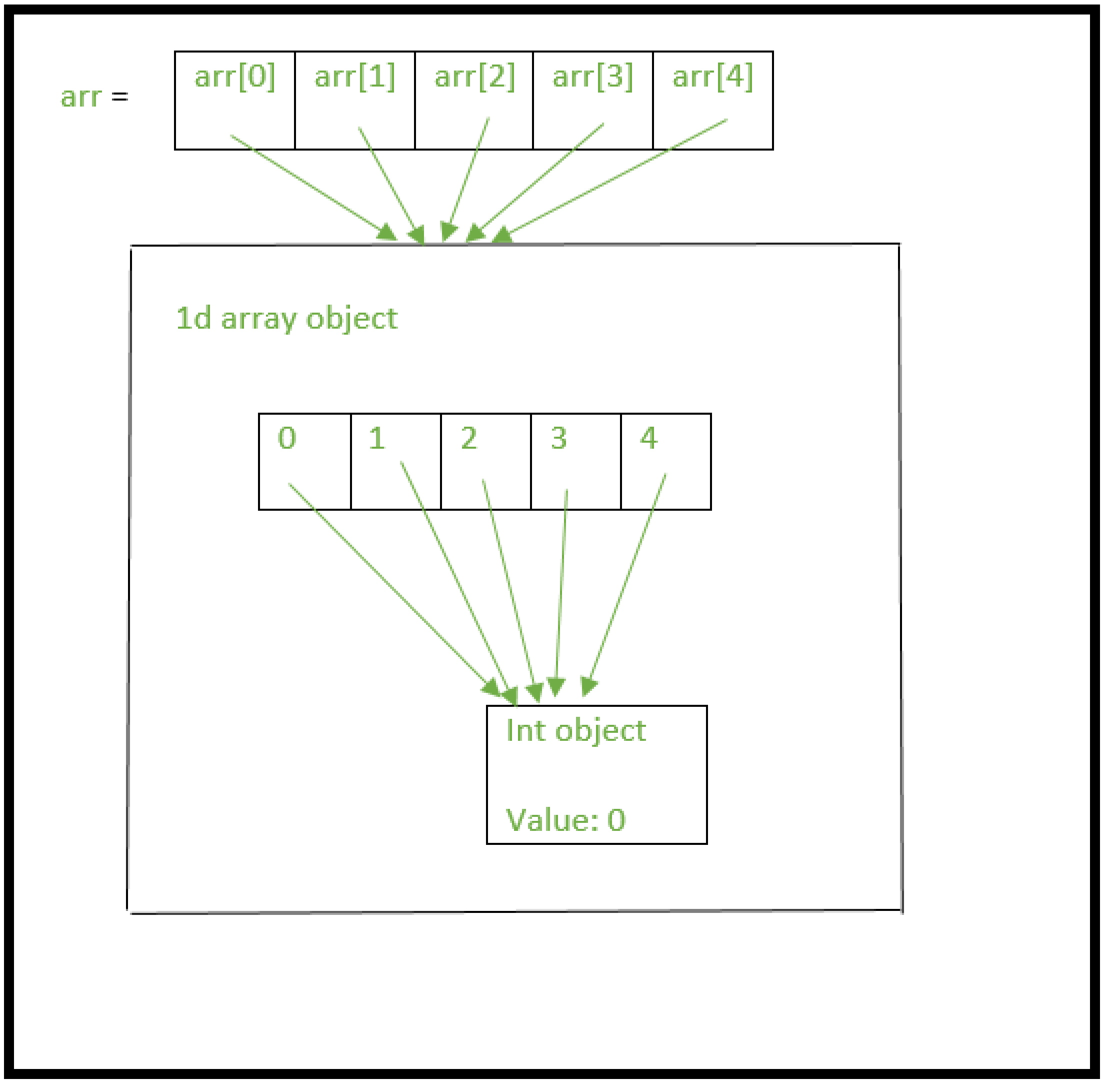
If we assign the 0th index to another integer say 1, then a new integer object is created with the value of 1 and then the 0th index now points to this new int object as shown below



Similarly, when we create a 2d array as “arr = [[0]*cols]*rows” we are essentially extending the above analogy.

1. Only one integer object is created.
2. A single 1d list is created and all its indices point to the same int object in point 1.
3. Now, arr[0], arr[1], arr[2] …. arr[n-1] all point to the same list object above in point 2.

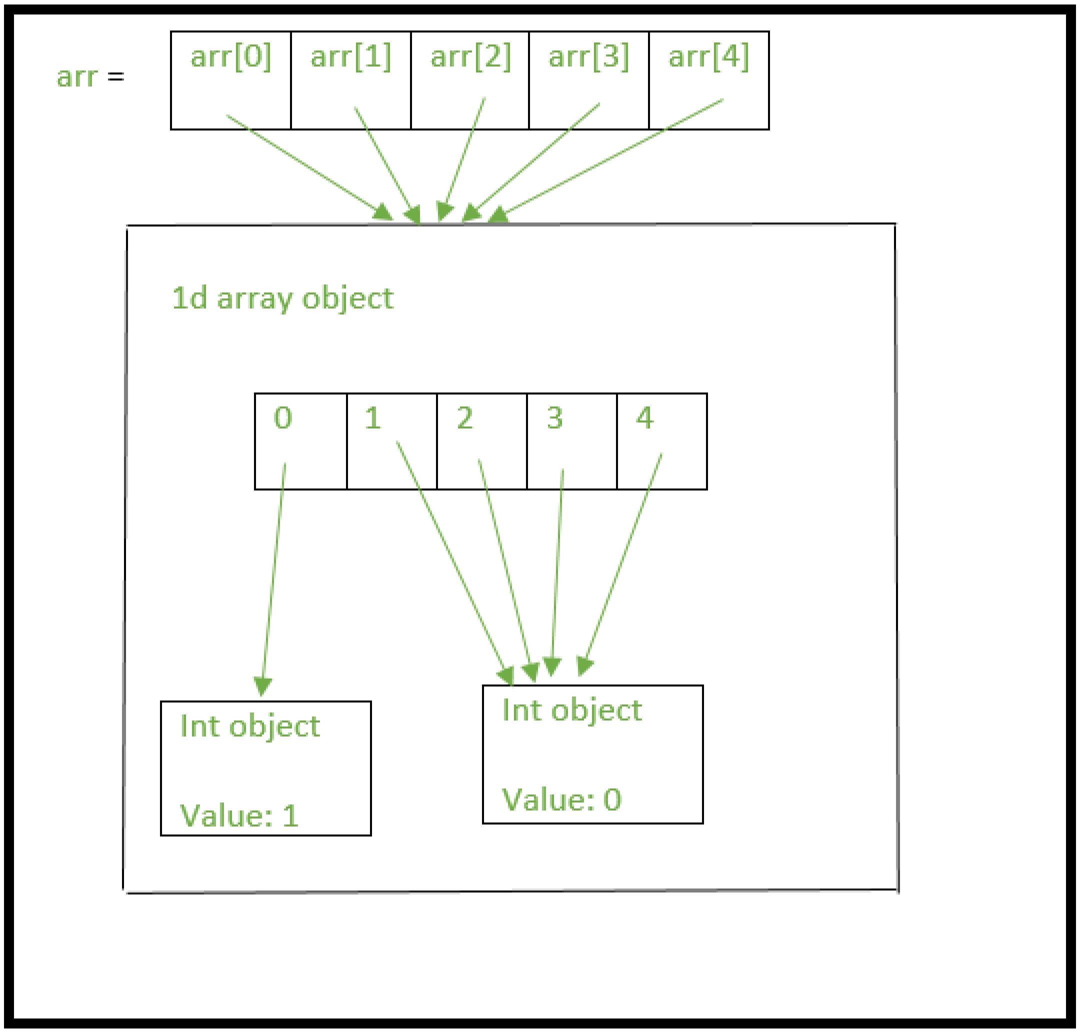
The above setup can be visualized in the image below.



Now lets change the first element in first row of “arr” as arr[0][0] = 1

- arr[0] points to the single list object we created we above.(Remember arr[1], arr[2] …arr[n-1] all point to the same list object too).
- The assignment of arr[0][0] will create a new int object with the value 1 and arr[0][0] will now point to this new int object.(and so will arr[1][0], arr[2][0] … arr[n-1][0])

This can be clearly seen in the below image.



So when 2d arrays are created like this, changing values at a certain row will affect all the rows since there is essentially only one integer object and only one list object being referenced by the all the rows of the array.

As you would expect, tracing out errors caused by such usage of shallow lists is difficult. Hence the better way to declare a 2d array is

- Python3

```
rows, cols = (5, 5)
```

```
print([[0 for i in range(cols)] for j in range(rows)])
```

Output

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

This method creates 5 separate list objects, unlike method 2a. One way to check this is by using the ‘is’ operator which checks if the two operands refer to the same object.

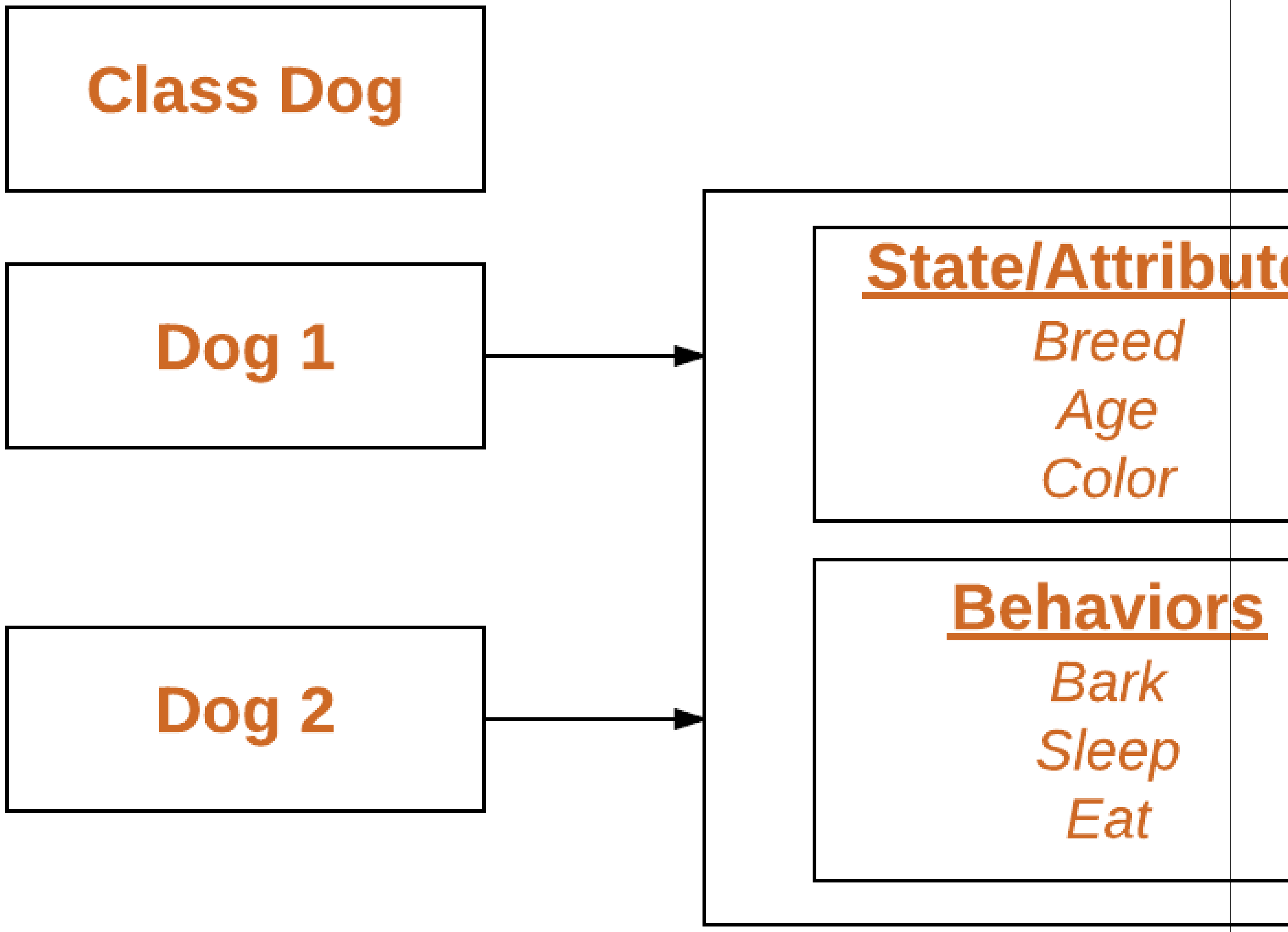
- Python3

```
rows, cols = (5, 5)



# method 2 2nd approach

arr = [[0 for i in range(cols)] for j in range(rows)]



# check if arr[0] and arr[1] refer to

# the same object

print(arr[0] is arr[1]) # prints False



# method 2 1st approach
```

```
arr = [[0]*cols]*rows




# check if arr[0] and arr[1] refer to the same object prints True because there is only one

#list object being created.

print(arr[0] is arr[1])
```

Output

```
False
True
```



Example of Python Class and object

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

- Python3

```
# Python3 program to
# demonstrate instantiating

# a class

class Dog:



    # A simple class

    # attribute

    attr1 = "mammal"

    attr2 = "dog"



    # A sample method

    def fun(self):

        print("I'm a", self.attr1)

        print("I'm a", self.attr2)
```

```
# Driver code

# Object instantiation

Rodger = Dog()



# Accessing class attributes

# and method through objects

print(Rodger.attr1)

Rodger.fun()
```

Output:

```
mammal

I'm a mammal

I'm a dog
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

Polymorphism in Python

What is polymorphism? Polymorphism refers to having multiple forms. Polymorphism is a programming term that refers to the use of the same function name, but with different signatures, for multiple types.

Example of in-built polymorphic functions:

```
# Python program for demonstrating the in-built poly-morphic functions

# len() function is used for a string
print (len("Javatpoint"))

# len() function is used for a list
print (len([110, 210, 130, 321]))
```

Output:

```
10
4
```

Examples of user-defined polymorphic functions:

```
# here, is a simple Python function
# for demonstrating the Polymorphism

def add(p, q, r = 0):
    return p + q + r

# Driver code
print (add(6, 23))
print (add(22, 31, 544))
```

Output:

```
29
597
```

Polymorphism with Class Methods

Below is an example of how Python can use different types of classes in the same way. For loops that iterate through multiple objects are created. Next, call the methods without caring about what class each object belongs to. These methods are assumed to exist in every class.

Example:

```
class xyz():
    def websites(self):
        print("Javatpoint is a website out of many availabe on net.")

    def topic(self):
        print("Python is out of many topics about technology on Javatpoint.")

    def type(self):
        print("Javatpoint is an developed website.")

class PQR():
    def websites(self):
        print("Pinkvilla is a website out of many availabe on net. .")

    def topic(self):
        print("Celebrities is out of many topics.")

    def type(self):
        print("pinkvilla is a developing website.")

obj_jtp = xyz()
obj_pvl = PQR()
for domain in (obj_jtp, obj_pvl):
    domain.websites()
    domain.topic()
    domain.type()
```

Output:

```
Javatpoint is a website out of many availabe on net.
Python is out of many topics about technology on Javatpoint.
Javatpoint is an developed website.
Pinkvilla is a website out of many availabe on net.
Celebrities is out of many topics.
pinkvilla is a developing website.
```

Polymorphism with Inheritance:

Polymorphism allows us to define methods in Python that are the same as methods in the parent classes. In inheritance, the methods of the parent class are passed to the child class. It is possible to change a method that a child class has inherited from its parent class. This is especially useful when the method that was inherited from the parent doesn't fit the child's class. We re-implement such methods in the child classes. This is **Method Overriding**.

Example:

```
class Birds:
    def intro1(self):
        print("There are multiple types of birds in the world.")
    def flight1(self):
        print("Many of these birds can fly but some cannot.")

class sparrow1(Birds):
    def flight1(self):
        print("Sparrows are the bird which can fly.")

class ostrich1(Birds):
    def flight1(self):
        print("Ostriches are the birds which cannot fly.")

obj_birds = Birds()
obj_spr1 = sparrow1()
obj_ost1 = ostrich1()

obj_birds.intro1()
obj_birds.flight1()

obj_spr1.intro1()
obj_spr1.flight1()

obj_ost1.intro1()
obj_ost1.flight1()
```

Output:

```
There are multiple types of birds in the world.
Many of these birds can fly but some cannot.
There are multiple types of birds in the world.
Sparrows are the bird which can fly.
There are multiple types of birds in the world.
Ostriches are the birds which cannot fly.
```

Polymorphism with a Function and Objects

We can also create functions that can take any object. This allows for polymorphism. Let's take the following example: let's make a function called "func()", which will take an object we will call "obj". Even though we use the name "obj", any object that is instantiated will be able to call into this function. Let's next to give the function something it can do with the 'obj object passed to it. Let's call these three methods websites(), topic(), and type(). Each of them is defined in the classes' xyz' and 'PQR'. If we don't already have instantiations of the 'xyz" and 'PQR classes, let us create them. We can then call their actions using the same function func().

Example:

```
def func(obj):
       obj.websites()
       obj.topic()
      obj.type()

obj_jtp = xyz()
obj_pvl = PQR()

func(obj_jtp)
func(obj_pvl)
```

Output:

```
Javatpoint is a website out of many availabe on net.
Python is out of many topics about technology on Javatpoint.
Javatpoint is a developed website.
Pinkvilla is a website out of many availabe on net. .
Celebrities is out of many topics.
pinkvilla is a developing website.
```

Code: Implementing Polymorphism with a Function

```
class xyz():
    def websites(self):
        print("Javatpoint is a website out of many availabe on net.")

    def topic(self):
        print("Python is out of many topics about technology on Javatpoint.")

    def type(self):
        print("Javatpoint is an developed website.")

class PQR():
    def websites(self):
        print("Pinkvilla is a website out of many availabe on net. .")

    def topic(self):
        print("Celebrities is out of many topics.")

    def type(self):
        print("pinkvilla is an developing website.")

def func(obj):
     obj.websites()
     obj.topic()
     obj.type()

obj_jtp = xyz()
obj_pvl = PQR()

func(obj_jtp)
func(obj_pvl)
```

Output:

```
Javatpoint is a website out of many availabe on net.
Python is out of many topics about technology on Javatpoint.
Javatpoint is a developed website.
Pinkvilla is a website out of many availabe on net. .
Celebrities is out of many topics.
```

pinkvilla is a developing website.

GraphicalUser Interfaces

Python GUI (Graphical User Interface) alludes to the visual elements and intelligent parts that permit users to collaborate with a software application. A GUI gives a natural and user-accommodating method for getting to an application's usefulness, and it ordinarily incorporates windows, buttons, menus, text boxes, and other visual elements. Python gives a scope of tools and frameworks for building GUI applications, each with its own assets and shortcomings. Probably the most famous Python GUI tools incorporate Tkinter, PyQt, PySide, Kivy, wxPython, PyGTK, PySimpleGUI, PyForms, PyQTGraph, and PyVista.

These GUI tools offer a scope of features, including support for numerous platforms, cross-platform similarity, high level gadget libraries, support for mixed media, 3D designs, and more. Developers can pick a GUI instrument in light of their task's particular necessities, programming abilities, and favoured programming style.

Let's take a more thorough look at the list that has been carefully picked below, explaining what it can accomplish to wow users with the applications made using such frameworks.

Most Popular GUI Tools:

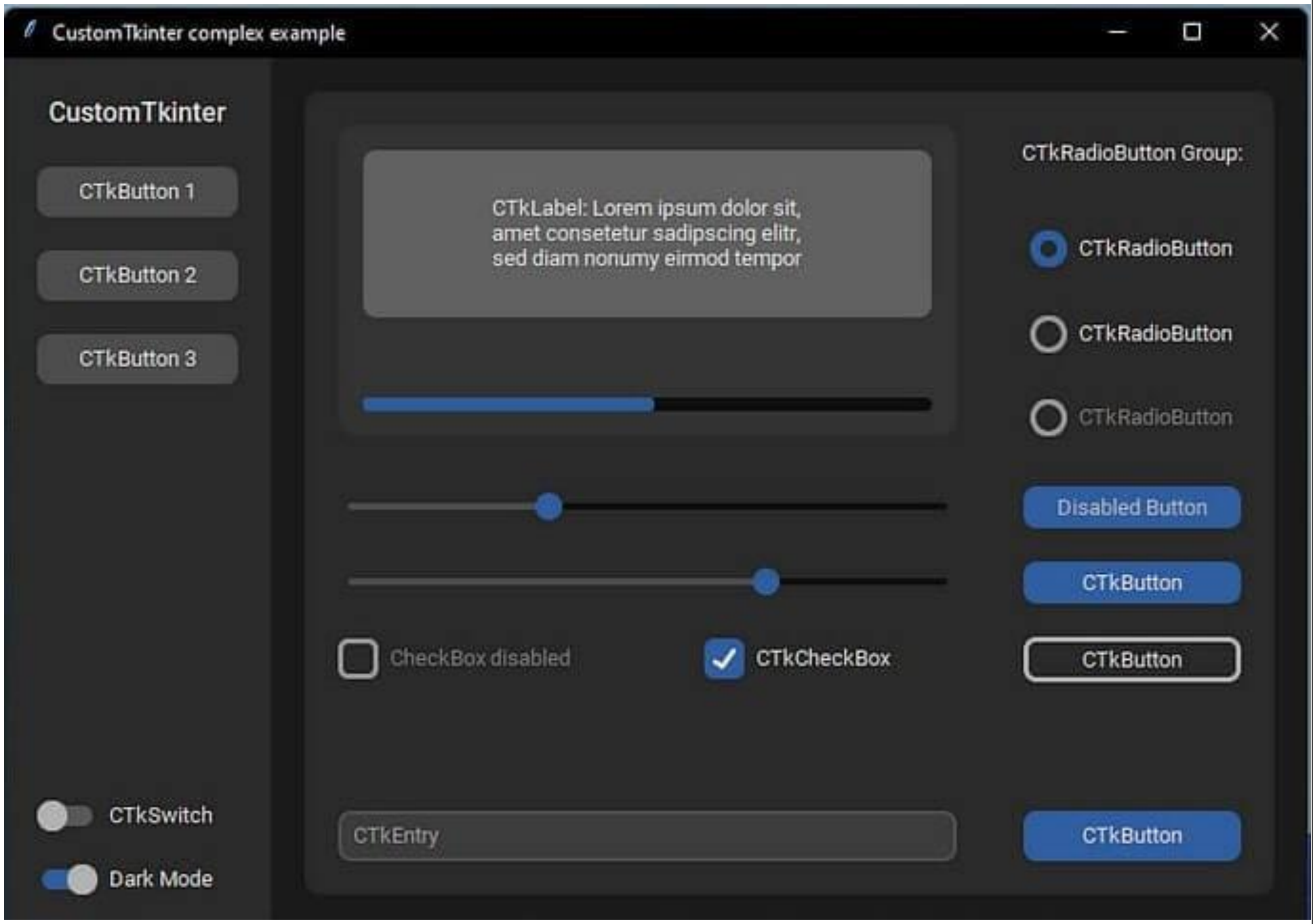
1. Tkinter

Tkinter is a standard Python library that gives a toolkit to building GUIs. It is a lightweight and simple to-utilize library that permits developers to make basic or complex interfaces. It provides a library of essential GUI Widgets that can be used to construct an open-source, straightforward Graphical User Interface. Some of the GUI Widgets with varying degrees of customizability include:

1. **Buttons:** which can be used to receive user input.
2. **Check buttons:** which can be used to combine selections like colours, monochrome images, border size.

3. **File dialogues:** which upload and download files from and to the app.
4. **Labels:** which display textual information.
5. **Canvas:** which gives developers space to draw and paint plots and graphs.

For constructing desktop or Python GUI apps, all we must do now is engage with the TKinter documentation more and get to know many of its GUI components.



CTkLabel: Lorem ipsum dolor sit, amet consetetur sadipscing elitr, sed diam nonumy eirmod tempor

2. Kivy

In essence, Kivy is a mixture of Python and Cython. Using its open-source framework and the more than 20 Widgets in its toolkit, developers may create simple user interfaces with multi-touch capabilities. figuring out whether Kivy supports the Natural User Interface (NUI) By using this, a user can innately get familiar with many of the interactions enabled by this open-source Python GUI Framework that are generally hidden.. Ingeniously, Kivy is fantastic news for interface designers as well because they will just need to write codes once and then apply them across many platforms while properly utilising potent design and graphics strategies. Still curious as to what Kivy is used for the most frequently! Our iOS and Android apps, as well as any Windows, Mac OS, Raspberry Pi, and Linux user interface, all clearly demonstrate this. To begin using this incredibly adaptable GUI framework, we should right away add it to our Python environment by following the installation instructions listed on its official website.

3. PyGTK

PyGTK is a Python restricting for the GTK+ toolbox, a famous library for building graphical user interfaces. GTK+ is an open-source cross-platform tool stash that gives a scope of gadgets and tools for building desktop applications. PyGTK permits developers to make GTK+ applications utilizing the Python programming language. It gives a scope of features, including support for gadget customization, occasion handling, and gadget pressing. PyGTK additionally incorporates support for internationalization, making it simple to make applications that can be utilized by individuals communicating in various dialects.

PyGTK is much of the time used to fabricate desktop applications for Linux-based frameworks, despite the fact that it can likewise be utilized on different platforms like Windows and macOS. PyGTK applications can be composed utilizing any Python proof-reader, and developers can utilize Dell, a GUI manufacturer, to outwardly make the application's interface.

PyGTK's primary benefit is its joining with the GTK+ tool stash, which gives a scope of strong and adaptable gadgets for making perplexing and adjustable user interfaces. It likewise offers help for theming, permitting developers to redo the presence of the application without composing any code.

4. Pyside2

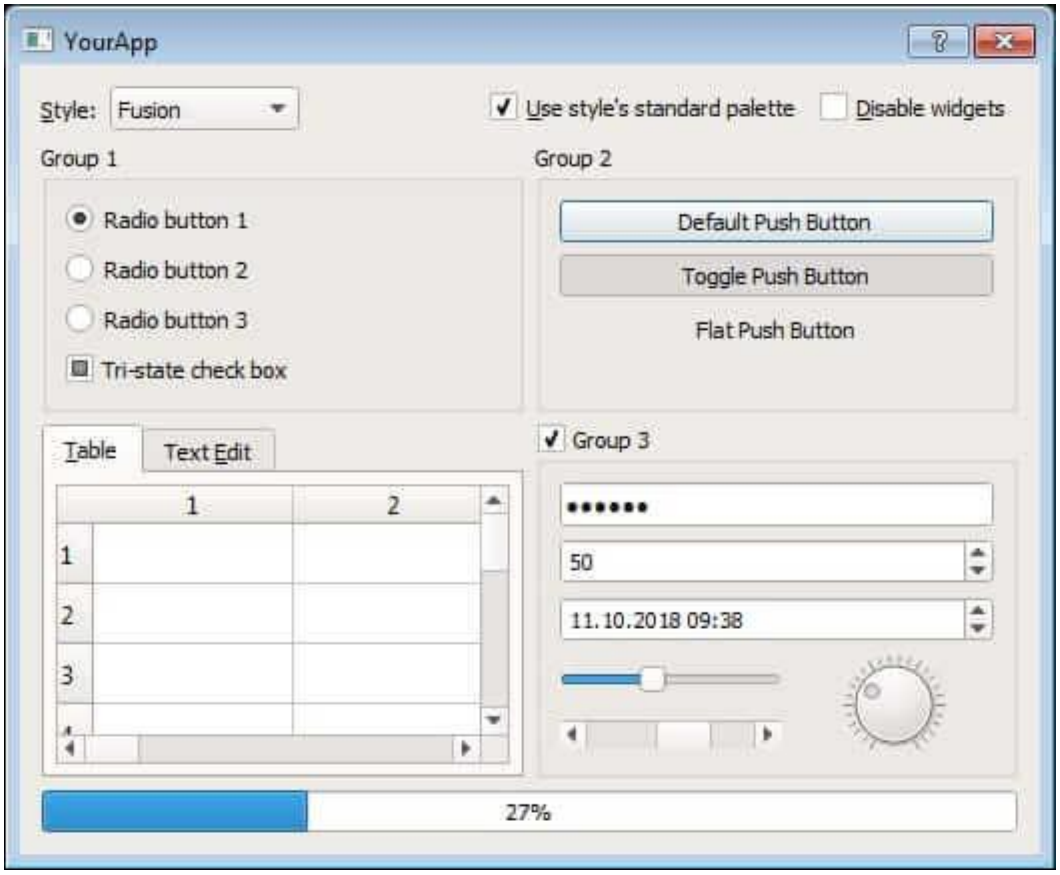
Pyside 2 is a solution for boosting any Python apps we already have that were created and developed by programmers and developers. Pyside 2 is also known as Qt for Python in the market. Additionally, we can look at the community that strongly supports the sharing of knowledge and insights among 1M Plus Python or other software developers to create Qt programmes in an enjoyable manner.

This GUI framework, known as the cross-platform Python Module for the time being, allows Py (i.e., Python) developers to easily access its set of modules and tools to design stunning and dynamic User Interfaces quickly and easily. Concerned about the documentation section's complexity! We should feel at ease because the process has been greatly simplified with video tutorials, examples, tutorials, and supplemental API Docs.

5. PyQt5

Another straightforward yet enticing cross-platform GUI framework for Python developers is PyQt5. We can quickly develop programmes that work with Mac OS X, Linux, Windows, Android, and Raspberry Pi. By combining various GUI widgets, such as sliders, list boxes, buttons, etc., and arranging them within the window, PyQt5 makes it possible to personalise themes and produce visually pleasing graphical designs that customers enjoy a lot.

This enables developers to create User Interfaces by directly manipulating data while maintaining consistency and overall code reuse. Additionally, PyQt5 has gradually introduced the modularity of the Python language through its extensions so that programmers can easily and robustly design both small-scale and large-scale applications. I really like PyQt5's features! Then, with the command, we can begin making stunning UIs.



6. PySimpleGUI

In the year 2018, Mike B created PySimpleGUI, which has made Python development incredibly versatile and straightforward for beginners trying their luck harder to gain recognition in GUI Development. We don't need to spend much time learning the intricate details of the complex GUI development possibilities that have been made available for developers and other aspiring Python Programmers thanks to this hassle-free framework.

Unquestionably, this framework's efficient utilisation of the advantages of four GUIs?TKinter, Remi, Qt, and WxPython?has commendably decreased the difficulty level of boilerplate code implementation and helped beginners create User-Interfaces by making diverse visual elements easily accessible.

7. wxPython

For Python programmers and developers at all skill levels, wxPython is a good GUI framework. Its cross-platform capabilities will work in the same way whether we run it on Mac OS or different Unix systems, with no additional or zero overheads. ComboBox, ToggleButton, StatusBar, StaticLine, and many other multi-purpose pythonic widgets are wrapped in wxPython, which enables novice or intermediate developers to construct native UIs for their Python projects with little to no adjustments.

Most significantly, this GUI framework is free for anybody to use, and because of this amazing feature, it is more likely that code problems will be rectified or upgraded with better code readability. As a result, the expense of creating a high-quality Python application need not be a major concern for developers because wxPython is open-source and allows for code changes at any time. As a result, we need try pip instal wxPython now to instal this appropriate GUI framework.

8. PyForms

PyForms is a Python GUI toolkit that permits developers to make rich and complex desktop applications rapidly and without any problem. It gives a scope of features that work on the improvement interaction, including a visual manager, a simplified structure manufacturer, and backing for different platforms.

PyForms is based on top of PyQt, a Python restricting for the famous Qt structure, and it gives an article situated Programming interface that permits developers to make custom gadgets and designs. It likewise incorporates a scope of pre-fabricated gadgets, including buttons, marks, text boxes, and more.

PyForms likewise incorporates support for cutting edge features like media playback, 3D designs, and backing for web advancements like HTML and JavaScript. It gives a scope of format chiefs, occasion handlers, and information restricting tools that improve on the advancement cycle and decrease how much standard code required.

9. PyQTGraph

PyQTGraph is a Python library for making quick and intuitive graphical plots and visualizations. It is based on top of PyQt, a Python restricting for the famous Qt structure, and it gives a scope of cutting-edge features that settle on it a well-known decision for logical and designing applications.

One of PyQTGraph's fundamental benefits is its speed. It utilizes equipment sped up designs to make plots and visualizations rapidly and effectively, causing it ideal for applications that to demand constant information handling and visualization. It likewise incorporates support for live information streaming, making it simple to make ongoing information visualization applications. PyQTGraph gives a scope of cutting-edge plotting tools, including line plots, dissipate plots, bar outlines, and 2D and 3D surface plots. It likewise incorporates support for cutting edge features, for example, multi-hub plots, logarithmic tomahawks, and information separating and smoothing.

10. PyVista

PyVista is a Python library for 3D visualization and analysis of logical data. It is based on top of the VTK library, a well-known C++ library for visualizing and investigating logical data, and it gives a scope of cutting-edge features for making intelligent 3D visualizations.

One of PyVista's primary benefits is its convenience. It gives a Pythonic interface that permits developers to make 3D visualizations rapidly and effectively, without composing any C++ code. It likewise incorporates support for cutting edge features, for example, volume delivering, shaping, and cutting, making it an incredible asset for visualizing and investigating logical data.

What is terminal based program in Python?

They are applications that run in the terminal, and you can write your own apps in Python. If you are on Linux, try the following: Open a terminal and enter top . You will see a bunch of processes that are running on your system, sorted by which ones are using the most resources.

They are applications that run in the terminal, and you can write your own apps in Python. If you are on Linux, try the following: Open a terminal and enter top . You will see a bunch of processes that are running on your system, sorted by which ones are using the most resources.

What are terminal apps?

A terminal application is simply an application that runs inside the terminal. By now, you have seen that most of the output from your programs is printed to the terminal. By learning a few more techniques such as clearing the terminal screen and "pickling" data, you can create full-fledged standalone applications that run in the terminal.

A terminal app starts just like any other program, but it finishes when the user selects an action that causes the program to quit. In terminal applications, this often means something like entering 'q' or 'quit'.

If you are on a linux system, you can run some terminal apps right now. Many of these are not written in Python, but that doesn't matter. They are applications that run in the terminal, and you can write your own apps in Python. If you are on Linux, try the following:

- Open a terminal and enter `top`.
    - You will see a bunch of processes that are running on your system, sorted by which ones are using the most resources.
    - You can press 'q' to quit this application.
- Open a terminal and enter `nano test_file.txt`.
    - This is a text editor that runs in the terminal. It is useful for quick edits.
    - If you are editing files on a remote server, you need a terminal-based editor. Nano is one of the simplest.
    - Press Ctrl-x to exit. If you have typed anything, you will be given the option to save your file.
- Open a terminal and enter `vi`.

    - This is another text editor. It is much less intuitive than nano at first, but once you learn how to use it you can edit files extremely quickly.
    - Enter `:q` to exit.

There are quite a few other terminal apps. Let's figure out how to write some of our own.

Why write terminal apps?

Unless you see people working in technical fields on a regular basis, it's quite possible you have not seen many people using terminal applications. Even so, there are a number of reasons to consider writing a few terminal apps of your own.

- **They can be much more fun and satisfying to write than simpler programs.**
    - Applications, by their nature, solve interesting problems. When you write an application, you are creating an environment in which people work and play. That's pretty satisfying.
- **Terminal apps let you play with complex code, without layers of abstraction between you and the user.**
    - When you write more complex graphical applications, there are layers of abstraction between you and the user. To get input in a terminal, you issue the command `input("Please tell me some information: ")`. When you want some information in a graphical program, you have to build text boxes and buttons for submitting information. That has gotten pretty simple these days, but it is still more complicated than what you will do when writing terminal applications. Terminal applications let you focus on getting the logic right, rather than building a graphical interface.
- **You will learn about user interaction issues.**
    - Terminal apps do have users, even if that is just you, and maybe your friends or family. If you see people using your applications at an early stage, you will write better code. There is nothing like watching people use your programs to make you code more defensively. You know to enter a string in some places, but what keeps your users from entering numbers? Influencing your users to give you the right information the first time, and being prepared

to deal with the wrong kind of data are good skills as a programmer. Having real users at an early stage in your programming career is a good thing.

top

Greeter - A simple terminal app

Let's define a simple terminal app that we can make in this notebook. Greeter will:

- Always display a title bar at the top of the screen, showing the name of the app that is running.
- Offer you three choices:
    - Enter a name.
        - If the program knows that name, it will print a message saying something like "Hello, old friend."
        - If the program does not know that name, it will print a message saying something like, "It's nice to meet you. I will remember you."
            - The next time the program hears this name, it will greet the person as an old friend.
    - See a list of everyone that is known.
    - Quit.
- The program will remember people, even after it closes.
- The program may list the number of known people in the title bar.

Now we will go over a few things we need to know in order to build this app, and then we will build it.

top

Clearing the screen

You may have noticed that your programs which produce a lot of output scroll down the terminal. This keeps your program from looking like a running application. It's fairly easy to clear the terminal screen any time you want to, though.

Let's make a really simple program to show this:

```
# Show a simple message.
print("I like climbing mountains.")
```

show output

Now we will modify the program so that the screen is cleared right after the message is displayed:

```
import os

# Show a simple message.
print("I like climbing mountains.")

# Clear the screen.
os.system('clear')
```

There is no output to show here, because as soon as the message is displayed the screen is cleared. Run the program on your computer to see this.

The first line imports the **module** "os". This is a set of functions that let you interact with your operating system's commands. The line `os.system('clear')` tells Python to talk to the operating system, and ask the system to run the `clear` command. You can see this same thing by typing `clear` in any open terminal window.

On a technical note, the screen is not actually erased when you enter the `clear` command. Instead, the terminal is scrolled down one vertical window length. If you scroll the terminal window up, you can see the old output. This is fine, and it can actually be good to be able to scroll back up and look at some output you might have missed.

On Windows: The command to clear the terminal screen is different on Windows. The command `os.system('cls')` should work.

Exercises

Simple Clear

- Print one line to the screen.
- Run your program to make sure the line actually prints.
- Add a call to your system's "clear" command.
- Run your program and make sure the line disappears.
    - If it didn't work, make sure you have the line `import os` at the top of your file.

top

A Persistent Title Bar

Let's consider how to make a title bar that stays at the top of the screen. We can't make this yet, because as our program creates more and more output, any title bar we print will disappear up the top of the screen.

Now that we know how to clear the screen, we can rebuild the screen that our user sees any time we want. Let's start out by printing a title bar for Greeter. If you want to follow along, type out the following code and save it as `greeter.py`:

```
# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.


# Display a title bar.
print("\t**********************************************")
print("\t***  Greeter - Hello old and new friends!  ***")
print("\t**********************************************")
```

show output

Everything is fine so far. Let's look at what happens when we have a bunch of output below the title bar. Copy the following code onto your machine and run it.

```
# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.

# Display a title bar.
print("\t**********************************************")
print("\t***  Greeter - Hello old and new friends!  ***")
print("\t**********************************************")

# Display a bunch of output, representing a long-running program.
for x in range(0,51):
    print("\nWe have done %d fun and interesting things together!" % x)
```

I won't include the output here. All you should see are a number of lines about doing fun and interesting things together, with the title bar far up your terminal window, well above the visible portion.

The sleep() function

Sometimes it is helpful to be able to pause a program for a moment. This can slow things down, and it can let us show output in intervals. Run this code on your own machine, and see if you can understand how it works:

```
# Import the sleep function.
from time import sleep

print("I'm going to sleep now.")

# Sleep for 3 seconds.
sleep(3)

print("I woke up!")
```

show output

If you ran this code, you should have seen the message "I'm going to sleep now." Then you should have seen nothing happen for 3 seconds, and then you should have seen the message "I woke up."

The first line imports the sleep() function from the **module** time. You will be seeing more and more import statements. Basically, we are importing a function from Python's standard library of functions.

The sleep function can accept decimal inputs as well, so you can pause for as short or as long in your programs as you want.

A slowly disappearing title bar

This time we are going to print out the title bar, but then we are going to print enough output that we start to lose the title bar. Run this code on your own machine to see the full effect:

```
from time import sleep

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.

# Display a title bar.
print("\t************************************************")
print("\t***  Greeter - Hello old and new friends!  ***")
print("\t************************************************")

# Print a bunch of information, in short intervals
names = ['aaron', 'brenda', 'cyrene', 'david', 'eric']

# Print each name 5 times.
for name in names:
    # Pause for 1 second between batches, and then skip two lines.
    sleep(1)
    print("\n\n")
    for x in range(0,5):
        print(name.title())
```

You should have seen the title bar appear, and then groups of names appear every second for a while. At some point, the title bar probably disappeared. Now we are going to modify the code so that the screen is cleared each time through the loop.

```
from time import sleep
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.

# Display a title bar.
print("\t**********************************************")
print("\t***  Greeter - Hello old and new friends!  ***")
print("\t**********************************************")

# Print a bunch of information, in short intervals
names = ['aaron', 'brenda', 'cyrene', 'david', 'eric']

# Print each name 5 times.
for name in names:
    # Clear the screen before listing names.
    os.system('clear')

    # Display the title bar.
    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")


    print("\n\n")
    for x in range(0,5):
        print(name.title())

    # Pause for 1 second between batches.
    sleep(1)
```

This time you should see the same output, but you should see a steady title bar, and the new information takes the place of the old information instead of being listed below the old output. We have a "running" application!

Reducing repeated code

If you understood the section Introducing Functions, you may have noticed that we have some repeated code in the last program listing. Any time you see a significant amount of repetition, you can probably introduce a function.

The repeated code in this example involves displaying the title bar. Let's write a function to show the title bar.

```
from time import sleep
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**************************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**************************************************")
```

```
# Print a bunch of information, in short intervals
names = ['aaron', 'brenda', 'cyrene', 'david', 'eric']

# Print each name 5 times.
for name in names:
    display_title_bar()

    print("\n\n")
    for x in range(0,5):
        print(name.title())

    # Pause for 1 second between batches.
    sleep(1)
```

Again, you really need to copy this code onto your own machine and run it locally to see this in action. You should see the same behavior as before.

It's interesting to note that having good comments makes writing functions easier. The comment just before the code for displaying the title bar was "Display the title bar." This practically gives us a name for the function that will take over this job: display_title_bar. If we use short, descriptive function names that tell us exactly what the function does, it becomes much easier to follow what is happening in the program. We can see that we don't even need a comment when the function is called, because the function name itself is so informative. The name display_title_bar is much better than something like title, although you don't want to go overboard with a name such as clear_screen_and_display_title_bar.

This code gets a bit hard to read overall though, because we have several sections of code. You can use comments to visually break up your programs. Here's what that can look like in our current program:

```
from time import sleep
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.


### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")


### MAIN PROGRAM ###

# Print a bunch of information, in short intervals
names = ['aaron', 'brenda', 'cyrene', 'david', 'eric']

# Print each name 5 times.
for name in names:
    display_title_bar()

    print("\n\n")
    for x in range(0,5):
        print(name.title())

    # Pause for 1 second between batches.
    sleep(1)
```

If this is more appealing visually to you, feel free to use these kind of "section header" comments in your longer programs.

top

Building Greeter

Let's continue building Greeter. There are a couple more things we will introduce, but you should be able to follow everything we do.

We will start with the previous part where we have a persistent title bar, but we will remove the sleep functions. We will begin by offering users the choice to see a list of names, or enter a new name.

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.


### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")


### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
choice = ''
while choice != 'q':
    display_title_bar()

    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    choice = input("What would you like to do? ")

    # Respond to the user's choice.
    if choice == '1':
        print("\nHere are the people I know.\n")
    elif choice == '2':
        print("\nI can't wait to meet this person!\n")
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

If you run this code, you will see that it doesn't quite work. There are no Python errors, but the logic is a little off. We can enter choices, but the call to `display_title_bar()` at the beginning of the loop clears the screen as soon as the output is printed.

We can fix this by moving the call to `display_title_bar()`, and putting it in two places. We call `display_title_bar()` once just before we enter the loop, when the program starts running. But we also call it right after the user makes a choice, and before we respond to that choice:

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.


### FUNCTIONS ###
```

```
def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")


### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
choice = ''
display_title_bar()
while choice != 'q':

    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    choice = input("What would you like to do? ")

    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        print("\nHere are the people I know.\n")
    elif choice == '2':
        print("\nI can't wait to meet this person!\n")
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

This works, so let's put the menu into a function:

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.



### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")

def get_user_choice():
    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    return input("What would you like to do? ")


### MAIN PROGRAM ###
```

```
# Set up a loop where users can choose what they'd like to do.
choice = ''
display_title_bar()
while choice != 'q':

    choice = get_user_choice()

    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        print("\nHere are the people I know.\n")
    elif choice == '2':
        print("\nI can't wait to meet this person!\n")
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

Now, let's make it so that the program actually does something.

- We will make an empty list to store names.
- We will print names from this list in choice 1.
- We will get a new name in choice 2.
    - We will store that new name in the list of names.

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.


### FUNCTIONS ###
```

```
def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t***********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t***********************************************")

def get_user_choice():
    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    return input("What would you like to do? ")


### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
names = []

choice = ''
display_title_bar()
while choice != 'q':

    choice = get_user_choice()

    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        print("\nHere are the people I know.\n")
        for name in names:
            print(name.title())
    elif choice == '2':
        new_name = input("\nPlease tell me this person's name: ")
        names.append(new_name)
        print("\nI'm so happy to know %s!\n" % new_name.title())
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

If you run this program, you will see that it works mostly as its supposed to. But the code in lines 37-50, where we are responding to the user's choice, is starting to get crowded. Let's move the code for choice 1 and choice 2 into separate functions:

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.



### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")

def get_user_choice():
    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    return input("What would you like to do? ")

def show_names():
    # Shows the names of everyone who is already in the list.
    print("\nHere are the people I know.\n")
    for name in names:
        print(name.title())

def get_new_name():
    # Asks the user for a new name, and stores the name.
    new_name = input("\nPlease tell me this person's name: ")
    names.append(new_name)
    print("\nI'm so happy to know %s!\n" % new_name.title())

### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
names = []

choice = ''
display_title_bar()
while choice != 'q':

    choice = get_user_choice()

    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        show_names()
    elif choice == '2':
        get_new_name()
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

This code doesn't behave any differently, but the main program itself is much easier to read. Lines 48-57 are a clean series of if-elif-else statements, each of which has a clear action.

Putting each action into its own function also lets us focus on improving that action. If you look at the function get_new_name(), you might notice that we are storing the name without checking anything about it. Let's put in a simple check to make sure we don't already know about this person.

```
import os

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.



### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')

    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")
```

```
def get_user_choice():
    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    return input("What would you like to do? ")

def show_names():
    # Shows the names of everyone who is already in the list.
    print("\nHere are the people I know.\n")
    for name in names:
        print(name.title())

def get_new_name():
    # Asks the user for a new name, and stores the name if we don't already
    #  know about this person.
    new_name = input("\nPlease tell me this person's name: ")
    if new_name in names:
        print("\n%s is an old friend! Thank you, though." % new_name.title())
    else:
        names.append(new_name)
        print("\nI'm so happy to know %s!\n" % new_name.title())

### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
names = []

choice = ''
display_title_bar()
```

```
while choice != 'q':

    choice = get_user_choice()

    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        show_names()
    elif choice == '2':
        get_new_name()
    elif choice == 'q':
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

This works well. Now the program only stores new names, and names that are already in the list are greeted as old friends.

It's interesting to note that you can run this program through python tutor, and step through each line of the code. It's pretty enlightening to see exactly how the Python interpreter steps through a program like this. You can see all of the jumps from the main program to each of the functions, and the change in flow when the user makes a particular choice. It's informative to see which lines are skipped when certain if tests pass or fail.

There is one last thing we'd like Greeter to do, to consider it a basic terminal app. Let's make it remember the list of names after the program closes. To do this, let's learn about pickling using a simpler example.

top

Basic Pickling

When we "pickle" something in the physical world, we soak it in salt and vinegar so it won't rot. "Pickling" an object in Python packages it up and stores it on disk in a way that we can get it

back in its original form later. You don't want to use pickle for imporant data, but it's a good way to get started with storing data after your program closes.
Here is a simple program that asks the user for some input, and then stores the input in a list. The program dumps the list to a file using pickle, and the next time the program runs it loads that data back in. Run this program on your computer, and see if it works for you.

```
import pickle

# This program asks the user for some animals, and stores them.

# Make an empty list to store new animals in.
animals = []

# Create a loop that lets users store new animals.
new_animal = ''
while new_animal != 'quit':
    print("\nPlease tell me a new animal to remember.")
    new_animal = input("Enter 'quit' to quit: ")
    if new_animal != 'quit':
        animals.append(new_animal)

# Try to save the animals to the file 'animals.pydata'.
try:
    file_object = open('animals.pydata', 'wb')
    pickle.dump(animals, file_object)
    file_object.close()

    print("\nI will remember the following animals: ")
    for animal in animals:
        print(animal)
except Exception as e:
    print(e)
    print("\nI couldn't figure out how to store the animals. Sorry.")
```

This program uses three new things:

- **A try-except block.**
  - A try-except block is used when you think a section of code might create an error. If an error occurs in a try block, the program does not end. Instead, program execution drops into the except block.
  - In this case, we try to open a file to write out a list of animals.
  - If the file can not be opened for some reason, for example because the program doesn't have permission to create a new file, then the program drops to the except block.
    - In this case, we print the actual error message and a friendlier message of our own.
- **Opening and closing files.**
  - Line 33 tries to open the file 'animals.pydata'.
    - Line 33 tells Python to open the file for writing. The 'b' argument tells Python to write the file in bytes.
    - If successful, the open file can be used through the file_object variable.
    - If the file does not yet exist, this line creates the file, in the same directory as the program.
  - Line 35 closes the file once we are finished working with it.
- **A call to pickle.dump().**
  - Line 34 'dumps' the list animals into the file that was opened. (It is not dumped in a format that we can read.)

Note: This may work slightly differently in Python 2, and on Windows or Mac. This notebook will be updated to include those specific differences.

When you run this program on your computer, look for the file animals.pydata in the same directory as the program file. If that worked, we need to modify the program so that the next time it runs it loads the data from animals.pydata back in.

We do that with a try-except block at the beginning of the program, which tries to open the file and read the data back into the animals list. If that doesn't work, which will happen when the program is run for the first time or if you delete animals.pydata, we make the same empty list we had before.

```
import pickle

# This program asks the user for some animals, and stores them.
#  It loads animals if they exist.

# Try to load animals. If they don't exist, make an empty list
#  to store new animals in.
try:
    file_object = open('animals.pydata', 'rb')
    animals = pickle.load(file_object)
    file_object.close()
except:
    animals = []

# Show the animals that are stored so far.
if len(animals) > 0:
    print("I know the following animals: ")
    for animal in animals:
        print(animal)
else:
    print("I don't know any animals yet.")

# Create a loop that lets users store new animals.
new_animal = ''
while new_animal != 'quit':
    print("\nPlease tell me a new animal to remember.")
    new_animal = input("Enter 'quit' to quit: ")
    if new_animal != 'quit':
        animals.append(new_animal)

# Try to save the animals to the file 'animals.pydata'.
try:
    file_object = open('animals.pydata', 'wb')
    pickle.dump(animals, file_object)
    file_object.close()

    print("\nI will remember the following animals: ")
    for animal in animals:
        print(animal)
except Exception as e:
    print(e)
    print("\nI couldn't figure out how to store the animals. Sorry.")
```

The new try-except block at the beginning of the file works like this:

- It tries to open a file to read in a list of animals.
- If the file exists, animals will contain the previously stored list.
- If the file does not exist, line 9 will create an error, which would normally end the program.
- Instead, the program drops to the except block, where an empty list is created.

Now the program should just build a cumulative list of animals every time it is run. This is our first example of **persistent** data, data that lasts even after our program stops running.

Now let's use these concepts to finish Greeter.

top

Exercises

Pickling Games

- Write a program that lets users enter a number of different games.
- Save the games to disk, using pickle, before the program closes.
- Load the games from the saved file at the beginning of your program.

top

Pickling in Greeter

We really just have a little left to do in order to finish Greeter. We need to dump the list of names before the program ends, and we need to load that list of names when the program starts. Let's do this by creating two new functions:

- The function load_names() will load the names from a file when the program first starts. This function will be called before the main loop begins.
- The function quit() will dump the names into a file just before the program ends.

```
import os
import pickle

# Greeter is a terminal application that greets old friends warmly,
#   and remembers new friends.



### FUNCTIONS ###

def display_title_bar():
    # Clears the terminal screen, and displays a title bar.
    os.system('clear')
    print("\t**********************************************")
    print("\t***  Greeter - Hello old and new friends!  ***")
    print("\t**********************************************")

def get_user_choice():
    # Let users know what they can do.
    print("\n[1] See a list of friends.")
    print("[2] Tell me about someone new.")
    print("[q] Quit.")

    return input("What would you like to do? ")

def show_names():
    # Shows the names of everyone who is already in the list.
    print("\nHere are the people I know.\n")
    for name in names:
        print(name.title())

def get_new_name():
    # Asks the user for a new name, and stores the name if we don't already
    #  know about this person.
    new_name = input("\nPlease tell me this person's name: ")
    if new_name in names:
        print("\n%s is an old friend! Thank you, though." % new_name.title())
    else:
        names.append(new_name)
        print("\nI'm so happy to know %s!\n" % new_name.title())

def load_names():
    # This function loads names from a file, and puts them in the list 'names'.
    #  If the file doesn't exist, it creates an empty list.
    try:
        file_object = open('names.pydata', 'rb')
        names = pickle.load(file_object)
        file_object.close()
        return names
    except Exception as e:
        print(e)
        return []
```

```
def quit():
    # This function dumps the names into a file, and prints a quit message.
    try:
        file_object = open('names.pydata', 'wb')
        pickle.dump(names, file_object)
        file_object.close()
        print("\nThanks for playing. I will remember these good friends.")
    except Exception as e:
        print("\nThanks for playing. I won't be able to remember these names.")
        print(e)
```

```
### MAIN PROGRAM ###

# Set up a loop where users can choose what they'd like to do.
names = load_names()

choice = ''
display_title_bar()
while choice != 'q':

    choice = get_user_choice()
    # Respond to the user's choice.
    display_title_bar()
    if choice == '1':
        show_names()
    elif choice == '2':
        get_new_name()
    elif choice == 'q':
        quit()
        print("\nThanks for playing. Bye.")
    else:
        print("\nI didn't understand that choice.\n")
```

We now have a long-running, standalone terminal application. It doesn't do a whole lot, but it shows the basic structure for creating terminal apps of your own. If you expand the list of choices and keep your code organized into clean, simple functions, you now know enough to make some pretty interesting programs.

Python - GUI Programming

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below.

- **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look at this option in this chapter.
- **wxPython** − This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython here.
- **PyQt** − This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt5 here.
- **PyGTK** − PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available here.
- **PySimpleGUI** − PySimpleGui is an open source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide and WxPython toolkits. For a detaile PySimpleGUI tutorial, click here.

- **Pygame** − Pygame is a popular Python library used for developing video games. It is free, open source and cross-platform wrapper around Simple DirectMedia Library (SDL). For a comprehensive tutorial on Pygame, visit this link.
- **Jython** − Jython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machinehttp: //www.jython.org.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

The tkinter package includes following modules −

- **Tkinter** − Main Tkinter module.
- **tkinter.colorchooser** − Dialog to let the user choose a color.
- **tkinter.commondialog** − Base class for the dialogs defined in the other modules listed here.
- **tkinter.filedialog** − Common dialogs to allow the user to specify a file to open or save.
- **tkinter.font** − Utilities to help work with fonts.
- **tkinter.messagebox** − Access to standard Tk dialog boxes.
- **tkinter.scrolledtext** − Text widget with a vertical scroll bar built in.
- **tkinter.simpledialog** − Basic dialogs and convenience functions.
- **tkinter.ttk** − Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main tkinter module.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps.

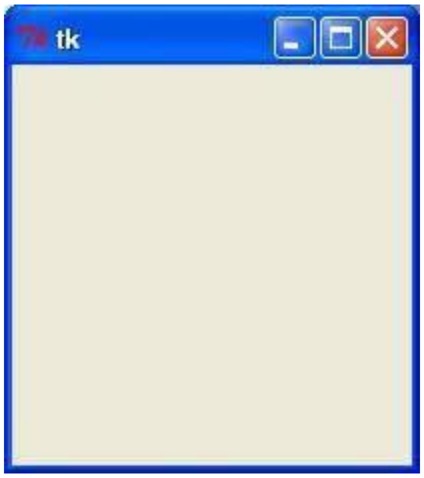
- Import the Tkinter module.
- Create the GUI application main window.

- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example

```
# note that module name has changed from Tkinter in Python 2
# to tkinter in Python 3

import tkinter
top = tkinter.Tk()

# Code to add widgets will go here...
top.mainloop()
```

This would create a following window −



When the program becomes more complex, using an object-oriented programming approach makes the code more organized.

```
import tkinter as tk
class App(tk.Tk):
  def __init__(self):
    super().__init__()
```

```
app = App()
app.mainloop()
```

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

Sr.No.	Operator & Description
1	<u>**Button**</u> The Button widget is used to display the buttons in your application.
2	<u>**Canvas**</u> The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
3	<u>**Checkbutton**</u> The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
4	<u>**Entry**</u> The Entry widget is used to display a single-line text field for accepting values from a user.
5	<u>**Frame**</u> The Frame widget is used as a container widget to organize other widgets.

6	**Label** The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
7	**Listbox** The Listbox widget is used to provide a list of options to a user.
8	**Menubutton** The Menubutton widget is used to display menus in your application.
9	**Menu** The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
10	**Message** The Message widget is used to display multiline text fields for accepting values from a user.
11	**Radiobutton** The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
12	**Scale** The Scale widget is used to provide a slider widget.
13	**Scrollbar** The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.

14	**Text** The Text widget is used to display text in multiple lines.
15	**Toplevel** The Toplevel widget is used to provide a separate window container.
16	**Spinbox** The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
17	**PanedWindow** A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
18	**LabelFrame** A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
19	**tkMessageBox** This module is used to display message boxes in your applications.

Let us study these widgets in detail.

Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors

- Relief styles
- Bitmaps
- Cursors

Let us study them briefly −

Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- **The pack() Method** − This geometry manager organizes widgets in blocks before placing them in the parent widget.
- **The grid() Method** − This geometry manager organizes widgets in a table-like structure in the parent widget.
- **The place() Method** − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly −

SimpleDialog

The simpledialog module in tkinter package includes a dialog class and convenience functions for accepting user input through a modal dialog. It consists of a label, an entry widget and two buttons Ok and Cancel. These functions are −

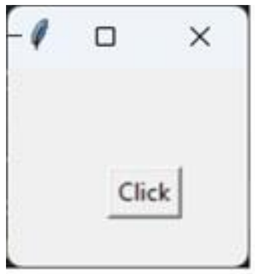
- **askfloat(title, prompt, **kw)** − Accepts a floating point number.
- **askinteger(title, prompt, **kw)** − Accepts an integer input.
- **askstring(title, prompt, **kw)** − Accepts a text input from the user.

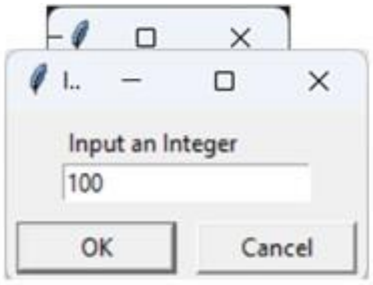
The above three functions provide dialogs that prompt the user to enter a value of the desired type. If Ok is pressed, the input is returned, if Cancel is pressed, None is returned.

askinteger

```
from tkinter.simpledialog import askinteger
from tkinter import *
from tkinter import messagebox
top = Tk()

top.geometry("100x100")
def show():
   num = askinteger("Input", "Input an Integer")
   print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** −





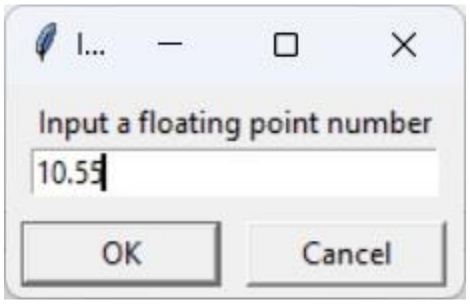
Input an Integer

100

askfloat

```
from tkinter.simpledialog import askfloat
from tkinter import *
top = Tk()

top.geometry("100x100")
def show():
   num = askfloat("Input", "Input a floating point number")
   print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

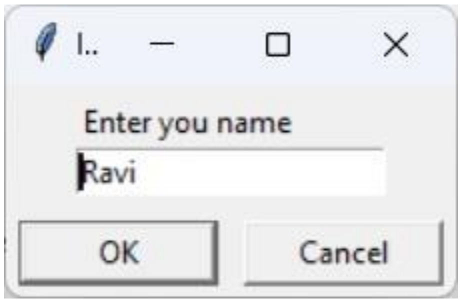
It will produce the following **output** −



askstring

```
from tkinter.simpledialog import askstring
from tkinter import *

top = Tk()

top.geometry("100x100")
def show():
   name = askstring("Input", "Enter you name")
   print(name)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** −



The FileDialog Module

The filedialog module in Tkinter package includes a FileDialog class. It also defines convenience functions that enable the user to perform open file, save file, and open directory activities.

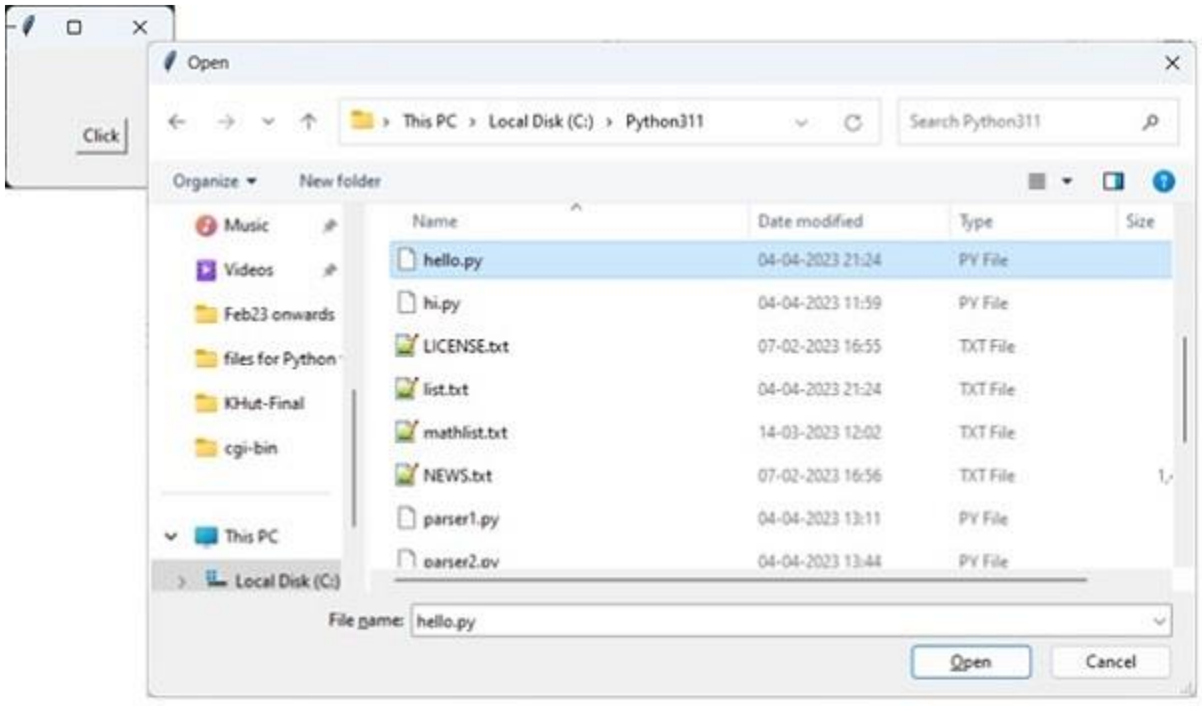
- filedialog.asksaveasfilename()
- filedialog.asksaveasfile()
- filedialog.askopenfilename()
- filedialog.askopenfile()
- filedialog.askdirectory()
- filedialog.askopenfilenames()
- filedialog.askopenfiles()

askopenfile

This function lets the user choose a desired file from the filesystem. The file dialog window has Open and Cancel buttons. The file name along with its path is returned when Ok is pressed, None if Cancel is pressed.

```
from tkinter.filedialog import askopenfile
from tkinter import *

top = Tk()

top.geometry("100x100")
def show():
   filename = askopenfile()
   print(filename)


B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)


top.mainloop()
```

It will produce the following **output** −



Name	Date modified	Type	Size
hello.py	04-04-2023 21:24	PY File	
hi.py	04-04-2023 11:59	PY File	
LICENSE.txt	07-02-2023 16:55	TXT File	
list.txt	04-04-2023 21:24	TXT File	
mathlist.txt	14-03-2023 12:02	TXT File	
NEWS.txt	07-02-2023 16:56	TXT File	1,
parser1.py	04-04-2023 13:11	PY File	
parser2.py	04-04-2023 13:44	PY File	

File name: hello.py

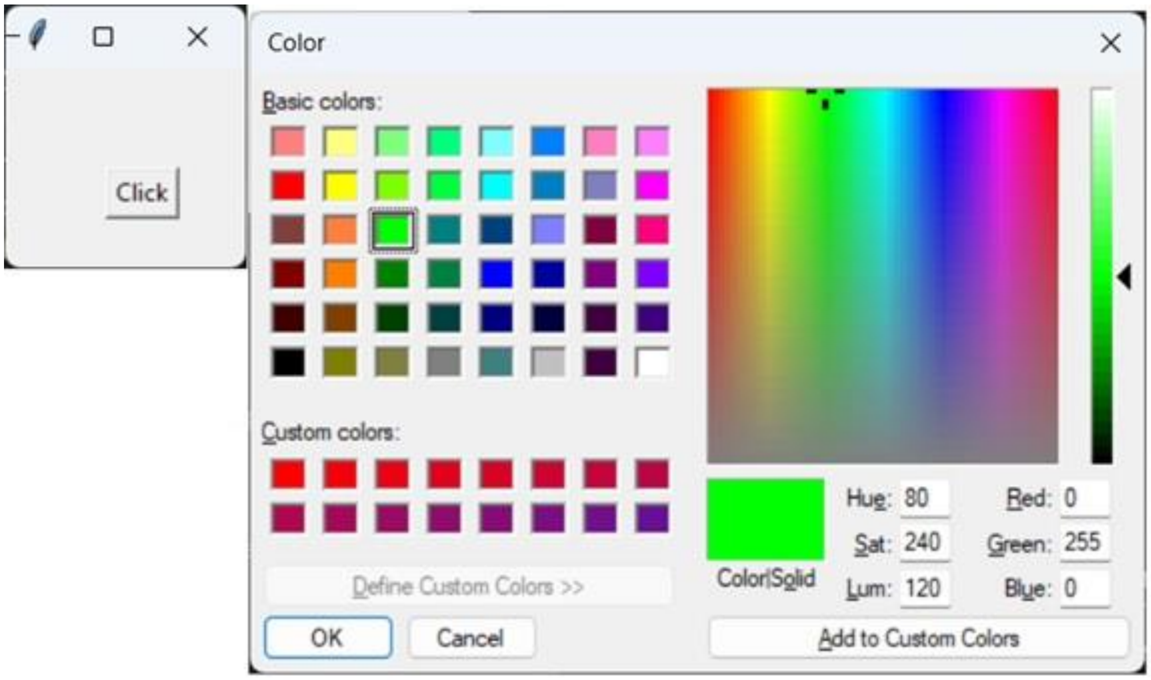
ColorChooser

The colorchooser module included in tkinter package has the feature of letting the user choose a desired color object through the color dialog. The askcolor() function presents with the color dialog with predefined color swatches and facility to choose custome color by setting RGB values. The dialog returns a tuple of RGB values of chosen color as well as its hex value.

```
from tkinter.colorchooser import askcolor
from tkinter import *
```

```
top = Tk()


top.geometry("100x100")
def show():
   color = askcolor()
   print(color)


B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)


top.mainloop()
```

It will produce the following **output** −



Hue: 80
Sat: 240
Lum: 120
Red: 0
Green: 255
Blue: 0

```
((0, 255, 0), '#00ff00')
```

ttk module

The term ttk stands from Tk Themed widgets. The ttk module was introduced with Tk 8.5 onwards. It provides additional benefits including anti-aliased font rendering under X11 and window transparency. It provides theming and styling support for Tkinter.

The ttk module comes bundled with 18 widgets, out of which 12 are already present in Tkinter. Importing ttk over-writes these widgets with new ones which are designed to have a better and more modern look across all platforms.

The 6 new widgets in ttk are, the Combobox, Separator, Sizegrip, Treeview, Notebook and ProgressBar.

To override the basic Tk widgets, the import should follow the Tk import −

```
from tkinter import *
from tkinter.ttk import *
```

The original Tk widgets are automatically replaced by tkinter.ttk widgets. They are Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar.

New widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the ttk.Style class for improved styling effects.

The new widgets in ttk module are −

- **Notebook** − This widget manages a collection of "tabs" between which you can swap, changing the currently displayed window.
- **ProgressBar** − This widget is used to show progress or the loading process through the use of animations.
- **Separator** − Used to separate different widgets using a separator line.

- **Treeview** − This widget is used to group together items in a tree-like hierarchy. Each item has a textual label, an optional image, and an optional list of data values.
- **ComboBox** − Used to create a dropdown list of options from which the user can select one.
- **Sizegrip** − Creates a little handle near the bottom-right of the screen, which can be used to resize the window.

Combobox Widget

The Python ttk Combobox presents a drop down list of options and displays them one at a time. It is a sub class of the widget Entry. Hence it inherits many options and methods from the Entry class.

Syntax

```
from tkinter import ttk

Combo = ttk.Combobox(master, values.......)
```

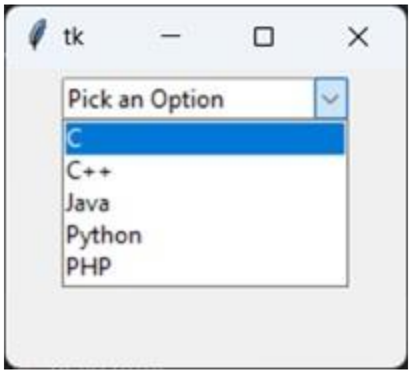
The get() function to retrieve the current value of the Combobox.

Example

```
from tkinter import *
from tkinter import ttk

top = Tk()
top.geometry("200x150")

frame = Frame(top)
frame.pack()

langs = ["C", "C++", "Java",
   "Python", "PHP"]
```

```
Combo = ttk.Combobox(frame, values = langs)
Combo.set("Pick an Option")
Combo.pack(padx = 5, pady = 5)
top.mainloop()
```

It will produce the following **output** −



Progressbar

The ttk ProgressBar widget, and how it can be used to create loading screens or show the progress of a current task.

Syntax

```
ttk.Progressbar(parent, orient, length, mode)
```

Parameters

- **Parent** − The container in which the ProgressBar is to be placed, such as root or a Tkinter frame.
- **Orient** − Defines the orientation of the ProgressBar, which can be either vertical of horizontal.
- **Length** − Defines the width of the ProgressBar by taking in an integer value.
- **Mode** − There are two options for this parameter, determinate and indeterminate.

Example

The code given below creates a progressbar with three buttons which are linked to three different functions.

The first function increments the "value" or "progress" in the progressbar by 20. This is done with the step() function which takes an integer value to change progress amount. (Default is 1.0)

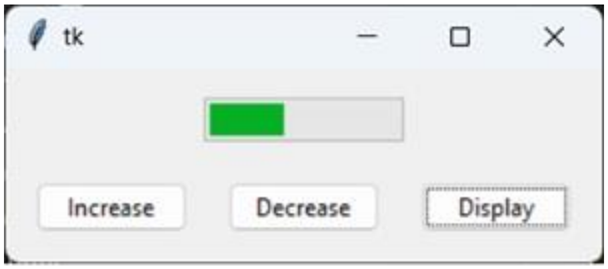
The second function decrements the "value" or "progress" in the progressbar by 20.

The third function prints out the current progress level in the progressbar.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
frame= ttk.Frame(root)
def increment():
   progressBar.step(20)

def decrement():
   progressBar.step(-20)

def display():
   print(progressBar["value"])

progressBar= ttk.Progressbar(frame, mode='determinate')
progressBar.pack(padx = 10, pady = 10)

button= ttk.Button(frame, text= "Increase", command= increment)
button.pack(padx = 10, pady = 10, side = tk.LEFT)

button= ttk.Button(frame, text= "Decrease", command= decrement)
```

```
button.pack(padx = 10, pady = 10, side = tk.LEFT)
button= ttk.Button(frame, text= "Display", command= display)
button.pack(padx = 10, pady = 10, side = tk.LEFT)

frame.pack(padx = 5, pady = 5)
root.mainloop()
```

It will produce the following **output** −



Notebook

Tkinter ttk module has a new useful widget called Notebook. It is a of collection of of containers (e.g frames) which have many widgets as children inside.

Each "tab" or "window" has a tab ID associated with it, which is used to determine which tab to swap to.

You can swap between these containers like you would on a regular text editor.

Syntax

```
notebook = ttk.Notebook(master, *options)
```

Example

In this example, add 3 windows to our Notebook widget in two different ways. The first method involves the add() function, which simply appends a new tab to the end. The other method is the insert() function which can be used to add a tab to a specific position.

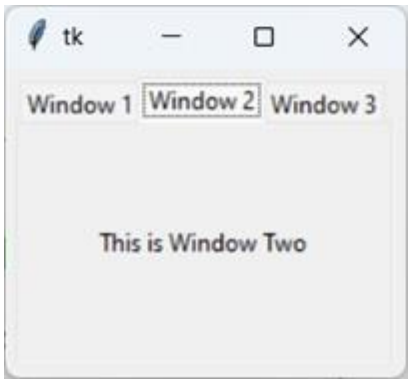
The add() function takes one mandatory parameter which is the container widget to be added, and the rest are optional parameters such as text (text to be displayed as tab title), image and compound.

The insert() function requires a tab_id, which defines the location where it should be inserted. The tab_id can be either an index value or it can be string literal like "end", which will append it to the end.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
nb = ttk.Notebook(root)

# Frame 1 and 2
frame1 = ttk.Frame(nb)
frame2 = ttk.Frame(nb)

label1 = ttk.Label(frame1, text = "This is Window One")
label1.pack(pady = 50, padx = 20)
label2 = ttk.Label(frame2, text = "This is Window Two")
label2.pack(pady = 50, padx = 20)

frame1.pack(fill= tk.BOTH, expand=True)
frame2.pack(fill= tk.BOTH, expand=True)
nb.add(frame1, text = "Window 1")
nb.add(frame2, text = "Window 2")

frame3 = ttk.Frame(nb)
label3 = ttk.Label(frame3, text = "This is Window Three")
label3.pack(pady = 50, padx = 20)
frame3.pack(fill= tk.BOTH, expand=True)
```

```
nb.insert("end", frame3, text = "Window 3")
nb.pack(padx = 5, pady = 5, expand = True)

root.mainloop()
```

It will produce the following **output** −



Treeview

The Treeview widget is used to display items in a tabular or hierarchical manner. It has support for features like creating rows and columns for items, as well as allowing items to have children as well, leading to a hierarchical format.

Syntax

```
tree = ttk.Treeview(container, **options)
```

Options

Sr.No.	Option & Description
1	**columns** A list of column names
2	**displaycolumns** A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the

	string "#all".
3	**height** The number of rows visible.
4	**padding** Specifies the internal padding for the widget. Can be either an integer or a list of 4 values.
5	**selectmode** One of "extended", "browse" or "none". If set to "extended" (default), multiple items can be selected. If "browse", only a single item can be selected at a time. If "none", the selection cannot be changed by the user.
6	**show** A list containing zero or more of the following values, specifying which elements of the tree to display. The default is "tree headings", i.e., show all elements.

Example

In this example we will create a simple Treeview ttk Widget and fill in some data into it. We have some data already stored in a list which will be reading and adding to the Treeview widget in our read_data() function.

We first need to define a list/tuple of column names. We have left out the column "Name" because there already exists a (default) column with a blank name.

We then assign that list/tuple to the columns option in Treeview, followed by defining the "headings", where the column is the actual column, whereas the heading is just the title of the column that appears when the widget is displayed. We give each a column a name. "#0" is the name of the default column.

The tree.insert() function has the following parameters −

- **Parent** − which is left as an empty string if there is none.
- **Position** − where we want to add the new item. To append, use tk.END
- **Iid** − which is the item ID used to later track the item in question.
- **Text** − to which we will assign the first value in the list (the name).

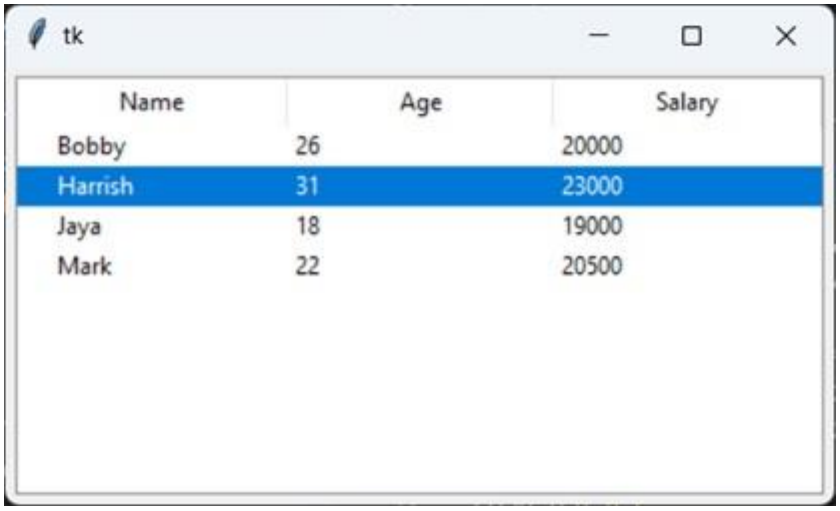
Value we will pass the the other 2 values we obtained from the list.

The Complete Code

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import simpledialog

root = tk.Tk()
data = [
  ["Bobby",26,20000],
  ["Harrish",31,23000],
  ["Jaya",18,19000],
  ["Mark",22, 20500],
]
index=0
def read_data():
  for index, line in enumerate(data):
    tree.insert('', tk.END, iid = index,
      text = line[0], values = line[1:])
columns = ("age", "salary")

tree= ttk.Treeview(root, columns=columns ,height = 20)
tree.pack(padx = 5, pady = 5)

tree.heading('#0', text='Name')
tree.heading('age', text='Age')
tree.heading('salary', text='Salary')
```

```
read_data()
root.mainloop()
```

It will produce the following **output** –



```
Name      Age   Salary
Bobby     26    20000
Harrish   31    23000
Jaya      18    19000
Mark      22    20500
```

Sizegrip

The Sizegrip widget is basically a small arrow-like grip that is typically placed at the bottom-right corner of the screen. Dragging the Sizegrip across the screen also resizes the container to which it is attached to.

Syntax

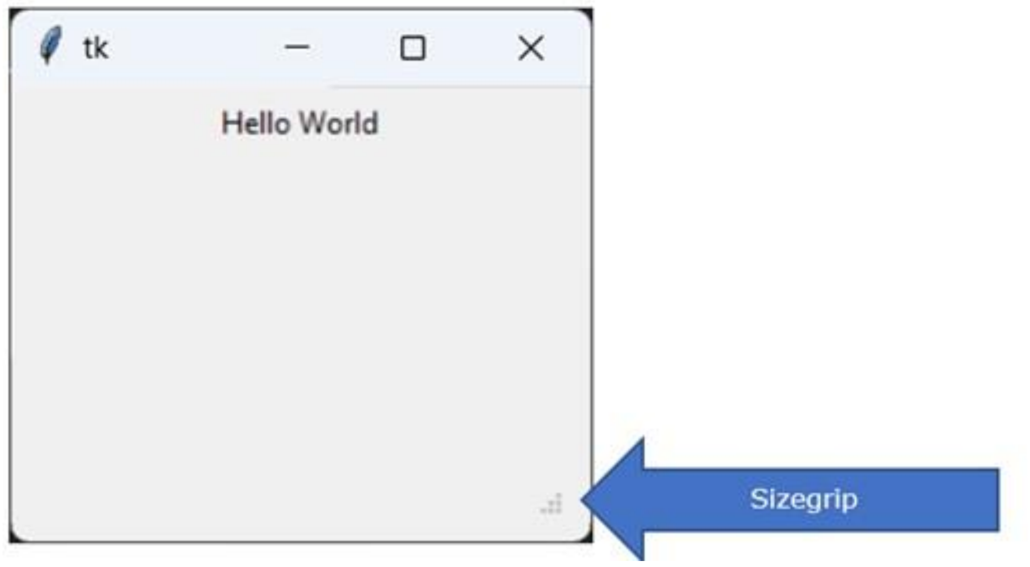
```
sizegrip = ttk.Sizegrip(parent, **options)
```

Example

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("100x100")

frame = ttk.Frame(root)
```

```
label = ttk.Label(root, text = "Hello World")
label.pack(padx = 5, pady = 5)
sizegrip = ttk.Sizegrip(frame)
sizegrip.pack(expand = True, fill = tk.BOTH, anchor = tk.SE)
frame.pack(padx = 10, pady = 10, expand = True, fill = tk.BOTH)

root.mainloop()
```

It will produce the following **output** −



Separator

The ttk Separator widget is a very simple widget, that has just one purpose and that is to help "separate" widgets into groups/partitions by drawing a line between them. We can change the orientation of this line (separator) to either horizontal or vertical, and change its length/height.

Syntax

```
separator = ttk.Separator(parent, **options)
```

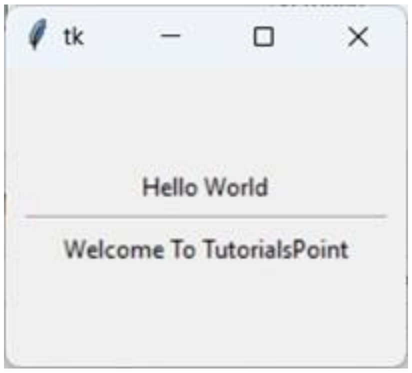
The "orient", which can either be tk.VERTICAL or tk.HORIZTONAL, for a vertical and horizontal separator respectively.

Example

Here we have created two Label widgets, and then created a Horizontal Separator between them.

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("200x150")

frame = ttk.Frame(root)

label = ttk.Label(frame, text = "Hello World")
label.pack(padx = 5)

separator = ttk.Separator(frame,orient= tk.HORIZONTAL)
separator.pack(expand = True, fill = tk.X)

label = ttk.Label(frame, text = "Welcome To TutorialsPoint")
label.pack(padx = 5)

frame.pack(padx = 10, pady = 50, expand = True, fill = tk.BOTH)

root.mainloop()
```

It will produce the following **output** −



Hello World

Welcome To TutorialsPoint

What is the Python window?

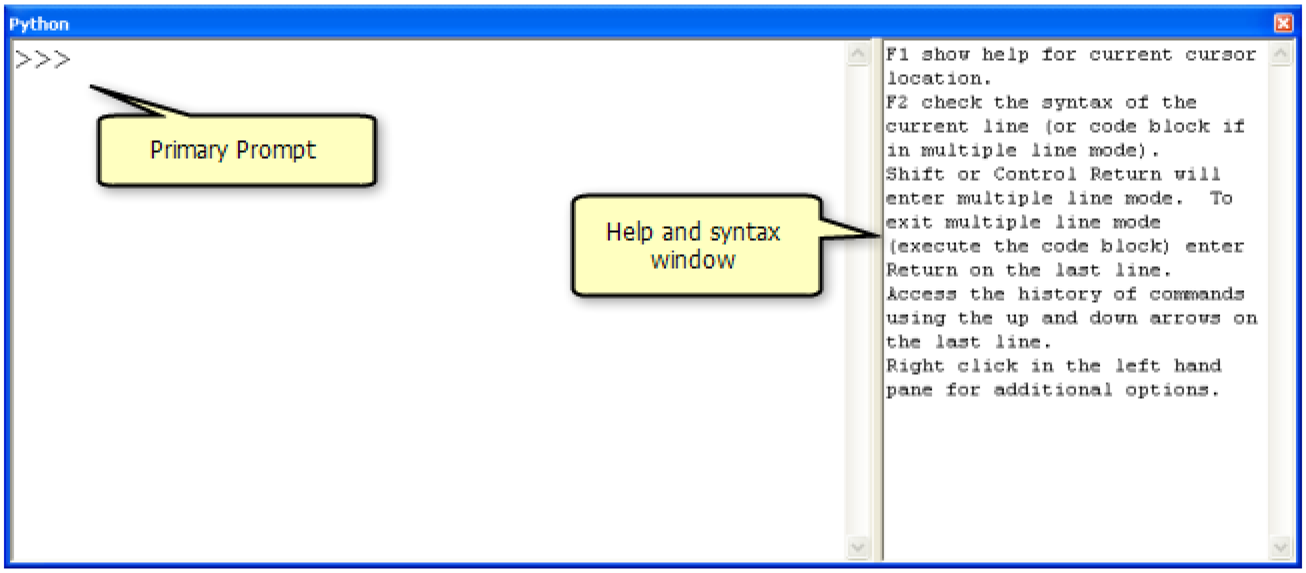
The **Python** window is an efficient and convenient location to use geoprocessing tools and Python functionality from within ArcGIS. The Python commands run from this window can range from single lines of code to complex blocks with logic. The **Python** window also provides a place to access additional functionality using custom or third-party Python modules and libraries.

The **Python** window also acts as a gateway to Python for those new to scripting, providing an intuitive interface that makes learning Python scripting in ArcGIS as easy as possible. In the **Python** window, various tool and method usage and syntax can be viewed or experimented with, and snippets of Python code can be entered or pasted into the window to test scripting ideas outside a stand-alone script. The **Python** window is a powerful mechanism for efficiently accessing and executing geoprocessing and scripting tasks and increasing productivity by placing Python functionality within ArcGIS Desktop applications.

Opening the Python window

The **Python** window can be opened within any ArcGIS Desktop application by clicking the Python window button on the Standard toolbar.

Below shows how the **Python** window appears when first opened:



```
>>>
```

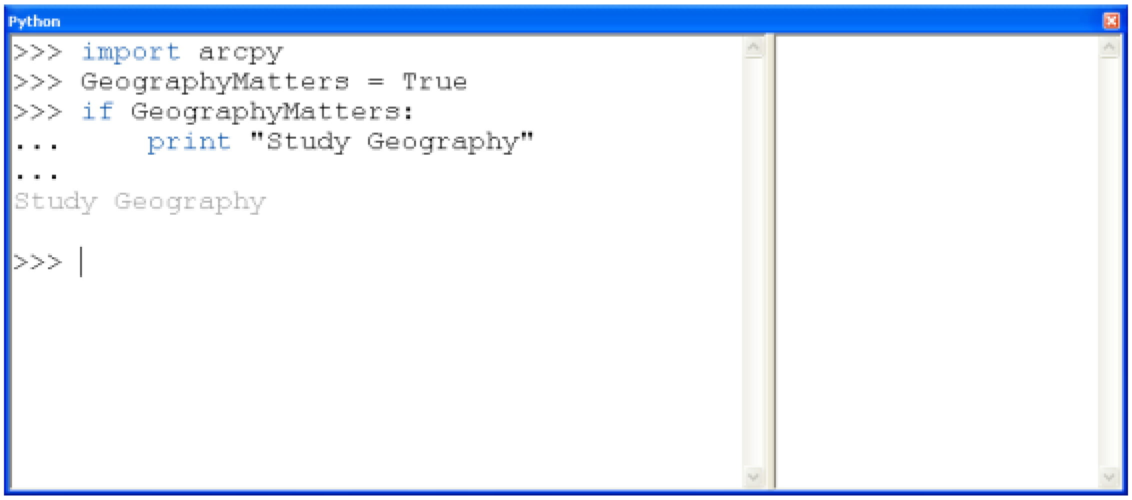
```
F1 show help for current cursor
location.
F2 check the syntax of the
current line (or code block if
in multiple line mode).
Shift or Control Return will
enter multiple line mode.  To
exit multiple line mode
(execute the code block) enter
Return on the last line.
Access the history of commands
using the up and down arrows on
the last line.
Right click in the left hand
pane for additional options.
```

An overview of the Python window

Once opened, the **Python** window can be moved by clicking the bar at the top and dragging to your preferred location. The window can be docked or undocked.

The window prompts for the next command with the primary prompt, three greater-than signs (>>>), and continuation lines prompt with the secondary prompt, three dots (**...**).

Continuation lines are needed when entering a multiline construct. See the following example using an **if** statement:



```
>>> import arcpy
>>> GeographyMatters = True
>>> if GeographyMatters:
...     print "Study Geography"
...
Study Geography

>>> 
```

The **Python** window contains two sections:

- The Python section on the left. This is where commands are entered.
- The Help section on the right. This is where command usage, help, and execution messages are viewed. This section can be hidden or placed to the right, left, top, or bottom of the Python section.

Components of Python Language

- A Components of Python Language is the smallest element in a program that is meaningful to the computer.
- These Components of Python Language define the structure of the language.
- It is also known as a token of python language.

The five main components (tokens) of 'Python' language are:

- The character set
- The Data types
- Constants
- Variables
- Keywords

The character set

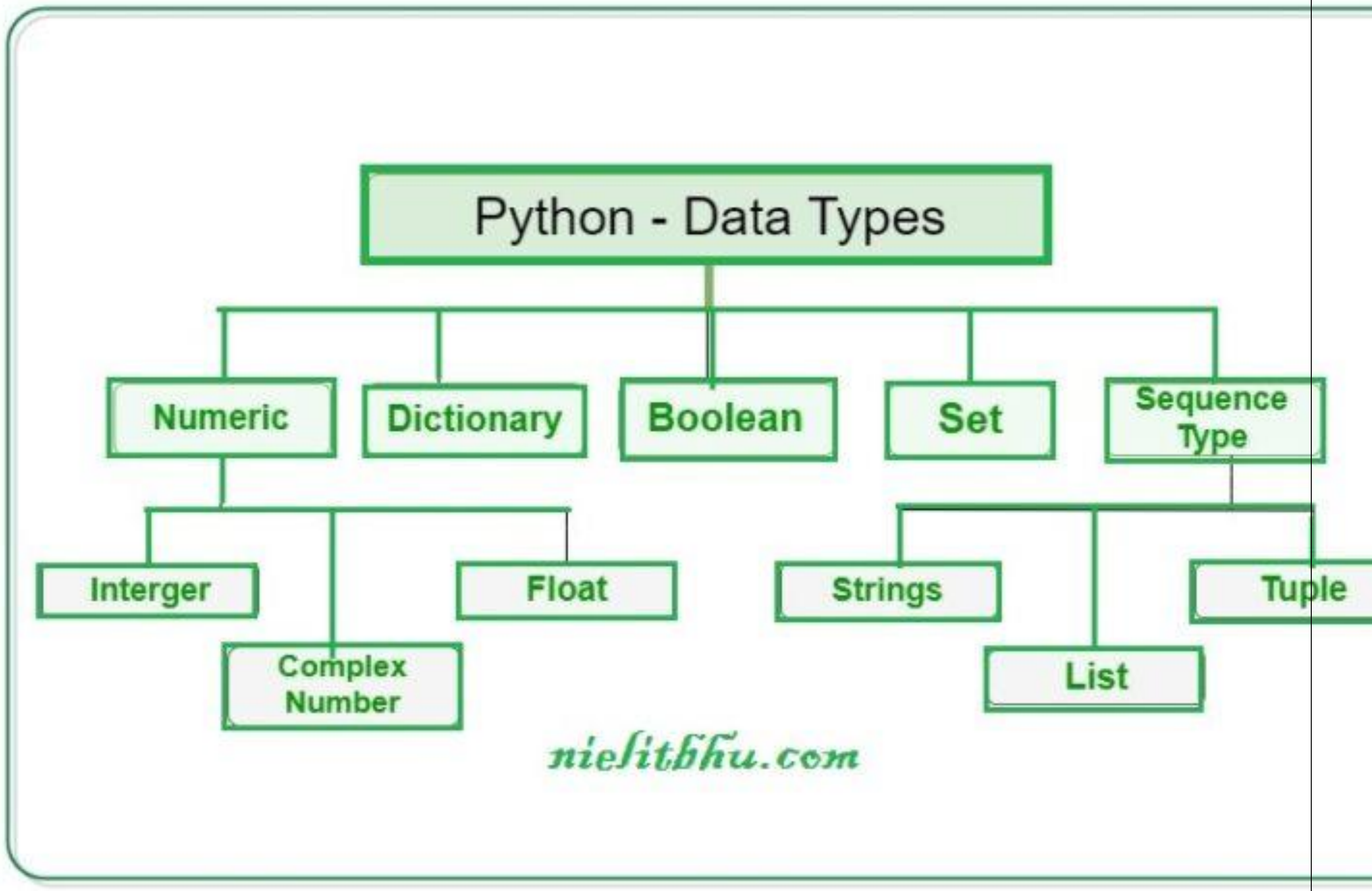
- Any alphabet, digits, or special symbol used to represent information is denoted by character.
- The characters in Pythons are grouped into four categories.

1	Letters	A – – – Z or a – z
2	Digits	0,1,—-9
3	Special Symbols	-.‘‘@#%'’’ &*() _-+ = I\{}[]:;”’‘< > , . ? /.

4	White spaces	blank space, horizontal tab, carriage return, new line, and form feed.

The Data types

- The power of a programming language depends, among other things, on the range of different types of data it can handle.
- Data values passed in a program may be of different types.



Note: Each data type will be learned in each different chapter.

Find the Data Type

type() function: using this function get the data type of any object or variable.

Example

y = 15

print(type(y))

Constants

- Constants are the fixed values that remain unchanged during the execution of a program and are used in assignment statements.

Variables

- Variables are the data items whose values may vary during the execution of the program.

Note: Python has no command for declaring a variable.

Rules to defines Variable Names

- A variable can have a short name (like x and y) or a more descriptive name (age, surname, 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 (**ram, Ram, and RAM are three different variables**)

Example to create variable in python

a = "NIELITBHU"

a_var = "NIELITBHU"

_a_var = "NIELITBHU"

aVar = "NIELITBHU"

aVAR = "NIELITBHU"

avar2 = "NIELITBHU"

Keywords

- Keywords are the words that have been **assigned specific meaning in the context** of python language programs.
- Keywords should **not be used as variable names** to avoid problems.
- There are **35 keywords** are found in the python programming language.

and	**continue**	**for**	**lambda**	**try**

as	**def**	**from**	**nonlocal**	**while**
assert	**del**	**global**	**not**	**with**
async	**elif**	**if**	**or**	**yield**
False	**await**	**else**	**import**	**pass**
None	**break**	**except**	**in**	**raise**
True	**class**	**finally**	**is**	**return**

Python full course

<u>What is Python</u>

Component of Python Language

Python Syntax

Python Operators

Comparison Operators

Logical operators

Python Identity operators

Python Membership operators

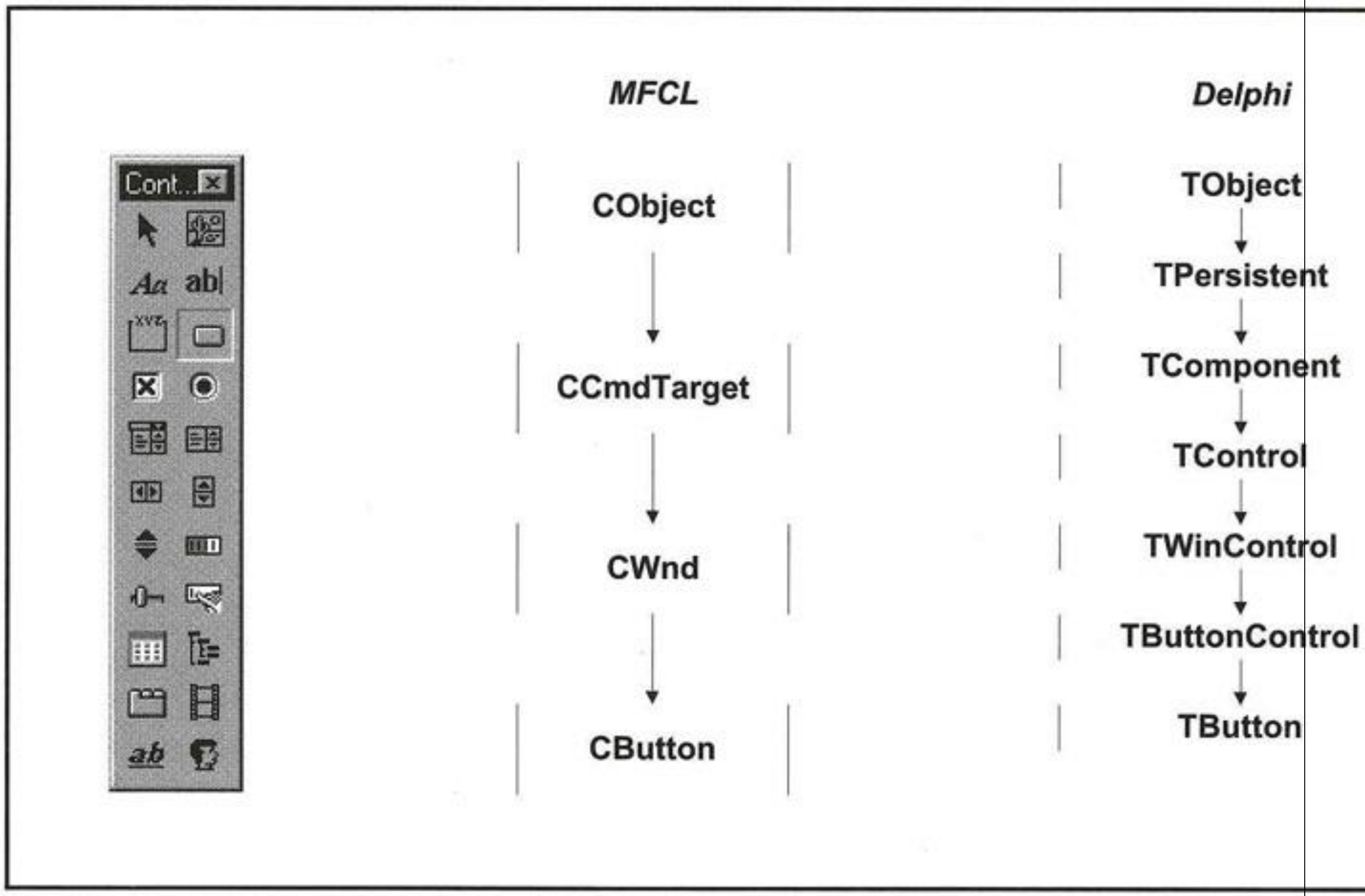
Python Bitwise operators

Python Comments

- ‹
- 1
- 2
- 3
- ›

Command Buttons

Command buttons give the user the opportunity to initiate an event (Figure 7.6). For example, the familiar OK and Cancel buttons are signals from the user that details in a dialog box are correct or that an action is to be abandoned. Buttons are also used to provide other choices for a user and a series of buttons may be placed on a toolbar to give the user fast access to actions that must otherwise be activated through the menu system. The class of command buttons is the base class for derivatives such as check boxes and radio buttons.



Command buttons

Command buttons consist of a rectangle with a caption. Usually, command buttons are created in such a way that a mouse click temporarily changes the button picture, giving the impression of a button being pressed, then released.

Command buttons respond to two major events: the mouse button click and double-click (though Visual Basic restricts this to click events only). In addition, you can code for events such as the mouse button being pressed or released, dragging the pointer over the button's "airspace," or receiving and losing the focus.

There is a special Default property that can be applied to one (and only one) button on a dialog box or window. This is determined by the BS_DEFPUSHBUTTON style, which identifies the control as the default button. The default button has a thicker border, and pressing the ENTER key has the same effect as...