

## PYTHON PROGRAMMING

### UNIT –V

**Errors and Exceptions:** Syntax Errors, Exceptions, Handling Exceptions, Raising Exceptions, User-defined Exceptions, Defining Clean-up Actions, Redefined Clean-up Actions.

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

**Programming:** Introduction to Programming Concepts with Scratch.

### Errors and Exceptions:

As human beings, we commit several errors. A software developer is also a human being and hence prone to commit errors with in the design of the software or in writing the code. The errors in the software are called „bugs“ and the process of removing them are called „debugging“. In general, we can classify errors in a program into one of these three types:

- a) Compile-time errors
- b) Runtime errors
- c) Logical errors

#### a) Compile-time errors or parsing error.

These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a colon in the statements like if, while, for, def, etc. will result in compile-time error. Such errors are detected by python compiler and the line number along with error description is displayed by the python compiler.

#### Common Python syntax errors include:

- leaving out a keyword
- putting a keyword in the wrong place
- leaving out a symbol, such as a colon, comma or brackets
- misspelling a keyword
- incorrect indentation
- empty block

#### Example: A Python program to understand the compile-time error. a = 1

```
if a == 1
print "hello"
```

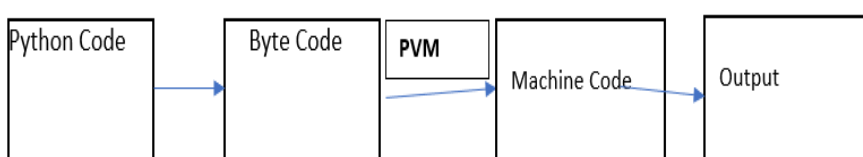
#### Output:

```
File ex.py, line 3 If a == 1
^
SyntaxError: invalid syntax
```

#### b) Runtime errors

When PVM (Python Virtual Memory ) cannot execute the byte code, it flags runtime error. For example, insufficient memory to store something or inability of PVM to execute some statement come under runtime errors. Runtime errors are not detected by the python compiler. They are detected by the PVM, Only at runtime.

PVM is also called Python Interpreter and this is the reason Python is called an Interpreted language.



**Some examples of Python runtime errors:**

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn't exist
- trying to access a file which doesn't exist

**Example: A Python program to understand the compile-time error.**

```
print "hai"+25
```

**Output:**

Traceback (most recent call last):

```
File "<pyshell#0>", line 1, in <module>
print "hai"+25
```

TypeError: cannot concatenate 'str' and 'int' objects

**c) Logical errors**

These errors depict flaws in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Python compiler or PVM. The programmer is solely responsible for them. In the following program, the programmer wants to calculate incremented salary of an employee, but he gets wrong output, since he uses wrong formula.

**some examples of mistakes which lead to logical errors:**

- using the wrong variable name
- indenting a block to the wrong level
- using integer division instead of floating-point division
- getting operator precedence wrong
- making a mistake in a boolean expression
- off-by-one, and other numerical errors

**Example: A Python program to increment the salary of an employee by 15%.**

```
def increment(sal):
    sal = sal * 15/100
    return sal
sal = increment(5000)
print "Salary after Increment is", sal
```

**Output:**

Salary after Increment is 750

From the above program the formula for salary is wrong, because only the increment but it is not adding it to the original salary. So, the correct formula would be:

```
sal = sal + sal * 15/100
```

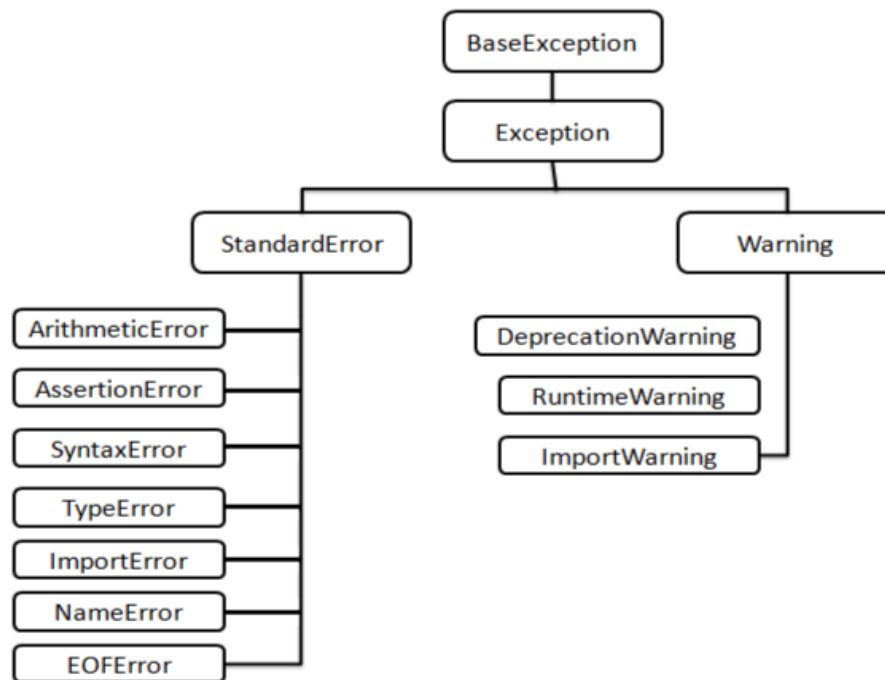
Compile time errors and runtime errors can be eliminated by the programmer by modifying the program source code.

In case of runtime errors, when the programmer knows which type of error occurs, he must handle them using exception handling mechanism.

**Exceptions:**

- An exception is a **runtime error** which can be handled by the programmer.
- That means if the programmer can guess an error in the program and he can do something to eliminate the harm caused by that error, then it is called an „exception“.
- If the programmer cannot do anything in case of an error, then it is called an „error“ and not an exception.
- All exceptions are represented as classes in python. The exceptions which are already available in python are called „built-in“ exceptions. The base class for all built-in exceptions is „BaseException“ class.
- From BaseException class, the sub class „Exception“ is derived. From Exception class, the sub classes „StandardError“ and „Warning“ are derived.

- All errors (or exceptions) are defined as sub classes of Standard Error. An error should be compulsory handled otherwise the program will not execute.
- Similarly, all warnings are derived as sub classes from „Warning“ class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot neglect.
- Just like the exceptions which are already available in python language, a programmer can also create his own exceptions, called „user-defined“ exceptions.
- When the programmer wants to create his own exception class, he should derive his class from Exception class and not from “BaseException” class.



### Exception Handling:

- The purpose of handling errors is to make the program *robust*. The word “robust” means “strong”. A robust program does not terminate in the middle.
- Also, when there is an error in the program, it will display an appropriate message to the user and continue execution. Designing such programs is needed in any software development.
- For that purpose, the programmer should handle the errors. When the errors can be handled, they are called exceptions.
- To handle exceptions, the programmer should perform the following four steps:

**Step 1:** The programmer should observe the statements in his program where there may be a possibility of exceptions. Such statements should be written inside a „try“ block. A try block looks like as follows:

*try:*

*statements*

The PVM understands that there is an exception in try block, it jumps into an „except“ block.

**Step 2:** The programmer should write the „except“ block where he should display the exception details to the user. Except block looks like as follows:

```
except exceptionname:  
    statements
```

The statements written inside an except block are called „handlers“ since they handle the situation when the exception occurs.

**Step 3:** If no exception is raised, the statements inside the „else“ block is executed. Else block looks like as follows:

```
else:  
  
    statements
```

**Step 4:** Lastly, the programmer should perform **clean up actions** like closing the files and terminating any other processes which are running. The programmer should write this code in the finally block. Finally block looks like as follows:

```
finally:  
    statements
```

The specialty of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. So, the data in the files will not be corrupted and the user is at the safe-side.

Here, the complete exception handling syntax will be in the following format:

```
try:  
    statements  
except Exception1:  
    statements  
except Exception2:  
    statements  
else:  
    statements  
finally:  
    statements
```

The following points are followed in exception handling:

- A single try block can be followed by several except blocks.
  - Multiple except blocks can be used to handle multiple exceptions.
  - We cannot write except blocks without a try block.
  - We can write a try block without any except blocks.
  - Else block and finally blocks are not compulsory.
  - When there is no exception, else block is executed after try block.
- Finally block is always execute

**List of Standard Exceptions:-**

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+C.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
IOError	Raised when an input/output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.

**IndexError:**The IndexError is thrown when trying to access an item at an invalid index.

Example: IndexError

```
>>> L1=[1,2,3]
```

```
>>> L1[3]
```

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

```
L1[3]
```

IndexError: list index out of range

**KeyError:**“The KeyError is thrown when a key is not found.

Example: KeyError

```
>>> D1={'1':"aa", '2':"bb", '3':"cc"}
```

```
>>> D1['4']
```

Traceback (most recent call last):

File "<pyshell#15>", line 1, in <module>

```
D1['4']
```

KeyError: '4'

**ImportError:**The ImportError is thrown when a specified function can not be found.

Example: ImportError

```
>>> from math import cube
```

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

```
from math import cube
```

ImportError: cannot import name 'cube'

**TypeError:**The TypeError is thrown when an operation or function is applied to an object of an inappropriate type.

Example: TypeError

```
>>> '2'+2
```

Traceback (most recent call last):

File "<pyshell#23>", line 1, in <module>

```
'2'+2
```

TypeError: must be str, not int

**ValueError:**The ValueError is thrown when a function's argument is of an inappropriate type.

Example: ValueError

```
>>> int('xyz')
```

Traceback (most recent call last):

File "<pyshell#14>", line 1, in <module>

```
int('xyz')
```

ValueError: invalid literal for int() with base 10: 'xyz'

**ZeroDivisionError:** The ZeroDivisionError is thrown when the second operator in the division is zero.

Example: ZeroDivisionError

```
>>> x=100/0
```

Traceback (most recent call last):

File "<pyshell#8>", line 1, in <module>

```
x=100/0
```

ZeroDivisionError: division by zero

**The Except Block** The “except” block is useful to catch an exception that is raised in the try block. When there is an exception in the try block, then only the except block is executed. it is written in various formats.

1. To catch the exception which is raised in the try block, we can write except block with the Exceptionclass name as:

*i. except Exceptionclass:*

2. We can catch the exception as an object that contains some description about the exception.

*i. except Exceptionclass as obj:*

3. To catch multiple exceptions, we can write multiple catch blocks. The otherway is to use a single except block and write all the exceptions as a tuple inside parentheses as:

*a. except (Exceptionclass1, Exceptionclass2, ):*

4. To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

*except:*

**Example: try...except blocks**

```
try:
```

```
    a=5
```

```
    b='0'
```

```
    print(a/b)
```

```
except:
```

```
    print('Some error occurred.')
```

```
print("Out of try except blocks.")
```

Output

Some error occurred.

Out of try except blocks.

You can mention a specific type of exception in front of the except keyword. The

**Example: Catch Specific Error Type**

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Output

Unsupported operation

Out of try except blocks

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

**Example: Multiple except Blocks**

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Output

Division by zero not allowed

Out of try except blocks

**else and finally:**

In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occurred or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code.



**Example: try, except, else, finally blocks**

```

try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )

```

Output1	Output2:	Output3:
try block Enter a number: 10 Enter another number: 2 else block Division = 5.0 finally block Out of try, except, else and finally blocks	try block Enter a number: 10 Enter another number: 0 except ZeroDivisionError block Division by 0 not accepted finally block Out of try, except, else and finally blocks	try block Enter a number: 10 Enter another number: xyz finally block Traceback (most recent call last): File "C:\python36\codes\test.py", line 3, in <module> y=int(input('Enter another number: ')) ValueError: invalid literal for int() with base 10: 'xyz'

The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed..

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Typically **the finally clause** is the ideal place for **cleaning up the operations** in a process.

For example closing a file irrespective of the errors in read/write operations.

**Raise an Exception :**

Python also provides the **raise** keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution. The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

**Example: Raise an Exception**

try:

```
x=int(input('Enter a number upto 100: '))
```

```
if x > 100:
```

```
    raise ValueError(x)
```

except ValueError:

```
    print(x, "is out of allowed range")
```

else:

```
    print(x, "is within the allowed range")
```

Output

Enter a number upto 100: 200

200 is out of allowed range

Enter a number upto 100: 50

50 is within the allowed range

Here, the raised exception is a ValueError type. However, you can define your custom exception type to be raised.

**User-Defined Exceptions:**

- Like the built-in exceptions of python, the programmer can also create his own exceptions which are called „User-defined exceptions“ or „Custom exceptions“. We know Python offers many exceptions which will raise in different contexts.
- But, there may be some situations where none of the exceptions in Python are useful for the programmer. In that case, the programme has to create his/her own exception and raise it.
- Python has many [built-in exceptions](#) which forces your program to output an error when something in it goes wrong. However, sometimes you may need to create custom exceptions that serves your purpose.
- In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived from this class.

**Example: User-Defined Exception in Python**

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

```
class ValueTooSmallError(Exception):
    pass
class ValueTooLargeError(Exception):
    pass
number = 10
while True:
    try:
        i_num = int(input("Enter a number: "))
        if(i_num < number):
            raise ValueTooSmallError
        elif(i_num > number):
            raise ValueTooLargeError
        else:
            break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
print("Congratulations! You guessed it correctly.")
```

**Output:**

```
Enter a number: 12
This value is too large, try again!
Enter a number: 0
This value is too small, try again!
Enter a number: 8
This value is too small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

**Defining Clean-up Actions**

**Cleanup actions:** Before leaving the **try** statement, “**finally**” clause is always executed, whether any exception is raised or not. These are clauses which are intended to define clean-up actions that must be executed under all circumstances.

Whenever an exception occurs and is not being handled by the **except** clause, first **finally** will occur and then the error is raised as default

**Programs illustrating “Defining Clean Up Actions”**

**Code 1 :** Code works normally and clean-up action is taken at the end

```
# clean up actions
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        print("I'm finally clause, always raised !! ")

divide(3, 2)
```

**Output :**

```
Yeah ! Your answer is : 1
I'm finally clause, always raised !!
```

As you can see, the **finally** clause is executed in any event. In real world applications, the **finally** clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

**Predefined Clean-up Actions**

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

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

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications.

**This pre-defined clean up actions automatically closes the file after its suite finishes, even though an exception raised on the way. It can be used by the keyword “with”.**

The **with** statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

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

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines.

## **Graphical User Interfaces(GUI):**

### **Introduction**

Most modern computer software employs a graphical user interface or GUI

- ▶ A GUI displays text as well as small images (called icons) that represent objects such as directories, files of different types, command buttons, and drop-down menus
- ▶ In addition to entering text at keyboard, the user of a GUI can select an icon with pointing device, such as mouse, and move that icon around on the display

### **The Behavior of Terminal-Based Programs and GUI-Based Programs**

Two different versions of the bouncy program from a user's point of view:

- Terminal-based user interface
- Graphical user interface

Both programs perform the same function

However, their behavior, or look and feel, from a user's perspective are quite different

### **The Bouncy Program**

Program computes and displays the total distance traveled by a ball, given three inputs:

- initial height from which ball dropped
- bounciness index
- number of bounces

The total distance travelled for a single bounce is the sum of the distance down and the distance back up

- For inputs (10, 0.6, 1) total distance is  $10 + 10*0.6 = 16$
- For inputs (10, 0.6, 2) total distance is  $16 + 6 + 6*0.6 = 25.6$

### **Terminal-based user interface**

## **The Terminal-Based Version**

```
Welcome to the bouncy program!

1 Compute the total distance
2 Quit the program

Enter a number: 1

Enter the initial height: 10
Enter the bounciness index: .6
Enter the number of bounces: 2

The total distance is 25.6

1 Compute a distance
2 Quit the program

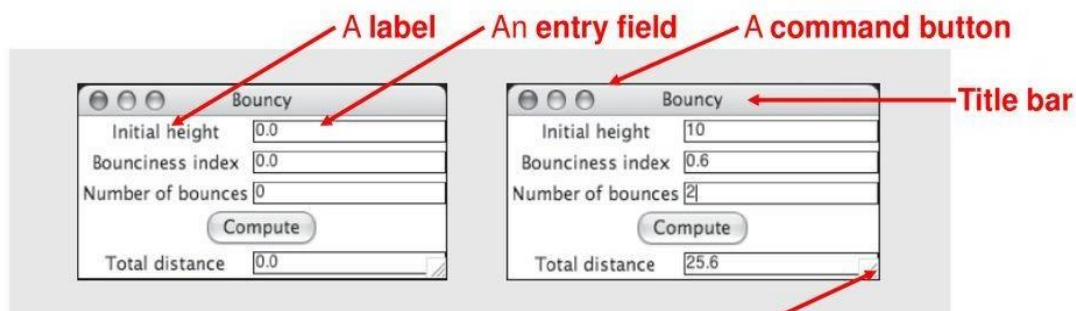
Enter a number: 2
```

**[FIGURE 9.1]** A session with the terminal-based `bouncy` program  
Fundamentals of Python: From First Programs Through Data Structures

## Graphical user interface

# The GUI-Based Version

- Uses a window that contains various components
  - Called **window objects** or **widgets**



**[FIGURE 9.2]** A GUI-based `bouncy` program

- Both programs perform exactly the same function – However, their behavior, or look and feel, from a user's perspective are quite different.
- Problems: – User is constrained to reply to a definite sequence of prompts for inputs
- Once an input is entered, there is no way to change it – To obtain results for a different set of input data, user must wait for command menu to be displayed again
- At that point, the same command and all of the other inputs must be re-entered – User can enter an unrecognized command

## The GUI-Based Version

- Uses a window that contains various components – Called window objects or widgets
- Solves problems of terminal-based version: Title bar, A label, An entry field, A command button.
- Event-Driven Programming : User-generated events (e.g., mouse clicks) trigger operations in program to respond by pulling in inputs, processing them, and displaying results.

**Coding phase:**

- Define a new class to represent the main window
- Instantiate the classes of window objects needed for this application (e.g., labels, command buttons).
- Position these components in the window.
- Instantiate the data model and provide for the display of any default data in the window objects .
- Register controller methods with each window object in which a relevant event might occur.
- Define these controller methods.
- Define a main that launches the GUI.

**Coding Simple GUI-Based Programs**

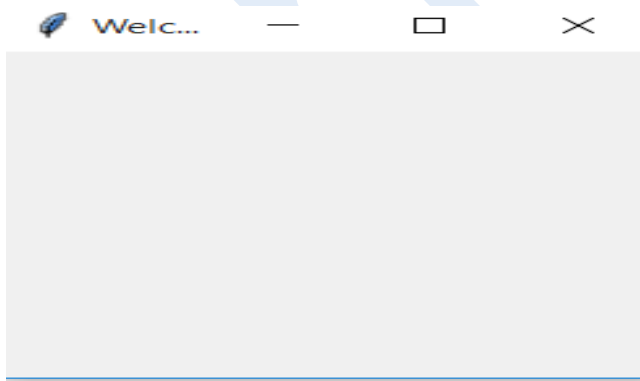
- There are many libraries and toolkits of GUI components available to the Python programmer –tkinter includes classes for windows and numerous types of window objects.

**Create GUI applications**

First, we will import Tkinter package and create a window and set its title:

```
from tkinter import *  
window = Tk()  
window.title("Welcome to tkinter")  
window.mainloop()
```

The last line which calls main loop function, this function calls the endless loop of the window, so the window will wait for any user interaction till we close it.



If you forget to call the mainloop function, nothing will appear to the user.

## Create a label

To add a label to our previous example, we will create a label using the label class like

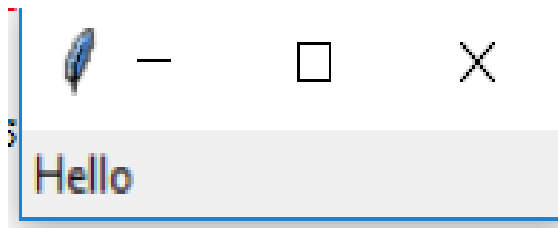
```
lbl = Label(window, text="Hello")
```

Then we will set its position on the form using the grid function and give it the location like

```
lbl.grid(column=0, row=0)
```

So the complete code will be like this:

```
from tkinter import *  
window = Tk()  
lbl = Label(window, text="Hello")  
lbl.grid(column=0, row=0)  
window.mainloop()
```



You can set the label font so you can make it bigger and maybe bold. You can also change the font style. To do so, you can pass the font parameter like this:

```
lbl = Label(window, text="Hello", font=("Arial Bold", 50))
```

## Setting window size

We can set the default window size using geometry function like this:

```
window.geometry('350x200')
```

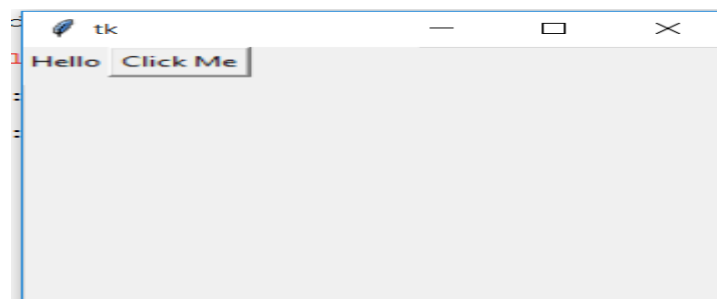
The above line sets the window width to 350 pixels and the height to 200 pixels.

## Adding a button

Let's start by adding the button to the window, the button is created and added to the window the same as the label:

```
btn = Button(window, text="Click Me")
```

```
btn.grid(column=1, row=0)
```





### Get input using Entry class (Tkinter textbox)

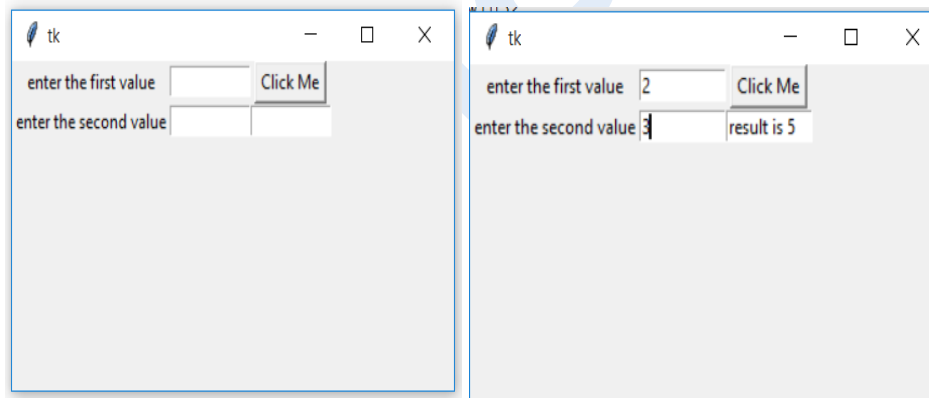
You can create a textbox using Tkinter Entry class like

```
this:txt = Entry(window,width=10)
```

Then you can add it to the window using grid function as usual.

```
from tkinter import *
window = Tk()
window.geometry('350x200')
lbl1 = Label(window, text="enter the first value")
lbl1.grid(column=0, row=0)
lbl2 = Label(window, text="enter the second value")
lbl2.grid(column=0, row=1)
txt1 = Entry(window,width=10)
txt1.grid(column=1, row=0)
txt2 = Entry(window,width=10)
txt2.grid(column=1, row=1)
txt3 = Entry(window,width=10)
txt3.grid(column=2, row=1)
def clicked():
    res=int(txt1.get())+int(txt2.get())
    txt3.insert(0,"result is {}".format(res))
    btn = Button(window, text="Click Me", command=clicked)
    btn.grid(column=2, row=0)
window.mainloop()
```

The above code creates a window with 2 labels, 3 text fields and reads two numbers from the user and finally displays the result in the third text field.



## Add a combobox

Combobox is a combination of Listbox and an entry field. It is one of the Tkinter widgets where it contains a down arrow to select from a list of options. It helps the users to select according to the list of options displayed. When the user clicks on the drop-down arrow on the entry field, a pop up of the scrolled Listbox is displayed down the entry field. The selected option will be displayed in the entry field only when an option from the Listbox is selected.

### Syntax:

```
combobox = ttk.Combobox(master, option=value, ...)
```

```
import tkinter as tk
from tkinter import ttk
```

```
# Creating tkinter window
window = tk.Tk()
window.title('Combobox')
window.geometry('500x250')
```

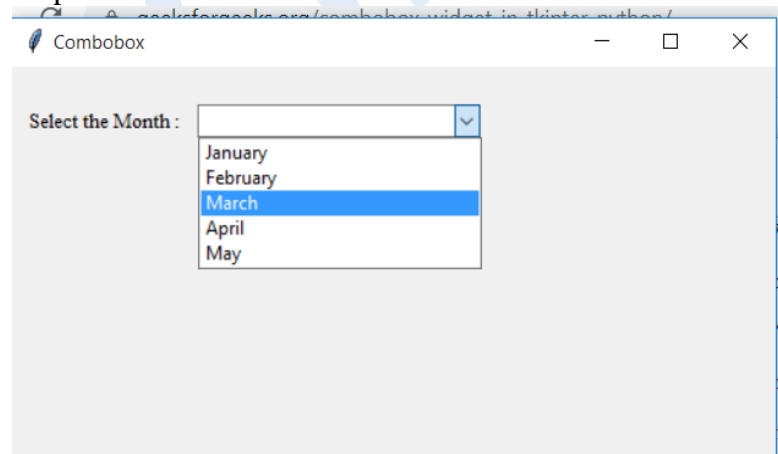
```
# label
ttk.Label(window, text = "Select the Month :",
          font = ("Times New Roman", 10)).grid(column = 0,
          row = 5, padx = 10, pady = 25)
```

```
# Combobox creation
n = tk.StringVar()
monthchoosen = ttk.Combobox(window, width = 27, textvariable = n)
```

```
# Adding combobox drop down list
monthchoosen['values'] = (' January', ' February', ' March', ' April', ' May')
```

```
monthchoosen.grid(column = 1, row = 5)
monthchoosen.current()
window.mainloop()
```

output:

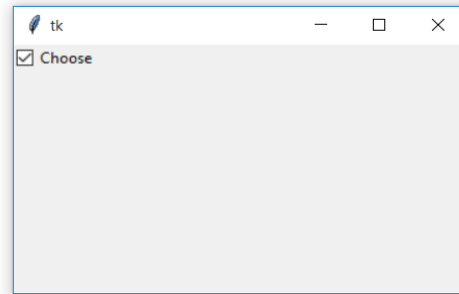


**Add a Checkbutton (Tkinter checkbox)**

To create a checkbutton, you can use Checkbutton class like this:

```
chk = Checkbutton(window, text='Choose')
```

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
chk_state = BooleanVar()
chk_state.set(True) #set check state
chk = Checkbutton(window, text='Choose',
var=chk_state) chk.grid(column=0, row=0)
window.mainloop()
```



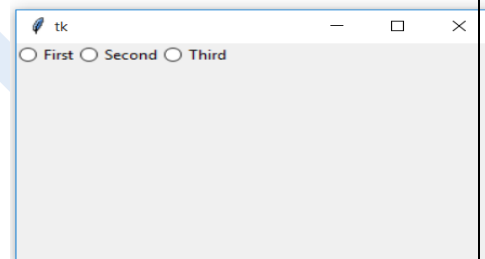
Here we create a variable of type BooleanVar which is not a standard Python variable, it's a Tkinter variable, and then we pass it to the Checkbutton class to set the check state as the highlighted line in the above example. You can set the Boolean value to false to make it unchecked.

**Add radio buttons widgets**

To add radio buttons, simply you can use RadioButton class like this:

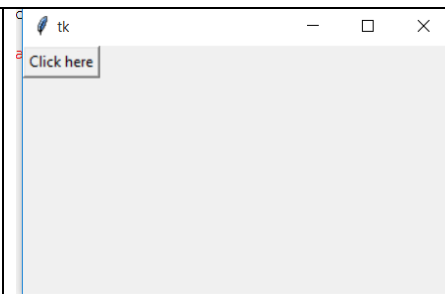
```
rad1 = Radiobutton(window, text='First', value=1)
```

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
rad1 = Radiobutton(window, text='First', value=1)
rad2 = Radiobutton(window, text='Second', value=2)
rad3 = Radiobutton(window, text='Third', value=3)
rad1.grid(column=0, row=0)
rad2.grid(column=1, row=0) rad3.grid(column=2, row=0)
window.mainloop()
```

**Create a MessageBox**

To show a message box using Tkinter, you can use messagebox library like this:

```
from tkinter import *
window = Tk()
window.geometry('350x200')
def
clicked():
    messagebox.showinfo('Messagetitle', 'Message
content')
    btn=Button(window, text='Clickhere',
command=clicked)
    btn.grid(column=0, row=0)
window.mainloop()
```



You can show a warning message or error message the same way. The only thing that needs to be changed is the message function.

### Show warning and error messages:

You can show a warning message or error message the same way. The only thing that needs to be changed is the message function

**messagebox.showwarning('Message title', 'Message content') #shows warning message**

**messagebox.showerror('Message title', 'Message content')**

You can choose the appropriate message style according to your needs. Just replace the showinfo function line from the previous line and run it.

### Show askquestion dialogs

To show a yes no message box to the user, you can use one of the following messagebox functions:

```
from tkinter import messagebox
```

```
res = messagebox.askquestion('Message title','Message content')
```

```
res = messagebox.askyesno('Message title','Message content')
```

```
res = messagebox.askyesnocancel('Message title','Message content')
```

- If you click OK or yes or retry, it will return True value, but if you choose no or cancel, it will return False.
- The only function that returns one of three values is askyesnocancel function, it returns True or False or None.

#### Example:

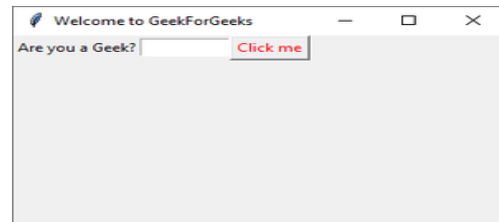
```
from tkinter import *
root = Tk()
# root window title and dimension
root.title("Welcome to GeekForGeeks")
# Set geometry(widthxheight)
root.geometry('350x200')
# adding a label to the root window
lbl = Label(root, text = "Are you a Geek?")
lbl.grid()

# adding Entry Field
txt = Entry(root, width=10)
txt.grid(column =1, row =0)

# button is clicked
def clicked():
    res = "You wrote" + txt.get()
    lbl.configure(text = res)
btn = Button(root, text = "Click me" ,
             fg = "red", command=clicked)
# Set Button Grid
btn.grid(column=2, row=0)

# Execute Tkinter
mainloop()
```

#### Output:



*Entry Widget at column 2 row 0*

**Other Useful GUI Resources:**

- GUI is nothing but a desktop app that provides you with an interface that helps you to interact with the computers and enriches your experience of giving a command (command-line input) to your code. They are used to perform different tasks in desktops, laptops, and other electronic devices, etc.
- Some of the applications where the power of GUI is utilized are:
- Creating a Calculator which would have a user-interface and functionalities that persists in a calculator.
- Text-Editors, IDE's for coding are on a GUI app.
- Sudoku, Chess, Solitaire, etc..., are games that you can play are GUI apps.
- Chrome, Firefox, Microsoft Edge, etc. used to surf the internet is a GUI app.
- Another interesting use-case would be - A GUI for controlling a Drone from your laptop.

**Let's see some of the frameworks that Python provides to develop a GUI:**

**PyQT** is one of the favored cross-platform Python bindings implementing the Qt library for the Qt application development framework. Nokia primarily owns Qt. Currently, PyQT is available for almost all operating systems like Unix/Linux, Windows, Mac OS X.

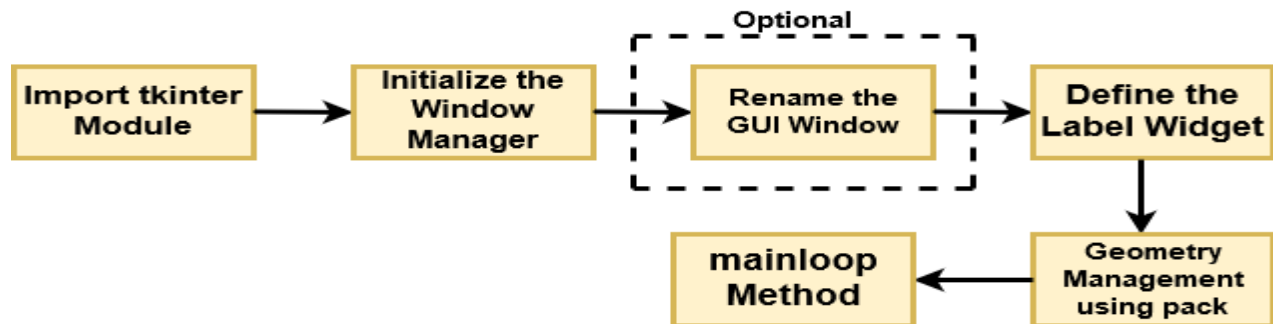
**Kivy** is for the creation of new user interfaces and is an OpenGL ES 2 accelerated framework. Much like **PyQt**, Kivy also supports almost all platforms like Windows, MacOSX, Linux, Android, iOS. It is an open-source framework and comes with over 20 pre-loaded widgets in its toolkit.

**Jython** is a Python port for Java, which gives Python scripts seamless access to Java class libraries on the local machine.

**WxPython**, initially known as WxWindows (now as a WxWidgets library), is an open-source abstract-level wrapper for cross-platform GUI library. It is implemented as a Python expansion module. With WxPython, you, as a developer, can create native applications for Windows, Mac OS, and Unix.

**PyGUI** is a graphical application cross-platform framework for Unix, Macintosh, and Windows. Compared to some other GUI frameworks, PyGUI is by far the simplest and lightweight of them all, as the API is purely in sync with Python. PyGUI inserts very less code between the GUI platform and Python application;

**Tkinter** commonly comes bundled with Python, using Tk and is Python's standard GUI framework. It is famous for its simplicity and graphical user interface. It is open-source and available under the Python License.



Flow Diagram for Rendering a Basic GUI

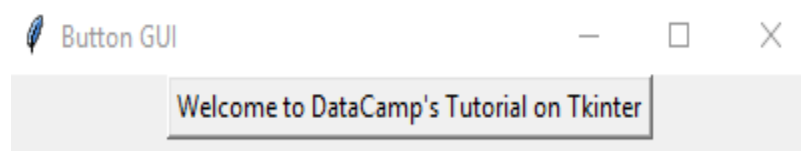
Let's break down the above flow diagram and understand what each component is handling!

- First, you import the key component, i.e., the Tkinter module.
- As a next step, you initialize the window manager with the `tkinter.Tk()` method and assign it to a variable. This method creates a blank window with close, maximize, and minimize buttons on the top as a usual GUI should have. Then as an optional step, you will Rename the title of the window as you like with `window.title(title_of_the_window)`.
- Next, you make use of a widget called Label, which is used to insert some text into the window. Then, you make use of Tkinter's geometry management attribute called `pack()` to display the widget in size it requires.
- Finally, as the last step, you use the `mainloop()` method to display the window until you manually close it. It runs an infinite loop in the backend.

you could have a widget Button, and the GUI will display a button instead of some text (Label).

```

import tkinter
window = tkinter.Tk()
window.title("Button GUI")
button_widget = tkinter.Button(window, text="Welcome to DataCamp's Tutorial on Tkinter")
button_widget.pack()
tkinter.mainloop()
  
```



## Introduction to Programming Concepts with Scratch.

**Scratch** is a visual programming language that allows students to create their own interactive stories, games and animations. As students design Scratch projects, they learn to think creatively, reason systematically, and work collaboratively. Scratch was created by the Lifelong Kindergarten group at MIT Media lab and is available for **free** download at <http://scratch.mit.edu>. Once Scratch is downloaded to a computer, you do not need Internet access to create a project.

## Scratch programming review

### programming concepts from Scratch

- simple actions/behaviors (e.g., move, turn, say, play-sound, next-costume)
- control
  - repetition (e.g., forever, repeat)
  - conditional execution (e.g., if, if-else, repeat-until)
  - logic (e.g., =, >, <, and, or, not)
- arithmetic (e.g., +, -, \*, /)
- sensing (e.g., touching?, mouse down?, key pressed?)
- variables (e.g., set, change-by)
- communication/coordination (e.g., broadcast, when-I-receive)

### Variable assignment

- In Scratch, a variable needs to be created before it can be assigned a value, whereas in Python a variable is created upon assignment with a value.
- In Python, it is necessary to surround strings (any text) with either single (') or double (") quotes.



```
foo = 10
bar = "some text"
```

### Increment a variable

- In Scratch, a variable's value can be increased or decreased.
- In Python, a variable's value can be increased or decreased by reassigning it to itself with the addition or subtraction of a number.



```
foo = foo + 1
###or
foo += 1
```

### Simple output

- In Scratch, you make a sprite talk to provide output to the user of the program.
- Python uses print statements to output to the **shell**.
- Again in Python, you need to use single or double quotes if you are printing strings.



```
print('Hello World!')
print(foo)
```

### Conditional loops

- Scratch's conditional loop repeats **until** a certain statement is True.
- Python's conditional loop repeats **as long as** a certain statement is True.
- There needs to be a colon (:) at the end of the statement in Python.
- Notice that the code that is **inside** the loop is **indented**. Indentation is normally four spaces or a single tab. This can be compared to the way the Scratch conditional loop block **brackets** the code within it.



```
while not foo > 10:
    print(foo)
```



- The example above isn't the simplest way of doing this in Python though. Using the while loop, it is easier to check that the variable is **less than or equal to** 10.

```
while foo <= 10:
    print(foo)
```

### Infinite loops

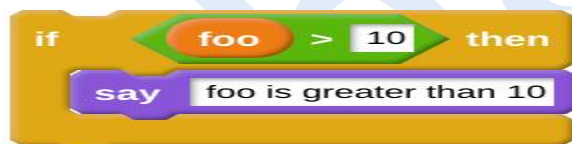
- Scratch has a specific type of loop that is infinite.
- In Python, a conditional loop is used that always evaluates to True.



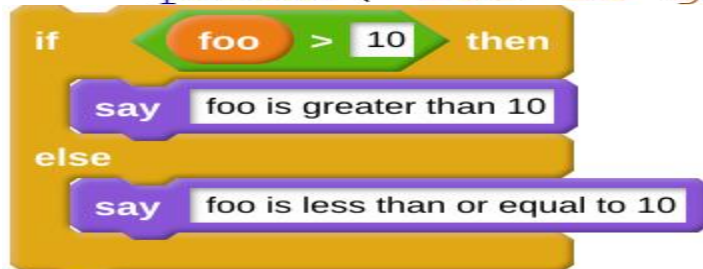
```
while True:
    foo += 1
```

### Conditional selection

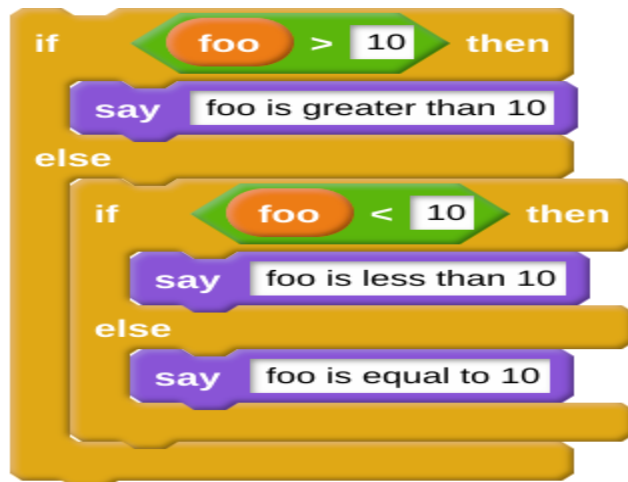
- Scratch has two selection blocks that can be used. If multiple conditions are required, they need to be nested within each other.
- Python has three selection statements: if, elif, and else. Again colons (:) and indentation are needed.



```
if foo > 10:
    print("foo is greater than 10")
```



```
if foo > 10:
    print("foo is greater than 10")
else:
    print("foo is less than or equal to 10")
```



```
if foo > 10:
    print("foo is greater than 10")
elif foo < 10:
    print("foo is less than 10")
else:
    print("foo is equal to 10")
```

### Testing for equality

- In Scratch, you can use the equal sign (=) to test if one value is the same as another value.
- In Python, a single equal sign is reserved for variable assignment, so a double equal sign (==) is used to test for equality.



```
foo == 1
```

### Lists

- Scratch lists are made in much the same way that variables are made.
- In Python, you use square brackets ([]) when creating a list, with commas between each pair of items.



```
hand = ['ace', 'king', 'queen', 'jack', 'ten']
```

- You can add to a list in both Scratch and Python.



```
hand.append('nine')
```

- And you can also remove items from lists in both languages. In Scratch the first item in a list is at position 1. In Python, however, the first item in a list is at position 0. That's because in Python, you always start counting from 0.



```
hand.pop(0)
```

### Concatenation

- Join strings together in Scratch using the join block.
- In Python, you can use the addition operator (+) to join strings.



```
foo = "hello" + "world!"
```

### Indexing

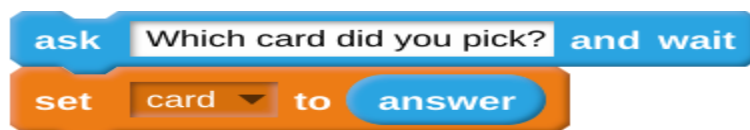
- In both languages, you can find an item in a list or string using the item index.



```
card = hand[0]  
letter = card[0]
```

### Input

- You can collect user input in Scratch by using the ask block.
- In Python you use the input() function.



```
card = input('Which card did you pick?')
```

## Downloading Scratch

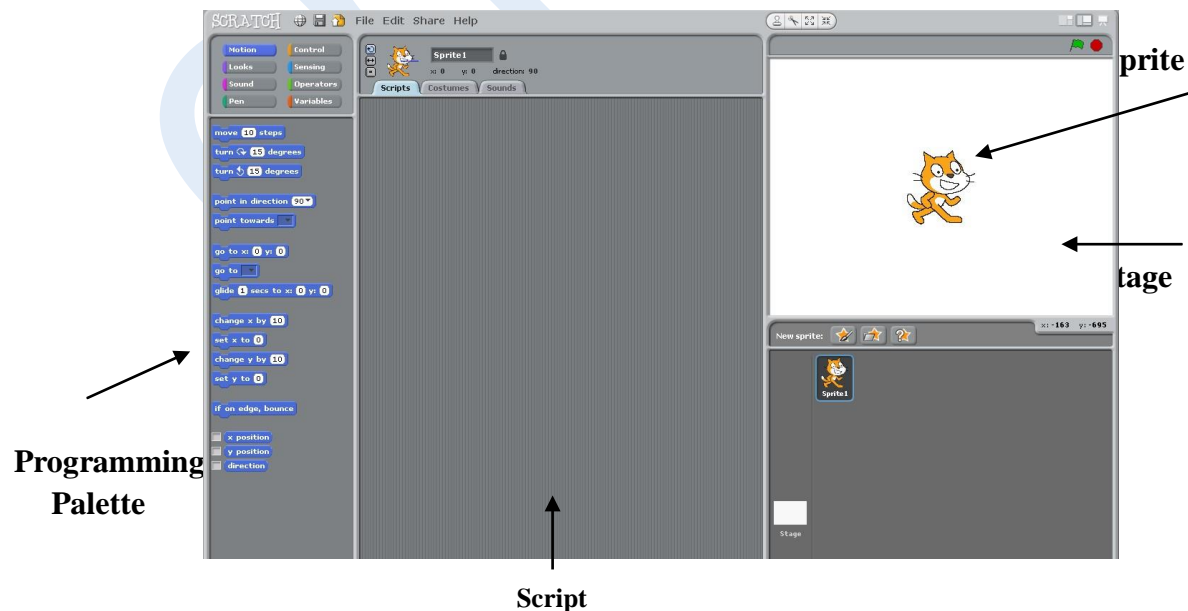


Go to the Scratch website at <http://scratch.mit.edu>. From the home screen, click on



## Elements of Scratch

There are four main elements of Scratch: the stage, the sprites, the script and the programming palette. These elements can be compared to a play.

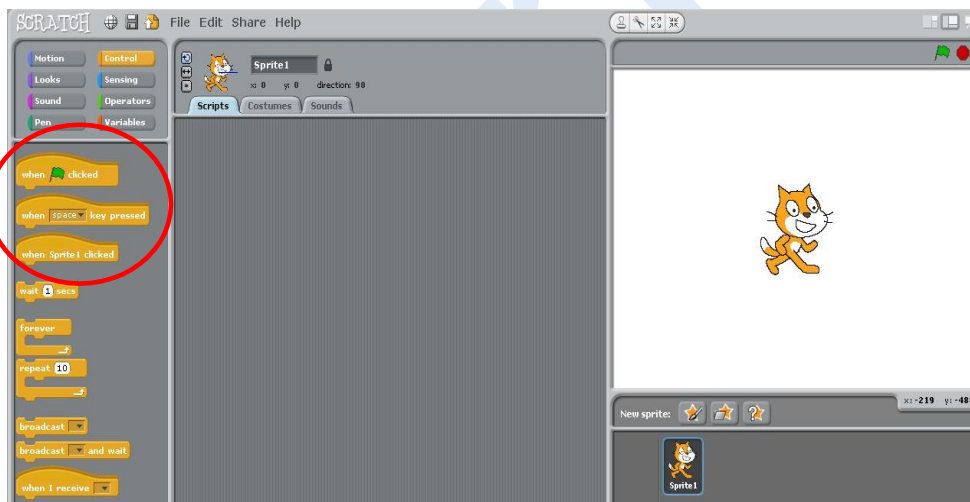
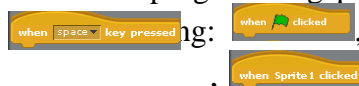


- 1) *Stage* – similar to the stage in a play. This is where everything will take place. The stage can be different backgrounds , just like in a play.
- 2) *Sprites* – are the actors or main characters of the project. Sprites are programmed to do something in Scratch.
- 3) *Script* – tells the actors what to say or do. Each sprite is programmed with a script.
- 4) *Programming palette* – elements used to program the sprite to do or say something. Sprites must be programmed to carry out every function you want them to perform.

### Programming a Sprite – Control, Motion,

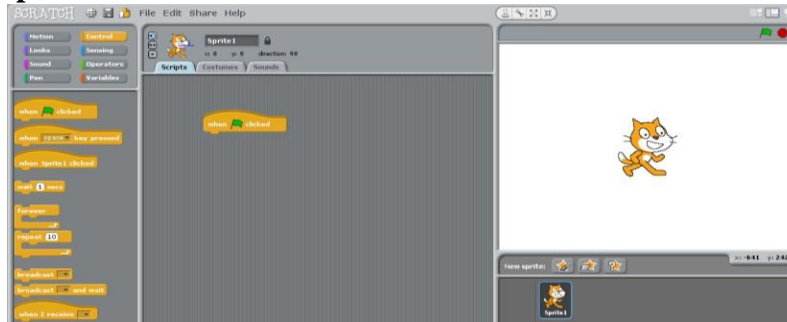
#### Looks and Sound Control blocks

Programming a sprite ALWAYS begins with a control block (orange/yellow category from the programming palette). There are three blocks that can be used to begin




- 1) *When the green flag is clicked* – The project will begin when the green flag in the upper right hand corner is clicked.
- 2) *When space key is pressed* – The project will begin when the space bar is pressed. The black drop down arrow indicates that you can choose a key different from the space bar; and that key will begin the project.
- 3) *When sprite 1 is clicked* – The project will begin when the sprite is click. Note: Click the sprite on the stage, NOT the small thumbnail sprite shown underneath the stage.

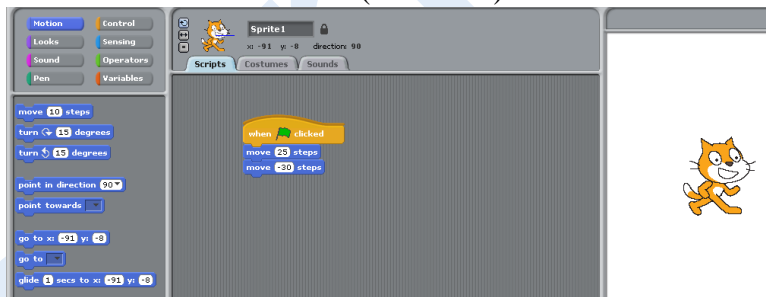
Drag the control block to the gray scripts area. **The next block will connect to this one like a puzzle piece.**



### Motion blocks

Motion blocks fall under the blue category. There are several blocks that will allow the sprite to move. This category teaches students to understand positives and negatives and other mathematical concepts, like degrees.

- 1)  – the sprite will move **X** steps. You can change the value whenever there is a fillable white area in a programming block. For example, 10 steps can be changed to 25 steps. Positive values move forward (to the right) and negative values move backward (to the left).

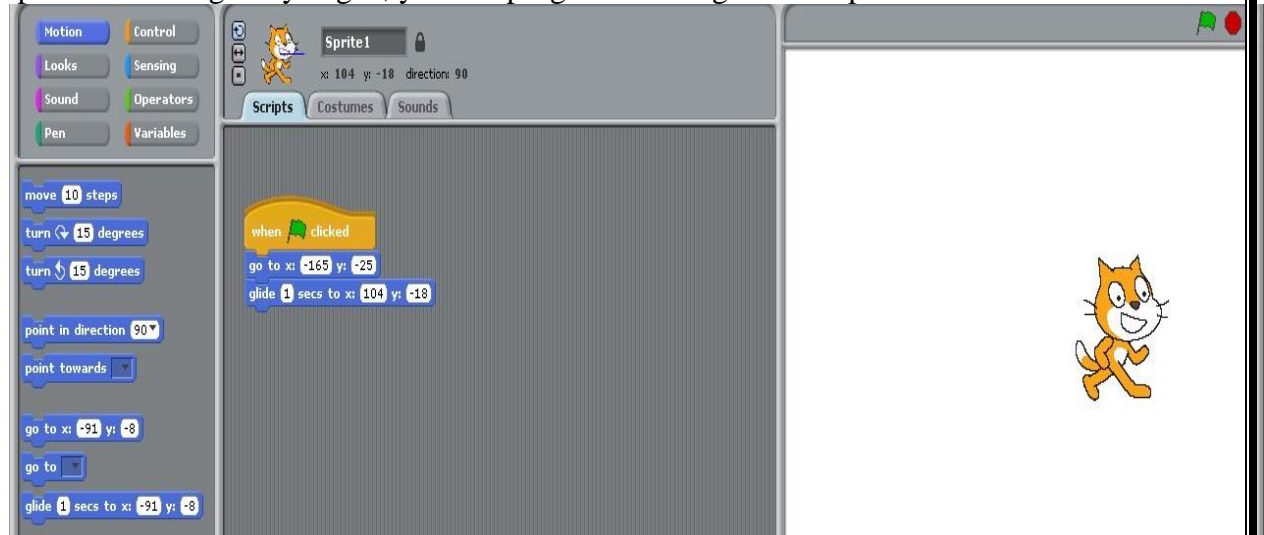


- 2) Glide\_\_sec to x: \_\_ y: \_\_ - The sprite will glide to a specified position in **X** seconds. Remember: the more seconds you use, the slower/longer it will take the sprite to glide across the stage. Find a position on the stage that you would like the sprite to glide to and then move it there. Above the scripts area, you will find X and Y positions. Use these values to fill in the X and Y area on the glide block. Be sure to use a (-) sign when necessary.



- 3) Go to x\_\_y: \_\_ - this block is used to place the sprite at a specific position

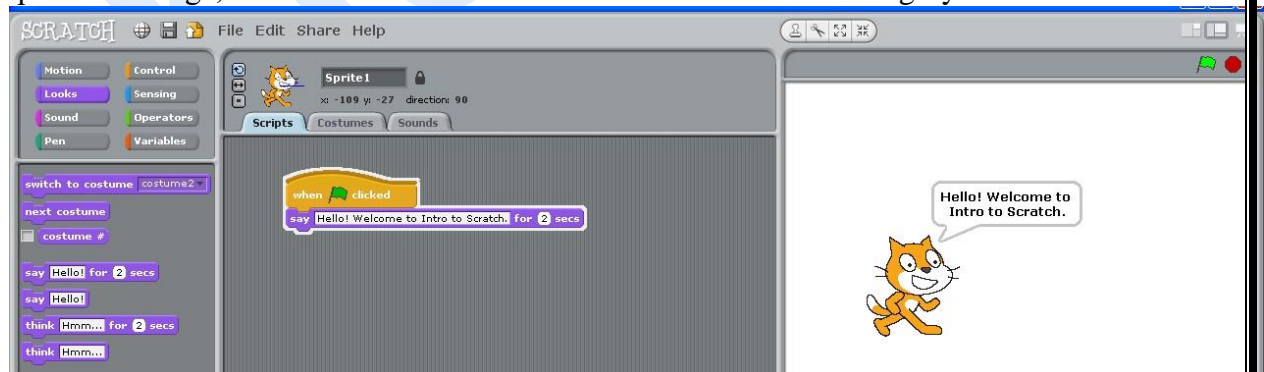
when the project begins, so you do not have to manually pick the sprite up and put it in that position every time you restart the project. For example, if the sprite glides when the green flag is clicked, it will move away from the beginning location. Instead of moving the sprite back to the spot where it originally began, you can program it to begin at that position.



### Looks blocks

Looks blocks fall under the purple category of the programming palette. There are several blocks to control what you **SEE** the sprite say or how the sprite looks.

- 1) Say "hello" for 2 seconds – allows you to program the sprite to give a word bubble that "says" what you have typed. Because the white space is fillable, you can delete hello and type another message. Note: You will **SEE** the sprite's message, not hear it because this block is under the looks category.



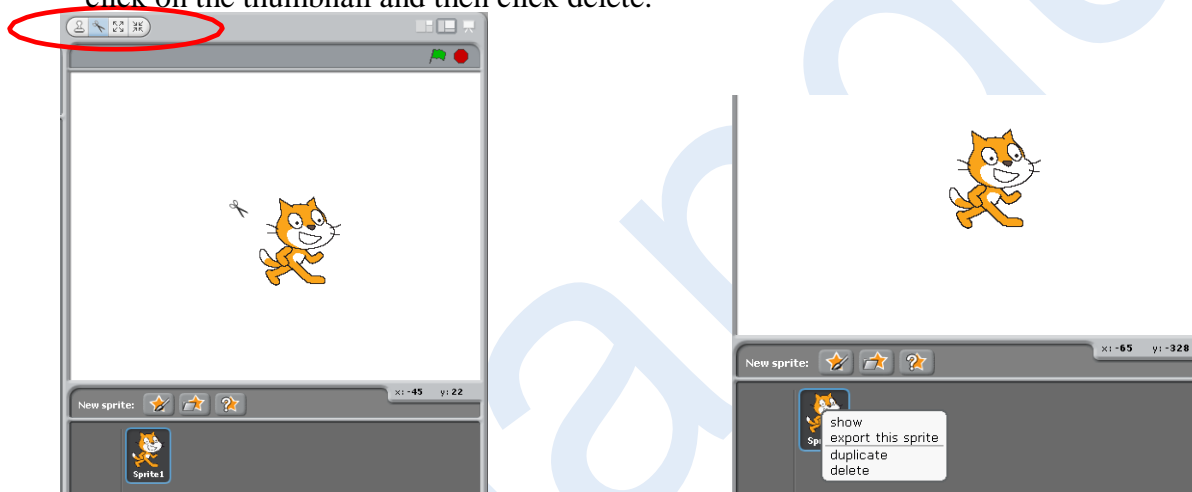
- 2) Switch to costume – If you click on the "Costumes" tab beside the word "Scripts," you will notice that some sprites have more than one costume. You can program the sprite to switch costumes. If the sprite does not have another costume, you can always create your own by clicking copy and then edit (see page 6 for more about painting a sprite).



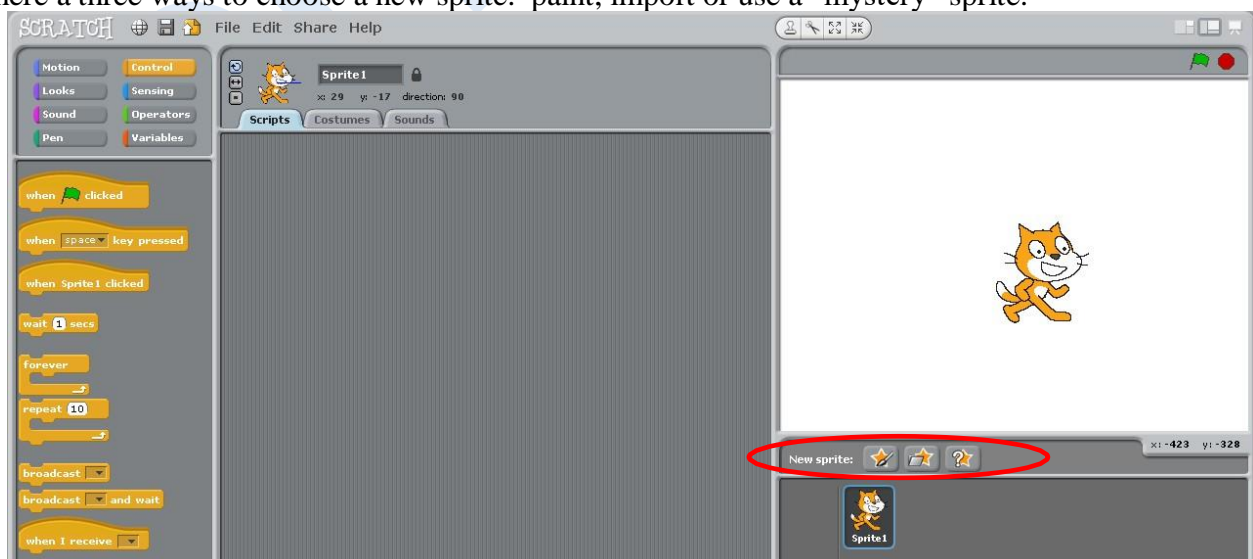


### Choosing a new sprite


If you do not wish to use the Scratch Cat as your sprite, there are two ways to delete it: 1) click on the scissors above the stage and then click on the sprite or 2) **RIGHT** click on the thumbnail and then click delete.

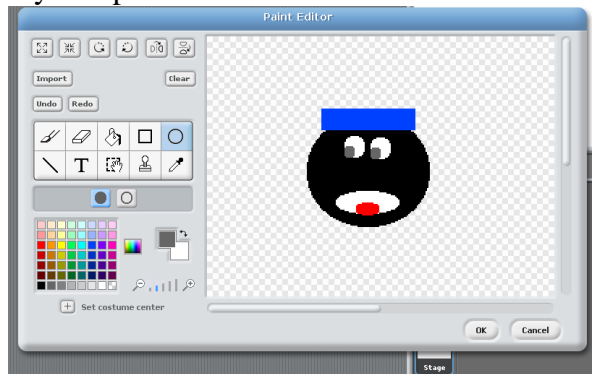



There are three ways to choose a new sprite: paint, import or use a “mystery” sprite.

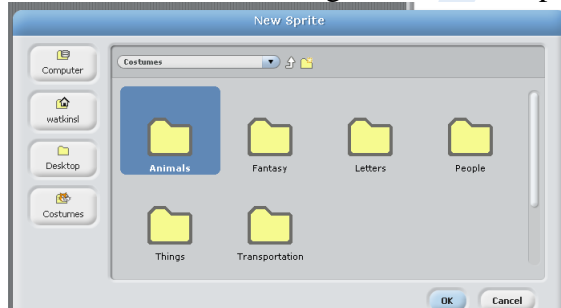





- 1) Paint a new sprite  – allows you to use the paint editor to create your very own sprite. You can use a paint brush, paint bucket, lines, circles and squares to create your sprite.



- 2) New sprite  – allows you to choose from sprites that are available in Scratch. Double click on one of the categories to find a sprite you wish to use.



- 3) Mystery sprite  – Scratch will choose a random sprite for you.

\*\*\*\*\* END\*\*\*\*\*