

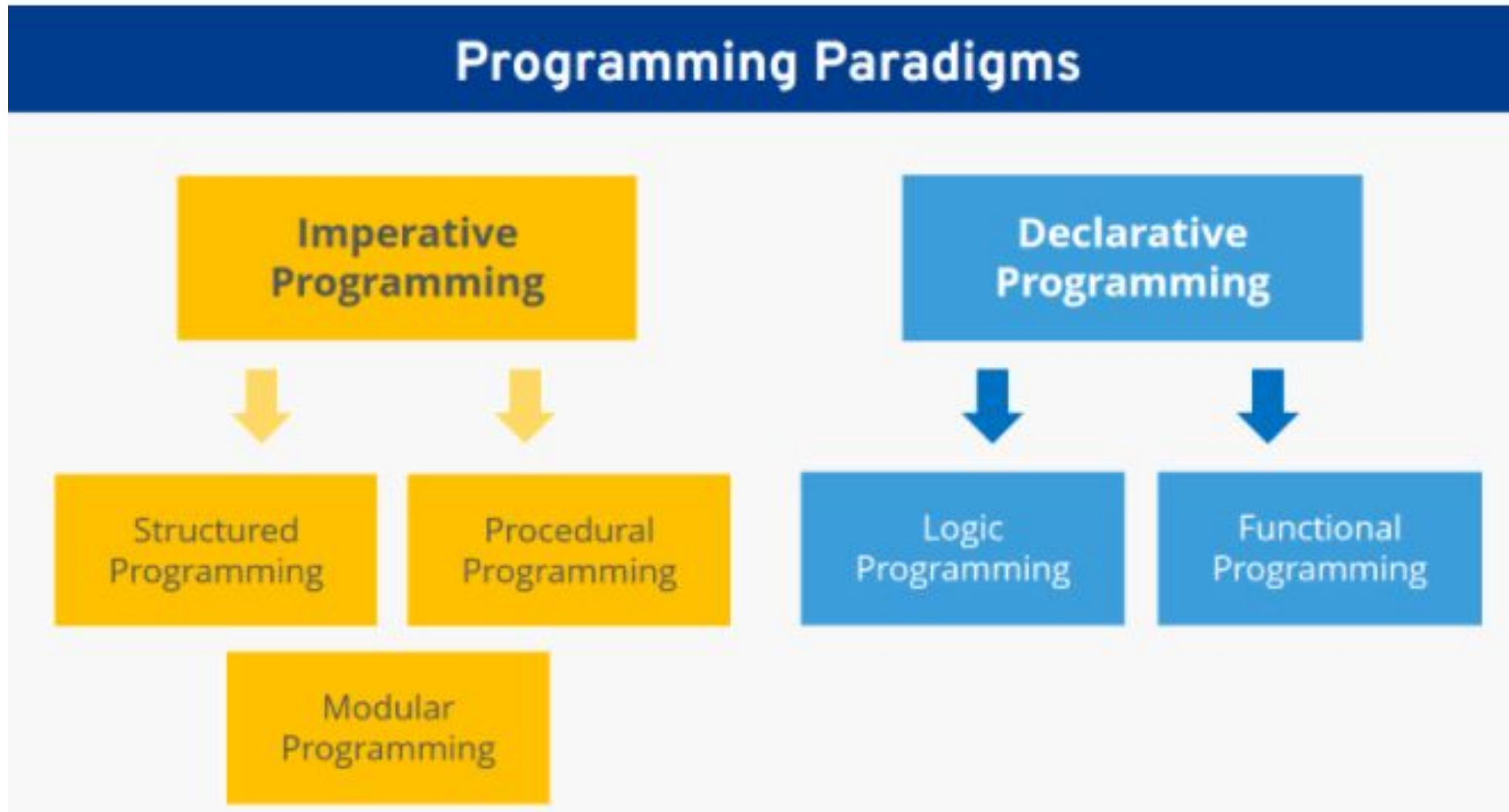
18CSC207J – Advanced Programming Practice

Declarative Programming Paradigm

Declarative Programming Paradigm

- Broadly speaking, computer programming languages have been divided into two categories---imperative languages and declarative languages.
- Imperative programming lays more stress on "how" a solution procedure is specified.
- Imperative: Programming with an explicit sequence of commands that update state.
- Declarative Programming stress on “what to do” rather than “how to do”.
- Declarative: Programming by specifying the result you want, not how to get it.
- Declarative programming languages is that they always describe the desired end result rather than outlining all the intermediate work steps.
- In declarative programming, the solution path to reach the goal is determined automatically.

Declarative Programming Paradigm



Declarative vs Imperative

Imagine you walk into your favorite coffee place and you would like to order some coffee.

The imperative approach will be:

- Enter the coffee shop
- Queue in the line and wait for the barista asking you for your order
- Order
- Yes, for takeaway, please
- Pay
- Present your loyalty card to collect points
- Take your order and walk away
- Imagine you walk into your favorite coffee place and you would like to order some coffee.

The declarative approach:

- A large latte for takeaway, please

Declarative vs Imperative

procedural (imperative)

how



1. Please, open the door.
2. Go outside.
3. Take the bucket I forgot there.
4. Bring it back to me

declarative (nonprocedural)

WHAT



1. Fetch the bucket, please.



OPTIMIZER



Declarative vs Imperative

Sum of the elements of list

`l1 = [12,76,34,96,45,62,71]`

#imperative approach

`Sum = 0`

`for i in l1:`

`sum += i`

#declarative approach

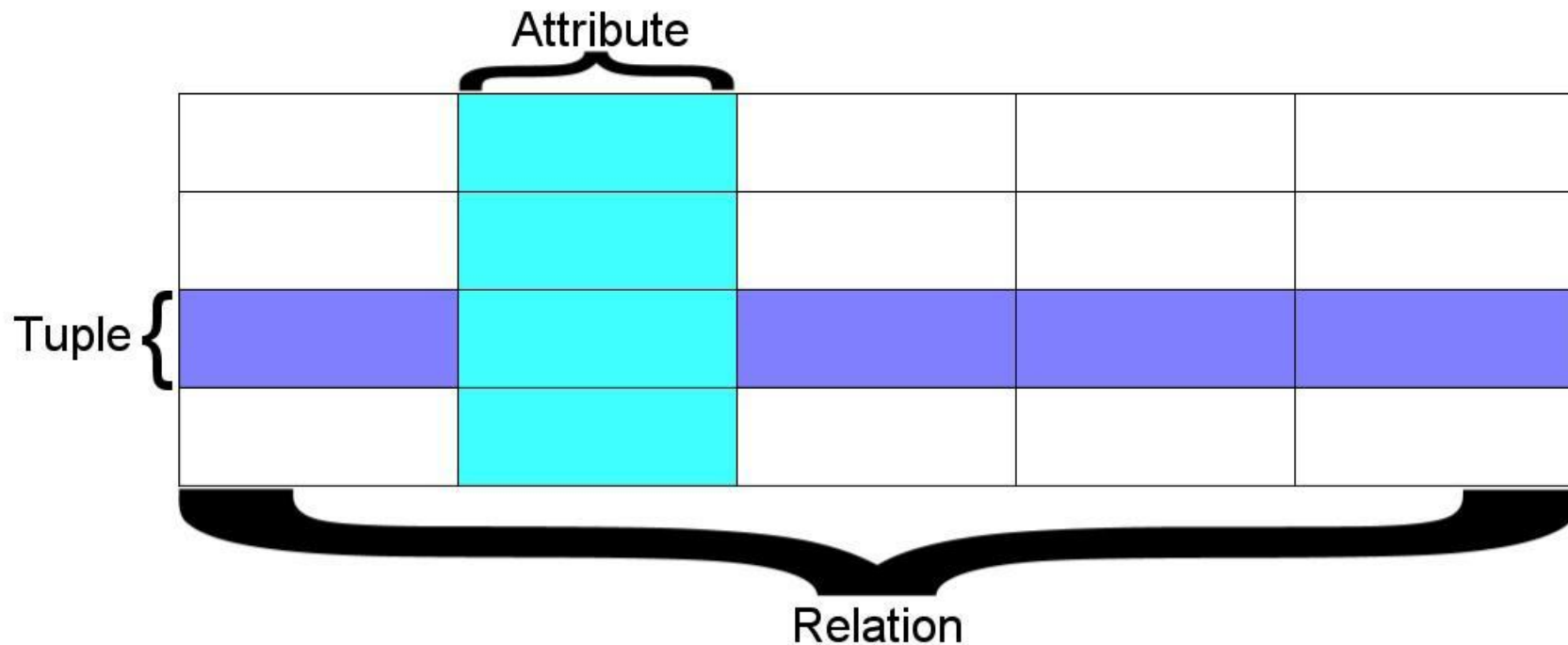
`Sum(l1)`

Declarative Programming Paradigm - Language

- SQLite3 - RDMS
- Relational databases model data by storing rows and columns in tables. The power of the relational database lies in its ability to efficiently retrieve data from those tables and in particular where there are multiple tables and the relationships between those tables involved in the query.
- Database - contains many tables
- Relation (or table) - contains tuples and attributes
- Tuple (or row) - a set of fields that generally represents an “object” like a person or a music track
- Attribute (also column or field) - one of possibly many elements of data corresponding to the object represented by the row

Declarative Programming Paradigm - Language

- A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints.



SQL

Structured Query Language is the language we use to issue commands to the database

- Create a table
- Retrieve some data
- Insert data
- Delete data

SQL – CRUD Operations

```
CREATE TABLE [IF NOT EXISTS] [schema_name].table_name (  
    column_1 data_type PRIMARY KEY,  
    column_2 data_type NOT NULL,  
    column_3 data_type DEFAULT 0,  
    table_constraints  
) [WITHOUT ROWID];
```

```
INSERT INTO table (column1,column2 ,..)  
VALUES( value1, value2 ,...);
```

SQL – CRUD Operations

```
SELECT * FROM tracks;
```

```
UPDATE employees  
SET lastname = 'Smith'  
WHERE employeeid = 3;
```

```
DELETE FROM table  
WHERE search_condition;
```

SQL – CURD Operations

```
import sqlite3
#create database named Student
conn = sqlite3.connect("Student.db")

#create table named Scores with necessary fields
conn.execute("create table Scores (regno int not null primary key,
sub1 int not null,
sub2 int not null, sub3 int not null)")

#insert some records into scores table
conn.execute("insert into Scores (regno,sub1,sub2,sub3) values(101,78,98,79) ")
conn.execute("insert into Scores (regno,sub1,sub2,sub3) values(112,87,81,91) ")

#update the score of subject3 to 89 for student with regno 101
conn.execute("update Scores set sub3 = 89 where regno=101")

#select the records from scores
rs= conn.execute("Select * from Scores")

#print the results of the result set
for row in rs:
    print(row[0]," ",row[1]," ",row[2]," ",row[3])

#delete the student record with regno 141
conn.execute("delete from Scores where regno=141")
```

IMPERATIVE PROGRAMMING PARADIGM

Mrs. S. Niveditha

Assistant Professor (Sr. G)

Department of Computer Science and
Engineering

SRMIST, Vadapalani, Chennai

Topics

- Program state, instructions to change the program state
- Combining Algorithms and Data Structures
- Imperative Vs Declarative Programming
- Other Languages: PHP, Ruby, Perl, Swift
- Demo: Imperative Programming in Python

INTRODUCTION

- In a computer program, a **variable** stores the data. The contents of these locations at any given point in the program's execution are called the **program's state**. Imperative programming is characterized by **programming with state** and commands which **modify the state**.
- The first imperative programming languages were **machine languages**.

Machine Language

- Each instruction performs as a very specific task, such as a load, a jump, or an ALU operation on a CPU register or memory.
- unit of data in a example:
- For *rt*, and *rd* indicate register operands shift
- *shamt* gives a amount
- – *rs*, the *address* or *immediate* fields contain an
- the *operand* directly

So, adding the registers 1 and 2 and placing the result in register 6 is encoded:

[op | rs | rt | rd |shamt| funct]

0 1 2 6 0 32 decimal

000000 00001 00010 00110 00000 100000 binary

Assembly Code

- 1011000001100001

- **Equivalent Assembly code**

- B0 61

- *B0 - 'Move a copy of the following value into AL'*
(AL is a register)
- *61 is a hexadecimal representation of the value 01100001 – 97*

- **Intel assembly language**

- MOV AL, 61h; Load AL with 97 decimal (61 hex)

Other Languages

- **FORTRAN** (FORmula TRANslation) was the first high level language to gain wide acceptance. It was designed for **scientific applications** and featured an algebraic notation, types, subprograms, and formatted input/output.
- **COBOL** (COmmon Business Oriented Language) was designed at the initiative of the U. S. Department of Defence in 1959 and implemented in 1960 to meet the need for **business data processing applications.**
- **ALGOL 60** (ALGorithmic Oriented 1960 Language) was designed in by an international committee for use in **scientific problem solving**

Evolutionary developments

Algol to PL / I

- Block structure
- Control statements
- Recursion

PL / I to FORTRA

- Subprograms
- Formatted IO

COBOL to PL/I

- File manipulation
- Record

LISP to PL/I

- Dynamic storage allocation
- Linked structures

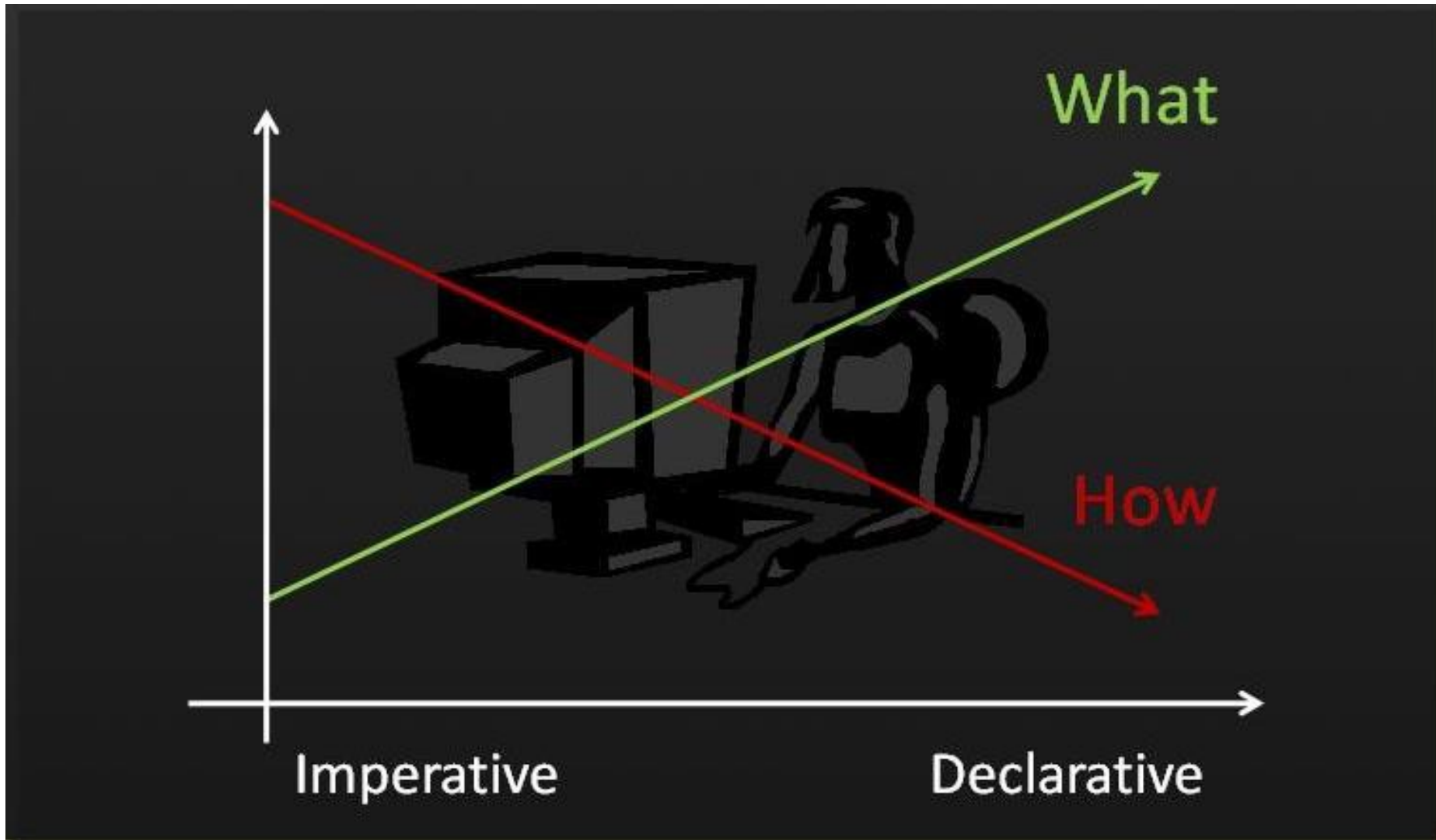
OVERVIEW

- In imperative programming, a name maybe later assigned to a value and reassigned to another value.
- The collection of names and the associated in values and the location of control the program constitute the state.
- The state is a logical model of storage which is an association between memory locations and values.
- A program in execution generates a sequence of states.
- The transition from one state to the next is determined by assignment operations and sequencing commands

Highlights on

- Assignment,
- goto commands
- structured programming
- Command
- Statement
- Procedure
- Control-flow
- Imperative language
- Assertions
- Axiomatic semantics
- State
- Variables
- Instructions
- Control structures

Declarative Vs Imperative



Declarative Vs Imperative

Declarative

```
# Declarative
```

```
small_nums = [x for x in range(20) if x < 5]
```

Imperative

```
# Imperative
```

```
small = []
```

```
for i in range(20):
```

```
    if i < 5:
```

```
        small.append(i)
```

0

1

2

3

4

DEMO

An algorithm to add two numbers entered by user

Step 1: Start

Step 2: Declare variables num1, num2 and sum. Step

3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to

sum. $\text{sum} \leftarrow \text{num1} + \text{num2}$

Step 5: Display sum

Step 6: Stop

Addition two numbers entered by user

```
num1 = 0
num2 = 0
sum = 0
num1 = input("Enter the First number ")
num2 = input("Enter the Second number ")
sum = int(num1) + int(num2)
print("\nSum:", sum)
```

Enter the First number 10

Enter the Second number 20

Sum: 30

An Algorithm to Get n number, print the same and find Sum of n numbers

Step 1: Start

Step 2: Declare variable $\text{sum} = 0$.

Step 3: Get the value of limit “n”.

Step 4: If limit is reached, goto Step 7 else goto Step 5 Step 5:

Get the number from user and add it to sum Step 6: Goto Step

4

Step 7: If limit is reached, goto Step 9 else goto Step 8 Step 8:

Print the numbers

Step 9: Goto Step 7

Step 9: Display sum

Step 10: Stop

```
sum = 0
num=[]
n = input("Enter the Total number of values ")
num = [ int(input("Enter value ")) for i in range(int(n))]
for i in range(int(n)):
    sum = sum + num[i]
print("\nYou have entered")
for i in range(int(n)):
    print(num[i])
print("..and the sum is", sum)
```

Enter the Total number of values 5

Enter value 55

Enter value 62

Enter value 12

Enter value 34

Enter value 20

You have entered

55

62

12

34

20

..and the sum is 183

```
Dict = {1: 'Song A', 2: 'Song B', 3: 'Song C'}
```

```
n=input("Select the number to play your favorite song ")  
# accessing a element using key  
print("You are listening to ")  
print(Dict[int(n)])
```

```
Select the number to play your favorite song 3  
You are listening to  
Song C
```

18CSC207J – Advanced Programming Practice

GUI & Event Handling Programming Paradigm

Event Driven Programming Paradigm

- Event-driven programming is a programming paradigm in which the flow of program execution is determined by events - for example a user action such as a mouse click, key press, or a message from the operating system or another program.
- An event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure.
- In a typical modern event-driven program, there is no discernible flow of control. The main routine is an event-loop that waits for an event to occur, and then invokes the appropriate event-handling routine.
- Event callback is a function that is invoked when something significant happens like when click event is performed by user or the result of database query is available.

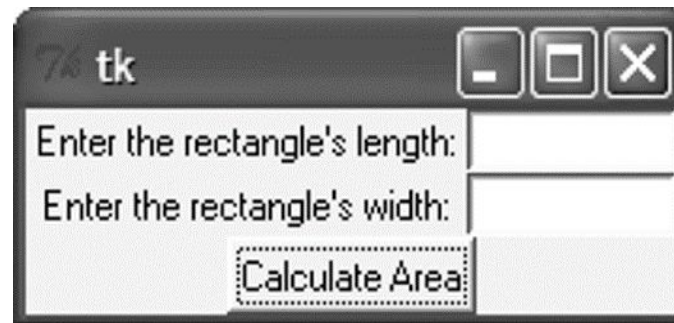
Event Handlers: Event handlers is a type of function or method that run a specific action when a specific event is triggered. For example, it could be a button that when user click it, it will display a message, and it will close the message when user click the button again, this is an event handler.

Trigger Functions: Trigger functions in event-driven programming are a functions that decide what code to run when there are a specific event occurs, which are used to select which event handler to use for the event when there is specific event occurred.

Events: Events include mouse, keyboard and user interface, which events need to be triggered in the program in order to happen, that mean user have to interacts with an object in the program, for example, click a button by a mouse, use keyboard to select a button and etc.

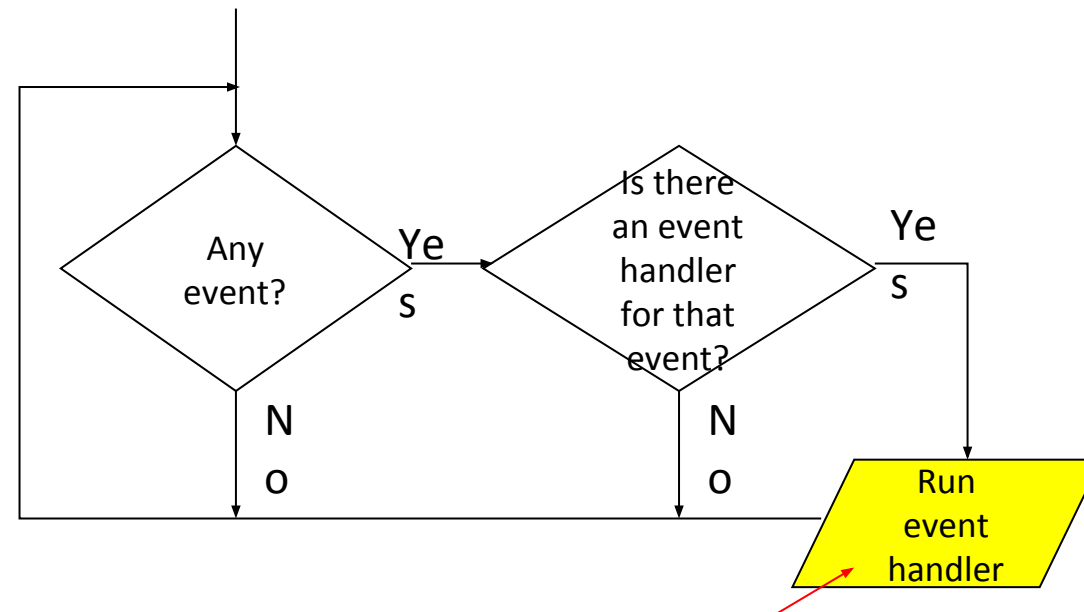
Introduction

- A graphical user interface allows the user to interact with the operating system and other programs using graphical elements such as icons, buttons, and dialog boxes.
- GUIs popularized the use of the mouse.
- GUIs allow the user to point at graphical elements and click the mouse button to activate them.
- GUI Programs Are Event-Driven
- User determines the order in which things happen
- GUI programs respond to the actions of the user, thus they are event driven.
- The tkinter module is a wrapper around tk, which is a wrapper around tcl, which is what is used to create windows and graphical user interfaces.



Introduction

- A major task that a GUI designer needs to do is to determine what will happen when a GUI is invoked
- Every GUI component may generate different kinds of “events” when a user makes access to it using his mouse or keyboard
- E.g. if a user moves his mouse on top of a button, an event of that button will be generated to the Windows system
- E.g. if the user further clicks, then another event of that button will be generated (actually it is the click event)
- For any event generated, the system will first check if there is an event handler, which defines the action for that event
- For a GUI designer, he needs to develop the event handler to determine the action that he wants Windows to take for that event.



GUI Using Python

- Tkinter: Tkinter is the Python interface to the Tk GUI toolkit shipped with Python.
- wxPython: This is an open-source Python interface for wxWindows
- PyQt –This is also a Python interface for a popular cross-platform Qt GUI library.
- JPython: JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine

Tkinter

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

Example:

1. Import the Tkinter module.
2. Create the GUI application main window.
3. Add one or more of the above mentioned widgets to the GUI application.
4. Enter the main event loop to take action against each event triggered by the user.

```
import Tkinter  
top = Tkinter.Tk()  
# Code to add widgets will go here...  
top.mainloop()
```

Tkinter Programming

- Tkinter is the standard GUI library for Python.
- Creating a GUI application using Tkinter

Steps

- Import the Tkinter module.

Import tKinter as tk

- Create the GUI application main window.

root = tk.Tk()

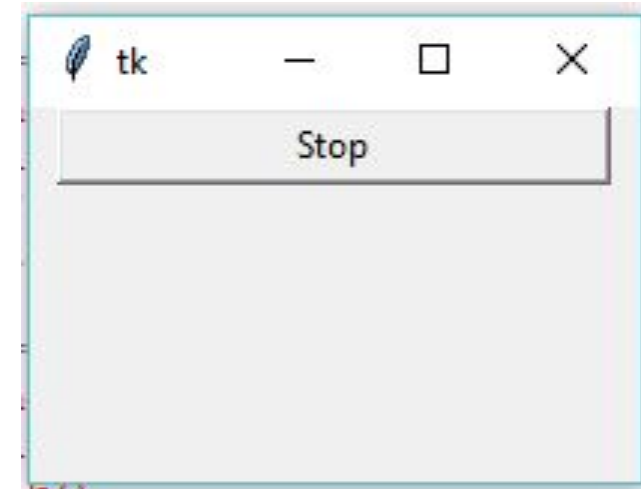
- Add one or more of the above-mentioned widgets to the GUI application.

button = tk.Button(root, text='Stop', width=25, command=root.destroy)

button.pack()

- Enter the main event loop to take action against each event triggered by the user.

root.mainloop()

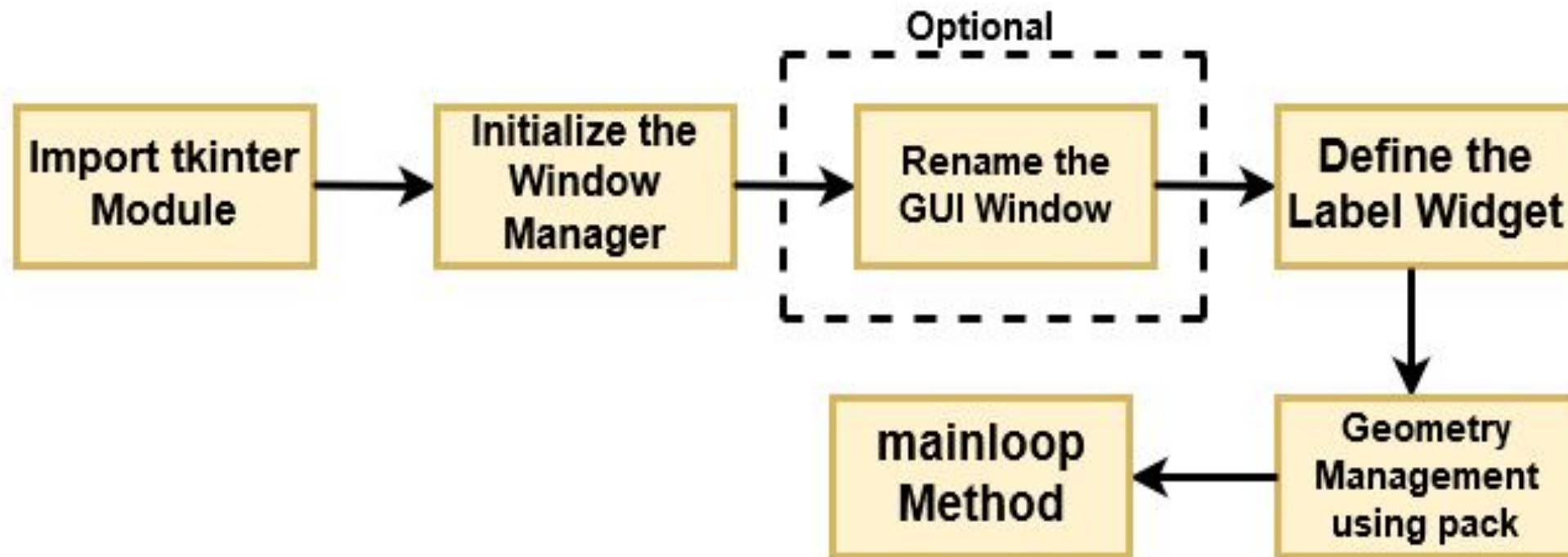


Tkinter widgets

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

Widget	Description
Label	Used to contain text or images
Button	Similar to a Label but provides additional functionality for mouse overs, presses, and releases as well as keyboard activity/events
Canvas	Provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps
Radiobutton	Set of buttons of which only one can be "pressed" (similar to HTML radio input)
Checkbutton	Set of boxes of which any number can be "checked" (similar to HTML checkbox input)
Entry	Single-line text field with which to collect keyboard input (similar to HTML text input)
Frame	Pure container for other widgets
Listbox	Presents user list of choices to pick from
Menu	Actual list of choices "hanging" from a Menubutton that the user can choose from
Menubutton	Provides infrastructure to contain menus (pulldown, cascading, etc.)
Message	Similar to a Label, but displays multi-line text
Scale	Linear "slider" widget providing an exact value at current setting; with defined starting and ending values
Text	Multi-line text field with which to collect (or display) text from user (similar to HTML TextArea)
Scrollbar	Provides scrolling functionality to supporting widgets, i.e., Text, Canvas, Listbox, and Entry
Toplevel	Similar to a Frame, but provides a separate window container

Operation Using Tkinter Widget



Geometry Managers

- The pack() Method – This geometry manager organizes widgets in blocks before placing them in the parent widget.

`widget.pack(pack_options)`

options

- `expand` – When set to true, widget expands to fill any space not otherwise used in widget's parent.
 - `fill` – Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).
 - `side` – Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.
- The grid() Method – This geometry manager organizes widgets in a table-like structure in the parent widget.

`widget.grid(grid_options)`

options –

- `Column/row` – The column or row to put widget in; default 0 (leftmost column).
- `Columnspan, rowsapn`– How many columns or rows to widgetoccupies; default 1.
- `ipadx, ipady` – How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- `padx, pady` – How many pixels to pad widget, horizontally and vertically, outside v's borders.

Geometry Managers

- The place() Method – This geometry manager organizes widgets by placing them in a specific position in the parent widget.

widget.place(place_options)

options –

- anchor – The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget)
- bordermode – INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.
- height, width – Height and width in pixels.
- relheight, relwidth – Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- relx, rely – Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- x, y – Horizontal and vertical offset in pixels.

Common Widget Properties

Common attributes such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

Label Widgets

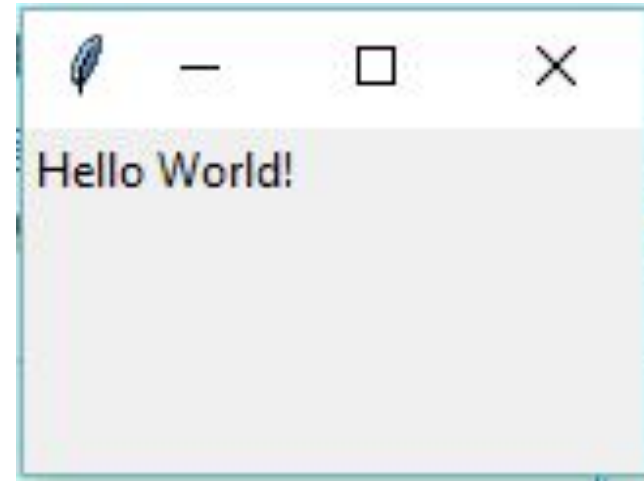
- A label is a widget that displays text or images, typically that the user will just view but not otherwise interact with. Labels are used for such things as identifying controls or other parts of the user interface, providing textual feedback or results, etc.

- Syntax

tk.Label(parent, text="message")

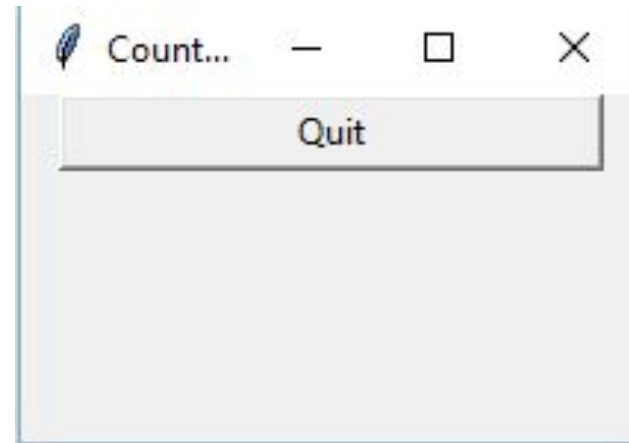
Example:

```
import tkinter as tk  
root = tk.Tk()  
label = tk.Label(root, text='Hello World!')  
label.grid()  
root.mainloop()
```



Button Widgets

```
import tkinter as tk  
  
r = tk.Tk()  
  
r.title('Counting Seconds')  
  
button = tk.Button(r, text='Stop', width=25, command=r.destroy)  
  
button.pack()  
  
r.mainloop()
```



Button Widgets

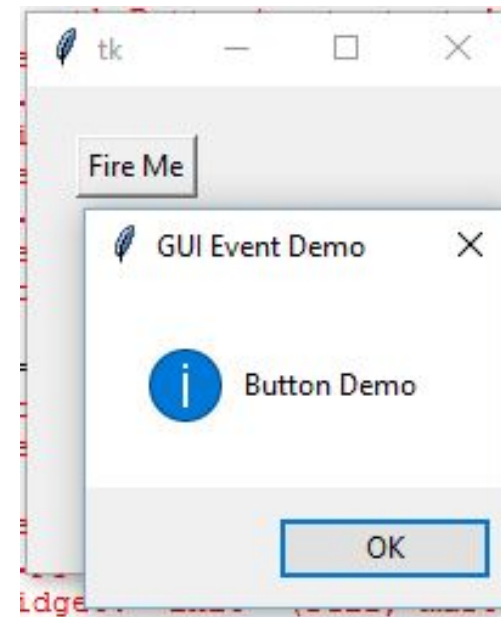
- A button, unlike a frame or label, is very much designed for the user to interact with, and in particular, press to perform some action. Like labels, they can display text or images, but also have a whole range of new options used to control their behavior.

Syntax

```
button = ttk.Button(parent, text='ClickMe', command=submitForm)
```

Example:

```
import tkinter as tk
from tkinter import messagebox
def hello():
    msg = messagebox.showinfo( "GUI Event Demo","Button Demo")
root = tk.Tk()
root.geometry("200x200")
b = tk.Button(root, text='Fire Me',command=hello)
b.place(x=50,y=50)
root.mainloop()
```



Button Widgets

- Button: To add a button in your application, this widget is used.

Syntax :

```
w=Button(master, text="caption" option=value)
```

- master is the parameter used to represent the parent window.
- activebackground: to set the background color when button is under the cursor.
- activeforeground: to set the foreground color when button is under the cursor.
- bg: to set the normal background color.
- command: to call a function.
- font: to set the font on the button label.
- image: to set the image on the button.
- width: to set the width of the button.
- height: to set the height of the button.

Entry Widgets

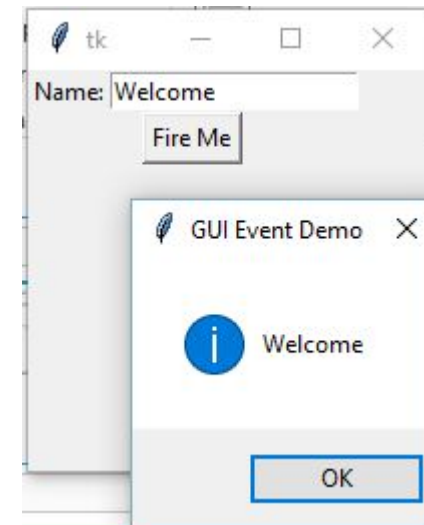
- An entry presents the user with a single line text field that they can use to type in a string value. These can be just about anything: their name, a city, a password, social security number, and so on.

Syntax

```
name = ttk.Entry(parent, textvariable=username)
```

Example:

```
def hello():  
    msg = messagebox.showinfo( "GUI Event Demo",t.get())  
  
root = tk.Tk()  
root.geometry("200x200")  
l1=tk.Label(root,text="Name:")  
l1.grid(row=0)  
t=tk.Entry(root)  
t.grid(row=0,column=1)  
b = tk.Button(root, text='Fire Me',command=hello)  
b.grid(row=1,columnspan=2);  
  
root.mainloop()
```



Canvas

- The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.
- It is used to draw pictures and other complex layout like graphics, text and widgets.

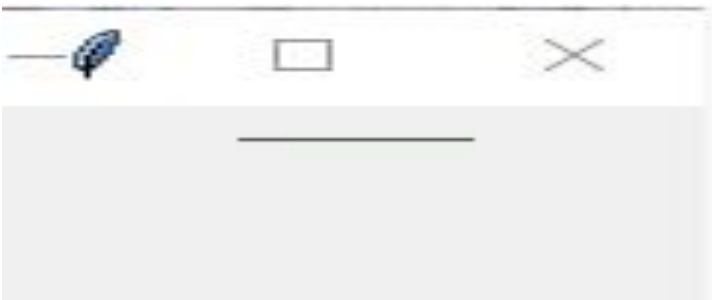
Syntax:

```
w = Canvas(master, option=value)
```

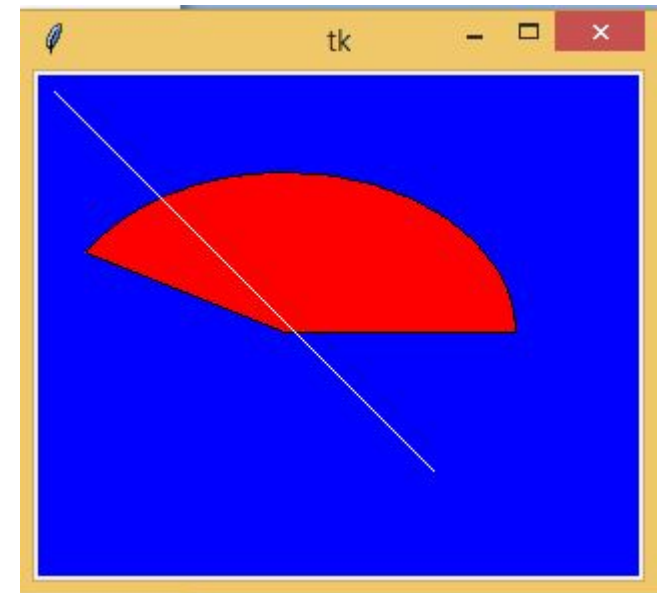
- master is the parameter used to represent the parent window.
- bd: to set the border width in pixels.
- bg: to set the normal background color.
- cursor: to set the cursor used in the canvas.
- highlightcolor: to set the color shown in the focus highlight.
- width: to set the width of the widget.
- height: to set the height of the widget.

Canvas

```
from tkinter import *  
  
master = Tk()  
  
w = Canvas(master, width=40, height=60)  
w.pack()  
  
canvas_height=20  
canvas_width=200  
  
y = int(canvas_height / 2)  
  
w.create_line(0, y, canvas_width, y )  
  
mainloop()
```



```
from tkinter import *  
  
from tkinter import messagebox  
  
top = Tk()  
  
C = Canvas(top, bg = "blue", height = 250, width = 300)  
coord = 10, 50, 240, 210  
  
arc = C.create_arc(coord, start = 0, extent = 150, fill = "red")  
  
line = C.create_line(10,10,200,200,fill = 'white')  
  
C.pack()  
  
top.mainloop()
```



Checkbox

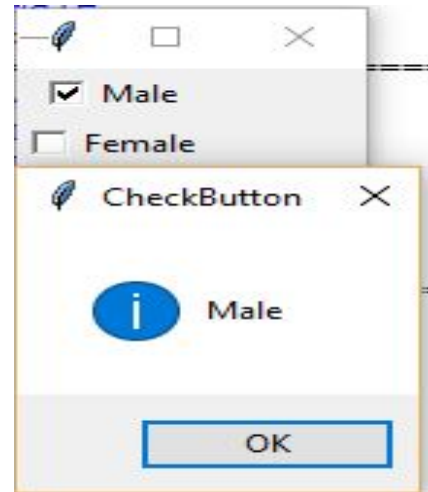
- A checkbox is like a regular button, except that not only can the user press it, which will invoke a command callback, but it also holds a binary value of some kind (i.e. a toggle). Checkbuttons are used all the time when a user is asked to choose between, e.g. two different values for an option.

Syntax

```
w = CheckButton(master, option=value)
```

Example:

```
from tkinter import *
root= Tk()
root.title('Checkbox Demo')
v1=IntVar()
v2=IntVar()
cb1=Checkbutton(root,text='Male', variable=v1,onvalue=1, offvalue=0, command=test)
cb1.grid(row=0)
cb2=Checkbutton(root,text='Female', variable=v2,onvalue=1, offvalue=0, command=test)
cb2.grid(row=1)
root.mainloop()
```



```
def test():
```

```
    if(v1.get()==1 ):
```

```
        v2.set(0)
```

```
        print("Male")
```

```
    if(v2.get()==1):
```

```
        v1.set(0)
```

```
        print("Female")
```

radiobutton

- A radiobutton lets you choose between one of a number of mutually exclusive choices; unlike a checkbox, it is not limited to just two choices. Radiobuttons are always used together in a set and are a good option when the number of choices is fairly small

- **Syntax**

w = CheckButton(master, option=value)

Example:

```
root= Tk()
```

```
root.geometry("200x200")
```

```
radio=IntVar()
```

```
rb1=Radiobutton(root,text='Red', variable=radio,width=25,value=1, command=choice)
```

```
rb1.grid(row=0)
```

```
rb2=Radiobutton(root,text='Blue', variable=radio,width=25,value=2, command=choice)
```

```
rb2.grid(row=1)
```

```
rb3=Radiobutton(root,text='Green', variable=radio,width=25,value=3, command=choice)
```

```
rb3.grid(row=3)
```

```
root.mainloop()
```

```
def choice():
```

```
    if(radio.get()==1):
```

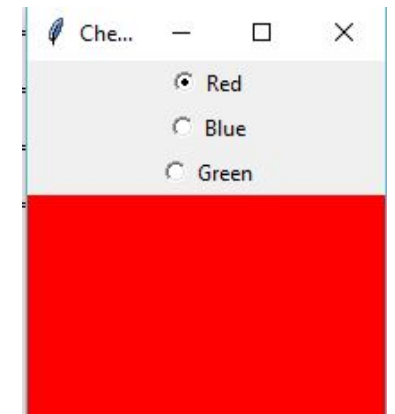
```
        root.configure(background='red')
```

```
    elif(radio.get()==2):
```

```
        root.configure(background='blue')
```

```
    elif(radio.get()==3):
```

```
        root.configure(background='green')
```



Scale

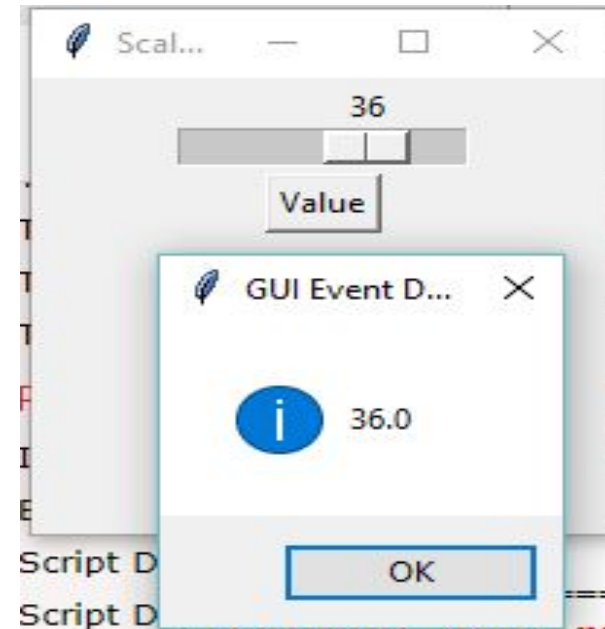
- Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them. We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative to the Entry widget when the user is forced to select only one value from the given range of values.

- Syntax**

w = Scale(top, options)

Example:

```
from tkinter import messagebox
root= Tk()
root.title('Scale Demo')
root.geometry("200x200")
def slide():
    msg = messagebox.showinfo( "GUI Event Demo",v.get())
v = DoubleVar()
scale = Scale( root, variable = v, from_ = 1, to = 50, orient = HORIZONTAL)
scale.pack(anchor=CENTER)
btn = Button(root, text="Value", command=slide)
btn.pack(anchor=CENTER)
root.mainloop()
```



Spinbox

- The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one.

Syntax

w = Spinbox(top, options)

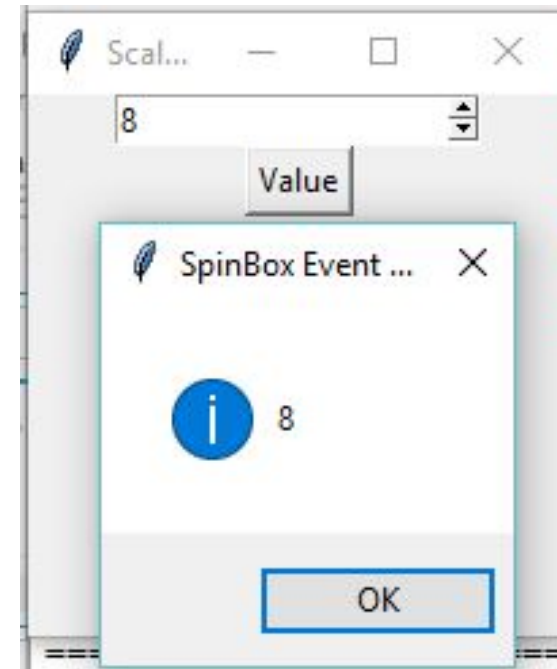
Example:

```
from tkinter import *
from tkinter import messagebox

root= Tk()
root.title('Scale Demo')
root.geometry("200x200")

def slide():
    msg = messagebox.showinfo( "SpinBox Event Demo",spin.get())

spin = Spinbox(root, from_ = 0, to = 25)
spin.pack(anchor=CENTER)
btn = Button(root, text="Value", command=slide)
btn.pack(anchor=CENTER)
root.mainloop()
```



Menubutton

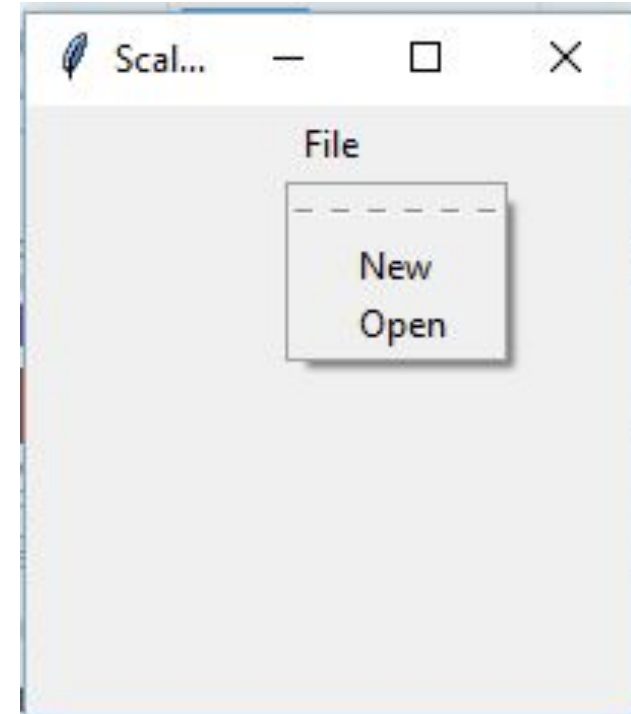
- Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

Syntax

w = Menubutton(Top, options)

Example:

```
from tkinter import *
from tkinter import messagebox
root= Tk()
root.title('Scale Demo')
root.geometry("200x200")
menubutton = Menubutton(root, text = "File", relief = FLAT)
menubutton.grid()
menubutton.menu = Menu(menubutton)
menubutton["menu"]=menubutton.menu
menubutton.menu.add_checkbutton(label = "New", variable=IntVar(),command=)
menubutton.menu.add_checkbutton(label = "Open", variable = IntVar())
menubutton.pack()
root.mainloop()
```



Menubutton

- Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

Syntax

w = Menubutton(Top, options)

Example:

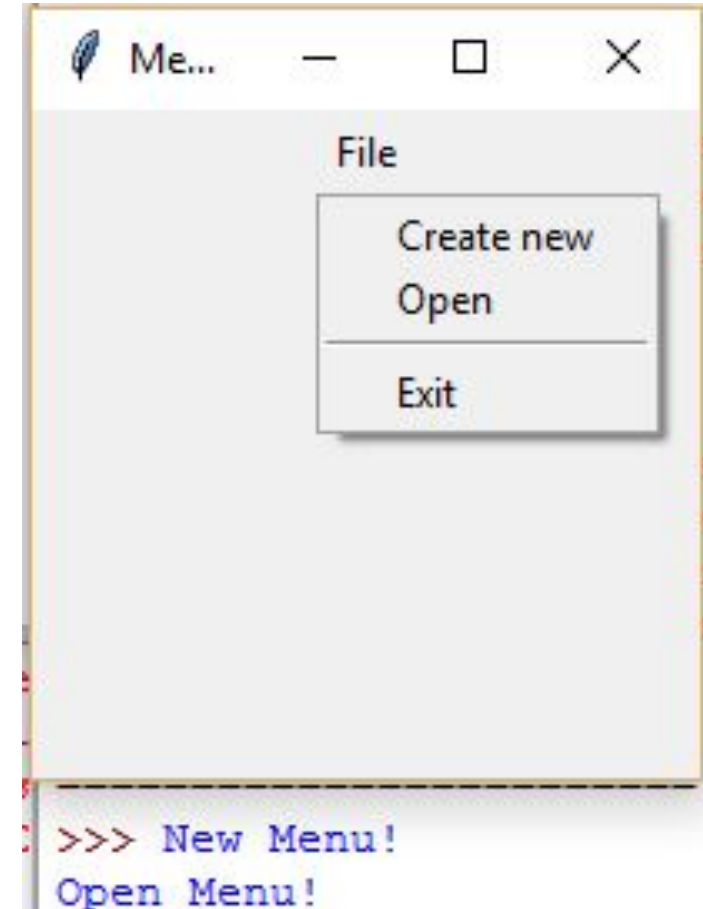
```
from tkinter import *
from tkinter import messagebox

root= Tk()
root.title('Menu Demo')
root.geometry("200x200")

def new():
    print("New Menu!")

def disp():
    print("Open Menu!")

menubutton = Menubutton(root, text="File")
menubutton.grid()
menubutton.menu = Menu(menubutton, tearoff = 0)
menubutton["menu"] = menubutton.menu
menubutton.menu.add_command(label="Create new",command=new)
menubutton.menu.add_command(label="Open",command=disp)
menubutton.menu.add_separator()
menubutton.menu.add_command(label="Exit",command=root.quit)
menubutton.pack()
```



Imperative Programming Paradigm

Introduction

- A program based on this paradigm is made up of a clearly-defined sequence of instructions to a computer.
- Control flow in imperative programming is explicit: commands show how the computation takes place, step by step. Each step affects the global state of the computation.
- Source code for imperative languages is a series of commands, which specify what the computer has to do – and when – in order to achieve a desired result. Values used in variables are changed at program runtime. To control the commands, control structures such as loops or branches are integrated into the code.
- Closest to the actual mechanical behavior of a computer \Rightarrow original imperative languages were abstractions of assembly language.
- Imperative programs define sequences of commands/statements for the computer that change a program state (i.e., set of variables)
 - Commands are stored in memory and executed in the order found
 - Commands retrieve data, perform a computation, and assign the result to a memory location

Introduction

- The imperative programming paradigm assumes that the computer can maintain through environments of variables any changes in a computation process. Computations are performed through a guided sequence of steps, in which these variables are referred to or changed. The order of the steps is crucial, because a given step will have different consequences depending on the current values of variables when the step is executed.
- When a variable has state, something must maintain that state, which means that the variable is tied to a specific processor. Imperative coding works on simple applications, but code executes too slowly for optimal results on complex data science applications.
- Owing to its comparatively slower and sequential execution strategy, it cannot be used for complex or parallel computations.

Introduction

Central elements of imperative paradigm:

- Assignment statement: assigns values to memory locations and changes the current state of a program
- Variables refer to memory locations
- Step-by-step execution of commands
- Control-flow statements: Conditional and unconditional (GO TO) branches and loops to change the flow of a program
- First Imperative Languages : FOTRAN, ALGOL, COBOL

Introduction

- Imperative programming languages are very specific, and operation is system-oriented. On the one hand, the code is easy to understand; on the other hand, many lines of source text are required to describe what can be achieved with a fraction of the commands using declarative programming languages.
- The different imperative programming languages can, in turn, be assigned to three further subordinate programming styles – structured, procedural, and modular.
 - The structured programming style extends the basic imperative principle with specific control structures: sequences, selection, and iteration.
 - The procedural approach divides the task a program is supposed to perform into smaller sub-tasks, which are individually described in the code.
 - The modular programming model goes one step further by designing, developing, and testing the individual program components independently of one another.
- Imperative programming languages are characterized by their instructive nature and, thus, require significantly more lines of code to express what can be described with just a few instructions in the declarative style.

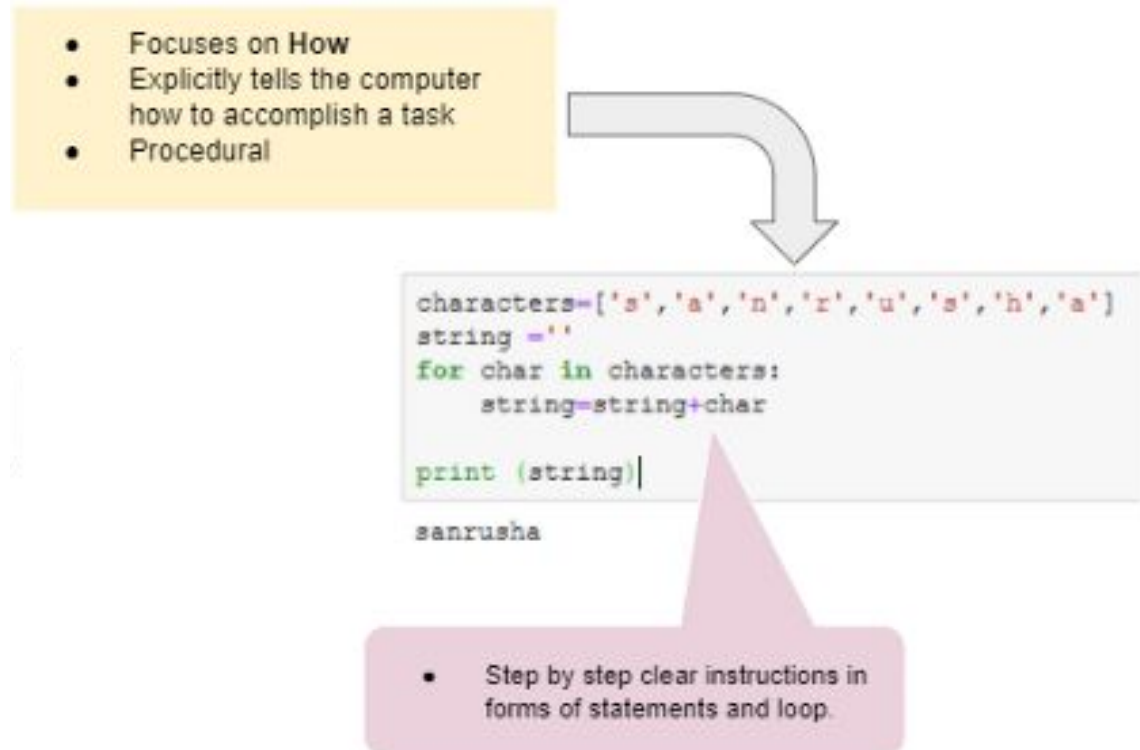
Imperative Programming Paradigm

- Imperative programming provides explicit steps to the computer using statements, loops, etc.

- Imperative Program

- Focus on **How**
- Explicitly tells the computer how to accomplish a task
- Procedural

- Focuses on How
- Explicitly tells the computer how to accomplish a task
- Procedural



```
characters=['s','a','n','r','u','s','h','a']  
string = ''  
for char in characters:  
    string=string+char  
print (string)|  
sanrussha
```

- Step by step clear instructions in forms of statements and loop.

- In a computer program, a variable stores the data. The contents of these locations at any given point in the program's execution are called the program's state.
- Imperative programming is characterized by programming with state and commands which modify the state.
- The first imperative programming languages were machine languages.
- In imperative programming set of commands can be grouped in a code block also called Procedure.

Example

Example:

```
sample_characters = ['p','y','t','h','o','n']
sample_string = ""
sample_string
for c in sample_characters:
    sample_string = sample_string + c
    print(sample_string)
```

```
characters=['s','a','n','r','u','s','h','a']
string=""
for char in characters:
    string=string+char

print (string)
```

```
def stringc(characters):
    string=""
    for char in characters:
        string=string+char
    return string
```

```
import functools
characters=['s','a','n','r','u','s','h','a']
string=functools.reduce(lambda s,c:s+c,characters)
```

Imperative

Advantages	Disadvantages
Easy to read	Code quickly becomes very extensive and thus confusing
Relatively easy to learn	Higher risk of errors when editing
Conceptual model (solution path) is very easy for beginners to understand	System-oriented programming means that maintenance blocks application development
Characteristics of specific applications can be taken into account	Optimization and extension is more difficult

Declarative Programming Paradigm

Introduction

- General programming paradigm in which programs express the logic of a computation without describing its control flow.
- Programs describe what the computation should accomplish, rather than how it should accomplish it.
- Typically avoids the notion of variable holding state, and function side-effects.
- Contrary to imperative programming, where a program is a series of steps and state changes describing how the computation is achieved.
- Includes diverse languages/subparadigms such as:
 - Database query languages (e.g. SQL, Xquery)
 - XSLT
 - Makefiles
 - Constraint programming
 - Logic programming
 - Functional programming

Introduction

“Pure” declarative languages:

- SQL (Structured Query Language – language to interact with databases):
 - `SELECT * FROM Customers WHERE Country='Mexico';`
- Markup languages, like HTML, CSS (Cascading Style Sheets – language to describe styling of e.g. HTML pages), ...
 - `<h1 style="color:blue;">This is a Blue Heading</h1>`
- Functional programming languages like Haskell (even though they allow some “encapsulated” imperative parts)