

# 18CSC207J – Advanced Programming Practice

## Unit 4

- Logic Programming Paradigm
- Dependent Type Programming Paradigm
- Network Programming Paradigm

## Unit 5

- Symbolic Programming paradigm
- Automata Programming Paradigm
- GUI Programming paradigm

# 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.

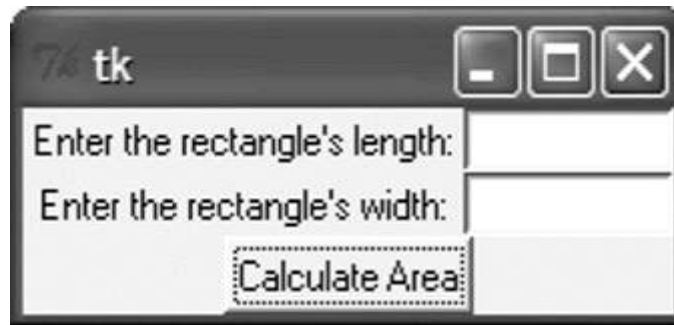
**Event Handlers:** Event handlers is a type of function or method that run a specific action when a specific event is triggered. For example, it could be a button that when user click it, it will display a message, and it will close the message when user click the button again, this is an event handler.

**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.

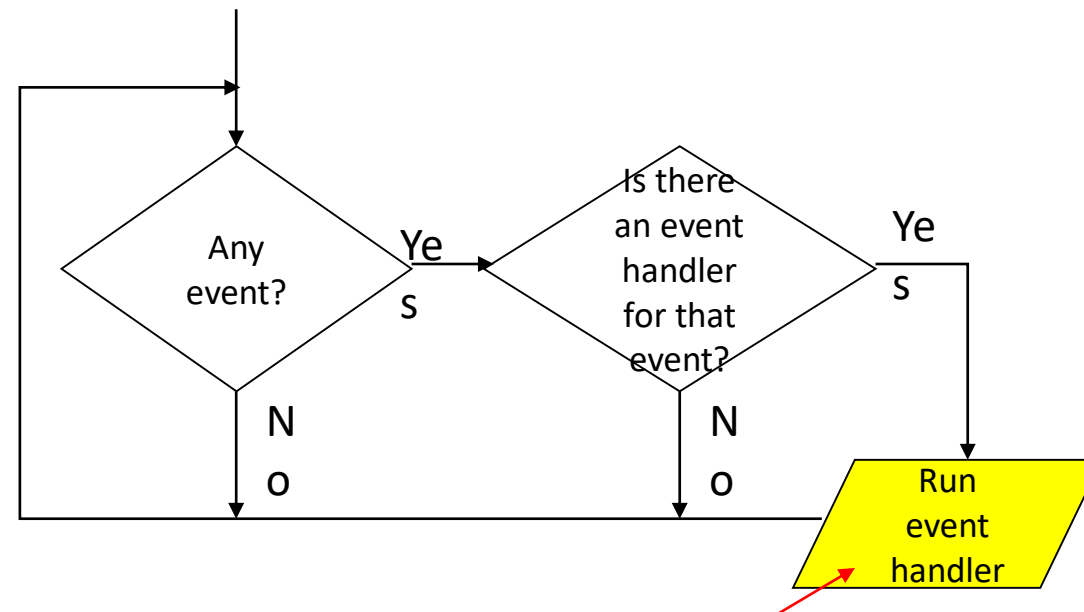
# Introduction

- 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 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()*

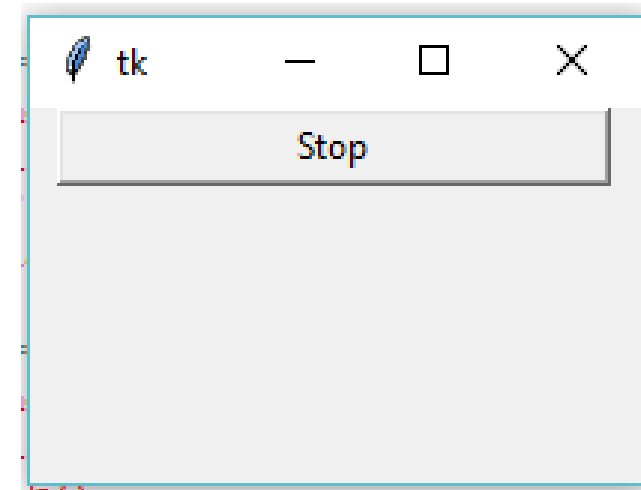
- Add one or more of the above-mentioned widgets to the GUI application.

*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()*



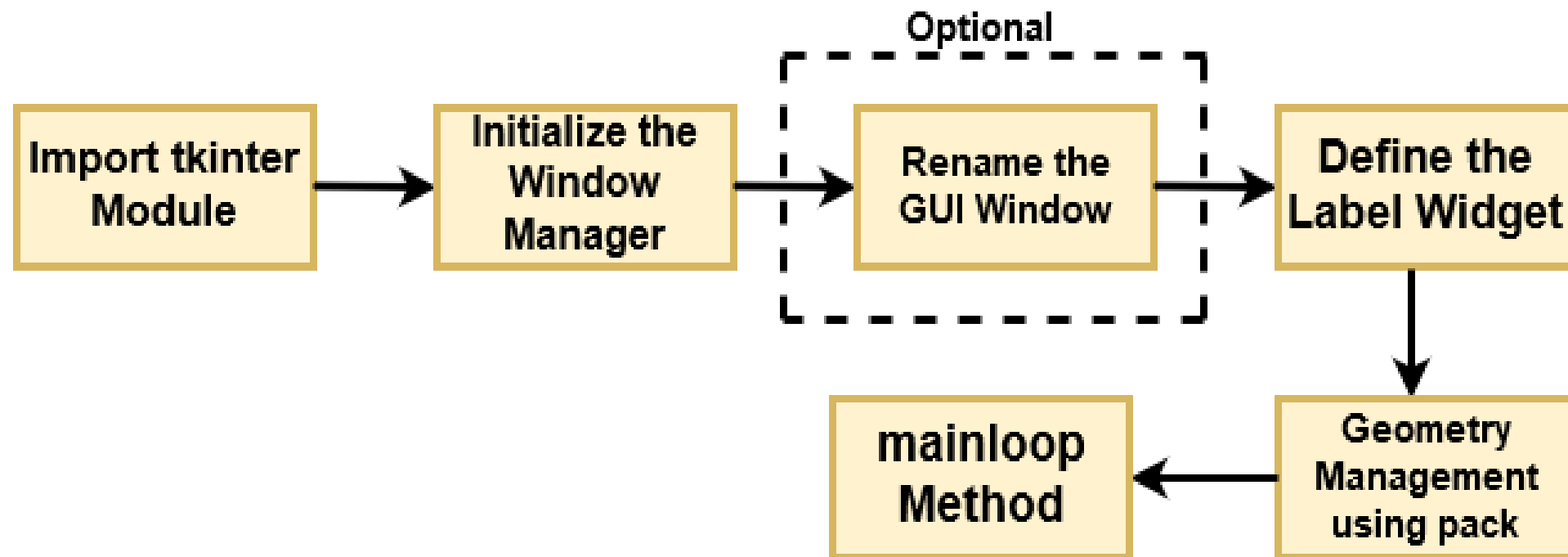


# Tkinter widgets

- 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( eg. Flat, Raised)
- Bitmaps
- Cursors

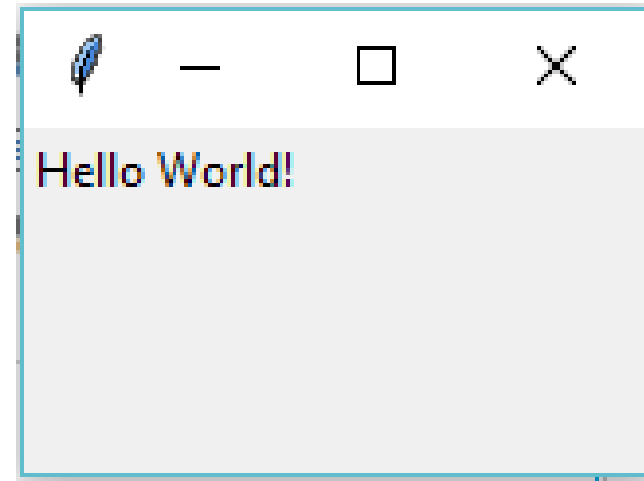
# Label Widgets

- 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")`*

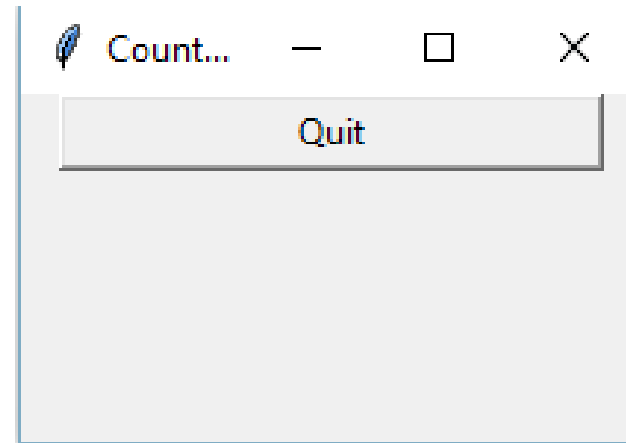
## **Example:**

```
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('Example')  
button = tk.Button(r, text='Stop', width=25)  
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)
```

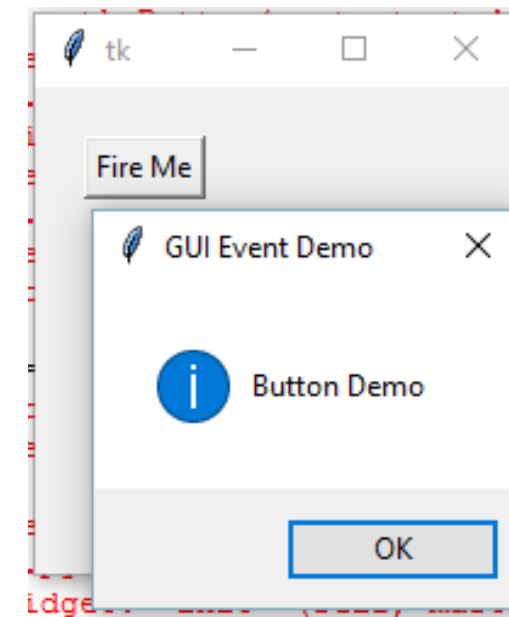
## Example:

```
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

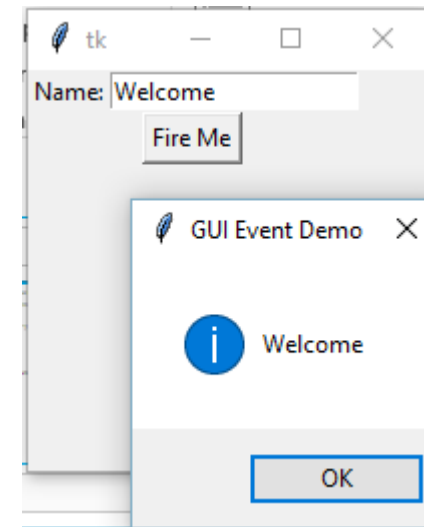
- An entry presents the user with a single line text field that they can use to type in a string value. These can be just about anything: their name, a city, a password, social security number, and so on.

## 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)`

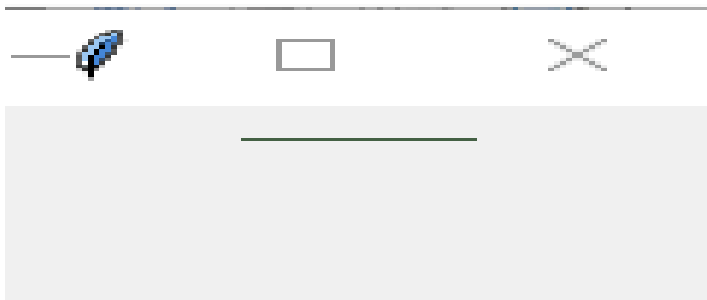
- master is the parameter used to represent the parent window.
- bd: to set the border width in pixels.
- bg: to set the normal background color.
- cursor: to set the cursor used in the canvas.
- highlightcolor: to set the color shown in the focus highlight.
- width: to set the width of the widget.
-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

**To Create line: `create_line(coords, options)`**

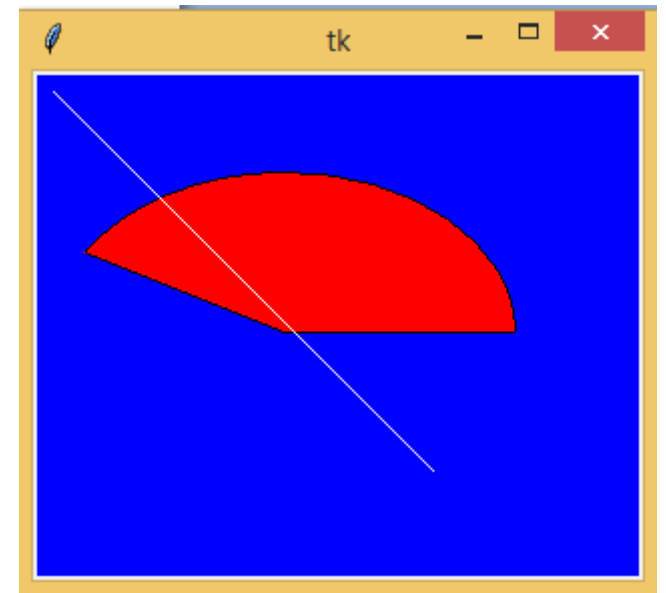
**To create\_arc(*x0, y0, x1, y1, option, ...*)**

# 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

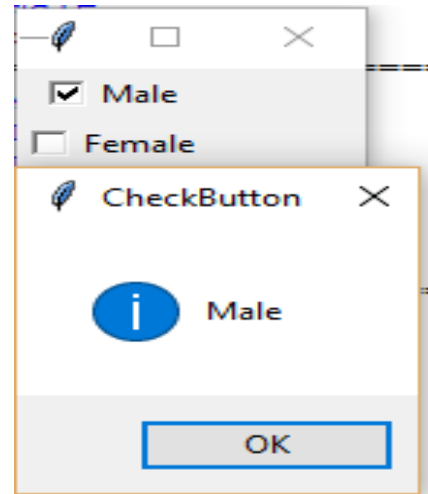
- A checkbox is like a regular button, except that not only can the user press it, which will invoke a command callback, but it also holds a binary value of some kind (i.e. a toggle). Checkbuttons are used all the time when a user is asked to choose between, e.g. two different values for an option.

## 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 checkbutton, 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 = Radiobutton(master, option=value)*

**Example:**

***from tkinter import \****

```
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)
```

```
def choice():
```

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

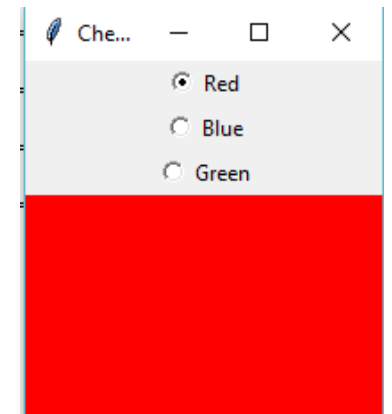
```
        root.configure(background='red')
```

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

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

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

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



# Scale

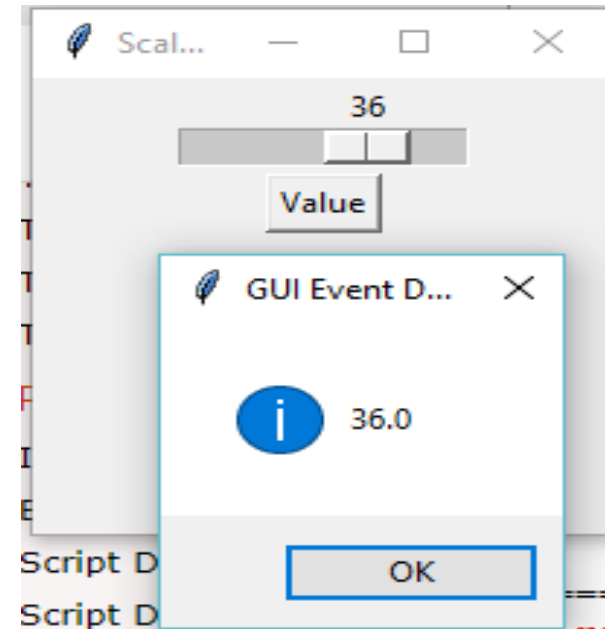
- Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them. We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative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

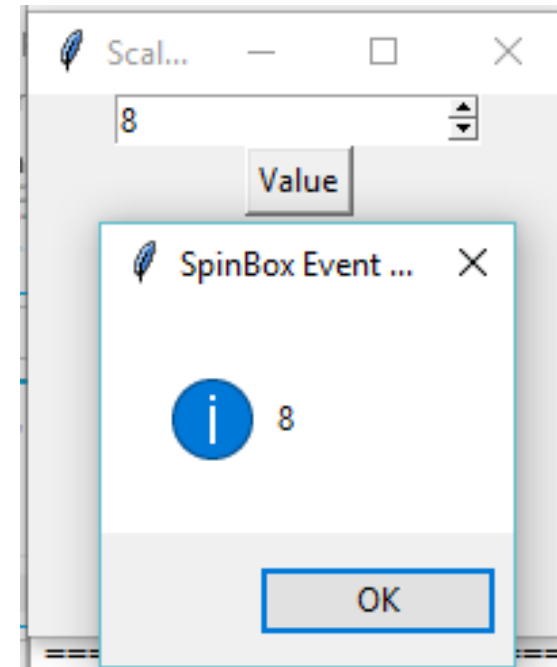
## Example:

```
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

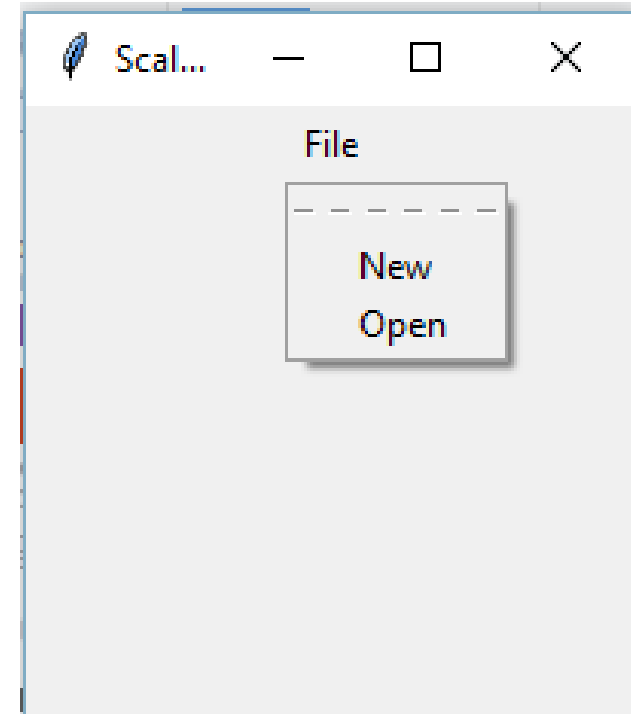
- Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

## Syntax

*w = Menubutton(Top, options)*

## Example:

```
from tkinter import *
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())
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

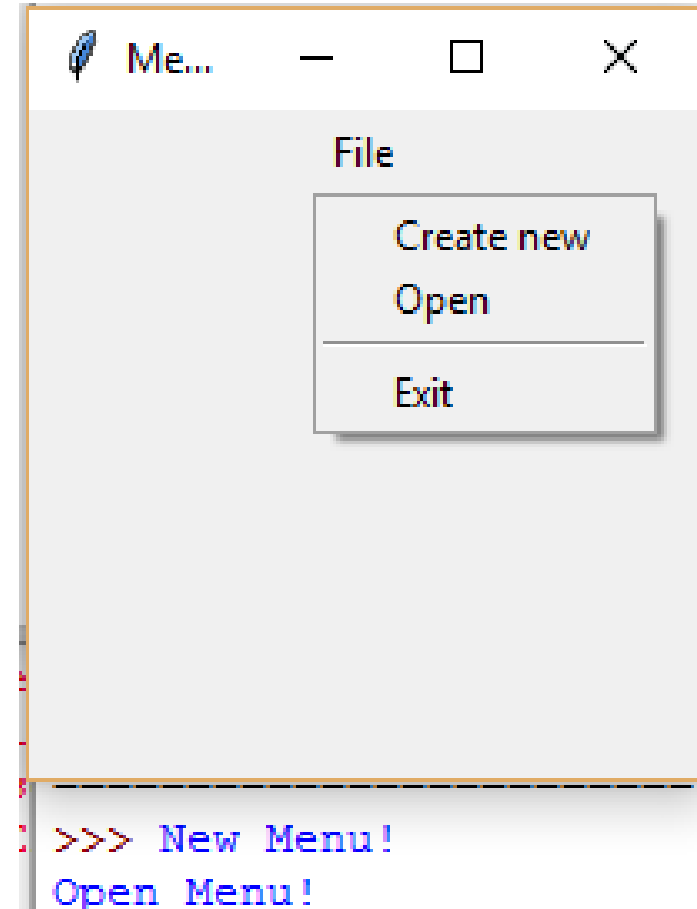
root= Tk()
root.title('Menu Demo')
root.geometry("200x200")

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()
```



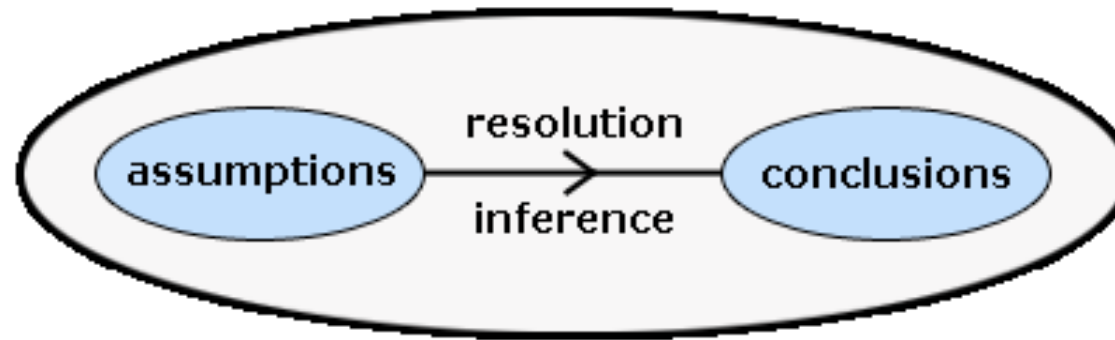
# Logical Programming Paradigm

# Logical Programming Paradigm

- It can be an abstract model of computation.
- Solve logical problems like puzzles
- Have knowledge base which we know before and along with the question you specify knowledge and how that knowledge is to be applied through a series of rules
- The Logical Paradigm takes a declarative approach to problem-solving.
- Various logical assertions about a situation are made, establishing all known facts.
- Then queries are made.

# Logical Programming Paradigm

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. A logic is a language. It has syntax and semantics. It. More than a language, it has inference rules.



## Syntax:

Syntax: the rules about how to form formulas; usually the easy part of a logic.

## Semantics:

About the meaning carried by the formulas, mainly in terms of logical consequences.

## Inference rules:

Inference rules describe correct ways to derive conclusions.

# Logical Programming Paradigm

## Logic :

A Logic program is a set of predicates. Ex parent, siblings

## Predicates :

Define relations between their arguments. Logically, a Logic program states what holds. Each predicate has a name, and zero or more arguments. The predicate name is a atom. Each argument is an arbitrary Logic term. A predicate is defined by a collection of clauses.

Example: Mother(x,y)

## Clause :

A clause is either a rule or a fact. The clauses that constitute a predicate denote logical alternatives: If any clause is true, then the whole predicate is true.

Example:

Mother(X,Y) <= female(X) & parent(X,Y) # implies X is the mother of Y, if X has to female

# Parts of Logical Programming Paradigm

- A series of definitions/declarations that define the problem domain (fact)
- Statements of relevant facts (rules)
- Statement of goals in the form of a query (query)

## Example:

**Given information about fatherhood and motherhood, determine grand parent relationship**

E.g. Given the information called facts

John is father of Lily

Kathy is mother of Lily

Lily is mother of Bill

Ken is father of Karen

Who are grand parents of Bill?

Who are grand parents of Karen?

# Logical Programming Paradigm

## Fact

A fact must start with a predicate (which is an atom). The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists.

Facts are axioms; relations between terms that are assumed to be true.

Example facts:

+big('horse')

+big('elephant')

+small('cat')

+brown('horse')

+black('cat')

+grey('elephant')

Consider the 3 fact saying 'cat' is a smallest animal and fact 6 saying the elephant is grey in color

## Rule

Rules are theorems that allow new inferences to be made.

dark(X)<=black(X)

dark(X)<=brown(X)

Consider rule 1 saying the color is black its consider to be dark color.



# Logical Programming Paradigm

## Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

```
print(pyDatalog.ask('father_of(X,jess)'))
```

## Output:

```
{('jack',)}
```

```
X
```

```
print(father_of(X,'jess'))
```

## Output:

```
jack
```

```
X
```

# Logical Programming Paradigm

```
from pyDatalog import pyDatalog

pyDatalog.create_atoms('parent,male,female,son,daughter,X,Y,Z')

+male('adam')
+female('anne')
+female('barney')
+male('james')
+parent('barney','adam')
+parent('james','anne')
```

#The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male.

#The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female.

```
son(X,Y)<= male(X) & parent(Y,X)
```

```
daughter(X,Y)<= parent(Y,X) & female(X)
```

```
print(pyDatalog.ask('son(adam,Y)'))
```

```
print(pyDatalog.ask('daughter(anne,Y)'))
```

```
print(son('adam',X))
```

# Logical Programming Paradigm

```
pyDatalog.create_terms('factorial, N')
```

```
factorial[N] = N*factorial[N-1]
```

```
factorial[1] = 1
```

```
print(factorial[3]==N)
```

# Logical Programming Paradigm

```
from pyDatalog import pyDatalog

pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager, indirect_manager, count_of_indirect_reports')

# Mary works in Production
+works_in('Mary', 'Production')
+works_in('Sam', 'Marketing')
+works_in('John', 'Production')
+works_in('John', 'Marketing')
+(manager['Mary'] == 'John')
+(manager['Sam'] == 'Mary')
+(manager['Tom'] == 'Mary')


indirect_manager(X,Y) <= (manager[X] == Y)
print(works_in(X, 'Marketing'))

indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)
print(indirect_manager('Sam',X))
```

# Logical Programming Paradigm

Lucy is a Professor

Danny is a Professor

James is a Lecturer

All professors are Dean

Write a Query to retrieve all deans?

Soln

```
from pyDatalog import pyDatalog
```

```
pyDatalog.create_terms('X,Y,Z,professor,lecturer, dean')
```

```
+professor('lucy')
```

```
+professor('danny')
```

```
+lecturer('james')
```

```
dean(X)<=professor(X)
```

```
print(dean(X))
```

# Logical Programming Paradigm

likes(john, susie).                /\* John likes Susie \*/

likes(X, susie).                /\* Everyone likes Susie \*/

likes(john, Y).                /\* John likes everybody \*/

likes(john, Y), likes(Y, john).    /\* John likes everybody and everybody likes John \*/

likes(john, susie); likes(john, mary). /\* John likes Susie or John likes Mary \*/

not(likes(john, pizza)).            /\* John does not like pizza \*/

likes(john, susie) :- likes(john, mary). /\* John likes Susie if John likes Mary.

rules

friends(X, Y) :- likes(X, Y), likes(Y, X).        /\* X and Y are friends if they like each other \*/

hates(X, Y) :- not(likes(X, Y)).                /\* X hates Y if X does not like Y. \*/

enemies(X, Y) :- not(likes(X, Y)), not(likes(Y, X)). /\* X and Y are enemies if they don't like each other \*/

# Network Programming Paradigm

# Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.



# What is Socket?

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.
- An endpoint is a combination of IP address and the port number.

For Client-Server communication,

- Sockets are to be configured at the two ends to initiate a connection,
- Listen for incoming messages
- Send the responses at both ends
- Establishing a bidirectional communication.

# Socket Types

## Datagram Socket

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

## Stream Socket:

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

# Socket in Python

```
sock_obj = socket.socket( socket_family, socket_type, protocol=0)
```

**socket\_family:** - Defines family of protocols used as transport mechanism.

Either AF\_UNIX, or

AF\_INET (IP version 4 or IPv4).

**socket\_type:** Defines the types of communication between the two end-points.

SOCK\_STREAM (for connection-oriented protocols, e.g., TCP), or

SOCK\_DGRAM (for connectionless protocols e.g. UDP).

**protocol:** We typically leave this field or set this field to zero.

**Example:**

```
#Socket client example in python
```

```
import socket
```

```
#create an AF_INET, STREAM socket (TCP)
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
print 'Socket Created'
```

# Socket Creation

```
import socket

import sys

try:

    #create an AF_INET, STREAM socket (TCP)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

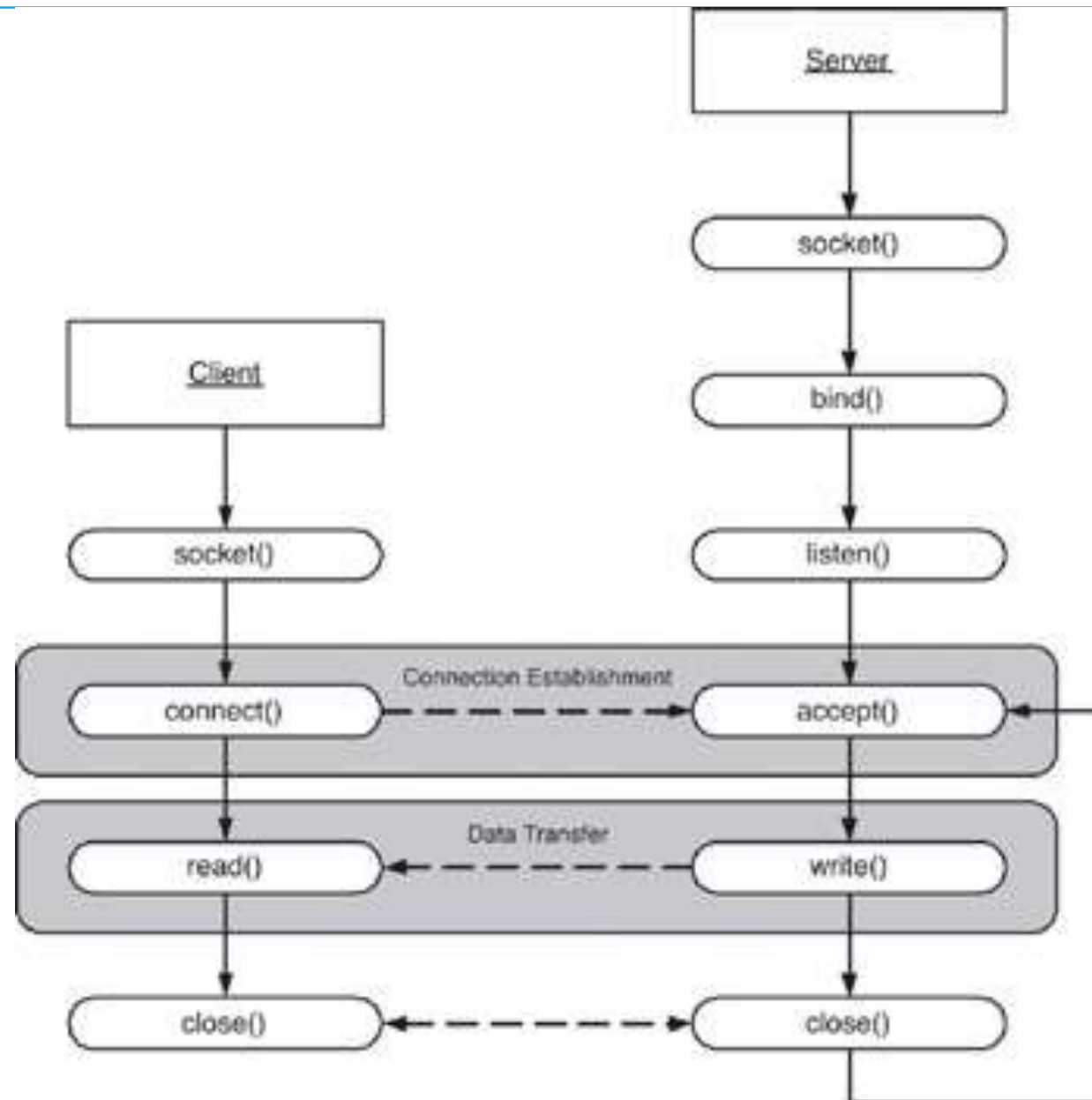
except socket.error, msg:

    print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]

    sys.exit();

print 'Socket Created'
```

# Client/server symmetry in Sockets applications



# Socket in Python

To create a socket, we must use `socket.socket()` function available in the Python socket module, which has the general syntax as follows:

***`S = socket.socket(socket_family, socket_type, protocol=0)`***

`socket_family`: This is either `AF_UNIX` or `AF_INET`. We are only going to talk about `INET` sockets in this tutorial, as they account for at least 99% of the sockets in use.

`socket_type`: This is either `SOCK_STREAM` or `SOCK_DGRAM`.

`Protocol`: This is usually left out, defaulting to 0.

## Client Socket Methods

Following are some client socket methods:

`connect( )` : To connect to a remote socket at an address. An address format(host, port) pair is used for `AF_INET` address family.

# Socket in Python

## Server Socket Methods

`bind( )`: This method binds the socket to an address. The format of address depends on socket family mentioned above(`AF_INET`).

`listen(backlog)` : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

`accept( )` : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(`conn`, `address`) where `conn` is a new socket object which can be used to send and receive data on that connection, and `address` is the address bound to the socket on the other end of the connection.

# General Socket in Python

`sock_object.recv():`

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

`sock_object.send():`

Apply this method to send messages from endpoints in case the protocol is TCP.

`sock_object.recvfrom():`

Call this method to receive messages at endpoints if the protocol used is UDP.

`sock_object.sendto():`

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

`sock_object.gethostname():`

This method returns hostname.

`sock_object.close():`

This method is used to close the socket. The remote endpoint will not receive data from this side.





# Simple TCP Client

```
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()      # Get current machine name
port = 9999                      # Client wants to connect to server's
                                # port number 9999

s.connect((host,port))
print s.recv(1024)               # 1024 is bufsize or max amount
                                # of data to be received at once
s.close()
```

# Simple UDP Server

```
#!/usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()          # Host IP
udp_port = 12345                                # specified port to connect

#print type(sock) =====> 'type' can be used to see type
                        # of any variable ('sock' here)

sock.bind((udp_host,udp_port))

while True:
    print "Waiting for client..."
    data,addr = sock.recvfrom(1024)          #receive data from client
    print "Received Messages:",data," from",addr
```

# Simple UDP Client

```
#!/usr/bin/python

import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()    # Host IP
udp_port = 12345                    # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port

sock.sendto(msg, (udp_host, udp_port))    # Sending message to UDP server
```

# Symbolic Programming Paradigm

# Introduction

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

It Covers the following:

- As A calculator and symbols
- Algebraic Manipulations - Expand and Simplify
- Calculus – Limits, Differentiation, Series , Integration
- Equation Solving – Matrices

# Calculator and Symbols

## Rational - $\frac{1}{2}$ , or $\frac{5}{2}$

```
>>import sympy as sym
```

```
>>a = sym.Rational(1, 2)
```

```
>>a
```

Answer will be  $\frac{1}{2}$

## Constants like pi,e

```
>>sym.pi**2
```

Answer is  $\pi^2$

```
>>sym.pi.evalf()
```

Answer is 3.14159265358979

```
>> (sym.pi + sym.exp(1)).evalf()
```

Answer is 5.85987448204884

## X AND Y

```
>> x = sym.Symbol('x')
```

```
>>y = sym.Symbol('y')
```

```
>>x + y + x - y
```

Answer is  $2*x$

# Algebraic Manipulations

**EXPAND (X+Y)\*\*3 = X+3X^2Y+3XY^2+Y**

```
>> sym.expand((x + y) ** 3)
```

Answer is  $x^{**3} + 3*x^{**2}*y + 3*x*y^{**2} + y^{**3}$

```
>> 3 * x * y ** 2 + 3 * y * x ** 2 + x ** 3 + y ** 3
```

Answer is  $x^{**3} + 3*x^{**2}*y + 3*x*y^{**2} + y^{**3}$

**WITH TRIGNOMETRY LIKE SIN,COSINE**

eg.  $\cos(X+Y) = -\sin(X)*\sin(Y) + \cos(X)*\cos(Y)$

```
>> sym.expand(sym.cos(x + y), trig=True)
```

Answer is  $-\sin(x)*\sin(y) + \cos(x)*\cos(y)$

**SIMPLIFY**

$(X+X*Y/X)=Y+1$

```
>> sym.simplify((x + x * y) / x)
```

Answer is:  $y+1$



# Calculus

## LIMITS compute the limit of

limit(function, variable, point)

limit( sin(x)/x , x, 0) =1

## Differentiation

diff(func,var) eg diff(sin(x),x)=cos(x)

diff(func,var,n) eg

## Series

series(expr,var)

series(cos(x),x) =  $1 - x/2 + x^2/24 + o(x)$

## Integration

Integrate(expr,var)

Integrate(sin(x),x) = -cos(x)

# Example

## Example:

```
In [23]: sym.expand(sym.cos(x + y), trig=True)
```

```
Out[23]: -sin(x)*sin(y) + cos(x)*cos(y)
```

```
In [24]: sym.limit(sym.sin(x) / x, x, 0)
```

```
Out[24]: 1
```

```
In [26]: sym.diff(sym.sin(x), x)
```

```
Out[26]: cos(x)
```

```
In [27]: sym.diff(sym.sin(2 * x), x)
```

```
Out[27]: 2*cos(2*x)
```

```
In [28]: sym.diff(sym.tan(x), x)
```

```
Out[28]: tan(x)**2 + 1
```

```
In [29]: sym.diff(sym.sin(2 * x), x, 1)
```

```
Out[29]: 2*cos(2*x)
```

```
In [30]: sym.diff(sym.sin(2 * x), x, 2)
```

```
Out[30]: -4*sin(2*x)
```

```
In [31]: sym.diff(sym.sin(2 * x), x, 3)
```

```
Out[31]: -8*cos(2*x)
```

```
In [31]: sym.diff(sym.sin(2 * x), x, 3)
```

```
Out[31]: -8*cos(2*x)
```

```
In [32]: sym.series(sym.cos(x), x)
```

```
Out[32]: 1 - x**2/2 + x**4/24 + O(x**6)
```

```
In [34]: sym.integrate(6 * x ** 5, x)
```

```
Out[34]: x**6
```

```
In [35]: sym.integrate(sym.sin(x), x)
```

```
Out[35]: -cos(x)
```

```
In [36]: sym.integrate(sym.log(x), x)
```

```
Out[36]: x*log(x) - x
```

```
In [37]: sym.integrate(2 * x + sym.sinh(x), x)
```

```
Out[37]: x**2 + cosh(x)
```

```
In [37]: sym.integrate(2 * x + sym.sinh(x), x)
```

```
Out[37]: x**2 + cosh(x)
```

```
In [38]: sym.integrate(sym.exp(-x ** 2) * sym.erf(x), x)
```

```
Out[38]: sqrt(pi)*erf(x)**2/4
```

```
In [39]: sym.integrate(x**3, (x, -1, 1))
```

```
Out[39]: 0
```

```
In [40]: sym.integrate(sym.sin(x), (x, 0, sym.pi / 2))
```

```
Out[40]: 1
```

# Example

## Example:

```
In [43]: sym.solve(x ** 4 - 1, x)
```

```
Out[43]: {-1, 1, -I, I}
```

```
In [44]: sym.solve(sym.exp(x) + 1, x)
```

```
Out[44]: ImageSet(Lambda(_n, I*(2*_n*pi + pi)), S.Integers)
```

```
In [46]: solution = sym.solve((x + 5 * y - 2, -3 * x + 6 * y - 15), (x, y))  
solution[x], solution[y]
```

```
Out[46]: (-3, 1)
```

```
In [47]: f = x ** 4 - 3 * x ** 2 + 1  
sym.factor(f)
```

```
Out[47]: (x**2 - x - 1)*(x**2 + x - 1)
```

```
In [48]: sym.satisfiable(x & y)
```

```
Out[48]: {x: True, y: True}
```

```
In [49]: sym.Matrix([[1, 0], [0, 1]])
```

```
Out[49]: Matrix(  
[1, 0],  
[0, 1])
```

```
In [51]: x, y = sym.symbols('x, y')  
A = sym.Matrix([[1, x], [y, 1]])  
A
```

```
Out[51]: Matrix(  
[1, x],  
[y, 1])
```

```
In [52]: A**2
```

```
Out[52]: Matrix(  
[x*y + 1, 2*x],  
[2*y, x*y + 1])
```

# Equation Solving

## **solveset()**

`solveset(x**4 - 1, x) = {-1, 1, -I, I}`

## **Matrices**

`A=[[1,2],[2,1]] find A**2`

# Automata Based Programming Paradigm

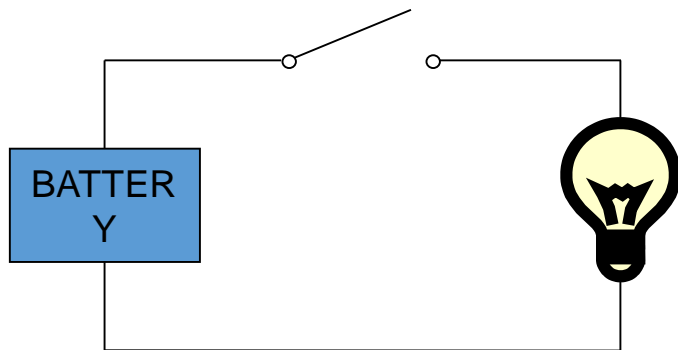
# Introduction

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other formal automation.

## What is Automata Theory?

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, ...

## Example:



input: switch

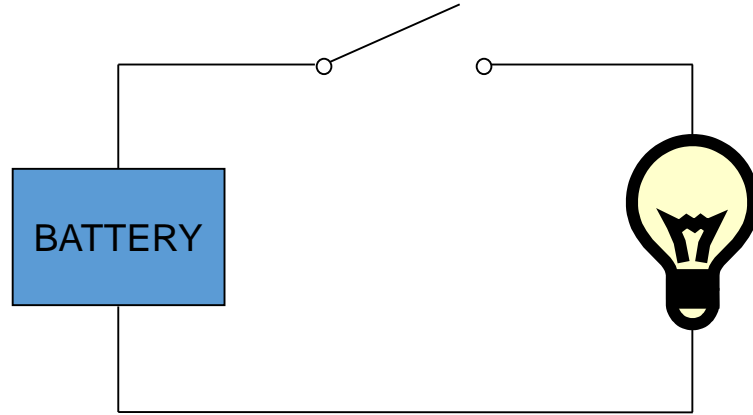
output: light bulb

actions: flip switch

states: on, off

# Simple Computer

Example:

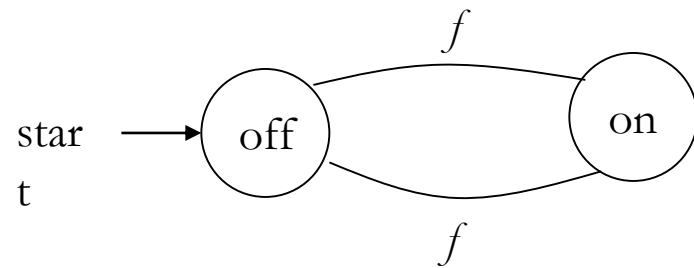


input: switch

output: light bulb

actions: flip switch

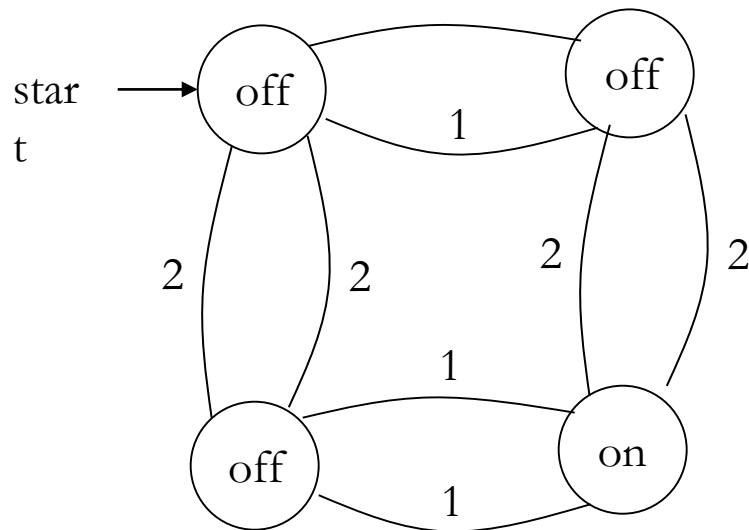
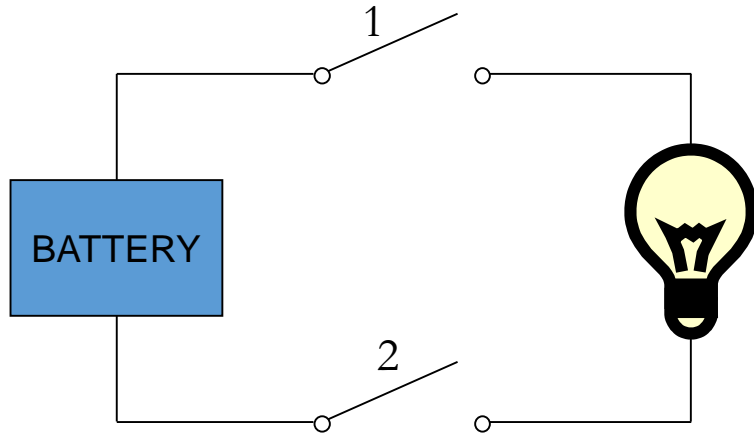
states: on, off



bulb is on if and only if there was an odd number of flips

# Another “computer”

Example:



inputs: switches 1 and 2

actions: 1 for “flip switch 1”

actions: 2 for “flip switch 2”

states: on, off

bulb is on if and only if both switches were flipped an odd number of times



# Types of Automata

finite automata	Devices with a finite amount of memory. Used to model “small” computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

# Alphabets and strings

A common way to talk about words, number, pairs of words, etc. is by representing them as strings

To define strings, we start with an alphabet

An **alphabet** is a finite set of symbols.

**Examples:**

$\Sigma_1 = \{a, b, c, d, \dots, z\}$ : the set of letters in English

$\Sigma_2 = \{0, 1, \dots, 9\}$ : the set of (base 10) digits

$\Sigma_3 = \{a, b, \dots, z, \#\}$ : the set of letters plus the special symbol #

$\Sigma_4 = \{ (, ) \}$ : the set of open and closed brackets

# Strings

A **string** over alphabet  $\Sigma$  is a finite sequence of symbols in  $\Sigma$ .

The empty string will be denoted by  $\epsilon$

## Examples:

abfbz is a string over  $\Sigma_1 = \{a, b, c, d, \dots, z\}$

9021 is a string over  $\Sigma_2 = \{0, 1, \dots, 9\}$

ab#bc is a string over  $\Sigma_3 = \{a, b, \dots, z, \#\}$

))()( is a string over  $\Sigma_4 = \{ (, ) \}$

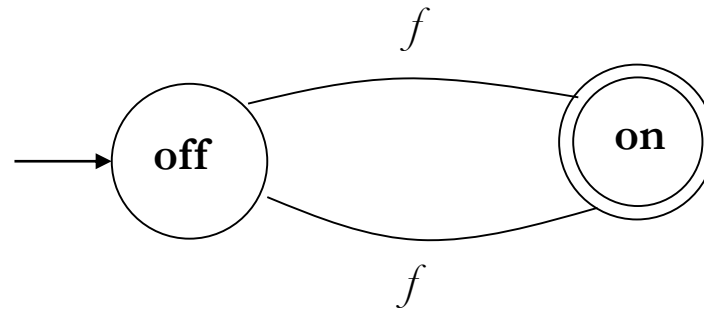
# Languages

A **language** is a set of strings over an alphabet.

Languages can be used to describe problems with “yes/no” answers, for example:

- $L_1 =$  The set of all strings over  $\Sigma_1$  that contain the substring “SRM”
- $L_2 =$  The set of all strings over  $\Sigma_2$  that are divisible by 7 = {7, 14, 21, ...}
- $L_3 =$  The set of all strings of the form  $s\#s$  where  $s$  is any string over  $\{a, b, \dots, z\}$
- $L_4 =$  The set of all strings over  $\Sigma_4$  where every ( can be matched with a subsequent )

# Finite Automata



There are states off and on, the automaton starts in off and tries to reach the “good state” on

What sequences of fs lead to the good state?

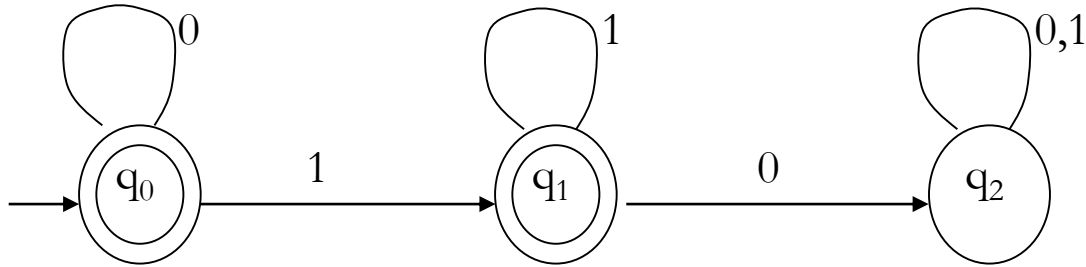
Answer:  $\{f, fff, fffff, \dots\} = \{f^n : n \text{ is odd}\}$

This is an example of a deterministic finite automaton over alphabet  $\{f\}$

# Deterministic finite automata

- A **deterministic finite automaton** (DFA) is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where
  - $Q$  is a finite set of **states**
  - $\Sigma$  is an **alphabet**
  - $\delta: Q \times \Sigma \rightarrow Q$  is a **transition function**
  - $q_0 \in Q$  is the **initial state**
  - $F \subseteq Q$  is a set of **accepting states** (or **final states**).
- In diagrams, the accepting states will be denoted by double loops

# Example



alphabet  $\Sigma = \{0, 1\}$

start state  $Q = \{q_0, q_1, q_2\}$

initial state  $q_0$

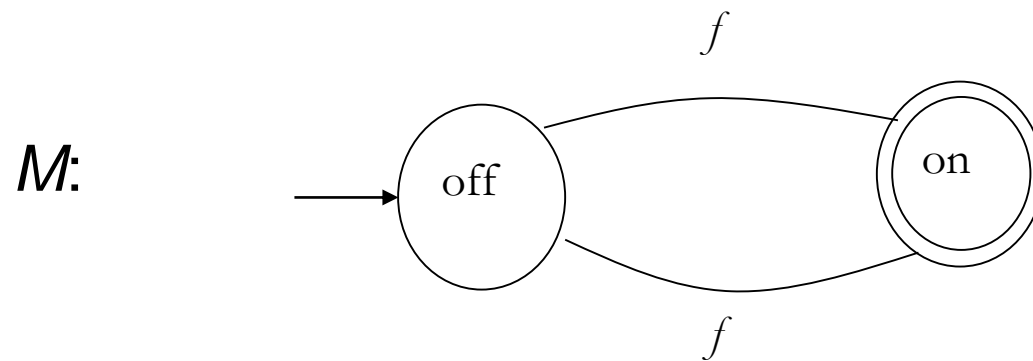
accepting states  $F = \{q_0, q_1\}$

transition function  $\delta$ :

		inputs	
		0	1
states	$q_0$	$q_0$	$q_1$
	$q_1$	$q_2$	$q_1$
	$q_2$	$q_2$	$q_2$

# Language of a DFA

The **language of a DFA**  $(Q, \Sigma, \delta, q_0, F)$  is the set of all strings over  $\Sigma$  that, starting from  $q_0$  and following the transitions as the string is read left to right, will reach some accepting state.

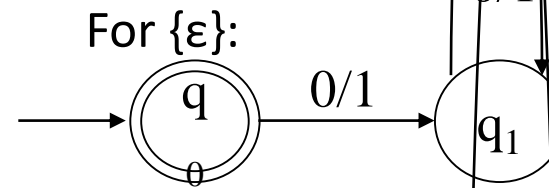
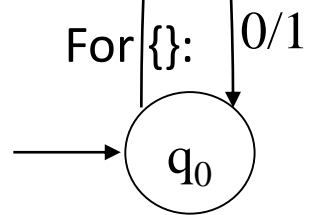


- Language of  $M$  is  $\{f, fff, fffff, \dots\} = \{f^n: n \text{ is odd}\}$

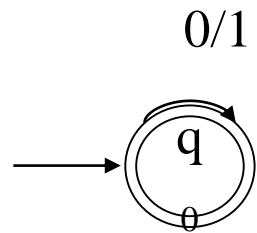


# Example of DFA

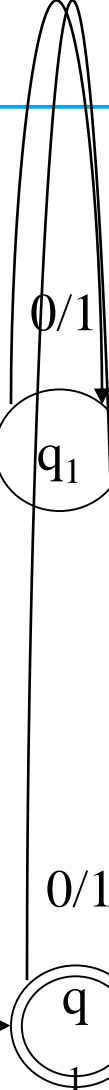
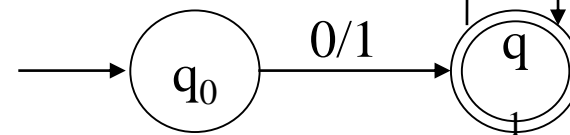
1. Let  $\Sigma = \{0, 1\}$ . Give DFAs for  $\{\}$ ,  $\{\epsilon\}$ ,  $\Sigma^*$ , and  $\Sigma^+$ .



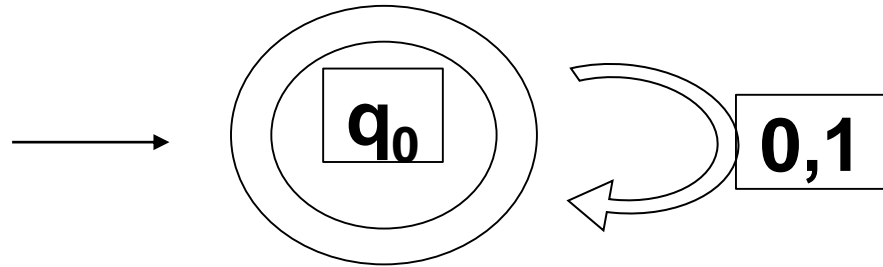
For  $\Sigma^*$ :



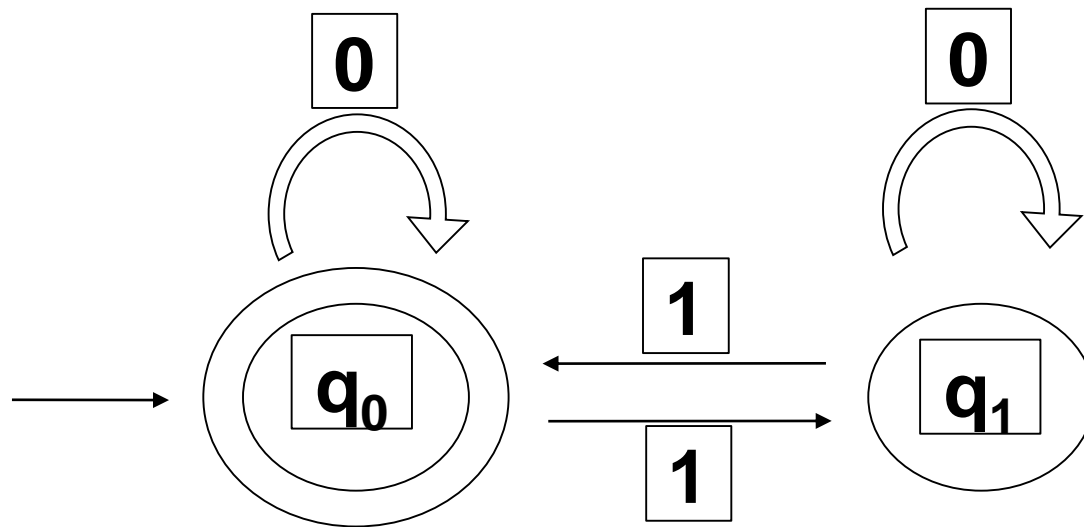
For  $\Sigma^+$ :



# Example of DFA



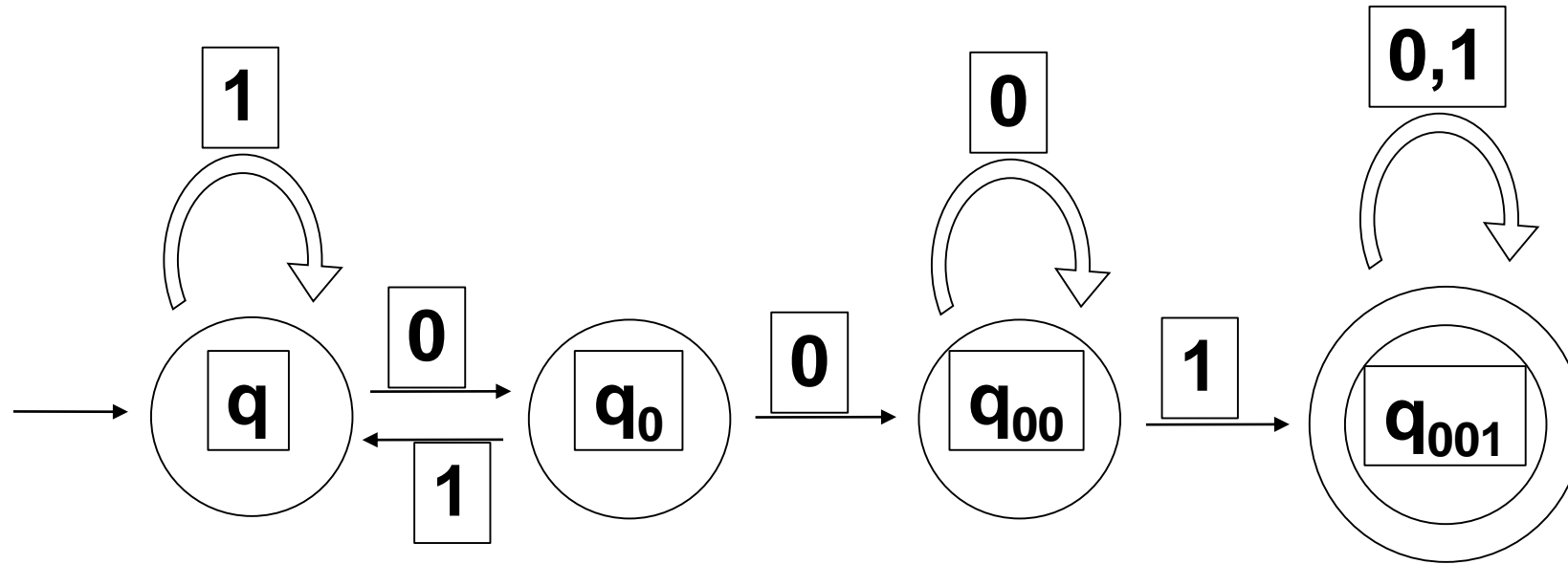
$$L(M) = \{0,1\}^*$$



$$L(M) = \{ w \mid w \text{ has an even number of 1s} \}$$

# Example of DFA

Build an automaton that accepts all and only those strings that contain 001



# Example of DFA using Python

```
from automata.fa.dfa import DFA
# DFA which matches all binary strings ending in an odd number of '1's
dfa = DFA(
    states={'q0', 'q1', 'q2'},
    input_symbols={'0', '1'},
    transitions={
        'q0': {'0': 'q0', '1': 'q1'},
        'q1': {'0': 'q0', '1': 'q2'},
        'q2': {'0': 'q2', '1': 'q1'}
    },
    initial_state='q0',
    final_states={'q1'}
)
dfa.read_input('01') # answer is 'q1'
dfa.read_input('011') # answer is error
print(dfa.read_input_stepwise('011'))
Answer # yields:
# 'q0'    # 'q0'    # 'q1'
# 'q2'    # 'q1'
```

```
if dfa.accepts_input('011'):
    print('accepted')
else:
    print('rejected')
```

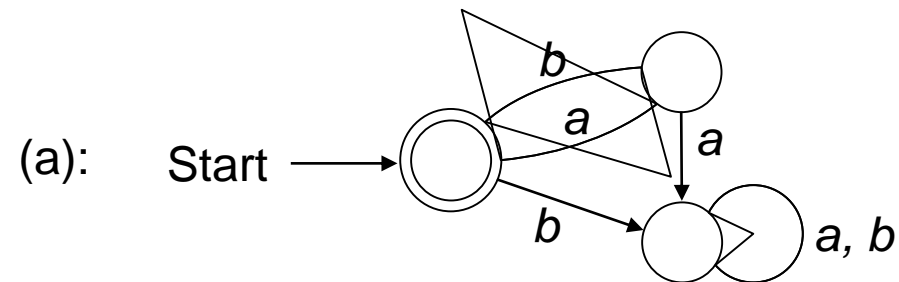
# Questions for DFA

Find an DFA for each of the following languages over the alphabet  $\{a, b\}$ .

(a)  $\{(ab)^n \mid n \in \mathbb{N}\}$ , which has regular expression  $(ab)^*$ .

*Solution*

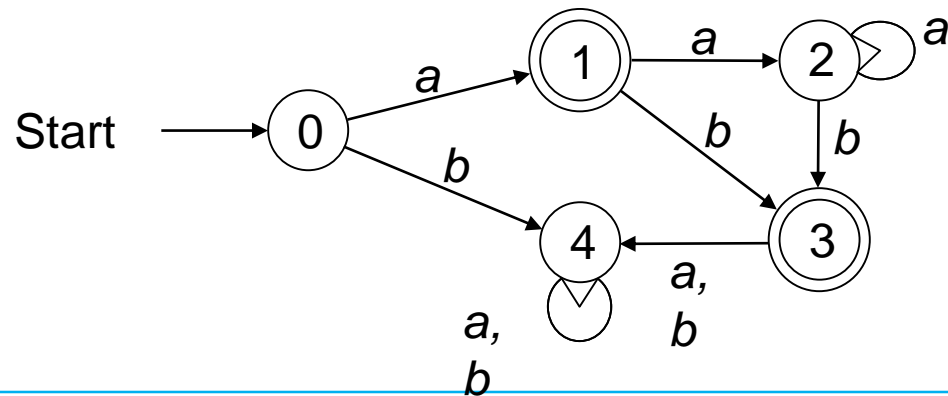
:



b) Find a DFA for the language of  $a + aa^*b$ .

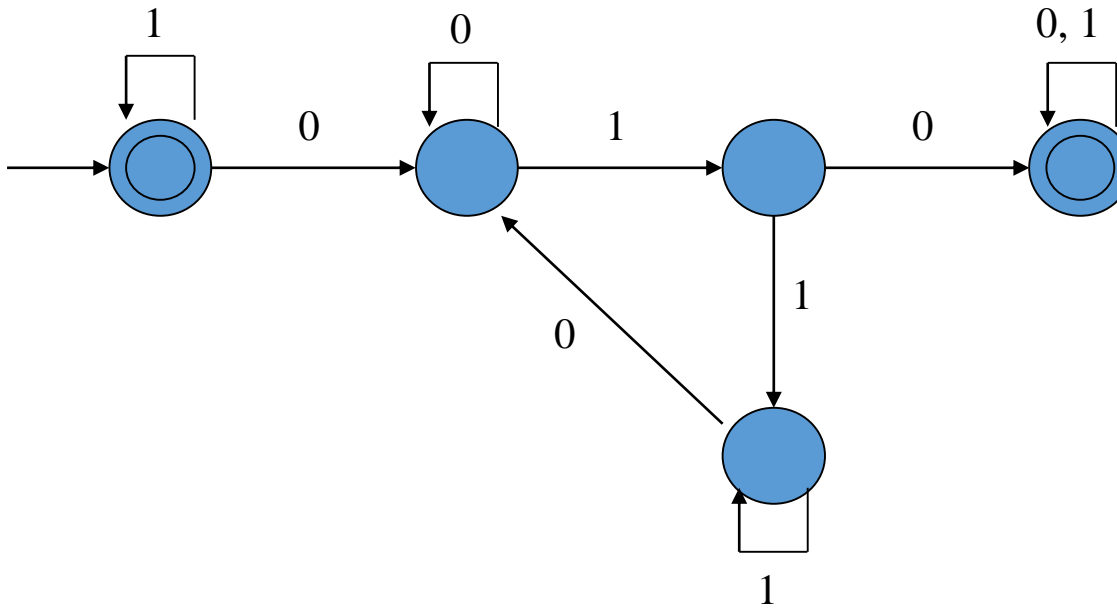
*Solution*

:



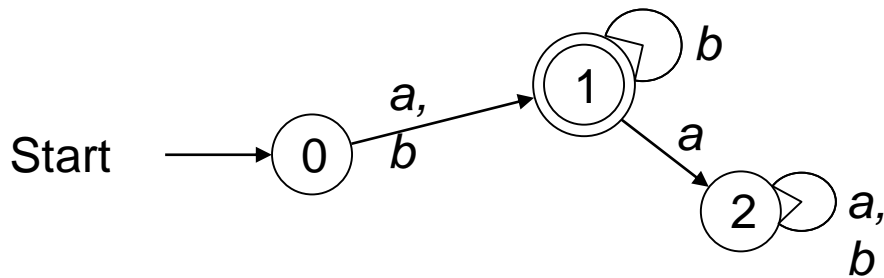
# Questions for DFA

c) A DFA that accepts all strings that contain 010 or do not contain 0.



# Table Representation of a DFA

A DFA over  $A$  can be represented by a transition function  $T : \text{States} \times A \rightarrow \text{States}$ , where  $T(i, a)$  is the state reached from state  $i$  along the edge labelled  $a$ , and we mark the start and final states. For example, the following figures show a DFA and its transition table.



	$T$	$a$	$b$
start	0	1	1
final	1	2	1
	2	2	2

# Sample Exercises - DFA

1. Write a automata code for the Language that accepts all and only those strings that contain 001
2. Write a automata code for  $L(M) = \{ w \mid w \text{ has an even number of 1s} \}$
3. Write a automata code for  $L(M) = \{0,1\}^*$
4. Write a automata code for  $L(M) = a + aa^*b$ .
5. Write a automata code for  $L(M) = \{(ab)^n \mid n \in \mathbb{N}\}$
6. Write a automata code for Let  $\Sigma = \{0, 1\}$ .

Given DFAs for  $\{\}$ ,  $\{\epsilon\}$ ,  $\Sigma^*$ , and  $\Sigma^+$ .

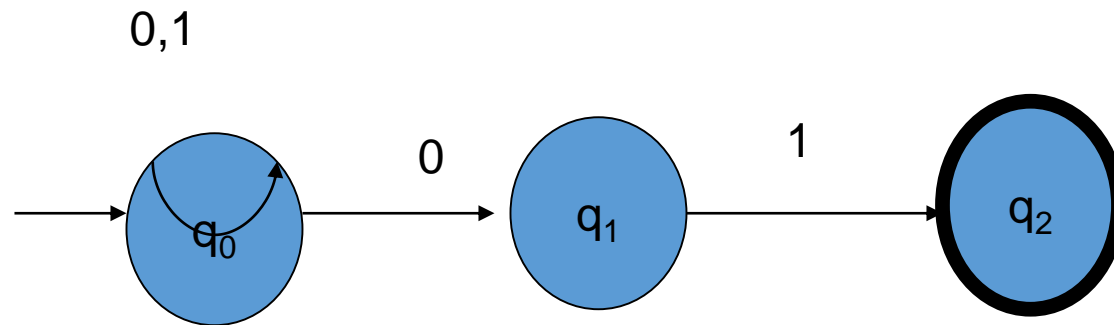


# NDFA

- A nondeterministic finite automaton  $M$  is a five-tuple  $M = (Q, \Sigma, \delta, q_0, F)$ , where:
  - $Q$  is a finite set of states of  $M$
  - $\Sigma$  is the finite input alphabet of  $M$
  - $\delta: Q \times \Sigma \rightarrow \text{power set of } Q$ , is the state transition function mapping a state-symbol pair to a subset of  $Q$
  - $q_0$  is the start state of  $M$
  - $F \subseteq Q$  is the set of accepting states or final states of  $M$

# Example NDFA

- NFA that recognizes the language of strings that end in 01



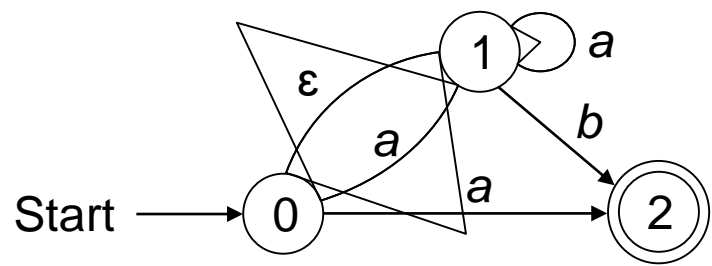
note:  $\delta(q_0, 0) = \{q_0, q_1\}$   
 $\delta(q_1, 0) = \{\}$

Exercise: Draw the complete transition table for this NFA

# NDFA

A nondeterministic finite automaton (NFA) over an alphabet  $A$  is similar to a DFA except that epsilon-edges are allowed, there is no requirement to emit edges from a state, and multiple edges with the same letter can be emitted from a state.

**Example.** The following NFA recognizes the language of  $a + aa^*b + a^*b$ .



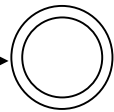
	$T$	$a$	$b$	$\epsilon$
start	0	$\{1, 2\}$	$\emptyset$	$\{1\}$
	1	$\{1\}$	$\{2\}$	$\emptyset$
final	2	$\emptyset$	$\emptyset$	$\emptyset$

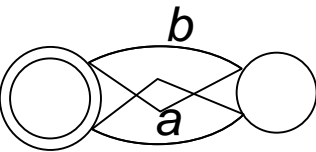
## Table representation of NFA

An NFA over  $A$  can be represented by a function  $T : \text{States} \times A \cup \{L\} \rightarrow \text{power}(\text{States})$ , where  $T(i, a)$  is the set of states reached from state  $i$  along the edge labeled  $a$ , and we mark the start and final states. The following figure shows the table for the preceding NFA.

# Examples

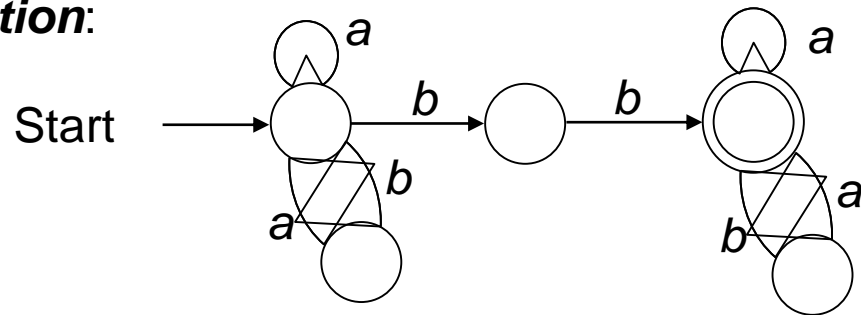
**Solutions:** (a): Start  $\longrightarrow$  

(b) Start  $\longrightarrow$    
:

(c): Start  $\longrightarrow$  

Find an NFA to recognize the language  $(a + ba)^*bb(a + ab)^*$ .

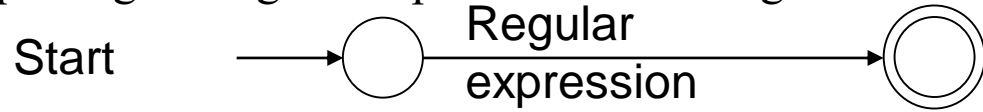
**A solution:**



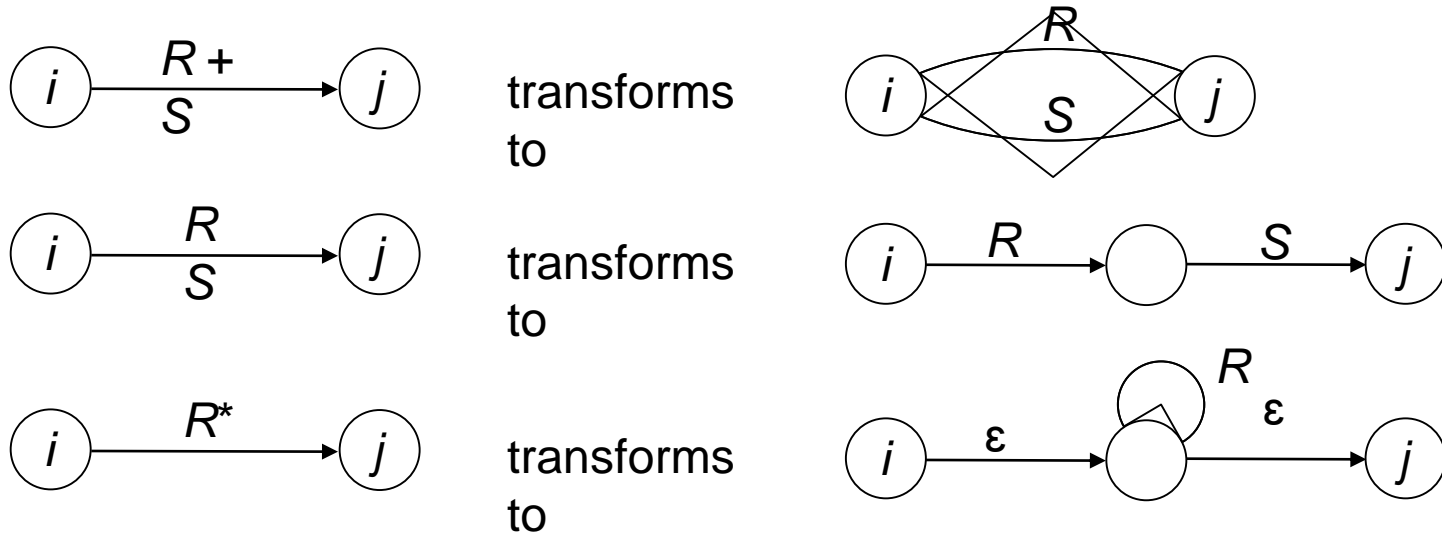
# Examples

Algorithm: *Transform a Regular Expression into a Finite Automaton*

Start by placing the regular expression on the edge between a start and final state:

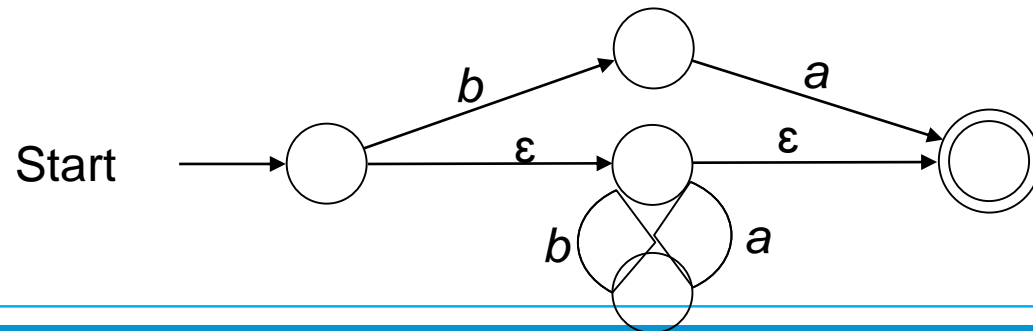


Apply the following rules to obtain a finite automaton after erasing any  $\emptyset$ -edges.



Quiz. Use the algorithm to construct a finite automaton for  $(ab)^* + ba$ .

**Answer:**



# Example of NFA using Python

```
from automata.fa.nfa import NFA
# NFA which matches strings beginning with 'a', ending with 'a', and
# containing
# no consecutive 'b's
nfa = NFA(
    states={'q0', 'q1', 'q2'},
    input_symbols={'a', 'b'},
    transitions={
        'q0': {'a': {'q1'}},
        # Use "" as the key name for empty string (lambda/epsilon)
        'q1': {'a': {'q1'}, '' : {'q2'}},
        'q2': {'b': {'q0'}}
    },
    initial_state='q0',
    final_states={'q1'}
)
```

```
nfa.read_input('aba')
ANSWER :{'q1', 'q2'}
```

```
nfa.read_input('abba')
ANSWER: ERROR
```

```
nfa.read_input_stepwise('aba')
```

```
if nfa.accepts_input('aba'):
    print('accepted')
else:
    print('rejected')
ANSWER: ACCEPTED
nfa.validate()
ANSWER: TRUE
```

# Sample Exercises - NFA

1. Write a automata code for the Language that accepts all end with 01
2. Write a automata code for  $L(M) = a + aa^*b + a^*b$ .
3. Write a automata code for Let  $\Sigma = \{0, 1\}$ .

Given NFAs for  $\{\}$ ,  $\{\epsilon\}$ ,  $\{(ab)^n \mid n \in \mathbb{N}\}$ , which has regular expression  $(ab)^*$ .

# Dependent type Programming Paradigm



# Introduction

## A constant problem:

- Writing a correct computer program is hard and proving that a program is correct is even harder
- Dependent Types allow us to write programs and know they are correct before running them.
- dependent types: you can specify types that can check the value of your variables at compile time

## Example:

Here is how you can declare a Vector that contains the values 1, 2, 3 :

```
val l1 = 1 :#: 2 :#: 3 :#: Vnil
```

This creates a variable l1 who's type signature specifies not only that it's a Vector that contains Ints, but also that it is a Vector of length 3. The compiler can use this information to catch errors. Let's use the vAdd method in Vector to perform a pairwise addition between two Vectors:

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
```

```
val l2 = 1 :#: 2 :#: 3 :#: VNil
```

```
val l3 = l1 vAdd l2
```

```
// Result: l3 = 2 :#: 4 :#: 6 :#: VNil
```

# Introduction

The example above works fine because the type system knows both Vectors have length 3. However, if we tried to vAdd two Vectors of different lengths, we'd get an error at compile time instead of having to wait until run time!

```
val l1 = 1 :# 2 :# 3 :# VNil
```

```
val l2 = 1 :# 2 :# VNil
```

```
val l3 = l1 vAdd l2
```

```
// Result: a *compile* error because you can't pairwise add vectors
```

```
// of different lengths!
```

Note:

You can express almost anything with dependent types. A factorial function which only accepts natural numbers, a login function which doesn't accept empty strings, a remove Last function which only accepts non-empty arrays. And all this is checked before you run the program.

# Introduction

A function has dependent type if the type of a function's result depends on the VALUE of its argument; this is not the same thing as a ParameterizedType. The second order lambda calculus possesses functions with dependent types.

## What does it mean to be “correct”?

Depends on the application domain, but could mean one or more of:

- Functionally correct (e.g. arithmetic operations on a CPU)
- Resource safe (e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . . )
- Secure (e.g. not allowing access to another user’s data)

## What is type?

- In programming, types are a means of classifying values
- Exp: values 94, "thing", and [1,2,3,4,5] are classified as an integer, a string, and a list of integers
- For a machine, types describe how bit patterns in memory are to be interpreted.
- For a compiler or interpreter, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a programmer, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

# Introduction

In computer science and logic, a dependent type is a type whose definition depends on a value.

It is an overlapping feature of type theory and type systems.

Used to encode logic's quantifiers like "for all" and "there exists".

Dependent types may help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.

# Quantifiers

A predicate becomes a proposition when we assign it fixed values. However, another way to make a predicate into a proposition is to quantify it. That is, the predicate is true (or false) for all possible values in the universe of discourse or for some value(s) in the universe of discourse. Such quantification can be done with two quantifiers: the universal quantifier and the existential quantifier.

**Universal:** Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing. The universal quantification of a predicate  $P(x)$  is the proposition “ $P(x)$  is true for all values of  $x$  in the universe of discourse” We use the notation  $\forall$  for Universal quantifier

$$\forall x P(x)$$

which can be read “for all  $x$ ”

## Example:

Let  $P(x)$  be the predicate “ $x$  must take a discrete mathematics course” and let  $Q(x)$  be the predicate “ $x$  is a computer science student”. The universe of discourse for both  $P(x)$  and  $Q(x)$  is all UNL students.

Express the statement “Every computer science student must take a discrete mathematics course”.

$$\forall x (Q(x) \rightarrow P(x))$$

Express the statement “Everybody must take a discrete mathematics course or be a computer science student”.

$$\forall x (Q(x) \vee P(x))$$

If  $x$  is a variable, then  $\forall x$  is read as:

**For all  $x$**

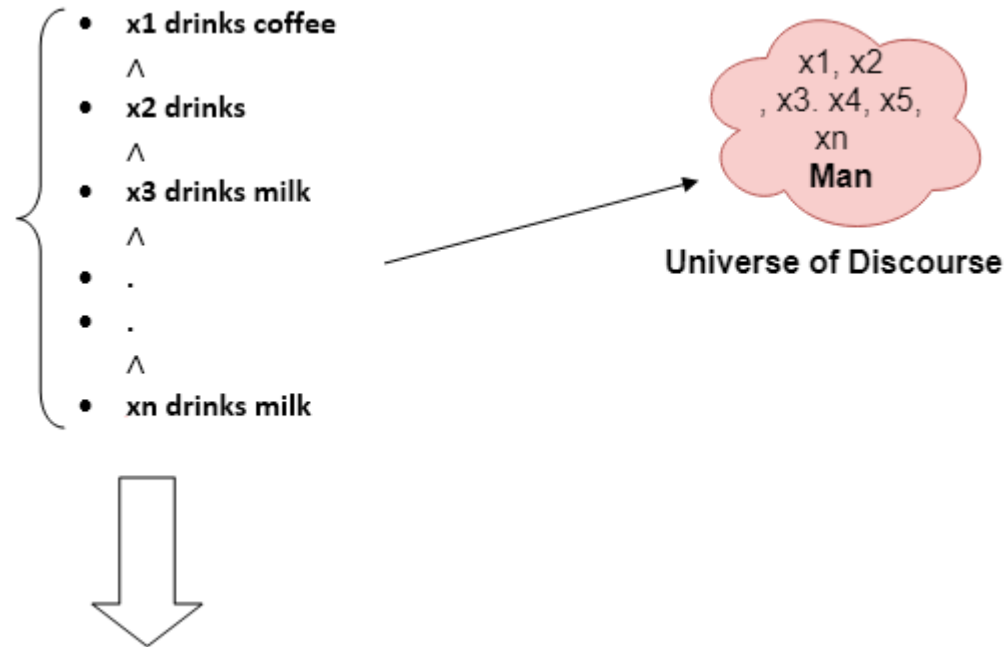
**For each  $x$**

**For every  $x$ .**

# Quantifiers

All man drink coffee.

Let a variable  $x$  which refers to a cat so all  $x$  can be represented in UOD as below:



So in shorthand notation, we can write it as :

$\forall x \text{ man}(x) \rightarrow \text{drink}(x, \text{coffee}).$

It will be read as: There are all  $x$  where  $x$  is a man who drink coffee.

# Existential Quantifiers

The existential quantification of a predicate  $P(x)$  is the proposition “There exists an  $x$  in the universe of discourse such that  $P(x)$  is true.” We use the notation

$$\exists x P(x)$$

which can be read “there exists an  $x$ ”

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator  $\exists$ , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

If  $x$  is a variable, then existential quantifier will be  $\exists x$  or  $\exists(x)$ . And it will be read as:

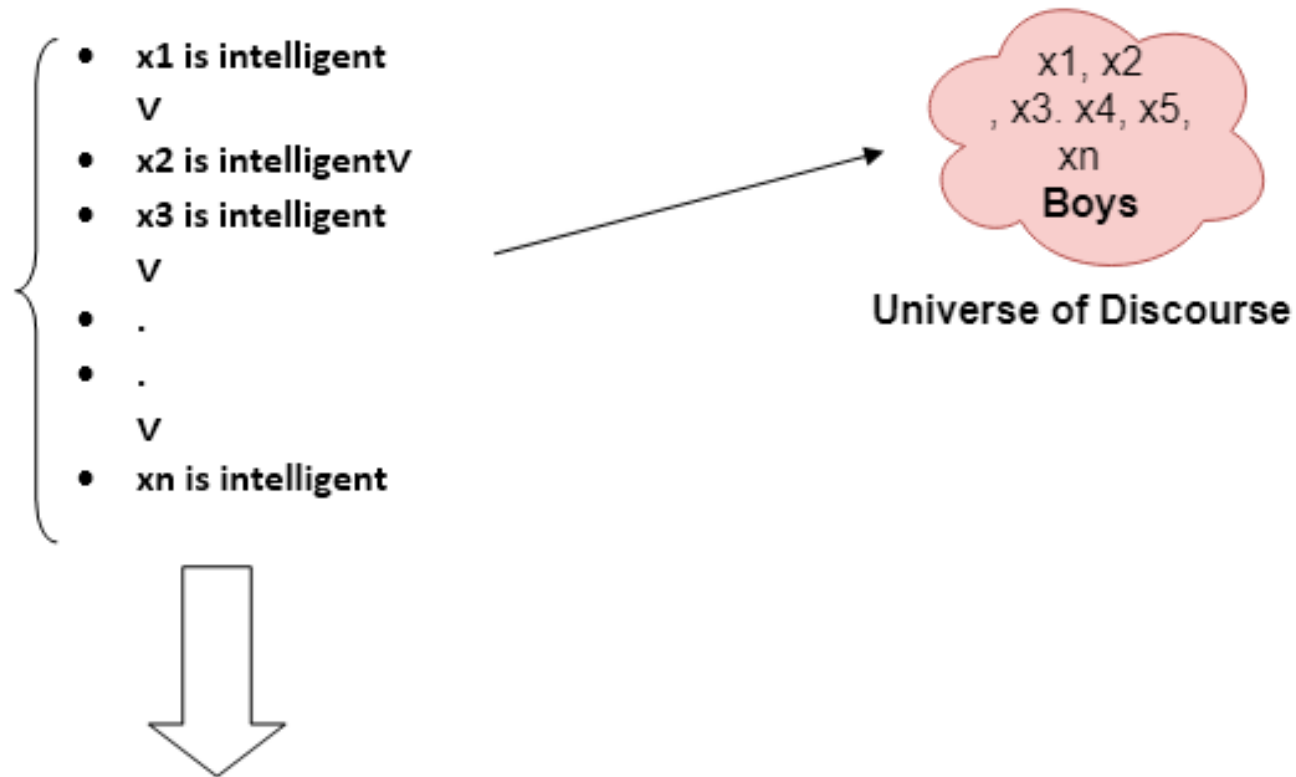
**There exists a 'x.'**

**For some 'x.'**

**For at least one 'x.'**

# Existential Quantifiers

Some boys are intelligent.



So in short-hand notation, we can write it as:

$\exists x: \text{boys}(x) \wedge \text{intelligent}(x)$

It will be read as: There are some  $x$  where  $x$  is a boy who is intelligent.



# Examples

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x).$$

2. Every man respects his parent.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

Since there is every man so will use  $\forall$ , and it will be represented as follows:

$$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent}).$$

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use  $\exists$ , and it will be represented as:

$$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket}).$$