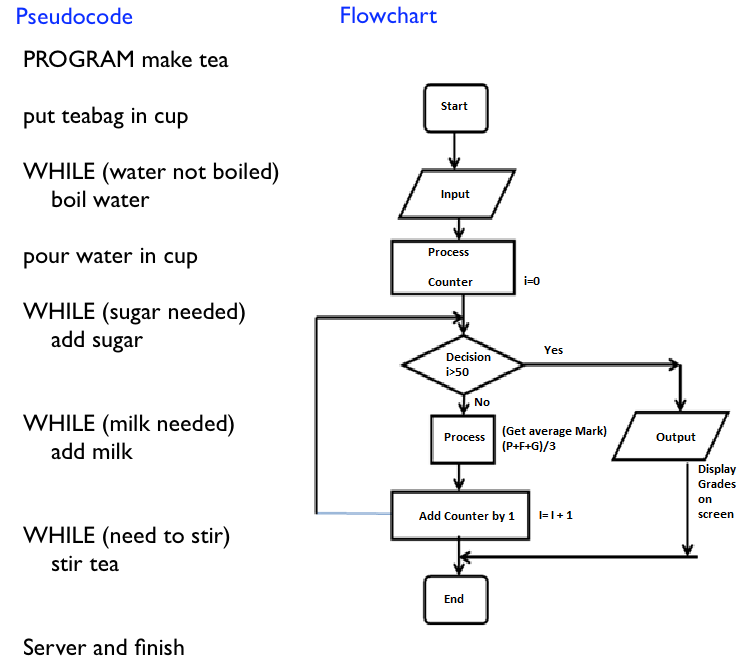
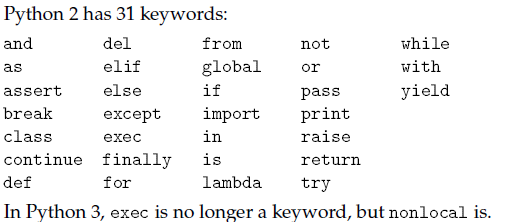
**FlowCharts & Psuedo code  
  
**

**Variable:**-Variable typeInt=3, Float=3.0, String=”Words” (‘w’, “w”, “””w”””, ‘’’w’’’)  
  
-Variable names (eg. Algebra\_1234)  
1)Variable names must start with letters or an underscore  
2)Variables are case sensitive  
3)The rest of the variable can consist of letters, numbers and underscores readable and case sensitive  
  
Avoid Key words: (cant be used for variable names)  
  
 **Operation:**Add: + , Substract: - , Multiply: \* , Divide: / , Parenthesis: () , Power: \*\* , Remainder: % ,  
Whole number in the division: //--(avoid sometimes produce funny5//0.1)

Int point operation: 1/3 = 0  
floating point : 1.0/3 = 0.3333string: “p” +”o”= ”po”  
 “spam”\*3=”spamspamspam”  
 **Decimal places**  
 z=1.666666,y=”Something %d” %z= “Something 1”  
 y=”Something %f” %z= “Something 1.66666666”  
 y=”Something %.3f”%z=”Something 1.667”  
 y=” Something %6.3f”%z=”Something 1.667”#place holders the spaces  
 y= "Time:%d:%d:%d"%(self.hrs,self.mins,self.seconds)  
**Comparison Operator:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Not Equal | Greater | Smaller | Greater or = | Smaller or = | Equal |
| x != y | x>y | x<y | x>=y | x<=y | x == y |

**Int:  
float:**y=round(y,2) #changed to two decimal place  
y=3.0-5/9.0<< make sure all float 5/9 will give you an error

**System**:  
import sys  
smallest =-sys.float\_info.max  
maximum= sys.maxint  
**String:**   
len(“words”)=5  
“s” in “string” <<<true  
“sentence”.count(“s”)<<<1  
.split(“:”)  
.replace(“hate”,”like”)  
s[::-]  
.isalnum()  
  
x=”sentence”  
for c in x:  
 print c  
  
x[0]=”s” Cant strings are inmutable cannot be changed.  
''.join changed form list to string

**Boolean: True / False**empty list becomes false bool(“”), bool(0+0j), bool(None) **Local and global variables.**

**Keywords:**print (“\n”) = newline  
 (“\t”) = Tab  
in used to check if a value is within another value (eg.“ham” in hamsandwhich …. returns True)

**Lists:**x=[“ham”,4.22,5]  
x.append(5) #Add stuff into list [“ham”,4.22,5,5]  
x.insert(0,3.1) #Insert stuff into the list [3.1,“ham”,4.22,5]  
x.pop(1) #remove stuff from the list [“ham” ,5]  
len(x) #number of items in list len([“ham”,4.22,5]) returns 3  
list(“ham”) #converting string into list [“h”,”a”,”m”]  
in #check if item is in list “ham” in x… returns True, “h” in x… returns False  
x[0],x[1] #first item, second item “ham”,4.22  
x[1:3]#slicing the list [“4.22”,5] 2nd item to 4th item not including 4thitem   
x[1:3:2] # slicing increase increment [“4.22”] increased increment to 2  
x[::-1] #reverse the list [1,2,3]>>>[3,2,1]  
x= [“c”]+x #adding a list in front   
x\*2 #repeat 2 times [“ham”,4.22,5,“ham”,4.22,5]  
list.index(“ham”) returns the first position of the index.

list(str) #changes to a list “1234”>>[1,2,3,4]

If list is empty if a==[]: print “list is empty”

A list is a iterable object

List\_1.sort() #arrange form lowest to highest

**Copying list**  
When list1=list2, then change list2[0]=”blue” changes list1 as well (automatically updates)  
list2 = list1[:] #works for shallow list, deep list will change when list1 changes  
  
from copy import deepcopy # or import copy  
list2=deepcopy(list1) #works for deep list

**Tuples:** #similar to list, use less memory but not adjustable,   
x=(“ham”,4.22,5)  
x[1] calling a tuple same as list

X[1]=3 inmutable cant be changed

**Dictionaries: aka hastable and maps**exDict={‘Jack’:[15,’blonde’],’Bob’:[22,’brown’]}  
print(exDict[“Jack”][1])

a={“test”:1}  
b=a #changes with a  
c=copy.deepcopy(a) #unaffected by a similar to list

mydict = {"cat":12, "dog":6, "elephant":23}  
mydict["mouse"] = mydict["cat"] + mydict["dog"]  
print(mydict["mouse"]) # ans =6+12=18  
  
dict1.keys() shows the keys in a list.

**Functions** (Def: defining a function)  
•Type conversion functions

|  |  |
| --- | --- |
| x=int(“32”) y=float(“32.0”) z=str(32) | x=int(“32.0”) y=float(“32.0”) z=str(32) |

•defining your own functions  
def celsius\_to\_fahrenheit(C): #Formal parameters  
 F=C\*9/5.0+32.0  
 return F

print celsius\_to\_fahrenheit (12) #Actual parameters

•defining your multivalued function  
def celsius\_to\_fahrenheit(C): #Formal parameters  
 F=C\*9/5.0+32.0,c  
 return F

x,y=celsius\_to\_fahrenheit (12) #Actual parameters

•Common functions  
range(4) = [ 0,1,2,3]

**Class** (attribute references and instantiation)  
**Interactive mode and script mode**

class Point:

""" Point class for representing and manipulating x,y coordinates. """

def \_\_init\_\_(self,x,y): #initialization

""" Create a new point at the origin """

self.x = x

self.y = y

def \_\_str\_\_(self):#to return a string a long with the values of intrest

return "Square of height and width %s." %(self.L)

p = Point() # Instantiate an object of type Point

q = Point() # and make a second point

print p # this will call upon the \_\_str\_\_ and return the statement

#you cam call for other function's class by typing:

#eg

#L2=Vector2D(finallinex,finalliney)

#lengths.append(L2.length())

#this is done in another class

**Loops:**# can only handle integer conditions  
•**While**counter=1  
while counter<=60:•**For**  
for x in **range** (0, 3):  
for s in “string”: (print 6 times because string has 6 alphabet

for number in [5,4,3,2,1,0]—repeat 6 times

for letter in **'Python':** # First Example

print 'Current Letter :', letter

fruits = ['banana', 'apple', 'mango']

for fruit in **fruits**: # Second Example

print 'Current fruit :', fruit

#Results

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Sets([2,3,”top”)={2,3,”top}  
 .add .remove .discard .clear()  
.union() .intersection()  
orders doesn’t matter.  
  
  
ramdom number generator  
ramdom ramdint(1,100)

File   
f=open('data.txt','r')  
l=f.readlines()  
for x in range(0,len(l)):

l[x]=l[x].strip()

eg.

def getMRT(f):

dict\_MRT={}

l=f.readlines()

#print l

for x in range(0,len(l)):

l[x]=l[x].strip()

j=l[x].split(",")

j\_name=j[0]

j.pop(0)

dict\_MRT[j\_name]=j

#print dict\_MRT

return dict\_MRT

def distance(d,s):

ls\_s=s.split(",")

#print ls\_s

ls\_vals=d.values()

for x in range(0,len(ls\_vals)):

counter=0

for y in range(0,len(ls\_vals[x])):

if ls\_vals[x][y]==ls\_s[0] or ls\_vals[x][y]==ls\_s[1]:

counter=y-counter

return counter

State Funcitons

|  |  |  |  |
| --- | --- | --- | --- |
| Current State | Input | Next State | Output |
| 0 | 0 | 0 | “At 0 , going to 0” |
| 0 | 1 | 1 | “At 0 , going to 1” |
| 1 | 0 | 0 | “At 1 , going to 0” |
| 1 | 1 | 1 | “At 1, going to 1” |
| 0 | # | 0 | “invalid input, no action” |
| 1 | # | 1 | “Invalid input, no action” |

|  |  |  |
| --- | --- | --- |
| Current State | Input | |
|  | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 0 | 1 |

2^n min number of state transition for n states machine

Create a file to record the states and run the states   
-Form libdw import sm  
sm.SM  
Object.start()

Digital World

# Firebase

URL : Where the Data is.

Token : Allows the python code to login into the Data in URL.

# KIVY

SimpleKivy(App):

def build (self)

build arrange and display the elements of GUI in an order.

build calls for the widgets and places them into the window created by SimpleKivy.run().

build recognises all the widgets and properties defined under the def build function and places them into the window accordingly.

return layout so that the build function can be placed into the window.

Does not take in other attributes other than self.

Callables can be made to link the widgets to the function via events such as onclick, ontouchdown.

# Switch from one screen to another

A class that changes a screen.

Define the individual screens using \_\_init\_\_ method.

For each screen, we can define whatever the current state of the screen is and also a button that allows for switching in-between the screens.

ms=MenuScreen(name='menu')

st=SettingsScreen(name='settings')

Give your 2 different screens a name. The names will allow you to call the screens in your above callables.

class SwitchScreenApp(App): 🡨 Takes in the App Functions.

class SettingsScreen(Screen): 🡨 Takes in the Screen functions

class MenueScreen(Screen):

App = The whole kivy app. Contains general functions like run and build.

Screen = Just the display screen. Gives you access to manager.transitions, manager.current.

def build(self):

sm=ScreenManager() 🡨 instead of using layout, use ScreenManager. Calls a moving screen.

ms=MenuScreen(name='menu') 🡨 Uses BoxLayout, etc as it is a screen.

st=SettingsScreen(name='settings')

sm.add\_widget(ms)

sm.add\_widget(st)

sm.current='menu'

return sm

bind = a link between a widget, an event and a call back.

self.increase\_01 = Button(text='Increase', font\_size = 20)

self.increase\_01.bind(on\_press=self.i) 🡨 Binds the function self.i to the button and will activate on press.

# PID

Proportional, Integral, Differential.

P = K(error)

I =

D =

# Deep Copy

Import copy

self. World=copy.deepcopy(world) 🡨 anti aliasing

does not change the global world variable.

**App.run()**  
runs the **event loop**, which waits for the user to do something and responds accordingly. It is an infinite loop; it runs until the user closes the window, or presses Control-C, or does something that causes the program to quit.

**Widget**

Widget are the elements that make up a GUI; they include:

**Button:** A widget, containing text or an image, that performs an action when pressed.

**Canvas:** A region that can display lines, rectangles, circles and other shapes.

**Entry:** A region where users can type text.

**Scrollbar:** A widget that controls the visible part of another widget.

**Frame:** A container, often invisible, that contains other widgets.

**Buttons (Buttons & Callbacks)**

The button that appears in the Frame is a graphical representation of this object; you can control the button by invoking methods on it.  
  
Buttons has parameters that control the appearance and function of the button. These

parameters are called **options**. Instead of providing values for all options, you can use

keyword arguments, like text='Press me.', to specify only the options you need and use

the default values for the rest.

When you add a widget to the Frame, it gets “shrink-wrapped;” that is, the Frame shrinks

to the size of the Button. If you add more widgets, the Frame grows to accommodate them.  
  
By default, KIVY stacks the widgets top-to-bottom and centers them.  
  
The option that controls the behavior of a button is command. The value of command is a

function that gets executed when the button is pressed.

**Callbacks (Buttons & Callbacks)**  
The value of the command option is a function object, which is known as a **callback** because

after you call add\_widget(button) to create the button, the flow of execution “calls back” when the user

presses the button.

This kind of flow is characteristic of **event-driven programming**. User actions, like button

presses and key strokes, are called **events**. In event-driven programming, the flow of

execution is determined by user actions rather than by the programmer.

**Menus (Menus & Callables)**

A Menubutton is a widget that looks like a button, but when pressed it pops up a menu.

After the user selects an item, the menu disappears.  
  
**Callables (Menus & Callables)**  
The command option for Menus is a Callable object.

The Callable object stores a reference to the function and the arguments as attributes. Later,

when the user clicks on a menu item, the callback calls the function and passes the stored

arguments.  
  
**Binding**  
A **binding** is an association between a widget, an event and a callback:  
For eg. when an event (like a button press) happens on a widget, the callback is invoked.  
  
**19.10 Glossary**

**GUI:** A graphical user interface.

**widget:** One of the elements that makes up a GUI, including buttons, menus, text entry

fields, etc.

**option:** A value that controls the appearance or function of a widget.

**keyword argument:** An argument that indicates the parameter name as part of the function

call.

**callback:** A function associated with a widget that is called when the user performs an

action.

**bound method:** A method associated with a particular instance.

**event-driven programming:** A style of programming in which the flow of execution is

determined by user actions.

**event:** A user action, like a mouse click or key press, that causes a GUI to respond.

**event loop:** An infinite loop that waits for user actions and responds.

**item:** A graphical element on a Canvas widget.

**bounding box:** A rectangle that encloses a set of items, usually specified by two opposing

corners.

**pack:** To arrange and display the elements of a GUI.

**geometry manager:** A system for packing widgets.

**binding:** An association between a widget, an event, and an event handler. The event

handler is called when the event occurs in the widget.

**Inheritance**  
The language feature most often associated with object-oriented programming is **inheritance**.

Inheritance is the ability to define a new class that is a modified version of an

existing class.

It is called “inheritance” because the new class inherits the methods of the existing class.

Extending this metaphor, the existing class is called the **parent** and the new class is called

the **child**.