

Python Conditional Statements

Conditional Statements are features of a **programming** language, which perform different computations or actions depending on whether the given **condition** evaluates to true or false. Conditional statements in python are of 3 types

- If statement
- If else statement
- If elif statement
- Nested if else

1. **If Statement** : if Statement is used to run a statement conditionally i.e. if given condition is true then only the statement given in if block will be executed.

```
if <condition>:  
    <if statement block >
```

For example consider the code given below

```
if (percentage > 33):  
    print ("Pass")
```

Explanation : In the above code if value of percentage is above 33 then only the message "Pass" will be printed.

2. **If else Statement** In the case of if else statement If given condition is true then the statement given in if block will be executed otherwise(else) the statements written in else block will be executed

```
if <condition>:  
    <if statement block >  
else:  
    <else statement block>
```

For example consider the code given below

```
if (percentage > 33):  
    print ("Pass")  
else:  
    print("Fail")
```

Explanation : In the above code if value of percentage is above 33 then only the message "Pass" will be printed otherwise it will print "Fail"

3. If elif Statement if elif is used for execution OF STATEMENTS based on several alternatives. Here we use one or more elif (short form of *else if*) clauses. Python evaluates each **condition** in turn and executes the statements corresponding to the **first if** that is true. If none of the expressions are true, and an else clause will be executed

Syntax:-

```
if <condition>:
```

```
    <statement(s)>
```

```
elif <condition>:
```

```
    <statement(s)>
```

```
.
```

```
.
```

```
else:
```

```
    <statement(s)>
```

Explanation : In the above code if value of percentage is above 33 then only the message **“Pass”** will be printed otherwise it will print **“Fail”**

Example:

```
If (percentage >90):
    Print(“Outstanding”)
elif (percentage >80):
    print (“Excellent”)
elif (percentage >70):
    print (“VeryGood”)
elif (percentage >60):
    print (“Good”)
elif (percentage >33):
    print (“Pass”)
else
    print(“Fail”)
```

Explanation : In the above code
if value of percentage is above 90 then it will print **“Outstanding”**
if value of percentage is above 80 then it will print **“Excellent”**
if value of percentage is above 70 then it will print **“Very Good”**
if value of percentage is above 60 then it will print **“Good”**
if value of percentage is above 33 then it will print **“Pass”**
if no condition is true then it will print **“Fail”**

In above code only 1 condition can be true at a time if no condition is true then else statement will be executed

4. Nested If else Statement A nested if is an if statement that is the target of another if statement. Nested if statement means an if statement within another if statement.

Syntax:-

```
if (<condition1>):
```

```
    statement(s)
```

```
    if (<condition2>):
```

```
        statement(s)
```

```
    else
```

```
else:
```

```
    if (<condition3>):
```

```
        statement(s)
```

```
    else
```

```
        Statement(s)
```

```
# Example
```

```
if color ="red":
```

```
    if item="fruit":
```

```
        print(" It is an Apple")
```

```
    else :
```

```
        print("It may be Tomato or Rose")
```

```
else:
```

```
    if color="Yellow":
```

```
        print("It is a Banana")
```

```
    else
```

```
        print("It may be corn or Marigold ")
```

OR

```
if color ="red":
```

```
    if item="fruit":
```

```
        print(" It is an Apple")
```

```
    else :
```

```
        print("It may be Tomato or Rose")
```

```
elif color="Yellow":
```

```
    print("It is a Banana")
```

```
else
```

```
    print("It may be corn or Marigold ")
```

Data types in Python

Every value in Python has a data type. Since everything is an object in Python programming, data types are actually classes, and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below

DATA TYPES IN PYTHON

- ❖ Python Numbers
- ❖ Python List
- ❖ Python Tuple
- ❖ Python Strings
- ❖ Python Set
- ❖ Python Dictionary

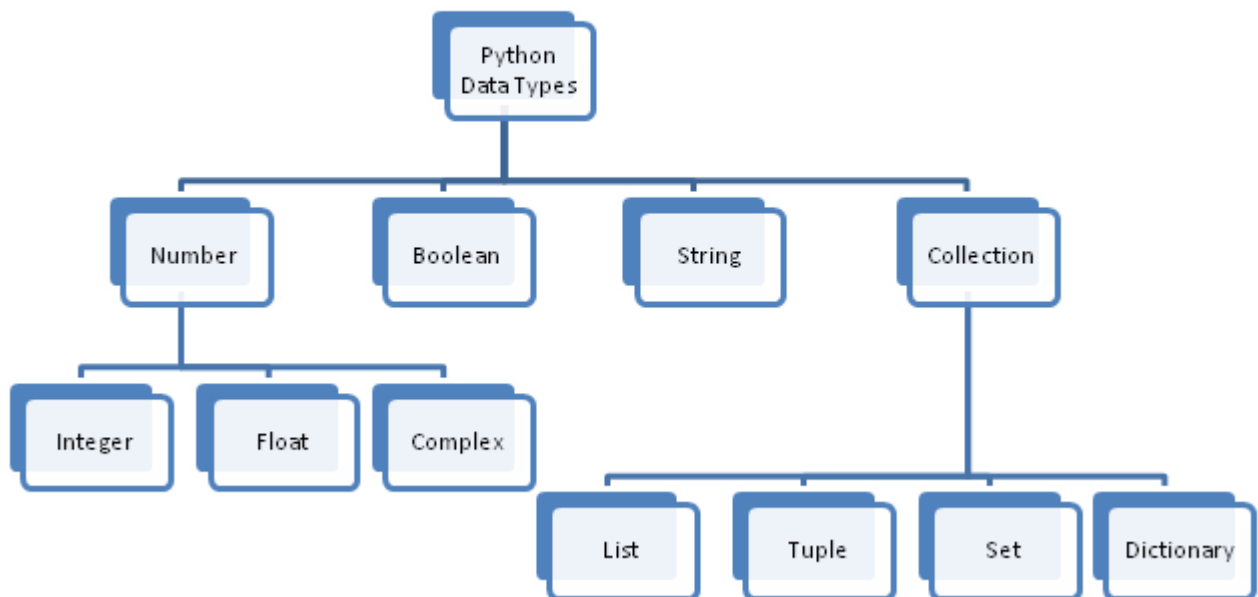


Chart : Python Data Types

1. Python Data Type – Numeric

Python numeric data type is used to hold numeric values like;

Data Type	Use
Int	holds signed integers of non-limited length.
Long	holds long integers(exists in Python 2.x,
Float	holds floating precision numbers and it's
complex	holds complex numbers.

2. Python Data Type – String

String is a **sequence of characters**. Python supports Unicode characters. Generally strings are represented by either single or double quotes.

```
>>> s1 = "This is a string"
>>> s2= '''a multiline String'''
```

Single Line String	"hello world"																																																																																																																																																									
Multi Line String	"""Gwalior Madhya Pradesh"""																																																																																																																																																									
Raw String	r"raw \n string" [Used when we want to have a string that contains backslash and don't want it to be treated as an escape character.]																																																																																																																																																									
Character	"C" [Single letter]																																																																																																																																																									
Unicode string	<p>u"\u0938\u0902\u0917\u0940\u0924\u093E" will print 'संगीता'</p> <table border="1"> <thead> <tr> <th></th> <th>0</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> <th>7</th> <th>8</th> <th>9</th> <th>A</th> <th>B</th> <th>C</th> <th>D</th> <th>E</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>U+090x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>अ</td> <td>आ</td> <td>इ</td> <td>ई</td> <td>उ</td> <td>ऊ</td> <td>ऋ</td> <td>ॠ</td> <td>एँ</td> <td>ऐ</td> <td>ए</td> </tr> <tr> <td>U+091x</td> <td>ऐ</td> <td>ऑ</td> <td>ओ</td> <td>ओ</td> <td>औ</td> <td>क</td> <td>ख</td> <td>ग</td> <td>घ</td> <td>ङ</td> <td>च</td> <td>छ</td> <td>ज</td> <td>झ</td> <td>ञ</td> <td>ट</td> </tr> <tr> <td>U+092x</td> <td>ठ</td> <td>ड</td> <td>ढ</td> <td>ण</td> <td>त</td> <td>थ</td> <td>द</td> <td>ध</td> <td>न</td> <td>न</td> <td>प</td> <td>फ</td> <td>ब</td> <td>भ</td> <td>म</td> <td>य</td> </tr> <tr> <td>U+093x</td> <td>र</td> <td>ॠ</td> <td>ल</td> <td>ळ</td> <td>ळ</td> <td>व</td> <td>श</td> <td>ष</td> <td>स</td> <td>ह</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+094x</td> <td>ी</td> <td>ु</td> <td>ू</td> <td>ृ</td> <td>ृ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+095x</td> <td>ॐ</td> <td>'</td> <td>_</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+096x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+097x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> </tbody> </table>		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	U+090x	ॐ	ॐ	ॐ	ॐ	ॐ	अ	आ	इ	ई	उ	ऊ	ऋ	ॠ	एँ	ऐ	ए	U+091x	ऐ	ऑ	ओ	ओ	औ	क	ख	ग	घ	ङ	च	छ	ज	झ	ञ	ट	U+092x	ठ	ड	ढ	ण	त	थ	द	ध	न	न	प	फ	ब	भ	म	य	U+093x	र	ॠ	ल	ळ	ळ	व	श	ष	स	ह	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+094x	ी	ु	ू	ृ	ृ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+095x	ॐ	'	_	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+096x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+097x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F																																																																																																																																										
U+090x	ॐ	ॐ	ॐ	ॐ	ॐ	अ	आ	इ	ई	उ	ऊ	ऋ	ॠ	एँ	ऐ	ए																																																																																																																																										
U+091x	ऐ	ऑ	ओ	ओ	औ	क	ख	ग	घ	ङ	च	छ	ज	झ	ञ	ट																																																																																																																																										
U+092x	ठ	ड	ढ	ण	त	थ	द	ध	न	न	प	फ	ब	भ	म	य																																																																																																																																										
U+093x	र	ॠ	ल	ळ	ळ	व	श	ष	स	ह	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+094x	ी	ु	ू	ृ	ृ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+095x	ॐ	'	_	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+096x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+097x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										

3. Python Data Type – List

List is a versatile data type exclusive in Python. In a sense it is same as array in C/C++. But interesting thing about list in Python is it can simultaneously hold different type of data.

Formally list is a ordered sequence of some data written using square brackets ([]) and commas(,)

```
my_list = []  
# empty list
```

```
my_list = [1, 2, 3]  
# list of integers
```

```
my_list = [1, "Hello", 3.4]  
# list with mixed data types
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list  
my_list = ["mouse", [8, 4, 6], ['a']]
```

4. Python Tuple

Tuple is an ordered sequences of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically. It is defined within parentheses () where items are separated by commas

```
>>> t = (50,'Learning is fun', 1+3j, 45.67) # Can store mixed data types
```

Advantages of Tuple over List

- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

5. Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}
```

printing set variable

```
print("a = ", a)
```

data type of variable a

```
print(type(a))    # <class 'set'>
```

6. Python Dictionary

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within Curly braces {} with each item being a pair in the form **key: value**.

Key and value can be of any type.

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> d1={'Name': 'Manasvi', 'Class': 9}
>>> d1
{'Name': 'Manasvi', 'Class': 9}
>>> d1['Name']
'Manasvi'
>>> d1['Class']
9
>>> |
```



Dictionary Values can be printed using key eg d1['Name']

Python Control Statements

In any programming language a program may execute sequentially, selectively or iteratively. Every programming language provides constructs to support Sequence, Selection and Iteration. In Python all these construct can broadly categorized in 2 categories.

A. Conditional Control Construct
(Selection, Iteration)

B. Un- Conditional Control Construct
(pass, break, continue, exit(), quit())

Python have following types of control statements

1. **Selection** (branching) Statement
2. **Iteration** (looping) Statement
3. **Jumping** (break / continue)Statement

**Conditional Control
Statements**

**Un Conditional Control
Statements**

Python Selection Statements

Python have following types of selection statements

1. if statement
2. if else statement
3. Ladder if else statement (if-elif-else)
4. Nested if statement

Python If statements

This construct of python program consist of one if condition with one block of statements. When condition becomes true then executes the block given below it.

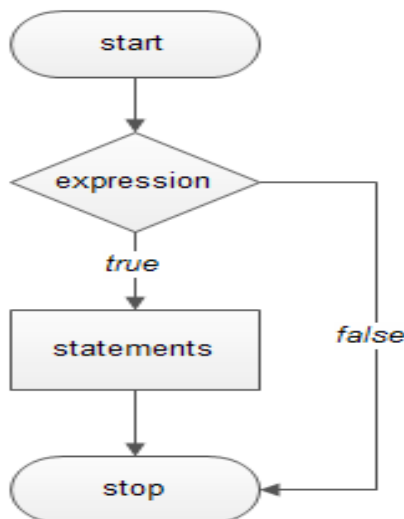
Syntax:



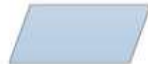
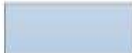

if (condition):

.....
.....
.....

Flow Chart: it is a graphical representation of steps an algorithm to solve a problem.

Flowchart



Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Example:

```
Age=int(input("Enter Age: "))  
If ( age>=18):  
    Print("You are eligible for vote")
```

```
If(age<0):  
    Print("You entered Negative Number")
```

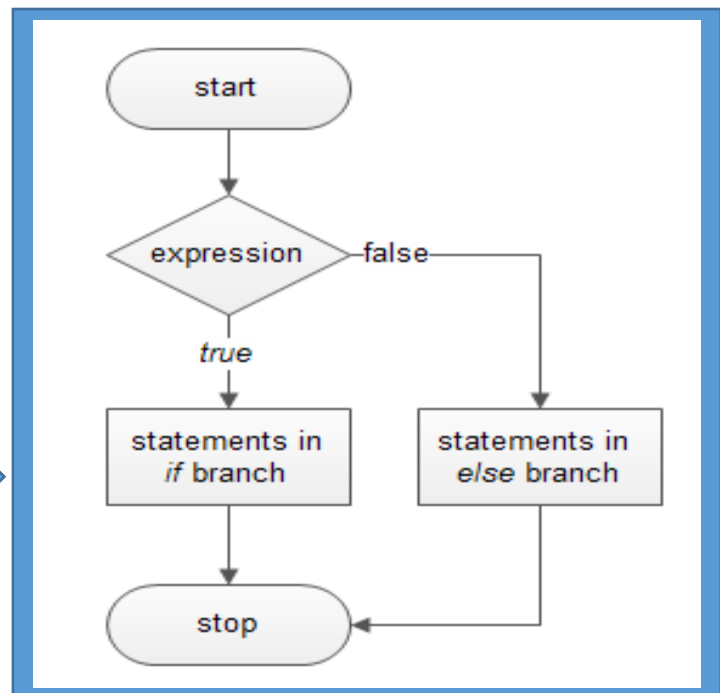
Python if - else statements

This construct of python program consist of **one if condition with two blocks**. When condition becomes true then executes the block given below it. If condition evaluates result as false, it will executes the block given below else.

Syntax:

```
if ( condition):  
    .....  
else:  
    .....
```

Flowchart →



Example-1:

```
Age=int(input("Enter Age: "))
```

```
if ( age>=18):
```

```
    print("You are eligible for vote")
```

```
else:
```

```
    print("You are not eligible for vote")
```

Example-2:

```
N=int(input("Enter Number: "))
```

```
if(n%2==0):
```

```
    print(N," is Even Number")
```

```
Else:
```

```
    print(N," is Odd Number")
```

Python Ladder if else statements (if-elif-else)

This construct of python program consist of **more than one if condition**. When first condition evaluates result as true then executes the block given below it. If condition evaluates result as false, it transfer the control at else part to test another condition. So, it is **multi-decision making** construct.

Syntax:

if (condition-1):

.....
.....

elif (condition-2):

.....
.....

elif (condition-3):

.....
.....

else:

.....
.....

Example:

```
num=int(input("Enter Number: "))
```

```
If ( num>=0):
```

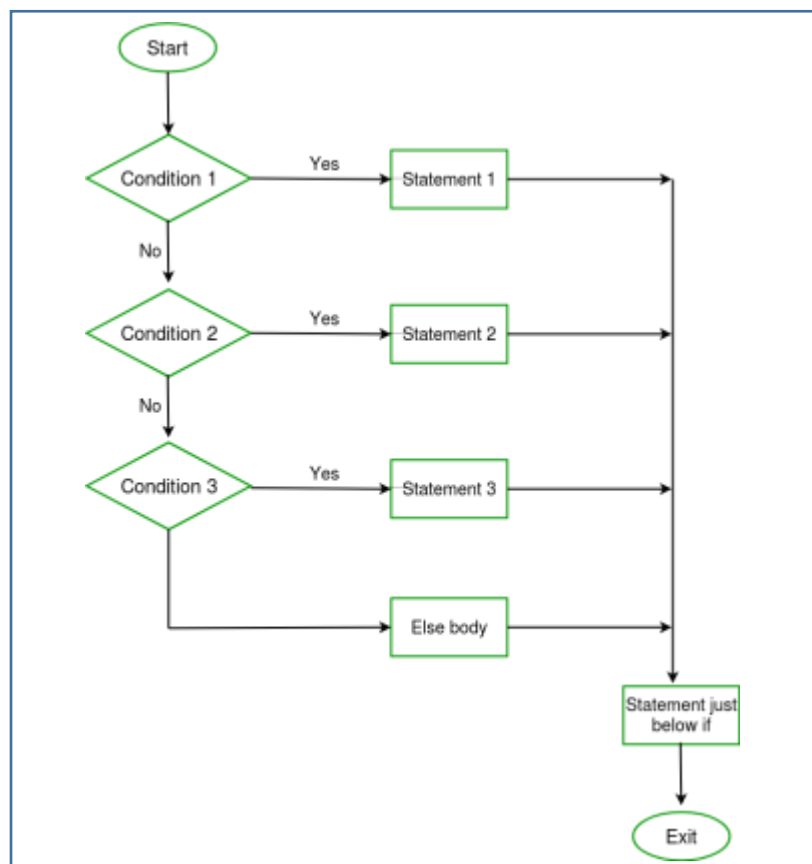
```
    Print("You entered positive number")
```

```
elif ( num<0):
```

```
    Print("You entered Negative number")
```

```
else:
```

```
    Print("You entered Zero ")
```



Python Nested if statements

It is the construct where one if condition take part inside of other if condition. This construct consist of **more than one if condition**. Block executes when condition becomes false and next condition evaluates when first condition became true.

So, it is also **multi-decision making** construct.

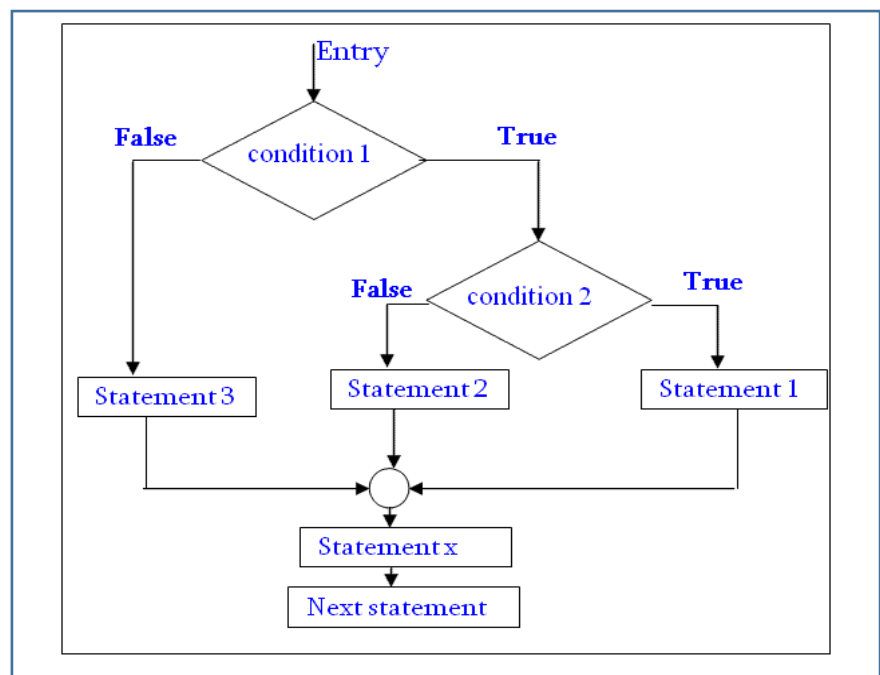
Syntax:

```
if ( condition-1):  
    if (condition-2):  
        .....  
        .....  
    else:  
        .....  
        .....  
else:  
    .....  
    .....
```

Example:

```
num=int(input("Enter Number: "))  
If ( num<=0):  
    if ( num<0):  
        Print("You entered Negative number")  
    else:  
        Print("You entered Zero ")  
else:  
    Print("You entered Positive number")
```

FlowChart



Program: find largest number out of given three numbers

```
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
z=int(input("Enter Third Number: "))
if(x>y and x>z):
    largest=x
elif(y>x and y>z):
    largest=y
elif(z>x and z>y):
    largest=z
print("Largest Value in %d, %d and %d is: %d"%(x,y,z,largest))
```

Program: calculate simple interest

Formula: principle x (rate/100) x time

```
p=float(input("Enter principle amount: "))
r=float(input("Enter rate of interest: "))
t=int(input("Enter time in months: "))
si=p*r*t/100
print("Simple Interest=",si)
```

Program: calculate EMI

Input the following to arrive at your Equal Monthly Installment -EMI:

1. Loan Amount: Input the desired loan amount that you wish to avail.
2. Loan Tenure (In Years): Input the desired loan term for which you wish to avail the loan.
3. Interest Rate (% P.A.): Input interest rate.
4. $EMI = [P * R * (1 + R)^N] / [(1 + R)^N - 1]$

```
P=int(input("Enter loan amount: "))
```

```

YR=float(input("Enter rate of interest P.A. : "))
T=int(input("Enter tenure(Installments) in years: "))
MR=YR/(12*100) # Monthly Rate
EMI=(P*MR*(1+MR)**T)/(((1+MR)**T)-1)
print("Principle Amount: ",P)
print("Rate of Interest(Yearly): ",YR)
print("No. of Installments: ",T)
print("EMI Amount: ",EMI)

```

Program: Sorting of three number. (Ascending and Descending)

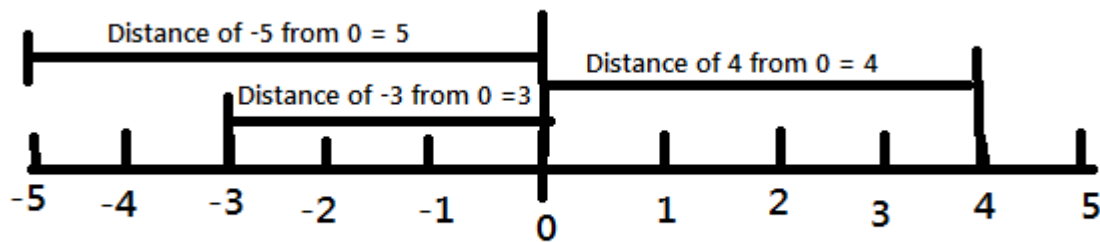
```

x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
z=int(input("Enter Third Number: "))
min=max=mid=None
if(x>=y and x>=z):
    if(y>=z):
        min,mid,max=z,y,x
    else:
        min,mid,max=y,z,x
elif(y>=x and y>=z):
    if(x>=z):
        min,mid,max=z,x,y
    else:
        min,mid,max=x,z,y
elif(z>=x and z>=y):
    if(x>=y):
        min,mid,max=y,x,z
    else:
        min,mid,max=x,y,z
print("Numbers in Ascending Order: ",min,mid,max)
print("Numbers in Descending Order: ",max,mid,min)

```

Program: Absolute Value

Absolute value of a given number is always measured as positive number. This number is the distance of given number from the 0(Zero). The input value may be integer, float or complex number in Python. The absolute value of given number may be integer or float.



Concept of Absolute Value

(i). Absolute Value of -5 is 5 (ii) Absolute Value of -3 is 3 (iii) Absolute Value of 4 is 4

```
n=float(input("Enter a number to find absolute value: "))
print("Absolute Value using abs(): ",abs(n))
if(n-int(n)>=0 or n-int(n)<=0): # This code is used to identify that number is float or int type.
    pass
else:
    n=int(n)
if(n<0):
    print("Absolute Value= ",n*-1)
else:
    print("Absolute Value= ",n)
```


Program: Calculate the Total selling price after levying the GST (Goods and Service Tax) as CGST and SGST on sale.

CGST (Central Govt. GST), SGST (State Govt. GST)

Sale amount	CGST Rate	SGST Rate
0-50000	5%	5%
Above 50000	18%	18%

```
amt=float(input("Enter total Sale Amount: "))
if(amt<=50000):
    rate=5
else:
    rate=18
cgst=sgst=amt*rate/100
tot_amt=amt+cgst+sgst
print("Amount of Sale: ",amt)
print("GST rate of Sale: ",rate)
print("CGST of Sale: ",cgst)
print("SGST of Sale: ",sgst)
print("Total Payable Amount of Sale: ",tot_amt)
```

Thanks

Python Iteration Statements

The iteration (Looping) constructs mean to execute the block of statements again and again depending upon the result of condition. This repetition of statements continues till condition meets True result. As soon as condition meets false result, the iteration stops.

Python supports following types of iteration statements

1. while
2. for

Four Essential parts of Looping:

- i. Initialization of control variable
- ii. Condition testing with control variable
- iii. Body of loop Construct
- iv. Increment / decrement in control variable

Python while loop

The while loop is conditional construct that executes a block of statements again and again till given condition remains true. Whenever condition meets result false then loop will terminate.

Syntax:

Initialization of control variable

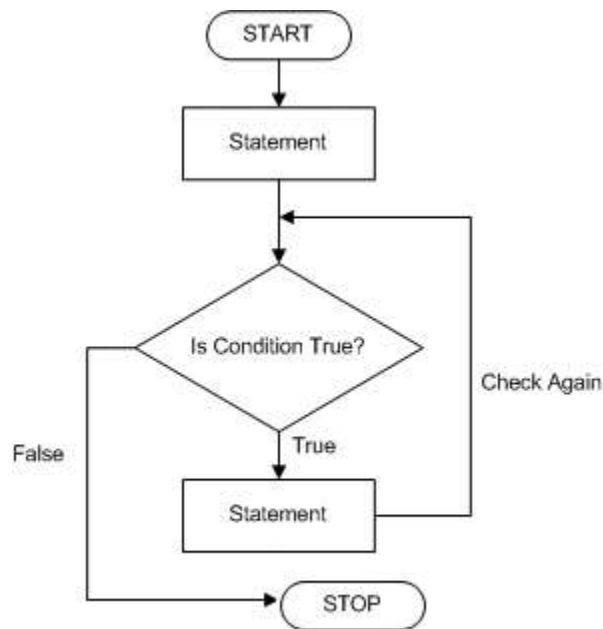
while (condition):

.....

 Updation in control variable

.....

Flowchart



Example: print 1 to 10 numbers

```
num=1    # initialization
while(num<=10):    # condition testing
    print(num, end=" ")
    num + = 1    # Increment
```

} Body of loop

Example: Sum of 1 to 10 numbers.

```
num=1
sum=0
while(num<=10):
    sum + = num
    num + = 1
print("The Sum of 1- 10 numbers: ",sum)
```

Example: Enter per day sale amount and find average sale for a week.

Python range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. The common format of range() is as given below:

range (**start value**, **stop value**, **step value**)

Where all 3 parameters are of integer type

Start value is Lower Limit

Stop value is Upper Limit

Step value is Increment / Decrement

Start and Step Parameters are optional default value will be as Start=0 and Step=1

Note: The Lower Limit is included but Upper Limit is not included in result.

Example

range(5) => sequence of 0,1,2,3,4

range(2,5) => sequence of 2,3,4

range(1,10,2) => sequence of 1,3,5,7,9

range(5,0,-1) => sequence of 5,4,3,2,1

range(0,-5) => sequence of [] blank list (default Step is +1)

range(0,-5,-1) => sequence of 0, -1, -2, -3, -4

range(-5,0,1) => sequence of -5, -4, -3, -2, -1

range(-5,1,1) => sequence of -5, -4, -3, -2, -1, 0

L=list(range(1,20,2))

Print(L) Output: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Python for loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a string etc.) With for loop we can execute a set of statements, and for loop can also execute once for each element in a list, tuple, set etc.

Example: print 1-10 numbers

```
for num in range(1,11,1):  
    print(num, end=" ")
```

Output: 1 2 3 4 5 6 7 8 9 10

Example: print 10-1 numbers

```
for num in range(10,0,-1):  
    print(num, end=" ")
```

Output: 10 9 8 7 6 5 4 3 2 1

Print each element in a fruit list:

```
fruits = ["mango", "apple", "grapes", "cherry"]
```

```
for x in fruits:
```

```
    print(x)
```

output:

```
mango  
apple  
grapes  
cherry
```

```
for x in "TIGER":
```

```
    print(x)
```

output:

```
T  
I  
G  
E  
R
```

Membership Operators:

The "in" and "not in" are membership operators. These operators check either given value is available in sequence or not. The "in" operator returns Boolean True result if value exist in sequence otherwise returns Boolean False.

The "not in" operator also returns Boolean True / False result but it works opposite to "in" operator.

else in for Loop

The `else` keyword in for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(4):  
    print(x, end=" ")  
else:  
    print("\nFinally finished!")
```

output: 0 1 2 3
 Finally finished!

Nested Loops

A nested loop is a loop inside another loop.

```
city = ["Jaipur", "Delhi", "Mumbai"]  
fruits = ["apple", "mango", "cherry"]  
for x in city:  
    for y in fruits:  
        print(x, ":", y)
```

output:

```
Jaipur : apple  
Jaipur : mango  
Jaipur : cherry  
Delhi : apple  
Delhi : mango  
Delhi : cherry  
Mumbai : apple  
Mumbai : mango  
Mumbai : cherry
```

Un- Conditional Control Construct

(pass, break, continue, exit(), quit())

pass Statement (Empty Statement)

The pass statement do nothing, but it used to complete the syntax of programming concept. Pass is useful in the situation where user does not requires any action but syntax requires a statement. The Python compiler encounters pass statement then it do nothing but transfer the control in flow of execution.

```
a=int(input('Enter first Number: '))
b=int(input('Enter Second Number: '))
if(b==0):
    pass
else:
    print('a/b=',a/b)
```

```
for x in [0, 1, 2]:
    pass
```

Jumping Statements

break Statement

The jump- break statement enables to skip over a part of code that used in loop even if the loop condition remains true. It terminates to that loop in which it lies. The execution continues from the statement which find out of loop terminated by break.

```
n=1
while(n<=5):
    print("n=",n)
    k=1
    while(k<=5):
        if(k==3):
            break
        print("k=",k, end=" ")
        k+=1
    n+=1
    print()
```

```
Output:
n= 1
k= 1 k= 2
n= 2
k= 1 k= 2
n= 3
k= 1 k= 2
n= 4
k= 1 k= 2
n= 5
k= 1 k= 2
```

Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

output: apple

Continue Statement

Continue statement is also a jump statement. With the help of continue statement, some of statements in loop, skipped over and starts the next iteration. It forcefully stop the current iteration and transfer the flow of control at the loop controlling condition.

```
i = 0
while i <=10:
    i+=1
    if (i%2==1):
        continue
    print(i, end=" ")
```

output: 2 4 6 8 10

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

output:

apple
cherry

Thanks

What is Python Module

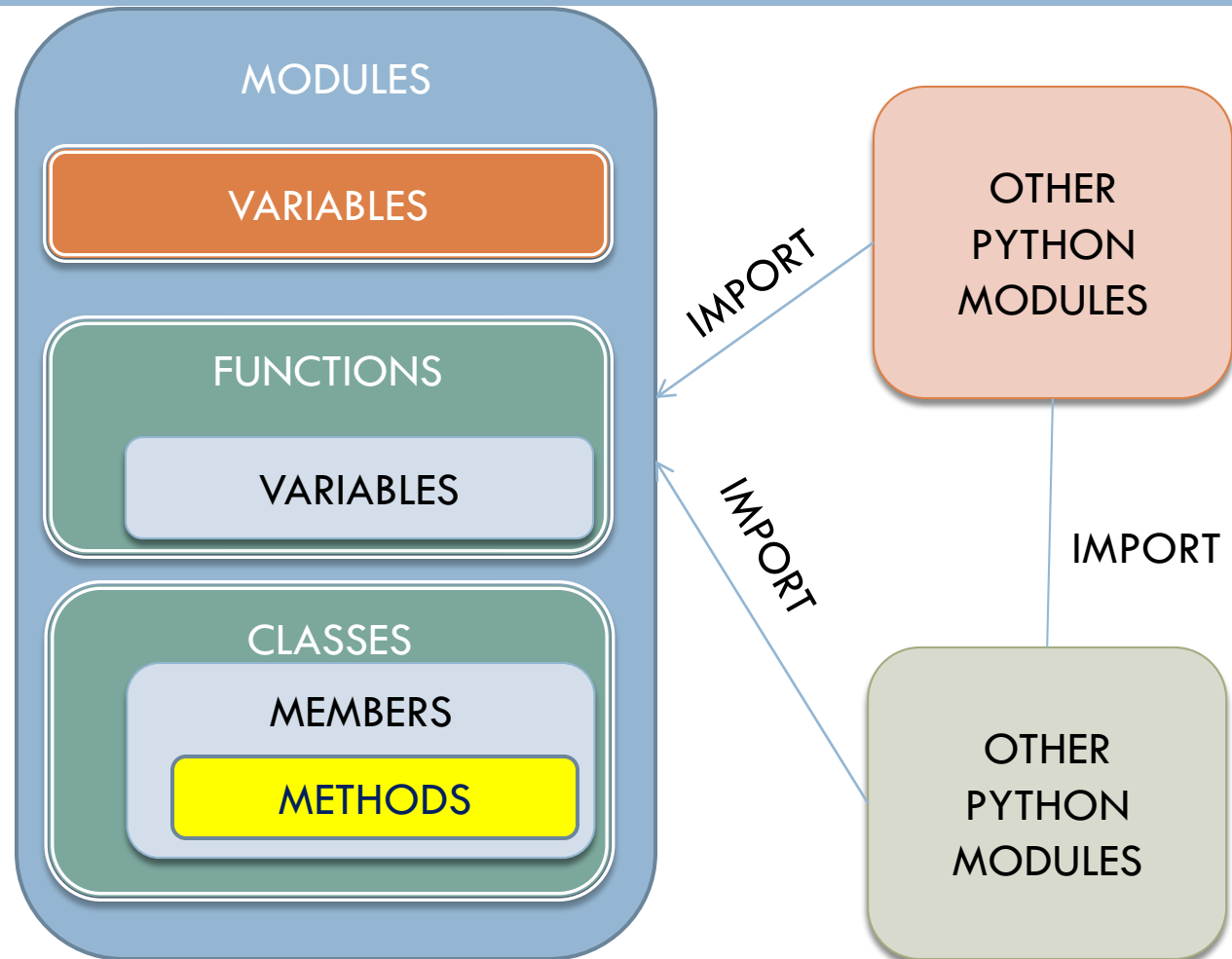
- A Module is a file containing Python definitions (docstrings) , functions, variables, classes and statements.
- Act of partitioning a program into individual components(modules) is called modularity. A module is a separate unit in itself.
 - ▣ It reduces its complexity to some degree
 - ▣ It creates numbers of well-defined, documented boundaries within program.
 - ▣ Its contents can be reused in other program, without having to rewrite or recreate them.

Structure of Python module

- A python module is simply a normal python file(.py) and contains functions, constants and other elements.
- Python module may contains following objects:

docstring	Triple quoted comments. Useful for documentation purpose
Variables and constants	For storing values
Classes	To create blueprint of any object
Objects	Object is an instance of class. It represent class in real world
Statements	Instruction
Functions	Group of statements

Composition/Structure of python module



Importing Python modules

- To import entire module
 - **import <module name>**
 - **Example:** `import math`

- To import specific function/object from module:
 - **from <module_name> import <function_name>**
 - **Example:** `from math import sqrt`

- **import *** : can be used to import all names from module into current calling module

Accessing function/constant of imported module

- To use function/constant/variable of imported module we have to specify module name and function name separated by dot(.). This format is known as dot notation.
 - `<module_name>.<function_name>`
 - **Example:** `print(math.sqrt(25))`
- However if only particular function is imported using **from** then module name before function name is not required. We will see examples with next slides.

Types of Modules

- There are various in-built module in python, we will discuss few of them
 - Math module
 - Random module
 - Statistical module

Math module

- This module provides various function to perform arithmetic operations.
- Example of functions in math modules are:

sqrt	ceil	floor	pow
fabs	sin	cos	tan

- Example of variables in math modules are:
 - pi
 - e

Math module functions

- **sqrt(x)** : this function returns the square root of number(x).

```
>>> import math
>>> print(math.sqrt(49))
7.0
```

module name is required before function name here

- **pow(x,y)** : this function returns the $(x)^y$

```
>>> from math import pow
>>> print(pow(2,6))
64.0
```

module name is not required before function name here

- **ceil(x)** : this function return the x rounded to next integer.

```
>>> import math
>>> print(math.ceil(45.25))
46
```

Math module functions

- **floor(x)** : this function returns the x rounded to previous integer.

```
>>> import math
>>> print(math.floor(5.9))
5
```

- **fabs(x)** : this function returns absolute value of float x. absolute value means number without any sign

```
>>> import math
>>> a=-8.5
>>> print(math.fabs(a))
8.5
```

- **sin (x)** : it return sine of x (measured in radian)

```
>>> import math
>>> print(math.sin(30))
-0.9880316240928618
```

```
>>> import math
>>> print(math.sin(30*math.pi/180))
0.49999999999999994
```

```
>>> import math
>>> print(round(math.sin(30*math.pi/180),1))
0.5
```

Math module functions

- **cos(x)** : it return cosine of x (measured in radian)

```
>>> import math
>>> print(math.cos(90))
-0.4480736161291701
```

- **tan(x)** : it return tangent of x (measured in radian)

```
>>> import math
>>> print(math.tan(45))
1.6197751905438615
```

- **pi** : return the constant value of pi (22/7)

```
>>> print(math.pi)
3.141592653589793
```

- **e** : return the constant value of constant e

```
>>> print(math.e)
2.718281828459045
```

Using Random Module

- Python has a module namely random that provides random – number generators. Random number means any number generated within the given range.
- To generate random number in Python we have to import random module
- 2 most common method to generate random number in python are :
 - random() function
 - randint(a,b) function

random() function

- It is floating point random number generator between 0.0 to 1.0. here lower limit is inclusive where as upper limit is less than 1.0.
- $0 \leq N < 1$
- Examples:

```
>>> import random
>>> a = random.random()
>>> print(a)
0.0888880146536
>>> |
```

Output is less than 1

random() function

- To generate random number between given range of values using random(), the following format should be used:
 - ▣ $\text{Lower_range} + \text{random()} * (\text{upper_range} - \text{lower_range})$
 - ▣ For example to generate number between 10 to 50:
 - $10 + \text{random()} * (40)$

randint() function

- Another way to generate random number is randint() function, but it generate integer numbers.
- Both the given range values are inclusive i.e. if we generate random number as :
 - ▣ randint(20,70)
 - In above example random number between 20 to 70 will be taken. (including 20 and 70 also)

```
>>> import random
>>> a = random.randint(10,20)
>>> print(a)
18
>>> █
```

E
X
A
M
P
L
E

```
import random
count=3
ans='y'
win=False
print("Guess what number computer generated between 20-30")
print("Total 3 chances are there ")
print("-----")
while ans=='y':
    num1 = random.randint(20,30)
    print("Change Remaining :",count)
    guess = int(input("Enter your answer :"))
    if num1 == guess:
        print("Congratulation! you guessed it right")
        win=True
    else:
        print("Wrong!")
        count-=1
        if count==0:
            print("Oops! You lost all your chances ")
            print("Number was :",num1)
    if win==True or count==0:
        ans=input("Play Again?")
        if ans=='y':
            count=3
            win=False
```



```
Guess what number computer generated between 20-30  
Total 3 chances are there
```

```
-----  
Change Remaining : 3  
Enter your answer :21  
Wrong!  
Change Remaining : 2  
Enter your answer :22  
Wrong!  
Change Remaining : 1  
Enter your answer :23  
Wrong!  
Oops! You lost all your chances  
Number was : 25  
Play Again?y
```

```
Change Remaining : 3  
Enter your answer :28  
Wrong!  
Change Remaining : 2  
Enter your answer :27  
Wrong!  
Change Remaining : 1  
Enter your answer :29  
Congratulation! you guessed it right  
Play Again?n
```

Just a Minute...

- Give the following python code, which is repeated four times. What could be the possible set of output(s) out of four sets (ddd is any combination of digits)

```
import random
```

```
print(15 + random.random()*5)
```

a)	b)	c)	d)
17.ddd	15.ddd	14.ddd	15.ddd
19.ddd	17.ddd	16.ddd	15.ddd
20.ddd	19.ddd	18.ddd	15.ddd
15.ddd	18.ddd	20.ddd	15.ddd

Just a Minute...

- What could be the minimum possible and maximum possible numbers by following code

```
import random
print(random.randint(3,10)-3)
```
- In a school fest, three randomly chosen students out of 100 students (having roll number 1 -100) have to present the bouquet to the guests. Help the school authorities choose three students randomly

Just a Minute...

What possible outputs(s) are expected to be displayed on screen at the time of execution of the program from the following code? Also specify the minimum values that can be assigned to each of the variables BEGIN and LAST.

```
import random

VALUES=[10,20,30,40,50,60,70,80];
BEGIN=random.randint(1,3)
LAST =random.randint(BEGIN,4)

for I in range(BEGIN, LAST+1):
    print VALUES[I], "-",
```

(i) 30 - 40 - 50 -

(ii) 10 - 20 - 30 - 40 -

(iii) 30 - 40 - 50 - 60 -

(iv) 30 - 40 - 50 - 60 - 70 -

Just a Minute...

Look at the following Python code and find the possible output(s) from the options (i) to (iv) following it. Also, write the maximum and the minimum values that can be assigned to the variable PICKER.

Note:

- Assume all the required header files are already being included in the code.
- The function randint() generates an integer between 1 to n

```
import random
```

```
PICKER=1+random.randint(0,2)
```

```
COLOR=["BLUE","PINK","GREEN","RED"]
```

```
for l in range(1,PICKER+1):
```

```
    for j in range(l+1):
```

```
        print(COLOR[j],end="")
```

```
    print()
```

(i)	BLUEPINK	(ii)	PINKGREEN	(iii)	BLUE	(iv)	BLUEPINK
	BLUEPINKGREEN		PINKGREENRED		BLUEPINK		BLUEPINKGREEN
					BLUEPINKGREEN		BLUEPINKGREENRED

What are the possible outcome(s) executed from the following code? Also specify the maximum and minimum values that can be assigned to variable PICK

```
import random
PICK = random.randint(0, 3)
CITY = ["DELHI", "MUMBAI", "CHENNAI", "KOLKATA"]
for i in CITY:
    for j in range(1, PICK) :
        print(i, end=" ")
    print()
```

1)
DELHIDELHI
MUMBAIMUMBAI
CHENNAICHENNAI
KOLKATAKOLKATA

2)
DELHI
DELHIMUMBAI
DELHIMUMBAICHENNAI

3)
DELHI
MUMBAI
CHENNAI
KOKLATA

4)
DELHI
DELHIMUMBAI
KOLKATAKOLKATAKOLKATA

randrange() function

- This function is also used to generate random number within given range.
- Syntax
 - ▣ `randrange(start,stop,step)`

```
import random
n1 = random.randrange(5,15)
n2 = random.randrange(5,15)
n3 = random.randrange(5,15)
n4 = random.randrange(5,15)
print(n1,n2,n3,n4)
```

11 8 5 12

It will generate random number between 5 to 14

random output between 5 to 14, may vary

randrange() function

```
import random
for i in range(20):
    n1 = random.randrange(1, 30, 2)
    print(n1, end='\t')
```

It will generate random number between 1 to 29 with stepping of 2 i.e. it will generate number with gap of 2 i.e. 1,3,5,7 and so on

```
25    11    15    9    3    7    19    13    17    7
27    11    27    5    21   7    17    9    25    7
```


Mathematics Game for Kids

```
import random
operators = ['+', '*', '-']
error = 0
score = 0
print("##### WELCOME TO SIMPLE CALCULATION GAME #####")
print("Rule : +4 for correct answer, -2 for wrong answer ")

for i in range(5):
    print("***50")
    n1 = random.randrange(1,100)
    n2 = random.randrange(1,100)
    i = random.randrange(0,3)
    op = operators[i]
    result = 0
    if op=='+' :
        result = n1 + n2
    elif op=='-' :
        if n1<n2:
            n1,n2=n2,n1
        result = n1 - n2
    elif op=='*' :
        result = n1 * n2
    print(n1,op,n2,'=')
    ask = int(input())
    if ask == result:
        score+=4
    else:
        score-=2

print("***50")
print("## YOU SCORED : ",score, " ##")
```

Mathematics Game for Kids

```
import random
operators = ['+', '*', '-']
error = 0
score = 0
print("##### WELCOME TO SIMPLE CALCULATION GAME #####")
print("Rule : +4 for correct answer, -2 for wrong answer ")

for i in range(5):
    print("*"*50)
    n1 = random.randrange(1,100)
    n2 = random.randrange(1,100)
    i = random.randrange(0,3)
    op = operators[i]
    result = 0
    if op=='+':
        result = n1 + n2
    elif op=='-':
        if n1<n2:
            n1,n2=n2,n1
        result = n1 - n2
    elif op=='*':
        result = n1 * n2
    print(n1,op,n2,'=')
    ask = int(input())
    if ask == result:
        score+=4
    else:
        score-=2

print("*"*50)
print("### YOU SCORED : ",score, " ##")
```

```
##### WELCOME TO SIMPLE CALCULATION GAME #####
Rule : +4 for correct answer, -2 for wrong answer
*****
93 * 50 =
11
*****
29 + 29 =
58
*****
80 + 22 =
102
*****
61 - 25 =
36
*****
43 - 43 =
0
*****
## YOU SCORED : 14 ##
```

Statistical Module

- This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.
- We will deal with 3 basic function under this module
 - Mean
 - Median
 - mode

Mean

- The mean is the average of all numbers and is sometimes called the arithmetic mean.

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90,100]
>>> mymean = statistics.mean(mynum)
>>> print(mymean)
55
```

55, is the average of all numbers in the list

Median

- The median is the middle number in a group of numbers.

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90]
>>> mymedian = statistics.median(mynum)
>>> print(mymedian)
50
```

With odd number of elements it will simply return the middle position value

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90,100]
>>> mymedian = statistics.median(mynum)
>>> print(mymedian)
55.0
```

With even number of elements, it will return the average of value at mid + mid-1 i.e. $(50+60)/2 = 55.0$

Mode

- The mode is the number that occurs most often within a set of numbers i.e. most common data in list.

```
>>> import statistics
>>> mynum = [10,20,10,40,20,10,70,80,90]
>>> mymode = statistics.mode(mynum)
>>> print(mymode)
10
```

Here, 10 occurs most in the list.

Python Classes and Objects

A Basic Introduction

Topics

- Objects and Classes
- Abstraction
- Encapsulation
- Messages

What are objects

- An object is a datatype that stores data, but ALSO has operations defined to act on the data. It knows stuff and can do stuff.
- Generally represent:
 - tangible entities (e.g., student, airline ticket, etc.)
 - intangible entities (e.g., data stream)
- Interactions between objects define the system operation (through message passing)

What are Objects

- A Circle drawn on the screen:
- Has **attributes** (knows stuff):
 - radius, center, color
- Has **methods** (can do stuff):
 - move
 - change color

Design of Circle object

- A `Circle` object:
 - `center`, which remembers the center point of the circle,
 - `radius`, which stores the length of the circle's radius.
 - `color`, which stores the color
- The `draw` method examines the `center` and `radius` to decide which pixels in a window should be colored.
- The `move` method sets the center to another location, and redraws the circle

Design of Circle

- All objects are said to be an *instance* of some *class*. The class of an object determines which attributes the object will have.
- A class is a description of what its instances will know and do.

Circle: classes and objects

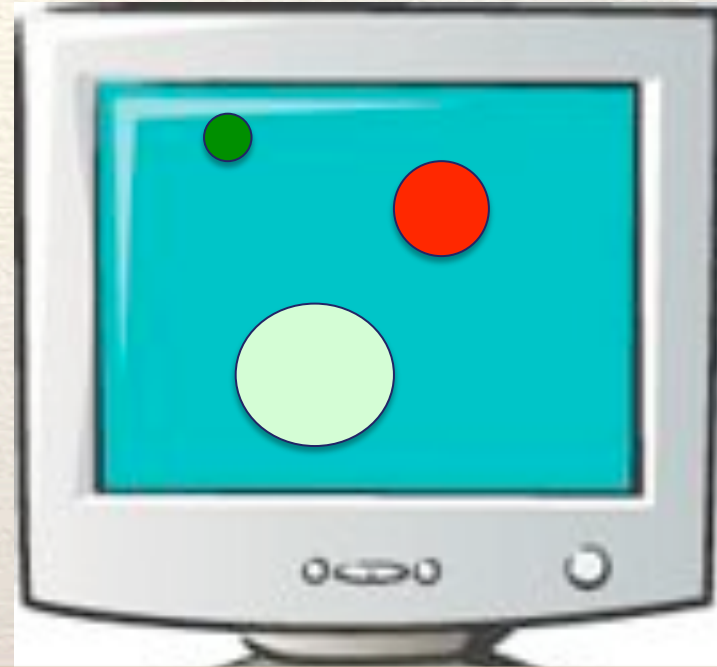
Classes are blueprints or directions on how to create an object

Circle Class

Attributes:
-location, radius,color

Methods:
- draw, move

Objects are instantiations of the class (attributes are set)



3 circle objects are shown (each has different attribute values)

Circle Class

```
class Circle(object):
```

Beginning of the class definition

```
    def __init__(self, center, radius):  
        self.center = center  
        self.radius = radius
```

The constructor. This is called when someone creates a new Circle, these assignments create attributes.

```
    def draw(self, canvas):  
        rad = self.radius  
        x1 = self.center[0]-rad  
        y1 = self.center[1]-rad  
        x2 = self.center[0]+rad  
        y2 = self.center[1]+rad  
        canvas.create_oval(x1, y1, x2, y2, fill='green')
```

A method that uses attributes to draw the circle

```
    def move(self, x, y):  
        self.center = [x, y]
```

A method that sets the center to a new location and then redraws it

objects/CircleModule.py

Constructors

- The object's constructor method is named `__init__`
- The primary duty of the constructor is to set the **state** of the object's attributes (instance variables)
- Constructors may have default parameters
- Calling an object's constructor (via the class name) is a signal to the interpreter to create (instantiate) a new object of the data type of the class
 - `myCircle = Circle([10,30], 20)` # Never pass "self", it's automatic

Creating a Circle

```
myCircle = Circle([10,30], 20)
```

- This statement creates a new `Circle` object and stores a reference to it in the variable `myCircle`.
- The parameters to the constructor are used to initialize some of the instance variables (`center` and `radius`) inside `myCircle`.

Creating a Circle

```
myCircle = Circle([10,30], 20)
```

- Once the object has been created, it can be manipulated by calling on its methods:

```
myCircle.draw(canvas)
```

```
myCircle.move(x, y)
```

Objects and Classes

- `myCircle = Circle([10,30], 20)`
- `myOtherCircle = Circle([4,60], 10)`
- `myCircle` and `myOtherCircle` are objects or instances of the Class `Circle`
- The circle class defines what a circle knows (attributes) and what it does (methods)... but to have a circle, you need to construct an object from that class definition
- Similar to a “list”. Python defines what a list is, and can do (slicing, indexing, `length(...)`, etc... but until you create one, you don't really have one

Using the Circle

- `from CircleModule import *`

```
myCircle = Circle([10,30], 20)
```

```
print
```

```
    "CENTER :"+str(myCircle.center)
```

```
>>> CENTER : (10, 30)
```

To get an instance variable from an object, use: `<<object>>.variable`

What happens if the instance variable doesn't exist?

Using Instance Variables

```
myCircle = Circle([10,30], 20)
print "CENTER :"+str(circle.carl)
>>> AttributeError: Circle
      instance has no attribute
      'carl'
```

Using Instance Variables

```
myCircle.bob = 234
```

What happens if you set an instance variable that doesn't exist?

Think: What happens if you assign ANY variable in python that doesn't exist?

```
john = 234
```

Python automatically creates a new variable if it doesn't exist. For instance variables this works the same... if you assign an instance variable that doesn't exist, Python just creates it...

Bad practice though... create all instance variables in the constructor!

Summary: Using instance variables

- Creating new instance variables just means assigning them a value:
 - `self.bob = 234` # In constructor
- Using instance variables is done through dot notation:
 - `val = myCircle.bob` # Outside the class definition
 - `val = self.bob` # Inside class methods

Attributes / Instance Variables

- Attributes represent the characteristics of a class. When an object is instantiated and the values are assigned to attributes, they are then referred to as instance variables.
- The values of the instance variables define the **state** of the individual object
- They are referred to as instance variables because the values assigned to an individual object (instance of a class) are unique to that particular class
- Attributes may be public or private (although due to their specific implementation, they are not truly private in Python)
- If the attributes are private, they serve to enforce the concept of **information hiding**

Using methods in Objects

- Methods are created just like a function, but inside a class:

```
class Circle:
```

```
    def myFunction(self, p1, p2):
```

```
        << something >>>
```

```
    def function2(self, input1='55'):
```

```
        <<something>>
```

- To use methods, call them using dot notation:

```
myCircle.myFunction(actualP1, actualP2)
```

Note: self is automatically passed in to all methods... you never pass it in directly!

Messages

- Process by which system components interact:
 - send data to another object
 - request data from another object
 - request object to perform some behavior
- Implemented as methods (not called functions).
 - Functions are processes that are object independent
 - Methods are dependent on the state of the object

Message Passing

- When calling a method in another class, OO uses the term “message passing” you are passing messages from one class to another
- Don't be confused... this is really just a new name for calling a method or a function

What is 'self'

- Self is a reference to the current instance. Self lets you access all the instance variables for the specific instance you're working with.
 - `myCircle.myFunction(actualP1, actualP2)`
- is like calling:
 - `Circle.myFunction(myCircle, actualP1, actualP2)`
- "self" really gets the value of "myCircle".. but it happens automatically!

Do this

Not this

Why use classes at all?

- Classes and objects are more like the real world. They minimize the semantic gap by modeling the real world more closely
- The semantic gap is the difference between the real world and the representation in a computer.
- - Do you care how your TV works?
 - No... you are a user of the TV, the TV has operations and they work. You don't care how.

Why use classes at all?

- Classes and objects allow you to define an interface to some object (it's operations) and then use them without know the internals.
- Defining classes helps modularize your program into multiple objects that work together, that each have a defined purpose

Encapsulation

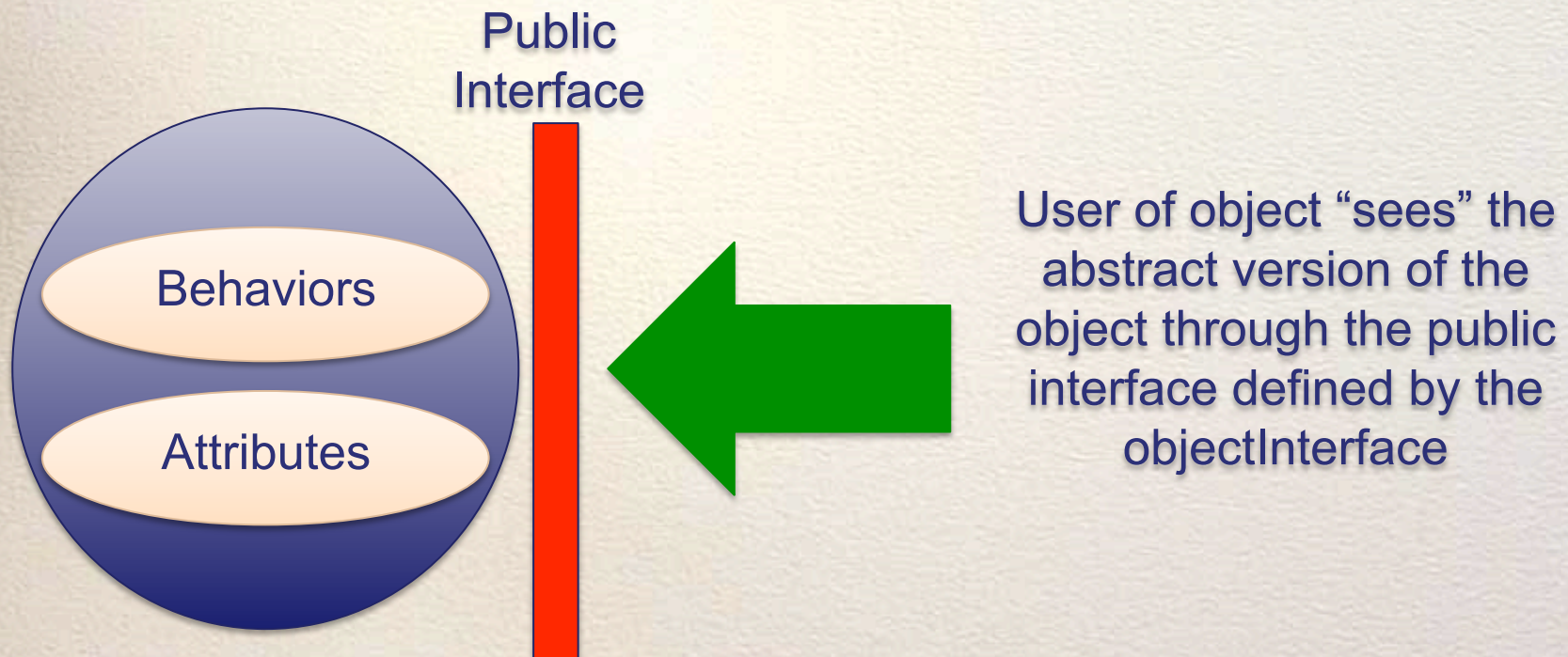
- **Attributes** and **behaviors** are enclosed (encapsulated) within the logical boundary of the object entity
 - In structured or procedural systems, data and code are typically maintained as separate entities (e.g., modules and data files)
 - In Object Technology systems, each object contains the data (attributes) and the code (behaviors) that operates upon those attributes

Abstraction

- Encapsulation implements the concept of **abstraction**:
 - details associated with object sub-components are enclosed within the logical boundary of the object
 - user of object only “sees” the public interface of the object, all the internal details are hidden

Note - In Python, encapsulation is merely a programming convention. Other languages (e.g., Java) enforce the concept more rigorously.

Abstraction



User of object “sees” the abstract version of the object through the public interface defined by the `objectInterface`

Encapsulation makes abstraction possible

Abstraction in your life



You know the public interface. Do you know implementation details?
Do you care?

As long as the public interface stays the same, you don't care about implementation changes

Implementing Public/Private Interfaces

Can we ENFORCE use of getters and setters? If I design a class I would like to make sure no one can access my instance variables directly, they **MUST** use my getters and setters

- CS211 Preview: In Java you will be able to enforce access restrictions on your instance variables... you can (and should) make them *private* so Java itself enforces data encapsulation.
- So... does Python support “*private*” instance variables? Yes (and no)

Implementing Public/Private Interfaces

- Python attributes and methods are public by default.
 - **public attributes**: any other class or function can see and change the attribute `myCircle.radius = 20`
 - **public method**: any other class or function can call the method `myCircle.method1()`
- Make things private by adding `__` (two underscores) to the beginning of the name:
 - `self.__radius = 20` **# Private attribute**
 - `def __method1():` **# Private method**

Implementing Public/Private Interfaces

- Private attributes can (almost) only be accessed by methods defined in the class
- Private methods can (almost) only be called by other methods defined in the class
- Idea: Everything defined in the class has access to private parts.

Hiding your private parts (in Python)

- You can create somewhat private parts in Python. Naming an instance variable with an `__` (two underscores) makes it private.

```
class Student(object):
    def __init__(self, first, last, age):
        self.__name = first
        self.__lastName = last
        self.__age = age

    def getName(self):
        return self.__name

def main():
    aStudent = Student("Karl", "Johnson", 18)
    nm = aStudent.name
    #nm = circ.getName()
    print nm
```

```
Traceback (most recent call last):
  File "Private.py", line 23, in <module>
    main()
  File "Private.py", line 17, in main
    nm = aStudent.name
AttributeError: 'Student' object has no attribute 'name'
...s/cs112/spring09/samplecode/objects >
```

Hiding your private parts (in Python)

- Be a little sneakier then.. use `__name`:

```
class Student(object):
    def __init__(self, first, last, age):
        self.__name = first
        self.__lastName = last
        self.__age = age

    def getName(self):
        return self.__name

def main():
    aStudent = Student("Karl", "Johnson", 18)
    nm = aStudent.__name
    #nm = aStudent.getName()
    print nm
```

```
Traceback (most recent call last):
  File "Private.py", line 23, in <module>
    main()
  File "Private.py", line 17, in main
    nm = aStudent.__name
AttributeError: 'Student' object has no attribute '__name'
...s/cs112/spring09/samplecode/objects >
```

Nice try, but that won't work!

Hiding your private parts (in Python)

- Be super sneaky then.. use `__Student__name`:

```
class Student(object):
    def __init__(self, first, last, age):
        self.__name = first
        self.__lastName = last
        self.__age = age

    def getName(self):
        return self.__name

def main():
    aStudent = Student("Karl", "Johnson", 18)
    nm = aStudent.__Student__name
    #nm = aStudent.getName()
    print nm
```

```
...s/cs112/spring09/samplecode/objects > python Private.py
Karl
...s/cs112/spring09/samplecode/objects > █
```

Ahh... you saw my private parts... that was rude!

So, it is possible to interact with private data in Python, but it is difficult and good programmers know not to do it. Using the defined interface methods (getters and setters) will make code more maintainable and safer to use

Getters and Setters (or) Accessors and Mutators

- These methods are a coding convention
- Getters/Accessors are methods that return an attribute
 - `def get_name(self):`
- Setters/Mutators are methods that set an attribute
 - `def set_name(self,newName):`

Why use getters?

- Definition of my getter:

```
def getName(self):  
    return self.name
```

What if I want to store the name instead as first and last name in the class?
Well, with the getter I only have to do this:

```
def getName(self):  
    return self.firstname + self.lastname
```

If I had used dot notation outside the class, then all the code **OUTSIDE** the class would need to be changed because the internal structure **INSIDE** the class changed. Think about libraries of code... If the Python-authors change how the Button class works, do you want to have to change YOUR code? No! Encapsulation helps make that happen. They can change anything inside they want, and as long as they don't change the method signatures, your code will work fine.

Getters help you hide the internal structure of your class!

Setters

- Another common method type are “setters”

```
def setAge(self, age):  
    self.age = age
```

Why? Same reason + one more. I want to hide the internal structure of my Class, so I want people to go through my methods to get and set instance variables. What if I wanted to start storing people's ages in dog-years? Easy with setters:

```
def setAge(self, age):  
    self.age = age / 7
```

More commonly, what if I want to add validation... for example, no age can be over 200 or below 0? If people use dot notation, I cannot do it. With setters:

```
def setAge(self, age):  
    if age > 200 or age < 0:  
        # show error  
    else:  
        self.age = age / 7
```

Getters and Setters

- Getters and setters are useful to provide data encapsulation. They hide the internal structure of your class and they should be used!

Printing objects

```
>>> aStudent = Student("Karl","Johnson", 18)
>>> print aStudent
<__main__.Student object at 0x8bd70>
```

Doesn't look so good! Define a special function in the class “`__str__`” that is used to convert your object to a string whenever needed

```
def __str__(self):
    return "Name is:"+ self.__name
```

```
Name is:KarlJohnson
```

-
- See BouncingBall Slides.

Data Processing with Class

- A class is useful for modeling a real-world object with complex behavior.
- Another common use for objects is to group together a set of information that describes a person or thing.
 - Eg., a company needs to keep track of information about employees (an `Employee` class with information such as employee's name, social security number, address, salary, etc.)

Data Processing with Class

- Grouping information like this is often called a *record*.
- Let's try a simple data processing example!
- A typical university measures courses in terms of credit hours, and grade point averages are calculated on a 4 point scale where an "A" is 4 points, a "B" is three, etc.

Data Processing with Class

- Grade point averages are generally computed using quality points. If a class is worth 3 credit hours and the student gets an “A”, then he or she earns $3(4) = 12$ quality points. To calculate the GPA, we divide the total quality points by the number of credit hours completed.

Data Processing with Class

- Suppose we have a data file that contains student grade information.
- Each line of the file consists of a student's name, credit-hours, and quality points.

Adams, Henry	127	228
Comptewell, Susan	100	400
DibbleBit, Denny	18	41.5
Jones, Jim	48.5	155
Smith, Frank	37	125.33

Data Processing with Class

- Our job is to write a program that reads this file to find the student with the best GPA and print out their name, credit-hours, and GPA.
- The place to start? Creating a `Student` class!
- We can use a `Student` object to store this information as instance variables.

Data Processing with Class

- ```
class Student:
 def __init__(self, name, hours, qpoints):
 self.name = name
 self.hours = float(hours)
 self.qpoints = float(qpoints)
```
- The values for `hours` are converted to `float` to handle parameters that may be floats, ints, or strings.
- To create a student record:  

```
aStudent = Student("Adams, Henry", 127, 228)
```
- The coolest thing is that we can store all the information about a student in a single variable!

# Data Processing with Class

- We need to be able to access this information, so we need to define a set of accessor methods.

- ```
def getName(self):  
    return self.name  
  
def getHours(self):  
    return self.hours  
  
def getQPoints(self):  
    return self.qpoints  
  
def gpa(self):  
    return self.qpoints/self.hours
```

These are commonly called “getters”

- For example, to print a student's name you could write:

```
print aStudent.getName()
```
- `aStudent.name`

Data Processing with Class

- How can we use these tools to find the student with the best GPA?
- We can use an algorithm similar to finding the max of n numbers! We could look through the list one by one, keeping track of the best student seen so far!

Data Processing with Class

Pseudocode:

```
Get the file name from the user
Open the file for reading
Set best to be the first student
For each student s in the file
    if s.gpa() > best.gpa
        set best to s
Print out information about best
```

Data Processing with Class

```
# gpa.py
# Program to find student with highest GPA
import string

class Student:

    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)

    def getName(self):
        return self.name

    def getHours(self):
        return self.hours

    def getQPoints(self):
        return self.qpoints

    def gpa(self):
        return self.qpoints/self.hours

def makeStudent(infoStr):
    name, hours, qpoints = string.split(infoStr, "\t")
    return Student(name, hours, qpoints)
```

```
def main():
    filename = raw_input("Enter name the grade file: ")
    infile = open(filename, 'r')
    best = makeStudent(infile.readline())
    for line in infile:
        s = makeStudent(line)
        if s.gpa() > best.gpa():
            best = s
    infile.close()
    print "The best student is:", best.getName()
    print "hours:", best.getHours()
    print "GPA:", best.gpa()

if __name__ == '__main__':
    main()
```

Helping other people use your classes

- Frequently, you will need to write classes other people will use
- Or classes you will want to use later, but have forgotten how

Answer: Document your class usage!

Putting Classes in Modules

- Sometimes we may program a class that could be useful in many other programs.
- If you might be reusing the code again, put it into its own module file with documentation to describe how the class can be used so that you won't have to try to figure it out in the future from looking at the code!

Module Documentation

- You are already familiar with “#” to indicate comments explaining what’s going on in a Python file.
- Python also has a special kind of commenting convention called the *docstring*. You can insert a plain string literal as the first line of a module, class, or function to document that component.

Module Documentation

- Why use a docstring?
 - Ordinary comments are ignored by Python
 - Docstrings are accessible in a special attribute called `__doc__`.
- Most Python library modules have extensive docstrings. For example, if you can't remember how to use `random`:

```
>>> import random
>>> print random.random.__doc__
random() -> x in the interval [0, 1).
```

Module Documentation

- Docstrings are also used by the Python online help system and by a utility called PyDoc that automatically builds documentation for Python modules. You could get the same information like this:

```
>>> import random
>>> help(random.random)
Help on built-in function random:

random(...)
    random() -> x in the interval [0, 1).
```

Module Documentation

- To see the documentation for an entire module, try typing `help(module_name)`!
- The following code for the projectile class has docstrings.

Module Documentation

```
# projectile.py

"""projectile.py
Provides a simple class for modeling the flight of projectiles."""

from math import pi, sin, cos

class Projectile:

    """Simulates the flight of simple projectiles near the earth's
    surface, ignoring wind resistance. Tracking is done in two
    dimensions, height (y) and distance (x)."""

    def __init__(self, angle, velocity, height):
        """Create a projectile with given launch angle, initial
        velocity and height."""
        self.xpos = 0.0
        self.ypos = height
        theta = pi * angle / 180.0
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)
```

Module Documentation

```
def update(self, time):
    """Update the state of this projectile to move it time seconds
    farther into its flight"""
    self.xpos = self.xpos + time * self.xvel
    yvell = self.yvel - 9.8 * time
    self.ypos = self.ypos + time * (self.yvel + yvell) / 2.0
    self.yvel = yvell

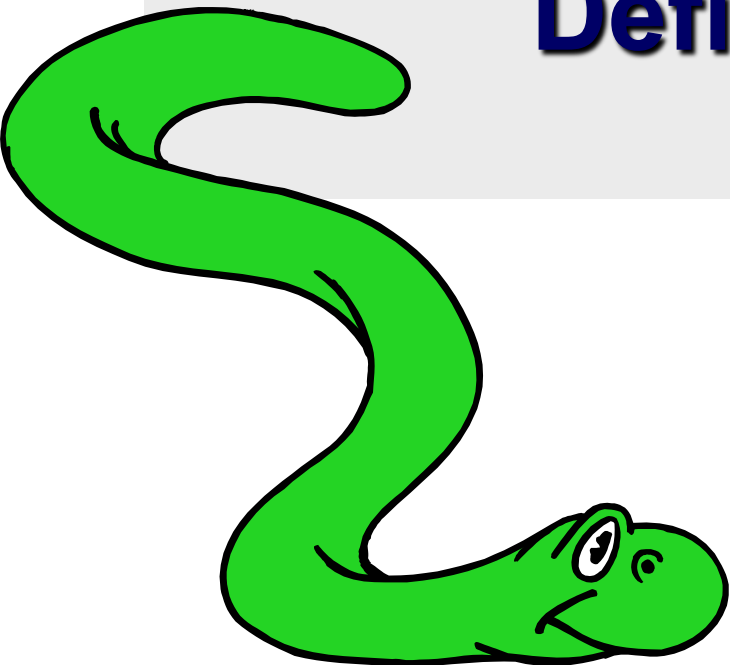
def getY(self):
    "Returns the y position (height) of this projectile."
    return self.ypos

def getX(self):
    "Returns the x position (distance) of this projectile."
    return self.xpos
```

PyDoc

- PyDoc The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.
- `pydoc -g` # Launch the GUI

Object Oriented Programming in Python: Defining Classes



It's all objects...

- Everything in Python is really an object.
- We've seen hints of this already...
`“hello”.upper()`
`list3.append('a')`
`dict2.keys()`
- These look like Java or C++ method calls.
- New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- *Instances* are objects that are created which follow the definition given inside of the class
- Python doesn't use separate class interface definitions as in some languages
- You just define the class and then use it

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block
- There must be a special first argument *self* in all of method definitions which gets bound to the calling instance
- There is usually a special method called *__init__* in most classes
- We'll talk about both later...

A simple class def: *student*

```
class student:  
    """A class representing a  
    student """  
    def __init__(self, n, a):  
        self.full_name = n  
        self.age = a  
    def get_age(self):  
        return self.age
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with () notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the `__init__` method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student("Bob", 21)
```

Constructor: `__init__`

- An `__init__` method can take any number of arguments.
- Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument `self` in the definition of `__init__` is special...

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument *self*
- In `__init__`, *self* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*

Self

- Although you must specify `self` explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):  
    self.age = num
```

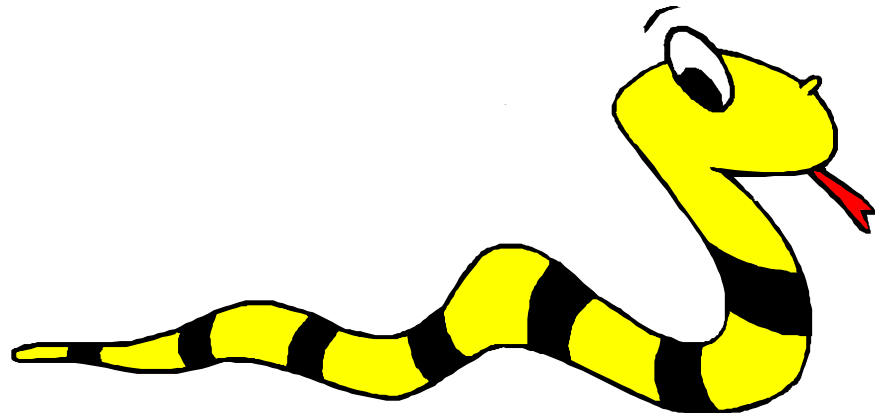
Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to “free”

- When you are done with an object, you don't have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There's also no “destructor” method for classes

Access to Attributes and Methods



Definition of student

```
class student:
    """A class representing a student
       """
    def __init__(self,n,a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```

Traditional Syntax for Access

```
>>> f = student("Bob Smith", 23)
```

```
>>> f.full_name # Access attribute  
"Bob Smith"
```

```
>>> f.get_age() # Access a method  
23
```

Accessing unknown members

- Problem: Occasionally the name of an attribute or method of a class is only given at run time...

- Solution:

```
getattr(object_instance, string)
```

- **string** is a string which contains the name of an attribute or method of a class
- **getattr(object_instance, string)** returns a reference to that attribute or method

getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)
>>> getattr(f, "full_name")
"Bob Smith"
>>> getattr(f, "get_age")
<method get_age of class
studentClass at 010B3C2>
>>> getattr(f, "get_age")() # call it
23
>>> getattr(f, "get_birthday")
# Raises AttributeError - No method!
```


hasattr(object_instance,string)

```
>>> f = student("Bob Smith", 23)
```

```
>>> hasattr(f, "full_name")
```

```
True
```

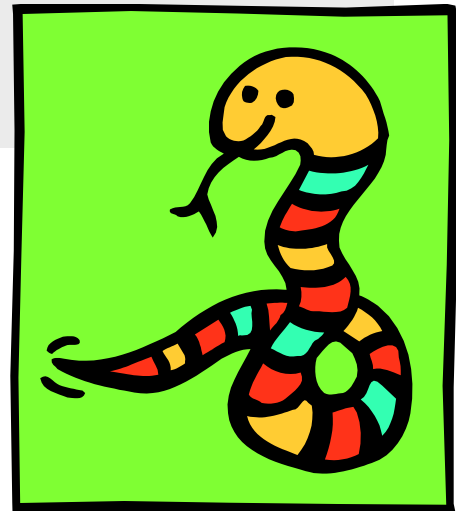
```
>>> hasattr(f, "get_age")
```

```
True
```

```
>>> hasattr(f, "get_birthday")
```

```
False
```

Attributes



Two Kinds of Attributes

- The non-method data stored by objects are called attributes
- *Data* attributes
 - Variable owned by a *particular instance* of a class
 - Each instance has its own value for it
 - These are the most common kind of attribute
- *Class* attributes
 - Owned by the *class as a whole*
 - *All class instances share the same value for it*
 - Called “static” variables in some languages
 - Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
 - Simply assigning to a name creates the attribute
 - Inside the class, refer to data attributes using **self** —for example, **self.full_name**

```
class teacher:
    "A class representing teachers."
    def __init__(self, n):
        self.full_name = n
    def print_name(self):
        print self.full_name
```

Class Attributes

- Because all instances of a class share one copy of a class attribute, when *any* instance changes it, the value is changed for *all* instances
- Class attributes are defined *within* a class definition and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
 - Access class attributes using `self.__class__.name` notation
-- This is just one way to do this & the safest in general.

```
class sample:  
    x = 23  
    def increment(self):  
        self.__class__.x += 1
```

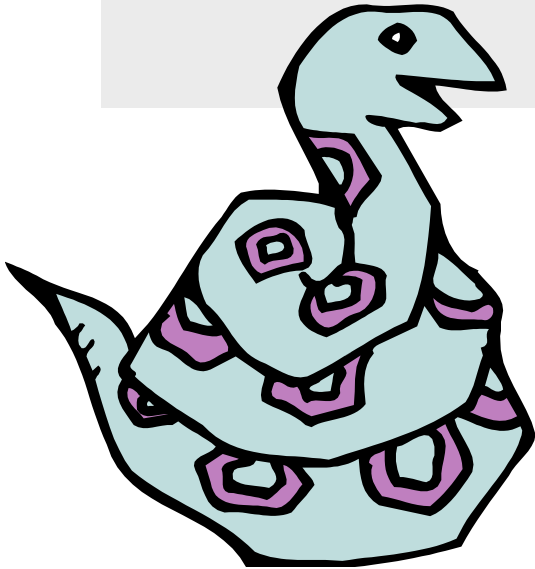
```
>>> a = sample()  
>>> a.increment()  
>>> a.__class__.x  
24
```

Data vs. Class Attributes

```
class counter:
    overall_total = 0
        # class attribute
    def __init__(self):
        self.my_total = 0
            # data attribute
    def increment(self):
        counter.overall_total = \
        counter.overall_total + 1
        self.my_total = \
        self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Inheritance



Subclasses

- Classes can *extend* the definition of other classes
 - Allows use (or extension) of methods and attributes already defined in the previous one
- To define a subclass, put the name of the superclass in parens after the subclass's name on the first line of the definition

```
Class Cs_student(student) :
```

- Python has no 'extends' keyword like Java
- Multiple inheritance is supported

Multiple Inheritance

- Python has two kinds of classes: old and new (more on this later)
- Old style classes use *depth-first, left-to-right* access
- New classes use a more complex, dynamic approach

```
class AO(): x = 0
class BO(AO): x = 1
class CO(AO): x = 2
class DO(BO,CO): pass

ao = AO()
bo = BO()
co = CO()
do = DO()
```

```
>>> from mi import *
>>> ao.x
0
>>> bo.x
1
>>> co.x
2
>>> do.x
1
>>>
```

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass
 - The old code won't get executed
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of method

```
parentClass.methodName (self, a, b, c)
```

- The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor

Definition of a class extending student

```
Class Student:  
    "A class representing a student."
```

```
def __init__(self,n,a):  
    self.full_name = n  
    self.age = a
```

```
def get_age(self):  
    return self.age
```

```
Class Cs_student (student):  
    "A class extending student."  
  
def __init__(self,n,a,s):  
    student.__init__(self,n,a) #Call __init__ for student  
    self.section_num = s  
  
def get_age(): #Redefines get_age method entirely  
    print "Age: " + str(self.age)
```

Extending `__init__`

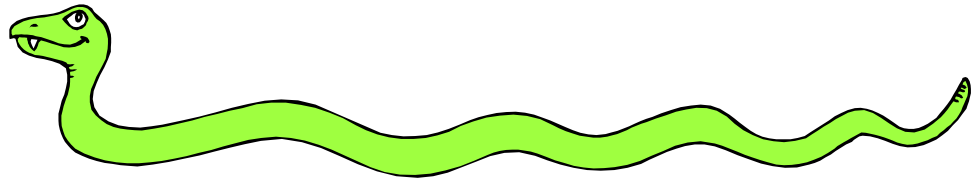
Same as redefining any other method...

- Commonly, the ancestor's `__init__` method is executed in addition to new commands
- You'll often see something like this in the `__init__` method of subclasses:

```
parentClass.__init__(self, x, y)
```

where `parentClass` is the name of the parent's class

Special Built-In Methods and Attributes



Built-In Members of Classes

- Classes contain many methods and attributes that are always included
 - Most define automatic functionality triggered by special operators or usage of that class
 - Built-in attributes define information that must be stored for all classes.
- All built-in members have double underscores around their names:

`__init__` `__doc__`

Special Methods

- E.g., the method `__repr__` exists for all classes, and you can always redefine it
- `__repr__` specifies how to turn an instance of the class into a string
 - `print f` sometimes calls `f.__repr__()` to produce a string for object `f`
 - Typing `f` at the REPL prompt calls `__repr__` to determine what to display as output

Special Methods – Example

```
class student:  
    ...  
    def __repr__(self):  
        return "I'm named " + self.full_name  
    ...
```

```
>>> f = student("Bob Smith", 23)
```

```
>>> print f
```

```
I'm named Bob Smith
```

```
>>> f
```

```
"I'm named Bob Smith"
```


Special Methods

- You can redefine these as well:

`__init__` : The constructor for the class

`__cmp__` : Define how `==` works for class

`__len__` : Define how `len (obj)` works

`__copy__` : Define how to copy a class

- Other built-in methods allow you to give a class the ability to use `[]` notation like an array or `()` notation like a function call

Special Data Items

- These attributes exist for all classes.

`__doc__` : Variable for documentation string for class

`__class__` : Variable which gives you a reference to the class from any instance of it

`__module__` : Variable which gives a reference to the module in which the particular class is defined

`__dict__` : The dictionary that is actually the namespace for a class (but not its superclasses)

- Useful:

- `dir(x)` returns a list of all methods and attributes defined for object `x`

Special Data Items – Example

```
>>> f = student("Bob Smith", 23)
```

```
>>> print f.__doc__
```

```
A class representing a student.
```

```
>>> f.__class__
```

```
< class studentClass at 010B4C6 >
```

```
>>> g = f.__class__("Tom Jones",  
34)
```

Private Data and Methods

- Any attribute/method with two leading underscores in its name (but none at the end) is **private** and can't be accessed outside of class
- Note: Names with two underscores at the beginning *and the end* are for built-in methods or attributes for the class
- Note: There is no 'protected' status in Python; so, subclasses would be unable to access these private data either