# Python Basics

---

**About Python:** Python is a high level scripting language with object oriented features.

## 1. Syntax

**Python programs can be written using any text editor and should have the extension .py. Python programs do not have a required first or last line, but can be given the location of python as their first line: #!/usr/bin/python and become executable. Otherwise, python programs can be run from a command prompt by typing python file.py. There are no braces {} or semicolons ; in python. It is a very high level language. Instead of braces, blocks are identified by having the same indentation.**

```
#!/usr/bin/python
if (x > y):
   print("x is greater than y")
   x = x -1
else:
   print("x is less than or equal to
y")
```

**Comments** are supported in the same style as Perl:

```
print("This is a test") #This is a comment.
#This is also a comment. There are no multi-line
comments.
```

## 2. Variables and Datatypes

**Variables in Python follow the standard nomenclature of an alphanumeric name beginning in a letter or underscore. Variable names are case sensitive. Variables do not need to be declared and their datatypes are inferred from the assignment statement.**

Python supports the following data types:

- boolean
- integer
- long
- float
- string
- list
- object
- None

**Example:**

```
bool = True
name = "Craig"
age = 26
pi = 3.14159
print(name + ' is ' + str(age) + ' years old.')
-> Craig is 26 years old.
```

**Variable Scope:** Most variables in Python are local in scope to their own function or class. For instance if you define a = 1 within a function, then a will be available within that entire function but will be undefined in the main program that calls the function. Variables defined within the main program are accessible to the main program but not within functions called by the main program.

**Global Variables:** Global variables, however, can be declared with the global keyword.

```
a = 1
b = 2
def Sum():
   global a, b
   b = a + b
Sum()
```

```
print(b)
-> 3
```

## 3. Statements and Expressions

Some basic Python statements include:

- **print**: Output strings, integers, or any other datatype.
- **The assignment statement**: Assigns a value to a variable.
- **input**: Allow the user to input numbers or booleans. WARNING: input accepts your input as a command and thus can be unsafe.
- **raw_input**: Allow the user to input strings. If you want a number, you can use the int or float functions to convert from a string.
- **import**: Import a module into Python. Can be used as import math and all functions in math can then be called by math.sin(1.57) or alternatively from math import sin and then the sine function can be called with sin(1.57).

Python expressions can include:

```
a = b = 5 #The assignment statement
b += 1 #post-increment
c = "test"
import os,math #Import the os and math modules
from math import * #Imports all functions from the math module
```

**Examples:**

```
print "Hello World"
print('Print works with or without parenthesis')
print("and single or double quotes")
print("Newlines can be escaped like\nthis.")
print("This text will be printed"),
print("on one line becaue of the comma.")
name = raw_input("Enter your name: ")
a = int(raw_input("Enter a number: "))
print(name + "'s number is " + str(a))
a = b = 5
a = a + 4
print a,b
9 5
```

## 4. Operators and Maths

**Operators:**

- Arithmatic: +, -, *, /, and % (modulus)
- Comparison: ==, !=, <, >, <=, >=
- Logical: and, or, not
- Exponentiation: **
- Execution: os.system('ls -l')
  #Requires import os

**Maths:** Requires import math

- `Absolute Value:` a = abs(-7.5)
- `Arc sine:` x = asin(0.5) `#returns in rads`
- `Ceil (round up):` print(ceil(4.2))
- `Cosine:` a = cos(x) `#x in rads`
- `Degrees:` a = degrees(asin(0.5)) `#a=30`
- `Exp:` y = exp(x) `#y=e^x`
- `Floor (round down):` a = floor(a+0.5)
- `Log:` x = log(y); `#Natural Log`
  x = log(y,5); `#Base-5 log`

- Log Base 10: x = log10(y)
- Max: mx = max(1, 7, 3, 4) #7
  mx = max(arr) #max value in array
- Min: mn = min(3, 0, -1, x) #min value
- Powers: x = pow(y,3) #x=y^3
- Radians: a = cos(radians(60)) #a=0.5
- Random #: Random number functions require import random
  random.seed() #Set the seed based on the system time.
  x = random() #Random number in the range [0.0, 1.0)
  y = randint(a,b) #Random integer in the range [a, b]
- Round: print round(3.793,1; #3.8 - rounded to 1 decimal
  a = round(3.793,0) #a=4.0
- Sine: a = sin(1.57) #in rads
- Square Root: x = sqrt(10) #3.16...
- Tangent: print tan(3.14)# #in rads

# 5. Strings

Strings can be specified using single quotes or double quotes. Strings do not expand escape sequences unless it is defined as a raw string by placing an r before the first quote: print 'I\'ll be back.'.

```
print r'The newline \n will not expand'
a = "Gators"
print "The value of a is \t" + a
-> The value of a is	Gators
```

If a string is not defined as raw, escapes such as \n, \r, \t, \\, and \" may be used.

Optional syntax: Strings that start and end with """ may span multiple lines: print """

```
This is an example of a string in the heredoc syntax.
This text can span multiple lines
"""
```

**String Operators:**

Concatenation is done with the + operator.

Converting to numbers is done with the casting operations:

```
x = 1 + float(10.5) #$x=11.5, float
x = 4 - int("3") #$x=1, int
```

You can convert to a string with the str casting function:

```
s = str(3.5)
name = "Lee"
print name + "'s number is " + str(24)
```

**Comparing Strings:**

Strings can be compared with the standard operators listed above: ==, !=, <, >, <=, and >=.

**String Functions:**

```
s = "Go Gators! Come on Gators!"
```

- **Extracting substrings**: Strings in Python can be subscripted just like an array: s[4] = 'a'. Like in IDL, indices can be specified with slice notation i.e., two indices separated by a colon. This will return a substring containing characters index1 through index2-1. Indices can also be negative, in which case they count from the right, i.e. -1 is the last character. Thus substrings can be extracted like
  x = s[3:9] #x = "Gators"
  x = s[:2] #x = "Go"
  x = s[19:] #x = "Gators!"
  x = s[-7:-2] #x = "Gator"
  However, strings are immutable so s[2] = 'a' would cause an error.
- int **count**(sub [,start[,end]]): returns the number of occurances of the substring sub in the string
  x = s.count("Gator") #x = 2
- boolean **endswidth**(sub [,start[,end]]): returns true if the string ends with the specified substring and false otherwise:

x = s.endswith("Gators") #x = False

- int **find**(sub [,start[,end]]): returns the numeric position of the first occurance of sub in the string. Returns -1 if sub is not found.

  x = s.find("Gator") #x = 3

  x = s.find("gator") #x = -1
- string **join**(array): combines elements of the string array into a single string and returns it. The separator between elements is the string providing this method.

  a = ['abc','def','ghi']

  t = "--"

  x = t.join(a) #x = abc--def--ghi
- int **len**(string): returns the length of the string

  x = len(s) #x = 26
- string **lower**(): returns a version of a string with all lower case lettters.

  print s.lower() #go gators! come on gators!
- string **replace**(old, new [,count]): returns a copy of the string with all occurances of old replaced by new. If the optional count argument is given, only the first count occurances are replaced.

  x = s.replace("Gators","Tigers",1) #x = Go Tigers! Come on Gators!
- int **rfind**(sub [,start[,end]]): same as find but returns the numeric position of the last occurance of sub in the string.

  x = s.rfind("Gator") #x = 19
- array **split**([sep [,maxsplit]]): splits a single string into a string array using the separator defined. If no separator is defined, whitespace is used. Consecutive whitespace delimiters are then treated as one delimiter. Optionally you can specify the maximum number of splits so that the max size of the array would be maxsplit+1.

  a = s.split() #a=['Go', 'Gators!', 'Come', 'on', 'Gators!']
- boolean **startswidth**(sub [,start[,end]]): returns true if the string starts with the specified substring and false otherwise:

  x = s.startswith("Go") #x = True
- string **strip**([chars]): returns a copy of the string with leading and trailing characters removed. If chars (a string) is not specified, leading and trailing whitespace is removed.
- string **upper**(): returns a version of a string with all upper case lettters.

## 6. Arrays

**Arrays in basic Python are actually lists that can contain mixed datatypes. However, the numarray module contains support for true arrays, including multi-dimensional arrays, as well as IDL-style array operations and the where function. To use arrays, you must import numarray or from numarray import *. Unfortunately, numarray generally only suports numeric arrays. Lists must be used for strings or objects. By importing numarray.strings and numarray.objects, you can convert string and object lists to arrays and use some of the numarray features, but only numeric lists are fully supported by numarray.**

- **Creating lists: A list can be created by defining it with []. A numbered list can also be created with the range function which takes start and stop values and an increment.**

  list = [2, 4, 7, 9]

  list2 = [3, "test", True, 7.4]

  a = range(5) **#a = [0,1,2,3,4]**

```
a = range(10,0,-2) #a = [10,8,6,4,2]
```

**An empty list can be initialized with [] and then the append command can be used to append data to the end of the list:**

```
a=[]
a.append("test")
a.append(5)
print a
-> ['test', 5]
```

**Finally, if you want a list to have a predetermined size, you can create a list and fill it with None's:**

```
a=[None]*length
a[5] = "Fifth"
a[3] = 6
print len(a)
-> 10
print a
-> [None, None, None, 6, None, 'Fifth', None, None, None, None]
```

- **Removing from lists: The pop method can be used to remove any item from the list:**

```
a.pop(5)
print a
-> [None, None, None, 6, None, None, None, None, None]
```

- **Creating arrays: An array can be defined by one of four procedures: zeros, ones, arange, or array. zeros creates an array of a specified size containing all zeros:**

```
a = zeros(5) #a=[0 0 0 0 0]
```

**ones similarly creates an array of a certain size containing all ones:**

```
a = ones(5) #a=[1 1 1 1 1]
```

**arange works exactly the same as range, but produces an array instead of a list:**

```
a = arange(10,0,-2) #a = [10 8 6 4 2] And finally, array can be used to convert a list to an array. For instance, when reading from a file, you can create an empty list and take advantage of the append command and lists not having a fixed size. Then once the data is all in the list, you can convert it to an array:
a = [1, 3, 9] #create a list and append it
a.append(3)
a.append(5)
print a
```

```
-> [1, 3, 9, 3, 5]
a = array(a)
print a
-> [1 3 9 3 5]
```

- **Multi-dimensional lists: Because Python arrays are actually lists, you are allowed to have jagged arrays. Multi-dimensional lists are just lists of lists:**

```
a=[[0,1,2],[3,4,5]]
print a[1]
-> [3, 4, 5]
s = ["Lee", "Walsh", "Roberson"]
s2 = ["Williams", "Redick", "Ewing", "Dockery"]
s3 = [s, s2]
print s3[1][2]
-> Ewing
```

- **Multi-dimensional arrays: However, numarray does support true multi-dimensinoal arrays. These can be created through one of five methods: zeros, ones, array, arange, and reshape. zeros and ones work the same way as single dimensions except that they take a tuple of dimensions (a comma separated list enclosed in parentheses) instead of a single argument:**

```
a = zeros((3,5))
a[1,2] = 8
print a
-> [[0 0 0 0 0]
    [0 0 8 0 0]
    [0 0 0 0 0]]
b = ones((2,3,4)) #create a 2x3x4 array containing all ones.
```

**array works the same way as for 1-d arrays: you can create a list and then convert it to an array. Note with multi-dimensional arrays though, trying to use array to convered a jagged list into an array will cause an error. Lists must be rectangular to be able to be converted to arrays.**

```
s = ["Lee", "Walsh", "Roberson", "Brewer"]
s2 = ["Williams", "Redick", "Ewing", "Dockery"]
s3 = [s, s2]
s4 = array(s3)
print s4 + "test"
```

```
-> [['Leetest', 'Walshtest', 'Robersontest', 'Brewertest'],
    ['Williamstest', 'Redicktest', 'Ewingtest', 'Dockerytest']]
print s4[:,1:3]
-> [['Walsh', 'Roberson'],
    ['Redick', 'Ewing']]
```

**arange also works the same as with 1-d arrays except you need to pass the shape parameter:**

```
a = arange(25, shape=(5,5))
```

**,br> And finally, reshape can be used to convert a 1-d array into a multi-dimensional array. To create a 5x5 array with the elements numbered from 0 to 24, you could use:**

```
b = arange(25)
b = reshape(b,5,5)
```

- **Array Dimensions and Subscripts: When creating a multi-dimensional array, the format is ([[depth,] height,] width). Therefore, when accessing array elements in a two dimensional array, the row is listed first, then the column. When accessing an element of a two-dimensional list, the following notation must be used:** `list[i][j]`**. However, two dimensional arrays can also use the notation:** `array[i,j]`**. In fact, this is the preferred notation of the two for arrays because you cannot use wildcards in the first dimension of the** `array[i][j]` **notation (i.e., array[1:3][4] would cause an error whereas array[1:3,4] is valid).**

  **Wildcards can be used in array subscripts using the** `:` **, which is known as slicing. This is similar to IDL, with one major difference: if** `a=[0 1 2 3 4 5]`**, in IDL** `a[1:4] = [1 2 3 4]`**, but in Python,** `a[1:4] = [1 2 3]`**. In Python, when slicing array[i:j], it returns an array containing elements from i to j-1. Just like with strings, indices of arrays can be negative, in which case they count from the right instead of the left, i.e.** `a[-4:-1] = [2 3 4]`**. A** `:` **can also specify the rest of the elements or up to element, or all elements and arrays or lists can be used to subscript other arrays:**

```
print a[:3] #[0 1 2]
print a[4:] #[4 5]
print a[:] #[0 1 2 3 4 5]
print a[[1,3,4]] #[1 3 4]
```

***Note that slicing in python does not create a new array but just a pointer to the original array. b=a[0:10] followed by b[0] = 5 also changes a[0] to 5. To avoid this, use b = copy(a[0:10])***

**Array Operators:**

- Concatenation:

- Lists: a + b
  For Lists, the + operator appends the list on the right (b) to the list on the left.
  a = ["Roberson", "Walsh"]
  b = ["Lee", "Humphrey"]
  -> a+b = ["Roberson", "Walsh", "Lee", "Humphrey"]
- Arrays: concatenate((a,b)[,axis])
  For arrays, use the numarry function concatenate. It also allows you to specify the axis when concatenating multi-dimensional arrays.
  b = arange(5)
  print concatenate((b, arange(6)))
  -> [0 1 2 3 4 0 1 2 3 4 5]
  b=reshape(b,5,1)
  print concatenate((b,a),axis=1)
  -> [[0 0 0 0]
  [1 0 0 0]
  [2 0 8 0]
  [3 0 0 0]
  [4 0 0 0]]

- Equality: a == b and Inequality: a != b
  For lists, these work the same as for scalars, meaning they can be used in if statments. For arrays, they return an array containing true or false for each array element.

**Array Functions:** All functions but len are for arrays only

- **len**: returns the length of a list/array.
  s = ["Lee", "Walsh", "Roberson", "Brewer"]
  print len(s) #4
- **argmax**([axis]): returns the index of the largest element in a 1D array or an array of the largest indices along the specified axis for a multi-dimensional array.
  a = array([[1,6,9], [2,4,0], [7,4,8]])
  print a.argmax(1)
  -> [2 1 2]
- **argmin**([axis]): returns the index of the smallest element in a 1D array or an array of the smallest ndices along the specified axis for a multi-dimensional array.
  b = array([2,4,7,1,3,-1,5])
  print b.argmin()
  -> 5
- **argsort**([axis]): returns an array of indices that allow access to the elements of the array in ascending order.
  print b.argsort()
  -> [5 3 0 4 1 6 2]
  print b[b.argsort()]
  -> [-1 1 2 3 4 5 7]
  print a.argsort(1)
  -> [[0 1 2]
  [2 0 1]
  [1 0 2]]
- **astype**(type): returns a copy of the array converted to the specified type.
  a = a.astype('Float64')
  b = b.astype('Int32')
- **copy**(): returns a copy of the array.
  c = a[:,2].copy()
  print c
  -> [9 0 8]
- **diagonal**(): for multi-dimensional arrays, returns the diagonal elements of the array, where the row and column indices are equal.

print a.diagonal()
-> [1 4 8]

- **info**(): prints informations about the array which may be useful for debugging.
- **max**(): returns the largest element in the array
  print a.max()
  -> 9
- **mean**(): returns the average of all elements in an array
  print a.mean()
  -> 4.55555555556
- **min**(): returns the smallest element in the array
  print b.min()
  -> -1
- **nelements**(): returns the total number of elements in the array
  print a.nelements()
  -> 9
- **product**(array [,axis]): returns the product of an array or an array of the products along an axis of an array.
  print product(b)
  -> -840
  print product(a,1)
  -> [ 54 0 224]
- **reshape**(array, shape): function that changes the shape of an array. But the new shape must have the same size as the old shape, otherwise an error will occur.
  c = reshape(a, 9)
  a = reshape(c,(3,3))
- **resize**(shape): shrinks/grows the array to a new shape. Can be called as a method (replaces old array) or a function. The new shape does not have to be the same size as the old shape. If it is smaller, values will be cut off, and it if is bigger, values will repeat.
  a.resize(5)
  print a
  -> [1 6 9 2 4]
  a.resize(2,6)
  print a
  -> [[1 6 9 2 4 0]
  [7 4 8 1 6 9]]
  c = resize(a,(2,2))
  print c
  -> [[1 6]
  [9 2]]
- **shape**(array): returns the dimensions of the array in a tuple
  print shape(a), shape(b), shape(a)[0]*shape(a)[1]
  -> (3,3) (7,) 9
- **sort**(array [,axis]): returns an array containing a copy of the data in the array and the elements sorted in increasing order. In the case of a multi-dimensional array, the data will be sorted along one axis and not across the whole array.
  print sort(b)
  -> [-1 1 2 3 4 5 7]
  print sort(a)
  -> [[1 6 9]
  [0 2 4]
  [4 7 8]]
  print sort(a,0)
  -> [[1 4 0]
  [2 4 8]
  [7 6 9]]

- **stddev**(): returns the std deviation of all elements in the array
  print a.stddev()
  -> 3.16666666667
- **sum**(): Can be called as a method or a function. The behavior is identical for 1-d arrays. But for multi-dimensional arrays, calling as a method returns the sum of the entire array, whereas calling it as a function allows you to specify an axis and returns an array with the sums along that axis.
  print a.sum()
  -> 41
  print sum(a)
  -> [10 14 17]
  print sum(a,1)
  -> [16 6 19]
- **trace**(): Returns the sum of the diagonal elements of an array
  print a.trace()
  -> 13
- **type**(): returns a string containing the type of the array.
  print a.type()
  -> Int32
- **tolist**(): returns a list containing the same data as the array.
  c = a.tolist()
- **transpose**(): Can be called as a method (replaces old array) or a function. Returns the transpose of the array.
  a.transpose()
  b = transpose(a)
- **where**(expr, 1, 0): Similar to the IDL where function. Returns an array of the same size and dimensions containing 1 if the condition is true and 0 if the condition is false. Any value may be substituted for 1 and 0, but they are the recommended values (i.e. true, false) so that **compress** can be used to extract values from the array: **compress**(mask_array, data_array).
  c = where(b > 2, 1, 0)
  print c
  -> [0 1 1 0 1 0 1]
  print compress(c,b)
  -> [4 7 3 5]
  c = where(a > 2, 1, 0)
  print c
  -> [[0 1 1]
  [0 1 0]
  [1 1 1]]
  print compress(c,a)
  -> [6 9 4 7 4 8]

# 7. Conditionals

- **if**:
  if expr: statement
- **if-else**:
  if expr: statement1
  else: statement2
- **if-elseif**: if expr: statement1
  elif expr: statement2
  else: statement3

  Multiple elifs can be included in the same if statement. There is no switch or

**Examples:**

```
if a > b: print "a is greater than b";

if (a > b):
   print "a is greater than b"
   print "blocks are defined by indentation"
elif (a < b):
   print "a is less than b"
else:
   print "a is equal to b"
```

case statement so multiple elifs must be used instead. While parenthesis are not required around the expression, they can be used.

## 8. Loops

- **for**: for var in range(start [,stop [,inc]]): statements
  Not unsimilar to IDL and basic, except for the range statement. var can be any variable. The range statement can take start and stop values, and an increment.
- **while**: while expr: statements
  Executes statements while the expression is true.
- **continue**: continue
  Skips the rest of the body of the loop for the current iteration and continue execution at the beginning of the next iteration.
- **break**: break
  Ends the execution of the current loop.
- **else**: else
  for and while loops can both have else clauses, which are executed after the loop terminates normally by falsifying the conditional, but else clauses are not executed when a loop terminates via a break statement.
- **foreach**: for x in array: statements
  Loops over the array given by array. On each iteration, the value of the current element is assigned to x and the internal array pointer is advanced by one.

**Examples:**

```
for j in range(10): print "Value number " +
str(j) +" is "+value[j]

for j in range(10,0,-2):
  x = x + j
  print x

while (b < a):
  print "b is less than a."
  b=b+1

for j in range(0,10):
  while(k < j):
    print "j = " + str(j) + " k = "+str(k)
    if (j == 1): break
    k=k+1
  print "j equals k or j equals 1"

a = ["abc","def","ghi"]
for x in a:
  print x
```

## 9. Functions

- **Definition:** Functions in Python are defined with the following syntax:
  ```
  def funct(arg_11, arg_2, ..., arg_n):
    print "This is a function."
    return value
  ```
  Any Python code, including other function and class definitions, may appear inside a function. Functions may also be defined within a conditional, but in that case the function's definition must be processed prior to its being called. Python does not support function overloading but does support variable number of arguments, default arguments, and keyword arguments. Return types are not specified by functions.
- **Arguments:** Function arguments are passed by value so that if you change the value of the argument within the function, it does not get changed outside of the function. If you want the function to be able to modify non-local variables, you must declare them as global in the first line of the function. Note that if you declare any variables as global, that name cannot be reused in the argument list, i.e. this would cause an error:
  ```
  function double(x):
    global x
  ```

```
  x = x*2
  return
double(x)
```
Instead this could be done
```
function double(n):
  n = n * 2
  return n
x = double(x)
```
Or
```
function doubleX():
  global x
  x = x * 2
  return
doubleX()
```

- **Default Arguments:** A function may define default values for arguments. The default must be a constant expression or array and any defaults should be on the right side of any non-default arguments.
```
def square(x = 5):
  return x*x
```
If this function is called with `square()`, it will return 25. Otherwise, if it is called with `square(n)` , it will return n^2.
- **Variable length argument lists:** Variable length arguments are supported by being wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur:
```
def var_args(arg1, arg2, *args):
```
- **Keyword arguments:** Functions can also be called using arguments of the form *keyword* = *value*:
```
def player(name, number, team="Florida"):
  print(name + "wears number " + str(number) + "for " + team)
player("Matt Walsh", 44)
player(number = 44, name = "David Lee")
player("Anthony Roberson", number = 1)
player(name = "J.J. Redick", number = 4, team = "Duke")
```
- **Return:** Values are returned from the function with the return command: `return var`. You can not return multiple values, but that can be achieved by returning an array or object. Return immediately ends execution of the function and passes control back to the line from which it was called.
- **Variable Functions:** Python supports the concept of variable functions. That means that if a variable can point to a function instead of a value. Objects within a method can be called similarly.
```
def test():
  print 'This is a test.'
var = test
var() #this calles test()
var = circle.setRadius
var(3) #this calls circle.setRadius(3)
```

## 10. Classes and OOP

Python supports OOP and classes to an extent, but is not a full OOP language. A class is a collection of variables and functions working with these variables. Classes are defined somewhat similarly to Java, but differences include `self` being used in place of **this** and constructors being named

**Example Class:**
```
class Rectangle:
#Optionally define variable width
  width = 0
#Constructor with default arguments
  def __init__(self, width = 0, height = 0):
```

__init__ instead of **classname**. Also note that self must be used every time a class-wide variable is referenced and must be the first argument in each function's argument list, including the constructor. In addition, functions and constructors cannot be overloaded, but as discussed above, do support default arguments instead. Like functions, a class must be defined before it can be instantiated. In Python, all class members are public.

- **Initializing vars**: Only constant initializers for class variables are allowed (n = 1). To initialize variables with non-constant values, you must use the constructor. You cannot declare unitialized variables.
- **Encapsulation**: Python does not really support encapsulation because it does not support data hiding through private and protected members. However some pseudo-encapsulation can be done. If an identifier begins with a double underline, i.e. __a, then it can be referred to within the class itself as self.__a, but outside of the class, it is named instance._classname__a. Therefore, while it can prevent accidents, this pseudo-encapsulation cannot really protect data from hostile code.
- **Inheritence**: Python allows classes to be extended (see right) by adding the base class name in parenthesis after the derived class name: class Derived(Base):. The child class takes all the variables and functions from the parent class and can extend that class by adding additional variables and adding or overriding functions. If class B extends class A, then A or B can be used anywhere an A is expected, but only B can be used where a B is expected because it contains additional information/methods not found in A. In addition, Python supports multiple inheritence: class Derived(Base1, Base2, Base3):
- **Abstract classes**: Abstract classes and interfaces are not supported in Python. In Python, there is no difference between an abstract class and a concrete class. Abstract classes create a template for other classes to extend and use. Instances can not be created of abstract classes but they are very useful when working with several objects that share many characteristics. For instance, when creating a database of people, one could define the abstract class "Person", which would contain basic attributes and functions common to all people in the database. Then

```
    self.width = width
    self.height = height
#functions
  def setWidth(self, width):
    self.width = width
  def setHeight(self, height):
    self.height = height
  def getArea(self):
    return self.width * self.height

arect = Rectangle() #create a new Rectangle with dimensions 0x0.
arect.setWidth(4)
arect.setHeight(6)
print arect.getArea()
 -> 24
rect2 = Rectangle(7,3) #new Rectangle with dimensions 7x3.
```

**Extended Class:**

```
class RectWithPerimeter(Rectangle):
#add new functions
  def getPerimeter(self):
    return 2*self.height + 2*self.width
  def setDims(self, width, height):
#call base class methods from Rectangle
    Rectangle.setWidth(self, width)
    Rectangle.setHeight(self, height)
arect = RectWithPerimeter(6,5) #Uses the constructor from Rectangle because no new constructor is provided to override it.
print arect.getArea() #Uses the getArea function from Rectangle and prints 30.
print arect.getPerimeter() #Uses getPerimeter from RectWithPerimeter and prints 22.
arect.setDims(4,9) #Use setDims to change the dimensions.
```

child classes such as "SinglePerson", "MarriedCouple", or "Athlete" could be created by extending "Person" and adding appropriate variables and functions. The database could then be told to expect every entry to be an object of type "Person" and thus any of the child classes would be a valid entry. In Python, you could create a class Person and extend it with the child classes listed above, but you could not prevent someone from instantiating the Person class.

- **Parent**: The parent keyword is not supported by Python, but you can call methods from the base classes directly: BaseClass.method_name(self, arguments) (see right).
- **Constructors**: Constructors are fuctions that are automatically called when you create a new instance of a class. They can be used for initialization purposes. A function is a constructor when it has the name __init__. When extending classes, if a new constructor is not defined, the constructor from the parent class is used (see right). When an object of type RectWithPerimeter is created, the constructor from Rectangle is called. If however, I were to add a function in RectWithPerimeter with the name __init__ , then that function would be used as its constructor.
- **Comparing Objects**: Objects can be compared using the == and != operators. Two objects are equal only if they are the same instance of the same object. Even if two objects have the same attributes and values and are instances of the same class, they are not equal if the are separate instances.

## 11. File I/O

- **Opening Files**: file **open**(string filename, string mode):
  open can be used to open files for reading, writing, and appending. It binds a named file object to a stream that can then be used to read/write data. Possible modes include:
  - 'r': Open for reading.
  - 'w': Open for writing. Any existing data will be overwritten.
  - 'a': Open for writing. New data will be appended to existing data.
  - 'b': Use this flag when working with binary files (e.g. 'rb').
- **Checking Files**: Python supports several methods of checking if a file exists and checking its properties:
  - bool **os.access**(string path, int mode): returns TRUE if the filename exists and matches the mode query. The mode query can be any of the following constants:

        - **os.F_OK**: test the existence of path
        - **os.R_OK**: tests if path exists and is readable
        - **os.W_OK**: tests if path exists and is writable
        - **os.X_OK**: tests if path exists and is executable
    - **File Operations**: Python also supports file operations such as renaming and deleting files. And of course any shell command can be excecuted via os.system.
        - bool **os.system**(string command): attempts to execute the supplied shell command and returns true if the command executed.
        - bool **chmod**(string path, int mode): Changes the permissions of path to mode. Mode should be defined as an octal (i.e. 0644 or 0777).
        - list **listdir**(string path): Returns a list containing all the files in the current directory. The special entries "." and ".." are not included.
        - bool **mkdir**(string pathname [, int mode]): Makes a directory pathname with permissions mode (e.g. mkdir('new_dir', 0700);)
        - bool **remove**(String filename): Deletes filename
        - bool **rename**(string oldname, string newname): Renames a file
        - bool **symlink**(string target, string link): Creates a symbolic link to the existing target with name link.
    - **Reading Files**: Files can be read by several methods.
        - string **read**([int length]): Reads up to a specified number of bytes from the file into a string. It will read until it encounters EOF or the specified length is reached (default is all data).
        - string **readline**([int length]): Reads one entire line from a file, or up to length bytes, into a string. Reading stops when length bytes have been read or a newline or EOF is reached. A trailing newline character is kept in the string (but may be absent on the last line of the file).
        - list **readlines**([int sizehint]): Reads from a file using readline() until EOF and returns a list containing the lines read. If sizehint is present, whole lines totaling approximately sizehint bytes are read.
    - **EOF**: end-of-file is reached when read or readline returns an empty string. while (s != ""):
      s = f.readline()
      do_something
    - **Writing to files**: Files that have been opened for writing with open can be written to by two methods.
        - void **write**(string string): Writes the contents of string to the file. Does not append a newline character to the string. Only strings can be written so other datatypes must be converted to strings.
        - void **writelines**(list data): Writes a list or array of strings to the file. Newlines will not be added between the elements of the list/array.
    - **Concurrency**: File locking is available through the flock method in the fcntl module. Though be warned, *flock does not work reliably on all operating systems*. Therefore you may want to develop your own semaphores instead. The syntax is: **flock**(fileDescriptor fd, int operation), where the file descriptor can be obtained by calling the **fileno()** method of a file object and operation can be LOCK_SH to acquire a shared lock (reader), LOCK_EX to acquire an exclusive lock (writer), LOCK_UN to release a lock, or LOCK_NB if you don't want flock to block while locking.
    - **Serializing Objects**: An object can be serialized with methods in the pickle module. This will create a string representation of the object that can be stored in a file and later reconstructed into the object. In this way, ints, floats, or any object can be written to a file in addition to strings. If the object is an instance of a class, that class must be defined or imported in the python program that unserializes the object (i.e. if you have an object of type A in a.py, serialize it, write it to a file, and on b.py you read it back in from the file, then class A must be defined in b.py or included via import a to unserialize the object. An easy solution is to put the definition of class A in a file to be imported in both a.py and b.py). Arrays can be serialized as well. If you have an object x, you can serialize it and save it to a file:
      f = open("file.dat","wb")
      pickle.dump(x,f)

```
f.close()
```
It can then be unserialized and restored by:
```
f = open("file.dat","rb")
y = pickle.load(f)
f.close()
```

- **Sockets**: To use sockets in Python, import socket . A server socket can then be opened with:
```
mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mySocket.bind(('', 2727))
mySocket.listen(1)
```
The first line creates a socket object. The second line binds the socket to an address. In this case, '' is a symbolic name meaning localhost and we select port 2727. The address parameter should be in the form of a tuple as shown above. Finally, the third line listens for connections made to a socket. The argument is the maximum number of queued connections. Now that a server socket is open, we need to be able to accept data:
```
conn, addr = mySocket.accept()
print 'Connected with ', addr
while True:
  data = conn.recv(1024)
  if not data: break
  print data
  conn.send("Data received")
conn.close()
```
The **accept**() method accepts a connection and returns a pair (conn, address), where conn is a new socket object usable to send and receive data on the connection and address is the address bound to the socket on the other end (client side) of the connection. We then enter a loop and receive data from the client using the **recv**(bufsize) method. recv returns a string of the data received with a maximum amount of data specified by bufsize. If data is false, we break out of the loop. Otherwise we print the data and use **send**(string) to send a message back to the client.
Now our server is complete, but we need a client-side socket:
```
cSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cSocket.connect(("polaris.astro.ufl.edu","2727"))
```
The first line of course creates a socket object. The second line is similar to **bind** except that it connects to an existing server socket specified by the address. Note that ("localhost", 2727) would be another valid address. Now we need to send and receive data:
```
cSocket.send("Hello world!")
data = cSocket.recv(1024)
cSocket.close()
print data
```
**send** and **recv** work just the same as they do in the server socket. We send data to the server ("Hello world!"), receive the response ("Data received"), close the connection (which causes data to become false on the server program and terminate the loop), and print out the data.

**Examples:**
```
file = open("data/teams.txt","rb")
team = "nonempty"
while (team != ""):
  team = file.readline()
  if (team != ""): print team[:-1]   #get rid of extra newline character
file.close()

file = open("data/teams.txt","rb")
team = file.readlines()
file.close()
```

```
list = ["Florida","Clemson","Duke"]
file = open("data/teams.txt","wb")
for j in list: file.write(j+"\n")
file.close()

import pickle, fcntl
player = Player("J.J. Redick", "Duke", 4)
file = open("data/players.txt", "a")
fcntl.flock(file.fileno(), fcntl.LOCK_EX)
pickle.dump(player, file)
fcntl.flock(file.fileno(), fcntl.LOCK_UN)
file.close()
```

## 12. Images in Python

**FITS files:** Python supports FITS files via the module pyfits. Once this module has been imported, you can read and write FITS files. FITS files are read and stored in and HDUList object, which has two components: header and data. The header is a list-like object and data is usually an array. To read in a FITS file, use

- HDUList **open**(string): open a filename
- **info**(): print a summary of the objects in the file.
  Note that FITS files can have what are called multiple extensions-- multiple images and/or headers in a single file. **info** will list all objects in the file, their name, type, cards (number of entries in the header), dimensions, and format (i.e., Int16 or Float32).

Now that you have a FITS object, you can access its header and data. Since each object within a file can have its own header and data, you would access the primary header as x[0].header and the data as x[0].data.

*Headers:*You can print the entire header by calling the x[0].header.ascardlist() method. You can access individual elements in the header directly by keyword (x[0].header['NAXIS1']) or by index (x[0].header[3]). If you know that a

**Examples:**

```
x = pyfits.open("NGC3031.fits")
x.info()

-> Filename: NGC3031.fits
No.  Name      Type    Cards Dimensions Format
0  PRIMARY PrimaryHDU     6  (530, 530)  UInt8

print x[0].header.ascardlist()

-> SIMPLE  =                    T
BITPIX  =                    8
NAXIS   =                    2
NAXIS1  =                  530
NAXIS2  =                  530
HISTORY Written by XV 3.10a

print x[0].header['NAXIS1']
-> 530
print x[0].header[3]
-> 530

print x[0].data[3,0:5]
-> [11 11 11 9 9]

x[0].data[3,0:3] = array([0,0,0])
print x[0].data[3,0:5]
-> [0 0 0 9 9]

x[0].data += 5 #using numarray to operate on entire array
print x[0].data[3,0:5]
-> [ 5 5 5 14 14]

x.writeto("new_file.fits")
x.close()
```

keyword is already present in the header, you can update its value using the same notation:

```
x[0].header['NAXIS1'] = 265
```

But if the keyword might not be present and you want to add it if it isn't, use the **update**() method instead:

```
x[0].header.update('NAXIS1',265)
```

*Data:* Since the data is an array, you can use any numarray methods on it. The data can thus be accessed using slice notation as well.

```
print shape(x[0].data)
print x[0].data[0:5,0:5]
```

*Writing FITS Files:* Once the data and header have been modified, you can write them back to a new FITS file using **writeto**(string). This writes to a new file and closes that file, but further operations can still be done on the data in memory. Note that if a file exists with the specified name, it will *NOT* be overwritten and an error will be raised. To close the input file, use `x.close()`.

## 13. Guess My Number

Here is the code for Guess My Number in Python, a program that generates a random number between 1 and 100 and asks the user to guess it. It will tell the user if the number is higher or lower after each guess and keep track of the number of guesses.

```
#!/usr/bin/python
import random, math
random.seed()
x = math.floor(random.random()*100)+1
z = 0
b = 0
while x != z:
   b=b+1
   z = input("Guess My Number: ")
   if z < x: print("Higher!")
   if z > x: print("Lower!")
print("Correct! " + str(b) + " tries.")
```

## 14. Python reference

**Python.org: includes an introductory tutorial and a full manual.**

**Numarray homepage: includes a full manual about numarray features.**