Welcome to GKTCS

IT Training .Consultancy . Software Development. Staffing

Surendra Panpaliya

Director, GKTCS Innovations Pvt. Ltd, Pune.

16 + Years of Experience (MCA, PGDCS, BSc. [Electronics] , CCNA)

- Founder,GKTCS Innovations Pvt. Ltd. Pune [Nov 2009 – Till date]
- 500 + Corporate Training for HP, IBM, Cisco,Wipro, Samsung etc.
- **Skills**
  - **Python, Perl,Jython, Django, Android ,**
  - **Ruby, Rail, Cake PHP, LAMP**
  - Data Communication & Networking, CCNA
  - UNIX /Linux Shell Scripting, System Programming
  - CA Siteminder, Autosys, SSO, Service Desk, Service Delivery
- Author of 4 Books
- National Paper Presentation Awards at BARC Mumbai

Agenda

Day 2	Module	Topics
	Module 6	Data Structures
	Module 7	Modules
	Module 8	Object Oriented Programming

Module 6: Data Structures

- **Types**
- **List**
- **Tuple**
- **Dictionary**
- **Sequences**
- **Set**

Types

- Immutable
  - Numbers
  - Strings
  - Tuples
- Mutable
  - Lists
  - Dictionaries
  - Sets
  - Most user defined objects

Lists

```
myList = ["a",5,3.25,2L,4+3j]

anotherList = ["a",myList, ["3","2"]]

anotherList2 = myList + myList
            # = ["a",5,...,"a",5,...]
```

Sequence Types

- Strings and lists are just a special case of “sequence types”.
- These types can be looped over, indexed, sliced, etc.
  - Tuples: (1,2,"b")
  - Lists: [1,"a",3]
  - …
- Anybody can define a new sequence type!

Sequence Operations

- Iteration:

```
for i in myList:
  print i
```

- Numeric indexing:

```
k = myList[3]
```

- Slicing:

```
k = mylist[2:5]
```

Iterating Over Sequences

```
strlist = ["abc", "def", "ghi"]
for item in strlist:
    for char in item:
        print char
```

Sequence Concatenation

```
>>> word = 'Help' + 'Me'
>>> print word
HelpMe
>>> list = ["Hello"] + ["World"]
>>> print list
['Hello', 'World']
```

Sequence Repetition

```
>>> word = "HelpMe"
>>> print '<' + word*5 + '>'
<HelpMeHelpMeHelpMeHelpMeHelpMe
>
>>> ['Hello', 'World'] *3
['Hello', 'World', 'Hello', 'Wo
rld', 'Hello', 'World']
```

Getting Sequence Length

- The len() function gets a sequence's length

```
>>> len( "abc" )
3
>>> len( ["abc","def"] )
2
```

Sequence Indexing

```
>>> str="abc"
>>> print str[0]
a
>>> print str[1]
b
```

Negative Indices

```
>>> word = 'HelpMe'
>>>  # The last character
>>> print word[-1]
e
>>> # The last-but-
one character
>>> print word[-2]
M
```

One Way of Thinking About It

- indices point between characters
- length of a slice is top minus bottom

P	y	t	h	o	n

0 1 2 3 4 5

-6 -5 -4 -3 -2 -1

Sequence Slicing

```
>>> word = "Python"
>>> word[1]
'y'
>>> word[0:2]
'Py'
>>> word[2:4]
'th'
```

- Length of a (positive) slice is top-bottom
- Avoids off-by-one errors

Negative Slicing

```
>>> word="Python"
>>> print word[-1]
n
>>> print word[0:-1]
Pytho
>>> print word[0:-2]
Pyth
>>> print word[2:-2]
th
```

Defaults

- Basic slice form is
  `obj[x:y]`
- If “x” is missing or None, it defaults to “0”.
- If “y” is missing or None, it defaults to the length of the string.

A Useful Invariant

```
>>> word = "Python"
>>> print word[:2] + word[2:]
Python

>>> print word[:3] + word[3:]
Python
```

- This works because Python treats the first slice parameter inclusively and the last exclusively!

Defaults

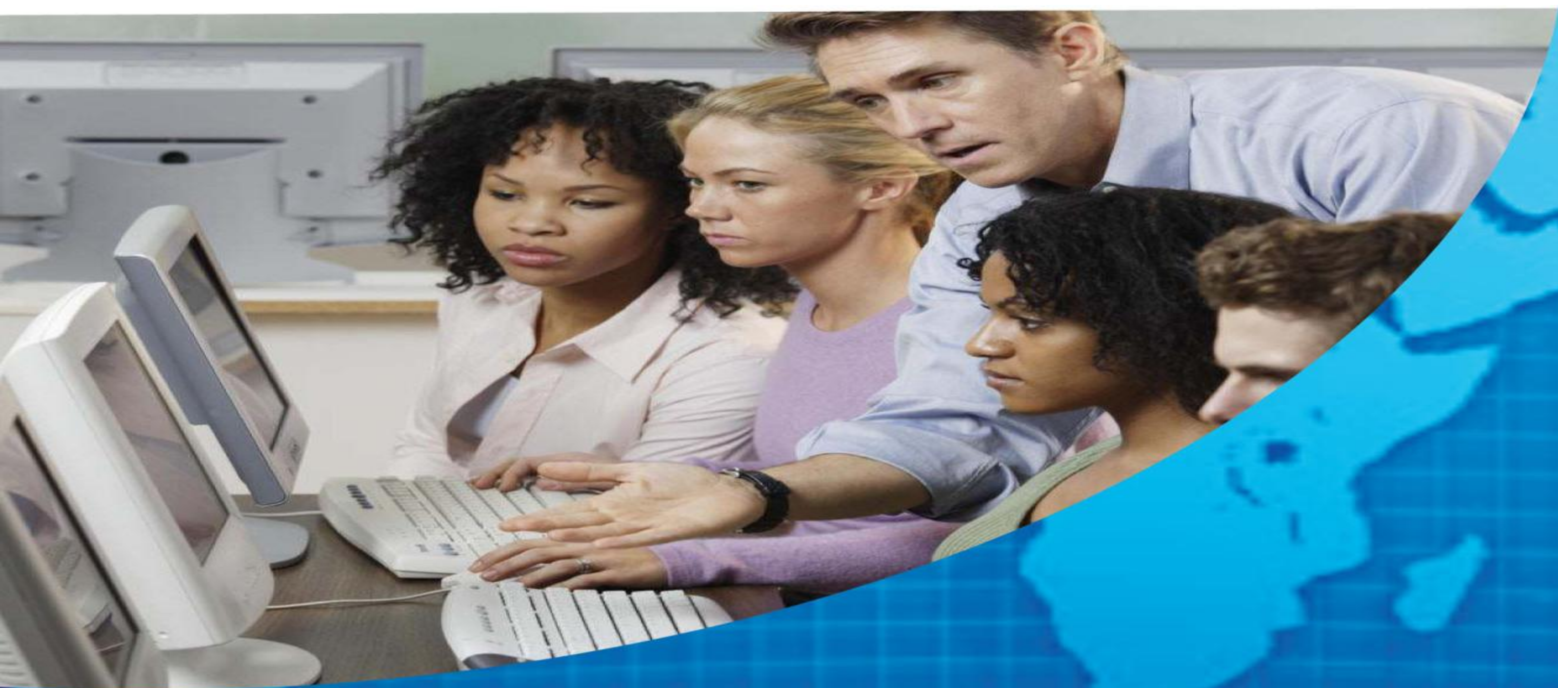
```
>>> word = "HelpMe"
>>> #All but first two characters
>>> print word[2: ]
lpMe
>>> #The last two characters
>>> print word[-2: ]
Me
>>> # First two characters
>>> print word[ :2]
He
>>> # All but last two characters
>>> print word[ :-2]
Help
```

Copying a Sequence

- The start and end can both be inferred:

```
>>> mylist=["a", "b", "c", "d"]
>>> anotherlist = mylist[ : ]
```

Mutability



Bindings

- When we assign a variable, we establish another reference or “binding” to the original value.
- a=b # the same object
- If you change a, you change b!

Lists Are Mutable

- Lists can be changed "in-place"

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> b = a
>>> print b
['spam', 'eggs', 100, 1234]
>>> a[2] = 5.5
>>> print a
['spam', 'eggs', 5.5, 1234]
>>> print b
['spam', 'eggs', 5.5, 1234]
```

Other List Mutations

```
>>> lst= ["a", "b", 5, 3, "g", 1.3]
>>> lst.append( "someitem" )
>>> lst.sort() # sorts lst in-place
>>> lst.reverse() # reverse in-place
>>> del lst[5] # delete sixth item
>>> lst2 = lst[:] # copy list
>>> lst2.reverse()
>>> print lst
['someitem', 'g', 'b', 'a', 5, 1.3]
>>> print lst2
[1.3, 5, 'a', 'b', 'g', 'someitem']
```

Mutation Method in Action

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> b = a
>>> b.append("abc")
>>> print a
['spam', 'eggs', 100, 1234, 'abc']
>>> print b
['spam', 'eggs', 100, 1234, 'abc']
```

Rebinding

```
>>> a=['spam','eggs',100,1234]
>>> b = a[ : ]
>>> a.append("abc")
>>> print a
['spam', 'eggs', 100, 1234, 'abc']
>>> print b
['spam', 'eggs', 100, 1234]
```

- “a” gets a new object that is a slice of b.
- “b” remains bound to the original object.
- “a” has “abc” appeneded.

Strings Are Not Mutable

```
>>> a = "abcdefgh"
>>> a[3]="k"
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item as
signment
>>> a =  a + "qrs"
>>> a
"abcdefghqrs"
```

Tuples

- Immutable list-like objects are called "tuples“

```
>>> tup = (“a”, 1, 5.3, 4)
>>> a[3]="k"
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't have
supt. assignment
```

Fun With Tuples

```
>>> cnum = 1 + 2j
>>> (a,b)=(cnum.real,cnum.imag)
>>> print a; print b
1
2
# old-fashioned swap
# temp=a; a=b; b=temp
>>> (a,b) = (b,a) # pythonic swap
>>> print a; print b
2
1
```

Real Work With Tuples

- Tuples can be returned from functions.
- This makes it easy to return multiple values without defining some kind of class. (unlike Java!)

```
>>> import time
>>> (year,month,day, hrs,mins,secs, day, date, dst) = time.localtime()
```

- Note: there is also a class-based way to deal with date/times.

Tuple Shortcut

- We can usually leave out the parens:

```
>>> j=1,2
>>> j=(1,2) # same as above
>>> a,b = 1,2
>>> a,b = b,a
>>> j=[1,2]
>>> a,b=j
>>> x,y = getXYCoords()
```

Dictionaries

- Serve as a lookup table
- Maps "keys" to "values".
- Keys can be of any immutable type
- Assignment adds or changes members
- keys() method returns keys

Dictionaries

```
>>> mydict={"a":"alpha",
 "b":"bravo","c":"charlie"}
>>> mydict["abc"]=10
>>> mydict[5]="def"
>>> mydict[2.52]=6.71
>>> print mydict
{2.52: 6.71, 5: 'def', 'abc':
 10, 'b': 'bravo', 'c':
 'charlie', 'a': 'alpha'}
```

Constructing Dictionaries

- Dictionaries can be constructed directly as before or by using the “dict” function.

```
>>> list_of_tuples = [("a", "alpha"), ("b", "bravo"), ("c", "charlie")]
>>> mydict = dict(list_of_tuples)
>>> print mydict
{'a': 'alpha', 'c': 'charlie', 'b': 'bravo'}
```

Dictionary Methods

```
>>> mydict={"a":"alpha", "b":"bravo", "c":"charlie"}
>>> mydict.keys()
['a', 'c', 'b']
>>> mydict.values()
['alpha', 'charlie', 'bravo']
>>> mydict.items()
[('a', 'alpha'), ('c', 'charlie'), ('b', 'bravo')]
>>> mydict.clear(); print dict
{}
```

More on Lists

- **`append(`*x*`)`**
  - Add an item to the end of the list; equivalent to a[len(a):] = [*x*].
- **`extend(`*L*`)`**
  - Extend the list by appending all the items in the given list; equivalent to a[len(a):] = *L*.
- **`insert(`*i*`, `*x*`)`**
  - Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, *x*) inserts at the front of the list, and a.insert(len(a), *x*) is equivalent to a.append(*x*).

Functions (cont…)

- **`remove(`*`x`*`)`**
  - Remove the first item from the list whose value is *x*. It is an error if there is no such item.
- **`pop([`*`i`*`])`**
  - Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.

Functions (cont…)

- **`index(x)`**
  - Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.
- **`count(x)`**
  - Return the number of times *x* appears in the list.
- **`sort()`**
  - Sort the items of the list, in place.
- **`reverse()`**
  - Reverse the elements of the list, in place

Using Lists as Stacks

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
```

Using Lists as Queues

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry") # Terry arrives
>>> queue.append("Graham") # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

Functional Programming Tools

- filter(function, sequence)" returns a sequence consisting of those items from the sequence for which function(item) is true.
- If sequence is a string or tuple, the result will be of the same type; otherwise, it is always a list.

Example of filter

```
>>> def f(x): return x % 2 != 0
  and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map function

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343,
512, 729, 1000]
```

A small variant

- More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence

An example

```
>>> seq = range(8)
>>> def add(x, y): return x+y
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

reduce function

- reduce(*function*, *sequence*)" returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on.

An example

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

What if the sequence is empty?

Handling it

```
>>> def sum(seq):
...         def add(x,y): return x+y
...         return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

List Comprehensions

- List comprehensions provide a concise way to create lists without resorting to use of map(), filter() and/or lambda.
- The resulting list definition tends often to be clearer than lists built using those constructs.
- Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses.

Examples

```
>>>freshfruit = [' banana', ' loganberry
  ', 'passion fruit ']
>>>[x.strip() for x in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

Examples

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

The del statement

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
```

Cont…

```
>>> del a[:]
>>> a
[]
```

Sets

- Python also includes a data type for *sets*.
- A set is an unordered collection with no duplicate elements.
- Basic uses include membership testing and eliminating duplicate entries.

Example

```
>>> basket = ['apple',
'orange', 'apple', 'pear',
'orange', 'banana']
>>> fruit = set(basket) #
create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
```

Example

```
>>> 'orange' in fruit # fast
  membership testing
True
>>> 'crabgrass' in fruit
False
```

Set operations

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a set
(['a', 'r', 'b', 'c', 'd'])
>>> a - b
set(['r', 'd', 'b'])
```

Set ops (cont...)

```
>>> a | b # letters in either a or b set
(['a', 'c', 'r', 'd', 'b', 'm',
'z', 'l'])
>>> a & b # letters in both a and b set
(['a', 'c'])
>>> a ^ b # letters in a or b but not both
  set
(['r', 'd', 'b', 'm', 'z', 'l'])
```

Looping through sequence

```
>>> for i, v in
  enumerate(['tic', 'tac',
  'toe']):
... print i, v
...
0 tic
1 tac
2 toe
```

Looping over two sequences

```
>>> questions = ['name', 'quest', 'favorite
  color']
>>> answers = ['lancelot', 'the holy grail',
  'blue']
>>> for q, a in zip(questions, answers):
... print 'What is your %s? It is %s.' % (q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Looping in reversed manner

```
>>> for i in reversed(xrange(1,10,2)):
... print i
...
9
7
5
3
1
```

Sorted looping

```
>>> basket = ['apple', 'orange',
'apple', 'pear', 'orange',
‘banana']
>>> for f in sorted(set(basket)):
... print f
...
apple
banana
orange
pear
```

Comparing Sequences

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' <
  'Python'
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2,
  ('abc', 'a'), 4)
```

Module 7: Modules

Module 7: Modules

- What is module?
- Use of modules
- Import statement
- Global and local module
- Standard library module
- User defined modules
- The dir() Function

What is a Module?

- A file containing some Python code
- OR
- - A .dll (.so on Unix) containing compiled code which follows some guidelines

- A namespace

- The atomic unit of distribution of Python code or Python extensions

A Python Module

```python
def hello_world():
        print "Hello world"
```

- Save this as “mymodule.py” Now we can use it:

```python
>>> import mymodule
>>> mymodule.hello_world()
```

- Or:

```python
>>> from mymodule import hello_world
>>> hello_world()
```

Byte-compiling

- Python automatically byte-compiles modules.
- Next execution does not require compilation.
- .py files get a .pyc in the same directory
- When the .py is updated, the .pyc is updated

- Python is a compiled language but not a native-compiled language: like Java or C#

Importing Modules

Importing Modules

- Use classes & functions defined in another file.
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*.
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

What's the difference?

What gets imported from the file and what name refers to it after it has been imported.

import …

```python
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text “somefile.” to the front of its name:

```python
somefile.className.method(“abc”)
somefile.myFunction(34)
```

*from … import **

```
from somefile import *
```

- *Everything* in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- *Caveat!* Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method(“abc”)
myFunction(34)
```

from … import …

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- After importing *className*, you can just use it without a module prefix. It's brought into the current namespace.
- *Caveat*! This will overwrite the definition of this particular name if it is already defined in the current namespace!

`className.method("abc")`	🡐 This got imported by this command.
`myFunction(34)`	🡐 This one didn't.

Commonly Used Modules

- Some useful modules to import, included with Python:

- Module: sys - Lots of handy stuff.
  - Maxint
- Module: os - OS specific code.
- Module: os.path - Directory processing.

More Commonly Used Modules

- Module: math - Mathematical code.
  - Exponents
  - sqrt
- Module: Random - Random number code.
  - Randrange
  - Uniform
  - Choice
  - Shuffle

Defining your own modules

- **You can save your own code files (modules) and import them into Python.**

Directories for module files

Where does Python look for module files?

- The list of directories in which Python will look for the files to be imported: sys.path

  (Variable named 'path' stored inside the 'sys' module.)

- To add a directory of your own to this list, append it to this list.

```
sys.path.append('/my/new/path')
```

How Python finds modules

- **sys.path** is the path which is traversed when looking for a module (during an import):

```
>>> import sys
>>> print sys.path
['directory1', 'directory2', 'directory3'
, ...]
```

- The search is sequential left to right until success (or end is reached)
- Various ways to change it: PYTHONPATH environment variable, Windows registry tricks, special magic ".pth" files, explicit code that modifies sys.path.

Python Standard Library

OS Module

- Many functions for interacting with the operating system:

```
import os
os.system('copy \\data\\mydata.fil \\backup\\mydata.fil')

curdir = os.getcwd()
os.chdir('\\temp')
```

The os Module

- A set of portable operating-system level commands.
  - **os.remove(path) / os.unlink(path)**
  - **os.rename(src, dest)**
  - **os.rmdir(path)**
  - **os.listdir(path)**
  - **os.chdir(path)**
  - **os.getcwd()**
  - **os.fork()**
  - **os.kill(pid, sig)**
  - **os.getgid()**
  - **os.getuid()**
  - **os.system(command)**
  - **...**
- Note: thin wrapper around POSIX system calls. Relies on underlying OS for functionality.

The os.path Module

```
>>> os.path.expandvars('$HOME/foo')
/home/davida/foo'
>>> os.path.exists('foo')
False
>>> os.path.isdir('/home/davida')
True
>>> os.path.join('/home/davida', 'fo
o', 'bar')
'/home/davida/foo/bar'
>>> os.path.splitext("/home/davida/f
oo.txt")
('/home/davida/foo', '.txt')
```

The sys Module

- Various odds and ends relating to the interpreter:
  - Change module search path
  - Control garbage collection
  - Control where stdin, stdout and stderr go
  - Determine the platform’s maximum native integer size

The sys Module

- The **platform** object in **sys** is a string corresponding to the underlying OS:

```
import sys

if sys.platform == 'win32':
    ...
elif sys.platform == 'linux2':
    ...
elif ...
```

Python 2.3 also includes *platform.py*, which gives much richer information.

File Wildcards

- glob module provides a function for making file lists from directory wildcard searches:

```
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

String Pattern Matching

- The “re” module provides Perl-style regular expression matching.

```
>>> import re
>>> re.findall(r'\bf[a-z]*’,
    'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1',
    'cat in the the hat')
'cat in the hat'
```

math Module

- Access to advanced math functionality:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

math Module

- The Usual Suspects:

acos (x)	asin (x)	atan (x)	atan2(x, y)
ceil (x)	cos (x)	cosh (x)	exp (x)
fabs (x)	floor (x)	fmod (x, y)	*frexp (x)*
hypot (x, y)		ldexp (x, y)	log (x)
log10 (x)	*modf (x)*	pow (x, y)	sin (x)
sinh (x)	sqrt (x)	tan (x)	tanh (x)

- The module also defines two mathematical constants:

pi = 3.14159265359 e = 2.71828182846

- cmath module defines same functions for complex numbers.

```
>>> cmath.log(-2)
(0.69314718056+3.14159265359j)
```

Random

```
>>> import random
>>> random.choice(['apple', 'pear', '
  banana'])
'apple‘
>>> random.random()
0.17970987693706186 # random float
>>> random.randrange(6)
4
# random integer chosen from range(6)
```

Dates and Times

```
# dates are easily constructed and
  formatted
  >>> from datetime import date
  >>> now = date.today()
  >>> now
  datetime.date(2011, 07, 2)
  >>> now.strftime("%m-%d- %y or %d
  %b %Y is a %A on the %d day of %B")

  '12-02-03 or 02Dec 2003 is a Tuesday on
  the 02 day of December'
```

Data Compression

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch’
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist'
```

Performance Measurement

```
>>> from timeit import Timer
>>> setupcode = "import urllib2"
>>> testcode = "urllib2.urlopen('http://www.python.org')"
>>> timer = Timer(testcode, setupcode)
>>> timer.timeit(1)
2.8277779817581177
>>> timer.timeit(5)
14.291818976402283
```

Standard Python modules

- * Using the sys module
- * sys.argv, sys.path, sys.version
- * An overview on __builtin__ and __future__ modules
- * Using the os module
- * Filesystem/directory functions
- * Basic process management functions
- * Recursive directory iteration using os.walk
- * Using the os.path module
- * Determining basename, dirname, path manipulation
- * File type/size/timestamp and other stat determination
- * Using the time and datetime modules
- * Using random, shutil, pprint, hashlib, md5, optparse
- and logging modules

Almost every program uses the sys library

```
>>> import sys
```

Almost every program uses the sys library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
```

Almost every program uses the sys library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
```

Almost every program uses the sys library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
>>> print sys.maxint
2147483647
```

Almost every program uses the sys library

```
>>> import sys
>>> print sys.version
2.7 (r27:82525, Jul  4 2010, 09:01:59)
[MSC v.1500 32 bit (Intel)]
>>> print sys.platform
win32
>>> print sys.maxint
2147483647
>>> print sys.path
['',
 'C:\\WINDOWS\\system32\\python27.zip',
 'C:\\Python27\\DLLs', 'C:\\Python27\\lib',
 'C:\\Python27\\lib\\plat-win',
 'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
```

sys.argv holds command-line arguments

sys.argv holds command-line arguments

Script name is sys.argv[0]

sys.argv holds command-line arguments

Script name is sys.argv[0]

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print i, "" + sys.argv[i] + ""
```

sys.argv holds command-line arguments

Script name is sys.argv[0]

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print i, "" + sys.argv[i] + ""
```

```
$ python echo.py
0 echo.py
$
```

sys.argv holds command-line arguments

Script name is sys.argv[0]

```
# echo.py
import sys
for i in range(len(sys.argv)):
  print i, "" + sys.argv[i] + ""
```

```
$ python echo.py
0 echo.py
$ python echo.py first second
0 echo.py
1 first
2 second
$
```

sys.stdin is *standard input* (e.g., the keyboard)

sys.stdin is *standard input* (e.g., the keyboard)

sys.stdout is *standard output* (e.g., the screen)

sys.stdin is *standard input* (e.g., the keyboard)

sys.stdout is *standard output* (e.g., the screen)

sys.stderr is *standard error* (usually also the screen)

sys.stdin is *standard input* (e.g., the keyboard)

sys.stdout is *standard output* (e.g., the screen)

sys.stderr is *standard error* (usually also the screen)

See the Unix shell lecture for more information

```python
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```python
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
$ python count.py < a.txt
48
$
```

```
# count.py
import sys
if len(sys.argv) == 1:
  count_lines(sys.stdin)
else:
  rd = open(sys.argv[1], 'r')
  count_lines(rd)
  rd.close()
```

```
$ python count.py < a.txt
48
$ python count.py b.txt
227
$
```

The more polite way

```python
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

The more polite way

```
'''Count lines in files.  If no filename arguments given,
read from standard input.'''

import sys

def count_lines(reader):
  '''Return number of lines in text read from reader.'''
  return len(reader.readlines())

if __name__ == '__main__':
  ...as before...
```

If the first statement in a module or function is a string, it is saved as a *docstring*

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```python
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
    adder - Addition utilities.
FUNCTIONS
    add(a, b)
        Add arguments.
>>>
```

If the first statement in a module or function is a string, it is saved as a *docstring*

Used for online (and offline) help

```
# adder.py
'''Addition utilities.'''

def add(a, b):
  '''Add arguments.'''
  return a+b
```

```
>>> import adder
>>> help(adder)
NAME
    adder - Addition utilities.
FUNCTIONS
    add(a, b)
        Add arguments.
>>> help(adder.add)
add(a, b)
       Add arguments.
>>>
```

When Python loads a module, it assigns a value to the module-level variable __name__

When Python loads a module, it assigns a value to the module-level variable __name__

main program
'__main__'

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

When Python loads a module, it assigns a value to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

← Always executed

When Python loads a module, it assigns a value

to the module-level variable __name__

main program	loaded as library
'__main__'	module name

```
...module definitions...

if __name__ == '__main__':
  ...run as main program...
```

Always executed

Only executed when

file run directly

```
# stats.py
'''Useful statistical tools.'''

def average(values):
  '''Return average of values or None if no data.'''
  if values:
    return sum(values) / len(values)
  else:
    return None

if __name__ == '__main__':
  print 'test 1 should be None:', average([])
  print 'test 2 should be 1:', average([1])
  print 'test 3 should be 2:', average([1, 2, 3])
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$
```

```
# test-stats.py
from stats import average
print 'test 4 should be None:', average(set())
print 'test 5 should be -1:', average({0, -1, -2})
```

```
$ python stats.py
test 1 should be None: None
test 2 should be 1: 1
test 3 should be 2: 2
$ python test-stats.py
test 4 should be None: None
test 5 should be -1: -1
$
```

Operating System Interface

OS module provides dozens of functions for interacting with the operating system

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Shutil module provides a higher level interface that is easier to use for file and directory

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

File Wildcards

- The glob module provides a function for making file lists from directory wildcard searches

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Command Line Arguments

- Utility scripts often process command line arguments, these arguments are stored in the sys module's argv attribute as a list
- The following output results from running
  - Python demo.py one two three

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Error Output Redirection and Program Termination

- Stderr is useful for emitting warnings and error messages

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

- The most direct way to terminate a script is to use
  - Sys.ext()

String Pattern Matching

- The re module provides regular expression tools for advanced string processing

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

- When only simple capabilities are needed, string methods are preferred because they are eas

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

Mathematics

- The math modules gives access to the underlying C library functions for floating point math

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

- The random module provides tools for making random selections

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

Internet Access

- Number of modules for accessing the internet
  - Urllib2 for retriving data
  - Smtplib for sending mail

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

Dates and Times

- Datetime modules supplies classes for manipulating dates and times in both simple and complex ways
- Focus of implementation is on efficient

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Data Compression

- Common data archiving and compression formats are directly supported by modules including zlib, gzip, bz2, zipfile, tarfile

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

Performance Measurement

- Python provides measurement tools to determine relative performance of different approaches to the same problem

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Quality Control

- The doctest module provides a tool for scanning a module and validating tests embedded in a program's docstrings

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

- The unittest module allows a more comprehensive set of tests to be maintained in a separate file

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Operating System Interface

- To interact with the OS in python you will want to become familiar with the OS module
- The command “import os” is used for this module
- Useful functions that help in using this module are dir(os) which returns a list of all module functions and help(os) which returns a manual page created from the module’s docstrings

shutil

- **shutil – High-level file operations**

```
from shutil import *
from glob import glob
print 'BEFORE:', glob('shutil_copyfile.*')
copyfile('shutil_copyfile.py', 'shutil_copyfile.py.copy')
print 'AFTER:', glob('shutil_copyfile.*')
```

Operating Systems Interface

- Daily file and directory management tasks can be performed with the shutil module
  - >>> import shutil
  - >>> shutil.copyfile('data.db', 'archive.db')
  - >>> shutil.move('/build/executables', 'installdir')

String Pattern Matching

- The re module provides regular expression tools for string processing.
  - >>> import re
  - >>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
  - ['foot', 'fell', 'fastest']
  - >>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat') 'cat in the hat'
- String methods are easier to read and debug, therefore are preferred when only simple capabilities are needed
  - >>> 'tea for too'.replace('too', 'two')
  - 'tea for two'

Mathematics

- The math module gives access to C library functions for floating point math

  - >>> import math
  - >>> math.cos(math.pi / 4.0)
  - 0.70710678118654757
  - >>> math.log(1024, 2)
  - 10.0

Mathematics

- Random Numbers can be created using the random module
  - >>> import random
  - >>> random.choice(['apple', 'pear', 'banana'])
  - 'apple'
  - >>> random.sample(xrange(100), 10) # sampling without replacement
  - [30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
  - >>> random.random() # random float
  - 0.17970987693706186
  - >>> random.randrange(6) # random integer chosen from range(6)
  - 4

hashlib

- **hashlib – Cryptographic hashes and message digests**
- All of the examples below use the same sample data:

 # hashlib_data.py

- **import hashlib**

lorem = ‘’’ The hashlib module deprecates the separate md5 and sha modules and makes their API consistent. To work with a specific hash algorithm, use the appropriate constructor function to create a hash object. Then you can use the same API to interact with the hash no matter what algorithm is being used.’’’

hashlib

- **import hashlib**
- **from hashlib_data import lorem**
- h = hashlib.md5()
- h.update(lorem)
- **print h.hexdigest()**

optparse

- **optparse – Command line option parser to replace getopt.**

```
import optparse
parser = optparse.OptionParser()
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")
print parser.parse_args(['-a', '-bval', '-c', '3'])
```

logging

- **logging – Report status, error, and informational messages.**

- The logging module defines a standard API for reporting errors and status information from applications and libraries.
- The key benefit of having the logging API provided by a standard library module is that all Python modules
- can participate in logging, so an application’s log can include messages from third-party modules.

logging

```
import logging
LOG_FILENAME = 'logging_example.out'
logging.basicConfig(filename=LOG_FILENAME, level=logging.DEBUG, )
logging.debug('This message should go to the log file')
f = open(LOG_FILENAME, 'rt')
try:
    body = f.read()
finally:
    f.close()
print 'FILE:'
print body
```

Internet Access

- Two of the simplest modules for accessing the internet are urllib2 and smtplib.

- Urllib2 is used for retrieving data
  - >>> import urllib2
  - >>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
  - ... if 'EST' in line or 'EDT' in line: # look for Eastern Time ... print line
Nov. 25, 09:43:32 PM EST

Dates and Times

- The datetime module supplies classes for manipulating dates and times.
- This module supports objects that are timezone aware

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Data Compression

- Common data archiving and compression formats are directly supported by the modules:
- zlib, gzip, bz2, zipfile, and tarfile

- >>> import zlib
- >>> s = 'witch which has which witches wrist watch'
- >>> len(s)
- 41
- >>> t = zlib.compress(s)
- >>> len(t)
- 37
- >>> zlib.decompress(t)
- 'witch which has which witches wrist watch'
- >>> zlib.crc32(s)
- 226805979

Performance Measurement

- Many users wish to know the performance of different approaches to the same problem
- The timeit module quickly can demonstrate performance advantages
  - >>> from timeit import Timer
  - >>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
    0.57535828626024577
  - >>> Timer('a,b = b,a', 'a=1; b=2').timeit()
    0.54962537085770791

csv – Comma-separated value files

- **Reading**
- Use reader() to create a an object for reading data from a CSV file. The reader can be used as an iterator to process the rows of the file in order. For example:

```
import csv
import sys
f = open(sys.argv[1], ‘rt’)
try:
    reader = csv.reader(f)
    for row in reader:
    print row
finally:
    f.close()
```

```
$ python csv_reader.py testdata.csv
```

csv – Comma-separated value files

- **Writing :** Writing CSV files is just as easy as reading them. Use writer() to create an object for writing, then iterate over the rows, using writerow() to print them.

```
import csv
import sys
f = open(sys.argv[1], 'wt')
try:
    writer = csv.writer(f)
    writer.writerow( ('Title 1', 'Title 2', 'Title 3') )
    for i in range(10):
    writer.writerow( (i+1, chr(ord('a') + i), '08/%02d/07' % (i+1)) )
finally:
    f.close()
print open(sys.argv[1], 'rt').read()

$ python csv_writer.py testout.csv
```

Quoting

- There are four different quoting options, defined as constants in the csv module.

1. QUOTE_ALL Quote everything, regardless of type.
2. QUOTE_MINIMAL Quote fields with special characters (anything that would confuse a parser configured with the same dialect and options). This is the default
3. QUOTE_NONNUMERIC Quote all fields that are not integers or floats. When used with the reader, input fields that are not quoted are converted to floats.
4. QUOTE_NONE Do not quote anything on output. When used with the reader, quote characters are included in the field values (normally, they are treated as delimiters and stripped).

Module 8: Object Oriented Programming Concepts

- Introduction to object oriented concepts
- Classes and Objects
- The “self” keyword
- Methods and Attributes
- Constructor and Destructor
- Instance and static member
- Class Inheritance
- Super keyword

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 <u>*all*</u> 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 definition: *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.
- Merely 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 here 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
```

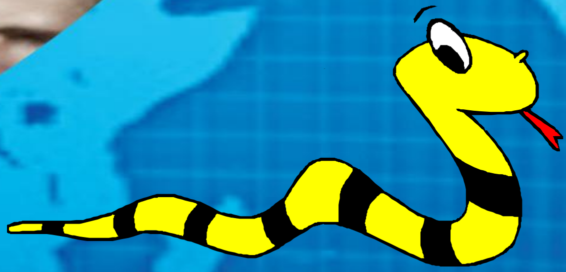
Calling a method:

```
>>> 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 an 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

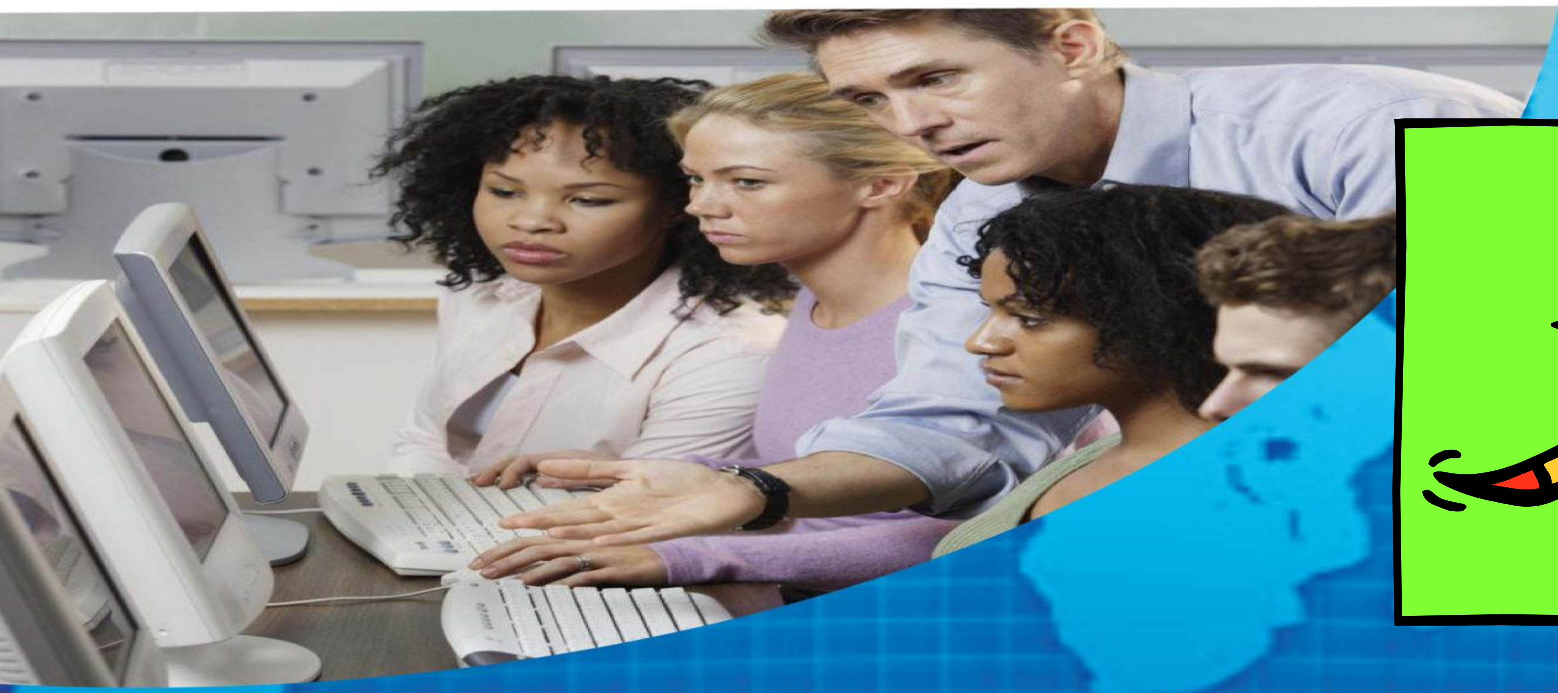
getattr(object_instance, string)

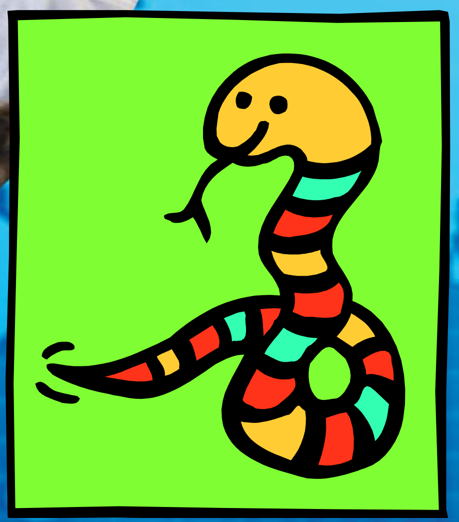
```
>>> f = student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
 <method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")()    # We can call this.
23

>>> getattr(f, "get_birthday")
      # Raises AttributeError – No method exists.
```

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 instances of the class share the same value for it.*
  - Called “static” variables in some languages.
  - Good for
    - class-wide constants
    - 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
  - *outside* of any method

- Since there is one of these attributes *per class* and not one *per instance*, they are accessed using a different notation:
  - Access class attributes using `self.__class__.name` notation.

```
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

Inheritance

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Multiple Inheritance

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Private Variables

- Any identifier of the form __spam
- textually replaced with _classname__spam

Defining own class

```
class Stack:
    def _ _init_ _(self, data):
        self._data = list(data)
    def push(self, item):
        self._data.append(item)
    def pop(self):
        item = self._data[-1]
        del self._data[-1]
        return item
```

Execution

```
>>> thingsToDo = Stack(['write to mom',
  'invite friend over', 'wash the kid'])
>>> thingsToDo.push('do the dishes')
>>> print thingsToDo.pop()
do the dishes
>>> print thingsToDo.pop()
wash the kid
```

UserList

```
from UserList import UserList   # subclass the UserList class
class Stack(UserList):
    push = UserList.append
    def pop(self):
        item = self[-1] # uses _ _getitem_ _
        del self[-1]
        return item
```

Execution

```
>>> thingsToDo = Stack(['write to mom', 'invite friend over', 'wash the kid'])
>>> print thingsToDo
['write to mom', 'invite friend over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:
...       print chore
```

What is an object?

- Objects are collections of data and functions that operate on that data.
- These are bound together so that you can pass an object from one part of your program and they automatically get access to not only the data *attributes* but the *operations* that are available too.
- This combining of data and function is the very essence of Object Oriented Programming and is known as *encapsulation*.

What is a Class?

- Data has various types so objects can have different types.
- These collections of objects with identical characteristics are collectively known as a *class*.
- We can define classes and create *instances* of them, which are the actual objects.
- We can store references to these objects in variables in our programs.

What are polymorphism and inheritance?

- If we have two objects of different classes but which support the same set of messages but with their own corresponding methods.
- We can collect these objects together and treat them identically in our program but the objects will behave differently.
- This ability to behave differently to the same input messages is known as *polymorphism*.

Inheritance

- Inheritance is often used as a mechanism to implement polymorphism.
- A class can *inherit* both attributes and operations from a *parent* or *super* class.
- A new class which is identical to another class in most respects does not need to re-implement all the methods of the existing class,
- it can inherit those capabilities and then *override* those that it wants to do differently

Using a trivial class and made up attributes

```
>>> class null: # a do nothing much class
        ... pass # do nothing statement
        ...
>>> a=null()        # a is an object created by null class.

>>> b=null()        # b is another object

>>> a.c=2           # give object a an attribute c with value 2

 >>> b.d=4          # same kind of deal

>>> a.c+b.d         # add the value attributes and print
 6
```

Methods

A class method is a function that knows its object.

```
>>> class rectangle:
        ... def area(self):
                ... return self.width*self.height
    ...
    >>> a=rectangle()
    >>> a.width=2
    >>> a.height=3
    >>> a.area()
     6
```

Constructor methods

```
# geometry module: constructorex.py
class rectangle:   # rectangle class
    # make a rectangle using top left and bottom right coordinates
    def __init__(self,tl,br):
        self.tl=tl self.br=br                self.width=abs(tl.x-br.x) # width        self.height=abs(tl.y-br.y) # height
    def area(self): # gets area of rectangle    return self.width*self.height
```

Constructor methods cont..

```
class coordinate: # coordinate class
    def __init__(self,x,y):
    # make a coordinate object with a
    #       reference (self), an x and a y
            self.x=x
            self.y=y
    def distance(self,another):
                                        # distance between 2 coordinates
    import math
    xdist=abs(self.x-another.x)
    ydist=abs(self.y-another.y)
 return math.sqrt(xdist**2+ydist**2)
                                        # pythagoras theorem
```

Constructorex Package

- Constructorex Package contains 2 classes, coordinate and rectangle. The following commands import this package, construct 2 coordinates and a rectangle and then calculate the area of the rectangle and the distance between the 2 coordinates:

```
>>> a=constructorex.coordinate(2,3)
>>> b=constructorex.coordinate(5,7)
>>> c=constructorex.rectangle(a,b)
>>> c.area()
12
>>> a.distance(b)
5.0
```

Class data attributes

- Object data attributes are either constructed with or added to each object of the class.
- However, consider the case of a washing machine factory. If each washing machine manufactured from a production line has its own unique serial number, how does the factory know which serial number to give to the next washing machine off the production line ?
- If serial numbers start with 1 and go up by 1 each time, the last issued is the same as the count of machines manufactured.
- For this we use a class attribute. Here is a class which simulates a washing machine, with class attribute: no_made.

washing.py class

- # washing module file: washing.py

```
class machine:
    no_made=0
    def __init__(self):
        machine.no_made+=1
        self.serial=machine.no_made
    def spin(self):
        print “wheeeeeeeeeeeeeeeeeeeeeeee!!"
    def wash(self):
        print "slosh slosh slosh slosh slosh“
    def label(self):
        print "washing machine: %d" % self.serial
```

washing.py class

- The following commands were used to test this class:

```
>>> import washing
>>> a=washing.machine()
 >>> a.wash()
slosh slosh slosh slosh slosh
>>> a.spin()
wheeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee!!
>>> a.serial
 1
>>> a.no_made
 1
>>> b=washing.machine()
 >>> washing.machine.no_made
2
```

washing.py class

```
>>> c=washing.machine()
>>> a.no_made
3
>>> b.serial
2
>>> c.serial
3
>>> c.label()
 washing machine: 3
>>> a.label()
 washing machine: 1
```

Notice that only one copy of the class attribute: no_made exists, but every object has its own serial number.

Inheritance

- Objects inherit behaviour from the classes that construct them through method objects, and class data attributes shared with other objects, as well as having object data of their own.
- Classes can also inherit from superclasses or parent classes. Again both data and functional attributes are inherited.
- Inheritance allows you to have a general class and to create a number of specialised versions of it.
- The child or specialised classes reuse code within the generalised parent class, and add some of their own, to override attributes within the parent, and by adding new ones.
- EX. Here we have a Suprex Deluxe washing machine which does all that our generalised washing machine does, but it has a model attribute, a tumble dry cycle and overides the label method within its parent.

suprex.py

```
#  file: suprex.py
from washing import machine
class deluxe(machine):
                    # deluxe subclasses parent class machine
          model="Suprex Deluxe" # adds an attribute

          def tumble_dry(self): # adds a method
                    print "tumble tumble chug tumble tumble chug"
          def label(self): # overrides a method in parent class
            print "Model: %s Serial No: %d" % (deluxe.model,self.serial)
```

suprex.py

- Here is a test run, with comments after # symbols
- >>> import suprex
- >>> a=suprex.deluxe() # make a suprex deluxe using parent constructor
- >>> b=suprex.machine() # suprex module can construct object of parent class
- >>> a.tumble_dry() # a suprex can tumble dry tumble tumble chug tumble tumble chug
- >>> a.wash() # suprex knows how to wash from parent slosh slosh slosh slosh slosh
- >>> b.wash() # so can an ordinary machine slosh slosh slosh slosh slosh

suprex.py

```
>>> b.tumble_dry() # ordinary machines can't tumble dry Traceback (most recent call last):
    File "<interactive input>", line 1, in ? AttributeError: machine instance has no attribute
    'tumble_dry'
>>> a.serial # a got this object attribute from parent class
 1
>>> b.serial # can the parent also count instances of children ?
2
>>> a.label() # suprex has its own method for this
Model: Suprex Deluxe Serial No. 1
 >>> b.label() # ordinary machines have other code washing machine:
 2
 >>> b.no_made # no_made attribute is accessible through both classes
 2
>>> a.no_made
 2
```

Polymorphism

- This word means something having many forms
- You can override built in names in Python, e.g. by defining your own len() function and localising the override to the scope where this is needed.
- You can override class methods by subclassing if this is useful.
- Python classes also allow you to define methods with special names e.g: __add__(), __del__(), so that you can define what happens when you use + and - operators between your objects.
- Many Python operators can be overriden for class objects.
- In the following example we use the __getitem__ method to override what happens when we index an object:

use of __getitem__ to intercept indexing operations

```
>>> class mystring:
        ... def __getitem__(self,index):
                ... import string
                ... capital=string.upper(self.contents[index])                    ...
    return capital
        ...

>>> a=mystring()
>>> a.contents="abcdefghijklmnopqrstuvwxyz"
>>> a[0] # __getitem__ method overides indexing operator
'A‘
 >>> a[25]
'Z'
>>> a.contents[25] # a.contents was and still is lower case
 'z‘
```

use of __repr__ to intercept print operations

```
>>> class printmachine(suprex.deluxe):
... def __repr__(self):
    ... return "Instance of model: %s Serial Number:            %d" %
    (self.model,self.serial)
...

>>> a=printmachine()
>>> print a
Instance of model: Suprex Deluxe Serial Number: 4
```

Controlling access to class and object attributes

- Up to a point people won't go into houses where they're not welcome.
- If the nature of your project is such that the security needs of your classes go beyond the assumption that unintended forms of access is someone else's problem,
- Python does allow you to code methods called __getattr__ and __setattr__ to intercept read and write access to your class attributes.
- These methods can force consistent attribute behaviour when unknown attributes are referenced or inappropriate access is made to values which should be managed inside the class.

Controlling access to class and object attributes

- **__getattr__(*self, name)***
- Returns a value for an attibute when the name is not an instance attribute nor is it found in any of the parent classes. *name is the attribute name. This method returns the attribute value or raises an* AttributeError exception.
- **__setattr__(*self, name, value)***
- Assigns a value to an attribute. *name is the attribute name, value is the value to assign to it.* Note that if you naively do 'self.name= value' in this method, you will have an infinite recursion of __setattr__() calls.
- To access the internal dictionary of attributes, __dict__, you have to use the following:
- 'self.__dict__[name] = value'.

A class which controls access to its attributes

```
class locked_data:
        max=100 # constant
        def __init__(self,module="WPA4"):
                self.module=module
        def __getattr__(self,attrib):
            if attrib == "title":
                return "Website Programming Applications IV“
            else: # redirect access to unknown attributes
                return    self.module
        def __setattr__(self,attrib,value):
            if attrib in ["module","title"]:
                self.__dict__[attrib]=value
        # List the attributes which can be written to here.
# Have to access through __dict__ to avoid infinite regression
            else:
                raise AttributeError
```

Demonstrates

This test run demonstrates attribute read redirections and write locking-mechanisms :

```
>>> from locked import locked_data
>>> a=locked_data()
>>> a.max # constant class attribute
    100
 >>> a.title # default values
 'Website Programming Applications IV‘
 >>> a.module
     'WPA4‘
 >>> a.thing # __getattr__ returns module for unknown attribute
'WPA4'
```

Demonstrates

>>> a.max=50 # __setattr__ prevents write to class constant Traceback (most recent call last): File "<interactive input>",line 1, in ? File "locked.py", line 15, in __setattr__

raise AttributeError

AttributeError

>>> a.max # a.max stays the same

100

Demonstrates

```
>>> a.thing=42 # can't write to non-existent attribute
 Traceback (most recent call last): File "<interactive input>", line
     1, in ?
 File "locked.py", line 15, in __setattr__ raise AttributeError
AttributeError
>>> a.title="Another" # can change module and/or title
 >>> a.module="new"
>>> a.title
'Another'
>>> a.module
 'new'
```

Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
class ai_student(student):
```

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

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 the 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 ai_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 for 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.

Super keyword

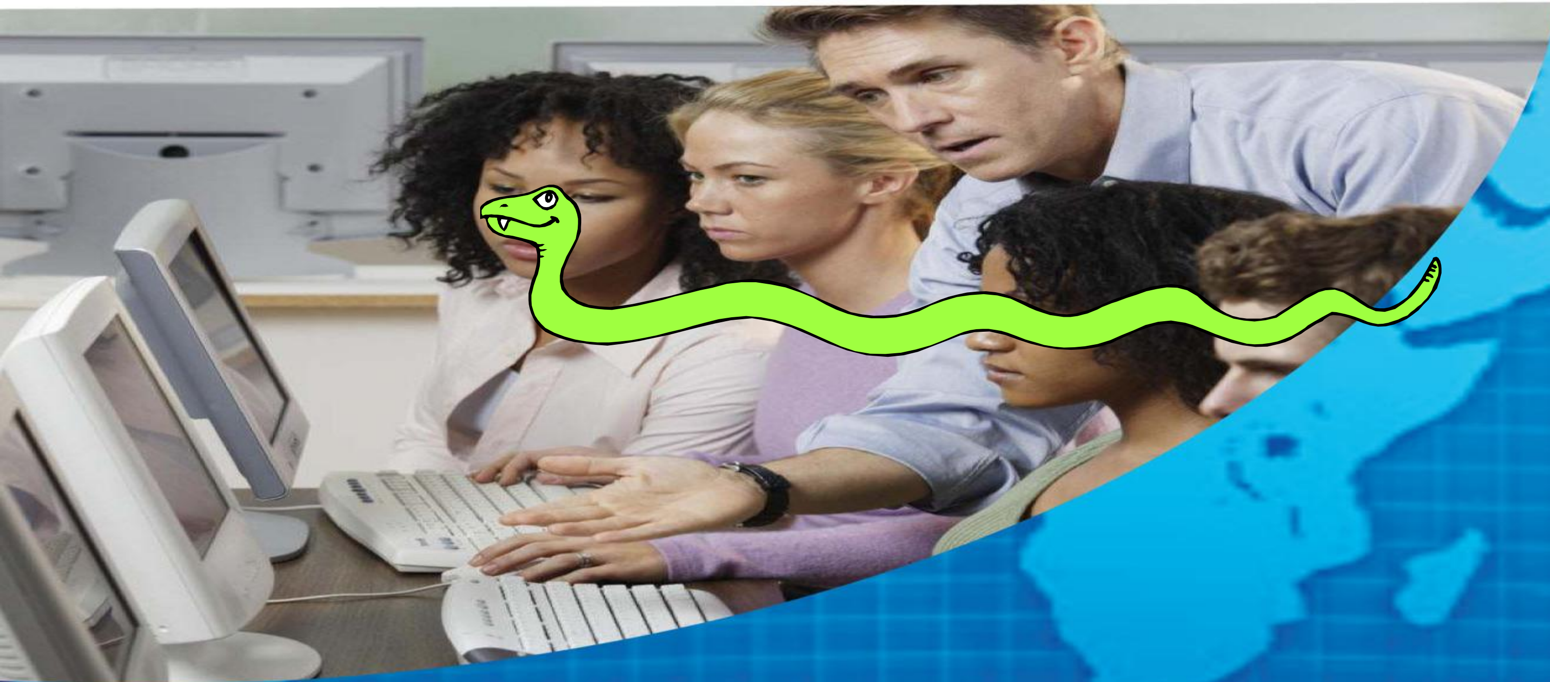
- We can use super() to distinguish between method functions with the same name defined in the superclass and extended in a subclass.
- **super(*type, variable)***
- This will do two things: locate the superclass of the given type, and it then bind the given variable to create an object of the superclass.
- This is often used to call a superclass method from within a subclass: ‘super(classname ,self).method()’

Super keyword

- Here's a template that shows how a subclass __init__() method uses super() to evaluate the superclass __init__() method.

```
class Subclass( Superclass ):
    def __init__( self ):
        super(Subclass,self)__init__()
        # Subclass-specific stuff follows
```

Special Built-In Methods and Attributes



Built-In Members of Classes

- Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The 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

- For example, the method `__repr__` exists for all classes, and you can always redefine it.
- The definition of this method 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.
  - If you type **`f`** at the prompt and hit ENTER, then you are also calling **`__repr__`** to determine what to display to the user 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 Data Items

- These attributes exist for all classes.

  `__doc__` : Variable storing the documentation string for that class.

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

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

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 or method with two leading underscores in its name (but none at the end) is private. It cannot be accessed outside of that 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.

Contact Us on:

G K T C S Innovations Pvt. Ltd.

IT Training, Consultancy, Software Development, Staffing
#11,4th Floor,Sneh Deep, Near Warje Flyover Bridge,
Warje-Malwadi, Pune -411058, Maharashtra, India.
Mobile: +91- 9975072320, 8308761477
Email : surendra@gktcs.com
Web: www.gktcs.com