19CAB04 – PROBLEM SOLVING AND PYTHON PROGRAMMING

By

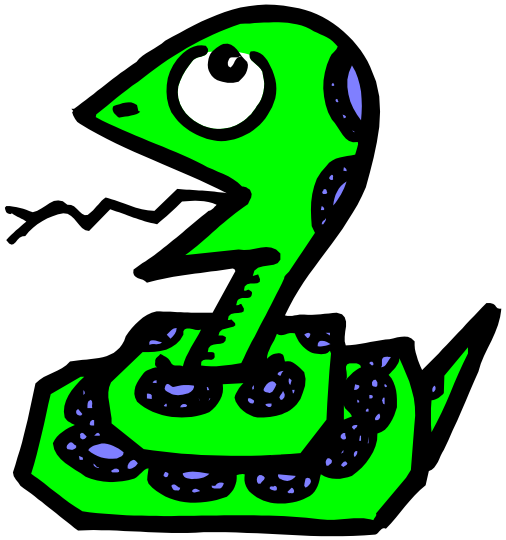
Dr. M. Moorthy, HoD / MCA

Muthayammal Engineering College

MCA - I Semester Regulation 2019

Presentation Overview

- History of Python
- Running Python and Output
- Data Types
- Input and File I/O
- Control Flow
- Functions
- Working with MySQL d/b



Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Hello World

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

```
>>> 'hello world!'
'hello world!'
```

Python Overview

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

Enough to Understand the Code

- Indentation matters to code meaning
  - Block structure indicated by indentation

- First assignment to a variable creates it
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- Assignment is `=` and comparison is `==`

- For numbers `+ - * / %` are as expected
  - Special use of **+** for string concatenation and **%** for string formatting (as in C's printf)

- Logical operators are words (`and, or, not`) *not* symbols

- The basic printing command is `print`

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines

•Use a newline to end a line of code

Use \ when must go to next line prematurely

•No braces {} to mark blocks of code, use *consistent* indentation instead

- First line with *less* indentation is outside of the block
- First line with *more* indentation starts a nested block

•Colons start of a new block in many constructs, e.g. function definitions, then clauses

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

```
and, assert, break, class, continue,
def, del, elif, else, except, exec,
finally, for, from, global, if,
import, in, is, lambda, not, or,
pass, print, raise, return, try,
while
```

Naming conventions

The Python community has these recommend-ed naming conventions

- joined_lower for functions, methods and, attributes
- joined_lower or ALL_CAPS for constants
- StudlyCaps for classes
- camelCase only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

The print Statement

- Elements separated by commas print with a space between them

- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including functions and classes
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

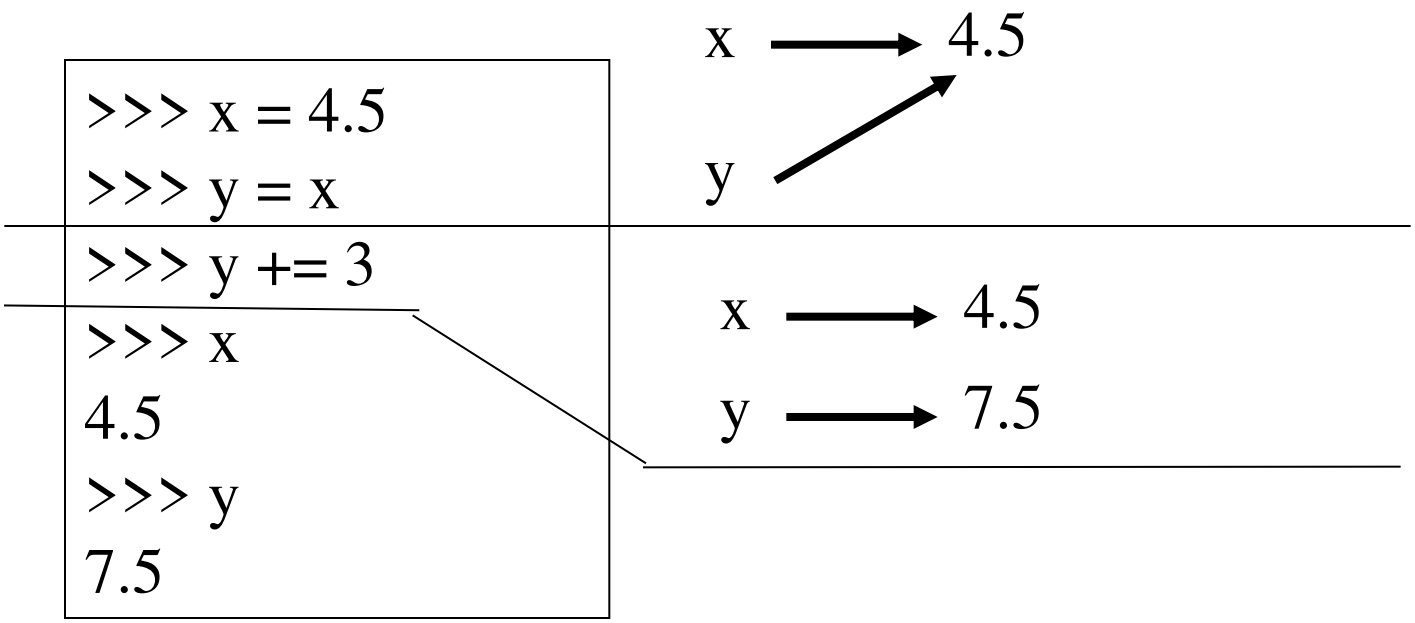
```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

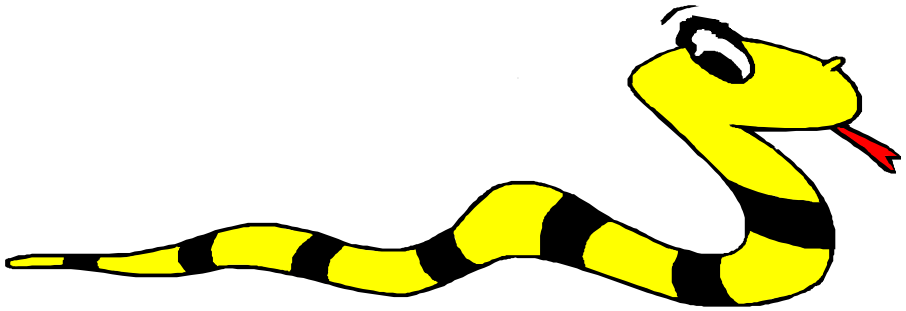
Numbers are *immutable*



String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```



String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)‘
>>> print s
And me too!
though I am much longer
than the others :)‘
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of 1st occurrence
3


>>> li.count('b')  # number of occurrences
2
>>> li.remove('b') # remove 1st occurrence
>>> li
 ['a', 'c', 'b']
```

Operations on Lists Only (Contd..)

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Summary: Tuples vs. Lists

- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and mehtods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print "What year were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" % (name, 2011 - birthyear)
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

Files: Input

inflobj = open('data', 'r')	Open the file 'data' for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable

- Python code files can be read into the interpreter using the **import** statement

Moving to Files

- In order to be able to find a module called myscripts.py, the interpreter scans the list sys.path of directory names.
- The module must be in one of those directories.

```
>>> import sys
>>> sys.path
['C:\\Python26\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python26.zip',
'C:\\Python26\\DLLs', 'C:\\Python26\\lib', 'C:\\Python26\\lib\\plat-win',
'C:\\Python26\\lib\\lib-tk', 'C:\\Python26', 'C:\\Python26\\lib\\site-packages']
>>> import myscripts
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import myscripts.py
ImportError: No module named myscripts.py
```

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions

- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)

- This **forces** the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print 'y = ',
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

```
~: python whileelse2.py
1
```

whileelse2.py

For Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :
    print x
```

```
~: python forloop1.py
1
7
13
2
```

forloop2.py

```
for x in range(5) :
    print x
```

```
~: python forloop2.py
0
1
2
3
4
```

range(N) generates a list of numbers [0,1, …, n-1]

For Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print x
    break
else :
    print 'i got here'
```

elseforloop.py

```
~: python elseforloop.py
1
```

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

funcasparam.py

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

highorder.py

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
~: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Modules: Imports

import mymodule	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
from mymodule import x	Imports x from mymodule right into this namespace
from mymodule import *	Imports all elements of mymodule into this namespace

Working with MySQL Database

- **What is MySQLdb?**

  MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

- The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.
- You can choose the right database for your application. Python Database API supports a wide range of database servers such as −

  GadFly
  mSQL
  MySQL
  PostgreSQL
  Microsoft SQL Server 2000
  Informix
  Interbase
  Oracle
  Sybase

Working with MySQL Database (Contd…)

- Here is the list of available Python database interfaces: Python Database Interfaces and APIs .You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

- The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following:
  - Importing the API module.
  - Acquiring a connection with the database.
  - Issuing SQL statements and stored procedures.
  - Closing the connection

Working with MySQL Database (Contd…)

```
#import database module
import MySQLdb
# Open database connection
db = MySQLdb.connect("127.0.0.1","root","root","mydb" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query
sql = "SELECT * FROM EMPLOYEE"
try:
 # Execute the SQL command
 cursor.execute(sql)

 # Fetch all the rows in a list of lists.
 results = cursor.fetchall()

  for row in results:
     fname = row[0] ; lname = row[1]; age = row[2]; sex = row[3]; income = row[4]
     # Now print fetched result
     print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
           (fname, lname, age, sex, income )
 except:
 print "Error: unable to fecth data"

# disconnect from server
db.close()
```

THANK YOU