INTRODUCTION TO PYTHON PROGRAMMING

PROGRAMMING OVERVIEW

DEFINITION

Programming is the art and science of solving computable problems.

COMMON PROGRAMMING TERMS

- Program
- Software/Application
- Algorithm
- Syntax and Semantics
- Software Development Life Cycle
- Programming Language
- Variables
- Keywords

PROGRAMMING OVERVIEW

ALGORITHM

An algorithm is a step by step method of solving a problem. It is commonly used for data processing, calculation and other related computer and mathematical operations.

PROGRAM

A program is a set of instruction written in a programming language to solve a particular problem.

SOFTWARE

A software is a collection programs written in one or more programming laguage to a problem or set of problems

PYTHON OPERATORS

ARITHMETIC OPERATORS (a=10, b=20)

Operator	Description	Example
+ Addition		a + b = 30
- Subtraction		a – b = -10
* Multiplication		a * b = 200
/ Division		b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

PYTHON OPERATORS

ASSIGNMENT OPERATORS (a=10, b=20)

Operator	Description	Example
=		c = a + b assigns value of a + b into c
+= Add AND		c += a is equivalent to c = c + a
-= Subtract AND		c -= a is equivalent to c = c - a
*= Multiply AND		c *= a is equivalent to c = c * a
/= Divide AND	Divides left hand operand by right hand operand and returns remainder	c /= a is equivalent to c = c / a

PYTHON OPERATORS

ASSIGNMENT OPERATORS (a=10, b=20)

Operator	Description	Example
%= Modulus AND		c %= a is equivalent to c = c % a
**= Exponent AND		c **= a is equivalent to c = c ** a
//= Floor Division		c //= a is equivalent to c = c // a

PYTHON OPERATORS

LOGICAL OPERATORS (a=true, b=true)

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

PYTHON OPERATORS

MEMBERSHIP OPERATORS

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

PYTHON OPERATORS

IDENTITY OPERATORS

Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
Is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

PYTHON BUILT-IN FUNCTIONS/METHODS

COMMON PYTHON BUILT-IN FUNCTION

- **print()** Prints to the standard output device
- **abs()** Returns the absolute value of a number
- **min()** Returns the smallest item in an iterable
- **max()** Returns the largest item in an iterable
- **len()** Returns the length of an object
- **open()** Opens a file and returns a file object
- **type()** Returns the type of an object
- **float()** Returns a floating point number
- **str()** Returns a string object
- **int()** Returns an integer number

PYTHON BUILT-IN FUNCTIONS/METHODS

COMMON PYTHON BUILT-IN FUNCTION

- **range()** Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
- **reversed()** Returns a reversed iterator
- **round()** Rounds a numbers
- **sorted()** Returns a sorted list
- **Sum()** Sums the items of an iterator
- **dict()** Returns a dictionary (Array)
- **list()** Returns a list
- **tuple()** Returns a tuple
- **set()** Returns a new set object
- **bool()** Returns the boolean value of the specified object
- **map()** Returns the specified iterator with the specified function applied to each item

PYTHON BUILT-IN FUNCTIONS/METHODS

COMMON STRING METHODS

- **lower()** returns the lowercase version of a string.
- **upper()** returns the uppercase version of a string.
- **strip()** if the string has whitespaces at the beginning or at the end, it removes it. E.g a = ' Mug ' a.strip() => a = ‘Mug’
- **replace()** replaces a given string with another text. Note, that it’s case sensitive. E.g a = 'muh' a.replace('h','g') => a = ‘mug’
- **Split()** splits your string into a list. Your argument specifies the delimiter. E.g a = 'Hello World' a.split(' ') => [‘Hello’, ‘World’]
- **join()** It joins elements of a list into one string. You can specify the delimiter again. (reverse for split() function) e,g a = ['Hello', 'World'] ' '.join(a) => a = 'Hello World‘
- **swapcase()** Swaps cases, lower case becomes upper case and vice versa

PYTHON BUILT-IN FUNCTIONS/METHODS

COMMON STRING METHODS

- **startswith()** Returns true if the string starts with the specified value
- **endswith()** Returns true if the string ends with the specified value
- **find()** Searches the string for a specified value and returns the position of where it was found
- **capitalize()** Converts the first character to upper case
- **count()** Returns the number of times a specified value occurs in a string
- **index()** Searches the string for a specified value and returns the position of where it was found
- **isdigit()** Returns True if all characters in the string are digits
- **islower()** Returns True if all characters in the string are lower case
- **isspace()** Returns True if all characters in the string are whitespaces
- **title()** Converts the first character of each word to upper case
- **encode()** Returns an encoded version of the string

CONDITIONAL

```
IF/ELSE CONSTRUCT
x = int(input("Please enter an integer: "))

 if x < 0:
    x = 0
    print('Negative changed to zero')
 elif x == 0:
   print('Zero')
 elif x == 1:
   print('Single')
 else:
   print('More')
```

CONTROL STRUCTURE/LOOP

There may be a situation when you need to execute a block of code several number of times.

A loop statement allows us to execute a statement or group of statements multiple times.

FOR LOOP

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

e.g

```
for x in "banana":
  print(x)
```

CONTROL STRUCTURE/LOOP

WHILE LOOP

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

e.g

```
count = 0
while (count < 9):
    print 'The count is:', count
    count = count + 1
print "Good bye!"
```

CONTROL STRUCTURE/LOOP

BREAK STATEMENT

With the break statement we can stop the loop even if the loop condition is true:

e.g

```
animals = [“cow", “goat", “dog"]
for x in animals:
  if x == “goat":
    break
  print(x)
```

or

```
i = 1
while i < 7:
  if i == 3:
    break
  print(i)
  i += 1
```

CONTROL STRUCTURE/LOOP

CONTINUE STATEMENT

With the continue statement we can stop the current iteration of the loop, and continue with the next:

e.g

```
animals = [“cow", “goat", “dog"]
for x in animals:
  print(x)
  if x == “goat":
     continue
```

or

```
i = 1
while i < 7:
   print(i)
   if i == 3:
      continue
   i += 1
```

CONTROL STRUCTURE/LOOP

PASS STATEMENT

- It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes.

The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) –

```
for letter in 'Python':
     if letter == 'h':
        pass
        print 'This is pass block'
     print 'Current Letter :', letter
print "Good bye!"
```

PYTHON DATA STRUCTURES

DATA STRUCTURE

Data structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. It is about rendering data elements in terms of some relationship, for better organization and storage.

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

PYTHON DATA STRUCTURES

LIST

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

list1 = ['physics', 'chemistry', 1997, 2000] OR list1 = list('physics', 'chemistry', 1997, 2000)

list2 = [1, 2, 3, 4, 5] OR list2 = list(1, 2, 3, 4, 5)

list3 = ["a", "b", "c", "d"]

PYTHON BUILT-IN FUNCTIONS/METHODS

COMMON LIST METHODS

- **append()** This method adds an element to the end of our list. a = [1, 4, 6] a.append(4) => a = [1, 4, 6, 4]
- **remove()** Removes the first item with the specified value
- **count()** Returns the number of elements with the specified value
- **clear()** Removes all the elements from the list
- **copy()** Returns a copy of the list
- **extend()** Add the elements of a list (or any iterable), to the end of the current list
- **insert()** Adds an element at the specified position
- **index()** Returns the index of the first element with the specified value
- **pop()** Removes the element at the specified position
- **reverse()** Reverses the order of the list
- **sort()** Sorts the list (This method accept 2 optional parameters: key and reverse)

PYTHON BUILT-IN FUNCTIONS/METHODS

LIST COMPREHENSION

- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
squares = [ ]
 for x in range(10):
    squares.append(x**2)
OR
squares = [x**2 for x in range(10)]
```

PYTHON BUILT-IN FUNCTIONS/METHODS

LIST COMPREHENSION

```
combs = [ ]
 for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))
```

OR

```
combs  = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

PYTHON DATA STRUCTURES

LIST SLICING

```
combs = [1, 2, 3, 4, 5, 6, 7, 8, 9 ]
 combs[1:]
  combs[2:6]
  combs[-len(combs):]
```

PYTHON DATA STRUCTURES

WHEN TO USE LIST

- When you need a mixed collection of data all in one place.
- When the data needs to be ordered.
- When your data requires the ability to be changed or extended. Remember, lists are mutable.
- When you don't require data to be indexed by a custom value. Lists are numerically indexed and to retrieve an element, you must know its numeric position in the list.
- When you need a stack or a queue. Lists can be easily manipulated by appending/removing elements from the beginning/end of the list.
- When your data doesn't have to be unique. For that, you would use sets.

PYTHON DATA STRUCTURES

PYTHON DICTIONARY

- A dictionary is a collection of key-value pairs which is ordered, changeable and indexed.
- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

E.g

```
dict = {'Name': ‘Sam', 'Age': 12, ‘Sex': ‘Male'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

Accessing a key that is not declared throws an error (KeyError), e.g

```
print "dict['Address']: ", dict[‘Address']
```

PYTHON DATA STRUCTURES

PYTHON DICTIONARY

UPDATING A DICTIONARY

```
dict = {'Name': 'Sam', 'Age': 12, 'Sex': 'Male'}
dict['Age'] = 15;       or   dict.get('Age')                    # update existing entry
dict['Subjects'] = ['Maths', 'English', 'Agric'];               # Add new entry
dict['Address'] = ''23 Solomon Road, Obalende, Lagos";          # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['Subjects']: ", dict['Subjects']
print "dict['Address']: ", dict['Address']
```

(a) More than one entry per key not allowed. The last assignment wins when duplicate occur.

(b) Keys must be immutable – numbers, string, tuple but something like ['key'] is not allowed.

PYTHON DATA STRUCTURES

PYTHON DICTIONARY

DELETING DICTIONARY ITEMS OR ENTIRE DICTIONARY

```
dict = {'Name': ‘Sam', 'Age': 12, ‘Sex': ‘Male'}
del dict['Name'];                    # remove entry with key 'Name'
dict.clear();                        # remove all entries in dict
del dict ;                           # delete entire dictionary
```

Try to do the following and the interpreter will throw an error because dict no longer exist :

```
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

PYTHON DATA STRUCTURES

COMMON DICTIONARY METHODS

- **keys()** Returns a list contianing the dictionary's keys
- **values()** Returns a list of all the values in the dictionary
- **clear()** Removes all the elements from the dictionary
- **items()** Returns a list containing the a tuple for each key value pair (eg for x,y in dict.items():)
- **update()** Updates the dictionary with the specified key-value pairs
- **Copy()** Returns a copy of the dictionary
- **get()** Returns the value of the specified key
- **pop()** Removes the element with the specified key
- **popitem()** Removes the last key-value pair
- **fromkeys()** Returns a dictionary with the specified keys and values
- **get()** Returns the value of the specified key

PYTHON DATA STRUCTURES

SET

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

myset = {“orange", "banana", “mango"}

You cannot access items in a set by referring to an index, since sets are unordered and unindexed.

But using the ‘in’ operator you can loop over the items in a set (set is an iterable).

add() Adds an element to the set

clear() Removes all the elements from the set

remove() Removes the specified element

difference() Returns a set containing the difference between two or more sets

union() Return a set containing the union of sets

intersection() Returns a set, that are the intersection of to other sets

update() Update the set with the union of this set and others

PYTHON DATA STRUCTURES

TUPLE

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

mytuple = (“orange", "banana", “mango“)

You can access tuple items using index

print mytuple[0]

You cannot remove or add items to a tuble once it has been declared (tuple is immutable).

mytuple[2] = ‘carrot’ #will throw an error

You can loop over the items in a tuple (tuple is an iterable).

count() Returns the number of times a specified value occurs in a tuple

index() Searches the tuple for a specified value and returns the position of where it was found

PYTHON FUNCTION

FUNCTION

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

ATTRIBUTES OF A FUNCTION

def - The def keyword indicates the start of a function.

functionName - Every function is identified by a name.

() - bracket comes after the function name.

Argument(s) – Arguments are parameter(s) supplied/passed with a function

: - It marks the end of function naming and whatever code that follows it must be indented.

Optional document string - "A string that describe what a function does".

return statement - It a statement that marks the end of a function, value(s) may or may not be returned.

PYTHON FUNCTION

FUNCTION EXAMPLE

```
def addNum(num1, num2):
    “”””This function accepts two number and return their sum”””””
     summed = num1 + num2
     return summed

print(addNum(5,10))

help(my_func)
my_func.__doc__
```

PYTHON FUNCTION

FUNCTION EXAMPLE

```
def factorial_iter(n):
    num = 1
    while n >= 1:
        num = num * n
        n = n - 1
    return num


print(factorial_iter(n))
```

PYTHON FUNCTION

FUNCTION WITH DEFAULT ARGUMENT

```
# Function definition is here
def printinfo( name, age = 35 ):
    "This function prints the info that is passed into the function"
     print "Name: ", name
     print "Age ", age
     return;
# Now you can call printinfo function
print(info( age=50, name="miki" ))
print(info( name="miki" ))
```

PYTHON FUNCTION

FUNCTION WITH VARIABLE-LENGTH ARGUMENT

```
# Function definition is here
def printinfo( arg1, *vartuple ):
      "This prints a variable passed arguments"
        print "Output is: "
        print arg1
        for var in vartuple:
              print var
         return;
 # Now you can call printinfo function
 print(info( 10 ))
print(info( 70, 60, 50 ))
```

PYTHON FUNCTION

RECURSION

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

```
def factorial_sol(n):
    if n<=1:
        return 1
    else:
        return n*factorial_sol(n-1)
n = 5
print (factorial_sol(n))
```

PYTHON FUNCTION

RECURSION

Fetching items of a nexted list example using recursion;

```
def open_item(mylist):
    for item in mylist:
        if type(item) ==list:
            open_item(item)
        else:
            print item

items =["dog", "ball","tv",["remote","laptop",["cat","pet", ['mtn', 'aitel', 'glo']],"man"]]
open_item(items)
```

PYTHON FILE I/O

FILE I/O

File I/O simply means file input and output.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation (creating, reading, updating, and deleting) using a **file** object.

Before you can read or write a file, you have to open it using Python's built-in *open()* function.

This function creates a **file** object, which would be utilized to call other support methods associated with it.

e.g

```
text_file = open("C:/Users/Uche/Desktop/file_data/write_it.txt", "r")
print text_file.read()
text_file.close()
```

PYTHON FILE I/O

FILE ACCESS MODE

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist. This is the default mode.
- ‘‘r+’’ - Opens a file for both reading and writing.
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- ‘‘a+’’ - Opens a file for both appending and reading.
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- ‘‘w+’’ - Opens a file for both writing and reading.
- "x" - Create - Creates the specified file, returns an error if the file exists

PYTHON FILE I/O

FILE METHODS

write(), writelines(), read(), readline() and close()

```
text_file = open(" C:/Users/Uche/Desktop/file_data/write_it.txt ", "w")
text_file.write("\nLine 1\n")
text_file.write("This is Line 2\n")
text_file.write("That makes this Line 3\n")
lines = ["List_Line 1\n",
         "This is List_Line 2\n",
         "That makes this List_Line 3\n"]
text_file.writelines(lines)
text_file.close()
```

FILE I/O

OS MODULE

This module provides an interface to interact with the computer's operating system.

It helps us to perform some operations in the the computer i.e

1 checking if a file exist or if an item is a file

if os.path.exists("demofile.txt"):

if os.path.isfile(f):

2 Renaming a file

os.rename(current_file_name, new_file_name)

3 To remove a file or folder os.remove(file_name) and os.rmdir(directory)

4 To create folder os.mkdir("newdir")

5 To get current working directory os.getcwd()

6 To change directory os.chdir('newdir')

FILE I/O

```
import os
dirName = 'C:\Users\Uche\Desktop\PythonDemos'
def directory_access(folderDirectory):
    folder = os.open(folderDirectory, 'r')
    for fileItems in folder:
        f = os.path.join(folder, fileItems)
        if os.path.isfile(f):
          print fileItems
        else:
           directory_access(fileItems)

directory_access(dirName)
```

DATA BASE MANAGEMENT SYSTEM

Database Management System (DBMS) is a collection of programs which enables its users to access database, manipulate data, reporting / representation of data.

DATABASE

Database is a systematic collection of data. Databases support storage and manipulation of data. Databases make data management easy. E.g a database of a company's sales or employees.

There are different types of DBMS however, we will limit are discuss to RDBMS (Relational DBMS) using mysql. Other example of RDBMS are Postgress, microsoft sql server, oracle, etc.

SQL

Structured Query language (SQL) **pronounced as "S-Q-L" or sometimes as "See-Quel**" is actually the standard language for dealing with Relational Databases.

DATA BASE MANAGEMENT SYSTEM

RDBMS Terminologies

- **Database** – A database is a collection of tables, with related data.
- **Table** – A table is a matrix with data. A table in a database looks like a simple spreadsheet.
- **Column** – One column (data element) contains data of one and the same kind, for example the column postcode.
- **Row** – A row (= tuple, entry or record) is a group of related data, for example the data of one subscription.
- **Primary Key** – A primary key is unique. A key value can not occur twice in one table. With a key, you can only find one row.
- **Foreign Key** – A foreign key is the linking pin between two tables
- **Referential Integrity** – Referential Integrity makes sure that a foreign key value always points to an existing row.
- **Index** – An index in a database resembles an index at the back of a book.

DATA BASE MANAGEMENT SYSTEM

SQL DATATYPES

- CHAR(size) Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters
- VARCHAR(size) Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type
- TEXT Holds a string with a maximum length of 65,535 characters
- TINYINT(size) -128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis
- INT(size) - 2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis
- BIGINT(size) -9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis

DATA BASE MANAGEMENT SYSTEM

SQL DATATYPES

- CHAR(size) Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters
- VARCHAR(size) Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type
- TEXT Holds a string with a maximum length of 65,535 characters
- TINYINT(size) -128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis
- INT(size) - 2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis
- BIGINT(size) -9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis

DATA BASE MANAGEMENT SYSTEM

SQL DATATYPES

- FLOAT(size,d)A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
- DATE()A date. Format: YYYY-MM-DD**Note:** The supported range is from '1000-01-01' to '9999-12-31'
- DATETIME()*A date and time combination. Format: YYYY-MM-DD HH:MI:SS**Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'
- TIMESTAMP()*A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MI:SS**Note:** The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC
- TIME()A time. Format: HH:MI:SS**Note:** The supported range is from '-838:59:59' to '838:59:59'

DATA BASE MANAGEMENT SYSTEM

SQL EXAMPLES

```
CREATE DATABASE  univelsity;

CREATE TABLE student (id INT(8) NOT NULL AUTO_INCREMENT PRIMARY KEY,
stu_name VARCHAR(100),
age INT(3) NOT NULL,
address VARCHAR(100),
email VARCHAR(50),
);
INSERT INTO student (stu_name, age, address, email) VALUES(‘Umu', ‘24', ‘Sabo yaba',
      ‘umu@yahoo.com');
```

DATA BASE MANAGEMENT SYSTEM

SQL EXAMPLES

ALTER TABLE student ADD department VARCHAR(100) ;

UPDATE fellows SET `department` = 'fullstack' WHERE `id` = 1;

RENAME TABLE `student` TO `students`;

SELECT * FROM students ORDER BY `id` desc;
SELECT COUNT(id) FROM students;
SELECT stu_name FROM students;
DELETE FROM students WHERE `id` = 1;

PYTHON OBJECT ORIENTED PROGRAMMING

Object-oriented Programming, or *OOP* for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Another common programming paradigm is *procedural programming*

PYTHON OBJECT ORIENTED PROGRAMMING

OOP TERMINOLOGIES

Class – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

Instance variable – A variable that is defined inside a method and belongs only to the current instance of a class.

Instance – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation – The creation of an instance of a class.

Method – A special kind of function that is defined in a class definition.

PYTHON OBJECT ORIENTED PROGRAMMING

OOP TERMINOLOGIES

Object – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Function overloading – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

Function overiding – This is when a particular function's implementation is available in both parent class and child class, here the implementation of the function in the child class is executed when called on the object of the child class.

Inheritance – The transfer of the characteristics of a class to other classes that are derived from it.

Data member – A class variable or instance variable that holds data associated with a class and its objects

PYTHON OBJECT ORIENTED PROGRAMMING

CLASS

```
class Animal:
    def talk(self):
        print "I have something to say!"
    def walk(self):
        print "Hey! I am walking here!"
    def clothes(self):
        print "I have nice clothes!“
##INITIALIZING
animal = Animal()
animal.talk()
animal.clothes()
```

PYTHON OBJECT ORIENTED PROGRAMMING

INHERITANCE, INSTANCE VARIABLE, METHOD OVERIDING

```
class Duck(Animal):
    def __init__(self, color = "white"):
        self._color = color
    def quack(self):
        print "Quaaaaaack"
    def walk(self):
        print "Walks like a duck."
    def get_color(self):
        return self._color
    def set_color(self,color):
        self._color = color
```

PYTHON OBJECT ORIENTED PROGRAMMING

INHERITANCE, INSTANCE VARIABLE, METHOD OVERIDING

```
class Dog(Animal):
    def clothes(self):
        print "i have brown and white gown"
```

PYTHON OBJECT ORIENTED PROGRAMMING

INHERITANCE, INSTANCE VARIABLE, METHOD OVERIDING

```
def main():
    donald = Duck()
    print donald.get_color()
    donald.set_color("blue")
    print donald.get_color()
    donald.walk()
    donald.clothes()
    fido = Dog()
    fido.walk()
    fido.clothes()

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

```
import pymysql

##   Connect to database
db = pymysql.connect("localhost","root","","python_class" )

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

### Prepare SQL query to INSERT a record into the database.
query = ("SELECT * FROM students")
```

```
##   # Execute the SQL command
cursor.execute(query)

for (id, age, address, email) in cursor:
  print("{}, {}, {}".format(id, age, address, email))
print('We are done!')

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

```
import pymysql

##   Connect to database
db = pymysql.connect("localhost","root","","python_class" )
### prepare a cursor object using cursor() method
cursor = db.cursor()

### Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME,
       LAST_NAME, AGE, SEX, INCOME)
       VALUES ('Mac', 'Mohan', 20, 'M', 2000)"
```

```
try:
##   # Execute the SQL command
   cursor.execute(sql)
##   # Commit your changes in the database
   db.commit()
except:
##   # Rollback in case there is any error
   db.rollback()

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

DATA SCIENCE WITH PYTHON

PYTHON MODULE FOR DATA SCIENCE

NUMPY

NumPy is the fundamental package for scientific computing with Python. It is useful in:

Creating powerful N-dimensional array object

Performing useful linear algebra operation and random number generation. etc

PANDAS

pandas is an open source library, providing high-performance, easy-to-use data structures and data analysis tools for the python programming language.

MATPLOTLIB

Matplotlib is a Python 2D plotting library, It is used for visualization.

NUMPY

```
import  numpy as np
```

ARRAY

```
a = np.array([1,2,3,4])
print(a)
```

ARANGE – works like range()

```
b = np.arange( 4 )
c = np.arange(2, 20, 3)
print(b)
Print(c)
d = np.arange(12).reshape(4,3)          # 2d array
print(d)
```

NUMPY

LINESPACE

```
np.linspace( 0, 2, 9 )          # 9 numbers from 0 to 2
```

RANDOM

```
np.random.rand(3,2)             #2d array of numbers from a uniform distribution over [0, 1)
np.random.random()              #random floats in the half-open interval [0.0, 1.0).
```

BASIC OPERATION

```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
c = a-b
print(c)

d = b**2
print(d)
```

PANDAS

import pandas as pd

Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). E.g

s = pd.Series(data, index=index)

Here, data can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

PANDAS

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
print(s)
print(s.index)
```

Passing a list of values, lets pandas create a default integer index if index is not specified

```
j = pd.Series([7, 3, 5, 6, 2, 8])
print(j)
 Passing  a dict
 d = {'b' : 1, 'a' : 0, 'c' : 2}
 x = pd.Series(d)
 print(x)
#operations ie slicing, mean(), median, max can be performed on series object
#name attribute     rename
```

PANDAS

Dataframe

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects.

It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

PANDAS

Dataframe

From dict of Series or dicts

```
d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']), 'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd']) }
df = pd.DataFrame(d)
print(df)                                    #You view index and columns using  df.index df.columns
```

Getting items base on specified index

```
df = pd.DataFrame(d, index=['d', 'b', 'a'])
print(df)
```

Getting items base on specified index and column

```
 df = pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
print(df)
```

PANDAS

Dataframe

From dict of ndarrays / lists

```
d = {'one' : [1., 2., 3., 4.],  'two' : [4., 3., 2., 1.]}
df = pd.DataFrame(d)
print(df)
```

From a list of dicts

```
data1 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
 df = pd.DataFrame(data1)
 print(df)
```

PANDAS

Dataframe

Column selection, addition, deletion

```
print(df['one'])

df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2

print(df)

del df['two']
three = df.pop('three')
print(df)
```

FIRST PYTHON CODE

FIRST CODE

```
birthYear = float(raw_input("nEnter your birth year, i.e 1990 "))
currentYear = float(raw_input("nEnter current year, i.e 2017 "))
age = currentYear - birthYear
print "Your age is  " + str(age) + "  years"
raw_input("Hit enter button to exit.")
```

LOOP

FOR CONSTRUCT

```
words = ['cat', 'window', 'defenestrate']
for w in words:
   print(w, len(w))
>>cat 3
     window 6
    defenestrate 12
for i in range(5):
    print(i)
>>0 1 2 3 4
```

ML/TENSORFLOW PREREQUISITES

CALCULUS

- $F(x) = 1/1 + e^{-x}$ or
- $F(z) = 1/1 + e^{-z}$ where $z = \theta^{T}x$

PYTHON MODULES

NumPy (for low-level math operations)

```
>>> a = np.arange(6)                    # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>> b = np.arange(12).reshape(4,3)       # 2d array
>>> print(b)
[[ 0  1  2] [ 3  4  5] [ 6  7  8] [ 9 10 11]]
```

NUMPY

```
b = np.arange(12).reshape(4,3)          # 2d array
>>> print(b)
[[ 0  1  2] [ 3  4  5]
[ 6  7  8] [ 9 10 11]]**
```

Array

```
a = np.array([1,2,3,4])

Linespacenp.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )       # useful to evaluate function at lots of
points**
```

NUMPY OPERATIONS

Basic Operation

```
a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
```

THANKS!

Any questions?