Python Tutorial

(Credit goes to www.tutorialspoint.com)

First Python Program

Let us execute the programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt −

```
$ python


Python 3.3.2 (default, Dec 10 2013, 11:35:01)

[GCC 4.6.3] on Linux

Type "help", "copyright", "credits", or "license" for more information.

>>>


On Windows:


Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

>>>
```

Type the following text at the Python prompt and press Enter −

```
>>> print ("Hello, Python!")
```

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result −

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have the extension **.py**. Type the following source code in a test.py file −

```
print ("Hello, Python!")
```

We assume that you have the Python interpreter set in **PATH** variable. Now, try to run this program as follows −

On Linux

```
$ python test.py
```

This produces the following result −

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result −

```
Hello, Python!
```

Let us try another way to execute a Python script in Linux. Here is the modified test.py file −

```
#!/usr/bin/python3

print ("Hello, Python!")
```

We assume that you have Python interpreter available in the /usr/bin directory. Now, try to run this program as follows −

```
$ chmod +x test.py     # This is to make file executable

$./test.py
```

This produces the following result −

Hello, Python!

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return

def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Lines and Indentation

Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
print ("True")

else:
print ("False")
```

However, the following block generates an error –

```
if True:
print ("Answer")
print ("True")

else:
print ("Answer")
print ("False")
```

Thus, in Python all the continuous lines indented with the same number of spaces would form a block. The following example has various statement blocks −

Note − Do not try to understand the logic at this point of time. Just make sure you understood the various blocks even if they are without braces.

```
#!/usr/bin/python3


import sys


try:
   # open file stream
   file = open(file_name, "w")


except IOError:
   print ("There was an error writing to", file_name)
   sys.exit()
print ("Enter '", file_finish,)
print "' When finished"
while file_text != file_finish:
   file_text = raw_input("Enter text: ")
   if file_text == file_finish:
      # close the file
                file.close
      break
   file.write(file_text)
   file.write("\n")
file.close()
file_name = input("Enter filename: ")
if len(file_name) == 0:
```

```
   print ("Next time please enter something")
   sys.exit()


try:
   file = open(file_name, "r")


except IOError:
   print ("There was an error reading file")
   sys.exit()
file_text = file.read()
file.close()
print (file_text)
```

Multi-Line Statements

Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple ('' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
```

```
sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python3



# First comment

print ("Hello, Python!") # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

Python does not have multiple-line commenting feature. You have to comment each line individually as follows −

```
# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt and, the statement saying "Press the enter key to exit", and then waits for the user to take action −

```
#!/usr/bin/python3


input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block. Here is a sample snip using the semicolon −

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

Groups of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example −

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with **-h** −

```
$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d     : debug output from parser (also PYTHONDEBUG=x)

-E     : ignore environment variables (such as PYTHONPATH)

-h     : print this help message and exit


[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic. Let us understand it.

Python Variables

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/python3
```

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string


print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously.

For example −

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String

- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del**statement. The syntax of the **del** statement is −

```
del var1[,var2[,var3[....,varN]]]
```

You can delete a single object or multiple objects by using the **del** statement.

For example −

```
del var
del var_a, var_b
```

Python supports three different numerical types −

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

All integers in Python3 are represented as long integers. Hence, there is no separate number type as long.

Examples

Here are some examples of numbers −

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j

080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python3

str = 'Hello World!'

print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result −

```
Hello World!
H
```

```
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python3

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print (list)          # Prints complete list

print (list[0])       # Prints first element of the list

print (list[1:3])     # Prints elements starting from 2nd till 3rd

print (list[2:])      # Prints elements starting from 3rd element

print (tinylist * 2)  # Prints list two times

print (list + tinylist) # Prints concatenated lists
```

This produces the following result −

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis.

The main difference between lists and tuples are – Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
#!/usr/bin/python3


tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )

tinytuple = (123, 'john')


print (tuple)           # Prints complete tuple

print (tuple[0])        # Prints first element of the tuple

print (tuple[1:3])      # Prints elements starting from 2nd till 3rd

print (tuple[2:])       # Prints elements starting from 3rd element

print (tinytuple * 2)   # Prints tuple two times

print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result –

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
#!/usr/bin/python3


tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )

list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]

tuple[2] = 1000    # Invalid syntax with tuple
```

```
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
#!/usr/bin/python3


dict = {}

dict['one'] = "This is one"

dict[2]     = "This is two"


tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print (dict['one'])       # Prints value for 'one' key

print (dict[2])           # Prints value for 2 key

print (tinydict)          # Prints complete dictionary

print (tinydict.keys())   # Prints all the keys

print (tinydict.values()) # Prints all the values
```

This produces the following result −

```
This is one
This is two
{'name': 'john', 'dept': 'sales', 'code': 6734}
dict_keys(['name', 'dept', 'code'])
dict_values(['john', 'sales', 6734])
```

Dictionaries have no concept of order among the elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type-names as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

S.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. The base specifies the base if x is a string.
2	**float(x)** Converts x to a floating-point number.
3	**complex(real [,imag])** Creates a complex number.
4	**str(x)** Converts object x to a string representation.
5	**repr(x)** Converts object x to an expression string.
6	**eval(str)** Evaluates a string and returns an object.
7	**tuple(s)** Converts s to a tuple.
8	**list(s)** Converts s to a list.

9	**set(s)** Converts s to a set.
10	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
11	**frozenset(s)** Converts s to a frozen set.
12	**chr(x)** Converts an integer to a character.
13	**unichr(x)** Converts an integer to a Unicode character.
14	**ord(x)** Converts a single character to its integer value.
15	**hex(x)** Converts an integer to a hexadecimal string.
16	**oct(x)** Converts an integer to an octal string.

Python Basic Operators

Operators are the constructs, which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called the operands and + is called the operator.

Types of Operator

Python language supports the following types of operators −

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

Python Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then −

Show Example

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1

% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** 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 Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume variable **a** holds the value 10 and variable **b** holds the value 20, then −

Show Example

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.

>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 20, then −

Show Example

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus	It takes modulus using two operands and assign the result to left	c %= a is equivalent to c = c

AND	operand	% a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Python's built-in function bin() can be used to obtain binary representation of an integer number.

The following Bitwise operators are supported by Python language −

Show Example

Operator	Description	Example
& Binary AND	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)

| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Python Logical Operators

The following logical operators are supported by Python language. Assume variable **a** holds True and variable **b** holds False then −

Show Example

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.

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 True.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

Show Example

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 Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below −

Show Example

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 Operators Precedence

The following table lists all operators from highest precedence to the lowest.

Show Example

S.No.	Operator & Description
1	** Exponentiation (raise to the power)
2	**~ + -** Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&**

	Bitwise 'AND'	
7	**^ |** Bitwise exclusive `OR' and regular `OR'	
8	**<= < > >=** Comparison operators	
9	**<> == !=** Equality operators	
10	**= %= /= //= -= += *= **=** Assignment operators	
11	**is is not** Identity operators	
12	**in not in** Membership operators	
13	**not or and** Logical operators	

Class Lab Work:

Basic.py

```
import sys
import time
import findprime
import pandas as pd

def stringops(inputstring):
    firstnum=inputstring.find('_')
    intstr=inputstring[firstnum + 1:len(inputstring)]
    print("intstr first: " + intstr)
    secondnum=intstr.find('_')
    intstr = intstr[secondnum + 1:len(intstr)]
    print("intstr second: " + intstr)
    thirdnum=intstr.find('_')
    intstr = intstr[thirdnum +1 :len(intstr)]
    print("intstr third: " + intstr)
    outputstring=intstr
    return outputstring

# lamda function
multi = lambda arg1, arg2: arg1 * arg2;


if __name__ == '__main__':
    start_time = time.time()
    print("Starting Time %s" % time.ctime())
    s="RAW_input_data_19042020"
    datetimevalue = stringops(s)
    #An Easy way to find that
    pos = s.rfind('_')
    newvalue = s[pos+1:len(s)]
    print("Using Function Method:" + datetimevalue)
    print("Using rfind Method:" + newvalue)
    #Python list - it is kind of arrary but it can contain different data types
    list = [1, 2, 3, 4, 5, 6, 7]
    print(list[4])
    print(list[3:5])
    list[2]=400
    print(list)
    #Python Tuples - it can't be updated
    t=('a','b','c',1,2,3,4)
    print(t)
    print(t[3])
    #Python Dictionary
    dict={'word1':1,'word2':3,'word4':7}
    print(dict['word1'])
    print(dict['word4'])
    print(dict.keys())
    print(dict.values())
    wordcount="I want to learn python. But I am affraid of python"
    words=wordcount.split()
    print(words)
    count = {}
    for word in words:
        if word in count:
```

```
        count[word]=count[word] + 1
    else:
        count[word] = 1
print(count)

a=13
b=a//3
print(b)
if b==1:
    print("First")
elif b==2:
    print("Second")
elif b == 3:
    print("Third")
else:
    print("Forth")

if b==4:
    print("Right Input Fourth")

#Array operations
a=[]
a=[1,2,3,4,5,6]
a[3]=10
print(a)

#Loops
print("For Loop Testing")
for j in a:
    print(j)

#While loop
print("While Loop Testing")
dec=7
result=""
while(dec > 0 ):
    mod = dec % 2
    dec = dec // 2 #Python has a difference in / and // operation
    print(mod)
    result=result + str(int(mod))
    #print(dec)

print("Binary Coversion of " + str(dec) + "is: "+ result)

#File Opreations
try:
    fo=open("D:\\Track\\Datasciencelab\\BDREN\\Software_Binaries\\trips.csv","r+")
    #fo.writelines("\n2020,Chittagong,Kaptai,700,900,1026,1027")
    while True:
        line=fo.readline()
        if not line:
            break;
        print(line) # You may not get the last line because we loaded a file object in earlier time

except IOError:
    print("IO Error Occured")
else:
    print("File operation is done succesfully")
```

```
print(str)
fo.close()

#Checking Lamda Function
print("Checking Lamda Function")
checklamda = multi(10,30)
print(checklamda)

#Understanding Import Function
usernumber = input("Please provide a number to check prime or not: ")

primeif=findprime.primeidentify(int(usernumber))
if(primeif == 1):
   print("Given number is prime")
else:
   print("Given number is not prime")

#Handling External Package
print("Practice using Pandas Package")
csvload = pd.read_csv("D:\\Track\\Datasciencelab\\BDREN\\Software_Binaries\\trips.csv", sep=',',
header=0,
                      index_col=False)
print(csvload)
print(csvload.groupby(['driverid']).count())
print(csvload['tripid'].count())


end_time = time.time()
print("Program Ending Time %s" % time.ctime())
print("%s Durtion in seconds " % (end_time - start_time) )
exit(0)
```

findprime.py

```
import sys
import math

def primeidentify(n):
   check=1
   if (n==0 or n==1 or n <0 ):
      return 0;
   elif (n==2):
      return 1;
   else:
      a=math.sqrt(n)
      i=2
      while(i <=a):

         mod = n % i
         if (mod == 0):
            check = 0;
            break;
         i = i + 1
   return check;
```

Decision Making/Loops related more exercise:

trainingclass.py

```
import fibonacci
import pandas as pd
if __name__=='__main__':
    print("This is my training program")
    a = 20
    p=a%2
    c=a // 2
    print("P=",p)
    if (p !=0 ):
        print("This is an odd number")
    else:
        print("This is an even number")

    while(a > 0):
        print(a);
        a=a-1;

    for i in range(1,c):
        print(i)

    var = "This is our first class on python"
    for letter in var:
        print(letter)

    x = int(input("Please enter a number input:"))
    if x < 0:
        x = 0
        print("Negative changed to zero")
    elif x == 0:
        print("Zero")
    elif x == 1:
        print("Single")
    else:
        print("More than one")
    # Fibonacci series print
    n = int (input("Please provide input to print Nth fibonacci series number:"))
    result = fibonacci.fibonacci_series(n)
    print(result)
```

fibonacci.py

```
def fibonacci_series(n):
    a = 0
    b = 1
    c = a + b
    i = 3
    if (n == 1):
        print(a)
    elif (n == 2):
        print(b)
    elif (n == 3):
        print(c)
    else:
```

```
    print("Calculating fibonacci number")
    while(i < n):
        r1 = c
        r2 = b
        r3 = r1 + r2
        i = i + 1
        c = r2
        b = r3

return r3
```