Python For Beginners

Presenter: Dhiraj Kafle

BSc. CSIT, Tribhuvan University

Software Engineer

Setting up Python:
(In Windows Operating System)

1. First need to install ‘virtualenv’, (if not already installed in the OS)

   ⇒ pip install virtualenv

2. Now, we can create python virtual environment.

   ⇒ virtualenv envname

   eg: virtualenv python_env

3. Finally, activate the environment.

   ⇒ envname\scripts\activate

   eg: python_env\scripts\activate

Setting up Python:
(In Linux/Mac Operating System)

1. First need to install python-venv (if not already installed in the OS)

   ⇒ sudo apt install pythonversion-venv

   eg: sudo apt install python3.10-venv

2. Now, we can create python virtual environment.

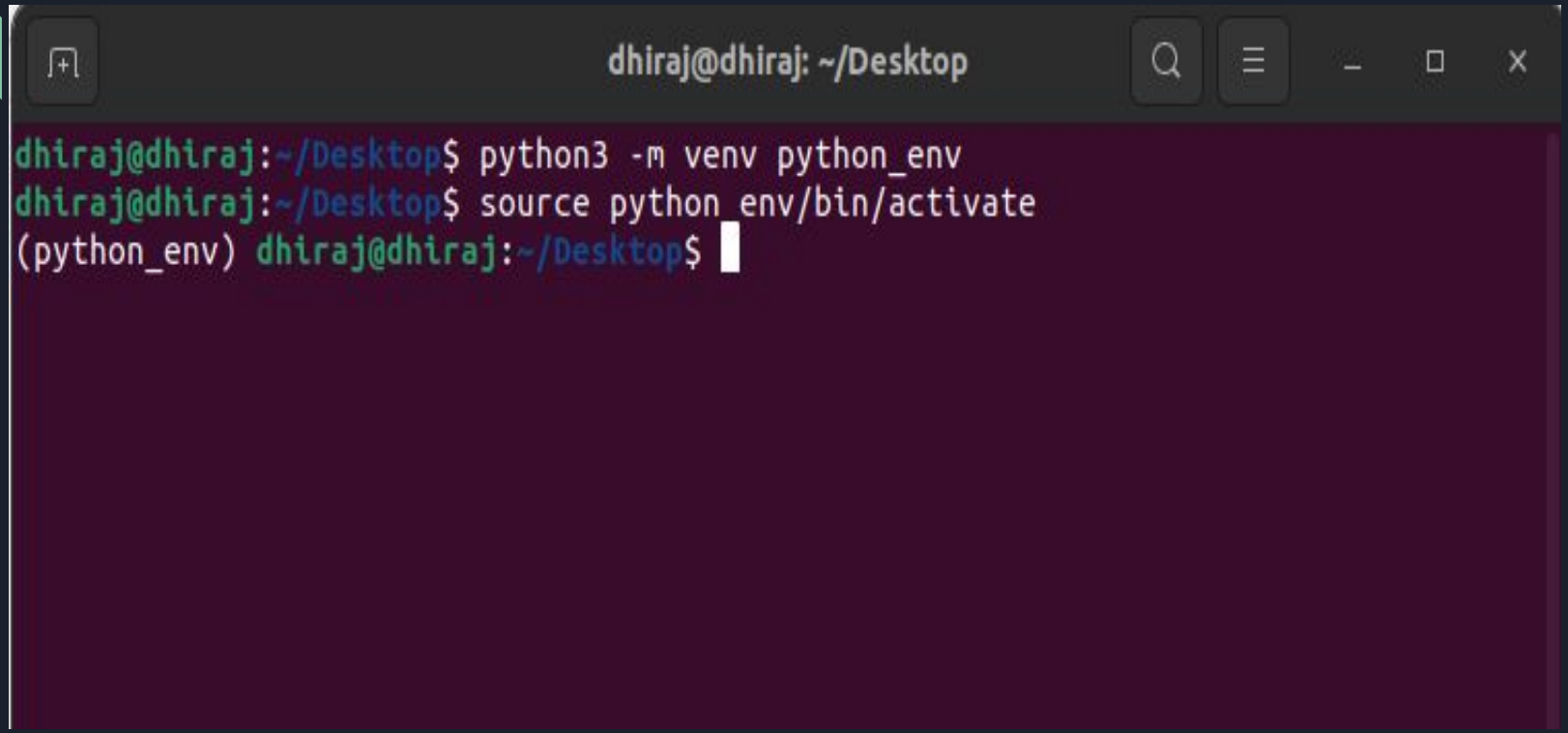
   ⇒ python3 -m venv envname

   eg: python3 -m venv python_env

3. Finally, activate the environment.

   ⇒ source envname/bin/activate

   eg: source python_env/bin/activate


dhiraj@dhiraj: ~/Desktop
dhiraj@dhiraj:~/Desktop$ python3 -m venv python_env
dhiraj@dhiraj:~/Desktop$ source python_env/bin/activate
(python_env) dhiraj@dhiraj:~/Desktop$

Python Basics

1. Data Types and Variables
2. Operators
3. Type Casting / Type Conversion
4. Data Structures

    A. List

    B. Tuples

    C. Sets

    D. Dictionary

5. Conditional Statements
6. Loops
7. Functions (Lambda Functions)
8. Decorators and Generators

Primary(Basic) Data Types

1. Int (Integers) ⇒ eg: 2, 1000, 0, -5 etc
2. Float ⇒ Floating point/decimal numbers. eg: 2.75, -4.99
3. Complex ⇒ Complex numbers. eg: 3 + 4j, 5 - 2j
4. Boolean ⇒ Represents truth/binary values. eg: True or False
5. String ⇒ Sequence of text characters. eg: ‘dhiraj’, “Python”
6. NonType ⇒ Represents absence of a value. eg: None

Variables Creation

NOTE: variables are used to store data / values.

Some variable creation examples:

```
first_name = "Dhiraj"
last_name = '''Kafle'''

a = 2
b = 5.6478

C = True
d= None
```

Operators

Assignment Operator (i.e =) :

```
a = 34
a  -= 12
print(a)
```

```
a += 12
a  /= 2
print(a)
```

Arithmetic Operators :

Operation	Operator	Example
Addition	+	9 + 3 = 12
Subtraction	-	9 - 3 = 6
Multiplication	*	9 * 3 = 27
Division	/	9 / 3 = 3
Floor Division	//	7 // 3 = 2

Bitwise Operators :

```
# bitwise and operation
    a = 7
    b = 6
    print(a & b)


# bitwise  or  operation
    a = 7
    b = 7
    print(a | b)


# bitwise negation operation
    c = 10
    print(~(c))
```

Comparison Operator :

```
b =  (14 <= 7)
print(b)

b =  (14 >= 7)
print(b)

b =  (14 < 7)
print(b)

b =  (14 > 7)
print(b)

b =  (14 == 7)
print(b)

b =  (14 != 7)
print(b)
```

Identity Operator:

It is used to compare the memory location of two or more variables

```
a = 100

b = 100


result = a is b

print(result)
```

Type Casting / Type Conversion

Implicit Type Conversion (on basis of hierarchy):

```
num_int = 123

num_flo = 1.23

num_new = num_int + num_flo

print("datatype of num_new:", type(num_new))
```

NOTE: Hierarchy of float is higher than integer.

Explicit Type Conversion

Float to integer and also Integer to Float :

```
a = 2

b = 4.0

converted_a = float(a)

converted_b = int(b)

sum = converted_a + converted_b

print("sum", sum)

print("Data type of sum", type(sum))
```

Explicit Type Conversion

String to integer :

```
d = "47"
converted_d = int(d)
print(type(converted_d))
```

Integer to String :

```
e = 47
converted_e = str(e)
print(type(converted_e))
```

Data Structures

Data Structure is a special format/ way to store data in a computer so that it can be used/retrieved efficiently.

Various Data Structures in Python:

1. List
2. Tuples
3. Sets
4. Dictionary

NOTE: NOTE: All above data structure's size can be dynamic in python.

List

NOTE: List size can be dynamic.

NOTE: List are mutable (items in the list can be changed/deleted).

SYNTAX for creating empty list:

```
first_list = []

second_list = list()
```

Creating list with items of different types:

```
c = [47, "Engineer", False, 6.9]
```

List Comprehension

NOTE: List comprehension in Python is a concise way to create a new list.

NOTE: It also provide a more readable and often faster alternative to using traditional for-loops.

SYNTAX : [expression for item in list if condition]

EXAMPLE:

```
number_list = [x for x in range(20) if x % 2 == 0]
print(number_list)
```

Tuples

NOTE: Tuples are immutable (meaning items in tuple cannot be changed/deleted).

SYNTAX for creating empty tuple:

```
emptyTuple = ()
emptyTuple2 = tuple()
```

Creating list with items of different types:

```
a = (1, 2, 3, "nepal", 3.4, 6, 9)
print(type(a))
```

Tuple Error Scenario

```
new_tuple = (1, 2, 3, 4, 5)

print(new_tuple)


# t[0] = 34              # throws an error    # Cannot change/update the values of a tuple
```

NOTE: Tuples are immutable.

Dictionary

NOTE: A Python dictionary is a collection of key-value pairs.

Each key is unique, and it maps to a value.

```
## SYNTAX  for creating tuple:
        my_dict = {key: ‘value’}
        new_dict = dict([(key, 'value')])


## Creating dictionary  with items of different types:
myDict = {
   "fast": "In a Quick Manner",
   "dhiraj": "A Programmer",
   "anotherdict": {'dhiraj': 'Player'}
}
```

Dictionary useful methods

To access key, value pair of the dictionary

```
myDict.items()
```

To only access keys of the dictionary

```
myDict.keys()
```

To only access values of the dictionary

```
myDict.values()
```

Dictionary Comprehension

NOTE: Dictionary comprehension is an concise way to create a new dictionary from an iterable.

Without using dictionary comprehension to find square

```
squares = {}
for x in range(6):
   squares[x] = x * x
print(squares)
```

Using Dictionary Comprehension to find square

```
squares = {x: x*x for x in range(6)}
print(squares)
```

Sets

NOTE: A set is an auto sorted collection of items.

NOTE: Every set element is unique (no duplicates).

NOTE: A set object does not support indexing.

SYNTAX for creating empty sets

```
a = set()
```

Creating set of various data types

```
my_set = {1.0, "Hello", (1, 2, 3)}
```

Conditional Statement

If else

```
sub1 = int(input("Enter first subject marks\n"))
sub2 = int(input("Enter second subject marks\n"))
sub3 = int(input("Enter third subject marks\n"))


If (sub1 < 33 or sub2 < 33 or sub3 < 33):
   print("You are fail because you achieved less than 33 in any one of the subjects")
elif (sub1 + sub2 + sub3) / 3 < 40:
   print("You are fail because you achieved less than 40%")
else:
   print("Congratulations ! Successfully Passed the exam.")
```

Conditional Statement

`in` and `not in`

```
list1 = [46, 47, 48]

if 47 in list1:
    print('you are the one’)

if 47 not in list1:
    print('you are unknown’)
```

Loops

for Loop

Note: Better to use for loop when you have a known set of items to iterate over or a definite range of values.

Example

languages = [“C”, "C++", "Java", “Ruby", “PHP”, “Python”, "C#"]

```
for language in languages:
    if language == "Python
        print(language + " found.")
        break
        print("This line never executes")
    print(language + ": is not the language what we are looking for.")
```

Loops

using for loop with range

SYNTAX : for i in range(initialization, stop_at, increment)

Note: By default starts at 0, and increment by 1.

```
# Example 1
for i in range(11):
    print("Iteration :", i)


# Example 2
languages = ["Python", "C++", "Java", "Perl", "C#"]
for i in range(len(languages)):
    print(i, languages[i])
```

Loops

while Loop

Note: Better to use a while loop when you need to keep iterating until a certain condition changes, and you don't know in advance how many iterations you'll need.

Example :

```
response = input("Do you want to continue ? Y/N: ")
while response.lower() == "y":
  response = input("Do you want to continue ? Y/N: ")
```

Note: In python there is no do while loop.

Functions / Methods

NOTE: Function is a block of reusable code that performs a specific task & u can define using ‘def’ keyword, followed by function name, parenthesis and colon.

```
# Example

def greet(name="ram"):                    ## Function Definition

   if name == "ram":

      return "Hi " + name + " ! Welcome."

print(greet())                            ## Function call
```

NOTE: variables inside function definition parentheses are called parameters.

NOTE: variable inside function call parentheses are called arguments.

Lambda Function (one-line function)

NOTE: lambda function is a small anonymous function defined with the lambda keyword.

Lambda function does not include a “return” statement.

Syntax ⇒ lambda arguments: expression

```
## without using lambda function
def doubling(x):
  return x * 2
print(doubling(10))
```

```
## Using lambda function
doubling = lambda x: x * 2
print(doubling(10))
```

Decorators

Decorator is a function that accepts function as argument and also returns function

NOTE: allows you to modify/validate the behavior of any function

EXAMPLE

```
def make_pretty(func):          ## make_pretty() is a custom decorator function
  def inner():                  ## inner() / wrapper() function is automatically called
    func()
  return inner


@make_pretty                    ## decorators are implemented using the @ syntax followed by the decorator function
def ordinary():
  print("Inside Ordinary Function")


ordinary()
```

OOP (Object oriented programming)

class & objects

NOTE: A class is a blueprint for creating objects (instances).

NOTE: an object is an instance of a class.

NOTE: Generally, a class contains, attributes,constructor & custom methods/functions

NOTE: `self` parameter gives the reference of the specific instance

Example:

```
class Dog():                                ## Class Definition
  def __init__(self, naam)                  ## __init__ is the  constructor / instance initializer
        self.name = naam
  def bark(self):                           ## bark() is a method of the class
    print(self.name, 'is barking & his age is ', self.age)


puppy = Dog('Puppy')                        ## object / instance creation
puppy.bark()                                ## calling bark() method using object
```

Class variable

Note: A class variable is a variable that is shared by all instances (objects) of a class.

Note: Class variables are defined within the class definition but outside of any instance methods/constructor.

Example:

```
class  Employee:
        Num_of emps = 0                          ## Num_of_emps is a class variable
        def __init__(self, first, last, pay, raise_amout= None):
            self.first = first
            self.last = last
            Employee.num_of_emps += 1            ## using class variable

          def fullname(self):
            return '{} {}'.format(self.first, self.last)
```

Methods

@classmethod ⇒ if only want to access the class attribute/variable

```
class Employee:

    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

emp_1 = Employee('dhiraj', 'kafle', 50000)
Employee.set_raise_amt(1.05)

print(Employee.raise_amt)
print(emp_1.raise_amt)
```

Methods

@staticmethod ⇒ if you never need to access class variable nor constructor attributes

```python
class Dog:

    dogs = []

    def __init__(self, name):
        self.name = name
        self.dogs.append(self)


    @staticmethod
    def bark(n):
        for _ in range(n):
            print('Bark !')


Dog.bark(5)
```

@property decorator

@property ⇒ allows you to define a `method/function` that can be accessed like an `attribute`.

```
class Employee:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)



emp_1 = Employee('John', 'Doe')

emp_1.first = 'Dhiraj'
emp_1.last = 'Kafle'

print(emp_1.fullname)
```

Magic / Dunder Method

It allow customization of how objects behave in various contexts, like initialization, printing, representation, and arithmetic operations.

```python
class Employee:
    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.email = first + '.' + last + '@email.com'
        self.pay = pay

    def __str__(self):
        return '{}'.format(self.email)

    def __len__(self):
        return len(self.fullname())


emp_1 = Employee('John', 'Doe', 50000)

# for use of __repr__() magic method
print(repr(emp_1))
print(emp_1.__repr__())

# for use of __str__() magic method
print(emp_1)
print(str(emp_1))

# for use of __len__() magic method
print(len(emp_1))
```

Inheritance

Inheritance allows one class (subclass to inherit & extend the properties and methods of another class(parent).

Python supports various types of following inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hybrid Inheritance

Single Inheritance ⇒ a child class (subclass) inherits from a single parent class (superclass)

```python
class Pet:                                  # Pet as parent class
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show(self):
        print(f"I am {self.name} & i am {self.age} years old.")

    def speak(self):
        print("what to speak ??")


# Note : super() references to the parent class

class Dog(Pet):                             # Dog as child class
    def __init__(self, name, age, color):
        super().__init__(name, age)         ## inherting constructor of the parent
        self.color = color

    def show(self):
        print(f"I am {self.name} & i am {self.age} years old & i looks {self.color}.")

    def speak(self):
        print("I can only bark")


p = Pet("Tim", 20)
p.show()
p.speak()

d = Dog("puppy", 5, "grey")
d.show()
d.speak()
```

Multiple Inheritance ⇒ a child class (subclass) inherits from a multiple parent class (superclass)

```
class Freelancer:

    company = "Google"
    level = 0

    def upgradeLevel(self):
        self.level = self.level + 1
        return self.level

class Employee:
    company = "Facebook"
    eCode = 120

    def upgradeLevel(self):
        print('Employee rank is being upgraded')

class Programmer(Freelancer, Employee):
    name = "Rohit"


p = Programmer()

print(p.upgradeLevel())
print(p.company)
```

Multilevel Inheritance ⇒ A class serves as a base class for another class, which in turn serves as a base class for yet another class.

```
class Person:
    country = "Nepal"

    def takeBreath(self):
        print("I am breathing...")


class Employee(Person):
    company = "Facebook"

    def takeBreath(self):
        print("I am an Employee so I am luckily breathing..")


class Programmer(Employee):
    company = "Google"

    def getSalary(self):
        print(f"No salary to programmers")

    def takeBreath(self):
        print("I am a Progarmmer so I am breathing++..")


pr = Programmer()
pr.takeBreath()
print(pr.company)
print(pr.country)
```

Method overriding

It allows a child class to override the implementation of a method that is already defined in its superclass/parent.

Note:Always higher priority for child class method implementation, if not found, then only looks for the method on super class.

```
# method overriding scenario
class Animal:
    def speak(self):
        print("Speaking")

class Dog(Animal):
    def speak(self):
        print("barking")

d =Dog()
d.speak()


# No method overriding scenario
class Animal:
    def speak(self):
        print("Speaking")

class Dog(Animal):
    pass


d = Dog()
d.speak()
```

Thank You !!