

Unit IV: Python packages and OOPS

Introduction to PIP, Installing Packages via PIP, Using Python Packages, Classes, self-variable, Methods, Constructor Method, Inheritance, Overriding Methods.

Introduction to PIP, Installing Packages via PIP

PIP is a package manager for Python, which stands for "Pip Installs Packages." It is the standard package manager for Python that allows you to easily install, manage, and distribute Python packages (also known as libraries or modules). Python packages are collections of pre-written code that can be used to extend the functionality of Python.

What is Python PIP?

Python PIP is the package manager for Python packages. We can use PIP to install packages that do not come with Python. **The basic syntax of PIP commands** in the command prompt is:

```
pip 'arguments'
```

How to Install Python PIP?

Python PIP comes pre-installed on 3.4 or older versions of Python. To check whether PIP is installed or not type the below command in the terminal.

```
pip --version
```

```
C:\Users\arshp>pip --version  
pip 23.2.1 from C:\Users\arshp\AppData\Local\Programs\Python\Python310\lib\site-packages\pip (python 3.10)
```

How to Install Package with Python PIP

We can install additional packages by using the Python pip install command.

Syntax:

```
pip install numpy
```

Example 1: When the required package is not installed.

```
C:\Users\hp>pip install numpy
Collecting numpy
  Using cached numpy-1.24.2-cp39-cp39-win_amd64.whl (14.9 MB)
Installing collected packages: numpy
Successfully installed numpy-1.24.2

C:\Users\hp>
```

Example 2: When the required package is already installed. Using Python PIP to install an existing package

```
C:\Users\hp>pip install numpy
Requirement already satisfied: numpy in c:\users\hp\appdata\local\programs\python\python39\lib\site-packages (1.24.2)

C:\Users\hp>
```

Specifying Package Version using Python PIP

We can also install the package of a specific version by using the below command.

Syntax:

```
pip install package_name==version
```

This will install the package with the specified version

Display Package information using Python PIP

We can use the Python pip show command to display the details of a particular package.

Syntax:

```
pip show numpy
```

Example:

```
C:\Users\arshp>pip show numpy
Name: numpy
Version: 1.25.2
Summary: Fundamental package for array computing in Python
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email:
License: BSD-3-Clause
Location: c:\users\arshp\appdata\local\programs\python\python310\lib\site-packages
Requires:
Required-by: contourpy, matplotlib, pandas
```

Required by shows the packages that require NumPy

Get a list of locally installed Python Modules using Python PIP

The Python pip list command displays a list of packages installed in the system.

Syntax:

pip list

Example:

```
Command Prompt
C:\>pip list
Package                                Version
-----
abs1-py                                0.9.0
alabaster                               0.7.12
anaconda-client                         1.7.2
anaconda-navigator                     1.9.7
anaconda-project                       0.8.3
asn1crypto                             1.0.1
astor                                   0.7.1
astroid                                 2.3.1
astropy                                 3.2.1
atomicwrites                           1.3.0
attrs                                  19.2.0
Babel                                   2.7.0
backcall                               0.1.0
backports.functools-lru-cache          1.5
```

Uninstall Packages with Python PIP

The Python pip uninstall command uninstalls a particular existing package.

Syntax:

```
pip uninstall numpy
```

Example:

```
C:\> Command Prompt
C:\> pip uninstall numpy
Uninstalling numpy-1.16.5:
  Would remove:
    d:\software\anaconda\lib\site-packages\numpy
    d:\software\anaconda\lib\site-packages\numpy-1.16.5-py3.7.egg-info
    d:\software\anaconda\scripts\f2py-script.py
    d:\software\anaconda\scripts\f2py.exe
Proceed (y/n)? y
  Successfully uninstalled numpy-1.16.5
```

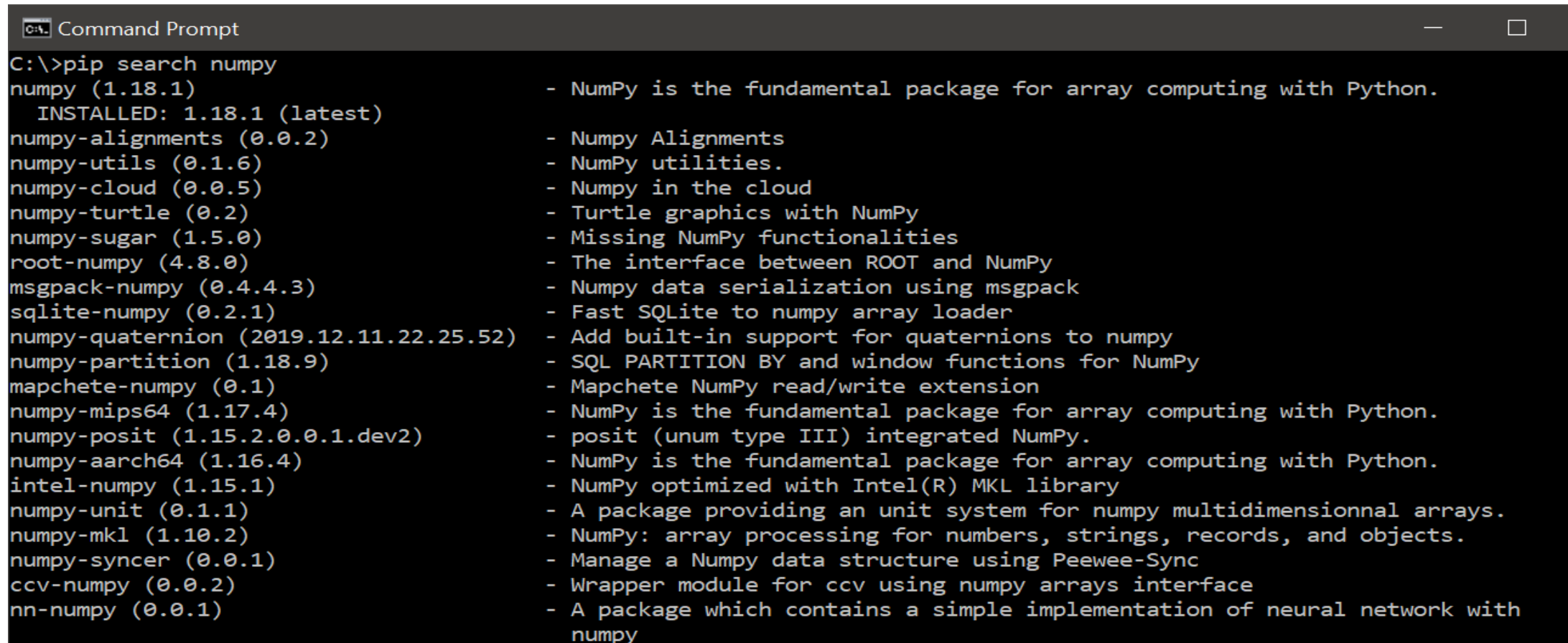
Search Packages with Python PIP

We can search for a particular existing package using the Python pip search command.

Syntax:

pip search numpy

Example:



```
C:\>pip search numpy
numpy (1.18.1)
  INSTALLED: 1.18.1 (latest)
numpy-alignments (0.0.2)
numpy-utls (0.1.6)
numpy-cloud (0.0.5)
numpy-turtle (0.2)
numpy-sugar (1.5.0)
root-numpy (4.8.0)
msgpack-numpy (0.4.4.3)
sqlite-numpy (0.2.1)
numpy-quaternion (2019.12.11.22.25.52)
numpy-partition (1.18.9)
mapchete-numpy (0.1)
numpy-mips64 (1.17.4)
numpy-posit (1.15.2.0.0.1.dev2)
numpy-aarch64 (1.16.4)
intel-numpy (1.15.1)
numpy-unit (0.1.1)
numpy-mkl (1.10.2)
numpy-syncer (0.0.1)
ccv-numpy (0.0.2)
nn-numpy (0.0.1)
```

- NumPy is the fundamental package for array computing with Python.
- Numpy Alignments
- NumPy utilities.
- Numpy in the cloud
- Turtle graphics with NumPy
- Missing NumPy functionalities
- The interface between ROOT and NumPy
- Numpy data serialization using msgpack
- Fast SQLite to numpy array loader
- Add built-in support for quaternions to numpy
- SQL PARTITION BY and window functions for NumPy
- Mapchete NumPy read/write extension
- NumPy is the fundamental package for array computing with Python.
- posit (unum type III) integrated NumPy.
- NumPy is the fundamental package for array computing with Python.
- NumPy optimized with Intel(R) MKL library
- A package providing an unit system for numpy multidimensionnal arrays.
- NumPy: array processing for numbers, strings, records, and objects.
- Manage a Numpy data structure using Peewee-Sync
- Wrapper module for ccv using numpy arrays interface
- A package which contains a simple implementation of neural network with numpy

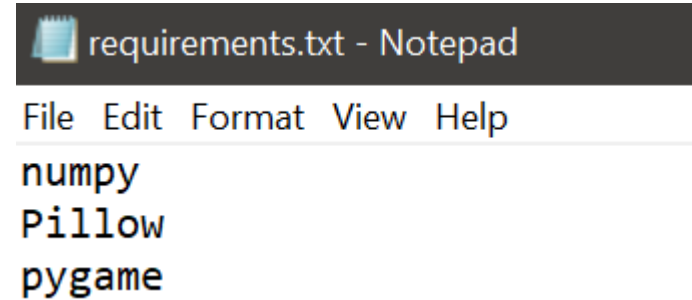
Using Requirement files with Python PIP

Let's suppose you want more than one package then instead of installing each package manually, you can install all the modules in a single go. This can be done by creating a requirements.txt file

Syntax:

```
pip install -r requirements.txt
```

Example:



```
requirements.txt - Notepad
File Edit Format View Help
numpy
Pillow
pygame
```



```
Command Prompt
C:\>pip install -r requirements.txt
Requirement already satisfied: numpy in d:\software\anaconda\lib\site-packages (from -r requirements.txt (line 1)) (1.18.1)
Requirement already satisfied: Pillow in d:\software\anaconda\lib\site-packages (from -r requirements.txt (line 2)) (6.2.0)
Collecting pygame (from -r requirements.txt (line 3))
  Downloading https://files.pythonhosted.org/packages/ed/56/b63ab3724acff69f4080e54c4bc5f55d1fbdeeb19b92b70acf45e88a5908/pygame-1.9.6-cp37-cp37m-win_amd64.whl (4.3MB)
    |████████████████████| 4.3MB 72kB/s
Installing collected packages: pygame
Successfully installed pygame-1.9.6
```

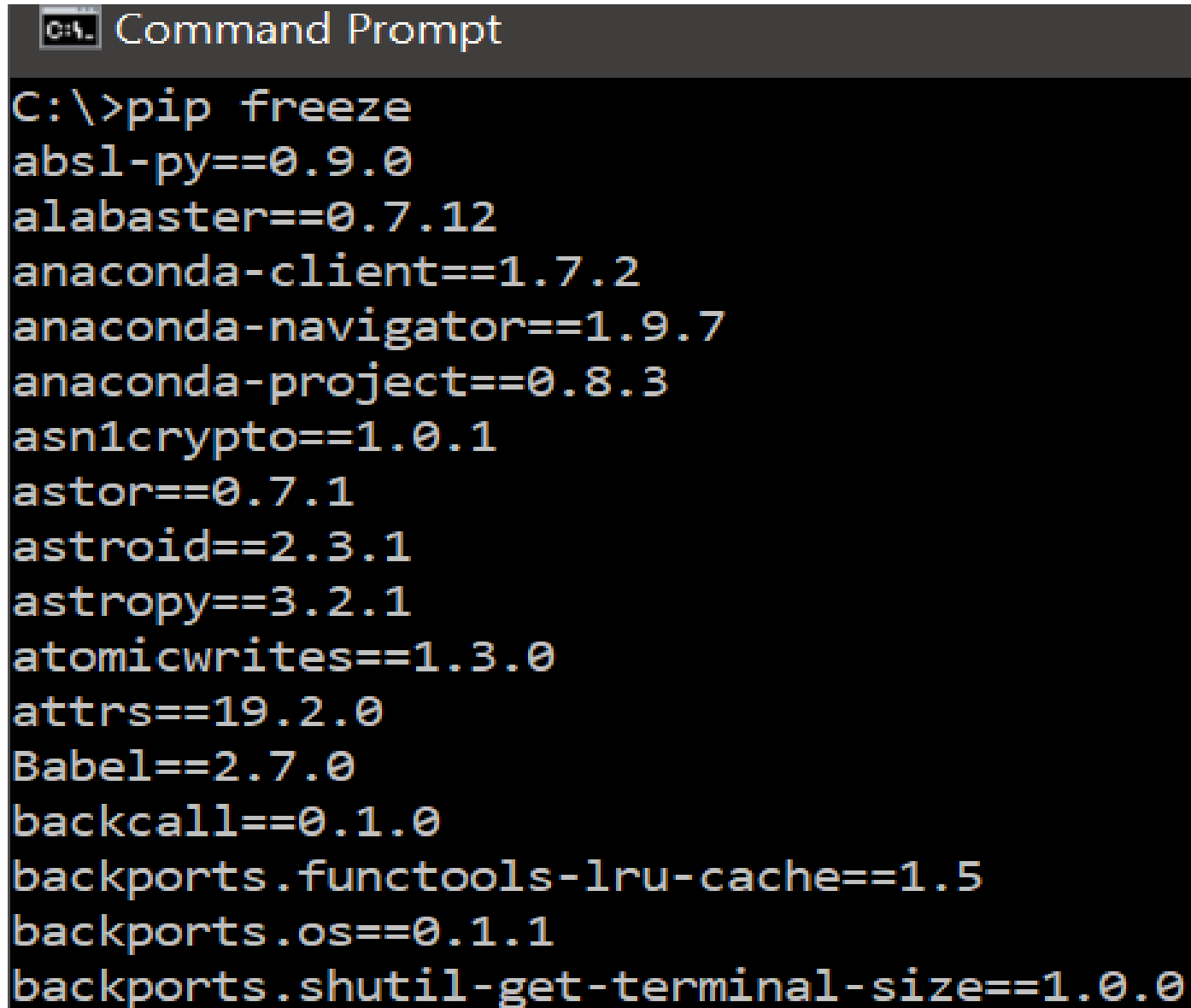

Listing additional Packages with Python PIP

The Python pip freeze command is used to list packages that don't come pre-installed with Python.

Syntax:

pip freeze

Example:



```
Command Prompt
C:\>pip freeze
absl-py==0.9.0
alabaster==0.7.12
anaconda-client==1.7.2
anaconda-navigator==1.9.7
anaconda-project==0.8.3
asn1crypto==1.0.1
astor==0.7.1
astroid==2.3.1
astropy==3.2.1
atomicwrites==1.3.0
attrs==19.2.0
Babel==2.7.0
backcall==0.1.0
backports.functools-lru-cache==1.5
backports.os==0.1.1
backports.shutil-get-terminal-size==1.0.0
```

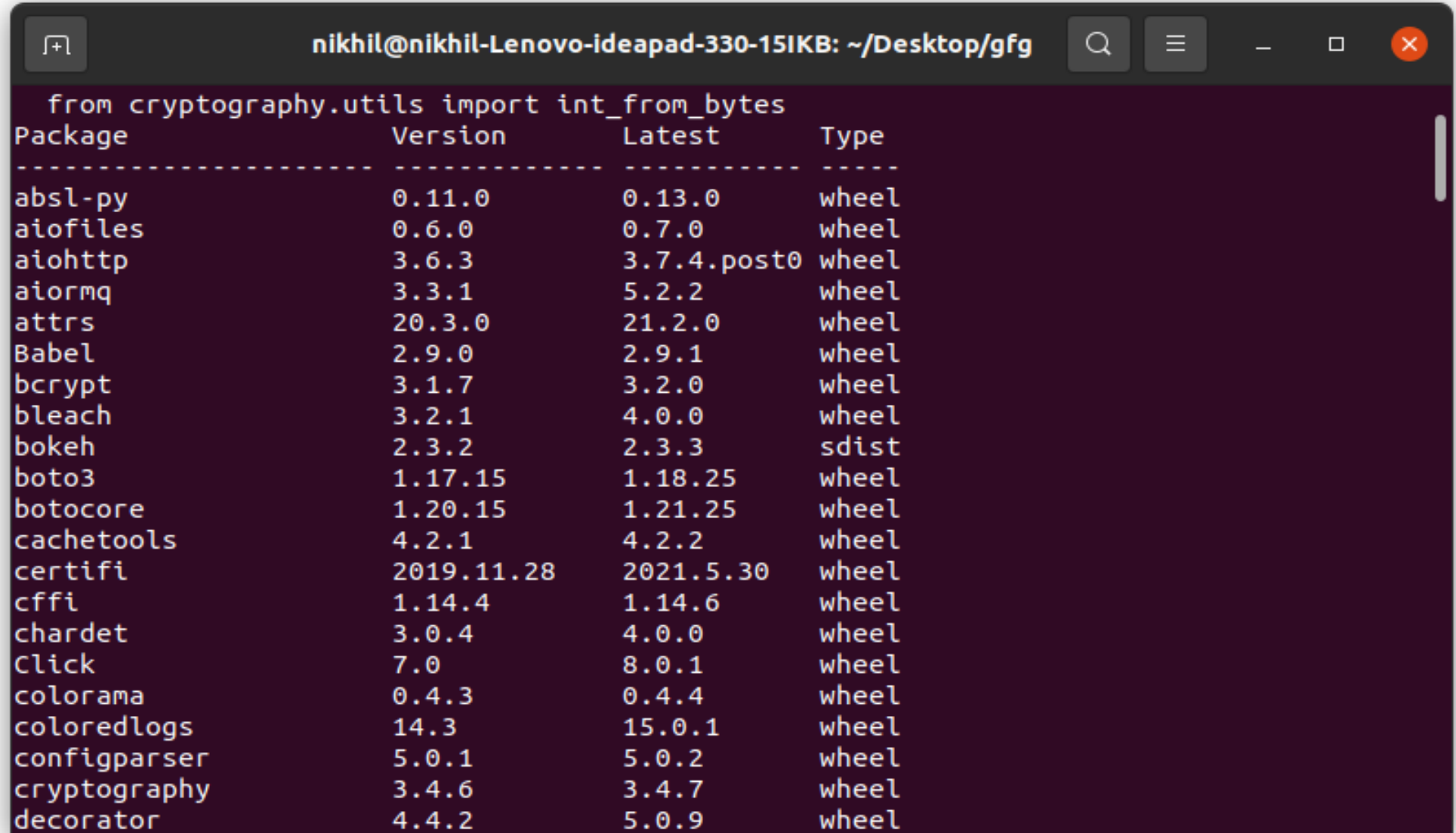
Listing Outdated Packages with Python PIP

Python `pip list --outdated` command is used to list all the packages that are outdated. This command cross-checks the installed package information with the PIP repository.

Syntax:

`pip list --outdated`

Example:



```
from cryptography.utils import int_from_bytes
Package          Version      Latest      Type
-----
absl-py          0.11.0       0.13.0      wheel
aiofiles         0.6.0        0.7.0       wheel
aiohttp          3.6.3        3.7.4.post0 wheel
aiormq           3.3.1        5.2.2       wheel
attrs            20.3.0       21.2.0      wheel
Babel            2.9.0        2.9.1       wheel
bcrypt           3.1.7        3.2.0       wheel
bleach           3.2.1        4.0.0       wheel
bokeh            2.3.2        2.3.3       sdist
boto3            1.17.15      1.18.25     wheel
botocore         1.20.15      1.21.25     wheel
cachetools       4.2.1        4.2.2       wheel
certifi          2019.11.28   2021.5.30   wheel
cffi             1.14.4       1.14.6      wheel
chardet          3.0.4        4.0.0       wheel
Click            7.0          8.0.1       wheel
colorama         0.4.3        0.4.4       wheel
coloredlogs      14.3         15.0.1      wheel
configparser     5.0.1        5.0.2       wheel
cryptography     3.4.6        3.4.7       wheel
decorator        4.4.2        5.0.9       wheel
```

Upgrading Packages with Python PIP

Python `pip install --user --upgrade` is used to update a package.

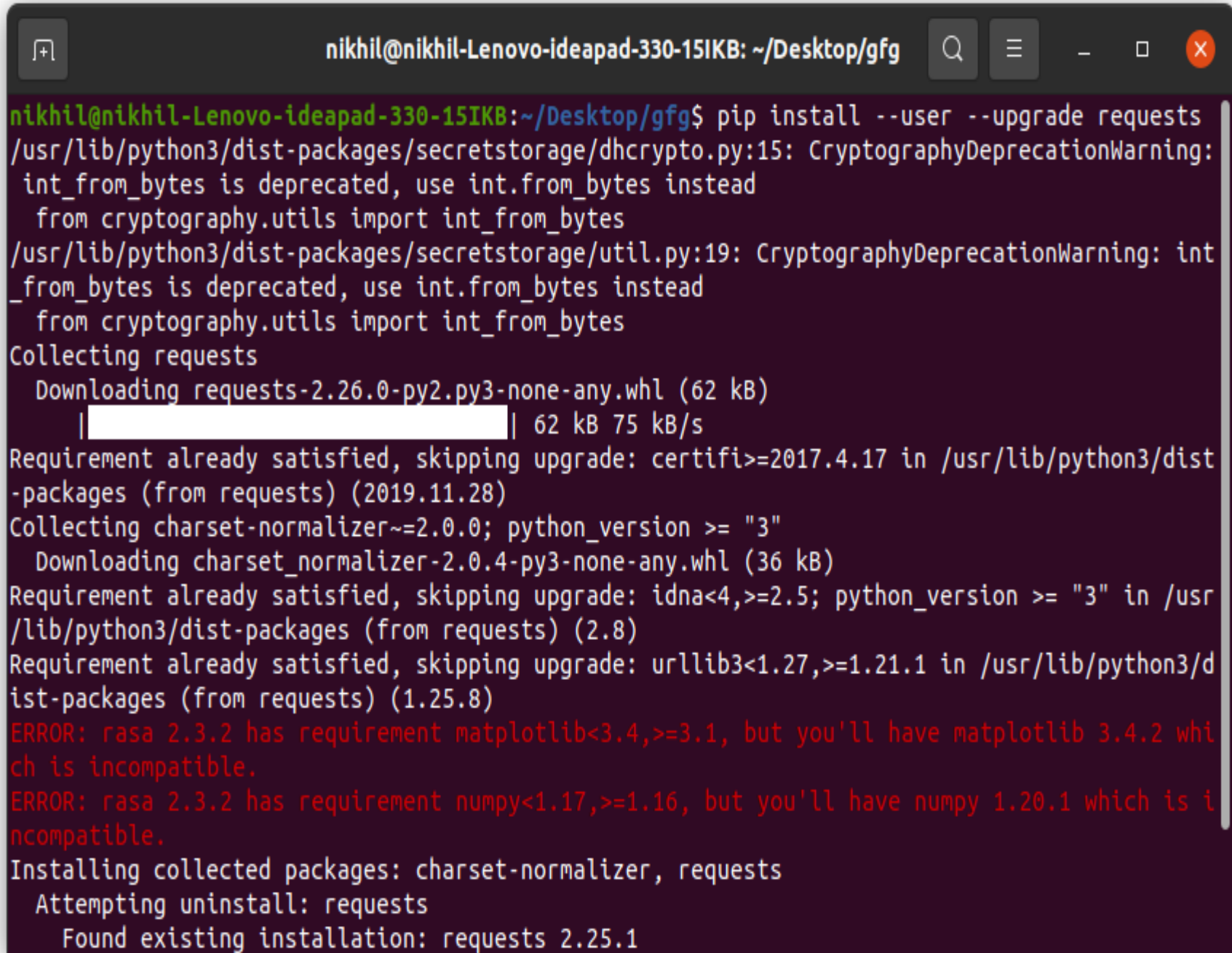
Syntax:

`pip install --user --upgrade package_name`

Example:

We can also upgrade any package to a **specific version** using the below command.

`pip install --user --upgrade package_name==version`

A terminal window titled 'nikhil@nikhil-Lenovo-ideapad-330-15IKB: ~/Desktop/gfg' with standard window controls. The terminal shows the command 'pip install --user --upgrade requests' being executed. It displays deprecation warnings for 'int_from_bytes' in 'dhcrypto.py' and 'util.py'. It then shows the collection and downloading of 'requests-2.26.0-py2.py3-none-any.whl' (62 kB) and 'charset-normalizer-2.0.4-py3-none-any.whl' (36 kB). It also shows that requirements for 'certifi', 'idna', and 'urllib3' are already satisfied. Finally, it shows an error message: 'ERROR: rasa 2.3.2 has requirement matplotlib<3.4,>=3.1, but you'll have matplotlib 3.4.2 which is incompatible.' and another error: 'ERROR: rasa 2.3.2 has requirement numpy<1.17,>=1.16, but you'll have numpy 1.20.1 which is incompatible.' The terminal ends with 'Installing collected packages: charset-normalizer, requests' and 'Attempting uninstall: requests' followed by 'Found existing installation: requests 2.25.1'.

```
nikhil@nikhil-Lenovo-ideapad-330-15IKB: ~/Desktop/gfg$ pip install --user --upgrade requests
/usr/lib/python3/dist-packages/secretstorage/dhcrypto.py:15: CryptographyDeprecationWarning:
int_from_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
/usr/lib/python3/dist-packages/secretstorage/util.py:19: CryptographyDeprecationWarning: int
_from_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
Collecting requests
  Downloading requests-2.26.0-py2.py3-none-any.whl (62 kB)
    |████████████████████| 62 kB 75 kB/s
Requirement already satisfied, skipping upgrade: certifi>=2017.4.17 in /usr/lib/python3/dist
-packages (from requests) (2019.11.28)
Collecting charset-normalizer~=2.0.0; python_version >= "3"
  Downloading charset_normalizer-2.0.4-py3-none-any.whl (36 kB)
Requirement already satisfied, skipping upgrade: idna<4,>=2.5; python_version >= "3" in /usr
/lib/python3/dist-packages (from requests) (2.8)
Requirement already satisfied, skipping upgrade: urllib3<1.27,>=1.21.1 in /usr/lib/python3/d
ist-packages (from requests) (1.25.8)
ERROR: rasa 2.3.2 has requirement matplotlib<3.4,>=3.1, but you'll have matplotlib 3.4.2 whi
ch is incompatible.
ERROR: rasa 2.3.2 has requirement numpy<1.17,>=1.16, but you'll have numpy 1.20.1 which is i
ncompatible.
Installing collected packages: charset-normalizer, requests
  Attempting uninstall: requests
    Found existing installation: requests 2.25.1
```

Downgrading Packages with Python PIP

the Python pip install – user command is used to downgrade a package to a specific version.

Syntax:

```
pip install --user  
package_name==version
```

Example:

```
nikhil@nikhil-Lenovo-ideapad-330-15IKB: ~/Desktop/gfg
nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/Desktop/gfg$ pip install --user requests==2.25.1
/usr/lib/python3/dist-packages/secretstorage/dhcrypto.py:15: CryptographyDeprecationWarning: int
_from_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
/usr/lib/python3/dist-packages/secretstorage/util.py:19: CryptographyDeprecationWarning: int_fro
m_bytes is deprecated, use int.from_bytes instead
  from cryptography.utils import int_from_bytes
Collecting requests==2.25.1
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/lib/python3/dist-packages (from req
uests==2.25.1) (1.25.8)
Requirement already satisfied: chardet<5,>=3.0.2 in /usr/lib/python3/dist-packages (from request
s==2.25.1) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/lib/python3/dist-packages (from request
s==2.25.1) (2019.11.28)
Requirement already satisfied: idna<3,>=2.5 in /usr/lib/python3/dist-packages (from requests==2.
25.1) (2.8)
ERROR: rasa 2.3.2 has requirement matplotlib<3.4,>=3.1, but you'll have matplotlib 3.4.2 which i
s incompatible.
ERROR: rasa 2.3.2 has requirement numpy<1.17,>=1.16, but you'll have numpy 1.20.1 which is incom
patible.
Installing collected packages: requests
  Attempting uninstall: requests
    Found existing installation: requests 2.26.0
    Uninstalling requests-2.26.0:
      Successfully uninstalled requests-2.26.0
Successfully installed requests-2.25.1
nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/Desktop/gfg$
```

Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to **bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.**

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



Python Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Creating a new class creates a new type of object, allowing new instances of that type to be made.

To understand the need for creating a class and object in Python let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Syntax: Class Definition

```
class ClassName:  
    # Statement
```

Syntax: Object Definition

```
obj = ClassName()  
print(obj.attr)
```

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: My class.Myattribute

Creating a Python Class

```
class Dog:  
    sound = "bark"
```

Example of Python Class and object

```
class Dog:  
  
    attr1 = "mammal"  
    attr2 = "dog"  
  
    def fun(self):  
        print("I'm a", self.attr1)  
        print("I'm a", self.attr2)
```

```
Rodger = Dog()  
print(Rodger.attr1)  
Rodger.fun()
```

Output:
mammal
I'm a mammal
I'm a dog

Self Parameter

When we call **a method of this object as myobject.method(arg1, arg2)**, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` – this is all the special self is about.

```
class GFG:
    def __init__(self, name, company):
        self.name = name
        self.company = company

    def show(self):
        print("Hello my name is " + self.name+" and I" +
              " work in "+self.company+".")

obj = GFG("John", "Google")
obj.show()
```


The Self Parameter does not call it to be Self, You can use any other name instead of it.

Here we change the self to the word someone and the output will be the same.

```
class GFG:
    def __init__(someone, name, company):
        someone.name = name
        someone.company = company

    def show(someone):
        print("Hello my name is " + someone.name +
              " and I work in "+someone.company+".")

obj = GFG("John", "Google")
obj.show()
```

Output: Output for both of the codes will be the same.

Hello my name is John and I work in Google

Pass Statement

The program's execution is unaffected by the pass statement's inaction. It merely permits the program to skip past that section of the code without doing anything.

```
class MyClass:  
    pass  
__init__() method
```

```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
    def say_hi(self):  
        print('Hello, my name is', self.name)
```

```
p = Person('Nikhil')  
p.say_hi()
```

Output:

Hello, my name is Nikhil

Defining instance variables using a constructor

```
class Dog:
    animal = 'dog'

    def __init__(self, breed, color):
        self.breed = breed
        self.color = color

# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)
```

#Class variables can be accessed using class

name also

```
print("\nAccessing class variable using class name")
```

```
print(Dog.animal)
```

Output:

Rodger details:

Rodger is a dog

Breed: Pug

Color: brown

Buzo details:

Buzo is a dog

Breed: Bulldog

Color: black

Defining instance variables using the normal method:

```
class Dog:
```

```
    animal = 'dog'
```

```
    # The init method or constructor
```

```
    def __init__(self, breed):
```

```
        self.breed = breed
```

```
    def setColor(self, color):
```

```
        self.color = color
```

```
    def getColor(self):
```

```
        return self.color
```

```
Rodger = Dog("pug")
```

```
Rodger.setColor("brown")
```

```
print(Rodger.getColor())
```

Output:

brown

Constructors in Python

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. **In Python the `__init__()` method is called the constructor** and is always called when an object is created.

Syntax of constructor declaration :

```
def __init__(self):  
    # body of the constructor
```

Types of constructors :

default constructor: The default constructor is a simple constructor which doesn't accept any arguments. Its definition has **only one argument which is a reference to the instance being constructed.**

parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its **first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.**

Example of default constructor :

```
class Google:
```

```
    def __init__(self):  
        self.g = "Google"
```

```
    def print_G(self):  
        print(self.g)
```

```
obj = Google()  
obj.print_G()
```

Output

Google

Example of the parameterized constructor :

```
class Addition:
```

```
    first = 0  
    second = 0  
    answer = 0
```

```
# parameterized constructor
```

```
    def __init__(self, f, s):  
        self.first = f  
        self.second = s
```

```
    def display(self):  
        print("First number = " + str(self.first))  
        print("Second number = " +  
str(self.second))  
        print("Addition of two numbers = " +  
str(self.answer))
```

```
    def calculate(self):  
        self.answer = self.first + self.second
```

```
obj1 = Addition(1000, 2000)  
obj2 = Addition(10, 20)  
obj1.calculate()  
obj2.calculate()  
obj1.display()  
obj2.display()
```

Output

First number = 1000
Second number = 2000
Addition of two numbers
= 3000
First number = 10
Second number = 20
Addition of two numbers
= 30

Example-2 :

```
class MyClass:
    def __init__(self, name=None):
        if name is None:
            print("Default constructor called")
        else:
            self.name = name
            print("Parameterized constructor called with name", self.name)

    def method(self):
        if hasattr(self, 'name'):
            print("Method called with name", self.name)
        else:
            print("Method called without a name")
```

```
obj1 = MyClass()
obj1.method()
obj2 = MyClass("John")
obj2.method()
```

Output

Default constructor called
Method called without a name
Parameterized constructor called with
name John
Method called with name John

Advantages of using constructors in Python:

- **Initialization of objects:** Constructors are used to initialize the objects of a class. They allow you to set default values for attributes or properties, and also allow you to initialize the object with custom data.
- **Easy to implement:** Constructors are easy to implement in Python, and can be defined using the `__init__()` method.
- **Better readability:** Constructors improve the readability of the code by making it clear what values are being initialized and how they are being initialized.
- **Encapsulation:** Constructors can be used to enforce encapsulation, by ensuring that the object's attributes are initialized correctly and in a controlled manner.

Disadvantages of using constructors in Python:

- **Overloading not supported:** Unlike other object-oriented languages, Python does not support method overloading. This means that you cannot have multiple constructors with different parameters in a single class.
- **Limited functionality:** Constructors in Python are limited in their functionality compared to constructors in other programming languages. For example, Python does not have constructors with access modifiers like public, private or protected.

Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration :

```
def __del__(self):  
    # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.

Python program to illustrate destructor

```
class Employee:  
    def __init__(self):  
        print('Employee created.')  
    def __del__(self):  
        print('Destructor called, Employee deleted.')
```

```
obj = Employee()  
del obj
```

Output

```
Employee created.  
Destructor called, Employee deleted.
```

Python program to illustrate destructor

```
class Employee:
```

```
    # Initializing
```

```
    def __init__(self):
```

```
        print('Employee created')
```

```
    # Calling destructor
```

```
    def __del__(self):
```

```
        print("Destructor called")
```

```
def Create_obj():
```

```
    print('Making Object...')
```

```
    obj = Employee()
```

```
    print('function end...')
```

```
    return obj
```

```
print('Calling Create_obj() function...')
```

```
obj = Create_obj()
```

```
print('Program End...')
```

Output

Calling Create_obj() function...

Making Object...

Employee created

function end...

Program End...

Destructor called

Example 3: Now, consider the following example :

Python program to illustrate destructor

```
class A:  
    def __init__(self, bb):  
        self.b = bb
```

```
class B:  
    def __init__(self):  
        self.a = A(self)  
    def __del__(self):  
        print("die")
```

```
def fun():  
    b = B()
```

```
fun()
```

Output

die

Advantages of using destructors in Python:

Automatic cleanup: Destructors provide automatic cleanup of resources used by an object when it is no longer needed. This can be especially useful in cases where resources are limited, or where failure to clean up can lead to memory leaks or other issues.

Consistent behavior: Destructors ensure that an object is properly cleaned up, regardless of how it is used or when it is destroyed. This helps to ensure consistent behavior and can help to prevent bugs and other issues.

Easy to use: Destructors are easy to implement in Python, and can be defined using the `__del__()` method.

Inheritance in Python

It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are:

It represents real-world relationships well.

It provides the reusability of a code. We don't have to write the same code again and again.

Also, it allows us to add more features to a class without modifying it.

It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Inheritance offers a simple, understandable model structure.

Less development and maintenance expenses result from an inheritance.

Python Inheritance Syntax

Class BaseClass:

 {Body}

Class DerivedClass(BaseClass):

 {Body}

Creating a Parent Class

```
class Person(object):  
  
    def __init__(self, name, id):  
        self.name = name  
        self.id = id  
  
    def Display(self):  
        print(self.name, self.id)  
  
emp = Person("Satyam", 102)  
emp.Display()
```

Output:

Satyam 102

Creating a Child Class

```
class Emp(Person):  
  
    def Print(self):  
        print("Emp class called")  
  
Emp_details = Emp("Mayank", 103)  
  
Emp_details.Display()  
  
Emp_details.Print()
```

Output:

Mayank 103
Emp class called

Example of Inheritance in Python

```
class Person(object):
```

```
    def __init__(self, name):  
        self.name = name
```

```
    def getName(self):  
        return self.name
```

```
    # To check if this person is an employee  
    def isEmployee(self):  
        return False
```

```
class Employee(Person):
```

```
    def isEmployee(self):  
        return True
```

```
emp = Person("Google1") # An Object of Person  
print(emp.getName(), emp.isEmployee())
```

```
emp = Employee("Google2")  
print(emp.getName(), emp.isEmployee())
```

Output:

```
Google1 False
```

```
Google2 True
```

Types of inheritance Python

1. Single Inheritance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

Example:

```
class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child(Parent):
    def func2(self):
        print("This function is in child class.")

object = Child()
object.func1()
object.func2()
```

Output:

This function is in parent class.
This function is in child class.

2. Multiple Inheritance:

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.

Example:

```
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father : RAM
Mother : SITA
```

3. Multilevel Inheritance :

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class.

Example:

```
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        Grandfather.__init__(self, grandfathername)

class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        Father.__init__(self, fathername, grandfathername)
```

```
def print_name(self):
    print('Grandfather name :',
self.grandfathername)
    print("Father name :",
self.fathername)
    print("Son name :", self.sonname)
```

```
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

4. Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance.

Example:

```
class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

5. Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Example:

```
class School:
    def func1(self):
        print("This function is in school.")

class Student1(School):
    def func2(self):
        print("This function is in student 1. ")

class Student2(School):
    def func3(self):
        print("This function is in student 2.")

class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")

object = Student3()
object.func1()
object.func2()
```

Output:

This function is in school.

This function is in student 1.

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. **To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.**

Protected members

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the **name of the member by a single underscore “_”**.

Note: The `__init__` method is a constructor and runs as soon as an object of a class is instantiated.

Python program to demonstrate protected members

```
class Base:
```

```
    def __init__(self):
```

```
        self._a = 2
```

```
class Derived(Base):
```

```
    def __init__(self):
```

```
        Base.__init__(self)
```

```
        print("Calling protected member of base class: ", self._a)
```

```
        self._a = 3
```

```
        print("Calling modified protected member outside class: ", self._a)
```

```
obj1 = Derived()
```

```
obj2 = Base()
```

```
print("Accessing protected member of obj1: ", obj1._a)
```

```
print("Accessing protected member of obj2: ", obj2._a)
```

Output:

Calling protected member of base class: 2

Calling modified protected member outside class: 3

Accessing protected member of obj1: 3

Accessing protected member of obj2: 2

Private members

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class.

However, to define a private member prefix the member name with double underscore “__”.

Note: Python’s private and protected members can be accessed outside the class through python name mangling.

Python program to demonstrate private members

```
class Base:
```

```
    def __init__(self):  
        self.a = "Google"  
        self.__c = "Google"
```

```
obj1 = Base()  
print(obj1.a)  
print(obj1.c)  
obj2=Derived()
```

```
class Derived(Base):
```

```
    def __init__(self):  
        Base.__init__(self)  
        print("Calling private member of base class: ")  
        print(self.__c)
```

Output:

Google

Traceback (most recent call last):

```
File "/home/f4905b43bfcf29567e360c709d3c52bd.py", line 25, in <module>  
    print(obj1.c)
```

AttributeError: 'Base' object has no attribute 'c'

Traceback (most recent call last):

```
File "/home/4d97a4efe3ea68e55f48f1e7c7ed39cf.py", line 27, in <module>  
    obj2 = Derived()
```

```
File "/home/4d97a4efe3ea68e55f48f1e7c7ed39cf.py", line 20, in __init__  
    print(self.__c)
```

AttributeError: 'Derived' object has no attribute '_Derived__c'

Polymorphism in Python

What is Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Python program to demonstrate in-built polymorphic functions

```
# len() being used for a string  
print(len("Googles"))
```

```
# len() being used for a list  
print(len([10, 20, 30]))
```

Output

5

3

Examples of user-defined polymorphic functions:

A simple Python function to demonstrate Polymorphism

```
def add(x, y, z = 0):  
    return x + y + z
```

```
# Driver code  
print(add(2, 3))  
print(add(2, 3, 4))
```

Output

```
5  
9
```

Polymorphism with class methods:

The below code shows how **Python can use two different class types, in the same way**. We create a for loop that iterates through a tuple of objects.

```
class India():  
    def capital(self):  
        print("New Delhi is the capital of India.")
```

```
def language(self):
    print("Hindi is the most widely spoken language of India.")

def type(self):
    print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of
India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance:

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. We re-implement the method in the child class. **This process of re-implementing a method in the child class is known as Method Overriding.**

<pre>class Bird: def intro(self): print("There are many types of birds.") def flight(self): print("Most of the birds can fly but some cannot.") class sparrow(Bird): def flight(self): print("Sparrows can fly.") class ostrich(Bird): def flight(self): print("Ostriches cannot fly.")</pre>	<pre>obj_bird = Bird() obj_spr = sparrow() obj_ost = ostrich() obj_bird.intro() obj_bird.flight() obj_spr.intro() obj_spr.flight() obj_ost.intro() obj_ost.flight()</pre>	<div>Output</div> <div>There are many types of birds.</div> <div>Most of the birds can fly but some cannot.</div> <div>There are many types of birds.</div> <div>Sparrows can fly.</div> <div>There are many types of birds.</div> <div>Ostriches cannot fly.</div>
--	--	--

Polymorphism with a Function and objects:

Code: Implementing Polymorphism with a Function

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")
```

```
class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")
```

```
def func(obj):
    obj.capital()
    obj.language()
    obj.type()
```

```
obj_ind = India()
obj_usa = USA()
```

```
func(obj_ind)
func(obj_usa)
```

Output

New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.