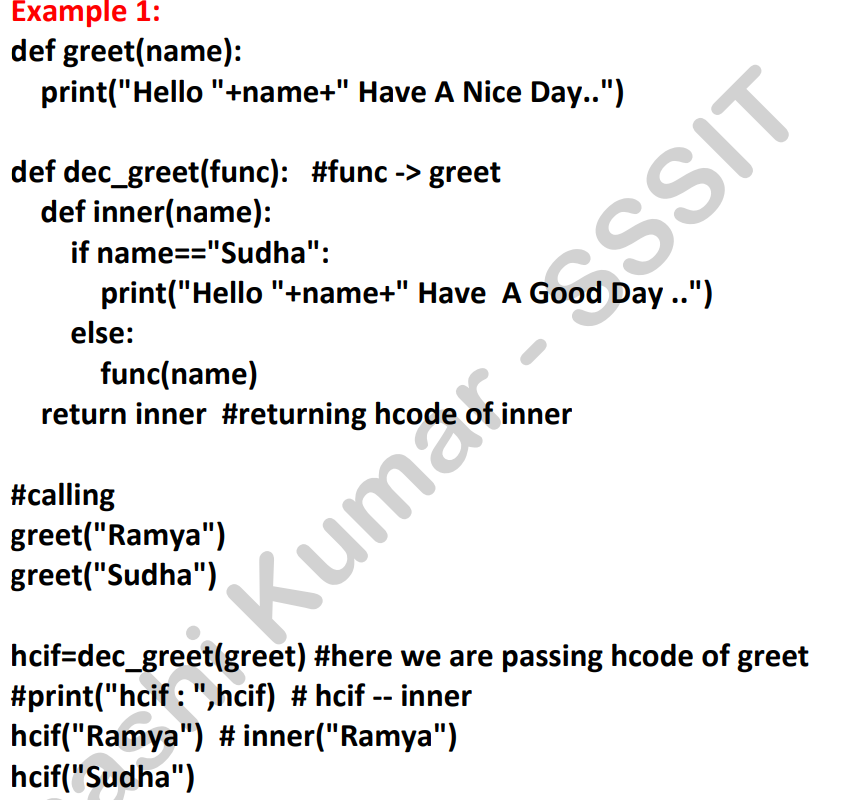
**Python Concepts**

**Decorators**:

Decorator is a function which is used to provide some additional functionality for the function which is already existed.

In Python, decorators are a design pattern that allows you to modify or extend the behavior of functions or methods without changing their actual code. They are often used to add functionality such as logging, access control, instrumentation, caching, and more. Decorators are applied to functions or methods using the **@decorator\_name** syntax.

**Code**:



**Getter and Setters:**

In simple terms, getters and setters are methods used to control access to the attributes (variables) of a class in Python. They allow you to retrieve (get) or update (set) the value of an attribute while maintaining control over how the attribute is accessed or modified.

**Pytest:**

Pytest is a popular testing framework for Python that is used to write simple and scalable test cases. It supports fixtures, parameterized testing, and has a rich plugin architecture. Pytest is known for its ease of use, concise syntax, and ability to handle complex testing needs.

# def new\_check\_div(func):

# def inner(a,b):

# if b==0:

# print('Division is not poosible for this sceneraio')

# return None

# else:

# return func(a,b)

# return inner

# @new\_check\_div

# def check\_div(a,b):

# c= a/b

# return c

# r=check\_div(6,0)

# print('result is ',r)

---------------------------------------------------------------------------------------------

# check\_div =new\_check\_div(check\_div)

# r = check\_div(6,2)

# print('result is ',r)

# class Developer:

# def \_\_init\_\_(self,name,age,companyName):

# self.naem = name

# self.age = age

# self.companyName = companyName

# @property

# def company(self):

# return self.companyName

# @company.setter

# def company(self,companyName):

# self.companyName = companyName

# developer\_object = Developer('siva',24,'DXC')

# print(developer\_object.company)

# def add\_sum(func):

# def inner(a,b):

# if a==0:

# print(" don't provide zero values for a varilable")

# return None

# else:

# return func(a,b)

# return inner

# @add\_sum

# def sum(a,b):

# s=a+b

# return s

# r=sum(0,20)

# print("reult is ",r)

# class email:

# def \_\_init\_\_(self,teampmail):

# self.teampmail = teampmail

# @property

# def notification(self):

# return self.teampmail

# @notification.setter

# def notification(self,teampmail):

# self.teampmail = teampmail

# email\_refference = email("siva@gmail.com")

# print(email\_refference.notification)

def test(\*\*x):

for i in x.items():

print(i)

test(id=123,name="siva",age=24)

**About API and Python Framework:**

An API (Application Programming Interface) is not a framework but a set of rules and tools for building software applications. It defines how different software components should interact. APIs can be used to allow different software systems to communicate with each other or to expose the functionality of a system to developers.

In the context of Python, several frameworks and libraries are designed to work with APIs. Here are some examples of Python frameworks and libraries related to APIs:

**Python Web Frameworks for Building APIs**

1. **Flask**
   * **Description**: A lightweight web framework for Python that can be used to build APIs.
   * **Usage**: Commonly used for creating RESTful APIs.
   * **Documentation**: Flask

python

Copy code

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route('/api', methods=['GET'])

def api():

return jsonify({'message': 'Hello, World!'})

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

1. **Django**
   * **Description**: A high-level web framework that includes features for building APIs, especially with the Django REST framework.
   * **Usage**: Suitable for building more complex applications with APIs.
   * **Documentation**: [Django](https://www.djangoproject.com/) and [Django REST Framework](https://www.django-rest-framework.org/)

python

Copy code

# views.py in Django with Django REST Framework

from rest\_framework.views import APIView

from rest\_framework.response import Response

class HelloWorld(APIView):

def get(self, request):

return Response({'message': 'Hello, World!'})

1. **FastAPI**
   * **Description**: A modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints.
   * **Usage**: Known for its high performance and automatic generation of OpenAPI documentation.
   * **Documentation**: FastAPI

python

Copy code

from fastapi import FastAPI

app = FastAPI()

@app.get("/api")

async def read\_root():

return {"message": "Hello, World!"}

1. **Tornado**
   * **Description**: A web framework and asynchronous networking library.
   * **Usage**: Used for handling long-lived network connections, such as WebSockets, in addition to HTTP requests.
   * **Documentation**: Tornado

python

Copy code

from tornado.web import Application, RequestHandler

from tornado.ioloop import IOLoop

class MainHandler(RequestHandler):

def get(self):

self.write({"message": "Hello, World!"})

app = Application([

(r"/api", MainHandler),

])

if \_\_name\_\_ == "\_\_main\_\_":

app.listen(8888)

IOLoop.current().start()

**Python Libraries for Consuming APIs**

1. **Requests**
   * **Description**: A popular library for making HTTP requests.
   * **Usage**: Used to interact with APIs and retrieve data.
   * **Documentation**: [Requests](https://requests.readthedocs.io/)

python

Copy code

import requests

response = requests.get('https://api.example.com/data')

data = response.json()

1. **httpx**
   * **Description**: A fully featured HTTP client for Python 3, which provides async capabilities.
   * **Usage**: Used for making HTTP requests, especially in asynchronous applications.
   * **Documentation**: [httpx](https://www.python-httpx.org/)

python

Copy code

import httpx

async def fetch\_data():

async with httpx.AsyncClient() as client:

response = await client.get('https://api.example.com/data')

return response.json()

**Summary**

* **API**: A set of rules and tools for building software applications and enabling interaction between them.
* **Frameworks**: Tools for building APIs in Python include Flask, Django, FastAPI, and Tornado.
* **Libraries**: Tools for consuming APIs in Python include Requests and httpx.

Each of these frameworks and libraries provides different features and capabilities, allowing you to choose one based on your project requirements.