Advanced Python

Type	Data science masterclass

1. Object-Oriented Programming (OOP) in Python

Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure code. It provides a way to model real-world entities and promotes code reusability, encapsulation, and abstraction.

Classes and Objects

A **class** is a blueprint for creating objects, and an **object** is an instance of a class.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Creating an object
my_car = Car("Toyota", "Corolla", 2022)
print(my_car.display_info())
```

Instance and Class Variables

Instance variables are specific to an object, while class variables are shared across all instances.

```
class Student:
    school = "XYZ High School"  # Class variable

    def __init__(self, name, grade):
        self.name = name  # Instance variable
        self.grade = grade  # Instance variable

student1 = Student("Alice", "A")
print(student1.name, student1.grade, student1.school)
```

Methods (`self` and `cls`)

- Instance methods use `self` to access instance variables.
- Class methods use `cls` to access class variables.

```
class Person:
    species = "Homo sapiens"

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

    def greet(self):  # Instance method
        return f"Hello, my name is {self.name}"

    @classmethod
    def get_species(cls):  # Class method
        return cls.species

person1 = Person("John")
print(person1.greet())
print(Person.get_species())
```

Constructors (`__init__`)

The `__init__` method initializes an object when it is created.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def describe(self):
        return f"This is {self.name}, a {self.breed}."

my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.describe())
```

Inheritance

Inheritance allows a class to derive attributes and methods from another class.

```
class Animal:
    def speak(self):
        return "I make a sound."

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

my_dog = Dog()
print(my_dog.speak())
```

---

2. Iterators and Generators

Understanding Iterators

An iterator is an object that implements `__iter__()` and `__next__()`.

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        self.current += 1
        return self.current - 1

counter = Counter(1, 5)
for num in counter:
    print(num)
```

Understanding Generators

Generators are iterators that use `yield` instead of `return`.

```
def count_up(start, end):
    while start < end:
        yield start
        start += 1

for num in count_up(1, 5):
    print(num)
```

Generator Expressions

A concise way to create generators.

```
gen = (x*x for x in range(5))
print(next(gen))  # 0
print(next(gen))  # 1
```

---

3. Decorators and Closures

Understanding Closures

A function that remembers variables from its outer scope even after the scope has finished executing.

```
def outer_func(x):
    def inner_func(y):
        return x + y
    return inner_func

closure_func = outer_func(10)
print(closure_func(5))  # 15
```

Function Decorators

A decorator modifies the behavior of a function without modifying its structure.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@decorator
def say_hello():
    print("Hello!")
```

```
say_hello()
```

Chaining Decorators

Multiple decorators can be applied to a function.

```
def uppercase_decorator(func):
    def wrapper():
        return func().upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper():
        return func() + "!!!"
    return wrapper

@exclamation_decorator
@uppercase_decorator
def greet():
    return "hello"

print(greet())  # HELLO!!!
```

---

4. Context Managers and `with` Statement

Using `with` Statement

The `with` statement ensures that resources are properly cleaned up.

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Implementing a Context Manager

A class implementing `__enter__` and `__exit__` methods can be used as a context manager.

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

with FileManager("example.txt", "w") as f:
    f.write("Hello, World!")
```

Using `contextlib`

Simplifying context manager creation using `contextlib`.

```
from contextlib import contextmanager

@contextmanager
def open_file(filename, mode):
    f = open(filename, mode)
    yield f
    f.close()

with open_file("example.txt", "w") as file:
    file.write("Hello, Python!")
```

5. Metaclasses and `type`

Understanding Metaclasses

Metaclasses define the behavior of classes themselves. A class is an instance of a metaclass, just as objects are instances of classes.

```
class Meta(type):
    def __new__(cls, name, bases, class_dict):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, class_dict)

class MyClass(metaclass=Meta):
    pass
```

Custom Metaclasses

You can define custom metaclasses to enforce class constraints or modify behavior.

```
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **k
wargs)
        return cls._instances[cls]

class SingletonClass(metaclass=SingletonMeta):
    pass

obj1 = SingletonClass()
obj2 = SingletonClass()
print(obj1 is obj2)  # True
```

6. Multi-threading and Multi-processing

Introduction to Concurrency

Concurrency allows multiple tasks to run seemingly simultaneously, improving efficiency.

`threading` Module

Using `threading` for lightweight concurrent tasks.

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()
```

`multiprocessing` Module

For true parallel execution, `multiprocessing` creates separate processes.

```
import multiprocessing

def worker():
    print("Worker process running")

process = multiprocessing.Process(target=worker)
process.start()
process.join()
```

Process Pools

Using `Pool` for managing multiple processes efficiently.

```
from multiprocessing import Pool

def square(n):
    return n * n

with Pool(4) as p:
    results = p.map(square, [1, 2, 3, 4])
print(results)
```

---

7. Async Programming (`asyncio`)

Understanding `async` and `await`

Asynchronous programming allows tasks to run without blocking execution.

```
import asyncio

async def say_hello():
    await asyncio.sleep(1)
    print("Hello, Async!")

asyncio.run(say_hello())
```

Event Loop and Tasks

Running multiple async tasks concurrently.

```
async def task1():
    await asyncio.sleep(1)
    print("Task 1 completed")

async def task2():
    await asyncio.sleep(2)
    print("Task 2 completed")
```

```
async def main():
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

---

8. Memory Management and Performance Optimization

Understanding Python Memory Management

Python manages memory with automatic garbage collection and reference counting.

```
import sys

a = []
print(sys.getrefcount(a))  # Returns reference count of the o
bject
```

Garbage Collection (`gc` module)

Python automatically removes unused objects, but manual garbage collection can optimize performance.

```
import gc

gc.collect()  # Forces garbage collection
```

Profiling Code (`cProfile` , `timeit`)

Using profiling tools to measure execution time and performance bottlenecks.

```
import timeit

def example_function():
```

```
    return sum(range(1000))

print(timeit.timeit(example_function, number=10000))
```

Using `lru_cache` for Performance Optimization

Memoization with `functools.lru_cache` can speed up repeated computations.

```
from functools import lru_cache

@lru_cache(maxsize=100)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(50))
```

---

9. Working with JSON, XML, and YAML

JSON Parsing (`json` module)

JSON (JavaScript Object Notation) is a popular format for data exchange.

```
import json

# Convert Python dictionary to JSON
person = {"name": "Alice", "age": 25, "city": "New York"}
json_data = json.dumps(person)
print(json_data)  # JSON formatted string

# Convert JSON back to Python dictionary
```

```
person_dict = json.loads(json_data)
print(person_dict["name"])  # Alice
```

XML Parsing (`xml.etree.ElementTree` module)

XML (Extensible Markup Language) is commonly used for structured data storage.

```
import xml.etree.ElementTree as ET

xml_data = """<person><name>Alice</name><age>25</age></person
>"""
root = ET.fromstring(xml_data)
print(root.find("name").text)  # Alice
```

YAML Parsing (`PyYAML` library)

YAML (Yet Another Markup Language) is human-readable and commonly used for configuration files.

```
import yaml

yaml_data = """
name: Alice
age: 25
city: New York
"""
parsed_yaml = yaml.safe_load(yaml_data)
print(parsed_yaml["name"])  # Alice
```

---

10. Regular Expressions (`re` module)

Pattern Matching and Searching

Regular expressions are used for pattern matching and text processing.

```
import re

text = "The price is $50."
match = re.search(r"\$\d+", text)
if match:
    print(match.group())  # $50
```

Using `match()`, `search()`, `findall()`, and `sub()`

```
pattern = r"\d+"
text = "There are 3 apples and 5 bananas."

print(re.findall(pattern, text))  # ['3', '5']
print(re.sub(pattern, "X", text))  # 'There are X apples and
X bananas.'
```

---

11. Advanced File Handling

Working with CSV Files (`csv` module)

CSV (Comma-Separated Values) files are widely used for tabular data.

```
import csv

# Writing to a CSV file
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Alice", 25])

# Reading from a CSV file
with open("data.csv", "r") as file:
    reader = csv.reader(file)
```

```
    for row in reader:
        print(row)
```

Working with ZIP and Tar Files

```
import zipfile

# Creating a ZIP file
with zipfile.ZipFile("files.zip", "w") as zipf:
    zipf.write("data.csv")
```

---

12. Networking and Sockets

Understanding Sockets (`socket` module)

Sockets enable communication between networked devices.

```
import socket

# Create a simple server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 8080))
server.listen(1)
print("Server listening on port 8080...")
conn, addr = server.accept()
print("Connection from", addr)
conn.sendall(b"Hello, Client!")
conn.close()
```

HTTP Requests with `requests` module

Fetching data from the web using the `requests` module.

```
import requests
```

```
response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.json())
```

---

13. Web Scraping

Basics of Web Scraping

Web scraping is the process of extracting data from websites programmatically.

```
import requests
from bs4 import BeautifulSoup

url = "https://example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")

print(soup.title.text)  # Extracting the title of the page
```

Using `BeautifulSoup`

`BeautifulSoup` helps parse and navigate HTML/XML content.

```
html_doc = """
<html><head><title>Sample Page</title></head>
<body><p class="content">Hello, World!</p></body>
</html>
"""

soup = BeautifulSoup(html_doc, "html.parser")
print(soup.find("p", class_="content").text)  # Hello, World!
```

Automating Web Actions with `Selenium`

`Selenium` allows automation of web interactions such as clicking buttons and filling forms.

```
from selenium import webdriver

# Initialize the browser driver
browser = webdriver.Chrome()
browser.get("https://example.com")

print(browser.title)  # Prints the title of the page
browser.quit()
```

---

14. Logging and Debugging

Using the `logging` Module

Logging helps track events and errors in applications.

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("This is an info message")
logging.error("This is an error message")
```

Debugging with `pdb`

Python's `pdb` module allows step-by-step debugging.

```
import pdb

def add(a, b):
    pdb.set_trace()  # Breakpoint
    return a + b
```

```
print(add(2, 3))
```

Exception Logging and Handling

Handling exceptions with proper logging ensures error tracking.

```
try:
    1 / 0
except ZeroDivisionError as e:
    logging.exception("An error occurred: %s", e)
```