#_ Mastering Python: Important Notes [Part #2]

1. Context Managers

Explanation:

Context managers simplify resource management in Python by providing a way to allocate and release resources. This is achieved using the `with` statement, where the setup and teardown logic is handled automatically.

Code Example:

```
from contextlib import contextmanager

@contextmanager
def manage_file(filename):
    file = open(filename, 'r')
    yield file
    file.close()

with manage_file('file.txt') as file:
    print(file.read())
```

2. Metaclasses

Explanation:

Metaclasses in Python are classes of classes. They allow you to control the creation and behavior of a class. The `type` function is commonly used as a metaclass.

Code Example:

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

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

By: Waleed Mousa

3. Decorator Stacking

Explanation:

Decorator stacking refers to applying multiple decorators to a function. The order matters, as each decorator wraps the function returned by the previous one.

Code Example:

```
def decorator_one(func):
    def wrapper():
        print("Decorator One")
        func()
    return wrapper

def decorator_two(func):
    def wrapper():
        print("Decorator Two")
        func()
    return wrapper

@decorator_one
@decorator_two
def my_function():
    print("Original Function")

my_function()
```

4. Function Annotations

Explanation:

Function annotations are used to provide metadata about the function's parameters and return value. This can be helpful for documentation and type hinting.

Code Example:

```
def add(a: int, b: int) -> int:
    """Add two integers."""
    return a + b
```

5. Function Overloading

Explanation:

Function overloading allows a function to have multiple behaviors based on its arguments. Python doesn't support function overloading directly, but you can achieve similar functionality using default arguments and type checking.

Code Example:

```
def multiply(a, b=None):
    if b is None:
        return a * a
    else:
        return a * b

print(multiply(3))    # Output: 9
print(multiply(3, 4)) # Output: 12
```

6. Abstract Base Classes (ABCs)

Explanation:

Abstract Base Classes are used to define a common interface for derived classes. They allow you to define methods that must be implemented in any child class.

Code Example:

```
from abc import ABC, abstractmethod

class MyABC(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass

class ConcreteClass(MyABC):
    def my_abstract_method(self):
        print("Implemented abstract method")

obj = ConcreteClass()
obj.my_abstract_method()  # Output: Implemented abstract method
```

By: Waleed Mousa

7. Dynamic Importing

Explanation:

Dynamic importing allows you to import modules and classes at runtime. You can use the `importlib` module or the `__import__` function for this purpose.

Code Example:

```
import importlib

math_module = importlib.import_module('math')
sqrt = math_module.sqrt
print(sqrt(16))  # Output: 4.0
```

8. Generators vs. Iterators

Explanation:

- **Iterators**: An object that allows iteration over its elements using `__iter__` and `__next__` methods.
- **Generators**: A simpler way to create iterators using functions with the `yield` statement.

Code Example:

```
# Iterator
class MyIterator:
    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        self.num *= 2
        return self.num

# Generator
def my_generator():
    num = 1
    while True:
        num *= 2
        yield num
```

By: Waleed Mousa

```
iter_obj = MyIterator()
gen_obj = my_generator()

print(next(iter(iter_obj)))  # Output: 2
print(next(gen_obj))         # Output: 2
```

9. Coroutines

Explanation:

Coroutines are a more generalized form of subroutines that allow multiple entry points for pausing and resuming execution. In Python, you can create coroutines using `async` and `await` keywords.

Code Example:

```
import asyncio

async def my_coroutine():
    print("Start")
    await asyncio.sleep(1)
    print("End")

asyncio.run(my_coroutine())
```

10. Asynchronous Context Managers

Explanation:

Asynchronous context managers allow managing resources in an asynchronous context using `async with`.

Code Example:

```
import aiofiles

async def read_file():
    async with aiofiles.open('file.txt', mode='r') as file:
        contents = await file.read()
        print(contents)

# Use asyncio.run(read_file()) to run the above code in an appropriate context
```

By: Waleed Mousa

11. Asynchronous Iterators

Explanation:

Asynchronous iterators allow asynchronous iteration using `__aiter__` and `__anext__` methods. They enable you to iterate over items in an asynchronous manner.

Code Example:

```
class AsyncCounter:
    def __init__(self, limit):
        self.limit = limit

    def __aiter__(self):
        self.count = 0
        return self

    async def __anext__(self):
        if self.count < self.limit:
            self.count += 1
            return self.count
        else:
            raise StopAsyncIteration

async def run():
    async for value in AsyncCounter(5):
        print(value)

# Use asyncio.run(run()) to execute the code
```

12. asyncio Library

Explanation:

The `asyncio` library is used to write concurrent code using the `async`/`await` syntax in Python. It provides an event loop that handles scheduling and execution of asynchronous tasks.

Code Example:

```
import asyncio

async def print_numbers():
```

```
    for i in range(5):
        print(i)
        await asyncio.sleep(1)

asyncio.run(print_numbers())
```

13. Concurrent.Futures

Explanation:

The `concurrent.futures` module provides a high-level interface for asynchronously executing functions using threads or processes.

Code Example:

```
from concurrent.futures import ThreadPoolExecutor

def task(n):
    return n * 2

with ThreadPoolExecutor() as executor:
    results = executor.map(task, [1, 2, 3])
    print(list(results))  # Output: [2, 4, 6]
```

14. ThreadPoolExecutor

Explanation:

The `ThreadPoolExecutor` class is used for managing thread-based parallelism. It enables concurrent execution of tasks in separate threads.

Code Example:

```
from concurrent.futures import ThreadPoolExecutor

def square(n):
    return n * n

with ThreadPoolExecutor() as executor:
    future = executor.submit(square, 3)
    print(future.result())  # Output: 9
```

15. ProcessPoolExecutor

Explanation:

The `ProcessPoolExecutor` class allows parallel execution of tasks across different processes, taking advantage of multiple CPU cores.

Code Example:

```
from concurrent.futures import ProcessPoolExecutor

def multiply(n):
    return n * 2

with ProcessPoolExecutor() as executor:
    results = list(executor.map(multiply, [1, 2, 3]))
    print(results)  # Output: [2, 4, 6]
```

16. Memory Management

Explanation:

Memory management in Python involves allocation, deallocation, and garbage collection of memory. Python's memory manager handles these aspects automatically, but you can also interact with it using some built-in functions.

Code Example:

```
import gc

# Manually triggering garbage collection
gc.collect()

# Getting count of objects being tracked by garbage collector
print(gc.get_count())
```

17. Profiling and Optimization

Explanation:

Profiling refers to the measurement of program execution to analyze its performance. Optimization involves improving code performance by reducing bottlenecks.

Code Example:

```
import cProfile

def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)

def main():
    print(factorial(10))

cProfile.run('main()')
```

18. Cython

Explanation:

Cython is a programming language that combines Python with C, enabling performance improvements. It allows Python code to be compiled to C, thus improving execution speed.

Code Example:

```
# my_cython_module.pyx
def multiply(a, b):
    return a * b
```

```
# Compile with Cython
cythonize -i my_cython_module.pyx
```

```
# Usage in Python
import my_cython_module
print(my_cython_module.multiply(3, 4))  # Output: 12
```

19. Memory Views

Explanation:

Memory views provide a way to access the internal data of an object without copying. They allow efficient manipulation of data structures that support the buffer protocol.

Code Example:

```
arr = bytearray([1, 2, 3, 4])
mem_view = memoryview(arr)
mem_view[1] = 5
print(arr)  # Output: bytearray(b'\x01\x05\x03\x04')
```

20. GIL (Global Interpreter Lock)

Explanation:

The Global Interpreter Lock (GIL) is a mutex that allows only one thread to execute in the interpreter at a time. This can limit the performance of CPU-bound multithreaded applications.

Code Example:

```
# The GIL can impact code like this:
from threading import Thread

def count_up_to(n):
    count = 0
    while count < n:
        count += 1

threads = [Thread(target=count_up_to, args=(100000000,)) for _ in range(2)]
for t in threads:
    t.start()
for t in threads:
    t.join()
# The performance may not scale linearly with the number of threads due to
the GIL
```

21. C Extensions

Explanation:

C Extensions enable the integration of C code with Python to achieve better performance in critical parts of the code. They allow developers to write native modules in C, which can be imported and used like regular Python modules.

Code Example:

```c
// example.c
#include <Python.h>

static PyObject* multiply(PyObject* self, PyObject* args) {
    int a, b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b))
        return NULL;
    return Py_BuildValue("i", a * b);
}

static PyMethodDef ExampleMethods[] = {
    {"multiply", multiply, METH_VARARGS, "Multiply two integers."},
    {NULL, NULL, 0, NULL} /* Sentinel */
};

static struct PyModuleDef examplemodule = {
    PyModuleDef_HEAD_INIT,
    "example",
    NULL,
    -1,
    ExampleMethods
};

PyMODINIT_FUNC PyInit_example(void) {
    return PyModule_Create(&examplemodule);
}
```

```
# Compiling
gcc -shared -o example.so -I /usr/include/python3.8 example.c
```

```
# Usage in Python
import example
print(example.multiply(2, 3))  # Output: 6
```

By: Waleed Mousa

22. C-API

Explanation:

Python's C-API allows interfacing between Python and C, enabling the creation of custom native extensions and interaction with Python objects from C code.

Code Example:

```
// Similar to the C Extension example above
```

23. NumPy

Explanation:

NumPy is a powerful library for numerical computations in Python, providing support for large arrays and matrices along with mathematical functions to operate on these arrays.

Code Example:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise addition
result = a + b
print(result)  # Output: [5 7 9]
```

24. SciPy

Explanation:

SciPy is an open-source library used for scientific and technical computing. It builds on NumPy and provides additional modules for optimization, integration, interpolation, and more.

Code Example:

```
from scipy.optimize import minimize

def objective_function(x):
```

```
    return x ** 2 + 3 * x - 5

result = minimize(objective_function, 0)
print(result.x)  # Output: Approximation of the minimum
```

25. pandas

Explanation:

pandas is a popular library for data manipulation and analysis. It provides data structures like DataFrame and Series to handle and analyze structured data.

Code Example:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
}
df = pd.DataFrame(data)

# Filtering by age
filtered_df = df[df['Age'] > 26]
print(filtered_df)  # Output: Data for Bob
```

26. Matplotlib

Explanation:

Matplotlib is a widely used library for creating static, interactive, and animated visualizations in Python.

Code Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [4, 6, 8]

plt.plot(x, y)
plt.xlabel('X Label')
plt.ylabel('Y Label')
```

```
plt.title('Line Plot')
plt.show()
```

27. Seaborn

Explanation:

Seaborn is a data visualization library based on Matplotlib. It provides a higher-level, easier-to-use interface for creating informative and attractive graphics.

Code Example:

```
import seaborn as sns

tips = sns.load_dataset('tips')
sns.boxplot(x='day', y='total_bill', data=tips)
plt.show()
```

28. Plotly

Explanation:

Plotly is a graphing library that enables the creation of interactive and visually appealing plots. It can be used for various types of plots and visualizations.

Code Example:

```
import plotly.express as px

iris = px.data.iris()
fig = px.scatter(iris, x="sepal_width", y="sepal_length", color="species")
fig.show()
```

29. Bokeh

Explanation:

Bokeh is a Python library for creating interactive web-based visualizations. It provides tools to produce elegant and interactive graphics that can be embedded in web applications.

Code Example:

```
from bokeh.plotting import figure, show

p = figure(title="Line Plot Example", x_axis_label='x', y_axis_label='y')
p.line([1, 2, 3], [4, 6, 5], line_width=2)
show(p)
```

30. scikit-learn

Explanation:

scikit-learn is a popular library for machine learning. It provides simple and efficient tools for data mining, data analysis, and modeling.

Code Example:

```
from sklearn.linear_model import LinearRegression

X = [[1], [2], [3]]
y = [2, 4, 6]
model = LinearRegression()
model.fit(X, y)

prediction = model.predict([[4]])
print(prediction)  # Output: Approximation of 8
```

31. TensorFlow

Explanation:

TensorFlow is an open-source machine learning framework developed by Google. It is designed to create deep learning models and other machine learning tasks.

Code Example:

```
import tensorflow as tf

# Define a simple Sequential model
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(10,)),
    tf.keras.layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Generate dummy data
import numpy as np
data = np.random.random((1000, 10))
labels = np.random.randint(2, size=(1000, 1))

# Train the model
model.fit(data, labels, epochs=10)
```

32. PyTorch

Explanation:

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab. It's known for its flexibility and is used for deep learning and scientific computing.

Code Example:

```
import torch
import torch.nn as nn

# Define a simple linear model
model = nn.Linear(10, 1)

# Create random input and output data
x = torch.randn(1000, 10)
y = torch.randn(1000, 1)

# Define loss function and optimizer
loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for t in range(100):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
```

```
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

33. Keras

Explanation:

Keras is a high-level neural networks API, capable of running on top of TensorFlow, CNTK, or Theano. It allows easy construction, training, and deployment of various kinds of neural network models.

Code Example:

```
from keras.models import Sequential
from keras.layers import Dense

# Define a Sequential model
model = Sequential([
    Dense(16, activation='relu', input_shape=(10,)),
    Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Generate dummy data
import numpy as np
data = np.random.random((1000, 10))
labels = np.random.randint(2, size=(1000, 1))

# Train the model
model.fit(data, labels, epochs=10)
```

34. Neural Network Architectures

Explanation:

Neural Network Architectures refer to the design and organization of layers within a neural network, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Generative Adversarial Networks (GANs).

Code Example (CNN with Keras):

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
```

35. Transfer Learning

Explanation:

Transfer Learning is the practice of starting with a pre-trained model and adapting the model to a new, similar task. It can reduce the training time and improve the performance of neural network models.

Code Example (Using pre-trained VGG16 model):

```
from keras.applications.vgg16 import VGG16
from keras.layers import GlobalAveragePooling2D, Dense
from keras.models import Model

# Load pre-trained VGG16 model + higher level layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224,
224, 3))
x = base_model.output
x = GlobalAveragePooling2D()(x)
predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=base_model.input, outputs=predictions)

# First: train only the top layers (which were randomly initialized)
for layer in base_model.layers:
    layer.trainable = False

model.compile(optimizer='adam', loss='binary_crossentropy')
```

36. Reinforcement Learning

Explanation:

Reinforcement Learning (RL) is a type of machine learning algorithm that is based on the idea of learning optimal actions through trial and error. Agents take actions in an environment to achieve a goal.

Code Example (Using Q-learning):

```
import numpy as np

# Dummy transition matrix (state, action) -> next state
transition_matrix = [[0, 1], [0, 2], [1, 2]]

# Q-table initialization
Q = np.zeros((3, 2))

# Hyperparameters
alpha = 0.1
gamma = 0.9

# Training loop
for episode in range(1000):
    state = 0
    done = False
    while not done:
        action = np.random.choice([0, 1])
        next_state = transition_matrix[state][action]
        reward = 1 if next_state == 2 else 0
        Q[state, action] = (1 - alpha) * Q[state, action] + alpha * (reward +
gamma * np.max(Q[next_state]))
        state = next_state
        if state == 2:
            done = True
```

37. Natural Language Processing (NLP)

Explanation:

Natural Language Processing (NLP) involves the application of algorithms to identify and extract the natural language rules in order for the computer to understand, interpret, and generate human languages.

Code Example (Using NLTK for tokenization):

```
import nltk

sentence = "Natural Language Processing is fascinating."
tokens = nltk.word_tokenize(sentence)
print(tokens) # Output: ['Natural', 'Language', 'Processing', 'is',
'fascinating', '.']
```

38. Regular Expressions

Explanation:

Regular Expressions provide a way to search, match, and manipulate strings using patterns. They are a powerful tool for text processing.

Code Example:

```
import re

pattern = r"\d+" # Matches one or more digits
result = re.findall(pattern, "The year is 2023.")
print(result) # Output: ['2023']
```

39. Parallel Computing

Explanation:

Parallel Computing is the simultaneous execution of multiple calculations or processes. It can significantly speed up computations.

Code Example (Using multiprocessing):

```
from multiprocessing import Pool

def square_number(n):
    return n * n

numbers = [1, 2, 3, 4]
with Pool() as pool:
    results = pool.map(square_number, numbers)
print(results) # Output: [1, 4, 9, 16]
```

40. Distributed Computing

Explanation:

Distributed Computing involves multiple computers cooperating to solve a computational problem. Apache Spark is commonly used for distributed data processing.

Code Example (Using Apache Spark):

```
from pyspark import SparkContext

sc = SparkContext("local", "Simple App")
numbers = sc.parallelize([1, 2, 3, 4])
squares = numbers.map(lambda x: x * x)
print(squares.collect()) # Output: [1, 4, 9, 16]
```

41. Web Scraping

Explanation:

Web Scraping is the extraction of data from websites. Libraries like BeautifulSoup can parse HTML and XML documents, making it easier to scrape data.

Code Example (Using BeautifulSoup):

```
from bs4 import BeautifulSoup
import requests

url = "http://example.com"
page = requests.get(url)
soup = BeautifulSoup(page.content, "html.parser")
```

```
title = soup.find('title')
print(title.text) # Output: Example Domain
```

42. API Integration

Explanation:

API Integration involves interacting with APIs to retrieve or manipulate data. Most modern APIs return data in JSON format.

Code Example (Requesting JSON data):

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
data = response.json()
print(data['title']) # Output: 'delectus aut autem'
```

43. GraphQL

Explanation:

GraphQL is a query language and runtime for APIs that prioritizes giving clients exactly the data they request and nothing more.

Code Example (Using `graphql-request` in Python):

```
from graphqlclient import GraphQLClient

client = GraphQLClient('https://api.graph.cool/simple/v1/swapi')
result = client.execute('''
{
  allPersons {
    name
  }
}
''')

print(result)
```

44. Docker

Explanation:

Docker is a platform used to develop, ship, and run applications inside containers. A Dockerfile defines how the container is built.

Code Example (Simple Dockerfile):

```
FROM python:3.8

WORKDIR /app
COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

CMD ["python", "app.py"]
```

45. Kubernetes

Explanation:

Kubernetes is an orchestration system for automating the deployment, scaling, and management of containerized applications.

Code Example (Kubernetes deployment YAML):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: my-image
```

46. Serverless Computing

Explanation:

Serverless Computing allows you to run code without provisioning or managing servers. AWS Lambda is a popular serverless platform.

Code Example (AWS Lambda function in Python):

```
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

47. Web Development Frameworks

Explanation:

Web Development Frameworks like Flask and Django help in building web applications by providing reusable code patterns.

Code Example (Using Flask):

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

48. RESTful APIs

Explanation:

RESTful APIs (Representational State Transfer) allow interaction with resources through stateless operations. They follow standard HTTP methods like GET, POST, PUT, DELETE.

Code Example (Using Flask to create a RESTful endpoint):

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = {'id': user_id, 'name': 'Alice'} # Example data
    return jsonify(user)

if __name__ == '__main__':
    app.run()
```

49. GraphQL APIs

Explanation:

GraphQL APIs provide a flexible query language that allows clients to request exactly the data they need. This can result in more efficient data retrieval.

Code Example (Using Graphene with Flask):

```
from flask import Flask
from flask_graphql import GraphQLView
import graphene

class Query(graphene.ObjectType):
    hello = graphene.String()

    def resolve_hello(self, info):
        return "World"

app = Flask(__name__)
app.add_url_rule('/graphql', view_func=GraphQLView.as_view('graphql',
schema=graphene.Schema(query=Query)))
```

```
if __name__ == '__main__':
    app.run()
```

50. WebSocket Communication

Explanation:

WebSockets provide full-duplex communication channels over a single TCP connection, allowing real-time updates between clients and servers.

Code Example (Using WebSockets with Flask-SocketIO):

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
socketio = SocketIO(app)

@app.route('/')
def main():
    return render_template('index.html')

@socketio.on('message')
def handle_message(msg):
    print('Received message:', msg)

if __name__ == '__main__':
    socketio.run(app)
```

51. OAuth and Authentication

Explanation:

OAuth is a standard protocol for access delegation commonly used as a way for users to grant access to their information on one site to another site.

Code Example (Using Flask-OAuthlib):

```
from flask import Flask, redirect, url_for
from flask_oauthlib.client import OAuth

app = Flask(__name__)
oauth = OAuth(app)
```

```python
google = oauth.remote_app('google',
    consumer_key='YOUR_KEY',
    consumer_secret='YOUR_SECRET',
    request_token_params={
        'scope': 'email',
    },
    base_url='https://www.googleapis.com/oauth2/v1/userinfo',
    request_token_url=None,
    access_token_method='POST',
    access_token_url='https://accounts.google.com/o/oauth2/token',
    authorize_url='https://accounts.google.com/o/oauth2/auth',
)

@app.route('/')
def index():
    return 'Welcome to the OAuth Example'

@app.route('/login')
def login():
    return google.authorize(callback=url_for('authorized', _external=True))

@app.route('/logout')
def logout():
    session.pop('google_token')
    return redirect(url_for('index'))

@app.route('/login/authorized')
def authorized():
    response = google.authorized_response()
    if response is None or response.get('access_token') is None:
        return 'Access denied'
    session['google_token'] = (response['access_token'], '')
    return 'Logged in'

if __name__ == '__main__':
    app.run()
```

52. API Security

Explanation:

API Security involves implementing measures to secure APIs against unauthorized access and attacks. Techniques include token-based authentication, encryption, and more.

Code Example (Using Flask-JWT for Token Authentication):

```
from flask import Flask, jsonify, request
from flask_jwt import JWT, jwt_required

app = Flask(__name__)
app.config['SECRET_KEY'] = 'super-secret'

def authenticate(username, password):
    # Validate user
    return {'id': 1, 'username': 'user'}

def identity(payload):
    return {'id': 1, 'username': 'user'}

jwt = JWT(app, authenticate, identity)

@app.route('/protected')
@jwt_required()
def protected():
    return jsonify({'message': 'This is a protected view.'})

if __name__ == '__main__':
    app.run()
```

53. SQLAlchemy

Explanation:

SQLAlchemy is an SQL toolkit and Object-Relational Mapping (ORM) library for Python. It allows interaction with relational databases in an object-oriented way.

Code Example:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)

engine = create_engine('sqlite:///users.db')
Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine)
session = Session()

new_user = User(name='Alice')
session.add(new_user)
session.commit()
```

54. NoSQL Databases

Explanation:

NoSQL databases store data in non-tabular formats, such as key-value pairs, documents, graphs, or wide-column stores. They can offer improved scalability and performance for some types of data.

Code Example (Using MongoDB with PyMongo):

```
from pymongo import MongoClient

client = MongoClient('localhost', 27017)
db = client['test_database']
collection = db['test_collection']

data = {'name': 'Alice', 'age': 30}
collection.insert_one(data)

for item in collection.find():
    print(item)
```

55. Multithreading

Explanation:

Multithreading involves running multiple threads concurrently within a single process. This allows for efficient execution of tasks, particularly for I/O-bound or high-level structured network code.

Code Example (Using `threading` module):

```
import threading

def print_numbers():
    for i in range(10):
        print(i)

thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_numbers)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

56. Multiprocessing

Explanation:

Multiprocessing involves running multiple processes concurrently. Unlike threads, processes do not share memory space. This can be more suitable for CPU-bound tasks.

Code Example (Using `multiprocessing` module):

```
from multiprocessing import Process

def print_numbers():
    for i in range(10):
        print(i)

process1 = Process(target=print_numbers)
process2 = Process(target=print_numbers)

process1.start()
process2.start()
```

```
process1.join()
process2.join()
```

57. Decorators

Explanation:

Decorators are a way to modify or enhance functions in Python without changing their code. They are often used for logging, enforcing access control, instrumentation, caching, etc.

Code Example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

58. Logging

Explanation:

Logging is the process of recording messages to understand the flow of a program and diagnose problems. Python's built-in logging module provides a flexible way to configure and generate log messages.

Code Example:

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

logger.info("This is an info message")
```

```
logger.warning("This is a warning message")
```

59. Package Management (pip)

Explanation:

`pip` is the package installer for Python. It allows you to install and manage additional packages that are not part of the Python standard library.

Code Example:

```
pip install requests
```

60. The Walrus Operator

Explanation:

Introduced in Python 3.8, the walrus operator (`:=`) allows you to assign a value to a variable as part of an expression.

Code Example:

```
if (n := len([1, 2, 3])) > 2:
    print(f"List has {n} elements.")
```

61. The Dis Module

Explanation:

The `dis` module in Python allows you to disassemble Python byte code, which can be useful for understanding low-level details.

Code Example:

```
import dis

def my_function():
    a = 10
    b = 20
    return a + b

dis.dis(my_function)
```

62. Functools.lru_cache

Explanation:

The `functools.lru_cache` decorator caches the results of function calls, providing a simple way to speed up functions that compute the same results repeatedly.

Code Example:

```
from functools import lru_cache

@lru_cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

63. The `else` Clause in Loops

Explanation:

An `else` clause after a loop (for/while) will run if the loop finishes normally (i.e., without encountering a `break` statement).

Code Example:

```
for i in range(3):
    print(i)
else:
    print("Loop finished without break.")
```

64. Function Parameter Unpacking

Explanation:

You can use the asterisk (`*`) to unpack lists/tuples and the double asterisk (`**`) to unpack dictionaries when calling a function.

Code Example:

```
def function(a, b, c):
    return a + b + c
```

By: Waleed Mousa

```
args = [1, 2, 3]
print(function(*args))  # Output: 6
```

65. The ast Module

Explanation:

The Abstract Syntax Tree (ast) module in Python allows you to interact with Python source code programmatically, such as analyzing, modifying, or even generating code.

Code Example:

```
import ast

source_code = 'a = 5 + 2'
parsed_code = ast.parse(source_code)
print(ast.dump(parsed_code))
```

66. Contextlib.suppress

Explanation:

contextlib.suppress is a context manager that suppresses specified exceptions if they occur within the block.

Code Example:

```
from contextlib import suppress

with suppress(ZeroDivisionError):
    result = 1 / 0
print("Code continues without raising an exception.")
```

67. Operator Overloading

Explanation:

Python allows classes to define custom behavior for operators by implementing special methods (e.g., __add__, __mul__).

Code Example:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

68. The `array` Module

Explanation:

The `array` module defines a sequence data structure that looks like a list but contains elements of the same type stored in a more memory-efficient way.

Code Example:

```
from array import array

arr = array('d', [1.0, 2.5, 3.1])
```

69. Monkey Patching

Explanation:

Monkey patching involves modifying or extending other classes or modules at runtime, usually for debugging or enhancing functionality.

Code Example:

```
def new_function(self):
    print("New functionality added.")

SomeClass.new_function = new_function
```

70. Generators with `yield from`

Explanation:

Using `yield from`, you can delegate part of its operations to another generator, simplifying the code.

Code Example:

```python
def chain(*iterables):
    for iterable in iterables:
        yield from iterable

result = list(chain([1, 2], [3, 4]))  # Output: [1, 2, 3, 4]
```

71. The shutil Module

Explanation:

The shutil module provides functions to handle high-level file operations, such as copying or moving files and directories.

Code Example:

```python
import shutil

shutil.copyfile('source.txt', 'destination.txt')
```

72. collections.Counter

Explanation:

collections.Counter is a subclass of the dictionary for counting hashable objects.

Code Example:

```python
from collections import Counter

count = Counter(['a', 'b', 'c', 'a', 'b', 'b'])
print(count)  # Output: Counter({'b': 3, 'a': 2, 'c': 1})
```

73. The pathlib Module

Explanation:

The pathlib module offers a set of classes to handle filesystem paths, providing a more object-oriented way to work with file paths.

Code Example:

```
from pathlib import Path

path = Path('folder/file.txt')
print(path.exists())
```

74. Descriptors

Explanation:

Descriptors provide a way to customize attribute access using methods like `__get__`, `__set__`, and `__delete__`.

Code Example:

```
class Descriptor:
    def __get__(self, instance, owner):
        print("Getting the attribute")

class MyClass:
    attribute = Descriptor()
```

75. `heapq` Module

Explanation:

The `heapq` module provides functions for implementing heaps (priority queues) based on regular lists.

Code Example:

```
import heapq

heap = [3, 1, 4, 1, 5]
heapq.heapify(heap)
print(heapq.heappop(heap))  # Output: 1
```

76. Named Expressions with `_`

Explanation:

In the REPL (interactive shell), you can use `_` to access the result of the last expression.

Code Example (in REPL):

```
>>> 5 + 2
7
>>> _ + 3
10
```

77. Enumerations with enum

Explanation:

The enum module defines a way to create enumerations, a set of symbolic names bound to unique constant values.

Code Example:

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

78. Virtual Environments with venv

Explanation:

The venv module provides support for creating lightweight, isolated Python environments, each with its own installation directories and dependencies.

Code Example:

```
python3 -m venv myenv
```

79. The queue Module

Explanation:

The queue module provides a way to create thread-safe queues that are used to safely communicate between threads.

Code Example:

```
from queue import Queue

q = Queue()
q.put('item')
item = q.get()
```

80. The dataclasses Module

Explanation:

Introduced in Python 3.7, dataclasses provide a decorator and functions for automatically adding special methods to classes.

Code Example:

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float
```

81. Recursive Functions

Explanation:

Recursive functions call themselves as part of their execution, often used for problems that can be divided into simpler subproblems.

Code Example:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

82. The itertools Module

Explanation:

The itertools module provides a collection of fast, memory-efficient tools for working with iterators, like combinations, permutations, and groupby.

Code Example:

```
from itertools import combinations

comb = combinations([1, 2, 3], 2)
print(list(comb))  # Output: [(1, 2), (1, 3), (2, 3)]
```

84. Using `locals()` and `globals()`

Explanation:

The `locals()` function returns a dictionary of the current namespace, and `globals()` returns a dictionary of the global namespace.

Code Example:

```
def my_function():
    x = 10
    print(locals())

my_function()  # Output: {'x': 10}
```

85. Co-routines with `yield`

Explanation:

Coroutines are a more generalized form of subroutines. They can be entered, exited, and resumed at many different points using the `yield` keyword.

Code Example:

```
def my_coroutine():
    while True:
        received = yield
        print('Received:', received)

coro = my_coroutine()
next(coro)
coro.send(42)  # Output: Received: 42
```

86. zip_longest from itertools

Explanation:

zip_longest zips input iterables, filling missing values with a specified fill value.

Code Example:

```
from itertools import zip_longest

result = zip_longest([1, 2], [3, 4, 5], fillvalue=0)
print(list(result))  # Output: [(1, 3), (2, 4), (0, 5)]
```

87. unittest.mock

Explanation:

unittest.mock provides tools for mocking objects in unit tests, allowing you to replace parts of your system under test with mock objects.

Code Example:

```
from unittest.mock import MagicMock

mock = MagicMock(return_value=42)
print(mock())  # Output: 42
```

89. Multithreading with the threading Module

Explanation:

The threading module enables concurrent programming using threads, allowing multiple operations to run simultaneously.

Code Example:

```
import threading

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

By: Waleed Mousa

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

90. The `__slots__` Attribute

Explanation:

`__slots__` is an attribute you can add to a class to save memory by preventing the dynamic creation of attributes.

Code Example:

```
class MyClass:
    __slots__ = ['name', 'age']
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

91. Bitwise Operators

Explanation:

Bitwise operators allow manipulation of individual bits in an integer, such as bitwise AND `&`, OR `|`, and XOR `^`.

Code Example:

```
x = 5  # Binary: 101
y = 3  # Binary: 011
print(x & y)  # Output: 1 (Binary: 001)
```

92. The `bz2` and `gzip` Modules

Explanation:

These modules provide tools for working with compressed files using the bzip2 and gzip file formats, respectively.

Code Example:

```
import bz2

content = b"Compress this content"
```

```
compressed_content = bz2.compress(content)
```

93. `memoryview` Objects

Explanation:

`memoryview` gives you shared memory access to data without copying it, allowing efficient manipulation of large data sets.

Code Example:

```
data = bytearray(b"Hello, world!")
view = memoryview(data)
chunk = view[7:13]
print(bytes(chunk))  # Output: b'world!'
```

94. Exception Chaining

Explanation:

In Python 3, you can use the `from` keyword to chain exceptions, which helps in understanding the cause-effect relationship between exceptions.

Code Example:

```
try:
    int("not_a_number")
except ValueError as e:
    raise TypeError("Something went wrong") from e
```

95. Deprecation Warnings with `warnings`

Explanation:

The `warnings` module allows you to warn users about changes in your code or deprecations.

Code Example:

```
import warnings

def old_function():
    warnings.warn("This function is deprecated", DeprecationWarning)
```

96. Context Variables with `contextvars`

Explanation:

`contextvars` module provides context variables, which are variables that have different values depending on their context, such as within different threads or asynchronous tasks.

Code Example:

```
from contextvars import ContextVar

var = ContextVar('var', default='default value')
```

97. `as_integer_ratio` Method

Explanation:

Float objects have a method called `as_integer_ratio`, which returns a pair of integers representing the floating-point number as a fraction.

Code Example:

```
x = 0.75
print(x.as_integer_ratio())  # Output: (3, 4)
```

98. The `cmd` Module for CLI Applications

Explanation:

The `cmd` module provides a simple framework for building command-line interfaces (CLI).

Code Example:

```
import cmd

class MyShell(cmd.Cmd):
    def do_greet(self, line):
        print("Hello, World!")

shell = MyShell()
shell.cmdloop()
```

By: Waleed Mousa

99. Property Decorators

Explanation:

Property decorators (`@property`, `@<name>.setter`, and `@<name>.deleter`) allow you to manage the attributes of a class through methods.

Code Example:

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value
```

100. The `codecs` Module

Explanation:

The `codecs` module provides functions for encoding and decoding data, including handling various text encodings.

Code Example:

```
import codecs

encoded = codecs.encode('text', 'base64')
print(encoded)  # Output: b'dGV4dA==\n'
```

101. Python's `__main__` Idiom

Explanation:

The `if __name__ == "__main__":` idiom allows you to write code that can be both used by other programs via `import` and run standalone.

Code Example:

```
def main():
    print("This script is running standalone")

if __name__ == "__main__":
    main()
```

102. The `inspect` Module

Explanation:

The `inspect` module provides several useful functions to get information about live objects such as modules, classes, methods, etc.

Code Example:

```
import inspect

def my_function():
    pass

print(inspect.isfunction(my_function))  # Output: True
```