☑ 5. Python — 50 preguntas técnicas

1. Explica la diferencia entre listas, tuplas y sets.

- **Lista (`list`)**: mutable, ordenada, permite elementos duplicados.
- `lst = [1, 2, 2, 3]`
- `lst.append(4)`
- **Tupla (`tuple`)**: inmutable, ordenada, permite duplicados.
- `tup = (1, 2, 2, 3)`
- **Set (`set`)**: mutable, no ordenado, no permite duplicados.
- `st = {1, 2, 2, 3}  # resultado: {1,2,3}`

---

2. ¿Qué es un diccionario y cómo funciona internamente?

- Colección de **pares clave-valor** (`dict`).
- Internamente usa **hash tables**, por eso el acceso es O(1).

```
d = {"name": "Alice", "age": 25}
print(d["name"])  # Alice
```

---

3. ¿Qué es list comprehension y cuándo utilizarlo?

- Sintaxis compacta para **crear listas** de manera declarativa.

```
squares = [x**2 for x in range(5)]
# [0,1,4,9,16]
```

- Útil para reemplazar loops simples y más legible.

---

4. Explica la diferencia entre deep copy y shallow copy.

- **Shallow copy:** copia solo el objeto externo, referencias internas compartidas.
- **Deep copy:** copia recursivamente el objeto y sus contenidos.

```
import copy
lst1 = [[1,2],[3,4]]
lst2 = copy.copy(lst1)    # shallow
lst3 = copy.deepcopy(lst1) # deep
```

---

5. ¿Qué es una función lambda?

- Función anónima de una línea.

```
f = lambda x: x**2
print(f(5))  # 25
```

---

6. ¿Qué son los decorators y para qué se usan?

- Funciones que modifican el comportamiento de otras funciones o métodos.

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

@decorator
def hello():
    print("Hello")
hello()
```

---

7. ¿Qué es un context manager y cómo funciona with?

- Gestiona recursos automáticamente (`open`, conexiones, locks).

```
with open("file.txt") as f:
    content = f.read()
# cierra el archivo automáticamente
```

---

8. ¿Qué es el GIL (Global Interpreter Lock)?

- Mecanismo que evita que **múltiples threads ejecuten bytecode al mismo tiempo** en CPython.
- Afecta el multithreading en CPU-bound, pero no I/O-bound.

---

9. Diferencias entre Python multithreading y multiprocessing

Aspecto	Threading	Multiprocessing
GIL	Sí afecta	No afecta
Ideal	I/O-bound	CPU-bound
Comunicación	Compartida	Pipes/Queues

---

10. ¿Qué es un generator y cómo se usa yield?

- Función que **produce valores uno a uno**, ahorrando memoria.

```
def gen():
    for i in range(3):
        yield i
for x in gen():
    print(x)
```

---

11. ¿Cómo manejar excepciones correctamente?

- Usar `try/except` para capturar errores y evitar crash.
- Siempre manejar solo errores esperados.

---

12. Explica try/except/else/finally

```
try:
    x = 1/1
except ZeroDivisionError:
    print("Error")
else:
    print("No hubo errores")
finally:
    print("Siempre se ejecuta")
```

---

13. ¿Qué es la programación orientada a objetos en Python?

- Modelo que organiza código en **clases y objetos**, con atributos y métodos.

---

14. Explica inheritance y polymorphism

- **Inheritance:** herencia de clases.
- **Polymorphism:** mismo método puede comportarse distinto según la clase.

```
class Animal:
    def speak(self): pass
class Dog(Animal):
    def speak(self): print("Woof")
```

---

15. Diferencias entre métodos @staticmethod y @classmethod

- **staticmethod:** no recibe `self` ni `cls`.
- **classmethod:** recibe `cls` y puede modificar la clase.

---

16. ¿Qué es un dataclass?

- Decorador que genera automáticamente métodos `__init__`, `__repr__`, etc.

```
from dataclasses import dataclass
@dataclass
class Point:
    x: int
    y: int
```

---

17. Explica cómo funciona *args y **kwargs

- `*args`: recibe múltiples argumentos posicionales.
- `**kwargs`: recibe múltiples argumentos nombrados.

```
def f(*args, **kwargs):
    print(args, kwargs)
f(1,2,a=3)
```

---

18. ¿Qué son los type hints?

- Sugerencias de tipo para variables, parámetros y retornos.

```
def add(a: int, b: int) -> int:
    return a+b
```

---

19. ¿Para qué sirve typing.Optional?

- Indica que una variable puede ser de un tipo o `None`.

```
from typing import Optional
def greet(name: Optional[str]):
    if name: print(f"Hello {name}")
```

---

20. ¿Qué es un virtual environment y por qué es importante?

- Entorno aislado de Python con paquetes específicos.
- Evita conflictos entre proyectos.

---

21. ¿Qué es pip y qué es poetry?

- **pip:** gestor de paquetes tradicional de Python.
- **poetry:** gestor avanzado que maneja dependencias, entornos y publicación.

---

22. ¿Cómo manejar dependencias en un proyecto grande?

- Usar virtual environments y `requirements.txt` o `pyproject.toml`.
- Lock files (`poetry.lock`, `pip freeze`) para reproducibilidad.

---

23. Diferencia entre .py, .pyc y pycache

- **.py:** código fuente.
- **.pyc:** bytecode compilado.
- **pycache:** carpeta donde se guardan archivos .pyc.

---

24. ¿Qué son los módulos y paquetes en Python?

- **Módulo:** archivo `.py`.
- **Paquete:** carpeta con `__init__.py` que contiene módulos.

---

25. ¿Cómo funciona la importación relativa?

- Importa módulos usando rutas relativas desde el paquete.

```
from .mymodule import func
```

---

26. ¿Qué es un iterator?

- Objeto que implementa `__iter__()` y `__next__()`.
- Permite recorrer elementos uno a uno.

---

27. Explica los métodos mágicos init, str, repr

- `__init__`: constructor.
- `__str__`: cadena legible para humanos.
- `__repr__`: cadena para desarrolladores / debug.

---

28. ¿Qué es asyncio y cuándo usarlo?

- Librería para **concurrencia asíncrona** en Python.
- Ideal para I/O-bound como requests HTTP.

---

29. Explica await, async y event loop

- **async def**: define función asíncrona.
- **await**: espera resultado de coroutine.
- **event loop**: ejecuta coroutines.

---

30. ¿Cómo leer y escribir archivos grandes eficientemente?

- Usar **chunks** o iteradores.

```
with open("big.txt") as f:
    for line in f:
        process(line)
```

---

31. ¿Qué es un memory leak en Python?

- Objetos no liberados por el GC, ocupando memoria innecesaria.

---

32. ¿Qué es el garbage collector?

- Módulo que libera memoria de objetos no referenciados.

---

33. Explica PEP 8 y por qué es importante

- Guía de estilo oficial de Python.
- Mejora legibilidad y consistencia del código.

---

34. Librerías para manipular datos

- **pandas:** DataFrames, CSV, Excel.
- **numpy:** arrays, operaciones vectorizadas.

---

35. ¿Cómo optimizar el rendimiento de pandas?

- Usar tipos adecuados (`category`), vectorización, filtrado selectivo.

---

36. Explica la vectorización en numpy

- Reemplaza loops por operaciones sobre arrays completas, más rápido.

```
import numpy as np
a = np.array([1,2,3])
b = a * 2  # vectorizado
```

---

37. ¿Cómo ejecutar consultas SQL desde Python?

- Usar **sqlite3**, `psycopg2` o `SQLAlchemy`.

```
import sqlite3
conn = sqlite3.connect("db.db")
cur = conn.cursor()
cur.execute("SELECT * FROM users")
```

---

38. ¿Qué es SQLAlchemy?

- ORM y toolkit para manejar bases de datos en Python.
- Permite consultas SQL y mapeo objeto-relacional.

---

39. ¿Cómo crear un API con FastAPI?

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/hello")
def read_root():
    return {"message": "Hello"}
```

---

40. ¿Qué es un dependency injection en FastAPI?

- Patrón para **inyectar dependencias** (DB, config) en endpoints de manera declarativa.

---

41. ¿Cómo manejar logs en Python?

- Usar módulo **logging**.

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info("Mensaje")
```

---

42. ¿Cómo escribir pruebas unitarias con pytest?

- Definir funciones que comiencen con `test_`.

```
def test_sum():
    assert 1+1 == 2
```

---

43. ¿Qué es un fixture en pytest?

- Función que prepara datos o recursos para tests.

```
import pytest
@pytest.fixture
def sample_data():
    return [1,2,3]
```

---

44. ¿Cómo usar mocking en pruebas?

- Simula funciones o servicios externos para tests.

```
from unittest.mock import Mock
mock = Mock()
mock.return_value = 5
```

---

45. ¿Cómo serializar datos en JSON?

```
import json
```

```
data = {"name":"Alice"}
json_str = json.dumps(data)
```

---

46. Explica el manejo de fechas con datetime

```
from datetime import datetime
now = datetime.now()
dt_str = now.strftime("%Y-%m-%d")
```

---

47. ¿Cómo asegurar que un programa es escalable?

- Diseñar modular, usar procesamiento paralelo y optimizar memoria.

---

48. ¿Cómo manejar concurrencia en acceso a archivos?

- Locks, `threading.Lock`, o colas para serializar accesos.

---

49. Consideraciones al procesar millones de registros

- Procesamiento por chunks, streams, vectorización y memoria eficiente.

---

50. ¿Cómo crear un paquete Python distribuible?

- Crear `setup.py` o `pyproject.toml`, definir módulos, usar `setuptools` y publicar en PyPI.