

# Entrevista para Programador Python

## Sección 1: Preguntas Generales de Python

1. Explica la diferencia entre `deepcopy` y `copy` en Python. ¿En qué casos usarías cada una?
  - `copy.copy()` crea una copia superficial del objeto, lo que significa que los objetos anidados siguen referenciando las mismas instancias.
  - `copy.deepcopy()` crea una copia completamente nueva, incluyendo objetos anidados.
  - Se usa `copy()` cuando solo se necesita duplicar la estructura externa y `deepcopy()` cuando se requiere independencia total entre copias.
2. Explica la diferencia entre `asyncio`, `threading` y `multiprocessing`. ¿Cuándo usarías cada uno?
  - `asyncio`: Maneja tareas asíncronas dentro de un solo hilo, útil para I/O concurrente sin bloqueos.
  - `threading`: Usa múltiples hilos dentro de un proceso, adecuado para tareas que requieren paralelismo en I/O.
  - `multiprocessing`: Crea procesos separados para aprovechar múltiples núcleos de CPU, ideal para cargas computacionales intensivas.
3. ¿Qué es un `decorator` en Python? ¿Cómo lo implementarías?
  - Los decoradores son funciones que modifican el comportamiento de otras funciones o métodos sin modificar su código fuente. Se utilizan para funcionalidades como logging, autenticación y control de acceso.
  - Ejemplo:

```
def decorator(func):
    def wrapper():
        print("Antes de la función")
        func()
        print("Después de la función")
    return wrapper

@decorator
def saludar():
    print("Hola")

saludar()
```

4. ¿Cómo manejarías errores en una aplicación backend para evitar que una excepción no manejada derribe el servicio?

- Usar bloques `try-except` para manejar excepciones específicas.
  - Registrar errores con `logging` para diagnóstico.
  - Implementar middleware de manejo de errores en frameworks como Flask/FastAPI.
  - Retornar respuestas HTTP adecuadas (ej. `400 Bad Request`, `500 Internal Server Error`).
5. ¿Qué ventajas e inconvenientes tiene el uso de `dataclasses` en Python?
- Ventajas: Menos código repetitivo, mejor legibilidad, soporte para `__repr__`, `__eq__`, entre otros.
  - Inconvenientes: No ofrecen validación de tipos nativa, son menos flexibles que las clases tradicionales.
6. ¿Qué es una `race condition` y cómo la mitigarías en Python?
- Ocurre cuando múltiples hilos o procesos acceden y modifican una variable compartida simultáneamente, causando resultados inesperados.
  - Se mitiga con:
    - Uso de `threading.Lock()` para asegurar acceso exclusivo.
    - Uso de colas (`queue.Queue()`) para sincronizar tareas.
    - Implementación de procesos separados con `multiprocessing` para evitar memoria compartida.
7. Explica la diferencia entre `list`, `tuple` y `set`.
- `list` es mutable y ordenada, `tuple` es inmutable y ordenada, `set` es mutable pero no ordenado y no permite duplicados.
8. ¿Qué es el GIL (Global Interpreter Lock) en Python y cómo afecta la concurrencia?
- El GIL es un bloqueo que permite que solo un thread ejecute código de Python a la vez, lo que puede limitar la concurrencia en programas multi-threaded. Para tareas CPU-intensivas, se recomienda `multiprocessing` en lugar de `threading`.
- 

## Sección 2: Preguntas Específicas de Django

1. Explica el patrón de arquitectura Model-Template-View (MTV) en Django. ¿Cómo se relaciona con MVC?
- Django sigue el patrón MTV:
    - **Model:** Define la estructura de la base de datos y su lógica de negocio.
    - **Template:** Maneja la presentación en HTML con sintaxis de plantillas de Django.
    - **View:** Procesa las solicitudes y devuelve respuestas usando modelos y templates.
  - Es similar a MVC, pero en Django el framework maneja la capa del controlador implícitamente.
2. ¿Cómo funciona el sistema de migraciones en Django?

- Django usa migraciones para gestionar cambios en la base de datos.
  - Se generan con `python manage.py makemigrations` y se aplican con `python manage.py migrate`.
  - Las migraciones permiten versionar los cambios en modelos sin afectar datos existentes.
3. ¿Cómo funciona el ORM de Django y qué ventajas tiene sobre escribir SQL directamente?
- El ORM (Object-Relational Mapper) permite interactuar con la base de datos usando objetos Python en lugar de SQL. Mejora la seguridad y facilita la portabilidad entre distintos motores de base de datos.
4. ¿Qué es un QuerySet y cómo optimizar consultas en Django ORM?
- Un QuerySet es una colección de objetos de base de datos representados en Python.
  - Optimización:
    - `select_related`: Reduce consultas al hacer joins en relaciones `ForeignKey`.
    - `prefetch_related`: Recupera relaciones `ManyToMany` o `reverse ForeignKey` en menos consultas.
    - `only` y `defer`: Permiten cargar solo ciertos campos del modelo para mejorar rendimiento.
5. ¿Cómo manejarías la serialización de datos en Django?
- Usando `serializers` de Django REST Framework (DRF).
  - Implementación básica:

```
from rest_framework import serializers
from .models import Usuario

class UsuarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = Usuario
        fields = '__all__'
```

6. ¿Cómo implementarías autenticación y autorización en Django REST Framework?
- DRF ofrece varios métodos como `TokenAuthentication`, `SessionAuthentication` y `JWT`.
  - Configuración en `settings.py`:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
```

```
    ),  
}
```

#### 7. ¿Cómo realizar pruebas en Django?

- Django ofrece TestCase basado en unittest. Se pueden usar métodos como `self.client.get()` para probar vistas y endpoints.

---

### Sección 3: Preguntas Específicas de Flask y FastAPI

#### 1. ¿Qué es Flask y cuándo conviene usarlo?

- Flask es un microframework para aplicaciones web en Python. Se usa cuando se necesita flexibilidad y no se requiere una estructura rígida como Django.

#### 2. ¿Cómo funcionan los Blueprints en Flask?

- Los Blueprints permiten modularizar una aplicación Flask en componentes reutilizables.

#### 3. ¿Cómo manejar rutas y solicitudes en Flask?

- Flask define rutas con decoradores:

```
from flask import Flask  
app = Flask(__name__)  
  
@app.route('/')  
def home():  
    return "Hola Mundo"
```

#### 4. ¿Qué diferencias hay entre Flask-SQLAlchemy y SQLAlchemy puro?

- Flask-SQLAlchemy es una extensión que facilita la integración de SQLAlchemy con Flask, manejando automáticamente el ciclo de vida de las sesiones y la configuración de la aplicación.

#### 5. ¿Qué ventajas ofrece FastAPI sobre Flask?

- FastAPI usa tipado de Python para validaciones automáticas, genera documentación OpenAPI y ofrece mejor rendimiento gracias a Starlette y Pydantic.

#### 6. ¿Cómo implementas validación de datos en FastAPI?

- FastAPI usa Pydantic para definir esquemas de validación:

```
from fastapi import FastAPI  
from pydantic import BaseModel
```

```

app = FastAPI()

class Usuario(BaseModel):
    nombre: str
    edad: int

@app.post("/usuarios/")
def crear_usuario(usuario: Usuario):
    return {"mensaje": "Usuario creado", "usuario": usuario}

```

### 7. ¿Cómo se manejan dependencias en FastAPI?

- FastAPI usa el sistema de `Depends()` para inyectar dependencias en rutas, útil para autenticación, bases de datos y validaciones.

### 8. ¿Cómo realizar pruebas en FastAPI?

- Se pueden usar `TestClient` de `fastapi.testclient` para realizar pruebas de endpoints sin ejecutar el servidor.

### 9. ¿Cómo manejarías errores globalmente en Flask y FastAPI?

- Flask: Usando `errorhandler`:

```

@app.errorhandler(404)
def pagina_no_encontrada(e):
    return "Página no encontrada", 404

```

- FastAPI: Con `ExceptionHandler`:

```

from fastapi import HTTPException

collection = [{"id": 1, "nombre": "Manzana"}, {"id": 2, "name": "Naranja"}, {"id": 3, "name": "Banana"}]

@app.get("/items/{item_id}", response_model=dict)
def get_item(item_id: int):
    item = next((x for x in collection if x["id"] == item_id), None)

    if item:
        return {"item": item}

    raise HTTPException(status_code=404, detail="Item no encontrado")

```