códigofacilito

Clase 4. Bootcamp AI Engineer con Python

Outputs Estructurados y Contratos

Function Calling + Pydantic: De texto libre a datos accionables

“El LLM habla en prosa. Tu base de datos habla en SQL.”

códigofacilito

Lo que aprenderás hoy

1. Por qué necesitamos salidas estructuradas en producción

2. Dominar Pydantic como sistema de contratos de datos

3. Usar JSON Mode y Structured Outputs de OpenAI

4. Implementar Function Calling (Tool Use)

5. Construir un extractor de texto desordenado a JSON validado

```python
from pydantic import BaseModel, EmailStr

class Contacto(BaseModel):
    nombre: str
    email: EmailStr
    edad: int | None = None

c = Contacto(nombre="Ana García", email="ana@corp.com")
print(c.model_dump_json())
# {"nombre": "Ana García", "email": "ana@corp.com", "edad": null}


c = Contacto(nombre="Ana", email="no-es-un-email")
# ValidationError: value is not a valid email address
```

Si el dato no cumple el contrato, Pydantic lo rechaza.

¿Por qué Pydantic + LLMs?

LLM (Impredecible)

Cada respuesta puede variar. Genera texto libre, formatos diferentes cada vez. Sin garantías de estructura.

Backend (Determinista)

Espera campos exactos, tipos fijos, validación estricta. Si el dato no cumple el contrato, error.

Pydantic para datos complejos – Modelos anidados y listas

```python
class Ingrediente(BaseModel):
    nombre: str
    cantidad: str
    unidad: str | None = None

class Receta(BaseModel):
    titulo: str
    tiempo_minutos: int
    dificultad: str
    ingredientes: list[Ingrediente]
    pasos: list[str]
```

```python
# JSON esperado del LLM:
# {
#   "titulo": "Pasta Carbonara",
#   "tiempo_minutos": 25,
#   "ingredientes": [
#     {"nombre": "spaghetti",
"cantidad": "400", "unidad": "g"}
#   ],
#   "pasos": ["Hervir agua...",
"Mezclar huevos..."]
# }
```

```python
response = client.responses.create(
    model="gpt-4o-mini",
    response_format={"type": "json_object"},
    messages=[
        {
            "role": "system",
            "content": "Responde SOLO con JSON válido con los campos: "
                       "nombre (str), email (str), empresa (str o null)."
        },
        {
            "role": "user",
            "content": "Hola, soy Carlos Méndez de TechCorp, "
                       "mi correo es carlos@techcorp.mx"
        }
    ]
)

data = json.loads(response.output_text)
# {"nombre": "Carlos Méndez", "email": "carlos@techcorp.mx"}
```

⚠️ Limitación: devuelve JSON, pero no garantiza los

```python
class DatosContacto(BaseModel):
    nombre: str
    email: str
    empresa: str | None

response = client.responses.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "Extrae los datos de contacto."},
        {"role": "user", "content": "Me llamo Ramsés Camas, doy clases "
                    "en codigofacilito, mail: ramsescamas@gmail.com"}
    ],
    response_format=DatosContacto
)

contacto = response.output_parsed

print(contacto.nombre)    # Ramsés Camas
print(contacto.empresa)   # codigofacilito
print(type(contacto))     # DatosContacto
```

✅ Ya no parseas JSON. Recibes un objeto Pydantic validado.

códigofacilito

JSON Mode vs Structured Outputs

	Característica	JSON Mode	Structured Outputs	Recomendación
Garantía	Solo JSON válido	Respeta el schema Pydantic	Structured Outputs	Actualización constante de contenidos para estar a la vanguardia del mundo tech.
Validación	Manual (tú parseas)	Automática	Structured Outputs	Siempre automática
Campos req.	Puede omitirlos	Se respetan	Structured Outputs	Garantizado
Caso de uso	Prototipo rápido / Producción confiable	Exploración flexible	Pipelines confiables	Siempre en prod
Regla	Prototipo → JSON Mode | Producción → Structured Outputs			

códigofacilito

Nivel 3: Function Calling (Tool Use)

Devuelves el resultado al modelo para la respuesta final

- **1. Tú defines funciones disponibles (tools) al modelo** Define las tools con nombre, descripción y parámetros JSON Schema
- **2. El usuario envía un mensaje natural** El mensaje va con la lista de tools disponibles
- **3. El modelo analiza si necesita llamar alguna función** Decide qué función usar basándose en el contexto del usuario
- **4. En caso de que si → genera los argumentos en JSON** El modelo NO ejecuta nada — solo genera los argumentos
- **5. Se ejecuta la función con esos argumentos** Llamadas a la función real con los argumentos generados

Function Calling: Paso 1 — Describe tus funciones

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "buscar_producto",
            "description": "Busca un producto en el inventario.",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": "Nombre o categoría"
                    },
                    "precio_max": {
                        "type": "number",
                        "description": "Precio máximo en MXN"
                    }
                },
                "required": ["query"]
            }
        }
    }
]
```

La descripción importa mucho — es documentación para el LLM.

```python
messages = [
    {"role": "system", "content": "Eres un asistente de tienda."},
    {"role": "user", "content": "¿Tienen laptops por menos de 15,000?"}
]

response = client.responses.create(
    model="gpt-4o-mini",
    messages=messages,
    tools=tools
)

message = response.output_text


if message.tool_calls:
    tool_call = message.tool_calls[0]
    args = json.loads(tool_call.function.arguments)
    print(f"Función: {tool_call.function.name}")
    print(f"Argumentos: {args}")
    # Función: buscar_producto
    # Argumentos: {"query": "laptops", "precio_max": 15000}
```

El modelo NO ejecutó nada — solo generó los argumentos.

```
resultado = buscar_producto(**args)
# [{"nombre": "Lenovo IdeaPad", "precio": 12999},
#  {"nombre": "HP 245 G9", "precio": 9499}]

messages.append(message)
messages.append({
    "role": "tool",
    "tool_call_id": tool_call.id,
    "content": json.dumps(resultado)
})

final = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
    tools=tools
)
print(final.output_text)
# "¡Sí! Tenemos 2 laptops:
#  - Lenovo IdeaPad: $12,999 MXN
#  - HP 245 G9: $9,499 MXN"
```

El modelo convierte datos crudos en respuesta natural.

Function Calling + Pydantic combinados

```python
from pydantic import BaseModel, Field

class BuscarProductoArgs(BaseModel):
    query: str = Field(
        description="Nombre o categoría del producto"
    )
    precio_max: float | None = Field(
        default=None,
        description="Precio máximo en MXN"
    )

# Generar el JSON Schema automáticamente
print(BuscarProductoArgs.model_json_schema())
# {
#   "properties": {
#     "query": {"description": "...", "type": "string"},
#     "precio_max": {"description": "...", "type": "number"}
#   },
#   "required": ["query"],
#   "type": "object"
# }
```

Nunca más escribes JSON schemas a mano. Un solo source of truth.

códigofacilito

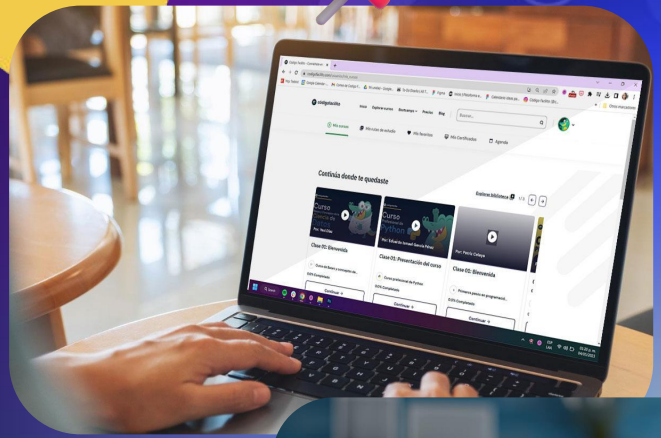
El panorama completo: ¿Cuándo

	Necesidad	Solución	Ejemplo	Cuándo
Datos JSON	"Dame datos en JSON"	Structured Outputs	Extraer entidades de un email	Actualización constante de contenidos para estar a la vanguardia del mundo tech.
Ejecutar código	"Decide si ejecutar código"	Function Calling	Asistente que busca en DB	Cuando el LLM necesita herramientas
Validar datos	"Valida antes de guardar"	Pydantic	Verificar tipos y formatos	Siempre, en todo pipeline
Producción	Las 3 juntas	Function Calling + Pydantic + Structured Outputs	Pipeline completo end-to-end	En producción real
Regla	Function Calling decide QUÉ hacer → Pydantic valida args → Structured Outputs formatea respuesta			

códigofacilito

/04

Live Coding: Extractor de Recetas



Práctica Paso 1 – Definir el contrato Pydantic

```python
class Ingrediente(BaseModel):
    nombre: str
    cantidad: str = Field(
        description="Cantidad numérica o descriptiva"
    )
    unidad: str | None = Field(
        default=None,
        description="g, ml, kg, piezas, o null si es 'al gusto'"
    )

class Receta(BaseModel):
    titulo: str
    tiempo_minutos: int
    dificultad: str = Field(
        description="fácil, media o difícil"
    )
    ingredientes: list[Ingrediente]
    pasos: list[str] = Field(
        description="Lista ordenada de instrucciones"
    )
```

Práctica Paso 2 – Structured Outputs en acción

```
texto_receta = "Para hacer unos chilaquiles verdes necesitas
como medio kilo de totopos, unos 200ml de salsa verde,
crema al gusto, queso fresco 150g, y cebolla. Primero
calienta la salsa, agrega los totopos. Sirve con crema,
queso y cebolla. Tarda como 15 minutos."

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "system",
            "content": "Eres un chef experto. Extrae recetas "
                       "de texto desordenado en formato "
                       "estructurado. Sé preciso."
        },
        {"role": "user", "content": texto_receta}
    ],
    response_format=Receta
)

receta = response.output_parsed
```

Práctica Paso 3 — Datos listos para producción

```python
print(receta.titulo)        # "Chilaquiles Verdes"
print(f"Tiempo: {receta.tiempo_minutos} min")  # 15 min

for ing in receta.ingredientes:
    print(f"  - {ing.cantidad} {ing.unidad or ''} {ing.nombre}")
# - 500 g totopos
# - 200 ml salsa verde
# - al gusto  crema
# - 150 g queso fresco
# - 1 pieza cebolla

for i, paso in enumerate(receta.pasos, 1):
    print(f"  {i}. {paso}")
# 1. Calentar la salsa verde en una sartén
# 2. Agregar los totopos y mezclar
# 3. Servir con crema, queso y cebolla

print(receta.model_dump_json(indent=2))
```

De texto desordenado → datos validados en menos de 20 líneas.

códigofacilito

Resumen: Lo que aprendiste hoy

- **Pydantic:** Define la forma exacta de tus datos y valida automáticamente. Es tu contrato de datos.
- **JSON Mode:** Fuerza al modelo a responder en JSON, pero sin garantía de esquema específico.
- **Structured Outputs:** El modelo respeta tu esquema Pydantic al 100%. Recibes objetos validados directamente.
- **Function Calling:** El modelo decide cuándo invocar funciones externas y genera los argumentos en JSON.
- **Producción:** Las tres técnicas se combinan para pipelines confiables de extremo a extremo.

códigofacilito

Desafío para la próxima clase

Tu turno — Extiende el extractor y aplicalo

- **Reto 1:** Agrega campo porciones: int al modelo Receta y prueba con diferentes textos
- **Reto 2:** Crea un modelo Factura con campos: emisor, receptor, conceptos, total, fecha
- **Reto 3:** Implementa reintentos — si Pydantic rechaza, reenvía al modelo con el error
- **Bonus:** Usa Function Calling para decidir entre extraer receta o facturar según el texto