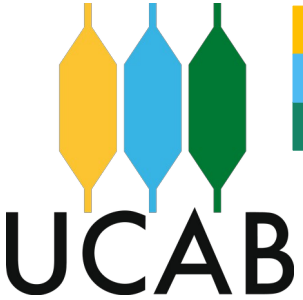

Programación de APIs con Python

Type Hints / Funciones Generadoras

Profesor:
José Gregorio Castillo Pacheco

Métodos Dunder

Temario

- Type Hints
- Métodos dunder
- Funciones Generadoras

Type Hints

- El manejo de tipo dinámico puede ser bueno en ciertas ocasiones, pero cuando desarrollamos aplicaciones empresariales es mejor restringir los tipos a usar por el usuario.
- Los desarrolladores que trabajan en Python entendieron que hacía falta manejar los tipos en este lenguaje y a partir de la versión 3.5 de Python se agregó el módulo `typing`
- Las sugerencias de tipo (type hints) suministran información acerca del tipo de datos esperados en variables, en los parámetros de las funciones y los tipos de retorno

Type Hints

Razones de la introducción de los type hints en Python:

- **Legibilidad y claridad**:
  - Los *type hint* favorecen la autodocumentación del código
  - Los desarrolladores pueden comprender más rápidos los tipos de variables y los parámetros de las funciones
  - Las anotaciones del tipo precisas mejoran la legibilidad del código.
- **Verificación estática del código**:
  - Python es dinámicamente tipado (los tipos se determinan en tiempo de ejecución)
  - Los *type hints* permiten a los verificadores de tipo (por ejemplo mypy) capturar errores relacionados al tipo antes del tiempo de ejecución.
  - La detección temprana conlleva a una mejor calidad de software

Type Hints

Razones de la introducción de los type hints en Python:

- **Se evitan bugs**:
  - Los conflictos de tipo tienden a producir errores frustantes en tiempo de ejecución
  - Los *type hint* ayudan a prevenir errores comunes
- **Verificación estática del código**:
  - IDEs (como VS Code, PyCharm) usan los type hints para mejorar la autocompletación y las sugerencias
  - Linters (verificadores, como flake 8) refuerzan los estándares de codificación basados en sugerencias de tipo.

Type Hints

Anotaciones básicas de tipo:

- **Type hints en variables**:
  - Se declara el tipo de la variable usando la sintáxis de dos puntos (:)
  - Ejemplo: `name: str = “Pepe”`
- **Type hints en funciones**:
  - Se especifíca el tipo de los parámetrios y el del retorno
  - Ejemplo:

```
def greet(name: str) -> str:
    return f"Hola, {name}"
```

Type Hints

Type hints comunes:

- **Tipos “primitivos”**:
  - int, float, bool, str, None, etc
- **Colecciones**:
  - Se debe primero usar *from typyng import List, Tuple, Dict*
  - List[int], Tuple[str, int], Dict[str, float], etc <= observar que se marcan con “[“ y “]”
- **Clases propias**:
  - Defina sus propias clases y úselas como type hints

Ver ejemplo: ejemplo_1.py y ejemplo_2.py

Nota: para usar **mypy** habilitaremos un ambiente virtual

Type Hints

Type hints avanzados:

- **Unión de tipos**:
  - Especifica posibles múltiples tipos de una variable, parámetro o respuesta
  - Ejemplo:

```
from typing import Union
def process_data(data: Union[int, float]) -> str  o
def process_data(data: int | float) -> str: (desde Python 3.10+)
```

- **Tipos opcionales**:
  - Se usa Optional para valores que pueden ser anulables
  - Ejemplo:

```
from typing import Optional
def get_name() -> Optional[str]:  o
def get_name -> str | None: (desde Python 3.10+)
```

Mas ejemplos, ver:
ejemplo_3.py

Type Hints

Type hints avanzados:

- **TypeVar**: es usado para declarar un tipo de variable que puede representar mútiples tipos sin especifivar uno inicialmente. Ver ejemplos 3a y 3b
- **NewType**: es una forma de crear un nuevo tipo que se basa en un tipo existente. Es útil cuando tienes variables que conceptualmente representan cosas diferentes pero tienen el mismo tipo subyacente. Ver ejemplo 3c.
- **Final Type**: Por convención las variables se decalaran con mayúsculas sostenidas. Este type hint, introducido en Python 3.8, proporciona una manera de declarar constantes. Esta forma de declaración agrega una capa extra de segiridad de tipos al código. Por ejemplo:

```
from typing import final

DATABASE: Final = “MySQL”
```

- **TypedDict:** Se utiliza para especificar los tipos de las claves y los valores dentro de un diccionario. Se explicará más en detallae cuando se vea Pydantic. Ver ejemplo_3d.py

Type Hints

Type hints avanzados:

- **Protocol**: Es la forma “pythonica” de definir el concepto de “duck-typing”, también llamado subtipado estructural.
  - Protocol permite especificar que cualquier objeto que se ajuste a una cierta “estructura” puede ser usado como argumento para una función o como tipo para una variable. Ver ejemplo_3e.py
- **Union Type**: se utiliza para indicar que una variable, un parámetro de función o un valor de retorno puede ser de múltiples tipos diferentes. Ver ejemplo_3f.py
- **Callable Type**: se utiliza para indicar que una variable se espera que contenga una función (o cualquier otro objeto "llamable", como métodos de clases u objetos con un método __call__). Esto es particularmente útil en escenarios donde pasas funciones como argumentos a otras funciones (callbacks) o las asignas a variables para su posterior ejecución. Ver ejemplo_3g.py

Type Hints

Type hints avanzados:

- **Annotated Type**: es una característica única diseñada para enriquecer las type hints permitiendo adjuntar metadatos adicionales más allá de la información del tipo básico. Basicamente, permite añadir "anotaciones" extra a una variable, parámetro o valor de retorno, proporcionando restricciones o información adicional que no afectan la tipificación . Ver ejemplo_3h.py
- **Alias Type**: se utilizan para simplificar la reutilización de type hints complejas y para mejorar la legibilidad del código. En lugar de escribir repetidamente una type hint larga, se define un alias con un nombre más corto y descriptivo, y luego se utiliza ese alias en las anotaciones de tipo. Ver ejemplo_3i.py

Métodos dunder

Métodos Dunder

- En Python, los métodos dunder son métodos que permiten a las instancias de una clase interactuar con las funciones y operadores incorporados del lenguaje.
- La palabra "dunder" viene de "double underscore", porque los nombres de los métodos dunder empiezan y terminan con dos guiones bajos, por ejemplo __str__ o __add__.
- Normalmente, los métodos dunder no son invocados directamente por el programador, lo que hace que parezca que son llamados por arte de magia. Por eso, a veces también se hace referencia a los métodos dunder como "métodos mágicos".
- Los métodos dunder son llamados implícitamente por el lenguaje.

Métodos Dunder

- Por ejemplo:

```
class Square:
    def __init__(self, side_length):
        """__init__ es el método dunder que INITcializa la instancia.
        Para crear un cuadrado, se necesita conocer la longitud de su lado,
        de manera se der pasado como un argumento, p. ejm. con Square(1).
        Para asegurar que la instancia conoce su propia longitud de lado,
        es salvada con self.side_length = side_length.
        """
        print("Dentro del init!")
        self.side_length = side_length


sq = Square(1)
# Dentro del init!
```

Métodos Dunder

- Los dos guiones bajos al principio y al final del nombre de un método dunder no tienen ningún significado especial.
- Son usados básicamente para evitar la colisión de nombres con otros métodos.Por ejemplo:

```
>>> sum(range(10))
45
>>> sum = 45
>>> sum(range(10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Métodos Dunder

Sobrescritura de operadores con métodos dunder

- Todos los operadores de Python, como **+**, **==**, e **in**, dependen de métodos dunder para implementar su comportamiento.
- Por ejemplo, cuando Python encuentra el código `value in container`, en realidad lo convierte en una llamada al método dunder apropiado **__contains__**, lo que significa que Python en realidad ejecuta la expresión `container.__contains__(value)`.

```
>>> my_list = [2, 4, 6]

>>> 3 in my_list
False
>>> my_list.__contains__(3)
False

>>> 6 in my_list
True
>>> my_list.__contains__(6)
True
```

Métodos Dunder

Sobrescritura de operadores con métodos dunder

Operador	Método Especial	Descripción
+	__add__(self, object)	Suma
-	__sub__(self, object)	Resta
*	__mul__(self, object)	Multiplicación
**	__pow__(self, object)	Exponenciación
/	__truediv__(self, object)	División
//	__floordiv__(self, object)	División Entera
%	__mod__(self, object)	Modulo
==	__eq__(self, object)	Igual a

Métodos Dunder

Sobrescritura de operadores con métodos dunder

Operador	Método Especial	Descripción
!=	__ne__(self, object)	Diferente de
>	__gt__(self, object)	Mayor que
>=	__ge__(self, object)	Mayor o igual que
<	__lt__(self, object)	Menor que
<=	__le__(self, object)	Menor o igual que
in	__contains__(self, value)	Operador de membresía
[index]	__getitem__(self, index)	Elemento en indice
len()	__len__(self)	Calcula número de elementos
str()	__str__(self)	Convierte objeto a string

Para una lista completa de los métodos dunder ver:
https://mathspp.com/blog/pydonts/dunder-methods#list-of-dunder-methods-and-their-interactions

Métodos Dunder

Sobrescritura de operadores con métodos dunder

Ejemplo en clase:

- Ver ejemplos:
  - special_methods.py
  - point.py

Funciones Generadoras

- Nos preguntamos porque cubrimos el tema de generadores, siendo un concepto muy relacionado con Python y poco con el desarrollo de APIs.

- Su principal uso, cuando a APIs se refiere, es en la generación de sesiones de databases.

- Para entender los generadores, debemos comprender los iteradores.
- Un iterador es un objteto que puede ser iterado usando un bucle (iterable).
- La diferencia es que nos entregan un elemento a la vez.
- Un objeto iterador debe implementar 2 métodos: `__iter__()` y `__next__()`.
- También podemos usar las funciones `iter()` y `next()`, que son formas más elegantes de llamar a los métodos.
- Ver ejemplo4.py

Funciones Generadoras

- Por su puesto que al crear un objeto de tipo Iterador, podemos recorrerlo con un bucle. Cuando los datos del objeto se hayan consumido en su totalidad se genera una excepción llamada StopIteration, ver ejemplo_5.py

- Ahora bien, si queremos crear una lista infinita de números enteros, nos llevaría a un colapso de la memoria RAM de la computadora, pero si usamos un iterador, es posible hacerlo y sólo consumir los datos que necesitemos

- Ver ejemplo_6.py

Funciones Generadoras

- Usualmente vemos el termino *iterable e iterator*, Iterable es un objeto sobre el cual podemos iterar, tales como strings, listas, sets, etc. Sin embargo no podemos tomar objetos iterables como iterators.
- Python cuenta con una función que convierte un iterable en un iterator, esta se llama, como era de esperarse, **`iter()`**

Ejemplo:

```
my_string = "Python"
next(my_string)
> TypeError: 'str' object is not an iterator
```

Directamente

```
my_string_new = iter(my_string)
next(my_string_new)
'P'
next(my_string_new)
'y'
```

Usando iter()

Funciones Generadoras

- Ahora bien no deseamos sobreescribir los métodos iter() y next() para trabajar con una iteración todo el tiempo. **Para ello usamos las funciones generadoras**. Un generador o función generadora es una función que no devuelve un único valor, sino un objeto iterador (un iterable) con una secuencia de valores cuando se itera sobre él.
- La ventaja principal de los generadores es que son útiles cuando queremos producir una gran secuencia de valores, pero no queremos almacenarlos todos a la vez en memoria.
- En Python, de forma similar a la definición de una función normal, podemos definir una función generadora utilizando la palabra clave **def**, pero en lugar de la sentencia **return** utilizamos la sentencia **yield**.

Formato general:

```
def nombre_generador(arg):
    # sentencias
    yield algo
```

la palabra clave **`yield`** se utiliza para producir un valor a partir del generador.

Funciones Generadoras

Memoria eficiente:

- Una función normal para devolver una secuencia creará toda la secuencia en memoria antes de devolver el resultado. Esto es excesivo si el número de elementos de la secuencia es muy grande.

- La implementación del generador de tales secuencias es amigable con la memoria y es preferible ya que sólo produce un elemento a la vez.

Funciones Generadoras

Hay varias razones que hacen de los generadores una implementación poderosa:

- **Fácil de implementar**: Los generadores pueden ser implementados de una manera clara y concisa en comparación a la clase *iterator*.

Ejemplo:

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        resultado = 2 ** self.n
        self.n += 1
        return resultado
```

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Funciones Generadoras

- Cuando se llama a la función generadora, ésta no ejecuta el cuerpo de la función inmediatamente. En su lugar, devuelve un objeto generador sobre el que se puede iterar para producir los valores.

```
def my_generator(n):

    # initialize counter
     value = 0

    # loop until counter is less than n
     while value < n:

          # produce the current value of the counter
          yield value

        # increment the counter
          value += 1
```

```
# iterate over the generator object
# produced by my_generator
for value in my_generator(3):
     # print each value produced by generator
     print(value)
```

Ver ejemplo_7.py y 7a

Funciones Generadoras

Representar un flujo infinito:

- Los generadores son medios excelentes para representar un flujo infinito de datos.
- Los flujos infinitos no pueden almacenarse en memoria, y como los generadores producen sólo un elemento cada vez, pueden representar un flujo infinito de datos.

Ejemplo:

```
def todos_pares():
    n = 0
    while True
        yield n
        n += 2
```

Funciones Generadoras

Encadenar generadores:

- Se pueden utilizar varios generadores para canalizar una serie de operaciones. Esto se ilustra mejor con un ejemplo.
- Supongamos que tenemos un generador que produce los números de la serie de Fibonacci. Y tenemos otro generador para elevar números al cuadrado.
- Si queremos averiguar la suma de los cuadrados de los números de la serie de Fibonacci, podemos hacerlo canalizando la salida de las funciones del generador juntas.

Funciones Generadoras

Encadenar generadores:

```
def numeros_fibonacci(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def cuadrado(nums):
    for num in nums:
        yield num**2

print(sum(cuadrado(numeros_fibonacci(10))))
```

Ver ejemplo_8.py

Funciones Generadoras

- Una **expresión generadora** es una forma concisa de crear un objeto generador
- Es similar a una comprensión de lista, pero en lugar de crear una lista, crea un objeto generador sobre el que se puede iterar para producir los valores del generador.
- Una expresión generadora tiene la siguiente sintaxis:

```
(expression for elemento in iterable)
```

Ejemplo:

```
# crear el objeto generador
generador_cuadrados = (i * i for i in range(5))

# iterar sobre el generador e imprimir los valores
for i in generador_cuadrados:
    print(i)
```

Ver ejemplo_9.py

Funciones Generadoras

- **CASO de USO:**
  - Podemos obtener un objeto session en demanda entregando (yield) el objeto session cuando sea requerido (ejm: la sesión de acceso a una base de datos).

F I N