Curso Avanzado de Python

API REST, Decoradores,
Generadores,

Decoradores en python

Si has trabajado en Python, es posible que los hayas visto...

```
# Builtin
@property, @classmethod
# Django, Flask
@login_required, @app.route
```

```
@login_required
def profile_view(request):
    # Sólo los usuarios conectados pueden acceder aquí
    ...
```

¿Pero que son?

Los decoradores son clases o funciones que envuelven a otras clases o funciones (valga de redundancia). Una función que toma como input otra función y a su vez retorna otra función. Estaremos trabajando con 3 minimo funciones: el input, el output y la función principal.

¿Y eso para que?

Los decoradores le permite, sobre sus funciones y clases:

- Sustituir o cambiar su comportamiento.
- Cambiar los argumentos y parámetros de entrada, o su salida.
- Registrarlos.

```python
#estructura
def funciona_a(funcion_b):

    def funcion_c():
        pass

    return funcion_c
```

```python
#estructura
def funciona_a(funcion_b):

    def funcion_c():
        #ejecutar otras cosas
        funcion_b()
        #ejecutar otras cosas

    return funcion_c
```

Cómo usarlos

Mediante el símbolo arroba (@), seguido del decorador, encima de la clase o función

```
@app.route("/")
def hello():
    return "Hello World!"
```

Vamos a crearlos en Replit

¿Qué pasa si nuestra función tiene argumentos?

Ejercicio 1

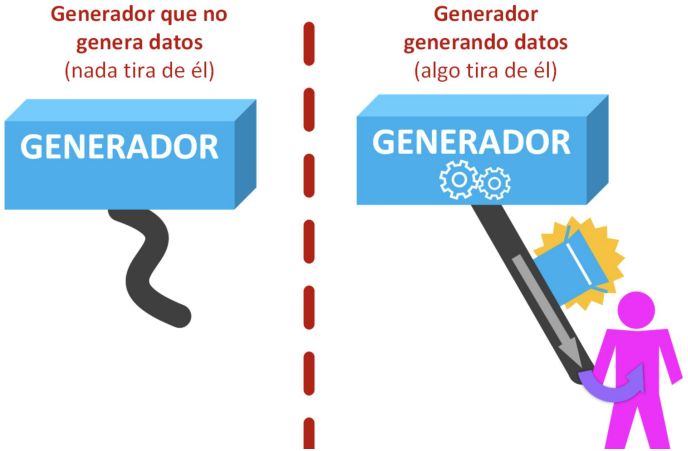
Crear con un decorador una función que imprima un mensaje si el divisor es 0, que no se puede dividir entre 0.

Ejercicio 2 - Decorador

Cree una función de autenticación para un usuario, que muestre el mensaje Error. El usuario no se ha autenticado, para cualquier función con parámetros y sin parámetros.

La Base del Streaming

El Generador de datos permitirá un flujo de datos (o Streaming) en tiempo real y tener cada uno de los datos solo cuando necesite ser consumido (procesados o utilizados por una persona). Adicionalmente, será muy importante para garantizar un trabajo óptimo y equilibrio entre el procesado de datos (procesador) y la memoria principal (la memoria RAM) de cualquier máquina (ordenador de sobremesa, portátil, Tablet, etc.).



Es decir, **los datos en un Generador no existen hasta que no son solicitados por un consumidor**.

Generadores en Python

Si alguna vez te has encontrado con una función en Python que no sólo tiene una sentencia `return`, sino que además devuelve un valor haciendo uso de `yield`, ya has visto lo que es un generador o *generator*. A continuación te explicamos cómo se crean, para qué sirven y sus ventajas. Es muy importante también no confundir los generadores con las corrutinas, que también usan `yield` pero de otra manera

Empecemos por lo básico. Seguramente ya sepas lo que es una función y cómo puede devolver un valor con `return`. Como hemos explicado, los generadores usan `yield` en vez de `return`. Vamos a ver que pasaría si cambiamos el return por el yield.

```
def funcion():
    return 5
```

```
def generador():
    yield 5
```

Generadores en Python

Si alguna vez te has encontrado con una función en Python que no sólo tiene una sentencia `return`, sino que además devuelve un valor haciendo uso de `yield`, ya has visto lo que es un generador o *generator*. A continuación te explicamos cómo se crean, para qué sirven y sus ventajas. Es muy importante también no confundir los generadores con las corrutinas, que también usan `yield` pero de otra manera

Empecemos por lo básico. Seguramente ya sepas lo que es una función y cómo puede devolver un valor con `return`. Como hemos explicado, los generadores usan `yield` en vez de `return`. Vamos a ver que pasaría si cambiamos el return por el yield.

```
def funcion():
    return 5
```

```
def generador():
    yield 5
```

Generadores en Python

```
def generador():
    n = 1
    yield n

    n += 1
    yield n

    n += 1
    yield n
```

```
g = generador()
print(next(g))
print(next(g))
print(next(g))
# Salida: 1, 2, 3
```

```
for i in generador():
    print(i)
# Salida: 1, 2, 3
```

Forma alternativa

```
lista = [2, 4, 6, 8, 10]
al_cuadrado = [x**2 for x in lista]
print(al_cuadrado)
# [4, 16, 36, 64, 100]
```

```
al_cuadrado_generador = (x**2 for x in lista)
print(al_cuadrado_generador)
# Salida: <generator object <genexpr> at 0x103e803b8>
```

```
for i in al_cuadrado_generador:
    print(i)
# Salda: 4, 16, 36, 64, 100
```

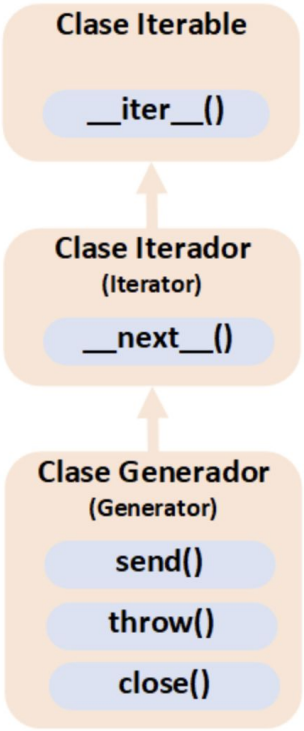
Hagamos este ejemplo en Codigo

Ahora vamos a hacerlo sin una funcion generadora

List Comprehensions vs Generators

La diferencia entre el ejemplo usando *list comprehensions* y *generators* es que en el caso de los generadores, los valores no están almacenados en memoria, sino que se van generando al vuelo. Esta es una de las principales ventajas de los generadores, ya que los elementos sólo son generados cuando se piden, lo que hace que sean mucho más eficientes en lo relativo a la memoria.

El objeto Generator es un iterador



Por tanto, **para que el "Generador" produzca datos tenemos que utilizarlo como cualquier otro Iterador con un bucle "for"** (porque puede como Iterable puede utilizar "iter()") para iterar por todos sus elementos **o utilizar "next"** (porque puede como Iterador puede utilizar "next()") para ir obteniendo elemento a elemento ("next" a "next")**.**

Ejemplo Clase

Llegados a este punto tal vez te preguntes para qué sirven los generadores. Lo cierto es que aunque no existieran, podría realizarse lo mismo creando una clase que implemente los métodos `__iter__()` y `__next__()`. Veamos un ejemplo de una clase que genera los primeros 10 números (0,9) al cuadrado.

Suma Números Naturales

queremos sumar los primeros 100 números naturales (referencia). Una opción podría ser crear una lista de todos ellos y después sumarla. En este caso, todos los valores son almacenados en memoria, algo que podría ser un problema si por ejemplo intentamos sumar los primeros 1e10 números.