Baldomero Sánchez

1. Funciones

Unidad 5

funciones

Las funciones lambda o anónimas son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño.

```
def suma(a, b):
        return a+b
```

las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una función.

```
(lambda a, b: a + b)(2, 4)
```

lambda no tiene un nombre, y por lo tanto salvo que se asignada a una variable

Funciones lambda

Se pueden pasar los parámetros indicando su nombre.

```
suma = lambda a, b: a + b
```

```
(lambda a, b: a + b)(2, 4)
```

Se puede tener un número variable de argumentos haciendo uso de *, lo conocido como tuple unpacking

```
(lambda a, b, c: a + b + c)(a=1, b=2, c=3) # 6
```

Si tenemos los parámetros de entrada almacenados en forma de key y value como si fuera un diccionario, también es posible llamar a la función.

```
(lambda *args: sum(args))(1, 2, 3) # 6
```

```
(lambda **kwargs: sum(kwargs.values()))(a=1, b=2, c=3) # 6
```

Funciones lambda

Es posible devolver más de un valor

```
x = lambda a, b: (b, a) print(x(3, 9)) #
Salida (9,3)
```

funciones

Factorial de un número

```
def factorial_recursivo(n):
        if n == 1:
                return 1
        else: return n * factorial_recursivo(n-1)

factorial_recursivo(5) # 120
```

Serie de Fibonacci

calcular el elemento 7, que será 0,1,1,2,3,5,8,13,

```
def fibonacci_recursivo(n):
        if n == 0:
                return 0
        elif n == 1:
                return 1
        else:
            return fibonacci_recursivo(n-1) + fibonacci_recursivo(n-2)
```

Funciones- Definición de Decoradores

Todo en Python es un objeto, incluso una función.
De hecho se puede asignar una función a una variable.

```
Hola
None
<function di_hola at
0x0000022DBEA23E20>
Hola
Hola
```

```
def di_hola():
    print("Hola")

f1 = di_hola() # Llama a la función
f2 = di_hola   # Asigna a f2 la función

print(f1)      # None, di_hola no devuelve
nada
print(f2)      # <function di_hola at
0x1077bf158>

#f1()          # Error! No es válido
f2()           # Llama a f2, que es
di_hola()

del f2         # Borra el objeto que es la
función
#f2()          # Error! Ya no existe

di_hola()      # Ok. Sigue existiendo
```

Funciones- Decoradores con parámetros

Tu decorador tenga algún parámetro de entrada para modificar su comportamiento.
Se puede hacer envolviendo una vez más la función como se muestra a continuación.

```
Imprimer esto antes y después
Entra en funcion suma
Imprimer esto antes y después
86
```

```
def mi_decorador(arg):
    def decorador_real(funcion):
        def nueva_funcion(a, b):
            print(arg)
            c = funcion(a, b)
            print(arg)
            return c
        return nueva_funcion
    return decorador_real

@mi_decorador("Imprimer esto antes y
después")
def suma(a, b):
    print("Entra en funcion suma")
    return a + b

print(suma(8,78))
```

Funciones- Decoradores

El decorador puede ser usado sobre funciones que tienen diferente número de parámetros de entrada, y su funcionalidad será siempre la misma.

Funciones- Definición de Decoradores

En Python se pueden definir funciones dentro de otras funciones.
La función operaciones define suma() y resta(), y dependiendo del parámetro de entrada op, se devolverá u otra

```
def operaciones(op):
    def suma(a, b):
        return a + b
    def resta(a, b):
        return a - b

    if op == "suma":
        return suma
    elif op == "resta":
        return resta

funcion_suma = operaciones("suma")
print(funcion_suma(5, 7)) # 12

funcion_suma = operaciones("resta")
print(funcion_suma(5, 7)) # -2
```

Funciones- Decoradores con parámetros

Tu decorador tenga algún parámetro de entrada para modificar su comportamiento.
Se puede hacer envolviendo una vez más la función como se muestra a continuación.

```
Imprimer esto antes y después
Entra en funcion suma
Imprimer esto antes y después
86
```

```
def mi_decorador(arg):
    def decorador_real(funcion):
        def nueva_funcion(a, b):
            print(arg)
            c = funcion(a, b)
            print(arg)
            return c
        return nueva_funcion
    return decorador_real

@mi_decorador("Imprimer esto antes y
después")
def suma(a, b):
    print("Entra en funcion suma")
    return a + b

print(suma(8,78))
```

Funciones- Ejemplo tipo loggers

Una de las utilidades más usadas de los decoradores son los logger.
Su uso nos permite escribir en un fichero los resultados de ciertas operaciones, que funciones han sido llamadas, o cualquier información que en un futuro resulte útil para ver que ha pasado.

```
def log(fichero_log):
    def decorador_log(func):
        def decorador_funcion(*args, **kwargs):
            with open(fichero_log, 'a') as opened_file:
                output = func(*args, **kwargs)
                opened_file.write(f"{output}\n")
        return decorador_funcion
    return decorador_log

@log('ficherosalida.log')
def suma(a, b):
    return a + b

@log('ficherosalida.log')
def resta(a, b):
    return a - b

@log('ficherosalida.log')
def multiplicadivide(a, b, c):
    return a*b/c

suma(10, 30)
resta(7, 23)
multiplicadivide(5, 10, 2)
```

ficherosalida.log

Funciones- Ejemplo uso Autorizado

Uso autorizado

Otro caso de uso muy importante y ampliamente usado en Flask, que es un framework de desarrollo web, es el uso de decoradores para asegurarse de que una función es llamada cuando el usuario se ha autenticado.

Realizando alguna simplificación, podríamos tener un decorador que requiriera que una variable autenticado fuera True. La función se ejecutará solo si dicha variable global es verdadera, y se dará un error de lo contrario.

Prueba a Cambiar la variable **autoenticado= false**

```
autenticado = True

def requiere_autenticación(f):
    def funcion_decorada(*args, **kwargs):
        if not autenticado:
            print("Error. El usuario no se ha autenticado")
        else:
            return f(*args, **kwargs)
    return funcion_decorada

@requiere_autenticación
def di_Tu_nombre():
    print("Usuario")

di_Tu_nombre()

multiplicadivide(5, 10, 2)
```

Funciones- Generators

En Python existen funciones que no sólo tiene una sentencia return, sino que además devuelve un valor haciendo uso de yield.

Es muy importante también no confundir los generadores con las corrutinas, que también usan yield

La diferencia entre función y generador

La llamada a la función devuelve 5 pero la llamada al generador() devuelve un objeto de la clase generador.

CONCLUSIÓN: si la función contiene al menos una sentencia yield la función es un generador

```
def funcion():
        return 45
```

```
def generador():
        yield  45
```

```
print(funcion())
print(generador())
# Salida: 45
# Salida: <generator object generador at 0x103e7f0a0>
```

Funciones- Iterando los Generadores

Otra de las características que hacen a los generators diferentes, es que pueden ser iterados, ya que codifican los métodos __iter__() y __next__(), por lo que podemos usar next() sobre ellos. Dado que son iterables, lanzan también un StopIteration cuando se ha llegado al final.

Que pasa ahora si intentamos llamar otra vez a next(). Si recuerdas, sólo tenemos una llamada a yield.
La salida producida es una excepción del tipo **StopIteration**, ya que el generador no devuelve más valores. Esto se debe a que cada vez que usamos next() sobre el generador, se llama y se continúa su ejecución después del último yield. Y en este caso cómo no hay más código, no se generan más valores.

```
def generador():
        yield  45
```

```
a = generador()
print(next(a))
# Salida:4 5
```

```
a = generador()
print(next(a))
print(next(a))
# Salida: 5
# Salida: Error! StopIteration:
```

```
def generador():
        yield  45
        yield  5
        yield  34
```

Funciones- Creando Generadores

Cada línea yield devuelve la operación realizada previamente, se encuentra entrelazada por next()
Se puede iterar utilizando bucle

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

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

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

Funciones- Otra forma alternativa

Los generadores también pueden ser creados de una forma mucho más sencilla y en una sola línea de código. Su sintaxis es similar a las list comprehension, pero cambiando el corchete [] por paréntesis ().

list compregensions 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

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

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

```
# Otra forma de un generador
 for i in al_cuadrado_generador:
                print(i)
 # Salda: 4, 16, 36, 64, 100
```

Funciones- Para que 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.

```
class AlCuadrado:
    def __init__(self):
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i > 9:
            raise StopIteration

        result = self.i ** 2
        self.i += 1
        return result


eleva_al_cuadrado = AlCuadrado()
for i in eleva_al_cuadrado:
    print(i)
#0,1,4,9,16,25,36,49,64,81
```

Funciones- Inicializar una clase __init__

El método __init__ es un método especial de una clase en Python. El objetivo fundamental del método __init__ es inicializar los atributos del objeto que creamos.
Las ventajas de implementar el método __init__ en lugar del método inicializar son:
1. El método __init__ es el primer método que se ejecuta cuando se crea un objeto.
2. El método __init__ se llama automáticamente. Es decir es imposible de olvidarse de llamarlo ya que se llamará automáticamente.

Otras características del método __init__ son:

- Se ejecuta inmediatamente luego de crear un objeto.
- El método __init__ no puede retornar dato.
- El método __init__ puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- El método __init__ es un método opcional, de todos modos es muy común declararlo.

Funciones- Inicializar una clase __init__

- **Confeccionar una clase que represente un empleado.**
- **Definir como atributos su nombre y su sueldo.**
- **En el método __init__ cargar los atributos por teclado y luego en otro método imprimir sus datos y por último uno que imprima un mensaje si debe pagar impuestos (si el sueldo supera a 3000)**

Funciones- Inicializar una clase __init__

```
class Empleado:

    def __init__(self):
        self.nombre=input("Ingrese el nombre del empleado:")
        self.sueldo=float(input("Ingrese el sueldo:"))

    def imprimir(self):
        print("Nombre:",self.nombre)
        print("Sueldo:",self.sueldo)

    def paga_impuestos(self):
        if self.sueldo>3000:
            print("Debe pagar impuestos")
        else:
            print("No paga impuestos")

# bloque principal

empleado1=Empleado()
empleado1.imprimir()
empleado1.paga_impuestos()
```

Funciones- Inicializar una clase __init__

```
class Empleado:

    def __init__(self):
        self.nombre=input("Ingrese el nombre del empleado:")
        self.sueldo=float(input("Ingrese el sueldo:"))

    def imprimir(self):
        print("Nombre:",self.nombre)
        print("Sueldo:",self.sueldo)

    def paga_impuestos(self):
        if self.sueldo>3000:
            print("Debe pagar impuestos")
        else:
            print("No paga impuestos")

# bloque principal

empleado1=Empleado()
empleado1.imprimir()
empleado1.paga_impuestos()
```

Funciones- iteradores en Python

Los iteradores están en todas partes en Python.
Están elegantemente implementados dentro de for bucles, comprensiones, generadores, etc. pero están ocultos a la vista.
Iterator en Python es simplemente un objeto sobre el que se puede iterar.
Un objeto que devolverá datos, un elemento a la vez que itere.
Técnicamente hablando, un **objeto iterador de** Python debe implementar dos métodos especiales __iter__() y __next__(), colectivamente, llamado **protocolo iterador**.
Un objeto se llama **iterable** si podemos obtener un iterador de él.
La mayoría de los contenedores integrados en Python como: lista , tupla , cadena , etc. son iterables.
La iter() función (que a su vez llama al __iter__() método) devuelve un iterador de ellos.

Funciones- Para que sirven los generadores (2)

Si 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.

```
def primerosn(n):
    nums = []
    for i in range(n):
        yield num
        num += 1
    return nums

print(sum(primerosn(100)))

# Salida: 4950
```

Funciones- Para que sirven los generadores (3)

Los generadores hace que no todos los valores estén almacenados en memoria sino que sean generados al vuelo

```
def primerosn(n):
    nums = []
    for i in range(n):
        nums.append(i)
    return nums

print(sum(primerosn(100)))

# Salida: 4950
```

La forma más sencilla de iterar es utilizando un rango de números y establecer la suma

```
print(sum(range(100)))
# Salida: 4950
```

Bibliografía y recursos online

- https://www.mclibre.org/consultar/python/index.html
- https://docs.python.org/es/3/reference
- https://www.python.org/dev/peps/pep-0257/
- https://docs.python.org/3/library/functions.html