1 Seminario 14- Python I

1.1 Decoradores

En Python, las funciones son *first class citizen*, lo que significa que el lenguaje permite pasar funciones como parámetros, retornarlas como valor de otras de funciones y asignarlas a variables o almacenarlas en cualquier estructura de datos.

> En programación funcional, se trabaja (casi) solamente con funciones. A pesar, de que Python no es un lenguaje puramente funcional, si soporta muchas de las características de lenguajes de este tipo, incluyendo las funciones como *first-class* objects

Los **decoradores** en Python proveen una sintaxis simple para llamar a funciones de orden superior. Una función de orden superior es una función que hace al menos una de las siguientes operaciones:

- Toma como argumento una o más funciones.
- Retorna una función como resultado.

Por tanto, por definición podemos decir que un **decorador** es una función que recibe como parámetro otra función y extiende el comportamiento de esta última sin modificarla explícitamente.

En otras palabras, los decoradores hacen posible que una función o clase A tome como argumento otra función B con el objetivo de devolver una tercera función C.

1.1.1 Sintaxis de los decoradores:

Empecemos con un ejemplo sencillo de lo que es un **decorador**:

```
def my_decorator(func):
    def wrapper():
        print('Antes de llamar a la función')
        func()
        print('Después de llamar a la función')
    return wrapper

def say_hello():
    print('Hola')
```

```
salut = my_decorator(say_hello)
salut()
```

Al ejecutar las líneas anteriores se imprimirá:

> Antes de llamar a la función
>
> Hola
>
> Después de llamar a la función

¿Qué es lo que ha sucedido? En efecto, se ha llamado al decorador `my_decorator` con la función `say_hello` como argumento y esta ha retornado a `wrapper` como nueva función.

Python brinda una forma más cómoda de usar un decorador en una función a través del símbolo @. El siguiente ejemplo hace exactamente lo mismo que el anterior:

```
def my_decorator(func):
    def wrapper():
        print('Antes de llamar a la función')
        func()
        print('Después de llamar a la función')
    return wrapper


@my_decorator
def say_hello():
    print('Hola')


say_hello()
```

1.1.2 Decorando funciones con argumentos

Supongamos que tenemos una función que recibe argumentos, ¿cómo se podría decorar? Veamos a continuación un ejemplo:

```
def run_twice(f):
    def wrapper():
        f()
        f()
    return wrapper

@run_twice
```

```
def greet(a):
    print(f'Hola {a}')

greet('Mundo')
```

Evidentemente lo anterior dará error. El problema es que la función `wrapper` no recibe ningún argumento, pero `a='Mundo'` fue pasado como parámetro. ¿Cómo podemos resolver este problema si no sabemos la cantidad de argumentos que recibe la función que se va decorar? La solución es usar `*args` y `**kargs`. Esto permitirá aceptar un número arbitrario de parámetros. Luego el decorador quedaría así:

```
def run_twice(f):
    def wrapper(*args,**kwargs):
        f(*args,**kwargs)
        f(*args,**kwargs)
    return wrapper
```

1.1.3 Retornando valores de funciones decoradas

Hasta ahora los ejemplos vistos son funciones que no retornan valores, ¿qué sucede si la función que se quiere decorar retorna algún valor? Veamos el siguiente ejemplo:

```
@run_twice
def return_greet(a):
    return f'Hola {a}'
```

Si se llama a la función `return_greet` con `a='Mundo'` el resultado sería `None`. El problema se aprecia fácilmente: la función wrapper no retorna nigun valor. Para arreglar esto se tiene que asegurar que la función wrapper devuelva el mismo valor de retorno que la función decorada. Esto se logra de la siguiente manera:

```
def run_twice(f):
    def wrapper(*args,**kwargs):
        return f(*args,**kwargs)
    return wrapper
```

1.1.4 Decorando clases

Hay dos formas de usar decoradores en clases. La primera es muy parecida a lo visto hasta ahora en funciones: se pueden decorar los métodos de una clase. Algunos de los decoradores *built-in* más usados en Python son `@classmethod`, `@staticmethod` y `@property`.

1.1.5 Ejemplos del uso de los decoradores de clases *built-in*

La siguiente definición de la clase Circle usa los decoradores mencionados anteriormente:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

    @property
    def area(self):
       return self.pi() * self.radius ** 2

    def cylinder_volume(self, height):
        return self.area * height

    @classmethod
    def unit_circle(cls):
        return cls(1)

    @staticmethod
    def pi():
        return 3.1415926535
```

En esta clase:

- `cylinder_volume` es un método ordinario.
- `radius` es una propiedad mutable, puede ser modificada (se puede acceder a ella como si fuera un atributo, sin paréntesis)
- `area` es una propiedad inmutable, no se puede modificar su valor
- `unit_circle` es un *class method*. No está asociado una instancia particular de `Circle`.

- `pi` es un método estático.

La otra forma de usar decoradores en clases es decorar la clase entera. El significado de la sintaxis es similar al de decorar funciones.

Un uso común de los decoradores de clases es ser una alternativa más simple a algunos usos de `metaclases`. En ambos casos, se está cambiando la definición de una clase dinámicamente.

Las clases también pueden ser decoradores, pero esto se va del ámbito del seminario, exhortamos al lector documentarse sobre este tema.

1.1.6 Múltiples decoradores en una sola función

Es posible aplicar varios decoradores a una misma función como se muestra a continuación:

```
from Code.decorators import timer, run_twice

@timer
@run_twice
def greet(a):
    print(f'Hola {a}')
```

Esto se puede ver cómo `timer(run_twice(greet))`

1.1.7 Decoradores con argumentos

En ocasiones es útil pasarles argumentos a los decoradores. Por ejemplo `@run_twice` podría ser extendido a `@repeat(num_times)`.

Hasta ahora, el nombre después del @ se refiere a una función que puede ser llamada con otra función. Para ser consistente, se necesitaría que `repeat(num_time=4)` retornara un objeto de tipo función que pueda servir como decorador. De forma general se quiere algo como lo siguiente:

```
def repeat(num_times):
    def decorator_repeat(func):
        ...  # Create and return a wrapper function
    return decorator_repeat
```

El decorador crea y retorna una función `wrapper` interna, y si se completa el ejemplo dentro de esta función se creará otra función. Quedaría así:

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat
```

Parece un poco confuso, pero lo que se ha hecho es seguir el mismo patrón hasta ahora, solo que se ha añadido otro wrapper más externo que es el encargado de recibir los argumentos del decorador.

1.2 Python y el chequeo de tipos:

Python es un lenguaje dinámicamente tipado. Esto significa que el intérprete de Python hace el chequeo de tipos cuando se ejecuta el código, y que el tipo de una variable tiene permitido cambiar durante su tiempo de vida.

En un lenguaje de tipado estático, las variables llevan asociado el tipo de dato y no se puede cambiar. En los lenguajes de tipado dinámico, el tipo va más bien asociado al valor de la variable y no a la variable en sí misma, por lo que una misma variable puede contener a lo largo de la ejecución distintos tipos de datos. Con tipado estático, es el mismo compilador el que comprueba que se asignan valores correctos a las variables, ya que se sabe de qué tipo son las variables y que tipo de valores pueden admitir. Con tipado dinámico, esta comprobación debe hacerse en tiempo de ejecución. Podemos asignar a cualquier variable cualquier tipo de dato, por lo que hasta la ejecución no se sabe qué tipo de valor tiene una variable. La ventaja del tipado estático es que pueden evitar muchos errores en tiempo de compilación, sin necesidad de esperar a la ejecución para verlos. La ventaja del tipado dinámico es su flexibilidad, este a su vez hace más sencilla la escritura de meta-programas: programas que reciben como datos código y lo manipula para producir nuevo código.

Supongamos que queremos ciertas características de los lenguajes con tipado estático en Python, como por ejemplo exigir que la función `_add_int_` definida a continuación reciba como parámetros dos *int*:

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

Podemos usar *type annotations* como se muestra a continuación:

```
def add_int(a: int,b: int):
    return a + b
```

Si se está trabajando en algún IDE como PyCharm este dará una advertencia si alguno de los parámetros que se está pasando a la función no es del tipo esperado. Sin embargo, si `a = 'a'` y `b = 'b'`, al ejecutar el código donde se llama a la función con dichos parámetros se obtendría `'ab'`

Para resolver esto podemos usar el siguiente decorador:

```
def typeCheck(*types):
    def wrapper(f):
        assert len(types) == f.__code__.co_argcount

        def new_f(*args):
            for (a, t) in zip(args, types):
               assert type(a) == t, f'{a} is not a valid type'
            return f(*args)

        return new_f

    return wrapper
```

El decorador `typeCheck` recibe como parámetro los tipos de los parámetros de la función que va a decorar y se encarga de hacer el chequeo de tipos. Quedaría así:

```
@typeCheck(int,int)
def add_int(a,b):
    return a + b
```

Incluso, podríamos extender el decorador anterior para no solo permitir que un parámetro sea de un tipo específico, sino de un conjunto.

```
def typeCheckExtended(*types):
    def wrapper(f):
        assert len(types) == f.__code__.co_argcount

        def new_f(*args):
            for (a, t) in zip(args, types):
                if type(t) is tuple:
                    assert any([type(a) == typex for typex in t]),
                        f'{a} is not a valid type'
                else:
                    assert type(a) == t, f'{a} is not a valid type'
```

```
            return f(*args)

        return new_f

    return wrapper
```

1.3 ¿Tiene utilidad hacer chequeo de tipos en un lenguaje dinámico?

Hacer chequeo de tipos en un lenguaje dinámico tiene como objetivo primordial evitar errores en tiempo de ejecución que el compilador pudiera obviar. Un ejemplo clásico es: sea `add(x,y)` una función que dados dos enteros devuelve la suma de estos, en caso de que llamemos `add(x,y)` donde `x` y `y` son objetos para los cuales la operación "suma" está definida, obtendríamos un resultado potencialmente erróneo, dado que tanto `x` como `y` podrían ser, por ejemplo, dos strings, y esto no se consideraría un error por parte del compilador, por ende, en tiempo de ejecución los errores serían bastante groseros.

El chequeo de tipos puede permitir al compilador traducir ciertas operaciones a instrucciones de bajo nivel cuando se está trabajando sobre objetos que permitan tal optimización. Otra utilidad, en este caso para lenguajes que como Python permiten anotaciones de tipo en los objetos, es que, para proyectos de gran envergadura, facilita enormemente la tarea de documentar el código, y agilizar así duras jornadas de debuggeo.

1.4 ¿Tendría sentido modificar el lenguaje para tener variables y parámetros con tipos?

Bueno la respuesta es que sí tendría sentido, de hecho, Python de cierta manera lo hace con las anotaciones de tipo. No es algo que le de tipos estáticamente pero igual ayuda, el intérprete de Python no asocia ningún significado particular a las anotaciones de variables y solo las almacena en un atributo especial. No tener preocupación por los tipos es algo muy cómodo, y le da mucha genericidad y extensibilidad a tu código, pero también puede ser algo peligroso, ya que es muy difícil saber cuál es el objeto esperado, o con qué objeto estamos trabajando. La solución en Python fueron las anotaciones de tipo. A diferencia de las declaraciones de variables en lenguajes de tipo estático, el objetivo de la sintaxis de anotación es proporcionar una manera fácil de especificar metadatos de tipo estructurado para bibliotecas y herramientas de terceros mediante el árbol de sintaxis abstracta y el atributo `__annotations__`.

```
from typing import List, Dict


primes: List[int] = []
word: str
MyDict: Dict[str, int] = {}
primes.append('a')
print (primes[0])
```

El atributo después de los dos puntos solo es para adjuntar sugerencias de variables, antes solo podía adjuntar sugerencias de tipo a variables con comentarios (por ejemplo, `primes = [] # List[int]`). Esto de los dos puntos es a partir de Python 3.6 y el compilador sí lo tiene en cuenta. Pero tanto `primes` como `Mydict` son `Generic types`, `primes` es una lista de tipo genérico y `Mydict` es un diccionario con llave y valor genérico.

También se introdujeron cambios adicionales junto con la nueva sintaxis; los módulos y las clases ahora tienen un atributo `__annotations__` en las que se adjunta el tipo de metadatos.

Ahora `__main__.__annotations__` contiene los tipos declarados:

```
from typing import List, get_type_hints
import __main__


primes: List[int] = []
word: str
print (get_type_hints(__main__))
# {'primes': typing.List[int], 'word': <class 'str'>}  esto es lo que imprime
```

Para las funciones es igual:

```
def pick(l: list, index: int) -> int:
    return l[index]
```

Aquí podemos ver que pick toma 2 parámetros, una lista `l` y un entero `index`. También debe devolver un entero. Estas son las sugerencias.

En varios códigos mostrados hemos utilizado un módulo llamado Typing, hablemos brevemente sobre esta caja negra:

Este módulo proporciona soporte en tiempo de ejecución para sugerencias de tipo. El soporte fundamentalmente consiste en los tipos `Any`, `Union`, `Tuple`, `Callable`, `TypeVar` y `Generic`. Se pueden hacer anotaciones de tipo en las variables, parámetros y valores de retorno de funciones de funciones, estas están disponibles en tiempo de ejecución a través

del atributo `__annotations__`. Hay una serie de tipos que vienen ya con el módulo para hacer chequeos como 'Dict', 'DefaultDict', 'List', 'Set', 'Iterable'. También se puede usar para clases y chequear si una clase es subclase de otra. Veamos el siguiente ejemplo:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

De esta forma bastante clara, queda reflejado los tipos que esta función espera y devuelve, resaltar además que cualquier subtipo es aceptado también como argumento.

Otra de las herramientas que ofrece *Typing* son los `Type alias`, los cuales son útiles para simplificar la sintaxis del programa y hacerlo más legible.

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Por supuesto, Typing aporta decoradores, algunos de los cuales cabe resaltar. Primeramente está `@overload`, el cual permite describir métodos que soportan múltiples combinaciones de argumentos. Toda serie de decoradores `@overload` debe estar seguido por una función `non-@overload-decorated` para el método en cuestión, por ejemplo:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

Otro bastante interesante es `@final`. Este se usa para indicar que el método decorado no puede ser *overridden*, o que la clase decorada no puede ser *subclassed*. Mostremos un ejemplo para cada caso.

```
class Base:
```

```
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
          ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

1.5 ¿La definición del lenguaje obliga a la necesidad de variables y parámetros sin tipos?

En Python no se supone que el código se ocupe de datos específicos, si se hace, estará limitado a trabajar solo con los tipos que se anticiparon cuando se escribió, a pesar de eso hay herramientas que nos permiten comprobar tipos, pero eso le quita flexibilidad al código. Aunque si tenemos mucha flexibilidad en el código, entonces tenemos que probarlo para detectar errores, ya que el compilador no va a hacer mucho por nosotros. Entonces qué es lo que queremos, un código más resumido, compacto y flexible pero propenso a errores, y desconocimiento de los tipos de datos de los objetos, o un código estáticamente tipado, donde conocemos el tipo de dato de cada objeto, pero con muy poca flexibilidad y donde es muy probable que el código sea más extenso. De cierta manera siempre que no sea forzado a que las variables no tengan tipo, la definición del lenguaje puede permitirse tipado estático, pero esto no es lo intuitivo. En Python se quiere un código para interfaces de los objetos, no para sus tipos de datos, estaríamos atando la libertad del tipado dinámico. El tipado estático no tiene por qué ser mejor que el dinámico ni viceversa ¿Qué prefieres tú? ¿Qué es mejor para lo que estás haciendo? Esa es una decisión del programador.