Universitat de Lleida
Escola Politècnica Superior

Jordi Virgili
jvirgili@diei.udl.cat

PROCESS MODELS (DEGREE IN COMPUTER SCIENCE)

BUENAS PRÁCTICAS EN PYTHON PT1.

ANOTACIONES:

Python (y django por ende) sigue una filosofía duck typing, ignorando los tipos y utilizando las funcionalidades.

De hecho, este mismo concepto, en el que los tipos son ignorados "de facto" y utilizados o casteados en función de la necesidad, es el que nos permie tanta plasticidad. Por eso en Python podemos multiplicar un char (3*'3'='333') Pero pongamos por ejemplo la función built-in de Python para ver la longitud de un array: len(). En realidad lo que hace es llamar la método __len__() de la clase del tipo del objeto, al ser un array nos devuelve el numero de elementos que contiene, pero en el caso de que estemos creando nosotros una clase:

```
class Foo():
    def __len__(self):
        return "a saber"
```

nos daría de resultado:

```
obj = Foo()
print(len(obj))# "a saber"
```

Las anotaciones en Python permiten especificar el tipo de los argumentos de entrada y de salida de una función. ¡Pero cuidado! El intérprete de Python va a ignorar estas anotaciones, se deben interpretar como un comentario para facilitar el mantenimiento y el trabajo en equipo. Saber los tipos de entrada y salida ayudarán a entender mejor que era lo que se pretendía con ese código. Un ejemplo muy sencillo, una función que suma dos caracteres:

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

En este caso no sabemos el tipo que espera, y claramente esta función debe de formar parte de un programa mayor. En el caso de que estemos utilizando esta función como parte de una incorporación nuestra, puede afectar que, por ejemplo, ¿haya coma flotante en el resultado? Para estos casos se puede (y debe según las últimas convenciones de Python) notar de la siguiente manera:

```
def suma(a: int, b: int) -> int:
    return a + b
```

Se puede acceder desde el código a las anotaciones mediante __annotations__:

```
print(suma.__annotations__)
# {'a': 'int',
#  'b': 'int',
#  'return': 'int'}
```

Aunque el ejemplo sea un poco naive, obviamente se pueden complicar, lo que resulta útil si no queremos leernos todo el código, en la primera línea podemos ver los tipos de entrada y la clase que retorna la salida.

```
def funcion(a: ClaseA, b: ClaseB, f: float, s:str) -> ClaseZ:
        [… Lógica extensa …]
        return resultado
```

Habitualmente, y si se tiene tiempo, lo que se suele hacer es una comprobación de tipos al inicio de la función:

```
def suma(a, b):

    if !isinstance(a, int) or !isinstance(b, int):
        raise Exception("Type error")
    else:
        return a + b
```

Con las anotaciones se puede a ver algo más dinámico:

```
def suma(a, b):

    if !isinstance(a, suma.__annotations__['a'])
    or !isinstance(b, suma.__annotations__['b']):
        raise Exception("Error de tipos, expected " +
        str(suma.__annotations__['a']))
    else:
        return a + b
```

Como esto no se comprueba, los errores se producen en tiempo de ejecución como si no hubiésemos especificado nada. Para eso se utilizan herramientas como “mypy” (paquete de pip para testing de tipos) que realiza un chequeo estático de tipos.

Si testeamos el código:

```
# suma.py
def suma(a: int, b: int) -> int:
    return a + b

print(suma(7.0, "3"))
```

El resultado es:

```
$ mypy suma_incorrecta.py
suma_incorrecta.py:5: error: Argument 1 to "suma" has incompatible
type "float"; expected "int"
suma_incorrecta.py:5: error: Argument 2 to "suma" has incompatible
type "str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

Es una filosofía que se usa poco, pero que puede ayudar (y mucho) cuando de trabaja con equipos sobre una misma funcionalidad.