Pythonic Style

Conociendo un lenguaje de programación, siempre es posible escribir rápidamente programas en otro lenguaje. Esto consiste básicamente en seguir pensando con las construcciones de nuestro lenguaje primero y transcribirlas en el segundo, casi literalmente. Esta forma de trabajar nos impone una limitación de entrada, porque estamos renunciando a las características y posibilidades expresivas que nos ofrece el lenguaje nuevo.

En el caso de Python, hay un buen número de facilidades que permiten escribir un código limpio, claro y eficiente.

En este pequeño documento, damos una pequeña selección de dichas facilidades, junto con alguna recomendación. Algunas de ellas son adecuadas para otros lenguajes, otras, no.

Ojalá te resulte útil.

Condicionales

La recomendación aquí es recordar el uso de la expresión condicional:

```
expr1 if condition else expr2
```

In [1]:

```python
# Manera clásica:

def f1(n):
    if n >= 0:
        absolute = n
    else:
        absolute = -n
    return absolute**3

# Mejor:

def f2(n):
    absolute = n if n >= 0 else -n
    return absolute**3

# Mejor:

def f3(n):
    return (n if n >= 0 else -n)**3
```

In [2]:
```
# Prueba de funcionamiento:

for i in [2, -2, 10, -10]:
    print(i, " -> ", f1(i), f2(i), f3(i))
```

```
2  ->  8 8 8
-2  ->  8 8 8
10  ->  1000 1000 1000
-10  ->  1000 1000 1000
```

Recorridos de listas

Aquí la recomendación es prescindir del índice cuando sea posible.

In [3]:
```
# Viejo estilo:

def suma_datos(lista):
    acum = 0
    n = len(lista)
    for i in range(n):
        acum = acum + lista[i]
    return acum

print(suma_datos([1, 2, 3, 4, 5]))

# Estilo pitónico:

def suma_datos(lista):
    acum = 0
    for x in lista:
        acum = acum + x
    return acum

print(suma_datos([1, 2, 3, 4, 5]))

# Más claro aún:

def suma_datos(lista):
    return sum(lista)

print(suma_datos([1, 2, 3, 4, 5]))
```

```
15
15
15
```

Más recorridos de listas, conjuntos y diccionarios: mejor con notación intensional

La notación intensional permite expresar con gran claridad situaciones frecuentes. He aquí unas pocas de dichas situaciones con listas, conjuntos y diccionarios.

- Con listas:

In [4]:
```
# Viejo estilo:

def lista_de_cuadrados_1(n):
    lista = []
    for i in range(n):
        lista.append(i**2)
    return lista

# Estilo pitónico:

def lista_de_cuadrados_2(n):
    return [i**2 for i in range(n)]

# Prueba de funcionamiento:

print(lista_de_cuadrados_1(10))
print(lista_de_cuadrados_2(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Con conjuntos. Completa tú un diseño más pitónico y añade una prueba de funcionamiento:

In [5]:
```
# Viejo estilo:

def conj_de_cuadrados_1(n):
    conj = set()
    for i in range(n):
        conj.add(i**2)
    return conj

# Estilo pitónico:

# ................ ¿Podrás diseñar tú esta solución?

# Prueba de funcionamiento:

print(conj_de_cuadrados_1(10))
# print(conj_de_cuadrados_2(10))
```

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

- Con diccionarios. Completa tú un diseño más pitónico y añade una prueba de funcionamiento:

In [6]:
```
# Viejo estilo:

def dicc_de_cuadrados_1(n):
    diccionario = dict()
    for i in range(n):
        diccionario[i] = i**2
    return diccionario

# Estilo pitónico:

# ..... ¿Podrás diseñar tú esta solución?

# Prueba de funcionamiento:

print(dicc_de_cuadrados_1(10))
# print(dicc_de_cuadrados_2(10))
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Tuplas y listas

- Asignación paralela

In [7]:
```
# Intercambio de valores entre dos variables:

# Viejo estilo:

a = 1
b = 666
print("Iniciales  : ", a, b)

aux = a
a = b
b = aux

print("Intercambio: ", a, b)

# Mejor:

a, b = b, a

print("Intercambio: ", a, b)
```

```
Iniciales  :  1 666
Intercambio:  666 1
Intercambio:  1 666
```

- Recorridos de listas de tuplas

In [8]:
```
# Viejo estilo:

from math import sqrt as raiz

def distancia_maxima_1(puntos):
    d_max = 0.0
    for punto in puntos:
        x = punto[0]
        y = punto[1]
        dist = raiz(x**2 + y**2)
        if dist > d_max:
            d_max = dist
    return d_max

# Estilo pitónico:

def distancia_maxima_2(puntos):
    d_max = 0.0
    for (x, y) in puntos:
        dist = raiz(x**2 + y**2)
        if dist > d_max:
            d_max = dist
    return d_max

lista = [(3, 4), (8, 6), (12, 5)]
print(distancia_maxima_1(lista))
print(distancia_maxima_2(lista))
```

```
13.0
13.0
```

In [9]:
```
# Mejor aún:

def distancia_maxima_3(puntos):
    distancias = [raiz(x**2 + y**2) for (x, y) in puntos]
    return max(distancias)

print(distancia_maxima_3(lista))
```

```
13.0
```

Formación de listas

Generar la lista de los primeros `n` cuadrados.

In [10]:

```
# Viejo estilo:

def generar_cuadrados(n):
    lista = []
    for i in range(n):
        lista.append(i**2)
    return lista

print(generar_cuadrados(10))

# Con f.o.s. (abusando un poco de ellas):

def generar_cuadrados(n):
    return list(map(lambda i: i**2, range(n)))

print(generar_cuadrados(10))

# Estilo pitónico, mejor:

def generar_cuadrados(n):
    return [i**2 for i in range(n)]

print(generar_cuadrados(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [11]:

```
# Ejercicio: mejorar el siguiente código.

def min_max(a, b):
    if a <= b:
        par = (a, b)
    else:
        par = (b, a)
    return par

def sum_maxs(lista_de_pares):
    suma = 0
    for i in range(len(lista_de_pares)):
        par = lista_de_pares[i]
        a = par[0]
        b = par[1]
        peque_grande = min_max(a, b)
        grande = peque_grande[1]
        suma = suma + grande
    return suma

print(sum_maxs([(1, 2), (4, 3), (5, 6), (8, 7)]))
```

```
20
```

¿Varios `return` en una función?

- Sí: no pasa nada. No es una mala práctica. La regla del *único punto de salida* es una costumbre de la vieja escuela, válida en lenguajes de bajo nivel como C, donde la gestión de los recursos es manual.

Es un mantra procedente de una mala interpretación histórica sobre un consejo necesario en los orígenes de programación estructurada, pero no está vigente en lenguajes como Java, Ruby o Python.

- La tendencia actual, y más en lenguajes como Python, es admitir varios returns cuando mejore la legibilidad.
- Sobre este asunto, he aquí una referencia o dos, entre muchas otras posibles.

https://codely.tv/blog/screencasts/varios-returns-una-funcion/ (https://codely.tv/blog/screencasts/varios-returns-una-funcion/)
https://www.anthonysteele.co.uk/TheSingleReturnLaw.html (https://www.anthonysteele.co.uk/TheSingleReturnLaw.html)

In [12]:
```
# Ejemplos:

def menor(a, b):
    if a <= b:
        return a
    else:
        return b

print(menor(2, 3), menor(7, 1), menor(5, 5))
```

```
2 1 5
```

In [13]:
```
# Ejemplos:

import math

def primo(n):
    # precondition: n >= 2
    raiz = int(math.sqrt(n))
    for d in range(2, raiz+1):
        if n%d == 0:
            return False
    return True

print([n for n in range(2, 100) if primo(n)])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 6
7, 71, 73, 79, 83, 89, 97]
```

Referencias

https://www.syntonize.com/codigo-limpio-en-python/ (https://www.syntonize.com/codigo-limpio-en-python/)

https://www.askpython.com/python/examples/pythonic-way-of-coding (https://www.askpython.com/python/examples/pythonic-way-of-coding)