

# Comentarios en Python

- PEP 8 y Flake 8

(Data Engineer)



# Introducción

Bienvenidos al módulo 8 - Comentarios en Python utilizando PEP 8 y Flake 8. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conocimientos generales acerca de PEP8 y Flake8.

Una vez finalizado este módulo estarás capacitado para las siguientes acciones:

- Comprender qué es PEP8 y por qué es necesario en nuestros proyectos.
- Conocer las principales normas de PEP8 mediante ejemplos y anti ejemplos.
- Instalar y utilizar el paquete Flake8, como herramienta de revisión automática para que nuestro código se adapte a PEP8.

# Tema 1. PEP 8

## Objetivos

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Comprender que es PEP8 y por que es necesario.
- Comprender y utilizar las principales normas que nos ofrece PEP8.

## Introducción

PEP8, es un documento que brinda pautas y mejores prácticas sobre cómo escribir código Python. Fue escrito en 2001 por Guido van Rossum, Barry Varsovia y Nick Coghlan. El enfoque principal de PEP8 es mejorar la legibilidad y la consistencia del código de Python.

Como vimos anteriormente, PEP significa Python Enhancement Proposal o Propuesta de mejora de Python, y hay varios de ellos. Un PEP es un documento que describe nuevas características propuestas para Python y documenta aspectos de Python, como diseño y estilo, para la comunidad.

A continuación aprenderemos las pautas establecidas en PEP 8, el cual está dirigido a programadores principiantes e intermedios.

## ¿Por qué necesitamos PEP 8?

PEP 8 existe para mejorar la legibilidad del código Python. Pero, ¿por qué es tan importante la legibilidad? ¿Por qué escribir código legible es uno de los principios rectores del lenguaje Python?

Como dijo Guido van Rossum, “El código se lee mucho más a menudo de lo que se escribe”. Podemos pasar unos minutos o un día entero escribiendo un código para procesar la autenticación del

usuario. Una vez que lo hemos escrito, nunca lo volveremos a escribir, pero definitivamente tendremos que leerlo de nuevo.

Muchas veces nos cuesta recordar nuestro código unos días o semanas después de haberlo escrito. Si seguimos PEP8, podremos estar seguros de haber nombrado bien las variables. Sabremos que hemos agregado suficientes espacios en blanco para que sea más fácil seguir los pasos lógicos en nuestro código, y también habremos comentado bien nuestro código. Todo esto resultará en un código mucho más legible y más fácil de leer.

Seguir PEP8 es particularmente importante al momento de desarrollar código en un entorno laboral. Escribir un código claro y legible muestra profesionalismo.

## Explorando PEP8

A continuación vamos a conocer algunas de las normas principales de PEP8, siguiendo la siguiente estructura.

- **Convenciones de nombres.**
  - Estilos de nombres.
  - Cómo elegir nombres.
- **Diseño del código.**
  - Tamaño máximo de línea.
  - Indentación.
  - Líneas en blanco verticales.
- **Espacios en blanco en expresiones y declaraciones.**
  - Situaciones molestas
  - Otras recomendaciones.
- **Imports**
- **Comentarios**

Para más detalles se puede consultar la documentación oficial en el siguiente [enlace](#).

## Convenciones de nombres

Cuando estamos escribiendo código tenemos que nombrar muchas cosas: variables, funciones, clases, paquetes, etc. Elegir nombres adecuados nos ahorrará tiempo y energía más adelante. Podremos averiguar, a partir del nombre, qué representa una determinada variable, función o clase. También evitar el uso de nombres inapropiados que podrían generar errores difíciles de depurar.

## Estilos de nombres

En la siguiente tabla encontraremos algunos de los estilos de nombre más comunes en Python.

Tipo	Convención	Ejemplo
Variable Función Método de una clase Módulo	Utilizar una palabra o palabras en minúsculas separadas con guiones bajos	var, mi_variable función, mi_función method, class method module, my_module
Constante	Utilizar una sola letra, palabra o palabras en mayúsculas separadas con guiones bajos.	CONSTANTE, MI_CONSTANTE MI_CONSTANTE_NUEVA
Clase	Cada palabra comienza con letra mayúscula. No separar con guiones bajos	Modelo, MiClase
Paquete	Utilizar una palabra o palabras en minúsculas. No separar con guiones bajos.	package, mi_package

Estas son algunas de las convenciones de nomenclatura comunes y ejemplos de cómo utilizarlas, pero para escribir código legible, aún debemos tener cuidado con la elección de letras y palabras. Además de elegir los estilos de nomenclatura correctos en nuestro código, también debemos elegir los nombres con cuidado. A continuación se presentan algunos consejos sobre cómo hacer esto de la manera más efectiva posible.

## Cómo elegir nombres

La mejor manera de elegir el nombre de nuestros objetos en Python es utilizar nombres que sean descriptivos.

Por ejemplo: Declaramos una función que recibe un nombre y apellido y lo imprime de atrás para adelante.

```
def r(x):
    y,z = x.split()
    print(z, y, sep=", ")

r('Luis Ocampos')
Ocampos, Luis
```

Si bien esta función no tiene problemas, utilizar r como nombre de la función o las letras xyz pueden resultar confusas tanto para nosotros mismos, como para otra persona que esté leyendo nuestro código. En ese sentido podemos reescribir la función de la siguiente manera.

```
def reverse_name(name):
    first_name,last_name = name.split()
    print(last_name, first_name, sep=", ")

reverse_name('Luis Ocampos')
Ocampos, Luis
```

Ahora podemos ver que el nombre de la función y las variables son descriptivas.

## Diseño del código

La forma en que diseñamos nuestro código tiene un papel muy importante en lo legible que es. A continuación aprenderemos algunas de las recomendaciones que se describen en PEP8.

## Tamaño máximo de línea

Las líneas deben limitarse a un máximo de 79 caracteres.

## Indentación

Utilizar siempre 4 espacios y nunca mezclar tabuladores y espacios.

```
#Tabulación
#-----

#Si: opción 1
foo = funcion_que_crea_bar(variable_1, variable2
                           variable_3)

# opción 2
foo = funcion_que_crea_bar(
    variable_1, variable2
    variable_3)

#No:
foo = funcion_que_crea_bar(variable_1, variable2
                           variable_3)
```

## Líneas en blanco verticales

A continuación se presentan tres pautas clave sobre cómo usar los espacios en blanco verticales.

- Separar las definiciones de las clases y funciones con dos líneas en blanco.

```
#Líneas verticales en blanco

class MyFirstClass:
    pass

class MySecondClass:
    pass

def top_level_function():
    return None
```

- Los métodos dentro de clases se separan con una línea en blanco.

```
class MyClass:
    def first_method(self):
        return None

    def second_method(self):
        return None
```

- Se recomienda utilizar líneas en blanco para separar partes del código, por ejemplo dentro de una función, que realizan tareas diferenciadas.

```
def calculate_variance(number_list):
    sum_list = 0
    for number in number_list:
        sum_list = sum_list + number
    mean = sum_list / len(number_list)

    sum_squares = 0
    for number in number_list:
        sum_squares = sum_squares + number**2
    mean_squares = sum_squares / len(number_list)

    return mean_squares - mean**2
```



## Espacios en blanco en expresiones y declaraciones

A continuación se resumen algunas de las recomendaciones que se describen en PEP8.

### Situaciones molestas

Debemos evitar espacios en blanco superfluos en las siguientes situaciones

- Inmediatamente dentro de paréntesis, corchetes o llaves:

```
# Si:
spam(ham[1], {eggs: 2})

# No:
spam( ham[ 1 ], { eggs: 2 } )
```

- Entre una coma final y un paréntesis de cierre siguiente.

```
# Si:
foo = (0,)

#No:
bar = (0, )
```

- Inmediatamente antes de una coma, punto y coma o dos puntos.

```
# Si:
if x == 4: print(x, y); x, y = y, x

# No:
if x == 4 : print(x , y) ; x , y = y , x
```

- Inmediatamente antes del paréntesis abierto que inicia la lista de argumentos de una llamada de función.

```
# Si:
spam(1)

# No:
spam (1)
```

- Inmediatamente antes del paréntesis abierto que inicia una indexación o división.

```
# Si:
dct['key'] = lst[index]

# No:
dct ['key'] = lst [index]
```

- Más de un espacio alrededor de un operador de asignación (u otro) para alinearlos con otro.

```
# Si:
x = 1
y = 2
long_variable = 3

#No:
x           = 1
y           = 2
long_variable = 3
```

## Otras recomendaciones

- Evitar los espacios en blanco finales en cualquier lugar. Como suele ser invisible, puede resultar confuso.
- Rodear siempre estos operadores binarios con un solo espacio a cada lado:
  - asignación (=)
  - asignación aumentada (+=, -= etc.)
  - comparaciones (==, <, >, !=, <>, <=, >=, in, , , )
  - booleanos (and, or, not).

```
# Si:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# No:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

- Las anotaciones de función deben usar las reglas normales para los dos puntos y siempre tener espacios alrededor de la ->flecha, si está presente.

```
# Si:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...

# No:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

- No usar espacios alrededor del signo = cuando se usa para indicar un argumento de palabra clave, o cuando se usa para indicar un valor predeterminado para un parámetro de función sin anotaciones.

```
# Si:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

# No:
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Sin embargo, al combinar una anotación de argumento con un valor predeterminado, debemos utilizar espacios alrededor del signo =:

```
# Si:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...

# No:
def munge(input: AnyStr=None): ...
def munge(input: AnyStr, limit = 1000): ...
```

## Imports

A continuación se resumen algunas de las recomendaciones que se describen en PEP8.

- Las importaciones deben realizarse en líneas separadas.

```
# Si:
import os
import sys

# No:
import os, sys
```

- Sin embargo cuando se importen varios elementos de una misma librería, si sería correcto importarlos en la misma línea.

```
# Si:
from matplotlib import pyplot
```

- Las importaciones siempre se colocan en la parte superior del archivo, justo después de los comentarios y cadenas de documentación del módulo, y antes de las constantes y globales del módulo.
- Las importaciones deben agruparse en el siguiente orden y se debe poner una línea en blanco entre cada grupo de importaciones.
  - Importaciones de biblioteca estándar.
  - Importaciones de terceros relacionados.
  - Importaciones específicas de bibliotecas/aplicaciones locales.

```
# Si:

# Bibliotecas estándar
import os
import sys

# Bibliotecas de terceros
import from geo.Geoserver import Geoserver

# Importaciones aplicaciones locales
import my_module
```

- Debe se recomienda utilizar imports absolutos, aunque se permiten imports relativos.
- No se recomienda utilizar import \*.

## Comentarios

A continuación se resumen algunas de las recomendaciones que se describen en la PEP8.

- Los comentarios que contradicen el código son peores que no comentar.
- Los comentarios deben actualizarse cuando se actualiza el código.
- Los comentarios deben ser oraciones completas. La primera palabra debe estar en mayúscula, a menos que sea un identificador que comience con una letra minúscula.
- Los comentarios en bloque generalmente consisten en uno o más párrafos contruidos a partir de oraciones completas, y cada oración termina en un punto.
- Debemos utilizar dos espacios después de un punto final de oración en comentarios de varias oraciones, excepto después de la oración final.
- Debemos asegurarnos que los comentarios sean claros y fácilmente comprensibles para otros hablantes del idioma en el que está escribiendo.
- Se recomienda escribir los comentarios en inglés, a menos que estemos seguros que serán leídos por personas que no dominen ese idioma.

## Docstrings

Las convenciones para escribir docstrings se encuentran en el [PEP 257](#). A continuación vemos algunas recomendaciones.

- Escribir docstrings para todos los módulos, funciones, clases y métodos públicos.
- Los docstrings no son necesarios para los métodos no públicos, pero debe tener un comentario que describa lo que hace el método. Este comentario debe aparecer después de la línea que inicia con un **def**.

## Tema 2. Flake8

### Objetivos

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Comprender que es Flake8.
- Instalar y utilizar Flake8 para revisar tu código.

### Introducción

En el tema anterior aprendimos sobre **algunas de las normas** que nos propone PEP8. A veces puede resultar complicado acordarnos de todas y cada una de ellas.

Afortunadamente existen algunas herramientas que nos pueden ayudar a ser mejores programadores, realizando la revisión de nuestro código. En ese sentido existen dos tipos de herramientas:

- **Linters:** analizan el código estáticamente para marcar errores de programación, detectar errores, errores de estilo y construcciones sospechosas. Ejemplos: [Flake8](#) y [Pycodestyle](#).
- **Formatter:** corrigen el estilo (espaciado, saltos de línea, comentarios), lo que nos ayuda a cumplir las reglas de programación y formato. Ejemplos: [Black](#) y [Autopep8](#).

Utilizar este tipo de herramientas nos convierte en mejores desarrolladores, ya que:

- Nos ayuda a escribir mejor código, comprobando los estándares de codificación.
- Nos ayuda a prevenir errores de sintaxis, mal formato, etc.
- Si estamos trabajando en equipo, nos permite ahorrar tiempo en la revisión del código.
- Son fáciles de utilizar.
- Son gratuitas.

## ¿Qué es Flake 8?

Flake8 es un [wrapper](#) que se basa en las siguientes herramientas:

- [PyFlakes](#): es un programa simple que verifica los archivos fuente de Python en busca de errores. Pyflakes analiza programas y detecta varios errores. Funciona analizando el archivo fuente, sin importarlo.
- [PyCodeStyle](#): es una herramienta para comparar el código Python con algunas de las convenciones de estilo en PEP8.
- [Ned Batchelder's McCabe script](#): es una herramienta que proporciona una medición de la complejidad ciclomática de un programa.

Flake8 ejecuta todas estas herramientas simplemente ejecutando el comando `flake8` y muestra todos los warnings por archivo.

Para instalarlo debemos abrir una terminal, posicionarnos en el entorno de python y ejecutar **`pip install flake8`**.

```
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> pip install flake8
Collecting flake8
  Downloading flake8-5.0.4-py2.py3-none-any.whl (61 kB)
    |-----| 61.9/61.9 KB 662.3 kB/s eta 0:00:00
Collecting pyflakes<2.6.0,>=2.5.0
  Downloading pyflakes-2.5.0-py2.py3-none-any.whl (66 kB)
    |-----| 66.1/66.1 KB 901.2 kB/s eta 0:00:00
Collecting mccabe<0.8.0,>=0.7.0
  Downloading mccabe-0.7.0-py2.py3-none-any.whl (7.3 kB)
```

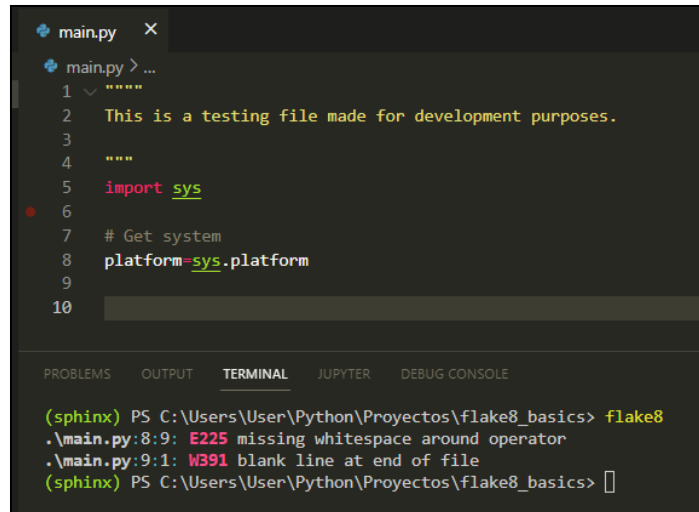
## Utilización

Una vez instalado, podremos comenzar a utilizarlo ejecutando el comando `flake8` en la terminal. Allí podremos ver el resultado de la inspección a nuestro código que será presentado a nosotros mediante un código una descripción.

En la página [Flake8 Rules](#) podemos consultar todos los códigos que nos proporciona Flake8, encontrar ejemplos, anti ejemplos y links a la documentación de PEP8 donde se describe la norma involucrada.

## Ejemplo 1:

En el siguiente código podemos visualizar las siguientes recomendaciones:



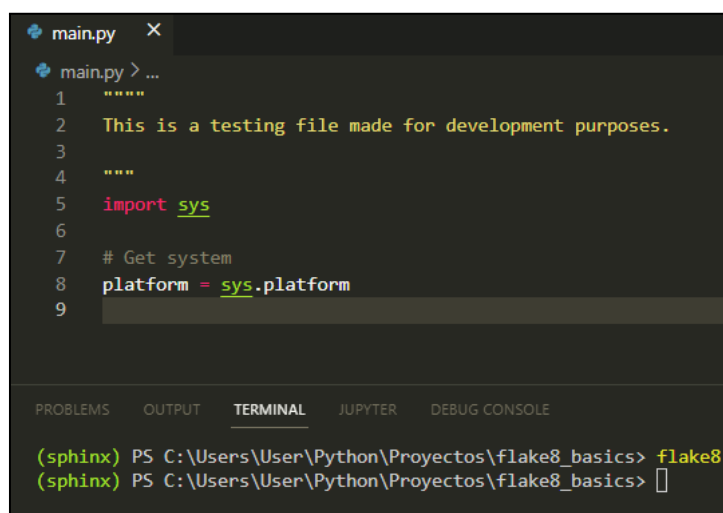
```

main.py
1  """
2  This is a testing file made for development purposes.
3
4  """
5  import sys
6
7  # Get system
8  platform=sys.platform
9
10
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> flake8
.\main.py:8:9: E225 missing whitespace around operator
.\main.py:9:1: W391 blank line at end of file
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics>

```

- [E225](#): ya que olvidamos dejar un espacio alrededor del operador `=`.
- [W391](#): ya que dejamos 2 líneas en blanco en la parte inferior de nuestro archivo.

Si corregimos esas recomendaciones y ejecutamos nuevamente flake8, podremos observar que ya no tenemos recomendaciones.



```


main.py
1  """
2  This is a testing file made for development purposes.
3
4  """
5  import sys
6
7  # Get system
8  platform = sys.platform
9
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> flake8
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics>

```



## Ejemplo 2:

En el siguiente ejemplo agregamos una línea de comentario que supera los 79 caracteres y obtuvimos el código de recomendación [E501](#).



```

4  """
5  import sys
6
7  # Get system
8  platform = sys.platform
9
10 # Long Comment test
11 # Lorem ipsum dolor sit amet consectetur adipisicing elit. Do
    voluptates ab quas alias? Quaerat quam quo delectus est. Cumq
    repellat iusto voluptatem tenetur maiores labore adipisci rer
    fugit.
12

```

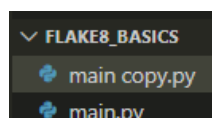
```

(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> flake8
.\main.py:11:80: E501 line too long (224 > 79 characters)
(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics>

```

## Ejemplo 3:

En el siguiente ejemplo duplicamos el archivo main, teniendo dos archivos .py en la carpeta de proyecto.

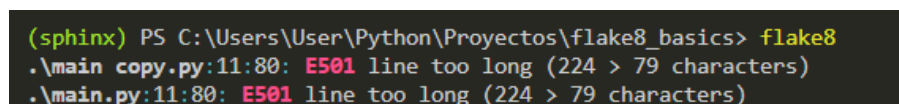


```

▼ FLAKE8_BASICS
  main copy.py
  main.py

```

Si ejecutamos el comando flake8 posicionados en la carpeta que contiene ambos archivos, podremos observar cómo son revisados.

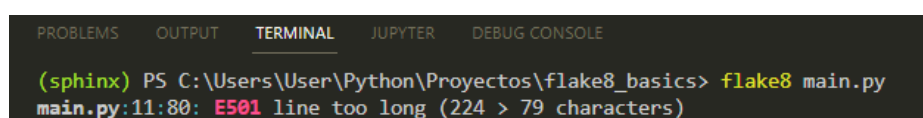


```

(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> flake8
.\main copy.py:11:80: E501 line too long (224 > 79 characters)
.\main.py:11:80: E501 line too long (224 > 79 characters)

```

En caso de necesitar revisar un archivo en particular, podemos ejecutar el flake8 [nombre del archivo]. En este caso flake8 main.py



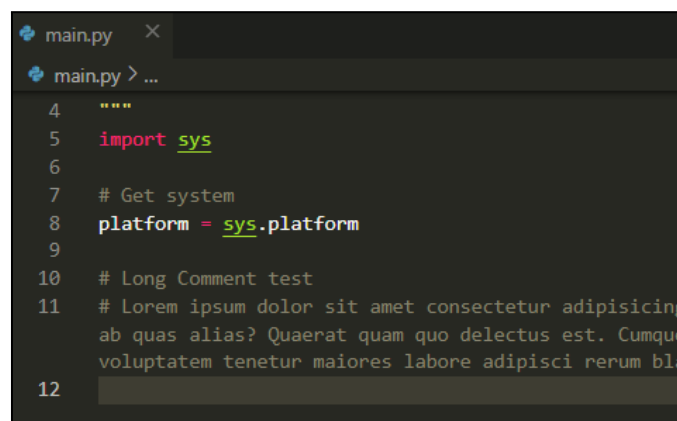
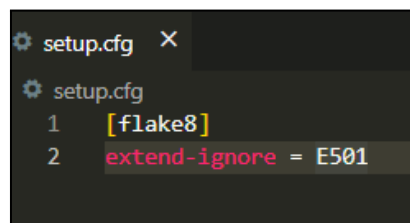
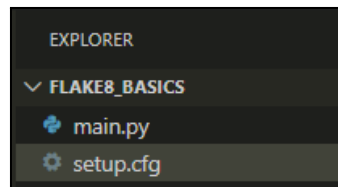
```

(sphinx) PS C:\Users\User\Python\Proyectos\flake8_basics> flake8 main.py
main.py:11:80: E501 line too long (224 > 79 characters)

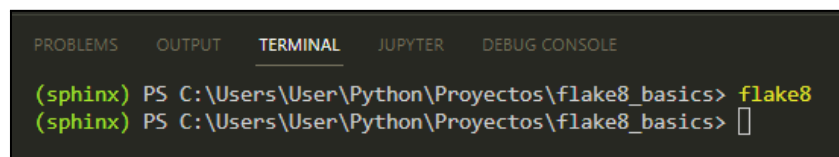
```

## Ejemplo 4:

En este ejemplo crearemos un archivo de configuración **setup.cfg** para modificar el comportamiento de Flake8 de forma tal que ignore el código E501 (más de 79 caracteres).



Podemos comprobar que al ejecutar flake8 ya no se detecta esa recomendación.

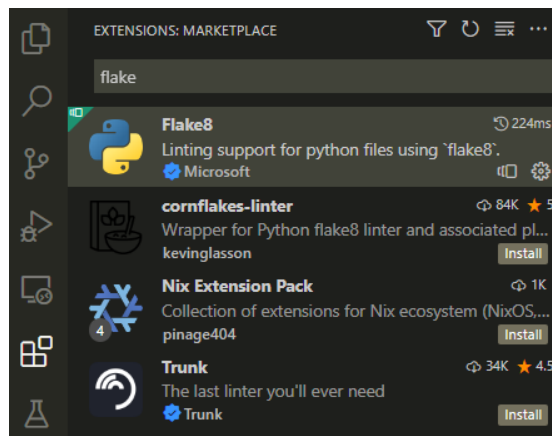


Para más información, podemos consultar la [documentación oficial](#).

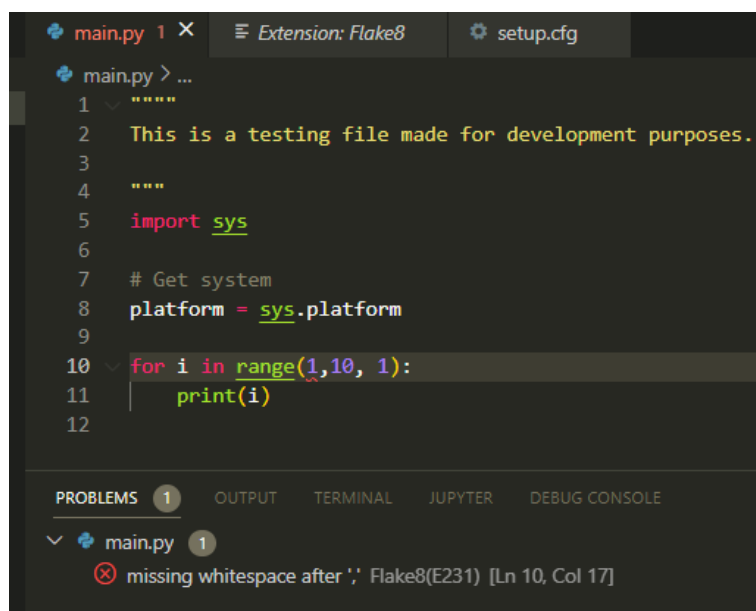
## Ejemplo 5:

Podemos implementar una extensión en nuestro editor de código para incorporar Flake8 y evitar la ejecución del comando `flake8` múltiples veces.

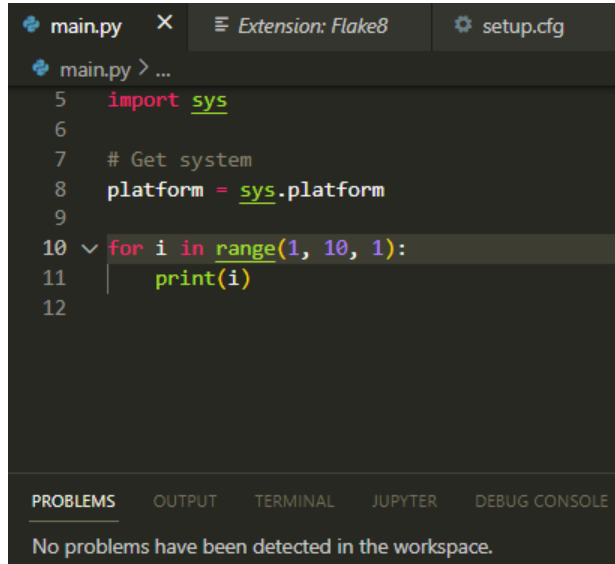
En ese sentido, el marketplace de Visual Studio Code pone a disposición la extensión Flake8, la cual se encuentra en un estado de “pre-release”, pero que aún así se puede instalar sin inconvenientes.



Una vez instalada, cada vez que guardemos nuestro archivo `.py` se ejecutará Flake8 y nos mostrará los resultados en la solapa Problems.



Una vez que hayamos corregido nuestro código y lo hayamos guardado, se nos borrará el error



```

main.py x  Extension: Flake8  setup.cfg
main.py > ...
5  import sys
6
7  # Get system
8  platform = sys.platform
9
10 v for i in range(1, 10, 1):
11     print(i)
12
PROBLEMS  OUTPUT  TERMINAL  JUPYTER  DEBUG CONSOLE
No problems have been detected in the workspace.

```

También podemos utilizar Flake8 en otros editores de código como [Atom](#).



## Cierre

Durante esta unidad aprendimos sobre PEP8 y Flake8.

Comenzamos desarrollando PEP8 y su importancia en el ámbito de los desarrolladores, ya que permite tener un código más legible para nosotros mismos y también para nuestros colegas.

Luego realizamos un recorrido por algunas de las normas de PEP8, viendo ejemplos y anti ejemplos para aprender cómo proceder en cada caso.

Finalmente incorporamos el paquete Flake8, el cual nos permitió revisar nuestro código automáticamente de acuerdo a PEP8 y de esta manera evitar recordar todas las normas.

Quedamos a disposición de las consultas que puedan surgir.



## Referencias

**PEP 8 – Style Guide for Python Code** | [peps.python.org](https://peps.python.org/pep-0008/). Recuperado de: <https://peps.python.org/pep-0008/>

**Flake8 Rules** | **Flake8 Rules**. Recuperado de: <https://www.flake8rules.com/>

**How to Write Beautiful Python Code With PEP 8** | **Real Python**.  
Recuperado de: <https://realpython.com/python-pep8/>

tiiiiiit by 