



# Buenas prácticas de programación en Python

**LECCIÓN 4: DEPURACIÓN Y REGLAS DE OPTIMIZACIÓN DE CÓDIGO.**

# ÍNDICE

<b>Presentación y Objetivos .....</b>	<b>2</b>
<b>1. Depuración de Código .....</b>	<b>3</b>
1.1 Creando puntos de parada .....	7
1.2 Reanudando la ejecución .....	7
1.3 Mostrando código adyacente .....	8
1.4 Avanzando línea a línea .....	9
1.5 Inspeccionando el contenido de nuestras variables .....	9
1.6 Eliminando puntos de parada .....	10
<b>2. Técnicas de optimización de código.....</b>	<b>12</b>
2.1 Listas de compresión .....	12
2.2 Map, filter y reduce .....	14
<b>3. Puntos clave .....</b>	<b>16</b>

# Depuración y reglas de optimización de código

## PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos a utilizar el depurador de código nativo de Python *pdb*. Adicionalmente, estudiaremos una serie de funciones que nos permitirán optimizar nuestros códigos implementados en Python. Estas funciones o reglas no solo conseguirán optimizar nuestro código, sino que aumentarán la interpretabilidad y elegancia de nuestros programas.



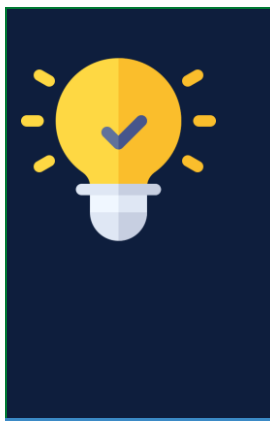
### Objetivos

- Comprender qué es el depurador de Python (*pdb*).
- Estudiar cómo el uso del depurador *pdb* nos permite ejecutar nuestras implementaciones línea a línea para identificar y solucionar posibles errores.
- Comprender qué es y cuál es la ventaja de integrar comprensión de listas en nuestras implementaciones.
- Comprender el potencial de las funciones *map*, *filter* y *reduce*.

## 1. DEPURACIÓN DE CÓDIGO

Depurar un programa es el proceso a través del cual identificamos y corregimos errores de programación. En inglés, a este proceso se le denomina como *debugging* porque se asemeja al acto de eliminar *bugs*, manera en la que se conocen a los errores de programación.

Para depurar un programa, se hace uso de un depurador. En Python, el depurador por excelencia es el **pdb** (Python DeBugger).



### **Importante**

Un depurador es un programa utilizado para detectar y eliminar los errores de otro programa (en nuestro caso, un programa escrito en Python). El depurador permite analizar el código instrucción a instrucción para identificar los problemas presentes en el código.

Con la herramienta *pdb* es posible insertar puntos de parada dentro del código fuente sin necesidad de usar un IDE o herramientas externas. Como cualquier otro depurador, *pdb* está dotado de las capacidades necesarias para poder imprimir el valor de las variables en un determinado punto de la ejecución, ejecutar y evaluar el código línea a línea e incluso examinar el funcionamiento de las funciones de nuestro código.

En este apartado mostraremos a través de un ejemplo guiado cuál es el funcionamiento del depurador de Python (*pdb*).

Para comenzar, necesitamos escribir un programa en Python para poder utilizar el depurador de Python. A continuación se muestra un programa de ejemplo para resolver una serie de operaciones con números enteros. Entre estas operaciones se destacan: listar los *n* primeros números primos, calcular el factorial de un número y determinar si un número dado es un número feliz.

**NOTA:** puede encontrar el código fuente de este ejemplo adjunto al material de esta lección.



### ***Presta atención***

Un número se define como feliz si la suma de los cuadrados de los dígitos del número es igual a 1. Por ejemplo, el número 7 es feliz porque:

$$7^2=49$$

$$4^2+9^2=97$$

$$9^2+7^2=130$$

$$1^2+3^2+0^2=10$$

$$1^2+0^2=1$$

```
def sumaCuadrados(n):
    sumaCuadrados = 0
    while(n):
        sumaCuadrados += (n % 10) * (n % 10)
        n = n // 10
    return sumaCuadrados

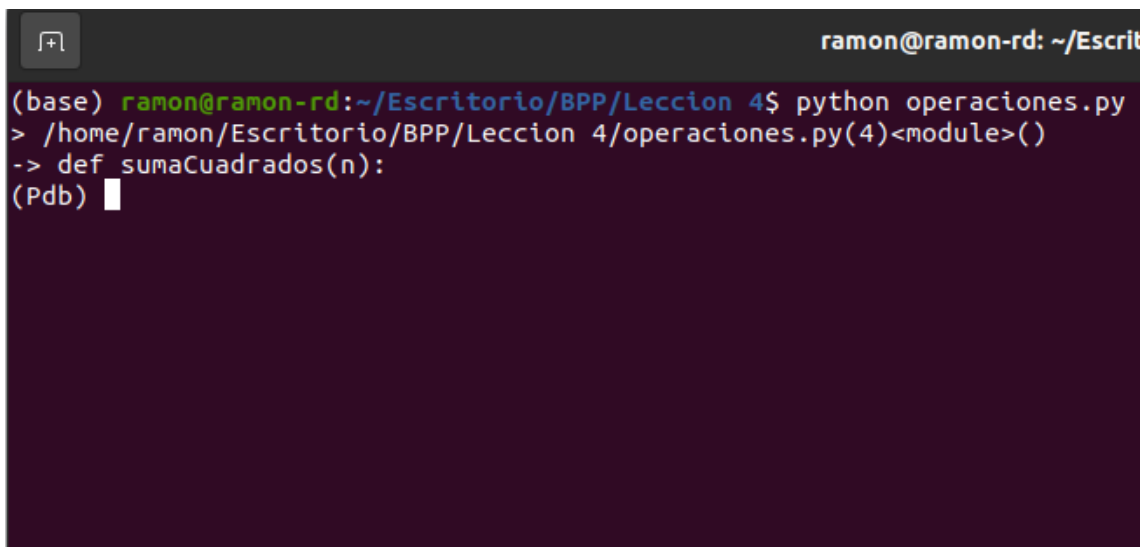
def esNumeroFeliz(n):
    s = n
    f = n
    while(True):
        s = sumaCuadrados(s)
        f = sumaCuadrados(sumaCuadrados(f))
        if(s != f):
            continue
        else:
            break
    return(s == 1)
```

```
def factorial(n):  
    resultado = 1  
    for i in range(1,n+1):  
        resultado *= i  
    return resultado  
  
def es_primo(n):  
    primo = True  
    for i in range(2, n):  
        if(n%i == 0):  
            primo = False  
    return primo  
  
def listar_n_primos(n):  
    for i in range(2,n+1):  
        if(es_primo(i)): print(f"{i} es primo")
```

```
if __name__ == '__main__':  
    esNumeroFeliz(7)  
    esNumeroFeliz(8)  
    factorial(5)  
    es_primo(17)  
    es_primo(4)
```

En las capturas anteriores se muestra el código implementado para las funciones mencionadas anteriormente. Si ejecutamos este código, comprobaremos que no obtenemos ningún resultado por pantalla (con la excepción de la función *listar\_n\_primos*, que incluye una llamada a la función *print* para mostrar el resultado por pantalla. Para saber qué está sucediendo dentro de las funciones que hemos implementado (sin incluir mensajes por pantalla usando la función *print*), vamos a utilizar el depurador *pdb* para inspeccionar cada una de las líneas que componen nuestras funciones. Para ello, incluimos al inicio de nuestro programa las siguientes líneas y ejecutamos de nuevo nuestro código:

```
import pdb  
pdb.set_trace()
```



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4$ python operaciones.py
> /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(4)<module>()
-> def sumaCuadrados(n):
(Pdb) █
```

En la captura anterior podemos comprobar que nuestro programa se ha detenido en la primera línea de nuestro programa, justo después del punto donde se importó la librería *pdb*. El depurador nos indica en primer lugar la ubicación del script que se está ejecutando (en este caso de ejemplo, la ruta es “/home/ramon/Escritorio/BPP/Leccion 4/operaciones.py”, y entre paréntesis el número de la línea del programa (4) donde se ha detenido la ejecución. A continuación, el depurador muestra el código fuente que hace referencia al punto donde se ha detenido la ejecución. Por último, se muestra el *prompt* de *pdb*, que nos permitirá ingresar un conjunto de comandos para indicarle al depurador qué hacer.

A continuación se muestra una lista del conjunto de comandos más utilizados con este depurador. Cada uno de estos serán mostrados en detalle a continuación.

- Crear puntos de parada (breakpoints)
- Reanudación de la ejecución (continue).
- Mostrar código adyacente
- Avanzar línea a línea
- Inspector de variables
- Eliminar puntos de parada (breakpoints)

## 1.1 Creando puntos de parada

Los puntos de parada, o breakpoints en inglés, son utilizados para mostrarle al depurador *pdb* en qué punto de nuestro programa queremos que se detenga la ejecución, con el objetivo de detallar qué está sucediendo en ese fragmento de código y depurar posibles errores.

Para insertar un punto de parada dentro de nuestro código haciendo uso de *pdb* utilizaremos el comando *break*. Este comando recibe como parámetro un número entero, que hace referencia a la línea donde queremos insertar el punto de parada. Partiendo del ejemplo descrito en la sección anterior, escribimos en el prompt de *pdb* el comando *break* seguido de la línea dónde vamos a detener nuestra ejecución. En este caso práctico, deseamos detener la ejecución dentro de la función *sumaCuadrados*, más específicamente en la línea 7. A lo que obtendremos la siguiente respuesta:

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python operaciones.py
> /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(4)<module>()
-> def sumaCuadrados(n):
(Pdb) break 7
Breakpoint 1 at /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py:7
(Pdb) █
```

Esto nos indica que el primer punto de parada que hemos creado se ha realizado con éxito, y se ha creado en nuestro programa “operaciones.py”, en la línea 7.

## 1.2 Reanudando la ejecución

En el instante en que queramos retomar la ejecución de nuestro programa basta con utilizar el comando *continue* en el *prompt* de *pdb*:

```
(Pdb) continue
> /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(7)sumaCuadrados()
-> sumaCuadrados += (n % 10) * (n % 10)
(Pdb) █
```



Nuestro programa seguirá con el transcurso de la ejecución, hasta encontrar un punto de parada o hasta que finalice con la ejecución de todo el programa. Como podemos comprobar en la captura anterior, nuestro código se ha ejecutado hasta encontrar el primer punto de parada en la línea 11, que se corresponde con la operación de sumar los cuadrados de los dígitos de un número.

### 1.3 Mostrando código adyacente

Puesto que existe la necesidad de depurar programas con un número elevado de líneas de código, es bastante frecuente sentirse perdido y no saber exactamente en qué punto de nuestro código nos encontramos o a qué función pertenece la línea en la que estamos detenidos.

Para solucionar esta casuística, *pdb* cuenta con un comando que nos permite mostrar las líneas de código adyacentes a la línea en la que se encuentra el punto de parada que estemos analizando. Hablamos del comando *list*:

```
(Pdb) list
2     pdb.set_trace()
3
4     def sumaCuadrados(n):
5         sumaCuadrados = 0
6         while(n):
7 B->         sumaCuadrados += (n % 10) * (n % 10)
8             n = n // 10
9         return sumaCuadrados
10
11     def esNumeroFeliz(n):
12         s = n
(Pdb) █
```

Nótese cómo en la captura anterior, tras ejecutar la orden *list* en el *prompt* de *pdb*, se muestra por pantalla el conjunto de líneas adyacentes a la que estamos analizando actualmente. Adicionalmente, se muestra en la línea 7 la letra **B** junto con una flecha → a modo de indicador del punto en el que se encuentra detenida la ejecución de nuestro programa. Por defecto, el comando *list* muestra por pantalla las 11 líneas de código que existen alrededor de la línea donde se encuentra el punto de parada.

## 1.4 Avanzando línea a línea

Como hemos visto, el comando *continue* le indica al *pdb* que debe seguir con la ejecución de nuestro programa hasta que esta finalice, o hasta que encuentre otro punto de parada definido. Sin embargo, *pdb* también nos permite ejecutar nuestro código línea a línea, tomando como partida el punto de parada que hayamos definido previamente:

```
(Pdb) next
> /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(8) sumaCuadrados()
-> n = n // 10
(Pdb) █
```

Como se muestra en la captura anterior, se ha terminado de ejecutar la línea 7 y actualmente, el depurador se encuentra detenido en la línea 8.

## 1.5 Inspeccionando el contenido de nuestras variables

Otro comando que resulta de gran importancia a la hora de depurar nuestro código con *pdb* es la orden *p*. Este comando se ejecuta junto con el nombre de una variable, con el objetivo de mostrar su contenido:

```
/home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(8) sumaCuadrados()
-> sumaCuadrados += (n % 10) * (n % 10)
(Pdb) list
2     pdb.set_trace()
3
4     def sumaCuadrados(n):
5         sumaCuadrados = 0
6         while(n):
7 B->         sumaCuadrados += (n % 10) * (n % 10)
8             n = n // 10
9         return sumaCuadrados
10
11     def esNumeroFeliz(n):
12         s = n
(Pdb) p sumaCuadrados
0
(Pdb) █
```

Inicialmente, la variable *sumaCuadrados* fue inicializada a 0, es por ello que el inspector de variable nos ha devuelto el resultado 0. Para verificar que su contenido cambia podemos ejecutar una vez la orden *next* y volvemos a mostrar el contenido de la variable *sumaCuadrados*:

```
(Pdb) next
> /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py(8)sumaCuadrados()
-> n = n // 10
(Pdb) p sumaCuadrados
49
(Pdb) █
```

Por ahora, tal y como podemos ver en los resultados de la captura anterior, nuestro código se está ejecutando correctamente.

## 1.6 Eliminando puntos de parada

Al igual que *pdb* nos permite insertar puntos de parada dentro de nuestro código, también nos permite eliminarlos. Tenemos dos formas de proceder:

- Eliminar un punto de parada determinado. Para ello ejecutamos el comando *clear n*, donde *n* es el número de punto de parada a eliminar (recuerda que al crear un punto de parada, *pdb* le asignó un número).

```
(Pdb) clear 1
Deleted breakpoint 1 at /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py:7
(Pdb) █
```

- Eliminar todos los puntos de parada existentes en nuestro código. Para ello, ejecutamos el comando *clear*. Tras ejecutarlo, nos pedirá confirmación, a lo que contestaremos con la letra *y*.

```
(Pdb) clear
Clear all breaks: y
Deleted breakpoint 2 at /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py:4
Deleted breakpoint 3 at /home/ramon/Escritorio/BPP/Leccion 4/operaciones.py:11
(Pdb) █
```



### ***Para más información***

Cabe destacar que la herramienta *pdb* dispone de una gran cantidad de comandos que facilitan la tarea de depurar un código escrito en Python. Se recomienda visitar la página oficial de *pdb*, donde se muestra con todo nivel de detalle cada una de sus funcionalidades:

<https://docs.python.org/3/library/pdb.html>

## 2. TÉCNICAS DE OPTIMIZACIÓN DE CÓDIGO

En esta sección se pretende mostrar una serie de técnicas y recomendaciones para mejorar la eficiencia de nuestros programas escritos en Python. En concreto estudiaremos el uso de:

- Listas de compresión
- Uso de las funciones **map**, **filter** y **reduce**

### 2.1 Listas de compresión

Supongamos que tenemos una frase y deseamos separar cada una de las letras que la componen. Por ejemplo, para la frase "Python es el mejor lenguaje de programación", queremos separar cada uno de sus caracteres de la siguiente manera: 'P','y','t','h','o','n',' ','e','s',' ','e','l',' ','mejor',' ','lenguaje',' ','d','e',' ','p','r','o','g','r','a','m','a','c','i','ó','n'.

Con lo que hemos aprendido hasta ahora, probablemente la forma de proceder sería construir un programa como el siguiente:

```
letras = []  
  
for i in "Python es el mejor lenguaje de Programación":  
    letras.append(i)  
  
print(letras)
```

Y tras ejecutarlo, obtenemos el siguiente resultado:

```
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python comprension.py  
['P', 'y', 't', 'h', 'o', 'n', ' ', 'e', 's', ' ', 'e', 'l', ' ', 'm',  
'e', 'j', 'o', 'r', ' ', 'l', 'e', 'n', 'g', 'u', 'a', 'j', 'e', ' ',  
'd', 'e', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'a', 'c', 'i', 'ó',  
'n']  
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$
```

Sin embargo, existe una forma más eficiente y elegante de realizar esta operación, haciendo uso de **comprensión de listas**.

La sintaxis de este tipo de operación es la siguiente:

*[expresion for item in list]*

Siguiendo la sintaxis anterior, podríamos escribir nuestro código de la siguiente manera:

```
letras2 = [i for i in "Python es el mejor lenguaje de Programación"]  
print(letras2)
```

Adicionalmente, también podemos incluir condicionales dentro de la comprensión de listas, siguiendo la siguiente sintaxis:

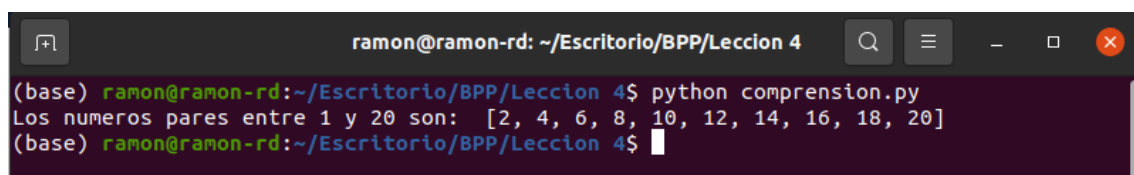
*[expresion for item in list if condicion]*

*[resultado1 if condicion1 else resultado2 for item in list]*

En el siguiente ejemplo resolvemos un problema ya conocido en este máster: encontrar los  $n$  números pares. En este caso lo resolveremos haciendo uso de comprensión de listas:

```
pares = [i for i in range(1,21) if i%2==0]  
print("Los numeros pares entre 1 y 20 son: ", pares)
```

A lo que obtenemos el siguiente resultado:



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4  
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python comprension.py  
Los numeros pares entre 1 y 20 son: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]  
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$
```

Por norma general, el uso de compresión de listas aumenta la eficiencia de nuestro código, ya que el tiempo de ejecución necesario para recorrer y operar con una lista de elementos es menor utilizando compresión de listas que haciendo uso de bucles tradicionales.

## 2.2 Map, filter y reduce

Existen tres funciones que, incluyéndolas cuando nuestro programa lo permita, mejoran la eficiencia e interpretabilidad de nuestro código.

En primer lugar, la función *map* nos permite aplicar una determinada función a un conjunto de elementos (uno por uno). Por ejemplo, imaginemos que hemos implementado el siguiente fragmento de código para calcular el cuadrado de un conjunto de números enteros contenidos en una lista:

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4
elementos = [2, 3, 5, 4, 1, 3]
cuadrados = []

for i in elementos:
    cuadrados.append(i**2)

print("El cuadrado de la lista ", elementos, " es ", cuadrados)
```

```
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python map.py
El cuadrado de la lista [2, 3, 5, 4, 1, 3] es [4, 9, 25, 16, 1, 9]
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$
```

La función *map* nos permite implementar esta operación de una forma alternativa más simple y eficiente:

```
def cuadrado(n):
    return n**2

cuadrados2 = list(map(cuadrado, elementos))

print("El cuadrado de la lista ", elementos, " es ", cuadrados2)
```

```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python map.py
El cuadrado de la lista [2, 3, 5, 4, 1, 3] es [4, 9, 25, 16, 1, 9]
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$

```

Otra función que nos permite implementar códigos más eficientes y simples es la función *filter*. Tal y como indica su nombre, esta función crea una lista de elementos para los cuales una determinada función ha devuelto *True*. Por ejemplo:

```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4
def es_par(n):
    return n%2==0

numeros = list(range(1,21))

pares = list(filter(es_par, numeros))

print(pares)
~

```

```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$ python filter.py
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4$

```

Por último, la función *reduce* nos permite realizar cálculos sobre los elementos de una lista y devolver un resultado determinado. A diferencia de las funciones *map* y *filter*, para poder utilizar la función *reduce* debemos importarla del paquete *functools*.



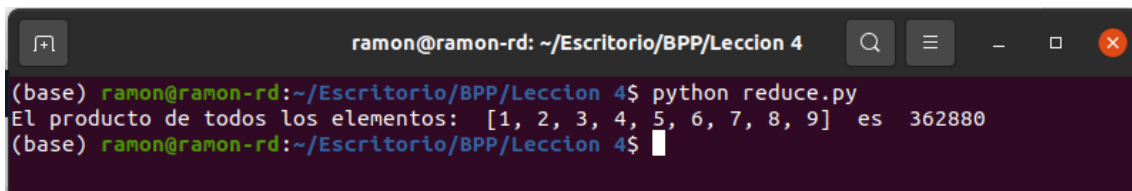
A continuación mostramos un ejemplo de su uso. Supongamos que queremos calcular el producto de una lista de números:

```
from functools import reduce

def multiplica(m,n):
    return m*n

numeros = list(range(1,10))
producto = reduce(multiplica, numeros)

print("El producto de todos los elementos: ", numeros, " es ", producto)
```



A terminal window titled 'ramon@ramon-rd: ~/Escritorio/BPP/Leccion 4' showing the execution of a Python script. The prompt is '(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4\$'. The command 'python reduce.py' has been entered, and the output is 'El producto de todos los elementos: [1, 2, 3, 4, 5, 6, 7, 8, 9] es 362880'. The prompt is now '(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion 4\$'.

### 3. PUNTOS CLAVE

En esta lección hemos aprendido

- El depurador de Python *pdb* es una herramienta fundamental que debe conocer todo programador de Python para identificar y solucionar posibles problemas en nuestras implementaciones.
- La comprensión de listas es una forma de recorrer y operar con listas más eficiente que los métodos tradicionales estudiados hasta ahora.
- Toda comprensión de listas puede ser escrita con bucles tradicionales, pero todo bucle no tiene porque poder ser implementado en forma de comprensión de listas.
- El uso de las funciones *map*, *reduce* y *filter* nos permite operar sobre listas de una forma más eficiente y elegante.

