Índice

[**El Zen de Python** 1](#_Toc109844146)

[**¿Qué es la documentación?** 2](#_Toc109844147)

[**¿Qué es un entorno virtual?** 3](#_Toc109844148)

[**Creación de un entorno virtual** 3](#_Toc109844149)

[**Creando un ambiente virtual con VENV** 3](#_Toc109844150)

[**Activación del ambiente virtual** 3](#_Toc109844151)

[**¿Qué es pip?** 4](#_Toc109844152)

[**Instalación de dependencias con pip** 4](#_Toc109844153)

[**Importante** 4](#_Toc109844154)

[**Comando >** 4](#_Toc109844155)

[**Instaladores de paquetes en python** 5](#_Toc109844156)

[**List comprehensions** 5](#_Toc109844157)

[**List comprehensions** 6](#_Toc109844158)

[**Estructura de una list comprehensions** 6](#_Toc109844159)

[**Qué significa cada sección** 6](#_Toc109844160)

[**¿Como se lee un list comprehensions?** 7](#_Toc109844161)

[**Dictionary comprehensions** 7](#_Toc109844162)

[**Estructura de un dictionary comprehensions** 7](#_Toc109844163)

[**Qué significa cada sección** 7](#_Toc109844164)

[**¿Como se lee un dictionary comprehensions?** 8](#_Toc109844165)

[**Funciones anónimas: lambda** 8](#_Toc109844166)

[**High order functions: filter, map y reduce** 9](#_Toc109844167)

[**Los errores en el código** 11](#_Toc109844168)

[**Lectura de un traceback** 12](#_Toc109844169)

[**Elevar una excepción** 12](#_Toc109844170)

[**Debugging** 12](#_Toc109844171)

[**Manejo de Excepciones** 13](#_Toc109844172)

[**Jerarquía de las excepciones y errores más comunes** 14](#_Toc109844173)

[**Assert statements** 15](#_Toc109844174)

[**Diferencia importante entre usar ‘assert’ o ‘try-except’** 15](#_Toc109844175)

[**¿Cómo trabajar con archivos?** 16](#_Toc109844176)

[**Modos de Apertura** 16](#_Toc109844177)

# **El Zen de Python**

El **Zen** de Python se compone por los principios para escribir tu código de manera clara, sencilla y precisa. Estos son:

* **Bello es mejor que feo:** Python es estéticamente superior a cualquier otro lenguaje de programación. Al momento de escribir código, es mejor que sea de manera limpia y estética.
* **Explícito es mejor que implícito:** Hacer más fácil que las otras personas entiendan el código.
* **Simple es mejor que complejo:** Es mejor tener una implementación simple, que ocupe pocas líneas de código y sea entendible, a que sea una larga y complicada.
* **Complejo es mejor que complicado:** Si tenemos que extendernos en la implementación y hacerla más compleja para que el código si se entienda, esto es mejor que hacerlo simple y mal.
* **Plano es mejor que anidado:** El anidamiento es cuando tenemos un bloque de código dentro de otro bloque de código (dependiendo de este). Esto se nota en Python por la indentación, nos quedarían estos bloques muy corridos a la derecha.

Es mejor evitar el anidamiento, y hacer las cosas de manera plana.

* **Espaciado es mejor que denso:** Por la indentación de Python (sus sangrías), este principio se nos hace imposible de esquivar. El código inevitablemente es espaciado.
* **La legibilidad es importante:** Es importante que otros programadores puedan entender lo que estamos escribiendo. Esto hace más fáciles las cosas cuando trabajemos con otros en los proyectos.
* **Los casos especiales no son lo suficientemente especiales como para romper las reglas (sin embargo, la practicidad le gana a la pureza):** Siempre que podamos respetar estas reglas que nos plantea Python, es mejor así. Sin embargo, si por el hecho de hacer un código muy puro o muy ‘Pythonista’, este pierde legibilidad, es mejor ser más prácticos y romper o saltearnos algunas de estas reglas para que el código sea más eficiente. Por lo tanto, llegado el momento debemos decidir si es mejor hacer las cosas de manera pura o práctica.
* **Los errores nunca deberían pasar silenciosamente (a menos que se silencien explícitamente):** Manejar los errores es fundamental. Cada error nos dice algo y hay que prestarle atención. A menos que seas capaz de silenciar un error explícitamente, aunque para esto hay que tener criterio.
* **Frente a la ambigüedad, evitar la tentación de adivinar:** Nuestro código debería solamente tener una interpretación. Si en un contexto significa algo, y en otro otra cosa, es mejor que lo revisemos y busquemos una solución.
* **Debería haber una, y preferiblemente sola, una manera obvia de hacerlo. (A pesar de que esa manera no sea obvia a menos que seas holandés):** Esto hace referencia al creador de Python ''Guido van Rossum", que de manera muy inteligente encontrar las soluciones precisas a los problemas, y deberíamos imitarlo.
* **Ahora es mejor que nunca:** Es mejor desarrollar nuestra solución cuánto antes, no dejarlo para mañana o para más adelante.
* **A pesar de que nunca es muchas veces mejor que ahora mismo:** Si por hacer las cosas ya y tenemos poco tiempo, si es mejor dejarlo para después y no hacerlo apurado y mal.
* **Si la implementación es difícil de explicar, es una mala idea, y si es fácil de explicar, es una buena idea:** Si somos capaces de explicar nuestra implementación a otros desarrolladores paso a paso, es una buena idea. En cambio, si no podemos hacerlo, significa que ni nosotros entendemos la implementación y deberíamos repensar nuestra forma de encarar la solución.
* **Los espacios de nombres son una gran idea, ¡Tengamos más de esos! (namespaces):** Es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo. (Lo veremos a profundidad más adelante).

# **¿Qué es la documentación?**

La documentación es la biblia de cualquier programador.

No puedes aspirar a aprender un lenguaje si no lees documentación. Es la documentación quien nos va a decir exactamente cómo funciona el lenguaje (y cualquier tecnología). No hay un solo desarrollador profesional que no lea documentación.

* Documentación oficial de python [aquí](https://docs.python.org/3/).
* Documentación oficial de PEP8 [aquí](https://peps.python.org/pep-0008/).

# **¿Qué es un entorno virtual?**

Sería una catástrofe tener que **instalar** y **actualizar** módulos para **cada** proyecto cuidando que ninguno se rompa, porque fácilmente podrías actualizar un módulo que, para un proyecto funcione, pero para otro deje de funcionar, es por eso que se crea el concepto de **entornos virtuales**. Este concepto lo tienen muchos lenguajes, y lo genial es que soluciona muy bien el problema de la compatibilidad entre proyectos, porque **cada entorno virtual es** **independiente** y funciona con las versiones que se les hayan instalado ahí dentro.

# **Creación de un entorno virtual**

## **Creando un ambiente virtual con VENV**

* Creación de ambiente Virtual: **python3 -m venv nombre\_venv**

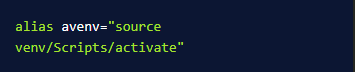
Usualmente el nombre del ambiente virtual es **venv**.

## **Activación del ambiente virtual**

* **Windows**: .\venv\Scripts\activate
* **Unix o MacOS**: source venv/bin/activate
* **Desactivar el ambiente virtual**: deactivate
* **Crear un alias en linux/mac**: alias nombre-alias="comando"

## **Crear un alias en git bash en windows**

* Ir a la carpeta **C:/Program Files/Git/etc/profile.d**
* Abrir el archivo **aliases.sh**.
* Pegar el alias con el siguiente formato:



* Guardar cambios y salir.

# **¿Qué es pip?**

Básicamente, **pip** es como el **npm** de JavaScript, y el archivo **requeriments.txt** es como el **package.json** de JavaScript.

# **Instalación de dependencias con pip**

Básicamente, **pip** es como el **npm** de JavaScript, y el archivo **requeriments.txt** es como el **package.json** de JavaScript.

Es importante recordar que esto se debe correr con el entorno virtual activado (**avenv**), de esta manera todas las dependencias que instalemos se guardaran para este entorno virtual (de lo contrario se guardarían de manera global, que es justo lo que no queremos).

## **Importante**

Si estás manejando **git**, es bueno siempre ignorar la carpeta **venv**, esto porque realmente no nos importa subir todo eso al repositorio, puedes mirarlo como que **venv** es el **node\_modules** de JavaScript, a fin de cuentas, cualquier otro programador que trabaje con nuestro código creará su propio entorno virtual e instalará las dependencias que dejamos en nuestro **requeriments.txt**.

## **Comando >**

Y un dato curioso es que, el operador **>** en la terminal es algo especial de UNIX, ya que este operador lo que hace **es redirigir la salida de cualquier comando hacia donde lo mandes,** por defecto la salida es en la terminal, pero al usar **>** le dijimos a la terminal que, en lugar de que la salida sea en la terminal, que se redirija al archivo **requeriments.txt**

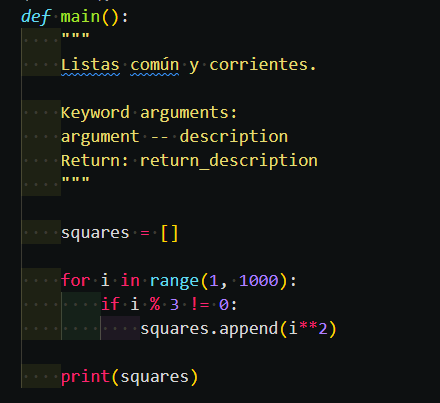
## **Instaladores de paquetes en python**

* pip
* pyenv

# **List comprehensions**

Básicamente son una forma de crear listas de una manera elegante simplificando el código al máximo.

Lista clásica



## **List comprehensions**



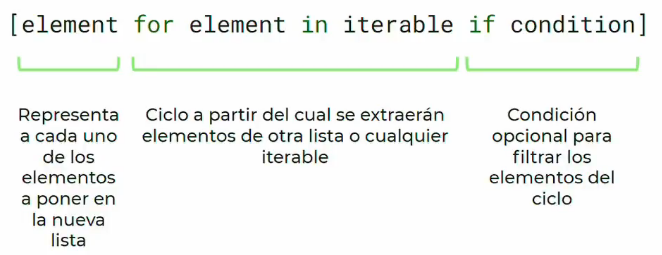
## **Estructura de una list comprehensions**



Esto se lee así, **elemento para cada elemento en el iterable, sólo sí la condición**.

## **Qué significa cada sección**

Un list comprehensions es una estructura de python que e utiliza para crear nuevas listas y su estructura es:



## **¿Como se lee un list comprehensions?**

Sí tomamos como ejemplo la imagen anterior de una estructura de un list comprehensions, se leería de la siguiente forma:

**Para cada elemento en el iterable, voy a guardar ese elemento sólo sí se cumple la condición**.

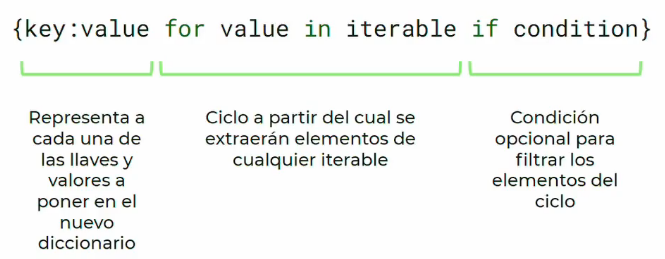
# **Dictionary comprehensions**

Básicamente son los mismo que las list comprehensions, es decir, son una forma de crear diccionarios de una manera elegante simplificando el código al máximo.

## **Estructura de un dictionary comprehensions**



## **Qué significa cada sección**



## **¿Como se lee un dictionary comprehensions?**

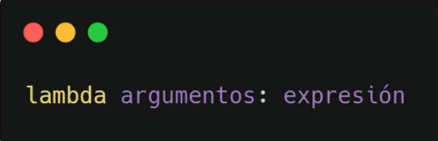
Sí tomamos como ejemplo la imagen anterior de una estructura de un dictionary comprehensions, se leería de la siguiente forma:

**Para cada elemento de un iterable colocamos una llave y un valor solamente si se cumple la condición**.

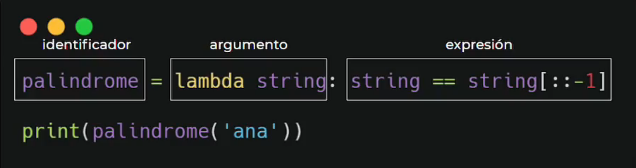
# **Funciones anónimas: lambda**

**Son funciones de una lista cuyo identificador es la variable** y **retornan** un objeto de tipo **funcion**. No necesitan ser definidas como una función normal ni que se le pase el parámetro entre paréntesis. Retorna el resultado por defecto, por lo que se puede omitir escribir **return**.

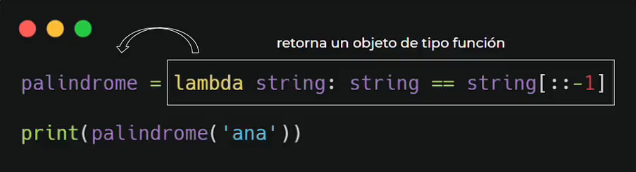
**variable = lambda parámetro: funcion de una línea**



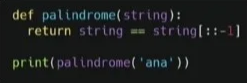
Saber si la palabra es un palíndromo o no.



Retorna un objeto de tipo **funcion**



Sin lambda.



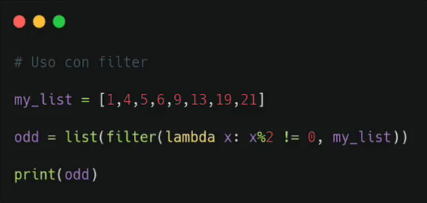
En python las funciones lambda (funciones anónimas en otros lenguajes) **solo se puede tener una expresión**.

# **High order functions: filter, map y reduce**

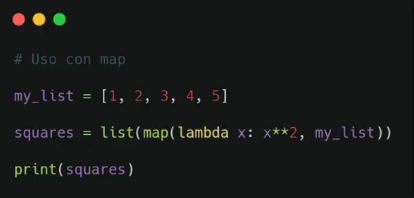
funciones de orden superior en español, **son funciones que reciben como parámetro a otra función**.

Funciones de orden superior de importancia:

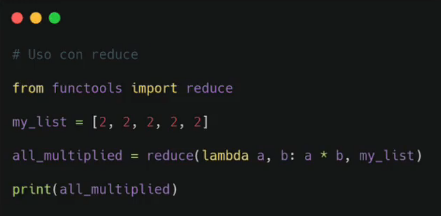
* **filter**: Recibe una función filtro (anónima) y un iterable (lista, tupla, etc) devolviéndonos un iterador. Objeto optimizado para recorrer elemento a elemento (iterar) por lo que no lo podemos imprimir de manera directa (para ello lo convertimos a una lista), su sintaxis es: **filter(<funcion filtro>, <iterable>)**



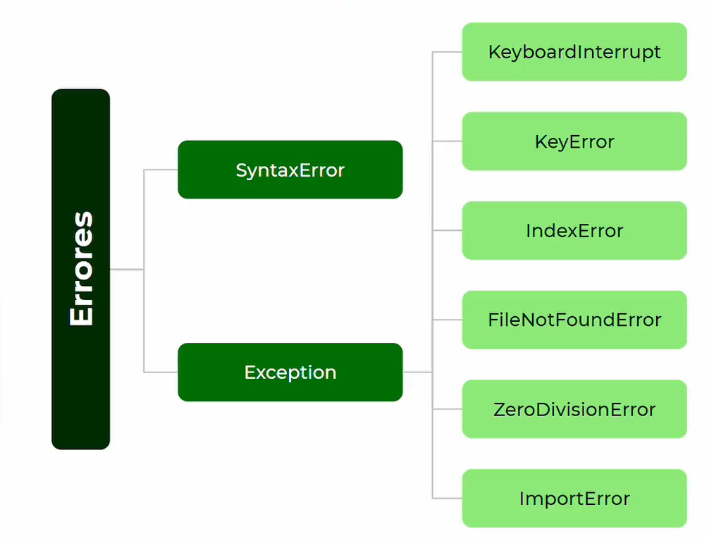
* **map**: Al igual que filter recibe una función anónima y un iterable como parámetros, pero en este caso map ejecuta la función sobre cada uno de los elementos del iterable, sintaxis: **map(<funcion>, <iterable>)**



* **reduce**: Tenemos que importar esta función desde **functools** para poder usarla, tiene los mismos argumentos que las anteriores funciones, reduce el iterable por medio de la función anónima, su sintaxis es: **reduce(<funcion reduccion>, <iterable>)**, la función de reducción necesita de dos parámetros, uno que almacena el resultado (o el primer valor del iterable) y otro que opera con el siguiente valor del iterable: **lambda a,b: <expresión>**



# **Los errores en el código**

Cuando python nos avisa que tenemos un error en el código nos avienta un mensaje que conocemos como traceback, puede ser debido a:

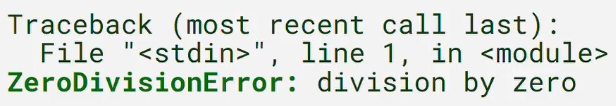
* **Errores de Sintaxis** (SyntaxError): Escribimos mal alguna palabra clave (typo), el programa no se ejecuta.
* **Excepciones** (Exception): Producen un colapso o interrupción de la lógica del programa en alguna línea en específico por ejemplo (todas las líneas anteriores se ejecutan), pueden ser de varios tipos, generalmente aparecen cuando no existe un componente clave en la ejecución o hay alguna imposibilidad lógica (matemática) para efectuar la instrucción, también pueden generarse dentro del código o fuera de él (elevar una excepción)

## **Lectura de un traceback**

* La manera correcta de leer un traceback es iniciar por el final, en el caso de un error de sintaxis nos indicará en qué línea se encuentra dicho error.
* En el caso de excepciones la última línea nos indicará el tipo de excepción que se generó (generalmente son auto explicativas, pero si no entiendes que paso puedes buscar este error)
* La penúltima línea nos indicará dende se encuentra el error (archivo y línea)
* La antepenúltima línea nos muestra “**most recent call last**” lo que significa que la llamada más reciente es la última (el programa se cerró después de esa llamada, se generó un error.)

## **Elevar una excepción**

* Cuando tenemos una excepción en python lo que sucede es que se crea un objeto de tipo exception que se va moviendo a través de los bloques de código hasta llegar al bloque principal si es que no se maneja dicha excepción en algún bloque intermedio el programa se interrumpe y genera el **traceback**.



# **Debugging**

Depuración en español, es una herramienta que traen varios editores de código con el objetivo de solucionar nuestros errores de lógica. Revisemos la herramienta debugging de VSCode.

En este entorno podemos acceder a funcionalidades como:

* **pause**: Permite pausar la ejecución del programa
* **step over**: Permite avanzar un solo paso en el programa
* **step in**: Ingresamos a un bloque secundario del programa (funciones)
* **step out**: Salimos del bloque secundario
* **restart**: Reinicia el programa
* **stop**: Detiene el programa

Además, podemos generar breakpoints, que son puntos en los que el programa se detendrá para ayudarnos a depurar el código

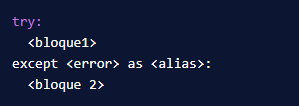
**Nota**:

Existen herramientas de debugging propias de python como el módulo **pdb** o los **breakpoints** (a partir de python 3.7)

# **Manejo de Excepciones**

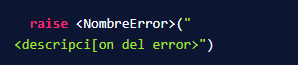
* **try except**: Anidamos nuestro programa en dos bloques de código, el primero es el programa per se (el que se ejecuta normalmente, sin errores) y el segundo representa las instrucciones a seguir en caso de error.

Su sintaxis es:



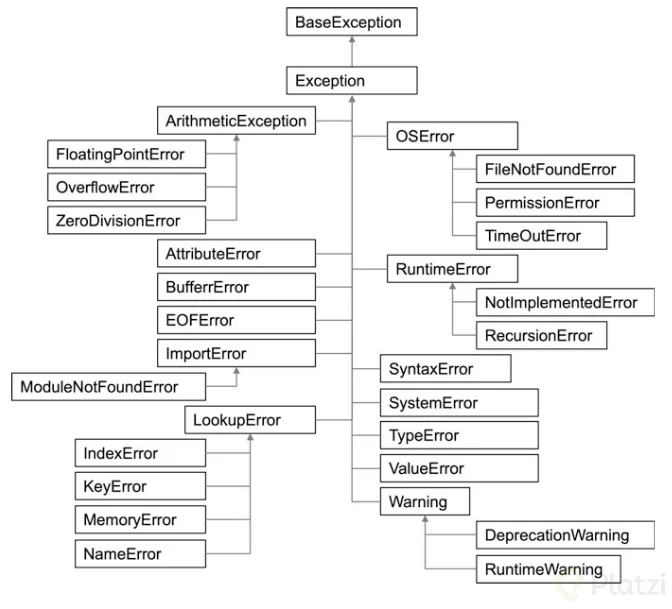
* <**error**> es un parámetro opcional, permite capturar sólo el tipo de error indicado, si no se coloca captura todos los errores posibles (es buena práctica capturar cada tipo de error por separado)
* **as <alias>** nos permite crear un alias al error, para trabajar con él.
* **raise**: Esta instrucción nos permite generar errores, es decir crear nuestros propios errores cuando detectemos que nuestro programa no actúa como debería con cierto tipo de datos

Su sintaxis es:



* **finally**: Es un bloque de código que se ejecuta exista un error o no (un tercer bloque después de try except), no es muy usual pero puede darse para cerrar archivos, conexiones a BBDD o liberar recursos.

## **Jerarquía de las excepciones y errores más comunes**

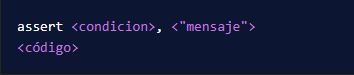


Más detalles de las jerarquías [aquí](https://docs.python.org/3/library/exceptions.html#exception-hierarchy).

# **Assert statements**

* Es una manera poco usual de manejar los errores en python.
* Evalúa una condicional, si esta se cumple continuamos con el flujo normal del python, si no se cumple eleva un error del tipo **AssertionError** y nos muestra un mensaje.

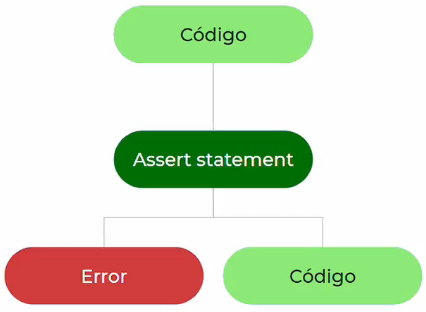
Su sintaxis es:



Se lee de la siguiente manera: **Afirmo qué, esta condición es verdadera sino imprimir mensaje de error**.

## **Diferencia importante entre usar ‘assert’ o ‘try-except’**

* **assert**: Además de imprimirse el mensaje de error que escribimos, se muestra también todo el **traceback**.
* **try-except**: Solo se muestra el mensaje de error que escribimos en el código o el mensaje de la excepción propiamente dicha.



# **¿Cómo trabajar con archivos?**

## **Modos de Apertura**

* **r**:Solo lectura.
* **r+**: Lectura y escritura.
* **w**: Solo escritura. Sobre escribe el archivo si existe. Crea el archivo si no existe.
* **w+**: Escritura y lectura. Sobre escribe el archivo si existe. Crea el archivo si no existe.
* **a**: Añadido (agregar contenido). Crea el archivo si éste no existe.
* **a+**: Añadido (agregar contenido) y lectura. Crea el archivo si éste no existe.

Existen varias extensiones de archivos con los que podemos interactuar con python (js, csv, py, css, json, xml)

Para abrir un archivo seguimos la siguiente estructura.

