

Python / Guía 7

Código ordenado, limpio y reutilizable

## 

## 

## OBJETIVOS DE LA GUÍA

En esta guía aprenderemos a:

* Crear módulos
* Crear paquetes
* Anotar tipos de datos
* Manejar errores
* Hacer pruebas unitarias
* Tener buenas prácticas de programación Python

para reutilizar código

# 

# Modularidad

La modularidad es la característica de un sistema, por la cual el mismo sistema o programa es entendido como la unión de varias partes que interactúan entre sí, realizando, cada una de estas partes, una tarea para cumplir su propio objetivo, en consecución de una finalidad mayor, que es el sistema.

De forma análoga a un organismo vivo, los módulos son a los órganos, como el sistema es al ser vivo.

## ¿Qué es un módulo?

Los módulos son archivos Python con extensión .py o .pyc (Python compilado), que posee su propio espacio de nombres y que puede contener variables, funciones e incluso otros módulos, que sirven para organizar y reutilizar el código en diferentes aplicaciones.

Los módulos son importados desde otros módulos usando la instrucción import seguido del nombre del módulo.

Si quieres la ayuda del módulo completo, usa la instrucción help() y el nombre del módulo dentro de los paréntesis.

Si quieres conocer la lista de funcionalidades de un módulo, usa la instrucción dir() y el nombre del módulo.

También puedes ver la ayuda de una funcionalidad invocando la instrucción help() el nombre del módulo, un punto, y el nombre de la funcionalidad.

**¡MANOS A LA OBRA!**

**Ejercicio 1**

Abre el intérprete de Python, y solamente usando el intérprete, importa el módulo random y debate con tus compañeros:

* ¿Qué hace el módulo random?
* ¿Cuáles son las funcionalidades de random ?
* ¿Qué hace la funcionalidad random.randint ?
* Utiliza la funcionalidad randint, al menos 4 veces, según te indique su ayuda

| Lista completa de módulos incorporados en la librería estándar de Python: <https://docs.python.org/es/3/library/> |
| --- |

## 

## Crea tus propios módulos

**¡MANOS A LA OBRA!**

**Ejercicio 2**

1. Crea una carpeta llamada 'proyecto'
2. Crea un archivo llamado 'matematica\_1.py' dentro de la carpeta 'proyecto'. Copia el siguiente código:

import math  
PI = math.pi  
def sumar(a, b):  
 print("El resultado de la suma es:", a + b)   
def restar(a, b):  
 print("El resultado de la resta es:", a - b)

| Luego de las líneas de importación (import) deja 2 líneas vacías, para mayor legibilidad (ver convenciones PEP8) |
| --- |

## 

1. Crea un archivo llamado 'matematica\_2.py' dentro de la carpeta 'proyecto'. Copia el siguiente código:

def multiplicar(a, b):  
 print("El resultado de la multiplicación es:", a \* b)  
def dividir (a, b):  
 print("El resultado de la división es:", a / b)

| Luego de cada función, deja 2 líneas vacías, para mayor legibilidad (ver convenciones PEP8) |
| --- |

1. Crea otro archivo llamado 'prueba.py' dentro de la carpeta 'proyecto'. Copia el siguiente código:

import matematica\_1  
 import matematica\_2  
 sumar(3, 3)  
 restar(3, 1)  
 multiplicar(3, 3)  
 dividir(3, 2)  
 print(PI)

1. Ejecuta el programa. ¿Qué es lo que pasa?
2. Cambia el código a la siguiente forma:

import matematica\_1  
 import matematica\_2  
 matematica\_1.sumar(3, 3)  
 matematica\_1.restar(3, 1)   
 matematica\_2.multiplicar(3, 3)   
 matematica\_2.dividir(3, 2)   
 print(matematica\_1.PI)

Esta forma es engorrosa, pues habría que escribir el nombre del módulo cada vez que queramos usar sus funcionalidades.

1. Prueba cambiar nuevamente el código de 'prueba.py' y corrige el error:

from matematica\_1 import sumar, restar, PI  
 from matematica\_2 import multiplicar, dividir  
 sumar(3, 3)  
 restar(3, 1)  
 multiplicar(3, 3)  
 dividir(3, 2)  
 print(PI)

1. Cambia nuevamente el código de 'prueba.py' e intenta ejecutar:

from matematica\_1 import \*  
 from matematica\_2 import \*  
 sumar(3, 3)  
 restar(3, 1)  
 multiplicar(3, 3)  
 dividir(3, 2)  
 print(PI)

| "Explícito es mejor que implícito". Es una buena práctica importar de forma explícita, sin hacer uso del comodín, pues podría traer conflictos con los nombres de funciones y variables que vamos a crear en el módulo. |
| --- |

1. Hay ocasiones en las que interesa, por colisión de otros nombres o para mejorar la legibilidad, usar un nombre diferente del módulo (u objeto) que estamos importando. Python nos ofrece esta posibilidad a través de la sentencia 'as'. Cambia nuevamente el código de 'prueba.py':

import matematica\_1 as m1  
 import matematica\_2 as m2  
 m1.sumar(3, 3)  
 m1.restar(3, 1)  
 m2.multiplicar(3, 3)  
 m2.dividir(3, 2)  
 print(m1.PI)

## ¿Qué es un paquete?

Un paquete es una carpeta o directorio donde se almacenan módulos relacionados entre sí. Sirven para organizar y reutilizar módulos. Tan solo debes tener un archivo llamado \_\_init\_\_.py, (doble guion bajo) dentro de una carpeta. El archivo puede estar vacío, o no.

Estructura de un paquete:

matematica/  
 |-- \_\_init\_\_.py  
 |-- matematica\_1.py  
 |-- matematica\_2.py

## 

## Crea tu propio paquete

**¡MANOS A LA OBRA!**

**Ejercicio 3**

Crearás un paquete y verás las formas posibles de usarlo. Es necesario que no te saltees partes del ejercicio. Al final verás cuál es el más legible y por qué.

1. Crea una carpeta llamada 'matematica' y mueve los archivos 'matematica\_1.py' y 'matematica\_2.py' dentro de la carpeta 'matematica'
2. Modifica 'prueba.py' y trata de ejecutarlo:

import matematica.matematica\_1 as m1  
 import matematica.matematica\_1 as m2  
 m1.sumar(3, 3)  
 m1.restar(3, 1)  
 m2.multiplicar(3, 3)  
 m2.dividir(3, 2)  
 print(m1.PI)

| Deja dos líneas de espacio después de las importaciones para mayor legibilidad (PEP8) |
| --- |

1. Crea un archivo '\_\_init\_\_.py' dentro de la carpeta 'matematica' y agrega el siguiente código:

from . import matematica\_1 as m1  
 from . import matematica\_2 as m2

| El punto indica "el mismo paquete" . Se llama "importación relativa", en contraposición a la "importación absoluta", cuando no se usa el punto. Cuando se usan dos puntos, se hace referencia a una carpeta que está en un nivel superior a la carpeta actual. |
| --- |

1. Modifica 'prueba.py' y ejecútalo:

import matematica as mat  
 mat.m1.sumar(3, 3)  
 mat.m1.restar(3, 1)  
 mat.m2.multiplicar(3, 3)  
 mat.m2.dividir(3, 2)  
 print(mat.m1.PI)

1. Modifica 'prueba.py' y ejecútalo:

from matematica import m1, m2  
 m1.sumar(3, 3)  
 m1.restar(3, 1)  
 m2.multiplicar(3, 3)  
 m2.dividir(3, 2)  
 print(m1.PI)

1. Modifica el archivo '\_\_init\_\_.py' con el siguiente código:

from .matematica\_1 import sumar, restar, PI  
 from .matematica\_2 import multiplicar, dividir

1. Modifica 'prueba.py' y ejecútalo:

import matematica  
matematica.sumar(3, 3)  
matematica.restar(3, 1)  
matematica.multiplicar(3, 3)  
matematica.dividir(3, 2)  
print(matematica.PI)

| Crear un paquete de esta forma te da la posibilidad de usar las funcionalidades que están repartidas en dos módulos, sin necesidad de invocar los módulos. |
| --- |

| Ten cuidado cuando crees módulos: no denomines a los módulos igual que a las funciones, o puedes tener problemas. |
| --- |

## If \_\_name\_\_ == "\_\_main\_\_"

Cuando el intérprete de Python lee un archivo de Python, primero establece algunas variables especiales. Luego ejecuta el código desde el archivo. Una de esas variables se llama \_\_name\_\_. Verás cómo usar if \_\_name\_\_ == "\_\_main\_\_", y por qué es importante.

### 

**¡MANOS A LA OBRA!**

**Ejercicio 4**

1. Crea otra carpeta cualquiera y crea 2 archivos en ella: archivo\_1.py y archivo\_2.py
2. En archivo\_1.py agrega:

print(f"Archivo 1: \_\_name\_\_ se establece en: {\_\_name\_\_}")

1. En archivo\_2.py agrega:

print(f"Archivo 2: \_\_name\_\_ se establece en: {\_\_name\_\_}")

1. Ejecuta cada archivo.
2. Ahora, en archivo\_1.py agrega la siguiente línea de código y ejecútalo:

import archivo\_2

La variable \_\_name\_\_ para el módulo que se ejecuta será siempre \_\_main\_\_. Pero la variable \_\_name\_\_ para todos los demás módulos que se importan se establecerá en el nombre de su módulo.

1. Ahora estás agregando una nueva función a archivo\_2.py. Agrega las siguientes líneas:

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

1. Quieres probar si la función anda bien, y agregas:

print(suma(2, 2))

1. Ejecuta archivo\_1.py. Hay un problema: no deberías ver el resultado de la prueba print(suma(2, 2). Puedes resolver esto modificando archivo\_2.py, que quedará de la siguiente manera:

print(f"Archivo 2: \_\_name\_\_ se establece en: {\_\_name\_\_}")  
 def suma(a, b):  
 return a + b  
 if \_\_name\_\_ == "\_\_main\_\_":  
 print(suma(2, 2))

1. Ejecuta archivo\_2.py.
2. Ejecuta archivo\_1.py. Ya no se ve la suma.

**Conclusión:** Podemos usar un bloque if \_\_name\_\_ == "\_\_name\_\_" para permitir o evitar que se ejecuten partes del código cuando sean importados los módulos.

## Revisemos lo aprendido hasta aquí

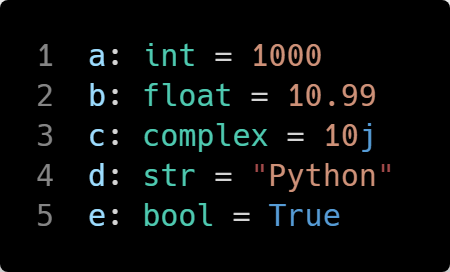
* Conocer el listado de módulos de la librería estándar de Python
* Crear módulos
* Crear paquetes
* Saber utilizar el bloque if \_\_name\_\_ == "\_\_name\_\_"
* Conocer buenas prácticas para manejar paquetes

# Anotaciones de tipo

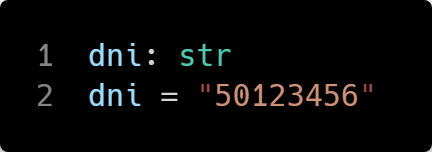
Python es de tipado dinámico, y esto hace que codificar sea muy rápido. Tipado dinámico significa que las variables pueden guardar un tipo de datos y luego pueden guardar otro. A veces, puede generar problemas en el momento de ejecución. Estos posibles errores se manejan utilizando buenas prácticas al momento de codificar. Además, obtendrás ayuda de parte del editor de código.

## En tipos de datos primitivos

Para la variable, basta escribir el nombre de la variable, dos puntos, y el tipo.

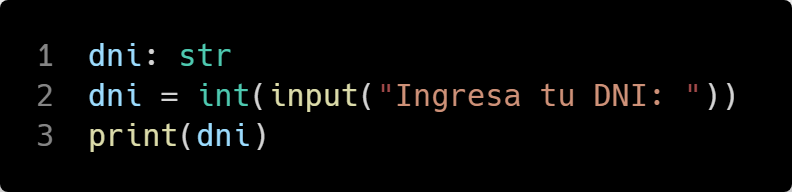


También es posible definir el tipo sin necesidad de inicializar la variable:



Si quieres convertir esta variable 'dni' en un entero, recibirás un aviso de Visual Studio Code:

**Ejecuta el siguiente código:**



Verás en la línea 2, que desde "int" hasta el final, el editor subraya con color rojo, y, si posicionas el cursor del mouse arriba, verás:

*<Expression of type "int" cannot be assigned to declared type "str" "int" is incompatible with "str".>*

| Si no ves este error, en Visual Studio Code presiona F1 y se abrirá una ventana pequeña donde escribirás "settings json". Selecciona "abrir configuración del usuario" . Se abrirá el archivo de configuración de VSCode. Agrega la siguiente línea dentro de las primeras llaves:  "python.analysis.typeCheckingMode": "basic", |
| --- |

Sin embargo, el código se ejecutará normalmente. ‎Python no tiene una herramienta oficial de verificación de tipos estáticos. Por el momento, la herramienta de terceros más popular es MyPy. Dado que MyPy es un paquete de terceros, debes instalarlo utilizando el siguiente comando **en la terminal:**‎

pip install mypy

| Si 'pip' no llega a funcionar, prueba lo siguiente:  python -m pip install mypy |
| --- |

Una vez instalado puedes ejecutar MyPy desde la terminal o consola:

mypy nombre\_del\_archivo.py

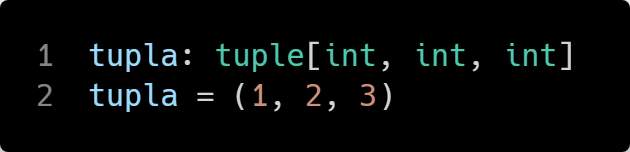
Trata de ejecutar MyPy en tu proyecto cada vez que crees un módulo.

## En colecciones

Si intentas agregar un elemento a la lista que no sea float, verás que el editor te marca un error:

lista.append("soy una cadena")

Esto sucede porque ha entendido que el tipo de datos que guarda la lista es un tipo float, de manera implícita. De forma explícita hazlo de la siguiente manera:

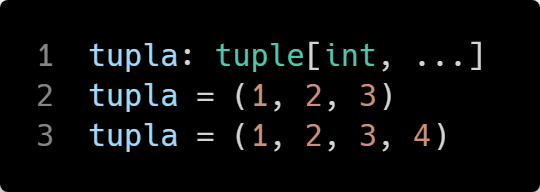


Intenta asignar a la variable un valor más, y verás un error:

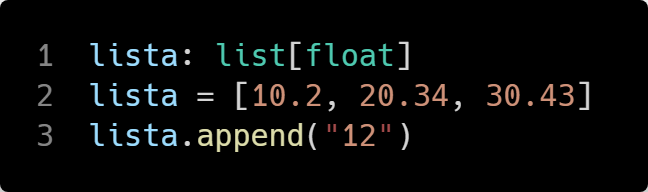
tupla = (1, 2, 3, 4)

Si intentas crear en una **tupla** de tamaño variable, utiliza el objeto 'Ellipsis' que consiste en tres puntos '...'

| Intenta ejecutar: print(type(...)) |
| --- |



En cambio, en las **listas,** solo debes anotar el tipo de datos dentro de los corchetes:

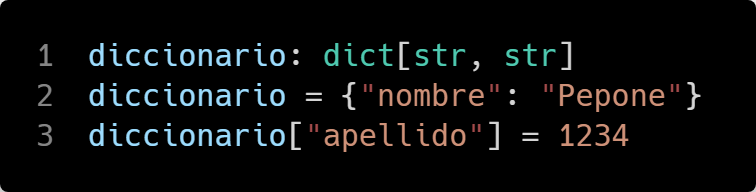


| Intenta agregar a la lista, en vez de "12", el entero 12. El editor entiende el 12 como un dato válido. Intenta validar con MyPy. |
| --- |

En los conjuntos sucede lo mismo:

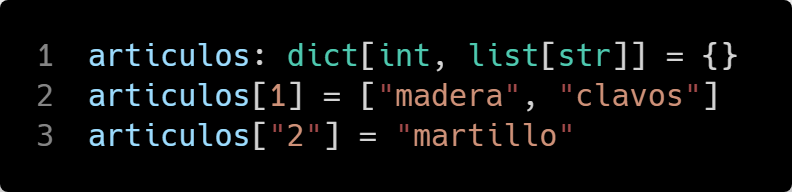


En los **diccionarios,** debes especificar el tipo de dato de la clave y el valor:

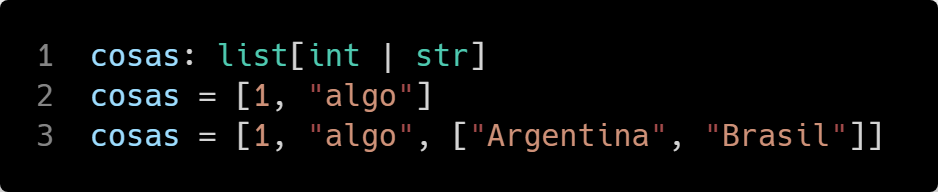


En **expresiones anidadas:**

**Repara el siguiente código y agrega "martillo" a la lista:**



Si quieres definir una lista de **varios tipos de datos,** no puedes hacer el mismo procedimiento que se realiza con las tuplas, debes usar el operador unión **'|':**

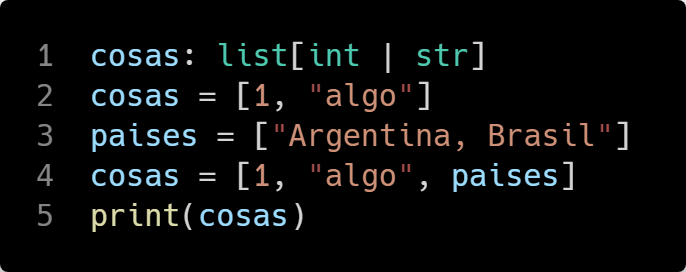


### 

**¡MANOS A LA OBRA!**

**Ejercicio 5**

1º Modifica el siguiente código. Agrega los países en la lista "cosas" en forma de cadenas, no en forma de lista.



## En funciones

Las anotaciones de tipo en las funciones siguen la siguiente estructura:

def nombre\_funcion(parametro: <tipo>, ...) -> <tipo>:  
 return <objeto>

Puedes ver que se utiliza '->' para indicar qué tipo de datos devuelve 'return'.

### 

**¡MANOS A LA OBRA!**

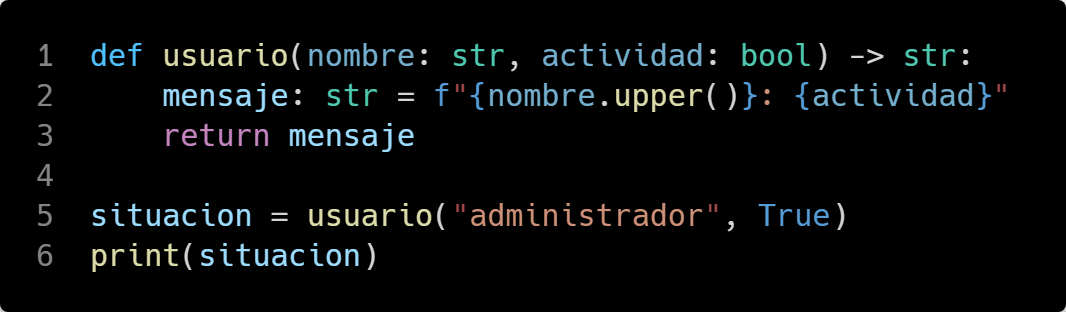
**Ejercicio 6**

1º Escribe el siguiente código:

def usuario(nombre, actividad):  
 mensaje = f"{nombre.upper()}: {actividad}"  
 return mensaje

2º Borra el fragmento de código que dice 'upper()'. Presiona 'control + espacio' luego del punto que está al lado de 'nombre'. ¿Ves alguna ayuda del editor de código? ¿Por qué?

3º Agrega las anotaciones de tipo según el siguiente código:

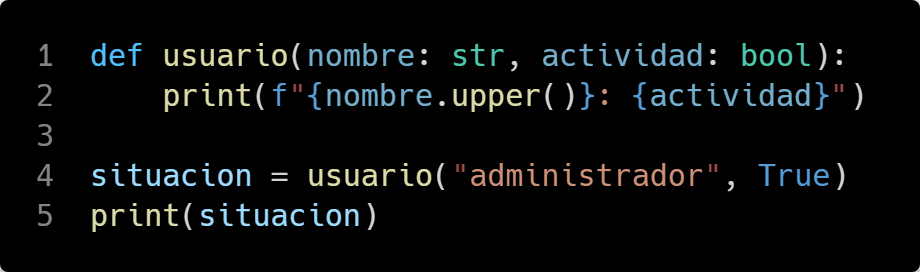


4º Ahora borra la parte donde dice 'upper()'. Presiona 'control + espacio' luego del punto que está al lado de 'nombre'. Ahora sí verás la ayuda del editor, pues reconoce que el tipo de datos de 'nombre' es una cadena. Mira cómo cambia de color 'upper()' respecto al ejemplo anterior. También puedes intentar invocar la función con distintos tipos de datos.

**Ejercicio 7**

1º Agrega la anotación de tipo de dato (que falta) para la devolución (return) de la función.

2º Corrige el siguiente código para que solamente imprima una línea, y no dos, como hace (sin tocar el cuerpo de la función)



3º Una forma de indicar que un tipo de datos es opcional, es definir la variable de la siguiente manera:

actividad: bool | None = None

Pero otra forma más limpia es importar el módulo incorporado typing. Escribe en la primera línea:

from typing import Optional

| El módulo typing se dedica exclusivamente a las anotaciones de tipo. |
| --- |

Y reemplaza el fragmento del parámetro 'actividad' con el siguiente:

actividad: Optional[bool] = None

4º Invoca a la función con un único y solo argumento.

| Puedes crear variables globales arriba o abajo de las funciones, pues lo primero que se ejecuta es el ámbito global. |
| --- |

## 

## Revisemos lo aprendido hasta aquí

* Entender la importancia de las anotaciones de tipo para crear código consistente
* Anotar tipos en tipos primitivos
* Anotar tipos en colecciones
* Anotar tipos en funciones
* Tener conocimiento del módulo MyPy, externo a Python
* Tener conocimiento del módulo Typing, integrado en Python

# Manejo de errores

## ¿Qué es una excepción?

Una excepción es un objeto Python que irrumpe el curso normal de la ejecución de un programa cuando hay un error. Una excepción contiene:

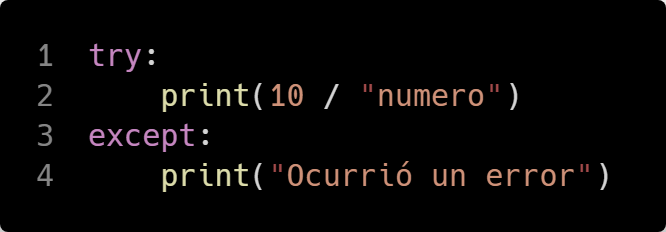
* ‎El tipo de error (nombre de excepción)‎
* ‎El estado del programa cuando se produjo el error‎.
* ‎Un mensaje de error que describe el evento de error.‎

‎Las excepciones son útiles para indicar diferentes tipos de posibles errores. Por ejemplo: ImportError, RuntimeError, NameError, TypeError, etc.

En Python, podemos lanzar una excepción en el bloque de prueba 'try' y atraparlo en el bloque excepción 'except'‎

try:  
 <bloque de prueba>  
 except:  
 <bloque de excepción>

**Ejecuta el siguiente código:**



| La instrucción 'pass' y los tres puntos '...' sirven para cerrar bloques cuando estás programando. Puede suceder que quieres probar un bloque try - except, una función, un bucle, un condicional, o cualquier bloque Python que encuentres, al que aún no quieres agregar código por alguna circunstancia. Es entonces que puedes usar esta instrucción para que el programa no muestre un error de sintaxis. Las instrucciones 'pass' o '...' no hacen "nada", pero sirven para cerrar bloques. Prueba, en el bloque except, escribir pass o ... y ejecuta el código. |
| --- |

No es una buena práctica usar un bloque except sin indicar el tipo de excepción que estamos gestionando, no solo porque puedan existir varias excepciones para capturar, también porque "lo explícito es mejor que lo implícito", según las convenciones de la comunidad Python.

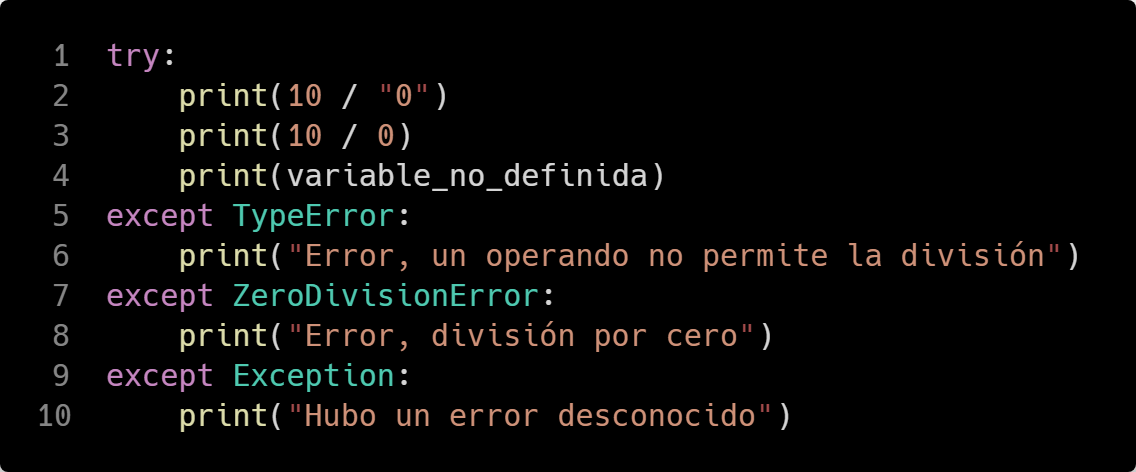
## Especificando excepciones

En el siguiente ejemplo vamos a mejorar el código anterior, capturando distintos tipos de excepciones:

* TypeError, si los operandos no permiten la división.
* ZeroDivisionError, si el denominador es cero.
* Exception, si ocurre error que no son los anteriores.

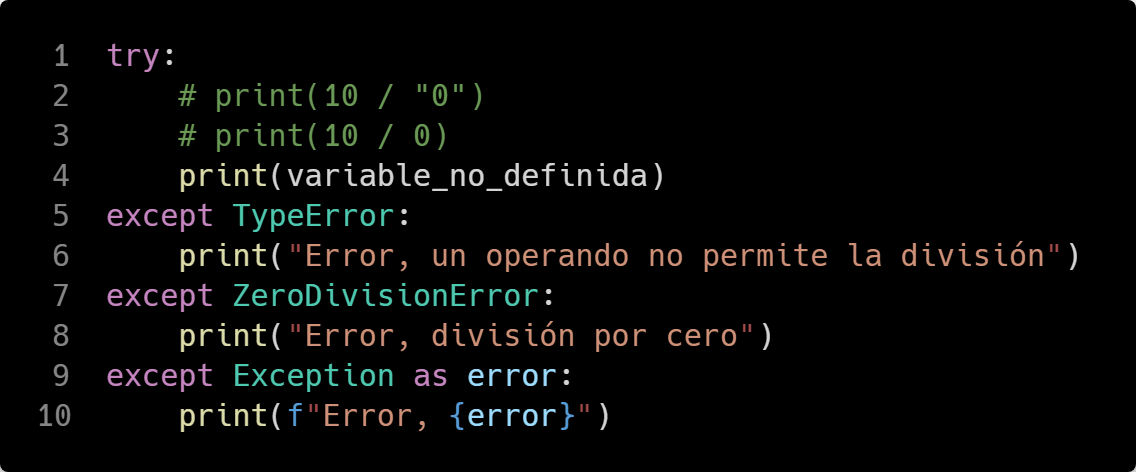
Observa su implementación, y comenta '#' los diferentes 'print' para ver qué ocurre:

**Ejecuta el siguiente código:**



Si quieres cubrir otro caso que llegara a capturar Exception , puedes ver el mensaje de la excepción creando un alias con 'as' que puede llamarse de cualquier forma, lo llamaremos 'error' simplemente:

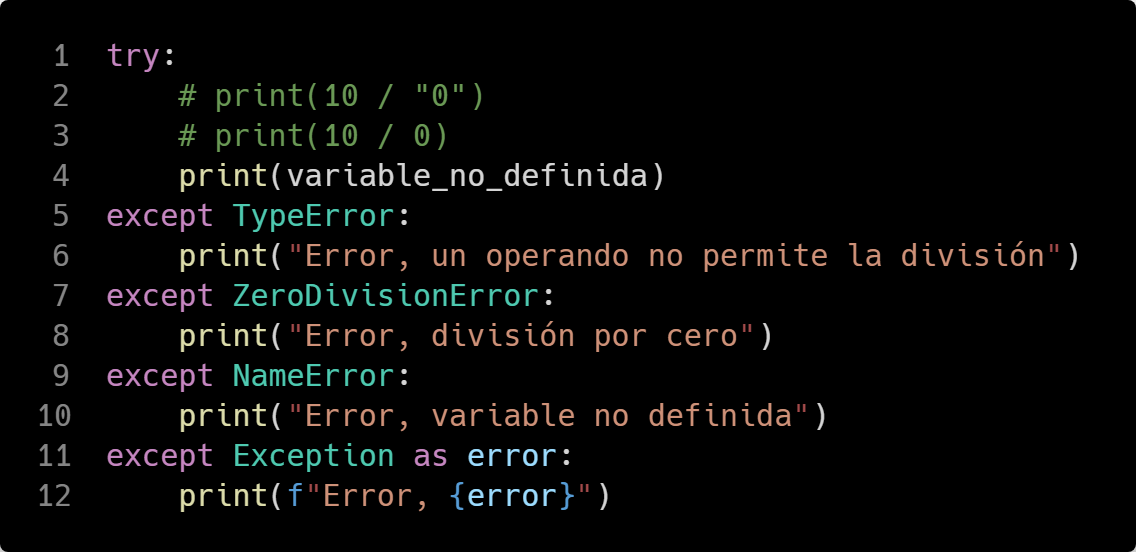
**Ejecuta el siguiente código:**



Utiliza type(error) para ver el nombre de la excepción, o también puedes probar con: type(error).\_\_name\_\_. Agrega:

print(f"Error, {type(error)}")

**Ejecuta el siguiente código:**



## Raise

Si quieres que el programa se detenga, por algún motivo, y necesitas un contexto, puedes levantar manualmente una excepción con la sentencia 'raise'. Puedes usarla sola, o acompañada del tipo de excepción que quieres levantar junto a un mensaje. Pruébala cambiando la línea 8 por:

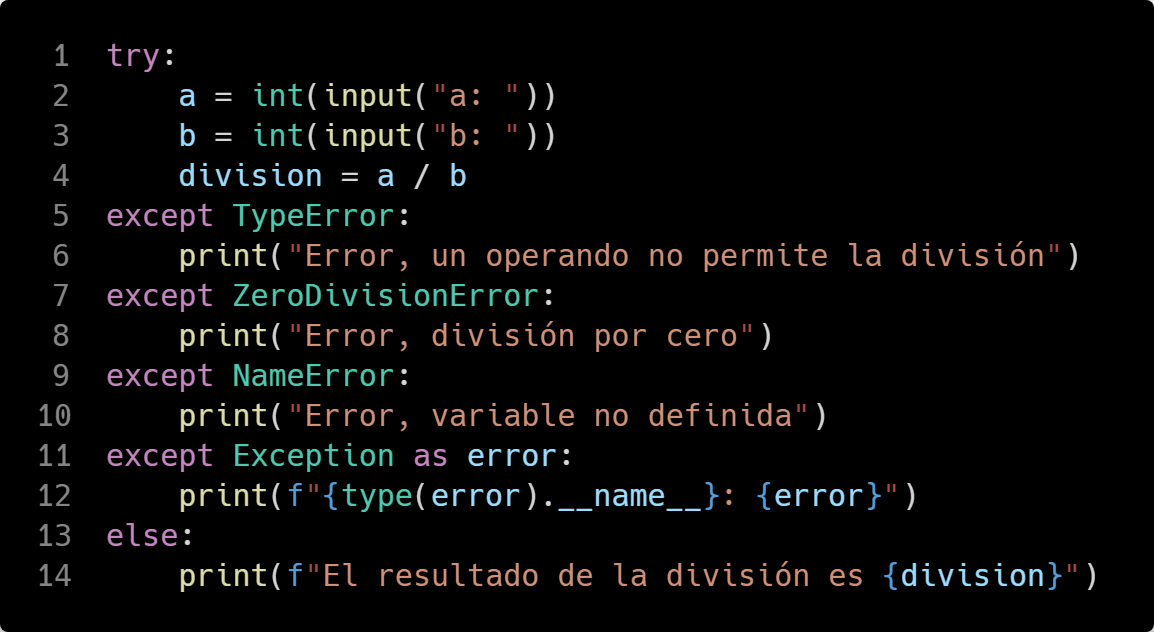
raise ZeroDivisionError("Error, división por cero")

## Else

Utiliza 'else' para ejecutar el bloque de código cuando no se lance una excepción.

try:  
 <bloque de prueba>  
 except:  
 <bloque de excepción>  
 else:  
 <bloque que no tuvo excepciones>

**Ejecuta el siguiente código:**



## 

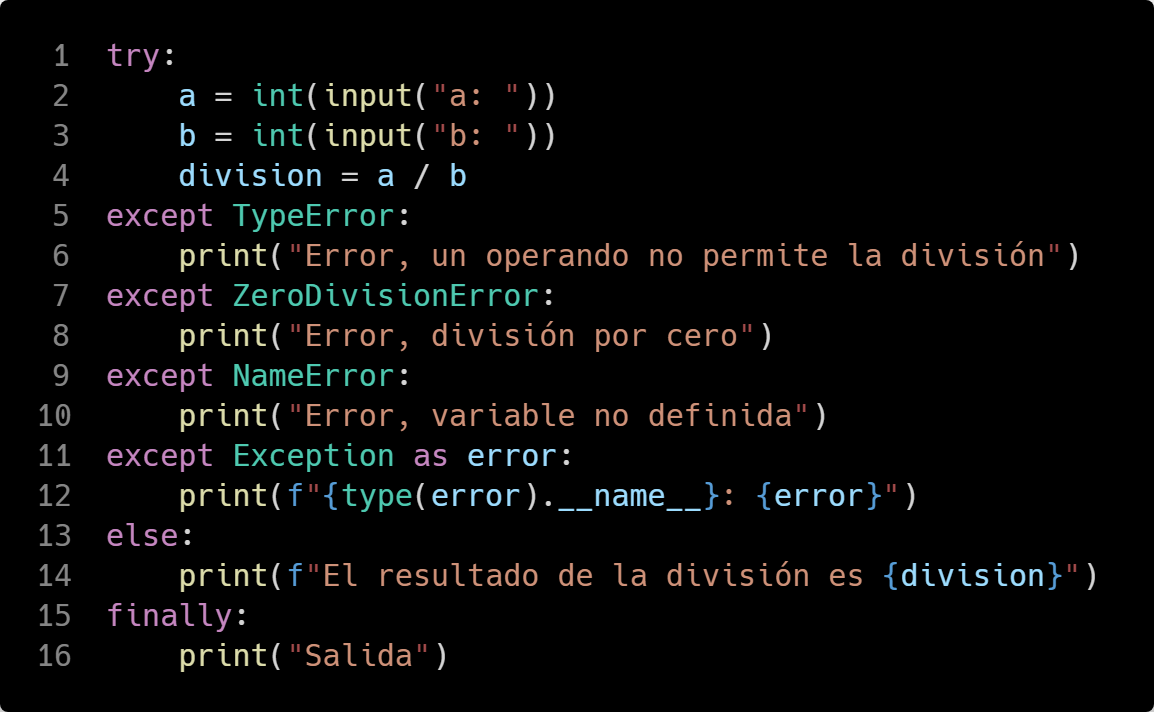
## Finally

Utiliza 'finally' para ejecutar un bloque de código siempre, incluso si se lanzó un error.

try:  
 <bloque de prueba>  
 except:  
 <bloque de excepción>  
 else:  
 <bloque que no tuvo excepciones>  
 finally:  
 <bloque que siempre se ejecuta>

Prueba el código siguiente, generando diferentes tipos de errores:

**Ejecuta el siguiente código:**



**¡MANOS A LA OBRA!**

Para los siguientes ejercicios, crea su correspondiente excepción para evitar que el programa se bloquee, y, además, explica con un mensaje al programador la causa del error:

**Ejercicio 8**

lista = [2, 4, 6, 8, 10]  
 lista[10]

**Ejercicio 9**

colores = {"blanco": "white", "verde": "green", "negro": "black"}  
colores["amarillo"]

**Ejercicio 10**

Crea una función que reciba como parámetro una lista y un elemento cualquiera (Any). La función debe agregar el "elemento" al final de la "lista". Si el elemento existe en la lista, debes lanzar un error de tipo ValueError y mostrar este mensaje en su lugar:

Error: No puedes agregar el elemento [elemento] porque ya existe en la lista.

## Revisemos lo aprendido hasta aquí

* Prever posibles errores del usuario o del programador
* Usar el bloque try - except para manejar excepciones
* Lanzar excepciones manualmente con raise

# Pruebas unitarias

## Pruebas de software

La prueba de software consiste en examinar su comportamiento para evaluar y verificar que sea coherente con las especificaciones. Los productos de software pueden tener miles de líneas de código y cientos de componentes que funcionan juntos. Si una sola línea no funciona correctamente, el error puede propagarse y causar otros errores. Entonces, para estar seguro de que un programa actúa como se supone que debe hacerlo, debe probarse.

Si eres propietario de una empresa, las pruebas de software son una actividad que debes considerar, ya que puede impactar en tu negocio. Por ejemplo, en mayo de 2022, Tesla retiró del mercado 130.000 automóviles debido a un problema en los sistemas de información y entretenimiento de los vehículos. Luego, este problema se solucionó con una actualización de software distribuida "por aire". Estas fallas costaron tiempo y dinero a la empresa, y también causaron problemas a los clientes, porque no pudieron usar sus autos por un tiempo. Efectivamente, probar el software cuesta dinero, pero también es cierto que las empresas pueden ahorrar en soporte técnico.

Se pueden enumerar cuatro niveles de prueba de software:

* Pruebas unitarias: prueban líneas de código específicas.
* Pruebas de integración: prueban la integración entre muchas unidades.
* Pruebas del sistema: prueban todo el sistema.
* Pruebas de aceptación: prueban el cumplimiento de los objetivos comerciales

## Pruebas unitarias

Este es el primer nivel de prueba, también llamado prueba de componentes . En esta parte, se prueban los componentes individuales del software. La unidad de software puede ser una función, o como veremos más luego, una clase.

### ¿Por qué deberías hacer pruebas unitarias?

La principal ventaja de las pruebas de software es que mejora la calidad del software. De acuerdo con el estándar ISO/IEC 9126-1 ISO 9126 , la calidad del software incluye estos factores: fiabilidad, funcionalidad, eficiencia, usabilidad, mantenibilidad, portabilidad.

Las pruebas unitarias se centran en verificar si un programa se está comportando correctamente o no, lo que significa verificar que el mapeo entre las entradas y las salidas se realice correctamente. Al ser una actividad de prueba de bajo nivel, la prueba unitaria:

* **Ayuda en la identificación temprana de errores,** para que no se propaguen a niveles más altos del sistema.
* **Simplifica la integración** de muchas unidades, al garantizar que todos los componentes funcionen bien individualmente, es más fácil resolver los problemas de integración.
* **Minimiza la regresión del código:** con una buena cantidad de casos de prueba, si algunas modificaciones al código fuente en el futuro causarán problemas, es más fácil localizar el problema.
* **Proporciona documentación:** al probar el mapeo correcto entre la entrada y la salida, las pruebas unitarias proporcionan documentación sobre cómo funciona el componente.

### ¿Qué partes de mi sistema debo probar?

Las pruebas exhaustivas son imposibles, por esta razón, no puedes probar todas las entradas y salidas posibles, pero debes priorizar sus pruebas en función de los riesgos involucrados. Un 90% de código probado es un buen número.

### Cualidades de una prueba unitaria

**Velocidad.** Las pruebas unitarias se ejecutan principalmente de forma automática, lo que significa que deben ser rápidas. Si se demoran más de 0,1 segundo, se considera que el código debería revisarse.

**Aislación.** Las pruebas unitarias son independientes por definición. Prueban la unidad de código individual y no dependen de nada externo (como un archivo o un recurso de red).

**Repetibilidad.** Las pruebas unitarias se ejecutan repetidamente y el resultado debe ser consistente a lo largo del tiempo.

**Confiabilidad.** Las pruebas unitarias fallarán solo si hay un error en el sistema bajo prueba. El entorno o el orden de ejecución de las pruebas no debería importar.

**Explicitación.** El nombre de la prueba debe proporcionar información relevante sobre la prueba en sí.

## Assert

Para introducirnos en las pruebas unitarias en Python, empezaremos a comprender la aserción o aseveración, es decir, la instrucción assert y su lanzamiento de error AssertionError.

Cuando queremos comprobar una expresión, podemos lanzar una excepción llamada AssertionError.

Ejecuta lo siguiente:

if 1 == 1:  
 raise AssertionError  
if 1 == 2:  
 raise AssertionError  
if int == type(3):  
 raise AssertionError  
if int == type(3.0):  
 raise AssertionError

Esta comprobación es más fácil hacerla con assert. Es decir, si el contenido existente dentro de assert es igual a False, se lanzará la excepción.

Ejecuta lo siguiente:

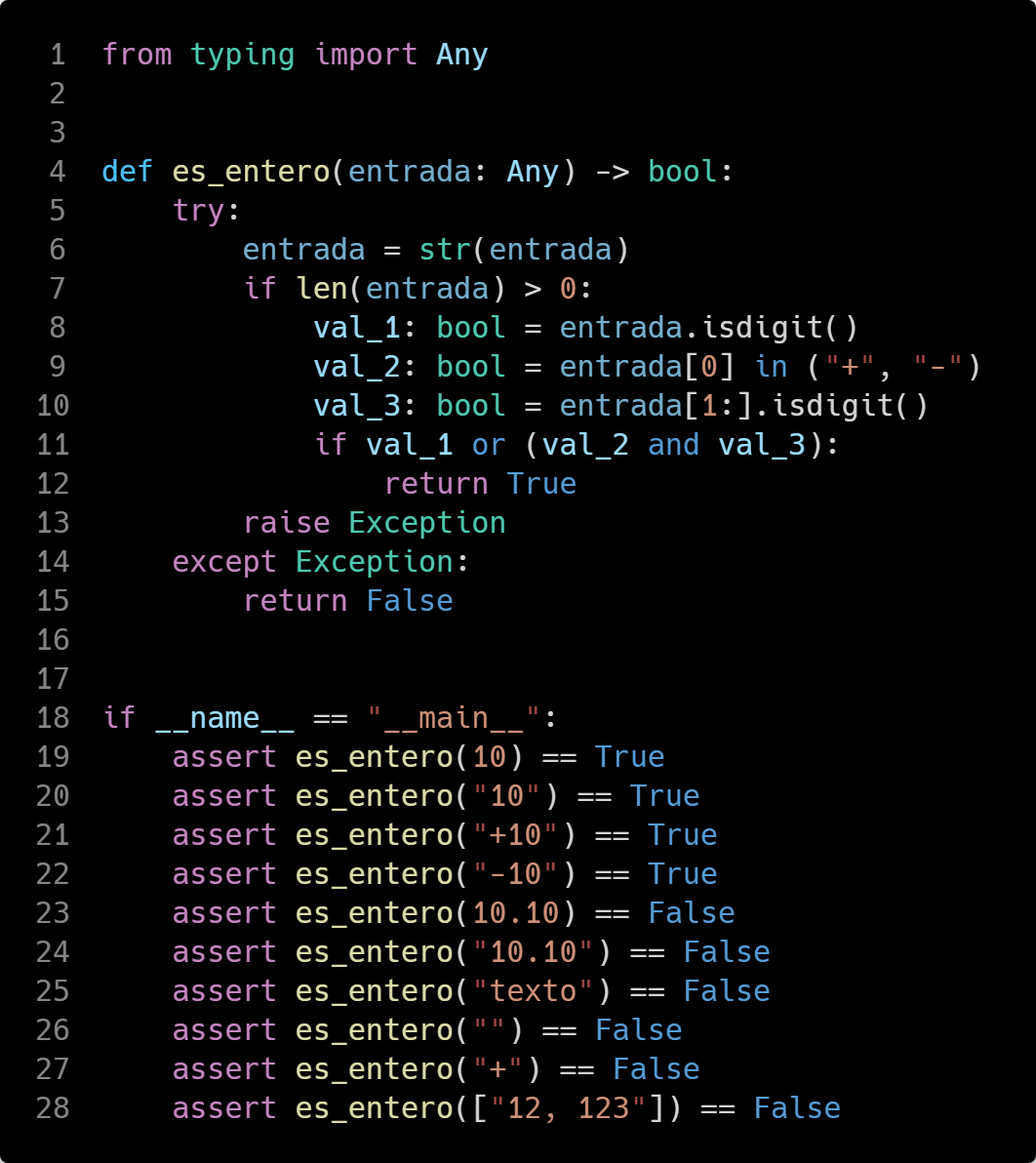
assert 1 == 1  
 assert 1 == 2  
 assert int == type(3)  
 assert int == type(3.0)

Retomando los principios de pruebas unitarias, las aserciones son para probar **errores del programador,** no errores del usuario. Las aserciones solo deben fallar mientras el programa está en desarrollo, y un usuario nunca debería ver un error de aserción en un programa terminado.

**¡MANOS A LA OBRA!**

**Ejercicio 11**

1º Escribe el siguiente código y ejecútalo. Trata de explicar por qué se escribe de esta manera, e intenta cambiar los valores:



## Pytest

La biblioteca estándar de Python incluye el marco de pruebas unitarias unittest. Este marco está inspirado en JUnit, que es un marco de pruebas unitarias en Java. A diferencia de unittest, Pytest es un módulo que no viene en la biblioteca estándar de Python, pero facilita aún más las pruebas unitarias, incluso las más complejas.

Pytest es una biblioteca externa, se debe instalar para poder usarse. En la consola o terminal, ejecuta:

pip install pytest

| Si 'pip' no llega a funcionar, prueba lo siguiente:  python -m pip install pytest |
| --- |

Si se instaló correctamente, debería funcionar lo siguiente, en la terminal:

pytest --version

**¡MANOS A LA OBRA!**

**Ejercicio 12**

1º Al código anterior que escribiste, el que contiene la función 'es\_entero', guárdalo con el nombre main.py.

2º Cambia la línea 18 (if \_\_name\_\_ == "\_\_main\_\_":) por la siguiente:

def test\_es\_entero():

3º En la terminal ejecuta el siguiente comando:

pytest main.py

4º Verás en la salida algo como lo siguiente:

=== test session starts ===  
 platform win32 -- Python 3 ...  
 rootdir: ...  
 collected 1 item  
 main.py . [100%]  
 === 1 passed in 0.05s ===

Esto significa que la prueba unitaria fue exitosa.

5º Intenta cambiar algún assert, en donde se espera True cambia a False y vuelve a ejecutar pytest main.py . Examina cómo Pytest ayuda a encontrar los errores.  
El directorio principal de las pruebas es el directorio o carpeta 'tests'. Puedes colocarlo en el nivel raíz del proyecto o junto con los módulos de código.

### Convenciones:

Existen convenciones de prueba en las que se basa Pytest. Al conocer estas reglas, puedes aprovechar la detección y ejecución automáticas de pruebas sin necesidad de ninguna configuración adicional.

* Utiliza el prefijo test en los archivos de prueba o el sufijo \_test. Test indica que el archivo contiene código de prueba.
* Crea una un paquete tests para colocar archivos de prueba y carpetas de pruebas anidadas.

Veamos cómo se ve proyecto Python con una carpeta para pruebas unitarias:

.  
├── paquete  
│ ├── \_\_init\_\_.py  
│ └── main.py  
└── tests  
 └── \_\_init\_\_.py  
 └── test\_main.py

El directorio o carpeta 'tests' se encuentra en la raíz del proyecto con un único archivo de prueba. En este caso, el archivo de prueba se denomina test\_main.py:

| Evita usar test (en singular) como nombre de directorio. El nombre test es un módulo de Python, por lo que la creación de un directorio con el mismo nombre lo invalidaría. Use siempre el plural tests en su lugar. |
| --- |

**Ejercicio 12**

1º En una carpeta vacía, crea dos carpetas: paquete y tests.

2º Convierte cada carpeta en un paquete Python, para eso crea un archivo \_\_init\_\_.py vacío en cada una de ellas.

3º Mueve el archivo 'main.py' que contiene la función 'es\_entero' dentro del módulo paquete.

4º Crea un archivo test\_main.py dentro del módulo tests.

5º En el archivo main.py deja solo la función es\_entero. Y a la función test\_main(), córtala y pégala en el archivo test.main.py. No te olvides de importar bien del módulo main la función es\_entero.

6º Ejecuta el siguiente comando:

pytest

7º Ahora prueba con el siguiente:

pytest -v

## Revisemos lo aprendido hasta aquí

* Conocer la importancia de las pruebas unitarias
* Saber cómo utilizar la instrucción assert
* Hacer pruebas unitarias con Pytest

# 

# Entornos virtuales

"Un entorno virtual es un entorno Python en el que el intérprete Python, las bibliotecas y los scripts instalados en él **están aislados de los instalados en otros entornos virtuales,** y (por defecto) cualquier biblioteca instalada en un «sistema» Python, es decir, uno que esté instalado como parte de tu sistema operativo." <https://docs.python.org/es/3.10/library/venv.html>

En otras palabras: Los entornos virtuales son carpetas de instalación que están aislados unos de otros. Este aislamiento te permite localizar la instalación de las dependencias (como las que hemos utilizado: MyPy y Pytest) de tu proyecto, sin obligarte a instalarlas en todo el sistema.

Imagina que tienes dos aplicaciones: Ferretería y Farmacia. Ambas usan el paquete ControlStock, pero requieren versiones diferentes. Si instalas ControlStock versión 1.2 para Ferretería, no podrás ejecutarla en Farmacia, porque está usando la versión 2.0. Aquí es donde los entornos virtuales son útiles:

* Puedes tener varios entornos, con varios conjuntos de paquetes, sin conflictos entre ellos. De esta manera, los requisitos de diferentes proyectos se pueden satisfacer al mismo tiempo.
* Puedes lanzar fácilmente tu proyecto con sus propios módulos dependientes.

| Hasta ahora hemos instalado las dependencias MyPy y PyTest en el "entorno global" de Python. No pasa nada. Puedes ver qué dependencias hay instaladas:  pip list  Y puedes desinstalarlas con el siguiente comando:  pip uninstall mypy |
| --- |

## ¿Cómo creo un entorno virtual?

Python trae consigo el módulo venv con el que puedes crear entornos virtuales de manera fácil. El comando es:

python -m venv nombre\_del\_entorno

**¡MANOS A LA OBRA!**

**Ejercicio 13**

1º Crea una carpeta llamada Negocio, será tu proyecto. Ejecuta en la terminal:

mkdir Negocio

2º Accede a la carpeta Negocio. Ejecuta en la terminal:

cd Negocio

3º Crea un entorno virtual de Python. Ejecuta en la terminal:

python -m venv .venv

| .venv o .env son nombres convencionales para los entornos virtuales, pero puedes usar otros. |
| --- |

4º Se creará una carpeta llamada .venv. Ahí adentro está el intérprete de Python, con sus librerías.

5º Abre Visual Studio Code desde la terminal. Ejecuta observando el punto después de code:

code .

6º Se abrirá una ventana de Visual Studio Code, con la carpeta Negocio abierta.

7º Presiona control + shift + P , verás que se despliega en la parte superior una ventana con el símbolo > y el cursor esperando a que escribas. Ahí escribe lo siguiente, seguido de la tecla enter:

python interpreter

8º Selecciona el entorno virtual (se verá algo parecido a lo siguiente):

Python 3.10... ('.venv': .venv)

9º Ahora vete al menú de Visual Studio Code, donde dice Terminal, y haz clic sobre Nueva Terminal. Una vez abierta la nueva terminal integrada de Visual Studio Code, verás que del lado izquierdo del cursor estará el nombre del entorno virtual (.venv). Entonces, ya puedes ejecutar:

pip list

10º Ahora, todo lo que instales con pip, se instalará solamente en este entorno virtual.

| Existen administradores de paquetes, que facilitan el uso de entornos virtuales. Son muy populares Conda, Pipenv y muchos más. Te recomiendo profundizar en el empleo de **venv**, y luego ver al moderno **Poetry** y al reciente **PDM**, pero, para cuando tengas más experiencia..  <https://python-poetry.org/> <https://pdm.fming.dev/latest/> |
| --- |

11º Instala MyPy y Pytest:

pip install mypy  
 pip install pytest

|  | **Ver video “**[**Entorno Virtual, MyPy y Pytest**](https://youtu.be/-4cjsxTvYRI)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Crear entornos virtuales con el módulo incorporado venv
* Activar entornos virtuales con Visual Studio Code