

# Python: Análisis de Datos y Ecosistemas Modernos de Analítica

---

INTRODUCCIÓN A  
PYTHON



---

CONOCER MÁS DE LAS PARTES  
FUNDAMENTALES DEL LENGUAJE

# Python

---

Ayer vimos un repaso general del lenguaje y sus principales características, hay aspectos adicionales que resultan indispensables conforme avanzamos en el desarrollo de aplicaciones más complejas.



# Excepciones y errores

---

Al igual que en otros lenguajes, durante la ejecución de un programa pueden existir errores y problemas causados por código erróneo, o inclusive dependencias externas que no funcionan de manera adecuada.

En Python, existen mecanismos que permiten que podamos tener control sobre estas situaciones.

[Built-in Exceptions — Python 3.12 documentation](https://docs.python.org/3.12/library/exceptions.html)



# Errores de sintaxis

---

Recordemos que Python generalmente detecta errores a medida que ejecuta el código, y no siempre antes de empezar un programa.

¡Sin embargo, existen errores que son detectados de manera inmediata!

```
>>> while True print('Hola')
      File "<stdin>", line 1
        while True print('Hola')
                        ^
SyntaxError: invalid syntax
```

Estos errores no pueden ser prevenidos, y la forma de solucionarlos es ajustar el código.



# Excepciones

---

A pesar de que una condición o código cumpla con una sintaxis correcta, es posible que el mismo cause errores tratando de ejecutarlo.

Los errores detectados durante la ejecución se conocen como **excepciones**.



# Excepciones

---

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + noexiste*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'noexiste' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```



# Excepciones

---

Existen escenarios en los que deseamos tener el poder de responder a una excepción en lugar de detener el programa y Python nos permite hacerlo mediante rutinas definidas en el lenguaje.

Sin embargo, tenemos que tener cuidado ya que existen ocasiones en las que esto no tiene sentido, y es mejor enfocarse en solucionar el problema de raíz **¿por qué estoy dividiendo para 0?**.





# Excepciones

---

Las excepciones en Python pueden ser de distintos tipos, cada uno describiendo con precisión el error.

El lenguaje **incluye errores predefinidos** y permite **crear excepciones personalizadas** adaptadas al contexto de la aplicación.

[Built-in Exceptions – Python 3.12 documentation](#)



# Controlando Excepciones

---

Es posible escribir manejar excepciones específicas, permitiendo controlar la situación según definiciones concretas mediante cláusulas **try-except**.

```
>>> while True:
...     try:
...         x = int(input("Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print("Número inválido, inténtelo de nuevo...")
```



# Encuesta

---

¿En qué situación se podría generar un ValueError en este código?

```
>>> while True:
...     try:
...         x = int(input("Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print("Número inválido, inténtelo de nuevo...")
```

La encuesta se abrirá por Zoom.



# Encuesta

---

¿En qué situación se podría generar un ValueError en este código?

int() lanza ValueError cuando el formato no puede convertirse en un número entero. Por ejemplo a = "hola", a = "3.14", etc.

```
>>> while True:
...     try:
...         x = int(input("Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print("Número inválido, inténtelo de nuevo...")
```



# Controlando Excepciones

---

## Cláusula try – Flujo básico

**try** → Ejecuta el bloque principal.

**Sin error** → El programa sigue normalmente.

**Con error y tipo coincide** (except) → Ejecuta el manejo definido.

**Con error y tipo no coincide** → El error se propaga.

**Sé explícito con las excepciones que esperas** para evitar ocultar errores reales e imprevistos.



# Controlando Excepciones

---

La clausula **try**, puede tener más de una cláusula **except**, es decir, en el contexto del código ejecutado, podemos actuar de manera diferente dependiendo del error.

```
try:
    codigo_que_puede_causar_error()
except ErrorD:
    print("D")
except ErrorC:
    print("C")
except ErrorB:
    print("B")
```

Python ejecutará **solo un bloque except**, y será **el primero** que coincida con el tipo de error.



# Controlando Excepciones

---

De igual manera, podemos controlar varios errores con el mismo código de control.

```
try:
    codigo_que_puede_causar_error()
except (RuntimeError, TypeError, NameError):
    # Todos estos se solucionan en el mismo bloque
    pass
```



# Controlando Excepciones

---

¿Cómo puede un error coincidir con varios tipos?

Las excepciones en Python son **objetos definidos por clases** → aplican **herencia** y **polimorfismo**.

Un **except** captura una excepción si:

- Es **exactamente** de la clase especificada, o
- Es de **una subclase** dentro de su jerarquía.





# Encuesta

---

En este ejemplo, ¿cuál será el bloque **except** que se imprimirá cuando emitamos este error?

La encuesta se abrirá por Zoom.

```
class MiErrorPropio(Exception):  
    pass  
  
class MiOtroError(MiErrorPropio):  
    pass  
  
try:  
    # Causemos un error a propósito  
    raise MiOtroError()  
except TypeError:  
    print("Error de tipo")  
except MiErrorPropio:  
    print("Mi error propio")  
except MiOtroError:  
    print("Mi otro error")
```



# Encuesta

---

En este ejemplo, ¿cuál será el bloque **except** que se imprimirá cuando emitamos este error?

“Error Uno”

1. Solo un bloque **except** se ejecuta como máximo.
2. Un **except** coincide si la excepción es **exactamente** del tipo declarado o de una clase **ascendente** en su jerarquía.
3. Python evalúa los **except** **de arriba hacia abajo** y se detiene en el primero que coincide.

```
class ErrorUno(Exception):  
    pass  
  
class ErrorDos(ErrorUno):  
    pass  
  
try:  
    # Causemos un error a propósito  
    raise ErrorDos()  
except TypeError:  
    print("Error Tres")  
except ErrorUno:  
    print("Error Uno")  
except ErrorDos:  
    print("Error Dos")
```



# Controlando Excepciones

---

Todas las excepciones en Python heredan de **BaseException**.

Capturar **BaseException** atrapa cualquier error, incluso los críticos.

## **Mala práctica:**

- Reduce la flexibilidad en el manejo de errores.
- Puede ocultar problemas graves.

Captura de forma explícita únicamente las excepciones que esperas.



# Controlando Excepciones

---

Un bloque **try...except** provee otra cláusula opcional llamada **else**, ésta se puede usar para ejecutar código en caso de que ningún bloque **except** haya sido ejecutado.

```
try:
    f = open("archivo", 'r')
except OSError:
    print('No se puede abrir el archivo')
else:
    print("archivo con", len(f.readlines()), "lineas")
    f.close()
```



# Controlando Excepciones

---

El flujo provee otra cláusula opcional llamada **finally**, esta se utiliza para ejecutar código **después** del try y los except, **haya o no ocurrido un error**.

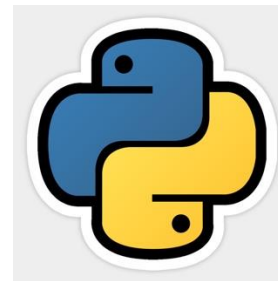
```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Adios!')
...
Adios!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```



# Controlando Excepciones

---

```
try:
    archivo = open("datos.txt", "r")
    contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no existe.")
else:
    print("Archivo leído correctamente.")
    print(f"Contenido: {contenido}")
finally:
    print("Fin del proceso.")
```



# Controlando Excepciones

---

Cuando una excepción ocurre, es posible acceder al objeto, el argumento de la excepción que incluye detalles de la misma

```
>>> try:
...     raise Exception('arroz', 'pollo')
... except Exception as instancia_error:
...     print(type(instancia_error))      # tipo del error
...     print(instancia_error.args)      # argumentos del error
...     print(instancia_error)           # resultado de __str__ en el error
...     x, y = instancia_error.args      # desempacar los argumentos
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('arroz', 'pollo')
('arroz', 'pollo')
x = arroz
y = pollo
```



# Forzando excepciones

---

La palabra clave **raise**

Permite al programador **lanzar** un error de forma intencional.

Se debe indicar una **instancia** de una clase que herede de `BaseException`:

- Puede ser una **excepción predefinida** (por ejemplo, `ValueError`, `TypeError`).
- O una **excepción personalizada** creada por nosotros.

```
>>> raise NameError('Hola')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Hola
```





# Excepciones del usuario

---

Es posible definir excepciones propias creando una clase que **herede de Exception**, ya sea **directamente o indirectamente**, a través de otra excepción que herede de Exception.

Aunque una clase de excepción puede implementar cualquier funcionalidad, lo más habitual es que sea **sencilla**.

```
# Definir una excepción personalizada
class MiError(Exception):
    def __init__(self, valor):
        super().__init__("El dato no puede ser negativo: " + valor)

# Usar la excepción personalizada
def procesar_dato(dato):
    if dato < 0:
        raise MiError(dato)
    return dato * 2
```



# Acciones de limpieza

---

La cláusula **finally** es por lo tanto bastante útil para realizar acciones de limpieza tal como cerrar recursos abiertos, o situaciones similares.

# Gestores de Contexto

---



# Contextos

---

## La cláusula with en Python

Se utiliza para **envolver bloques de código** con métodos definidos por un **gestor de contexto**.

Un gestor de contexto es un objeto que define:

- Una **rutina de entrada** (`__enter__`) → prepara el contexto.
- Una **rutina de salida** (`__exit__`) → libera o limpia recursos.

Usos comunes:

- Guardar y restaurar estados globales.
- Gestión automática de archivos, conexiones o recursos externos.



# Contextos

---

Un gestor de contexto es, en esencia, una **interfaz**: cualquier clase que implemente `__enter__` y `__exit__` puede actuar como tal.

Permite envolver código para **gestionar recursos automáticamente**.

Durante el manejo de archivos, se puede usar un gestor de contexto para garantizar que se **abran y cierren automáticamente**, sin que el usuario deba escribir código adicional para liberar el recurso.



# Contextos

---

En el siguiente ejemplo vemos dos formas de abrir un archivo.

La función `open()` implementa la **interfaz de gestor de contexto**, por lo que puede usarse con la cláusula `with`.

```
archivo = open("archivo", "w")  
  
##### MUCHAS OPERACIONES  
  
archivo.close()
```

```
with open("archivo", "w") as archivo:  
  
    ##### MUCHAS OPERACIONES
```

*Sin un gestor, es necesario que cerremos los recursos explícitamente.*

# Funciones y decoradores

---



# Funciones en Python

---

En Python, **las funciones son objetos**, al igual que cualquier otro elemento del lenguaje.

Esto **no es igual** en todos los lenguajes de programación.

```
>>> def sumar_uno(numero):  
...     return numero + 1  
  
>>> sumar_uno(2)  
3
```

Recordemos: Una función es una rutina que recibe 0 o más argumentos y devuelve un valor.





# Funciones en Python

---

Al ser un objeto de primera clase, significa que pueden asignarse a variables, pasarse como argumentos o devolverse desde otras funciones, tal como ocurre con otros tipos.

```
def decir_hola(nombre):  
    return f"Hola {nombre}"  
  
def decir_buenas(nombre):  
    return f"Buenas {nombre}"  
  
def llamar_saludo(funcion_de_saludo):  
    return funcion_de_saludo("Daniel")
```

```
>>> llamar_saludo(decir_hola)  
'Hola Daniel'  
  
>>> llamar_saludo(decir_buenas)  
'Buenas Daniel'
```



# Funciones en Python

---

Esto nos brinda gran flexibilidad al trabajar con funciones, ya que podemos utilizarlas de diversas maneras y adaptarlas a distintas necesidades a medida que nuestras aplicaciones se vuelven más complejas.



# Funciones internas

---

Es posible definir funciones internas dentro de otras funciones; estas permanecen limitadas al espacio de nombres de la función que las contiene y, por lo tanto, solo pueden ejecutarse desde ese mismo contexto.

```
def padre():  
    print("Hola padre()")  
  
    def primer_hijo():  
        print("Hola primer_hijo()")  
  
    def segundo_hijo():  
        print("Hola segundo_hijo()")  
  
    segundo_hijo()  
    primer_hijo()
```

En consecuencia, son accesibles únicamente dentro del bloque lógico correspondiente.



# Funciones internas

---

El orden en que se declaren las funciones internas no afecta su comportamiento, ya que su ejecución dependerá del punto del código en el que se utilicen.

De hecho, estas funciones no existen en el contexto del programa **hasta que la función que las contiene sea invocada.**



# Encuesta

---

¿Qué se imprimirá cuando se ejecute el código ?

```
def padre():  
    print("Hola padre")  
  
    def primer_hijo():  
        print("Hola primer_hijo")  
  
    def segundo_hijo():  
        print("Hola segundo_hijo")  
  
    segundo_hijo()  
    primer_hijo()  
  
padre()
```

```
?
```

```
?
```

```
?
```

La encuesta se abrirá por Zoom.



# Encuesta

---

¿Qué se imprimirá cuando se ejecute el código ?

```
def padre():  
    print("Hola padre")  
  
    def primer_hijo():  
        print("Hola primer_hijo")  
  
    def segundo_hijo():  
        print("Hola segundo_hijo")  
  
    segundo_hijo()  
    primer_hijo()  
  
padre()
```

```
Hola padre  
Hola segundo_hijo  
Hola primer_hijo
```

La encuesta se abrirá por Zoom.



# Funciones en return

---

Python puede devolver funciones internas como objetos, lo que permite construir y retornar funciones de forma dinámica para ser utilizadas en otros contextos o momentos de ejecución.

```
def obtener_hijo(num):  
    def primer_hijo():  
        return "Hola soy Daniel"  
  
    def segundo_hijo():  
        return "Hola soy Juan"  
  
    if num == 1:  
        return primer_hijo  
    else:  
        return segundo_hijo  
  
# Notemos que debemos llamar al resultado ya que es un objeto de función  
print(obtener_hijo(1)()) #Hola soy Daniel  
print(obtener_hijo(2)()) #Hola soy Juan
```



# Funciones de orden superior

---

Ahora que entendemos que Python nos permite trabajar y mover funciones con facilidad, podemos introducir el concepto de **funciones de orden superior** (*higher-order functions*).

## Funciones de orden superior:

Reciben una o más funciones como argumentos.

Devuelven otra función como resultado.

Este tipo de funciones permite construir comportamientos dinámicos y reutilizables, y se utiliza con frecuencia en patrones como decoradores, *callbacks*, *etc.*





# Decoradores

---

Un decorador es un caso particular de **función de orden superior**.

Son una herramienta muy útil para **modificar o ampliar comportamientos** sin cambiar el código original, favoreciendo la creación de soluciones modulares, reutilizables y fáciles de mantener.



# Una función que “decora”

---

Un decorador es una función que recibe otra función como argumento y devuelve una nueva función que incorpora comportamientos adicionales, sin modificar el código original de la función decorada.

```
def mi_decorador(funcion):  
    def funcion_final():  
        print("Imprimir algo antes de llamar la funcion.")  
        funcion()  
        print("Imprimir algo luego de llamar la funcion.")  
    return funcion_final  
  
def decir_hola():  
    print("Hola!")  
  
decir_hola = mi_decorador(decir_hola)
```

```
>>> decir_hola()  
Imprimir algo antes de llamar la funcion.  
Hola!  
Imprimir algo luego de llamar la funcion.
```



# Una función que “decora”

---

El proceso de decorar es nada más el proceso de aplicar la transformación deseada. Es decir

```
decir_hola = mi_decorador(decir_hola)
```

```
def mi_decorador(funcion):  
    def funcion_final():  
        print("Imprimir algo antes de llamar la funcion.")  
        func()  
        print("Imprimir algo luego de llamar la funcion.")  
    return funcion_final  
  
def decir_hola():  
    print("Hola!")  
  
decir_hola = mi_decorador(decir_hola)
```

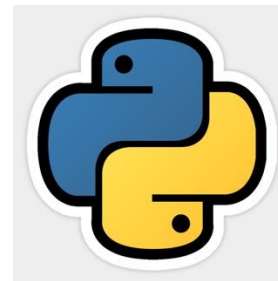


# Una función que “decora”

---

En resumen, un decorador envuelve una función y modifica su comportamiento, lo que permite añadir acciones de forma reutilizable, como escribir en archivos, registrar datos, mostrar información útil o cargar recursos importantes, sin alterar el código original de la misma.

```
def mi_decorador(funcion):  
    def funcion_final():  
        print("Imprimir algo antes de llamar la funcion.")  
        func()  
        print("Imprimir algo luego de llamar la funcion.")  
    return funcion_final  
  
def decir_hola():  
    print("Hola!")  
  
decir_hola = mi_decorador(decir_hola)
```



# Una función que “decora”

---

El siguiente ejemplo demuestra su poder, por ejemplo esto permite que “Hola” solo se imprima en las mañanas!

```
from datetime import datetime

def no_durante_el_dia(funcion):
    def funcion_modificada():
        if 7 <= datetime.now().hour < 22:
            funcion()
        else:
            pass # No hagamos nada, es de noche!
    return funcion_modificada

def decir_hola():
    print("Hola!")

decir_hola = no_durante_el_dia(decir_hola)
```



# Una función que “decora”

---

Python ofrece una forma más limpia y legible de aplicar decoradores mediante el uso del símbolo @, colocado justo encima de la definición de la función que se desea decorar.

```
from datetime import datetime

def no_durante_el_dia(funcion):
    def funcion_modificada():
        if 7 <= datetime.now().hour < 22:
            funcion()
        else:
            pass # No hacemos nada, es de noche!
    return funcion_modificada


@no_durante_el_dia
def decir_hola():
    print("Hola!")
```



# Encuesta

---

¿Cuál será el resultado de esta llamada?

 La función predefinida `.upper()` convierte todos los caracteres de una cadena en mayúsculas.

La encuesta se abrirá por Zoom.


```
def a_mayusculas(funcion):  
    def funcion_envoltura():  
        resultado = funcion()  
        return resultado.upper()  
    return funcion_envoltura  
  
@a_mayusculas  
def saludar():  
    return "hola mundo"  
  
print(saludar())
```



# Encuesta

---

¿Cuál será el resultado de esta llamada?

 La función predefinida `.upper()` convierte todos los caracteres de una cadena en mayúsculas.

HOLA MUNDO

La encuesta se abrirá por Zoom.

```
def a_mayusculas(funcion):  
    def funcion_envoltura():  
        resultado = funcion()  
        return resultado.upper()  
    return funcion_envoltura  
  
@a_mayusculas  
def saludar():  
    return "hola mundo"  
  
print(saludar())
```



# Laboratorio

---

El Github incluye el primer laboratorio para repasar los conceptos.

<https://github.com/danoc93/ista-python-analisis-2025>