# Python:
## Conceptos de Python en Producción

# Python

A pesar de que acabamos de ver un repaso general del lenguaje y sus características, existen temas que son necesarios a medida que desarrollamos 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

# Excepciones y errores

# 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

Como se puede ver en el ejemplo anterior, las excepciones vienen en diferentes tipos, tipos que describen exactamente el error causado.

**El lenguaje viene con una serie de errores predeterminados.**

Python de igual manera nos permite definir errores personalizados, que de manera más acertada reflejen los problemas potenciales en el contexto de nuestra aplicación.

Built-in Exceptions — Python 3.12 documentation

# Controlando Excepciones

Es posible escribir código que controle excepciones específicas, es decir que permita que el código corrija la situación de manera automática.

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

# Controlando Excepciones

La cláusula **try**, al igual que otros lenguajes, declara el inicio de un bloque en el cuál queremos tener control sobre un error o errores posibles.

- Primero, el código en el bloque entre **try** y **except** es ejecutado.
- Si no hay error, el código continúa con su flujo normal.
- Si hay un error, y si el error está definido en el **except**, el código dentro de ese bloque es ejecutado.
- Si hay un error, y el error no es igual a ninguno de los definidos en el bloque **except**, el programa propaga el error.

# 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")
```

Como máximo Python ejecutará un solo bloque **except**.

# 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

Las excepciones son objetos definidos por clases, es decir, las reglas de herencia y polimorfismo aplican las cláusulas **except.**

Una excepción es compatible con un bloque except si el bloque especifica aquella clase, **o cualquiera en su árbol de clases.**

# Encuesta

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

```python
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")
```

La encuesta se abrirá por Zoom.

# Encuesta

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

**“Mi error propio”**

1. Un solo bloque es ejecutado como máximo.
2. Un **except** aplica a un error si es que es igual al error declarado **o a una de las clases ascendentes en su árbol genealógico**.
3. Python ejecuta el código de arriba hacia abajo.

```
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")
```

# Controlando Excepciones

Todas las excepciones heredan de la excepción base **BaseException,** es decir que cualquier error en el programa podría capturarse con esta clase, sin embargo esto no es una buena práctica ya que no permite que tengamos flexibilidad y además es necesario ser explícito con los errores que esperamos como programadores.

**De otra manera podríamos enmascarar problemas graves!**

# 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

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 **raise** permite al programador decidir como levantar un error de manera forzada.

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

Simplemente se debe definir una instancia de un objeto que herede de la clase **BaseException**, es decir cualquier error predefinido o cualquier error creado por nosotros.

# Excepciones del usuario

Un usuario de Python puede definir sus propias Excepciones creando una clase que herede de la clase **Exception** de manera directa, o indirecta (a través de otra excepción que posea esta clase en su árbol de ascendencia).

**A pesar de que una clase para excepciones puede hacer lo que sea, es común que sean clases muy simples en las que como mucho agreguemos un mensaje específico.**

# Acciones de limpieza

Un bloque **try...except** posee una última cláusula opcional, llamada **finally.** Esta cláusula se ejecuta SIEMPRE al terminar el bloque, haya habido o no captura de errores.

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

# 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** se usa para envolver bloques de código con métodos definidos por un **gestor de contexto**.

Un gestor de contexto no es más que un objeto que define un contexto de ejecución que maneja una rutina de entrada,  y una rutina de salida.

Los gestores de contexto son generalmente utilizados para guardar y restaurar estados globales, bloqueo de recursos, o hasta gestión de los mismos.

3. Data model — Python 3.10.5 documentation

# Contextos

Un gestor de contexto no es más que una interfaz. Es decir, cualquier clase puede implementarla y ofrecer los beneficios del mismo.

Por ejemplo, es muy común usarlo en el contexto de archivos en donde mediante un contexto podemos asegurarnos que Python cierre y abra el recurso de manera automática!

# Contextos

Un gestor de contexto no es más que una interfaz. Es decir, cualquier clase puede implementarla y ofrecer los beneficios del mismo.

Por ejemplo, es muy común usarlo en el contexto de archivos en donde mediante un contexto podemos asegurarnos que Python cierre y abra el recurso de manera automática!

# Contextos

En el ejemplo siguiente podemos ver como podemos abrir un archivo de dos formas distintas!

*open* implementa la interfaz de gestor de contexto!

```
archivo = open("archivo", "w")

##### MUCHAS OPERACIOES

archivo.close()
```

```
with open("archivo", "w") as archivo:

    ##### MUCHAS OPERACIOES
```

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

# Funciones y decoradores

# Funciones en Python

En Python las funciones como todo lo demás son un objeto.

**¡Esto no es igual en todos los lenguajes!**

```
>>> 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 algo.**

# Funciones en Python

Al ser un objeto de primera clase, Python permite que las funciones sean pasadas o usadas como argumentos, tal como una variable int, string, o booleana!

```
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 permite muchas facilidades al trabajar con funciones ya que podemos utilizarlas de muchas maneras a medida que nuestras aplicaciones se vuelven más complejas.

# Funciones internas

Es posible definir funciones internas dentro de otras funciones. Estas funciones pertenecen exclusivamente al espacio de nombres de la función que la contiene y por lo tanto solo pueden ser ejecutada desde dicho contexto.

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

    def primer_hijo():
        print("Hola primer_hijo()")

    def segundo_hijo():
        print("Hola segundo_hijo() ")

    segundo_hijo()
    primer_hijo()
```

# Funciones internas

El orden de declaración de las mismas no importa, ya que su ejecución se determinará por el código en el que se usen.

Es más, las funciones internas no existen en el contexto del programa hasta que la función padre sea invocada!

# Encuesta

¿Qué se imprimirá cuando se ejecute el código ?

```python
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, en caso de que queramos construir funciones de forma dinámica!

```
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 alto nivel

Ahora que entendemos que el lenguaje nos permite trabajar y mover funciones de manera muy sencilla, debemos entender el concepto de funciones de primer nivel!

Una **función de alto nivel** es una función que:

- Recibe una o más funciones como argumentos
- Devuelve otra función como resultado

# Decoradores

Un decorador es una forma muy sencilla de implementar funciones de alto nivel en Python!

Son una herramienta bastante útil para crear código modular y fácil de manejar.

# Una función que “decora”

Un decorador es una función que agrega comportamientos a otra función.

```
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 a una función y cambia su comportamiento. Esto es útil para realizar acciones en funciones de manera reusable (escribir a archivos, reportar datos, imprimir información útil, cargar datos importantes, etc.)

```
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 de aplicar decoradores de manera más limpia, mediante la arroba @.

```
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

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