CLASE 15

Excepciones II + Debugging

raise, excepciones personalizadas, y codigo limpio

Python Fundamentals

Por que LANZAR errores a proposito?

Escenario: Funcion que recibe datos de usuario

```
                    def registrar_usuario(nombre, edad):
# Que pasa si edad = -5?
# Que pasa si edad = "veinte"?
# Que pasa si nombre = ""?
pass
```

OPCION A: Ignorar y seguir -> Datos basura en el sistema

OPCION B: Lanzar excepcion -> Forzar datos validos

Las excepciones son CONTRATOS. Si no cumples las reglas, el codigo se niega a continuar.

RAISE - Lanzar excepciones

Sintaxis:

```
raise TipoDeError("Mensaje descriptivo")
```

Ejemplo:

```
def dividir(a, b):
    if b == 0:
        raise ValueError("No puedes dividir entre cero")
    return a / b

# Uso:
resultado = dividir(10, 2)   # 5.0
resultado = dividir(10, 0)
# ValueError: No puedes dividir entre cero
```

raise es como decir: "ALTO. Algo esta mal. No voy a continuar."

RAISE para validar datos

```
                            def registrar_usuario(nombre, edad):
    """Registra un usuario con validaciones."""

    # Validar nombre
    if not isinstance(nombre, str):
        raise TypeError("El nombre debe ser string")
    if len(nombre.strip()) == 0:
        raise ValueError("El nombre no puede estar vacio")

    # Validar edad
    if not isinstance(edad, int):
        raise TypeError("La edad debe ser un numero entero")
    if edad < 0 or edad > 150:
        raise ValueError("La edad debe estar entre 0 y 150")

    # Si llegamos aqui, los datos son validos
    return {"nombre": nombre.strip(), "edad": edad}
```

registrar_usuario("Ana", 25)
-> {"nombre":"Ana","edad":25}

registrar_usuario("", 25)
-> ValueError: nombre vacio

registrar_usuario("Ana", -5)
-> ValueError: edad 0-150

TypeError = tipo incorrecto (str vs int) | ValueError = valor incorrecto (rango invalido)

EJERCICIO 1: Validar datos de tarea

Mision: Crear funcion crear_tarea(titulo, prioridad, categoria)

Validaciones:
1. titulo: string no vacio
2. prioridad: 'alta', 'media' o 'baja'
3. categoria: string no vacio
4. Invalido -> raise ValueError
5. Valido -> retornar diccionario

```
                    # Ejemplo de uso esperado:
crear_tarea("Estudiar", "alta", "estudio")
# {'titulo':'Estudiar','prioridad':'alta',
#  'categoria':'estudio','completada':False}

crear_tarea("", "alta", "estudio")
# ValueError: El titulo no puede estar vacio

crear_tarea("Estudiar", "urgente", "estudio")
# ValueError: Prioridad debe ser alta/media/baja
```

Usa: if valor not in ["alta", "media", "baja"]

Tiempo: 10 minutos

RE-RAISE - Atrapar, hacer algo, y volver a lanzar

```
                                    def procesar_archivo(nombre):
    try:
        with open(nombre) as f:
            datos = f.read()
        return datos
    except FileNotFoundError:
        print(f"Log: No se encontro {nombre}")  # Hacer algo
        raise  # Volver a lanzar la misma excepcion

# Uso:
try:
    procesar_archivo("no_existe.txt")
except FileNotFoundError:
    print("El archivo no existe")
```

Output:

```
                Log: No se encontro no_existe.txt
El archivo no existe
```

"raise" solo (sin tipo) vuelve a lanzar la excepcion actual. Muy usado para logging.

Excepciones Personalizadas

Sintaxis basica:

```
class MiError(Exception):
    """Descripcion de cuando usar este error."""
    pass
```

Ejemplo para el proyecto de tareas:

```
class TareaError(Exception):
    """Error base para el sistema de tareas."""
    pass

class TituloVacioError(TareaError):
    """El titulo de la tarea esta vacio."""
    pass

class PrioridadInvalidaError(TareaError):
    """La prioridad no es valida."""
    pass
```

Heredar de Exception (o de otra excepcion) hace que sea "atrapable" con try/except

Usando excepciones personalizadas

```
class TareaError(Exception):
    pass

class TituloVacioError(TareaError):
    pass

class PrioridadInvalidaError(TareaError):
    pass

def crear_tarea(titulo, prioridad, categoria):
    if not titulo or len(titulo.strip()) == 0:
        raise TituloVacioError("El titulo no puede estar vacio")

    if prioridad not in ["alta", "media", "baja"]:
        raise PrioridadInvalidaError(
            f"'{prioridad}' no es valida")

    return {"titulo": titulo.strip(),
            "prioridad": prioridad}
```

```
# Uso con manejo especifico:
try:
    tarea = crear_tarea("", "alta", "est")
except TituloVacioError:
    print("Escribe un titulo")
except PrioridadInvalidaError:
    print("Usa: alta, media o baja")
except TareaError:
    print("Error general")
```

Orden de except:
MAS especifico -> MENOS especifico
TituloVacio -> Prioridad -> TareaError

Cuando crear excepciones propias?

Usar ValueError/TypeError	Usar excepcion personalizada
Errores genericos	Errores de tu dominio
Scripts pequenos	Proyectos grandes
Una sola funcion	Multiples modulos
No necesitas diferenciar errores	Manejar cada error diferente

Resumen practico:

Script pequeno -> raise ValueError("mensaje")

Proyecto grande -> raise MiProyectoError("mensaje")

En duda, empieza con ValueError. Crea excepciones propias cuando las necesites.

BREAK - 10 MINUTOS

Volvemos con tecnicas de debugging

DEBUGGING - Encontrar y arreglar bugs

Bug = Error en la LOGICA de tu codigo (no un crash)

```
                            def calcular_promedio(notas):
    total = 0
    for nota in notas:
        total += nota
    return total / len(notas)

# Funciona?
print(calcular_promedio([10, 8, 9]))  # 9.0
print(calcular_promedio([]))  # ZeroDivisionError
```

3 tecnicas que veremos:

1. print() debugging

2. assert

3. Debugger de VS Code

Print Debugging - El clasico

Codigo con bug:

```
def buscar_palabra(texto, palabra):
    palabras = texto.split()
    for i in range(len(palabras)):
        if palabras[i] == palabra:
            return i
    return -1

# Bug: No encuentra "Python"
resultado = buscar_palabra(
    "Aprender Python, es genial", "Python")
print(resultado)  # Da -1 en vez de 1
```

Agregar prints de debug:

```
def buscar_palabra(texto, palabra):
    palabras = texto.split()
    print(f"DEBUG: {palabras}")
    print(f"DEBUG: buscando '{palabra}'")

    for i in range(len(palabras)):
        print(f"DEBUG: comparando "
              f"'{palabras[i]}' con '{palabra}'")
        if palabras[i] == palabra:
            return i
    return -1
```

Output revela el bug:

```
DEBUG: ['Aprender', 'Python,', 'es', 'genial']
DEBUG: comparando 'Python,' con 'Python'    <-- AHA! 'Python,' != 'Python'
```

Despues de encontrar el bug, BORRA los prints de debug. El codigo de produccion no debe tenerlos.

ASSERT - Validar suposiciones

Sintaxis:

```
assert condicion, "Mensaje si falla"
```

Ejemplo:

```
def calcular_promedio(notas):
    assert len(notas) > 0, "La lista no puede estar vacia"
    assert all(0 <= n <= 100 for n in notas), "Notas entre 0 y 100"

    return sum(notas) / len(notas)

# Uso:
print(calcular_promedio([90, 85, 88]))   # 87.67
print(calcular_promedio([]))
# AssertionError: La lista no puede estar vacia
print(calcular_promedio([90, 150]))
# AssertionError: Notas entre 0 y 100
```

raise vs assert:

raise -> Errores que PUEDEN ocurrir en produccion (datos de usuario)

assert -> Errores que NUNCA deberian ocurrir (bugs del programador)

Debugger de VS Code - La herramienta profesional

El debugger es como poner PAUSA en tu codigo y ver que esta pasando

1. Click en el margen izquierdo -> Crear breakpoint (punto rojo)
2. F5 o 'Run and Debug' -> Iniciar debugging
3. El codigo se PAUSA en el breakpoint
4. Ver valores de variables en el panel izquierdo
5. F10 (Step Over) -> Avanzar linea por linea

Botones del debugger:

`F5`	Continuar hasta siguiente breakpoint
`F10`	Step Over - avanzar una linea
`F11`	Step Into - entrar a una funcion
`Shift+F5`	Detener debugging

PEP 8 - La guia de estilo de Python

PEP 8 = Python Enhancement Proposal #8 = Como escribir codigo Python bonito

Por que importa?

- Codigo mas facil de LEER
- Codigo mas facil de MANTENER
- Consistencia en equipos
- Esperado en trabajos profesionales

"El codigo se lee MUCHAS mas veces de las que se escribe"

PEP 8 no son reglas absolutas, son RECOMENDACIONES de la comunidad Python

Reglas principales de PEP 8

Regla	Mal	Bien
Indentacion: 4 espacios	`if x:\n  print()`	`if x:\n    print()`
Lineas <= 79 chars	`Linea de 150 caracteres...`	`Dividir en multiples lineas`
2 lineas entre funciones	`def a():\n  pass\ndef b():`	`def a():\n  pass\n\n\ndef b():`
Espacios en operadores	`x=1+2`	`x = 1 + 2`
Sin espacio antes de :	`def func() :`	`def func():`
Imports arriba	`Import dentro de funcion`	`Imports al inicio del archivo`

```
# Ejemplo de linea larga dividida:

# MAL
resultado = funcion_larga(arg1, arg2, arg3, arg4)

# BIEN
resultado = funcion_larga(
    arg1, arg2,
    arg3, arg4
)
```

```
# Orden de imports:

import os          # Libreria estandar
import sys

import requests    # Terceros

from mi_proyecto import utils  # Propios
```

Convenciones de nombres

Elemento	Convencion	Ejemplo
Variables	snake_case	mi_variable, contador_tareas
Funciones	snake_case	calcular_promedio(), obtener_usuario()
Constantes	MAYUSCULAS	MAX_INTENTOS, PI
Clases	PascalCase	MiClase, TareaError
Modulos	snake_case	mi_modulo.py, utils.py

MAL

```
# MAL
def CalcularPromedio(ListaNotas):
    maxNota = 100
    return sum(ListaNotas) / len(ListaNotas)
```

BIEN

```
# BIEN
def calcular_promedio(lista_notas):
    MAX_NOTA = 100
    return sum(lista_notas) / len(lista_notas)
```

Los nombres deben ser DESCRIPTIVOS. "x" no dice nada, "contador_tareas" si.

Antes y despues

ANTES

```
def f(l):
t=0
for i in l:
    if i>0:t+=i
return t/len(l) if len(l)>0 else 0
```

DESPUES

```
def calcular_promedio_positivos(numeros):
"""Calcula el promedio de los numeros
positivos en la lista."""
if not numeros:
    return 0

suma_positivos = 0
for numero in numeros:
    if numero > 0:
        suma_positivos += numero

return suma_positivos / len(numeros)
```

Mejoras aplicadas:

- Nombre de funcion descriptivo
- Nombres de variables claros
- Espacios donde corresponde
- Docstring explicativo
- Manejo de caso vacio legible

Ambas funciones hacen EXACTAMENTE lo mismo. Pero una se entiende en 2 segundos.

EJERCICIO 2: Refactorizar codigo feo

CODIGO ORIGINAL

```
def p(d,k):
    if k in d:
        return d[k]
    else:
        return None

def a(l,v):
    l.append(v)
    return l

def c(l):
    n=[]
    for i in l:
        if i not in n:n.append(i)
    return n
```

Tu mision:

1. Renombrar funciones descriptivamente
2. Renombrar variables con nombres claros
3. Agregar espacios donde corresponda
4. Agregar docstrings
5. Seguir PEP 8

Pista: lean el codigo para entender que hace
ANTES de renombrar.

p = buscar/obtener | a = agregar | c = filtrar

Tiempo: 10 minutos

Excepciones en Python - Resumen completo

Clase 14: try/except/else/finally, multiples except

Clase 15: raise, excepciones propias, debugging, PEP 8

Concepto	Uso
`try/except`	Atrapar errores
`raise`	Lanzar errores
`except Error as e`	Capturar mensaje del error
`finally`	Codigo que siempre ejecuta
`else`	Codigo si NO hubo error
`class MiError(Exception)`	Crear excepcion propia
`assert`	Validar suposiciones (debugging)

Proxima clase: Modulos utiles (os, sys, pathlib, datetime)

Las excepciones bien usadas hacen tu codigo ROBUSTO y PROFESIONAL