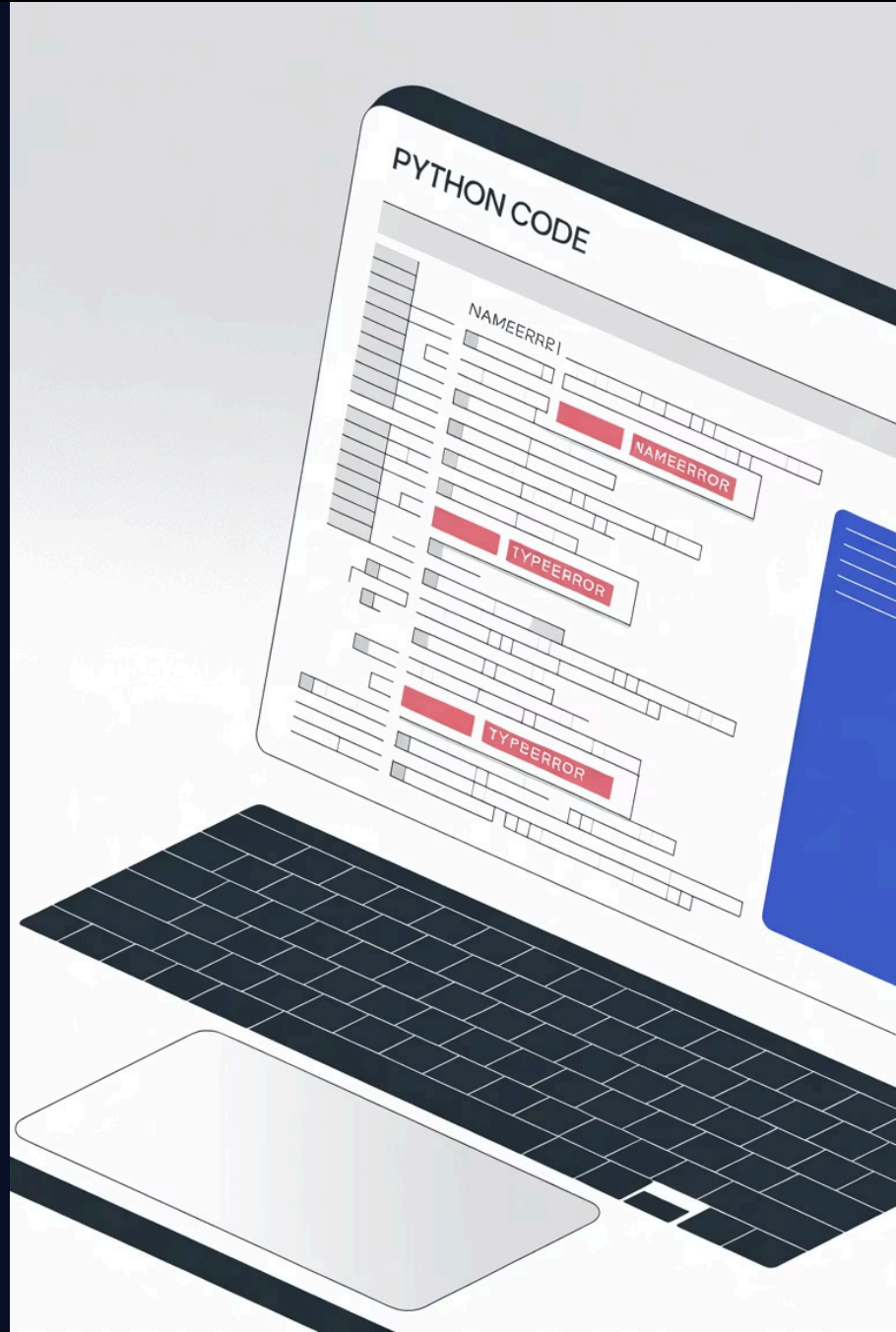
Universidad de los Andes | Educación Continua
Vicerrectoría Académica

Introducción al Backend y Arquitectura Cliente-Servidor

Curso: Desarrollo Backend con Django

Manejo de Errores, Módulos y Paquetes en Python



Agenda

1

Excepciones y Errores

Conceptos básicos y tipos comunes

2

Manejo de Excepciones

try, except, finally y raise

3

Excepciones Personalizadas

Creación y uso

4

Módulos y Paquetes

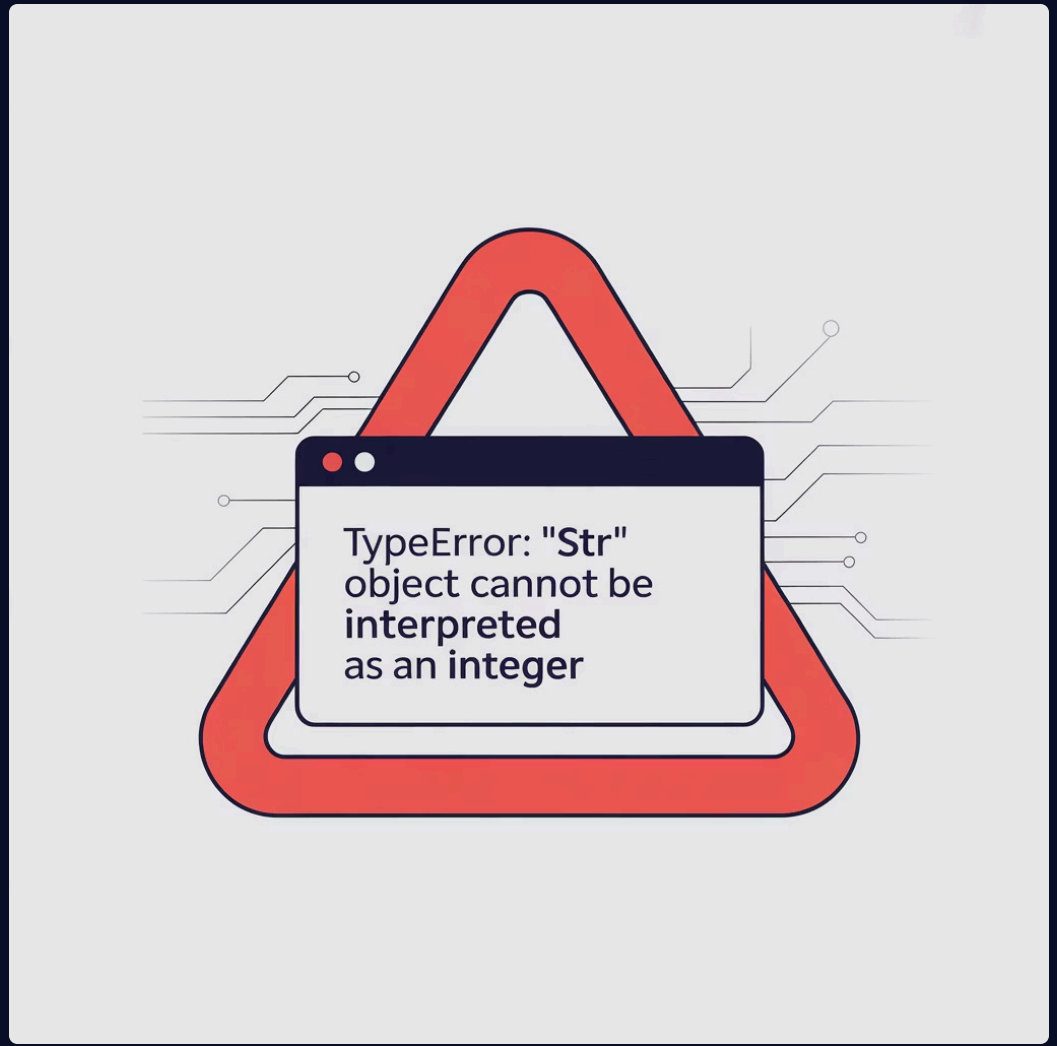
Organización de código

¿Qué es una excepción?

Una **excepción** es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de instrucciones.

En Python, las excepciones son objetos que representan errores.

Cuando ocurre un error, Python lanza (o "arroja") una excepción.



Si la excepción no es capturada y manejada, el programa se detendrá.

Errores comunes en Python

SyntaxError

Error en la escritura del código

```
if x = 5:  # Error: debe ser ==
```

NameError

Variable o función no definida

```
print(variable_no_definida)
```

TypeError

Operación con tipos incompatibles

```
"texto" + 5  # No se puede sumar texto y número
```

ZeroDivisionError

División por cero

```
resultado = 10 / 0
```

¿Por qué ocurren los errores?

Los errores son inevitables en el desarrollo de software y pueden ocurrir por múltiples razones:

- Entrada de datos incorrecta
- Problemas de conexión a servicios externos
- Errores de lógica en el código
- Recursos insuficientes (memoria, espacio)
- Problemas de permisos

Un buen desarrollador no evita los errores, sino que aprende a manejarlos correctamente.

Manejo de Excepciones: try-except

El bloque try-except permite ejecutar código que podría generar errores sin detener el programa:

```
try:
    # Código que podría generar un error
    numero = int(input("Ingrese un número: "))
    resultado = 10 / numero
    print(f"El resultado es: {resultado}")
except ValueError:
    # Se ejecuta si ocurre un ValueError
    print("Error: Debe ingresar un número válido")
except ZeroDivisionError:
    # Se ejecuta si ocurre un ZeroDivisionError
    print("Error: No se puede dividir por cero")
```

Python busca la primera cláusula `except` que coincida con la excepción lanzada.

Capturando múltiples excepciones

Forma 1: Múltiples bloques except

```
try:
    # Código que podría fallar
except TypeError:
    # Manejo de TypeError
except ValueError:
    # Manejo de ValueError
```

Forma 2: Excepciones agrupadas

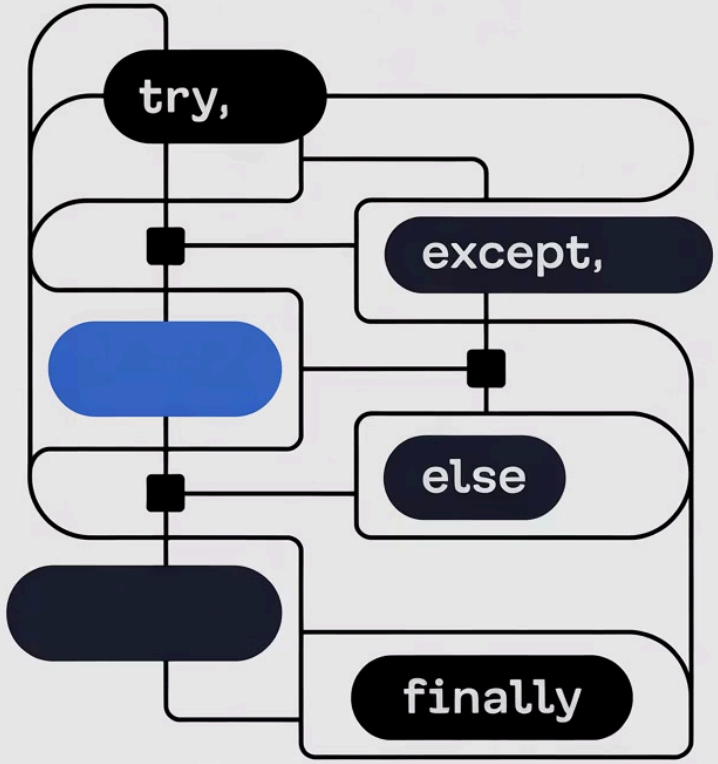
```
try:
    # Código que podría fallar
except (TypeError, ValueError):
    # Manejo para ambos tipos
```

Forma 3: Capturar cualquier excepción

```
try:
   # Código que podría fallar
except Exception as e:
   print(f"Ocurrió un error: {e}")
   # Podemos acceder a la información del error
```

Bloques else y finally

```
try:
   archivo = open("datos.txt", "r")
   contenido = archivo.read()
except FileNotFoundError:
   print("El archivo no existe")
else:
   # Se ejecuta si NO ocurre ninguna excepción
   print(f"Contenido leído: {len(contenido)} caracteres")
finally:
   # Se ejecuta SIEMPRE, haya error o no
   if 'archivo' in locals():
      archivo.close()
      print("Proceso completado")
```



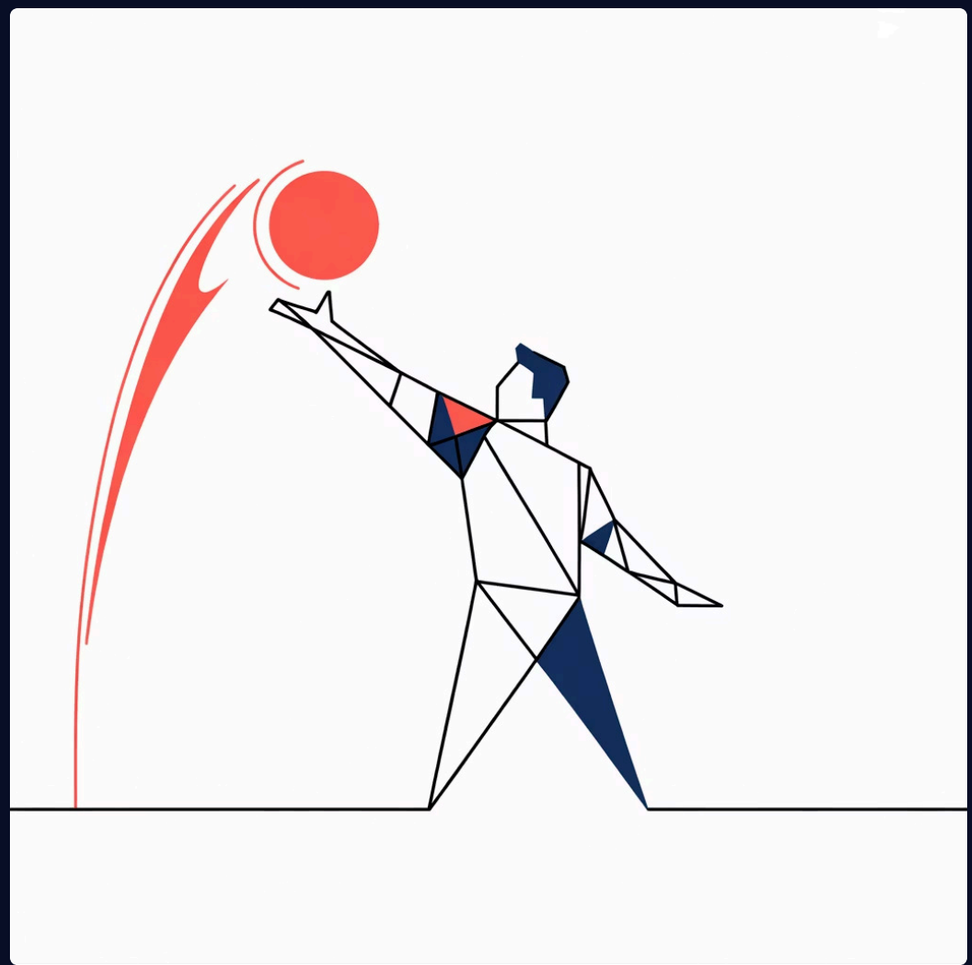
La instrucción raise

`raise` permite lanzar excepciones de forma manual cuando detectamos una condición de error.

```
def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("No se permite dividir por cero")
    return a / b

try:
    resultado = dividir(10, 0)
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

También podemos re-lanzar una excepción después de manejarla:

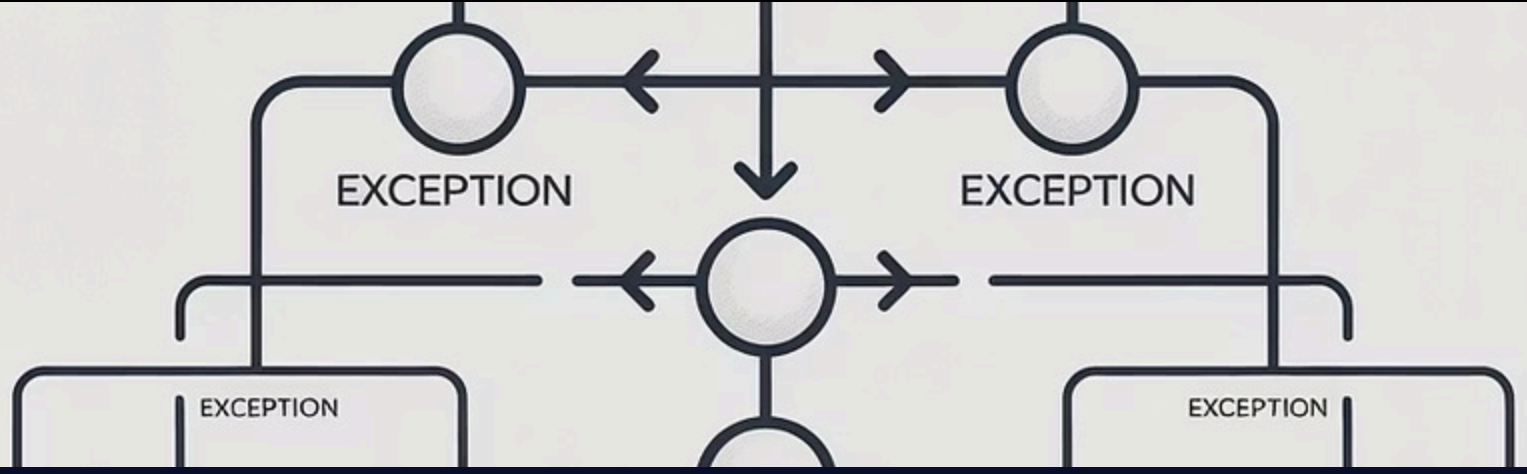
```
try:
    # Código que podría fallar
except ValueError:
    print("Ocurrió un ValueError")
    # Hacemos algo con el error
    raise  # Re-lanza la última excepción
```



Excepciones Personalizadas

Podemos crear nuestras propias clases de excepciones heredando de `Exception`:

```
class EdadInvalidaError(Exception):
    """Excepción lanzada cuando la edad está fuera del rango permitido"""
    def __init__(self, edad, mensaje="Edad fuera del rango permitido (0-120)"):
        self.edad = edad
        self.mensaje = mensaje
        super().__init__(self.mensaje)

def verificar_edad(edad):
    if not 0 <= edad <= 120:
        raise EdadInvalidaError(edad)
    return f"Edad válida: {edad} años"

try:
    print(verificar_edad(150))
except EdadInvalidaError as e:
    print(f"Error: {e.mensaje}. Valor recibido: {e.edad}")
```



Jerarquía de Excepciones

Las excepciones en Python forman una jerarquía de clases:

Es importante conocer esta jerarquía al capturar excepciones, ya que si capturamos una excepción padre, también capturaremos todas sus excepciones hijas.

Buenas prácticas en el manejo de excepciones

Ser específico

Captura las excepciones más específicas posibles, no uses `except:` sin especificar el tipo.

Bloques try pequeños

Envuelve en el bloque try sólo el código que puede generar la excepción.

Documentar excepciones

Documenta qué excepciones puede lanzar tu función y en qué circunstancias.

Logging en vez de print

Usa el módulo logging para registrar errores en lugar de print.

Limpiar recursos

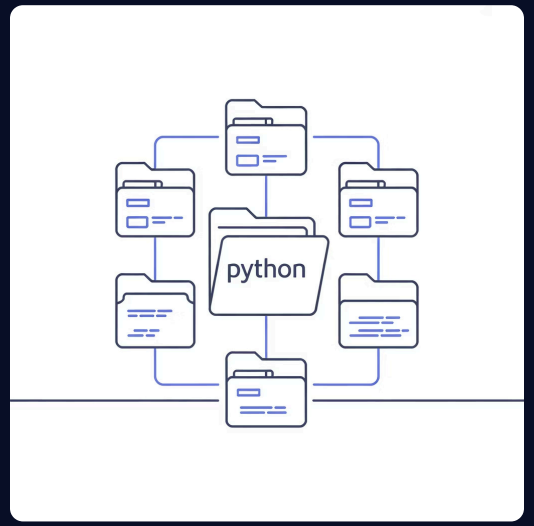
Usa finally o with para asegurar que los recursos se liberen correctamente.

¿Qué es un módulo en Python?

Un **módulo** es simplemente un archivo `.py` que contiene código Python reutilizable.

Los módulos permiten:

- Organizar el código en archivos separados
- Reutilizar funcionalidad en diferentes proyectos
- Evitar conflictos de nombres
- Mantener el código más limpio y legible



Cada módulo tiene su propio namespace, lo que evita colisiones de nombres entre diferentes partes del código.

Importación de módulos

1

import módulo

```
import math
resultado = math.sqrt(16)  # 4.0
```

2

from módulo import objeto

```
from math import sqrt
resultado = sqrt(16)  # 4.0
```

3

import módulo as alias

```
import math as m
resultado = m.sqrt(16)  # 4.0
```

4

from módulo import *

```
from math import *  # No recomendado
resultado = sqrt(16)  # 4.0
```

⚠️ La importación con `*` no es recomendada porque puede causar conflictos de nombres.

Módulos de la biblioteca estándar

Módulo math

```
import math

# Funciones matemáticas
print(math.sqrt(25))  # 5.0
print(math.factorial(5))  # 120
print(math.sin(math.pi/2))  # 1.0

# Constantes
print(math.pi)  # 3.141592653589793
print(math.e)  # 2.718281828459045
```

Módulo datetime

```
from datetime import datetime, timedelta

# Fecha y hora actual
ahora = datetime.now()
print(ahora) # 2023-10-25 15:30:45.123456

# Operaciones con fechas
mañana = ahora + timedelta(days=1)
print(mañana)

# Formateo de fechas
print(ahora.strftime("%d/%m/%Y %H:%M"))
```

Más módulos útiles de la biblioteca estándar

random

Generación de números aleatorios y selecciones al azar.

```
import random
print(random.randint(1, 10))
print(random.choice(['a', 'b', 'c']))
```

os

Interacción con el sistema operativo.

```
import os
print(os.getcwd())  # Directorio actual
os.mkdir("nueva_carpeta")
```

json

Manejo de datos en formato JSON.

```
import json
datos = {"nombre": "Ana", "edad": 25}
json_str = json.dumps(datos)
```

re

Expresiones regulares.

```
import re
coincide = re.match(r'\d+', "123abc")
print(coincide.group()) # 123
```

Creando nuestro propio módulo

Para crear un módulo, simplemente guardamos código en un archivo .py:

Archivo: calculadora.py

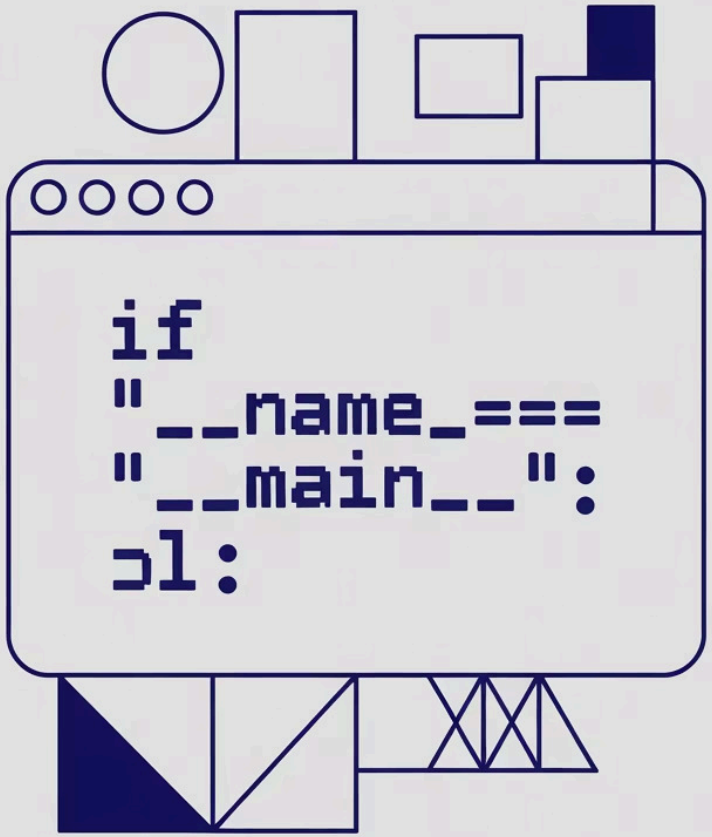
```
# calculadora.py
"""
Módulo que proporciona funciones matemáticas básicas.
"""

def sumar(a, b):
    """Suma dos números y devuelve el resultado."""
    return a + b

def restar(a, b):
    """Resta dos números y devuelve el resultado."""
    return a - b

PI = 3.14159  # Constante
```

Usando nuestro módulo

```
# main.py
import calculadora

print(calculadora.sumar(5, 3))  # 8
print(calculadora.restar(10, 4))  # 6
print(calculadora.PI)  # 3.14159

# También podemos importar elementos específicos
from calculadora import sumar, PI
print(sumar(7, 2))  # 9
print(PI)  # 3.14159
```

Los módulos se buscan en:

1. Directorio actual
2. PYTHONPATH
3. Directorios de instalación de Python



Variable especial __name__

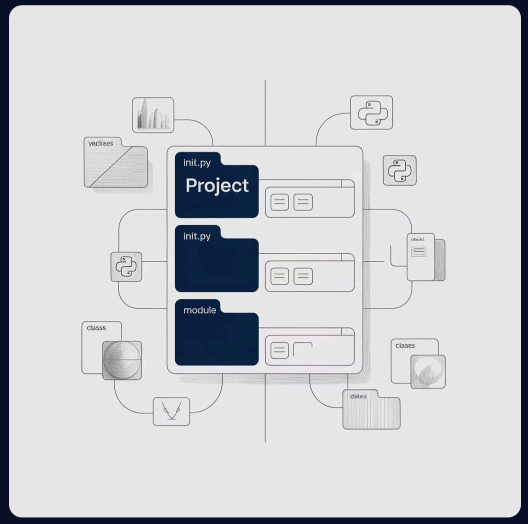
```
# mimodulo.py
def saludar(nombre):
    return f"Hola, {nombre}!"

print("Módulo cargado:", __name__)

if __name__ == "__main__":
    # Este código solo se ejecuta si el archivo
    # se ejecuta directamente, no al importarlo
    print("El módulo está siendo ejecutado directamente")
    print(saludar("Estudiante"))
```

Esta técnica permite que un módulo sea tanto importable como ejecutable por sí mismo.

¿Qué es un paquete en Python?

Un **paquete** es una carpeta que contiene varios módulos relacionados y un archivo especial llamado `__init__.py`.

Los paquetes permiten organizar módulos relacionados en una estructura jerárquica, como un sistema de archivos.



Un paquete puede contener sub-paquetes, creando estructuras complejas para proyectos grandes.

Estructura básica de un paquete

```
mi_paquete/           # Carpeta principal (nombre del paquete)
    __init__.py       # Archivo que hace que Python trate la carpeta como paquete
    modulo1.py        # Un módulo dentro del paquete
    modulo2.py        # Otro módulo
    subpaquete/       # Un subpaquete (otra carpeta)
        __init__.py   # Archivo que hace que esta carpeta sea un subpaquete
        modulo3.py    # Módulo dentro del subpaquete
```

El archivo `__init__.py` puede estar vacío, pero debe existir para que Python reconozca la carpeta como un paquete.

Importación desde paquetes

Podemos importar módulos o elementos específicos desde un paquete:

```
# Importar un módulo del paquete
import mi_paquete.modulo1

# Usar una función del módulo
resultado = mi_paquete.modulo1.funcion()

# Importar un módulo con alias
import mi_paquete.modulo2 as m2
m2.otra_funcion()

# Importar una función específica
from mi_paquete.modulo1 import funcion
resultado = funcion()
```

Importar desde subpaquetes:

```
# Importar un módulo de un subpaquete
import mi_paquete.subpaquete.modulo3

# Usar una función del subpaquete
mi_paquete.subpaquete.modulo3.funcion()

# Importar directamente
from mi_paquete.subpaquete.modulo3 import funcion
funcion()
```

El archivo __init__.py

El archivo `__init__.py` se ejecuta cuando se importa el paquete. Puede estar vacío o contener código de inicialización:

```
# mi_paquete/__init__.py

# Variables del paquete
VERSION = "1.0.0"
AUTOR = "Tu Nombre"

# Importar elementos para hacerlos accesibles directamente desde el paquete
from .modulo1 import funcion1, funcion2
from .modulo2 import Clase1

# Al hacer esto, podemos usar:
# from mi_paquete import funcion1
# en lugar de:
# from mi_paquete.modulo1 import funcion1
```

El punto `.` antes del nombre del módulo indica una importación relativa dentro del mismo paquete.

Creando un paquete completo

Crear la estructura de carpetas

Organiza tus carpetas y archivos según la estructura de paquetes deseada.

Agregar archivos __init__.py

Coloca un archivo __init__.py en cada carpeta que quieras que sea un paquete o subpaquete.

Escribir los módulos

Desarrolla cada módulo .py con su funcionalidad específica.

Configurar importaciones

Define qué se exporta en cada __init__.py para facilitar las importaciones.

Instalar el paquete (opcional)

Crear un setup.py para instalar el paquete en el sistema si lo deseas.

Preguntas de reflexión

1

¿Qué puede pasar si no se manejan correctamente las excepciones?

Los programas pueden terminar abruptamente, dejando recursos sin liberar (archivos abiertos, conexiones de red), y proporcionando mensajes de error crípticos para los usuarios.

2

¿Cuándo crear una excepción personalizada?

Cuando los errores tienen un significado específico en el dominio de tu aplicación y necesitas proporcionar información contextual relevante.

3

¿Cómo decidir entre módulos y funciones?

Usa módulos cuando el código se vuelve demasiado grande o cuando la funcionalidad es reutilizable en diferentes contextos.

Bibliotecas populares y su organización

Las bibliotecas Python populares son excelentes ejemplos de organización en paquetes:

Requests

Biblioteca para hacer peticiones HTTP.

```
import requests

response =
requests.get('https://api.exampl
e.com/data')
print(response.status_code)
print(response.json())
```

Pandas

Análisis de datos.

```
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
print(df.describe())
```

Django

Framework web completo.

```
from django.http import
HttpResponse

def index(request):
    return HttpResponse("Hola,
mundo!")
```

Explora el código fuente de estas bibliotecas para aprender cómo organizan proyectos complejos.

Recursos adicionales

Documentación oficial de Python

- Tutorial sobre errores y excepciones
- Tutorial sobre módulos

Libros recomendados

- "Fluent Python" de Luciano Ramalho
- "Python Cookbook" de David Beazley y Brian K. Jones

Herramientas útiles

- pylint: Para análisis estático de código
- pytest: Framework para pruebas unitarias
- poetry: Gestión de dependencias y paquetes

Resumen

Excepciones

Usa try/except/else/finally para manejar errores de forma controlada. Crea excepciones personalizadas para tus necesidades específicas.

Módulos

Archivos .py que contienen código reutilizable. Organiza tu código en módulos para facilitar su mantenimiento.

Paquetes

Carpetas con módulos relacionados y un archivo __init__.py. Permite organizar proyectos complejos.

Buenas prácticas

Usa nombres descriptivos, documenta tu código, maneja adecuadamente los errores y organiza tu proyecto en una estructura lógica.

Universidad de los Andes | Educación Continua
Vicerrectoría Académica

#EDCOUNIANDES

https://educacioncontinua.uniandes.edu.co/

Contacto: educacion.continua@uniandes.edu.co

Universidad de los Andes | Vigilada Mineducación. Reconocimiento como Universidad: Decreto 1297 del 30 de mayo de 1964. Reconocimiento personería jurídica: Resolución 28 del 23 de febrero de 1949 MinJusticia.