«Talento Tech»

Automation Testing

Clase 3

Buenos Aires Ciudad

Clase N° 3 | Fundamentos de Python Parte 2 y Control de Versiones

Temario:

- Estructuras de datos:
  - Listas, tuplas y diccionarios
  - Operaciones comunes
- Funciones:
  - Definición y llamada
  - Parámetros y retorno de valores
- Manejo de excepciones:
  - Try, except, finally
  - Tipos de excepciones
- Introducción a Git y GitHub:
  - Conceptos básicos (repositorio, commit, push, pull)
  - Crear y clonar repositorios
  - Comandos esenciales
- Ejercicios prácticos

Objetivos de la clase:

En esta clase nos enfocaremos en fortalecer las bases de la programación estructurada a través del uso de funciones. Exploraremos la diferencia entre un enfoque lineal y uno modular, aprendiendo a organizar el código en bloques claros y reutilizables. A partir de un script de calculadora, trabajaremos en su refactorización para que cada operación sea manejada desde una función específica, mejorando la legibilidad y el mantenimiento del código. Además, incorporaremos manejo de errores mediante excepciones controladas, y daremos nuestros primeros pasos en el flujo de trabajo con Git, aprendiendo prácticas colaborativas esenciales para trabajar en equipo a través de GitHub. Esta combinación entre organización del código y control de versiones sentará las bases para un desarrollo más profesional y colaborativo.

Funciones en Python

Una función es un bloque de código con un propósito definido. A diferencia de un programa lineal —donde las instrucciones se ejecutan una tras otra—, con funciones agrupamos tareas específicas, facilitamos la reutilización y mejoramos la mantenibilidad.

Ventajas de usar funciones:

- **Reutilización**: Escribís la lógica una sola vez y la invocás donde sea necesario.
- **Claridad**: Cada función tiene un nombre que describe lo que hace.
- **Mantenimiento**: Si detectás un error, lo corregís en un solo lugar, no en todo el flujo.

Sintaxis de una función

```
def saludar(nombre):
  """Imprime un saludo al usuario"""
  print(f"¡Hola, {nombre}!")
```

- **def**: palabra clave para definir.
- **saludar**: nombre de la función.
- **(nombre)**: parámetro de entrada.
- **docstring**: breve documentación.
- **return**: devuelve un valor al invocar (opcional).

Calculadora modular: cada operación en su propia función

En la clase anterior trabajamos con una calculadora básica estructurada de forma lineal, donde todas las operaciones, entradas y salidas estaban integradas en un único bloque de código. Si bien era funcional, esta estructura no permitía reutilizar ni organizar fácilmente las diferentes partes del programa.

Calculadora básica en Python (programación lineal)

Integramos todo en un único flujo:

```
num1 = float(input("Primer número: "))
num2 = float(input("Segundo número: "))
oper = input("Operación (+, -, *, /): ")
if oper == '+':
  result = num1 + num2
elif oper == '-':
  result = num1 - num2
elif oper == '*':
  result = num1 * num2
elif oper == '/':
  result = num1 / num2
else:
  result = "Inválida"
print(f"Resultado: {result}")
```

Refactorización con funciones

Hasta ahora, construimos una calculadora básica en Python utilizando un enfoque **lineal**, donde todas las instrucciones se ejecutaban una tras otra dentro de un único bloque de código. Si bien esto funciona, **no es la forma más eficiente ni escalable** de programar, especialmente cuando el proyecto crece o se vuelve más complejo.

En esta etapa del curso, vamos a **refactorizar** ese mismo código aplicando el concepto de **funciones**, uno de los pilares de la programación modular. Esto significa que cada parte del proceso —como sumar, restar, multiplicar o dividir— vivirá dentro de su propia función independiente

¿Por qué es mejor usar funciones?

- **Claridad**: cada función tiene un nombre que indica claramente qué hace.
- **Reutilización**: podés usar la misma función en distintos lugares del programa sin repetir código.
- **Mantenimiento**: si hay un error en una operación, solo necesitás corregirlo en un lugar.

- **Testeo**: es más fácil probar funciones individuales para asegurarse de que funcionen correctamente (lo que será clave cuando veamos Pytest).

```
def sumar(a, b):
    return a + b
def restar(a, b):
    return a - b
def multiplicar(a, b):
    return a * b
def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero.")
    return a / b

# Menú interactivo
def calculadora():
    print("\n--- CALCULADORA PYTHON ---")
    a = float(input("Primer número: "))
    b = float(input("Segundo número: "))
    print("1) Sumar  2) Restar  3) Multiplicar  4) Dividir")
    opcion = input("Elije (1-4): ")
    try:
        if opcion == '1': resultado = sumar(a, b)
        elif opcion == '2': resultado = restar(a, b)
        elif opcion == '3': resultado = multiplicar(a, b)
        elif opcion == '4': resultado = dividir(a, b)
        else:
            print("Opción inválida.")
            return
        print(f"Resultado: {resultado}")
    except ValueError as e:
        print(f"Error: {e}")

if __name__ == '__main__':
    calculadora()
```

¿Qué hicimos en este ejemplo?

1. Creamos funciones específicas para cada operación matemática (`sumar`, `restar`, `multiplicar`, `dividir`)
2. Incluimos control de errores con `try-except` para evitar que el programa se rompa si el usuario comete un error (por ejemplo, dividir por cero).
3. Agrupamos todo en una función principal llamada `calculadora()`, que actúa como interfaz interactiva con el usuario.

4. Usamos `if __name__ == '__main__'` para que el programa se ejecute solo si lo ejecutamos directamente (buena práctica en Python).

¿Qué mejora?

- Cada operación vive en su propia función.
- Facilita agregar nuevas operaciones.
- Simplifica la identificación y corrección de errores.

Este pequeño cambio en la estructura marca una gran diferencia en cómo escribimos y entendemos nuestro código. A partir de ahora, vas a empezar a pensar en términos de **bloques funcionales reutilizables**, lo cual es fundamental para automatizar pruebas, trabajar en equipo y desarrollar proyectos más robustos.

Manejo de excepciones: control de errores

Cuando escribimos programas, especialmente aquellos que interactúan con usuarios o datos externos, es fundamental prever posibles errores que podrían hacer que el código se detenga inesperadamente. Para eso usamos el manejo de excepciones, una herramienta que nos permite anticipar y controlar errores de forma ordenada y segura.

Estructura básica

```
try:
    valor = int(input("Ingresa un número entero: "))
    resultado = 10 / valor
    print(f"10 / {valor} = {resultado}")
except ZeroDivisionError:
    print("Error: División por cero.")
except ValueError:
    print("Error: Entrada inválida, no es un número entero.")
finally:
    print("Operación finalizada.")
```

¿Qué hace cada bloque?

- `try:`
  Aquí colocamos el código que **puede generar un error**. Python intentará ejecutarlo normalmente.

- `except:`
  Si ocurre un error en el bloque `try`, Python busca una excepción que **coincida con el tipo de error**. Si la encuentra, ejecuta ese bloque en lugar de detener el programa.

- `finally:`
  Este bloque **siempre se ejecuta**, ocurra o no un error. Es útil para mostrar mensajes finales, liberar recursos o cerrar archivos.

Tipos de excepciones comunes:

- `ZeroDivisionError`
- `ValueError`
- `KeyError`

Flujo de trabajo básico con Git

Git es una herramienta de control de versiones que nos permite registrar, rastrear y compartir los cambios realizados en nuestros proyectos de código. Esencial en cualquier trabajo colaborativo o desarrollo profesional.

El flujo típico para registrar y compartir cambios:

1. Editás archivos en tu proyecto.
2. `git add` agrega cambios al área de preparación.
3. `git commit` crea un snapshot en el historial.
4. `git push` envía los commits al repositorio remoto.

Creación de repositorios

- `git init:` convierte la carpeta actual en un repositorio Git local.
- `git clone URL_DEL_REPOSITORIO`: descarga una copia de un repositorio remoto.

Comandos básicos

- `git add archivo.txt`
- `git commit -m "Descripción breve del cambio"`
- `git push origin main`
    - **origin**: nombre por defecto del repositorio remoto.
    - **main**: rama principal (antes `master`).

Ramas (branches) y fusiones (merges)

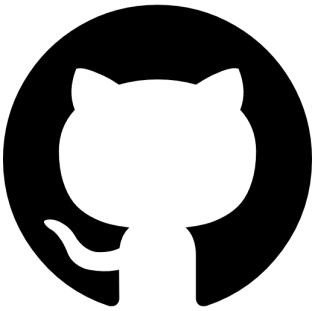
Las ramas permiten trabajar en nuevas funcionalidades sin afectar la rama principal.

```
git branch nueva-funcionalidad  # crea rama
git checkout nueva-funcionalidad # cambia a la rama

# trabajás...
git add .
git commit -m "Implementar nueva funcionalidad"
git checkout main
git merge nueva-funcionalidad # fusiona cambios
git branch -d nueva-funcionalidad  # elimina rama local
```

Colaboración en proyectos mediante GitHub

GitHub es un servicio que aloja repositorios remotos y facilita la colaboración:



- **Fork**: copia personal de un repositorio para trabajar independientemente.
- **Pull Request (PR)**: propuesta de cambios al repositorio original.
- **Revisiones de código**: comentarios y discusión antes de fusionar.

Buenas prácticas de versionado

- **Mensajes de commit claros**: usar infinitivo y describir el cambio (p.ej., `Agregar función dividir`).
- **Commits pequeños y frecuentes**: facilitan revertir y entender el historial.
- **Uso de ramas**: mantener `main` estable, desarrollar en ramas separadas y fusionar cuando estén listas.
- **Modelos de branching**: adoptar Git Flow, GitHub Flow o similares para orden y consistencia.

🗨️**Storytelling**

¡Un nuevo desafío en Talento Lab!

Tras haber configurado tu entorno y repasado la teoría, Silvia y Matías quieren que afiances lo aprendido con un ejercicio concreto antes de avanzar a conceptos más avanzados.

Silvia comenta:

"Queremos que domines las estructuras de control y la entrada/salida de datos. Por eso, el primer script será una calculadora básica que reciba números y muestre resultados."

Matías agrega:

"Este ejercicio te servirá para interiorizar condicionales y bucles. Más adelante lo modularizaremos con funciones."

Ejercicios Prácticos

1. Refactorizar el script de la calculadora lineal en al menos 4 funciones separadas.

   Tendrias que hacer mínimo, 4 funciones, una por cada tipo de operación. Como ser: Sumar(), Restar(), Multiplicar() y Dividir()

2. Añadir manejo de excepciones para entradas inválidas y división por cero.
3. **Sincronizar con Git: Flujo Git local**:
   - `git init` para crear un repositorio en la carpeta del ejercicio.
   - `git add y git commit` de forma incremental.
   - Crear y cambiar a la rama `feature/calculadora-lineal`.
   - Fusionar con `main` al finalizar.
4. **Subir a Github:**
   - Subir la rama `en la que estes trabajando`


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad