Lectura

Manejo de errores y depuración

Python Intermedio

Introducción

La depuración y el manejo adecuado de errores son habilidades esenciales en cualquier lenguaje de programación, especialmente en Python. Esta unidad explora herramientas fundamentales como las estructuras ***try, except, finally***; el uso del depurador **(debugger)** en entornos de desarrollo integrados **(IDE)**, así como el control de versiones con **Git**, los cuales permiten al desarrollador diagnosticar, corregir y prevenir errores de forma efectiva y profesional.

Manejo de excepciones

Sentencias *try*, *except* y *finally*

En Python, las excepciones son interrupciones que ocurren cuando el intérprete encuentra una situación que no puede resolver, durante la ejecución del código. Estas interrupciones pueden deberse a errores de lógica, operaciones inválidas o recursos inaccesibles, entre otros.

Para evitar que un programa se detenga bruscamente cuando ocurre un error, Python proporciona las sentencias try, except y finally, que permiten manejar errores de forma controlada. La estructura general del manejo de excepciones es:

```
try:
    # Bloque de código que puede generar una excepción
except NombreDeExcepcion:
    # Código que se ejecuta si ocurre esa excepción
except OtraExcepcion:
    # Otro tipo de excepción específica
except Exception:
    # Excepción general para capturar cualquier otro error
finally:
    # Código que se ejecuta siempre, ocurra o no una excepción, es opcional.
```

En donde:

- ***try*:** Contiene el bloque de código que podría generar una excepción.
- ***except*:** Define qué hacer si ocurre un error específico.
- ***finally*:** Se ejecuta siempre, ocurra o no una excepción, y se usa comúnmente para cerrar archivos, liberar recursos o mostrar mensajes finales. Es un bloque opcional.

Uso de excepciones específicas

Python permite capturar excepciones específicas, según **el tipo de error** que se espera que pueda ocurrir. Es recomendable manejar primero las

excepciones específicas, ya que permiten una respuesta más precisa y controlada. Un ejemplo de lo anterior es:

```
try:
    numero = int(input("Ingresa un número: "))
    resultado = 100 / numero
    print("Resultado:", resultado)
except ValueError:
    print("Error: El valor ingresado no es un número entero.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
```

En este ejemplo:

- ➔ Si el usuario ingresa “abc”, se lanza un *ValueError*.
- ➔ Si el usuario ingresa “0”, se lanza un *ZeroDivisionError*.

Los tipos comunes de excepciones son:

Excepción	Descripción
ZeroDivisionError	División por cero.
ValueError	Conversión inválida de tipos de datos.
TypeError	Operación entre tipos incompatibles.
IndexError	Acceso a índices inexistentes en listas.
KeyError	Búsqueda de una clave inexistente en un diccionario.

FileNotFoundError	Archivo no encontrado al intentar abrirlo.

A continuación, se presenta un ejemplo **del manejo de los diversos tipos de excepciones:**

```
try:
    numero = int(input("Ingresa un número: "))
    resultado = 100 / numero
    print("Resultado:", resultado)
except ValueError:
    print("Error: Debes ingresar un número válido.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
except Exception as e:
    print("Ocurrió un error inesperado:", str(e))
finally:
    print("Ejecución finalizada.")
```

Este enfoque permite manejar diferentes tipos de errores de manera precisa, mejorando la robustez del código y la experiencia del usuario.

Uso de excepciones generales (Exception)

Cuando se desea capturar cualquier tipo de error, incluso si no se anticipó específicamente, se puede utilizar ***except Exception:*** Esta técnica debe aplicarse con precaución, ya que puede ocultar errores inesperados, si no se combina con un buen diagnóstico (*print, log, traceback*, etc.).

Ejemplo:

```
try:
    # Bloque propenso a errores
    archivo = open("datos.txt", "r")
    contenido = archivo.read()
    print(contenido)
except Exception as e:
    print("Ocurrió un error inesperado:", str(e))
finally:
    print("Fin del programa.")
```

Aquí, el *except Exception* atrapa cualquier error, mostrando el mensaje de error sin detener el programa.

Orden recomendado en múltiples excepciones

Es importante recordar que Python **evalúa los bloques *except* de arriba hacia abajo**. Por ello:

- ➔ Las excepciones más específicas deben colocarse primero.
- ➔ La excepción genérica (*Exception*) debe ir al final, para evitar que intercepte errores que deberían haber sido tratados específicamente.

Ejemplo incorrecto:

```
try:
    pass
except Exception:
    pass
except ZeroDivisionError:  # Esta nunca se alcanzará
    pass
```

Ejemplo correcto:

```
try:
    pass
except ZeroDivisionError:
    pass
except Exception:
    pass
```

Uso del bloque finally

El bloque finally contiene instrucciones que se ejecutan siempre, ocurra o no una excepción. Es ideal para liberar recursos; cerrar archivos, conexiones de red o bases de datos, y dejar el sistema en un estado limpio.

Ejemplo:

```
try:
    archivo = open("archivo.txt", "r")
    # Lectura del archivo
except FileNotFoundError:
    print("El archivo no fue encontrado.")
finally:
    print("Intento de lectura finalizado.")
```

Importancia en el desarrollo profesional

Un programa bien estructurado debe ser capaz de **anticipar y gestionar errores** sin interrumpir su funcionamiento general. Por ejemplo, en aplicaciones de bases de datos, análisis financiero o interacción con usuarios, el manejo de excepciones es indispensable para evitar pérdidas de datos o bloqueos inesperados del sistema.

El uso de ***try, except y finally*** se convierte en una práctica esencial en el ciclo de vida del software, ya que permite identificar puntos críticos del código y mantener la ejecución controlada, incluso ante situaciones imprevistas.

Debugger en las IDE

¿Qué son los *breakpoints*?

Un *breakpoint* (punto de interrupción) es una herramienta utilizada **durante el proceso de depuración (*debugging*)** que permite detener la ejecución del programa en una línea específica del código. Al hacerlo, el programador puede inspeccionar el estado del programa en ese momento: variables, flujo, condiciones, objetos, etc. Los ***breakpoints*** son fundamentales para:

- ➔ Identificar errores lógicos y comportamientos inesperados.
- ➔ Revisar el valor de las variables paso a paso.
- ➔ Comprender cómo fluye la ejecución del programa.
- ➔ Probar hipótesis sin modificar el código con print().

Utilidad del depurador en las IDE

El depurador o debugger es una herramienta integrada en la mayoría de los **Entornos de Desarrollo Integrado (IDE)**, como Visual Studio Code, PyCharm, Thonny, entre otros. Este permite al usuario:

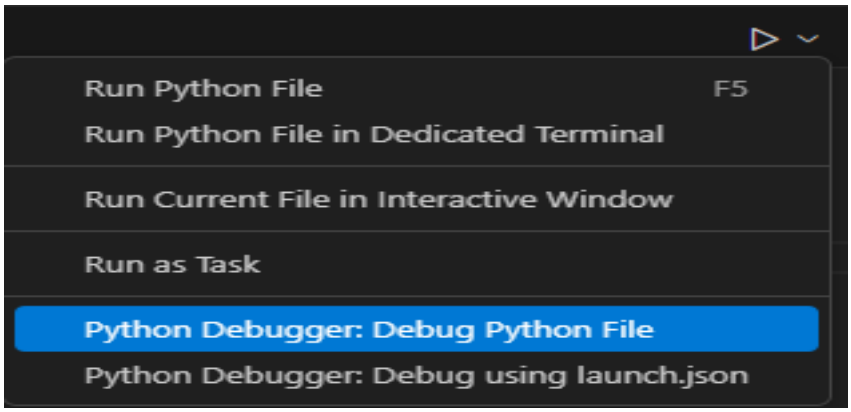
- ➔ Ejecutar el código paso a paso (step into, step over, step out).
- ➔ Ver valores de variables en tiempo real.
- ➔ Evaluar expresiones mientras se ejecuta el código.
- ➔ Detectar y aislar errores sin interrumpir completamente el programa.
- ➔ Establecer condiciones en breakpoints para que solamente se activen bajo ciertas circunstancias.

Ejemplo en Visual Studio Code:

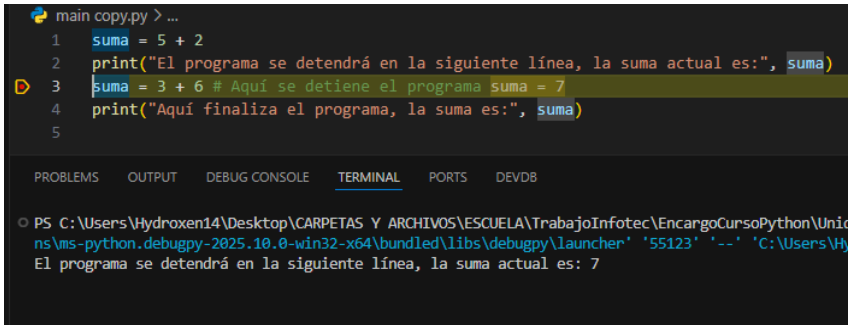
1. Coloca un breakpoint al lado izquierdo del número de la línea donde deseas detener la ejecución.

```
1  suma = 5 + 2
2  print("El programa se detendrá en la siguiente línea, la suma actual es:", suma)
3  suma = 3 + 6 # Aquí se detiene el programa
4  print("Aquí finaliza el programa, la suma es:", suma)
```

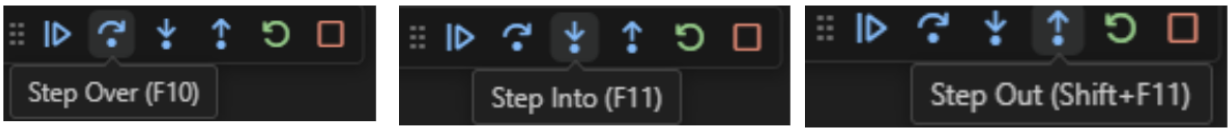
2. Ejecuta el programa en modo **depuración.**



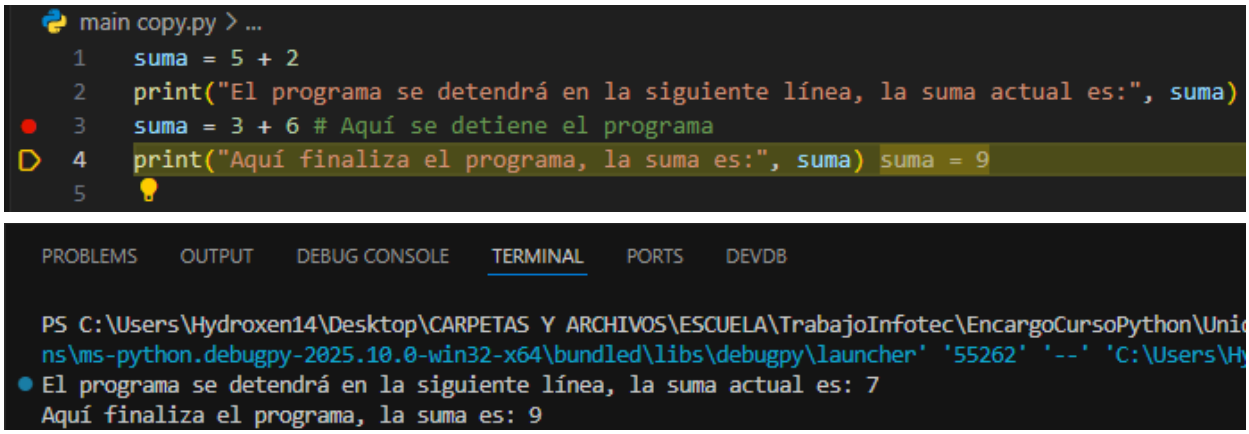
3. El programa se detendrá en el **breakpoint** y mostrará el valor actual de todas las variables.



4. Utiliza los botones **“step over”** y **“step into”** para navegar el código.



5. Observa **cómo cambian las variables y el flujo del programa con cada paso**.



¿Por qué usar el depurador y no solo *print()*?

Aunque imprimir valores con ***print()*** puede parecer más sencillo, esta técnica no escala bien y puede ocultar o alterar errores. El *debugger* permite un análisis detallado, no invasivo y controlado, especialmente útil en programas complejos o con múltiples funciones, clases e interacciones entre objetos.

Uso de Git para control de versiones

Git es un sistema de control de versiones distribuido, el cual permite a los desarrolladores **rastrear cambios en el código fuente** a lo largo del tiempo; **colaborar en equipo** de manera efectiva y **mantener un historial completo de modificaciones.** Su uso se ha vuelto fundamental en el desarrollo de software moderno.

¿Qué es el control de versiones?

El control de versiones es un sistema que:

- **Guarda** cambios en archivos a lo largo del tiempo.
- **Permite** recuperar versiones anteriores.
- **Facilita** el trabajo colaborativo entre varios desarrolladores.
- **Ayuda** a detectar y revertir errores.

Git es el sistema de control de versiones más popular y es utilizado junto con plataformas como **GitHub, GitLab o Bitbucket**.

Principales comandos de Git

A continuación se muestran **los comandos más utilizados al trabajar con Git desde la terminal:**

- ***git init*:** Inicializa un nuevo repositorio en el directorio actual.
- ***git clone <url>*:** clona un repositorio existente desde un servidor remoto (ej. GitHub).
- ***git status*:** muestra el estado de los archivos (modificados, nuevos, sin seguimiento).
- ***git add <archivo>*:** agrega archivos al área de preparación (*staging area*).
- ***git commit -m "mensaje"*:** registra los cambios en el historial del repositorio.
- ***git log*:** muestra el historial de commits.
- ***git push*:** envía los commits al repositorio remoto.
- ***git pull*:** trae los cambios del repositorio remoto al local.
- ***git branch*:** lista las ramas del repositorio.
- ***git checkout <rama>*:** cambia a una rama específica.

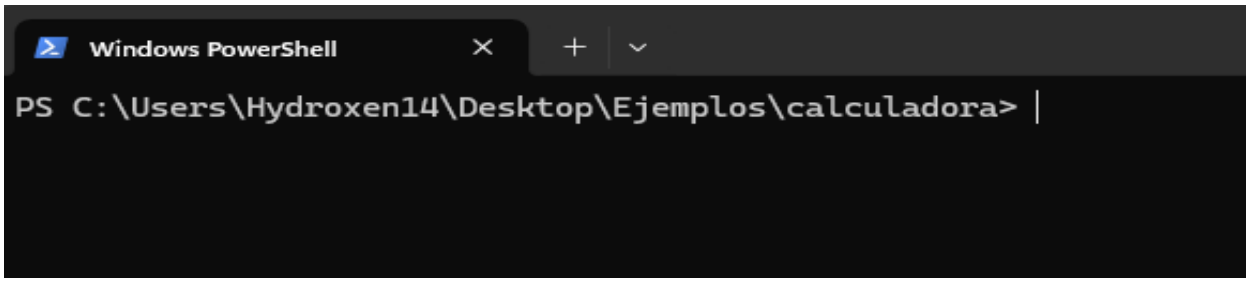
Buenas prácticas al usar Git

- ➔ Realizar commits frecuentes con mensajes descriptivos.
- ➔ Trabajar en ramas para desarrollar nuevas funcionalidades sin afectar la rama principal.
- ➔ Hacer pull antes de hacer push, para evitar conflictos.
- ➔ Usar .gitignore para excluir archivos que no deben almacenarse en el repositorio (por ejemplo, archivos temporales, configuraciones locales, contraseñas).

Caso práctico: Control de versiones con Git

Estás desarrollando un pequeño sistema de calculadora en Python y deseas llevar un control de versiones local utilizando **Git**, sin usar **GitHub** ni otros servicios remotos. Tu objetivo es **guardar el historial de cambios, probar nuevas versiones y regresar a versiones anteriores, si es necesario.**

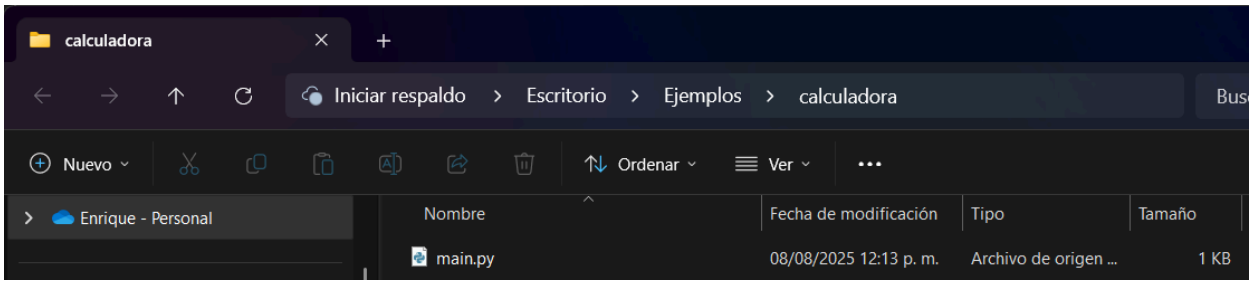
1. **Abrir una terminal:**

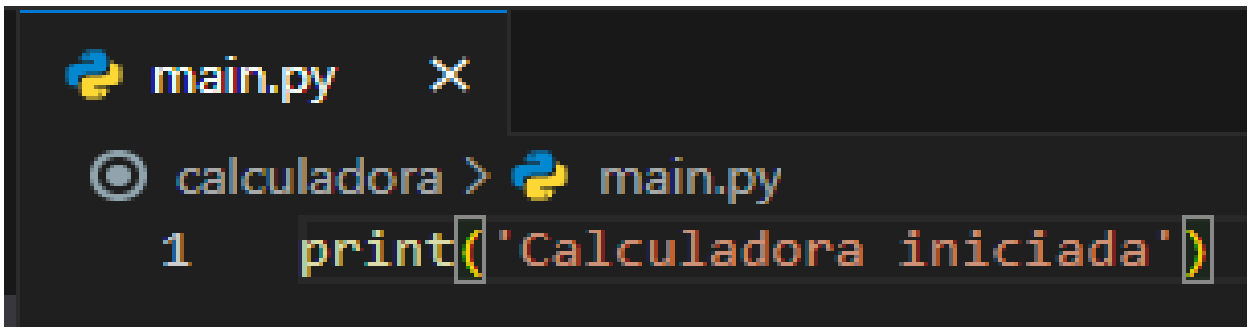


2. **Crear carpeta del proyecto:**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos> mkdir calculadora


    Directorio: C:\Users\Hydroxen14\Desktop\Ejemplos


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        08/08/2025  12:11 p. m.                calculadora


PS C:\Users\Hydroxen14\Desktop\Ejemplos> cd calculadora
```

3. **Crear archivo principal .py con el siguiente contenido:**





4. **Inicializar repositorio local de Git:**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git init
Initialized empty Git repository in C:/Users/Hydroxen14/Desktop/Ejemplos/calculadora/.git/
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora>
```

5. **Verificar el estado del repositorio:**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        main.py

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora>
```

6. **Agregar archivo al área de preparación (staging):**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git add main.py
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora>
```

Esto indica que deseas guardar ese archivo en la siguiente versión (commit).

7. **Hacer primer commit:**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git commit -m "Versión inicial de la calculadora"
[master (root-commit) 967e935] Versión inicial de la calculadora
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 main.py
```

Esto guarda el estado actual del archivo con una descripción clara.

8. **Realizar cambios y registrar versiones:**
   - Edita el archivo main.py y reemplaza su contenido con:

```python
def sumar(a, b):
    return a + b

print("Resultado:", sumar(5, 7))
```

- Guarda el archivo y ejecuta:

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git add main.py
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git commit -m "Agregada función de suma"
[master 99a3351] Agregada función de suma
 1 file changed, 0 insertions(+), 0 deletions(-)
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora>
```

Ahora tienes dos versiones registradas del archivo: ***una inicial y una con la función sumar.***

9. **Consultar historial de versiones:**

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git log
commit 99a3351a28def4ee4c708bbfdff3e449534f654b (HEAD -> master)
Author: TuNombre <TuNombre@prueba.com>
Date:   Fri Aug 8 12:19:18 2025 -0600

    Agregada función de suma

commit 967e9354f2cb88a5fc61ac3d5aa3b717b29c330e
Author: TuNombre <TuNombre@prueba.com>
Date:   Fri Aug 8 12:16:46 2025 -0600

    Versión inicial de la calculadora
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora>
```

Verás una lista de commits con sus identificadores ***(hash)****, fechas y mensajes.*

10. **Comparar versiones:**

    Si deseas comparar lo que cambió entre la primera y la segunda versión, ejecuta:

```
git diff HEAD~1 HEAD
```

En donde:

- **HEAD~1** es el id del primer elemento.

- **HEAD** es el id del segundo elemento.

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git diff 960268fb032c4bdfa3b706668eeb0e7bda1b4c32 920698f5c2c1582b3aae35231c256c840bf725b1
diff --git a/main.py b/main.py
index 28c2313..3c9d974 100644
--- a/main.py
+++ b/main.py
@@ -1 +1,4 @@
-print('Calculadora iniciada')
\ No newline at end of file
+def sumar(a, b):
+    return a + b
+
+print("Resultado:", sumar(5, 7))
\ No newline at end of file
```

11. **Volver a una versión anterior (sin borrar los cambios):**

```
git checkout HEAD~1 main.py
```

```
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git checkout 960268fb032c4bdfa3b706668eeb0e7bda1b4c32 main.py
Updated 1 path from f532c67
```

Esto restaura temporalmente el archivo main.py a su versión anterior, ideal para revisión o pruebas.

Para volver al último commit:

```
Windows PowerShell
PS C:\Users\Hydroxen14\Desktop\Ejemplos\calculadora> git checkout main.py
Updated 0 paths from the index
```

Conclusión

El manejo de errores y la depuración, son habilidades fundamentales para el desarrollo profesional en Python. A través de las estructuras ***try, except y finally***, el programador puede anticiparse a posibles fallos en la ejecución, manteniendo la estabilidad del sistema y proporcionando respuestas adecuadas al usuario. El uso de excepciones específicas permite respuestas

más precisas, mientras que el uso de excepciones generales, ayuda a capturar situaciones imprevistas. Combinadas con un manejo adecuado del bloque ***finally***, estas estructuras fortalecen la confiabilidad del software.

Por otro lado, la incorporación de herramientas de depuración, como los ***breakpoints*** y el ***debugger*** de las **IDEs**, permite observar el comportamiento interno del código de forma controlada. Por último, **Git** proporciona un entorno confiable para registrar cambios, coordinar colaboraciones y mantener un historial limpio y seguro del desarrollo del software. Estas herramientas son esenciales para analizar el flujo de ejecución, verificar el estado de las variables y corregir errores lógicos sin interrumpir completamente el desarrollo. Su uso refuerza, no solo la organización del trabajo, sino también la profesionalización del proceso de programación. En conjunto, estas técnicas fomentan la escritura de código **más limpio, predecible y profesional**, alineado con las mejores prácticas del desarrollo de software moderno.

Elaboró: Enrique Quezada Próspero
Contenido: Enrique Quezada Próspero
DI: Génesis Equihua

Referencias:

1. Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
2. GitHub Docs. (2025). *Git Handbook*. Get Started/ Using Git. https://docs.github.com/en/get-started/using-git/about-git
3. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
4. Microsoft. (2025a). *Debug Python in Visual Studio Code*. https://code.visualstudio.com/docs/python/debugging
5. Python Software Foundation. (2025b). *Errors and Exceptions*. https://docs.python.org/3/tutorial/errors.html
6. Python Software Foundation. (2025c). *The Python Debugger (pdb)*. https://docs.python.org/3/library/pdb.html