😊 Python Debugger

What is the Python Debugger (PDB)?

Python Debugger (PDB) is a built-in module in Python that allows developers to **step through** code, **examine variables**, and **identify errors** interactively. It helps in **troubleshooting and debugging** programs by enabling controlled execution, breakpoints, and step-by-step inspection.

Use Cases of Python Debugger

1. **Finding Logical Errors** – Helps trace incorrect outputs by examining variable values at different points.
2. **Step-by-Step Execution** – Allows running code one line at a time to understand program flow.
3. **Fixing Crashes and Exceptions** – Identifies the exact location where an error occurs.
4. **Testing Edge Cases** – Useful for verifying how code behaves with unusual inputs.
5. **Optimizing Code Execution** – Helps analyze loops and conditions to improve efficiency.

Benefits of Using Python Debugger

- **Interactive Code Inspection** – Allows checking variable values in real-time.
- **Precise Error Identification** – Pinpoints the exact line where an issue occurs.
- **Step-by-Step Execution** – Makes it easier to follow program logic.
- **Efficient Debugging** – Reduces trial-and-error debugging by providing direct insights.
- **Breakpoints for Control** – Enables stopping execution at specific points to analyze code behavior.

Python Debugger (PDB) is an essential tool for **efficient debugging**, making it easier to track issues and improve code quality.

😄 To begin with the Lab

1. You've probably used a variety of print statements to try to find errors in your code. A better way of doing this is by using Python's built-in debugger module (pdb). The pdb module implements an interactive debugging environment for Python programs.
2. It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.
3. In the first example below, you can see that we defined values for x, y and z then printed the result. We got result 1 but for result 2 we got an error.

[1]:
```
x = [1,3,4]
y = 2
z = 3

result = y + z
print(result)
result2 = y+x
print(result2)
```

```
5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 7
      5 result = y + z
      6 print(result)
----> 7 result2 = y+x
      8 print(result2)

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

4. We start by importing pdb or python debugger, then we write our statement. After that when we execute this, it will return us the result 1, but for result 2, we know it will cause an error.
5. So, for that the debugger started, and we can type in the values to know the result. This is what the debugger is offering us. Now to quit the debugger, type **q** and press enter.

[*]:
```
import pdb

x = [1,3,4]
y = 2
z = 3

result = y + z
print(result)

# Set a trace using Python Debugger
pdb.set_trace()

result2 = y+x
print(result2)
```

```
5
--Return--
None
> c:\users\pulkit\appdata\local\temp\ipykernel_24316\1419980936.py(11)<module>()

ipdb>  y
2
ipdb>  x
[1, 3, 4]
ipdb>  z
3
ipdb>  y+z
5
ipdb>  y+x
*** TypeError: unsupported operand type(s) for +: 'int' and 'list'
ipdb>  x+z
*** TypeError: can only concatenate list (not "int") to list
ipdb>
```

```
(Pdb) q
---------------------------------------------------------------------------
BdbQuit                                   Traceback (most recent call last)
<ipython-input-2-1084246755fa> in <module>()
      9
     10 # Set a trace using Python Debugger
---> 11 pdb.set_trace()
     12
     13 result2 = y+x

C:\Users\Marcial\Anaconda3\lib\bdb.py in trace_dispatch(self, frame, event, arg)
     53             return self.dispatch_call(frame, arg)
     54         if event == 'return':
---> 55             return self.dispatch_return(frame, arg)
     56         if event == 'exception':
     57             return self.dispatch_exception(frame, arg)

C:\Users\Marcial\Anaconda3\lib\bdb.py in dispatch_return(self, frame, arg)
     97             finally:
     98                 self.frame_returning = None
---> 99             if self.quitting: raise BdbQuit
    100             # The user issued a 'next' or 'until' command.
    101             if self.stopframe is frame and self.stoplineno != -1:

BdbQuit:
```