What is `pdb`?

`pdb` (Python Debugger) is a built-in module in Python used for interactive debugging. It allows you to pause the execution of your program at certain points, inspect variables, execute code, and step through your program line by line to find bugs or understand the flow.

Basic Commands in `pdb`

Here are some basic commands you'll frequently use while debugging:

1. **`pdb.set_trace()`**:
   - Inserts a breakpoint in your code.
   - When the program execution reaches this line, it pauses, and you can interact with the debugger.
2. **`continue` (or `c`)**:
   - Resumes program execution until the next breakpoint or error.
3. **`step` (or `s`)**:
   - Steps into a function. If the current line is a function call, it will stop at the first line inside that function.
4. **`next` (or `n`)**:
   - Moves to the next line within the same function, but does not step into any function calls.
5. **`list` (or `l`)**:
   - Displays a few lines of code around the current line to give you context.
6. **`print` (or `p`)**:
   - Evaluates and prints the value of an expression or variable.
   - Example: `p my_var`
7. **`where` (or `w`)**:
   - Shows the current position in the program, including the call stack (a trace of function calls leading to the current point).

8. **quit (or q):**
    - Exits the debugger and stops the program.
9. **args (or a):**
    - Prints the arguments passed to the current function.

How to Use pdb

Step 1: Setting a Breakpoint

You can set a breakpoint in your code by adding pdb.set_trace() at the point where you want to start debugging. The program will pause execution when it hits this line.

Example:

python Copy code

```python
import pdb

def add_numbers(a, b):
    # Add two numbers and return the result
    return a + b

def main():
    x = 10
    y = '20'

    pdb.set_trace()  # Pause here to start debugging

    result = add_numbers(x, y)  # This will raise an error because x is an integer
    print(result)

if __name__ == "__main__":
    main()
```

Step 2: Running the Code

To run the code with pdb, save it in a file (e.g., debug_example.py), then run it using the Python interpreter:

bash Copy code

```bash
python debug_example.py
```

When the program execution reaches `pdb.set_trace()`, you'll see a `pdb` prompt (`(Pdb)`), where you can start entering debugger commands.

Step 3: Debugging Example

At the `(Pdb)` prompt, you can use the following commands:

1. **Inspecting variables**:

bash Copy code

```bash
(Pdb) p x
10
(Pdb) p y
'20'
```

2. **Step into the `add_numbers` function**:

bash Copy code

```bash
(Pdb) step
> /path/to/your/code.py(4)add_numbers()
-> return a + b
```

Now you are inside the `add_numbers` function.

3. **Inspect the arguments**:

bash Copy code

```bash
(Pdb) a
a = 10
b = '20'
```

Here, you can see that `b` is a string, which will cause an error when you try to add it to `a`, which is an integer.

4. **Continue execution until the error**:

```bash
(Pdb) continue
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The program resumes until the TypeError is raised, and you see the error message.

5. **Exiting the debugger**: If you want to stop debugging, you can type:

```bash
(Pdb) quit
```

Common pdb Commands in Action

Here's a list of some commonly used commands in pdb, with examples:

Command	Description	Example
`p expression`	Prints the value of the given expression or variable.	`p x`
`step` or `s`	Step into the next line, even into called functions.	`(Pdb) s`
`next` or `n`	Go to the next line, but don't step into functions.	`(Pdb) n`
`continue` or `c`	Continue execution until the next breakpoint or error.	`(Pdb) c`
`args` or `a`	Print the argument list of the current function.	`(Pdb) a`
`list` or `l`	List the lines of code around the current execution point.	`(Pdb) l`
`where` or `w`	Print the current position, including the call stack (shows where you are in the program).	`(Pdb) w`
`quit` or `q`	Exit the debugger and terminate the program.	`(Pdb) q`
`break line_no`	Set a breakpoint at the specified line number.	`break 10`
`clear`	Remove all breakpoints.	`(Pdb) clear`

Command	Description	Example
p expression	Prints the value of the given expression or variable.	p x
step or s	Step into the next line, even into called functions.	(Pdb) s
next or n	Go to the next line, but don’t step into functions.	(Pdb) n
continue or c	Continue execution until the next breakpoint or error.	(Pdb) c
args or a	Print the argument list of the current function.	(Pdb) a
list or l	List the lines of code around the current execution point.	(Pdb) l
where or w	Print the current position, including the call stack (shows where you are in the program).	(Pdb) w
quit or q	Exit the debugger and terminate the program.	(Pdb) q
break line_no	Set a breakpoint at the specified line number.	break 10
clear	Remove all breakpoints.	(Pdb) clear

Advanced Debugging Techniques

1. **Setting Breakpoints Without set_trace()**: You can set breakpoints without using pdb.set_trace() by invoking the debugger at runtime. For example:

```bash
python -m pdb your_script.py
```

This will start the script in the pdb debugger from the beginning.

2. **Breakpoints in Specific Lines**: You can set breakpoints on specific lines before running the code.

Example:

```bash
(Pdb) break 12  # Set a breakpoint at line 12
```

3. **Conditional Breakpoints**: You can add conditions to breakpoints, so the debugger will only stop if a certain condition is met.

Example:

```bash
(Pdb) break 12, x > 5  # Break at line 12 only if x > 5
```

Debugging Best Practices

1. **Start with `pdb.set_trace()`**: Set breakpoints in your code using `pdb.set_trace()` where you suspect issues, so you can analyze variables and flow at that point.
2. **Use Step (`s`) and Next (`n`) Effectively**:
   - Use step to investigate function internals and understand detailed flow.
   - Use next to skip over function calls when you're only interested in high-level logic.
3. **Modify Variables On-The-Fly**: You can modify variables during debugging to test different scenarios without restarting the program.

Example:

```bash
(Pdb) x = 20  # Change the value of x
```

4. **Use `continue` to Skip Long Debug Sessions**: If you're confident the code until a certain point is fine, use `continue` to skip ahead to the next breakpoint or error.

Conclusion

The pdb debugger is a powerful tool for diagnosing issues in Python code, providing interactive control over execution and inspection of variables. It’s especially helpful when dealing with complex code or when traditional print statements don't provide enough insight.