MODULE 4 PYTHON

Introduction to python Programming

Syllabus: Organizing Files: The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,**Debugging:** Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE‟s Debugger.
Textbook 1: Chapters 9-10

1. ORGANIZING THE FILES

Python can automate file management tasks like copying, renaming, moving, or compressing files to save time and avoid errors. Examples include:

- Copying all PDF files from multiple folders.
- Renaming files, like removing leading zeros.
- Creating ZIP backups of folders.

To see file extensions (e.g., .txt, .pdf), make sure they're visible in your file browser. In Windows, enable this via Control Panel > Appearance > Folder Options > View > Uncheck "Hide extensions for known file types."

2. THE Shutil MODULE

The shutil module in Python provides functions to handle file and folder operations like copying, moving, renaming, and deleting files. Here's a simple explanation with examples:

shutil.copy(source, destination)

- This function copies a file from the source path to the destination path.
- If a folder is specified as the destination, the original filename is used for the new copied file.
- Example

```
import shutil, os
os.chdir('C:\\')
shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
```

- Output: `C:\\delicious\\spam.txt`

shutil.copytree(source, destination)

This function copies an entire folder along with all its subfolders and files to a new location.

Example

```
import shutil, os
os.chdir('C:\\')
shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
```

Moving and Renaming Files and Folders

shutil.move(source, destination)

- Moves a file or folder from source to destination.
- If destination is a folder, the file is moved into that folder.

Example:

```
import shutil
shutil.move('C:\\bacon.txt', 'C:\\eggs')
```
Output: C:\\eggs\\bacon.txt

Renaming Files You can move and rename files at the same time.

Example:

```
shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
```

Output: C:\\eggs\\new_bacon.txt

Issues to Watch For:

If the destination folder doesn't exist, an error occurs.

```
shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
```

Output: Error because the folder doesn't exist.

Be careful not to overwrite files accidentally.

Example:

```
shutil.move('C:\\bacon.txt', 'C:\\eggs')
```

If the eggs folder exists, the file bacon.txt will be moved there. If a file with the same name already exists, it will be overwritten.

Permanently Deleting Files and Folders

os.unlink(path)

Deletes a single file.

os.rmdir(path)

Deletes an empty folder.

shutil.rmtree(path)

Deletes a folder and all its contents (files and subfolders).

Example:

```
import os

os.unlink('file.txt')  # Deletes a file

os.rmdir('empty_folder')  # Deletes an empty folder

shutil.rmtree('folder')  # Deletes a folder and all its contents
```

Caution!

Be careful when deleting files or folders. Always check what will be deleted first to avoid accidental loss of important data. Use print statements to review the files before deleting them

2. WALKING A DIRECTORY TREE

The concept of walking a directory tree in Python involves going through each folder, its subfolders, and files within them. This allows you to perform tasks such as renaming, copying, deleting, or organizing files systematically.

os.walk() Function

The os.walk() function helps traverse the directory tree. It takes the path of a folder and returns:

1. Current folder's name.
2. List of subfolders.
3. List of files in the current folder.

Example Scenario

Let's say you have a folder C:\delicious with subfolders and files inside it. You want to print the names of all subfolders and files in each folder.

Example Program:

```
import os
```

```
for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)
    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': ' + filename)
    print('')
```

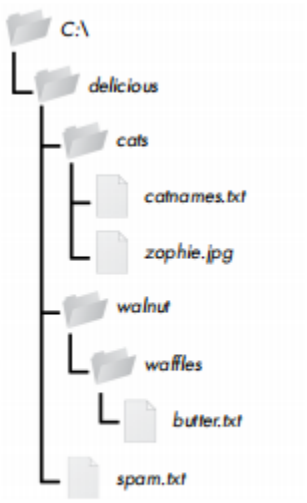

Figure 9-1: An example folder that contains three folders and four files

Output:

- The current folder is C:\delicious
    - SUBFOLDER OF C:\delicious: cats
    - SUBFOLDER OF C:\delicious: walnut
    - FILE INSIDE C:\delicious: spam.txt
- The current folder is C:\delicious\cats
    - FILE INSIDE C:\delicious\cats: catnames.txt
    - FILE INSIDE C:\delicious\cats: zophie.jpg
- The current folder is C:\delicious\walnut
    - SUBFOLDER OF C:\delicious\walnut: waffles
    - The current folder is C:\delicious\walnut\waffles
        - FILE INSIDE C:\delicious\walnut\waffles: butter.txt

Explanation

- os.walk('C:\\delicious') goes through each subfolder and file within C:\delicious.
- It provides:
    - Current folder (folderName).
    - Subfolders (subfolders).
    - Files (filenames).

You can use the information from os.walk() to perform actions on each folder or file as needed.

3. COMPRESSING FILES WITH THE ZIPFILE MODULE

ZIP files are compressed files that can contain multiple files and subfolders. Python provides functions in the zipfile module to create, read, extract, and modify ZIP files.

1. Reading ZIP Files

To work with a ZIP file, you need to create a ZipFile object:

```
import zipfile, os
os.chdir('C:\\')  # Move to folder with ZIP file
exampleZip = zipfile.ZipFile('example.zip')
```

- exampleZip.namelist() lists all files and folders inside the ZIP.
- exampleZip.getinfo('spam.txt') gives details like file size and compressed size.

You can calculate how much space is saved by compression:

```
'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo.compress_size, 2))
```

2. Extracting from ZIP Files To extract files from a ZIP file:

```
exampleZip.extractall()  # Extract all files to current directory
exampleZip.extract('spam.txt', 'C:\\new_folder')  # Extract specific file
```

extractall() extracts everything.

extract() extracts a specific file and optionally puts it in a different folder.

3. Creating and Adding to ZIP Files To create a new ZIP file and add files:

```
newZip = zipfile.ZipFile('new.zip', 'w')  # 'w' means write mode
newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
newZip.close()
```

The write() method compresses and adds files into the ZIP file.

compress_type=zipfile.ZIP_DEFLATED uses a common compression algorithm.

4. Appending to an Existing ZIP File To add files to an existing ZIP file:

```
newZip = zipfile.ZipFile('new.zip', 'a')  # 'a' means append mode
newZip.write('eggs.txt', compress_type=zipfile.ZIP_DEFLATED)
```

```
newZip.close()
```

Append mode ('a') allows adding new files without deleting existing content.

Conclusion

- ZIP Files: Compressed files that hold multiple files and folders.
- ZipFile Object: Acts like a File object, allowing interaction with ZIP files (reading, writing, extracting).
- extractall(): Extracts all files from ZIP.
- extract(): Extracts specific files.
- write(): Adds a file to a ZIP file with compression.
- append mode: Adds files to an existing ZIP file without removing current contents.

4. PROJECT: RENAMING FILES WITH AMERICAN-STYLE DATES TO EUROPEAN-STYLE DATES SAY YOUR BOSS EMAILS

This project involves writing a Python program to rename files with American-style dates (MM-DD-YYYY) in their filenames to European-style dates (DD-MM-YYYY).

Step-by-Step Explanation:

Step 1: Create a Regex for American-Style Dates

The first part of the program involves creating a regex pattern to match filenames with American-style dates.

1. Import necessary modules:

```
import shutil, os, re
```

2. Create a Regex:

```
datePattern = re.compile(r"""^(.*?) # all text before the date

 ((0|1)?\d)- # one or two digits for the month

 ((0|1|2|3)?\d)- # one or two digits for the day

 ((19|20)\d\d) # four digits for the year

 (.*?)$ # all text after the date

""", re.VERBOSE)
```

Explanation:

`^(.*?)` matches everything before the date.

`((0|1)?\d)` matches the month, with optional leading zeros or `0` or `1` in some cases.

`((0|1|2|3)?\d)` matches the day, following similar logic.

`((19|20)\d\d)` matches years from 1900 to 2099.

`(.*?)$` matches everything after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program loops over the filenames in the current working directory and matches them against the regex pattern.

1. Loop through filenames:

```
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)
```

2. **Skip files without dates**:

```
if mo == None:
    continue
```

3. Extract different parts of the filename:

```
beforePart = mo.group(1)
monthPart = mo.group(2)
dayPart = mo.group(4)
yearPart = mo.group(6)
afterPart = mo.group(8)
```

1. Each part is matched using a corresponding group in the regex.

Step 3: Form the New Filename and Rename the Files

Once you have the different parts, concatenate them to form the European-style date format.

Concatenate the parts:

```
euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart + afterPart
```

Get the absolute paths:

absWorkingDir = os.path.abspath('.')

amerFilename = os.path.join(absWorkingDir, amerFilename)

euroFilename = os.path.join(absWorkingDir, euroFilename)

Rename the files:

shutil.move(amerFilename, euroFilename)

Testing the Program

Before actually renaming the files, you can print the filenames to confirm they will be renamed correctly:

print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))

Uncomment the `shutil.move()` line after confirming the results.

Example Workflow:

1. The program searches for files like 03-25-2025-report.docx and renames them to 25-03-2025-report.docx.
2. It skips files that do not contain dates in the American format.
3. The new filenames are then moved/renamed.

Similar Programs:

This concept can be expanded to other tasks like:

- Adding a prefix to filenames (spam_eggs.txt → spam_eggs.txt).
- Changing European-style dates to American-style dates.
- Removing leading zeros from filenames (spam0042.txt → spam42.txt).

5. DEBUGGING

Debugging means finding and fixing mistakes (called bugs) in your code. Bugs are errors that cause your program to behave in unexpected ways. Debugging helps you figure out what's going wrong so you can fix it.

Scenario: Making a Peanut Butter Sandwich (only for understanding purpose)

Imagine you're writing instructions for a robot to make a peanut butter sandwich. You give it these steps:

1. Get two slices of bread.
2. Spread peanut butter on one slice.

3. Put the other slice on top.

But when you run the program, the robot puts the peanut butter on the outside of the sandwich instead of inside! Something went wrong, so now you need to debug.

Debugging Tools: To debug your program, you have **tools and techniques** to help you:

1. Logging: Think of logging as leaving notes while the robot works. For example:

- After Step 1, leave a note: “Got the bread.”
- After Step 2, leave a note: “Spread peanut butter.”
- After Step 3, leave a note: “Sandwich done.”

When the sandwich looks wrong, you can read the notes to see where the problem happened. For instance, if the note says: “Spread peanut butter on both sides,” you’ll know that Step 2 needs fixing.

2. Assertions: Assertions are checkpoints in your program. They stop the robot if something goes wrong. For example:

- After Step 2, add an assertion: "Check that peanut butter is on **one** slice of bread, not both.”

If the robot spreads peanut butter on both slices, the assertion will catch the mistake immediately, so you can fix it early.

3. Debugger: The debugger works like a super-slow robot. Instead of doing all the steps quickly, it goes **one step at a time**. This way, you can check exactly what the robot is doing and what the sandwich looks like after each step. It’s like watching the robot in slow motion to see where it messes up.

Why Debugging Helps

Without debugging tools, you’d be guessing why your sandwich looks wrong. With them, you can find out exactly where the robot misunderstood your instructions and fix it without frustration. Debugging is a normal and important part of programming—everyone makes mistakes, even experts

6. RAISING EXCEPTIONS

When you write a program, sometimes things don’t go as planned—someone might give wrong inputs or do something unexpected. **Exceptions** are Python’s way of stopping and saying, "Hey, something’s not right!" You can handle these exceptions to prevent the program from crashing, and you can even create your own exceptions to stop the program when specific rules aren’t followed.

Scenario: Making a Cake(for understanding purpose)

Imagine you’re writing a recipe for a cake. Here’s what you want to do:

1. You need exactly **one type of flour**.
2. The oven temperature should be more than **300°F**.

3. The baking time should be more than **15 minutes**.

If someone gives wrong information, you want to stop and say what went wrong, instead of baking a bad cake.

How to Raise Exceptions

You can create rules for your recipe and raise exceptions if those rules aren't followed. For example:

```
def bakeCake(flour, temperature, time):
  if len(flour) != 1:
    raise Exception("You must choose exactly one type of flour!")
  if temperature <= 300:
    raise Exception("The oven temperature must be greater than 300°F!")
  if time <= 15:
    raise Exception("The baking time must be longer than 15 minutes!")
   print("Cake is baking!")
```

Here's how this works:

1. If there's more than one type of flour, the program stops and says: **"You must choose exactly one type of flour!"**
2. If the temperature is too low, it says: **"The oven temperature must be greater than 300°F!"**
3. If the baking time is too short, it says: **"The baking time must be longer than 15 minutes!"**

Handling Exceptions with try and except

Now, if you're trying out different combinations of ingredients, you don't want the program to crash every time. Instead, you can use try and except to catch the exceptions and print friendly error messages.

```
for flour, temp, time in [("all-purpose", 350, 20), ("", 400, 30), ("whole-wheat, almond", 200, 25)]:

  try:
    bakeCake(flour, temp, time)
  except Exception as err:
    print("An error occurred:", err)
```

What Happens When You Run This

1. The first combination works fine:
    - Flour: "all-purpose," Temperature: 350°F, Time: 20 minutes.
    - Output: **"Cake is baking!"**
2. The second combination is missing flour:
    - Output: **"An error occurred: You must choose exactly one type of flour!"**
3. The third combination has multiple types of flour and low temperature:
    - Output: **"An error occurred: You must choose exactly one type of flour!"**

Why Raise Exceptions?

Using exceptions lets you **enforce rules** in your program. If something isn't right, you can stop, explain the problem, and prevent bad results. By using try and except, you can handle these errors gracefully and keep the program running for other tasks

7. GETTING THE TRACEBACK AS A STRING

When Python encounters an error (called an exception), it provides a detailed report called a traceback. The traceback tells you:

1. What went wrong (the error message).
2. Where it happened (the file and line number).
3. How it got there (a list of function calls leading to the error, called the call stack).

You can see the traceback when Python prints it out during an error, but you can also save this information to a file for later debugging using the traceback module.

Scenario: A Cookie-Baking Program (for understanding purpose)

Imagine you're writing a program that calls multiple functions to bake cookies. If something goes wrong, you want to know exactly which step caused the problem. Here's the setup:

Program

1. A startBaking function starts the process.
2. A mixIngredients function mixes the ingredients.
3. A bake function bakes the cookies.

```
def startBaking():
   mixIngredients()
def mixIngredients():
   bake()
def bake():
   raise Exception("Oven temperature is too low!")
 startBaking()
```

Traceback: What Happens When It Breaks When you run this code, Python crashes and shows something like this:

```
Traceback (most recent call last):
  File "cookieProgram.py", line 9, in <module>
    startBaking()
  File "cookieProgram.py", line 2, in startBaking
    mixIngredients()
  File "cookieProgram.py", line 5, in mixIngredients
```

```
  bake()
 File "cookieProgram.py", line 8, in bake
  raise Exception("Oven temperature is too low!")
Exception: Oven temperature is too low!
```

This tells you:

1. The error message is: "Oven temperature is too low!"
2. The error happened in the bake function (line 8).
3. The call stack shows how the program got there:
   startBaking → mixIngredients → bake.

Getting and Saving the Traceback

Instead of crashing the program, you can catch the error and save the traceback to a file using the traceback module. For example:

```
import traceback
try:
  startBaking()
except Exception as e:
  with open('errorLog.txt', 'w') as errorFile:
    errorFile.write(traceback.format_exc())
  print("An error occurred, but the details have been saved to errorLog.txt.")
```

How It Works:

1. try block: Runs the code and watches for errors.
2. except block: Catches the error when it happens.
3. traceback.format_exc(): Gets the traceback as a string.
4. Saving to a file: Writes the traceback to a file (e.g., errorLog.txt) so you can look at it later.

Example Output

When you run this updated program, you'll see:

An error occurred, but the details have been saved to errorLog.txt.

If you open errorLog.txt, it will contain the full traceback:

```
Traceback (most recent call last):
 File "cookieProgram.py", line 9, in <module>
  startBaking()
 File "cookieProgram.py", line 2, in startBaking
  mixIngredients()
 File "cookieProgram.py", line 5, in mixIngredients
```

```
  bake()
File "cookieProgram.py", line 8, in bake
  raise Exception("Oven temperature is too low!")
Exception: Oven temperature is too low!
```

Why Is This Useful?

1. **Debug Later:** If the program is used by someone else, they don't see scary error messages. Instead, you get all the details in a log file.
2. **Graceful Handling:** Your program doesn't crash—it keeps running after saving the error.
3. **Track Issues:** The traceback helps you pinpoint exactly where things went wrong, making debugging much easier.

8. ASSERTIONS

An assertion is like a quick check in your program to make sure something is working as expected. It says:

1. "I expect this to be true."
2. "If it's not true, stop everything and show me what's wrong!"

If the condition in the assertion fails, Python raises an AssertionError and stops running. This helps catch mistakes early before they cause bigger problems later.

How Assertions Work

An assertion is written like this:

assert condition, "Error message if condition is False"

- Condition: Something that should always be true (like x > 0 or status == "open").
- Error message: A helpful message to explain what went wrong.

Example: Airplane Door Simulation

Imagine you're coding a simulation of an airplane's pod bay doors. You want to ensure the doors are always "open" before allowing passengers to board.

```
podBayDoorStatus = 'open'  # The doors are open
assert podBayDoorStatus == 'open', 'The pod bay doors must be "open".'
podBayDoorStatus = 'closed'  # Someone closed the doors!
assert podBayDoorStatus == 'open', 'The pod bay doors must be "open".'
```

If the doors are not open, Python will stop running and show this:

```
AssertionError: The pod bay doors must be "open".
```

This helps you immediately notice the problem and fix it.

When to Use Assertions

- Use assertions for sanity checks during development.
- Assertions are for programmer errors, not user input errors.
    - Good for: Ensuring internal logic is correct.
    - Not for: Handling invalid user input (use exceptions for that).

Traffic Light Simulation Example

Imagine you're building a traffic light simulation where:

1. North-south (NS) and east-west (EW) lights must always follow traffic rules.
2. One light must always be red to avoid accidents.

Code to Switch Traffic Lights

```
def switchLights(stoplight):
   for key in stoplight.keys():
      if stoplight[key] == 'green':
         stoplight[key] = 'yellow'
      elif stoplight[key] == 'yellow':
         stoplight[key] = 'red'
      elif stoplight[key] == 'red':
         stoplight[key] = 'green'
  # Assert that at least one light is red
   assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

Example Traffic Light

```
market_2nd = {'ns': 'green', 'ew': 'red'}

switchLights(market_2nd)
```

What Happens?

1. When you switch the lights, they change like this:
    - 'ns': 'green' becomes 'yellow'.
    - 'ew': 'red' becomes 'green'.
2. Now, no light is red!
3. The assertion catches this mistake and stops the program immediately:

```
AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

This prevents potential accidents in the simulation by ensuring the logic is fixed right away.

Why Use Assertions?

1. Catch Bugs Early: They help detect logic errors before they cause bigger problems.
2. Fail Fast: If something's wrong, the program stops immediately, pointing you to the issue.
3. Save Debugging Time: You know exactly where the problem is and what caused it.

Disabling Assertions

When you're done testing, you can disable assertions by running Python with the -O option (optimized mode). This skips all assertions to improve performance in the final product.

Conclusion

- Assertions are sanity checks to ensure your program logic is correct.
- They're for development, not for handling user errors.
- Use assertions to fail fast and debug more effectively.
- Example: Ensure one traffic light is always red to prevent simulation accidents.

9.LOGGING

Logging is a way to track events or messages from your program as it runs. It helps programmers see what the program is doing, find issues, and understand its behavior. Instead of just printing messages using print(), logging gives a more structured and organized way to collect information about the program's execution.

Concept of Logging in Simple Terms

1. **Why Use Logging?**
    - Imagine you are debugging a program and want to know what value a variable has at a specific point. Instead of printing this information with print(), you can log it.
    - Logging also shows the sequence in which events happen, helping you catch problems in your code.
2. **How Does It Work?**
    - You "log" a message using Python's logging module.
    - Each log message gets automatically timestamped and labeled (e.g., DEBUG, INFO, WARNING) so you can understand its context.
3. **Levels of Logging Logging has levels of importance:**
    - DEBUG: Detailed information, usually for debugging.
    - INFO: General messages about program progress.
    - WARNING: Something unexpected happened, but the program is still running.
    - ERROR: A problem occurred, but the program can still continue.
    - CRITICAL: A serious error occurred, and the program may not recover.
4. **Flexible Control**
    - You can choose what level of messages to show. For example:
        - Show everything (DEBUG level).
        - Show only errors and critical problems (ERROR level).
    - Logging can be disabled altogether once the program works as expected.

5. Logging to a File

- Instead of showing log messages on the screen, you can save them in a text file for later review.

Example Scenario Imagine This Situation:You're writing a program that calculates the factorial of a number. However, there's a bug in your code. You want to find where the error occurs without printing hundreds of messages manually.

Code Example

```
import logging
# Setup logging to show DEBUG messages and format them
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
# Start of program
logging.debug('Start of program')
def factorial(n):
    logging.debug(f'Start of factorial({n})')  # Log the start of the function
    total = 1
    for i in range(1, n + 1):  # Corrected the range to start from 1
        total *= i
        logging.debug(f'i is {i}, total is {total}')  # Log each step in the loop
    logging.debug(f'End of factorial({n})')
    return total
result = factorial(5)  # Calculate factorial of 5
print(result)  # Show result
logging.debug('End of program')
```

Table 10-1: Logging Levels in Python

Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.

Table 10-1 *(continued)*

Level	Logging Function	Description
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

How Logging Helps

1. If the program doesn't work as expected, the log messages help pinpoint the issue.
    - **Example:** If i starts from 0 instead of 1, the log will show i = 0, causing incorrect results.
2. You can save logs to a file for review without cluttering your screen:

logging.basicConfig(filename='logfile.txt', level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')

Why Logging is Better Than print()

- More organized: Logs show timestamps and levels of importance.
- Easily disable or enable: Use logging.disable(logging.CRITICAL) to turn off all logs.
- Keeps code clean: No need to remove debug messages manually.
- Saves time: Provides detailed insights into program execution.

Logging is like having a diary of what your program does, helping you find and fix problems much faster

10. IDLE'S DEBUGGER

The IDLE Debugger is a tool in Python's IDLE environment that helps you find and fix errors in your code by running it one line at a time. While using the debugger, you can check the values of variables at different points in your program to understand what's happening.

Figure 10-1: The Debug Control window

Concept of IDLE's Debugger in Simple Terms

1. **What Does It Do?**
    - It pauses your program after each line of code.
    - You can then examine the state of your program (e.g., the values of variables).
    - It lets you control how the program runs (line-by-line or skipping over certain sections).
2. **Why Use It?**
    - To find bugs: If something in your program isn't working, the debugger helps you figure out where the problem is.

- To understand code flow: It shows the order in which your code is executed.

3. How It Works:

- When you enable the debugger, it shows:
  - The next line of code to be executed.
  - Local variables: Variables inside the current function.
  - Global variables: Variables defined outside of functions or available to the whole program.
- It pauses until you tell it what to do next.

4. Debugging Buttons:

- Go: Runs the program normally until it ends or hits a breakpoint (a marker where the program pauses).
- Step: Runs the next line of code and pauses again.
- Over: Runs the next line of code, skipping over the details of any function calls.
- Out: Runs the code until the current function is finished.
- Quit: Stops the debugger and the program.

Example Scenario

Imagine you wrote a program to calculate the average of three numbers:

```
def calculate_average(a, b, c):
    total = a + b + c
    average = total / 3
    return average
result = calculate_average(4, 5, 6)
print("The average is:", result)
```

But it gives the wrong output. To debug this, you can use the IDLE Debugger.

Steps to Debug Using IDLE

1. Open your program in IDLE.
2. Click **Debug > Debugger** in the IDLE menu to enable the debugger.
3. Run your program. The Debug Control window appears.
4. Use the buttons:
   - **Step**: To see each line of code execute. You can check the values of total and average after each step.
   - **Over**: If you don't need to see inside built-in functions (like print()).
   - **Out**: To exit a function after stepping into it.
5. Check the variable values in the **Locals** or **Globals** sections of the Debug Control window.

Why It's Useful

- If total has the wrong value, you'll catch the error as soon as the total = a + b + c line runs.
- If the average is calculated incorrectly, you'll see it immediately after the division step.

Using IDLE's Debugger lets you find errors without guessing, making debugging faster and easier

The IDLE debugger lets you control how your program runs and pauses. Here's what each button does:

1. Go

- **What it does**: Runs your program **normally** until it finishes or reaches a **breakpoint** (a special pause point you set in your code).
- **When to use it**: When you're done stepping through the code and want to see the rest of the program run without interruptions.

2. Step

- **What it does**: Runs the **next line of code** and then pauses.
  - If the next line is a function call, the debugger **steps into** the function to show you the first line of its code.
- **When to use it**: When you want to see **every single line** of code being executed, including what happens inside functions.

3. Over

- **What it does**: Runs the **next line of code**, but if it's a function call, it **skips over** the details inside the function. The function runs, but the debugger pauses after the function finishes.
- **When to use it**: When you don't need to see the details inside a function (e.g., you know how print() works, so you skip its internal code).

4. Out

- **What it does**: Runs the program until it finishes the **current function** and pauses once it's back to the main code.
- **When to use it**: If you've stepped into a function by mistake or are done inspecting it and want to return to the main program.

5. Quit

- **What it does**: Stops the debugger and **ends the program** immediately.
- **When to use it**: If you don't want to debug anymore or need to stop the program right away.

```
def add_numbers(a, b):
    return a + b
x = 5
y = 10
result = add_numbers(x, y)
print(result)
```

- If you click **Step**, the debugger will pause at each line (even inside add_numbers).
- If you click **Over** at result = add_numbers(x, y), it will skip the details of add_numbers and pause after it returns the value.
- If you're inside the add_numbers function and click **Out**, the debugger will finish running the function and return to the main program.
- If you're done with debugging, click **Go** to run the rest of the program normally.
- If you need to stop the program, click **Quit**.

Figure 10-2: The Debug Control window when the program first starts under the debugger

Debugging the Number Adding Program:

The Problem: The code asks for three numbers, but it **joins them as text** instead of adding them as numbers.

Example output:

Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342

Solution:
Use the **debugger** to find the issue by running the program step-by-step.

How to Debug:

- Enable **Debug Mode** in IDLE (click Debug > Debugger, check all boxes).
- Run the program (press **F5**).
- The program **pauses** at each line of code, letting you see the value of variables.

What You'll See:

- When entering numbers, you'll notice they're stored as **strings** (text).
- The bug happens because the program **concatenates** strings (`'5' + '3' + '42'`) instead of adding numbers.

Fix:
Convert inputs to numbers using `int()`:

```
first = int(input('Enter the first number to add: '))
second = int(input('Enter the second number to add: '))
third = int(input('Enter the third number to add: '))
print('The sum is', first + second + third)
```

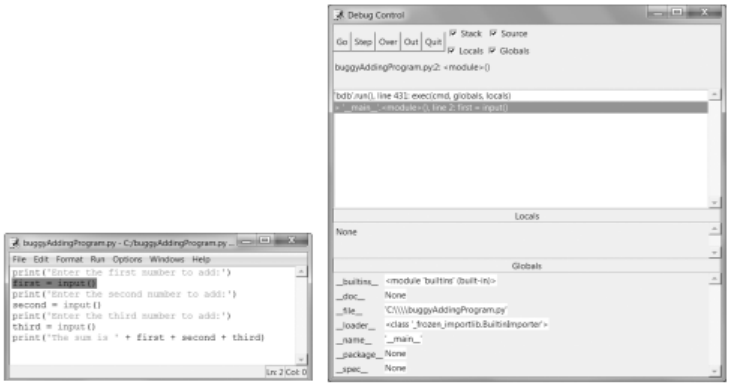

Figure 10-3: The Debug Control window after clicking Over

Debugging with "Over" and Breakpoints

Using "Over"

1. Click **Over** to move step-by-step through the program.
2. When the program reaches `input()`, the debugger pauses, and you type your input (e.g., 5).
3. Repeat this for the next inputs (e.g., 3 and 42).
4. By the end, you'll notice the variables are **strings** ('5', '3', '42'), causing the bug when concatenating them.

Breakpoints

- **What are they?**
  A breakpoint is a marker you place on a line of code to make the debugger **pause** only when it reaches that line.

- **Why use them?**
  They save time by letting the program run normally until the code you're interested in.
- **Example with Coin Flip Program**:
  - Add a breakpoint on `print('Halfway done!')`.
  - The debugger will skip all earlier steps and pause only when 500 coin flips are complete.
  - From there, you can check variable values or step through the code.
- **How to Set/Remove Breakpoints**:
  - **Set**: Right-click a line in the editor > Choose **Set Breakpoint** (line turns yellow).
  - **Remove**: Right-click the line > Choose **Clear Breakpoint** (yellow disappears).

Figure 10-4: The Debug Control window on the last line. The variables are set to strings, causing the bug.

Figure 10-5: Setting a breakpoint