MODULE-4

Organizing Files

Dr.Sandeep Bhat
Professor,CSE Dept.
SIT,Mangaluru

THE SHUTIL MODULE

- The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs.

- To use the shutil functions, you will first need to use import shutil.

Copying Files and Folders

- The shutil module provides functions for copying files, as well as entire folders.
- Calling **shutil.copy**(*source*, *destination*) will copy the file at the path *source* to the folder at the path *destination*.
- (Both *source* and *destination* can be strings or Path objects.)
- If *destination* is a filename, it will be used as the new name of the copied file.
- This function returns a string or Path object of the copied file.

Eg.

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
❶ >>> shutil.copy(p / 'spam.txt', p / 'some_folder')
'C:\\Users\\Al\\some_folder\\spam.txt'
❷ >>> shutil.copy(p / 'eggs.txt', p / 'some_folder/eggs2.txt')
WindowsPath('C:/Users/Al/some_folder/eggs2.txt')
```

Cont..

- The first **shutil.copy()** call copies the file at *C:\Users\Al\spam.txt* to the folder *C:\Users\Al\some_folder*.
- The return value is the path of the newly copied file.
- Note that since a folder was specified as the destination, the original *spam.txt* filename is used for the new, copied file's filename.
- The second **shutil.copy()** call also copies the file at *C:\Users\Al\eggs.txt* to the folder *C:\Users\Al\some_folder* but gives the copied file the name *eggs2.txt*.

Moving and Renaming Files and Folders

- Calling **shutil.move**(*source*, *destination*) will move the file or folder at the path *source* to the path *destination* and will return a string of the absolute path of the new location.

- If *destination* points to a folder, the *source* file gets moved into *destination* and keeps its current filename.

Eg.

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

Cont..

- Assuming a folder named *eggs* already exists in the C:\ directory, this **shutil.move()** call says, "Move C:*bacon.txt* into the folder C:*eggs*."
- If there had been a *bacon.txt* file already in C:*eggs*, it would have been overwritten.
- Since it's easy to accidentally overwrite files in this way, you should take some care when using **move().**
- The *destination* path can also specify a filename. In the following example, the *source* file is moved *and* renamed.

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module.

- Calling `os.unlink(path)` will delete the file at *path*.
- Calling `os.rmdir(path)` will delete the folder at *path*. This folder must be empty of any files or folders.
- Calling `shutil.rmtree(path)` will remove the folder at *path*, and all files and folders it contains will also be deleted.

Cont..

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    #os.unlink(filename)
    print(filename)
```

Cont..

- Now the **os.unlink()** call is commented, so Python ignores it.
- Instead, you will print the filename of the file that would have been deleted.
- Running this version of the program first will show you that you've accidentally told the program to delete ***.rxt*** files instead of ***.txt*** files.

Safe Deletes with the send2trash Module

- Since Python's built-in **shutil.rmtree()** function irreversibly deletes files and folders, it can be dangerous to use.
- A much better way to delete files and folders is with the third-party **send2trash** module.
- You can install this module by running pip install --user send2trash from a Terminal window.

Cont..

- **Using send2trash is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them.**

- **If a bug in your program deletes something with send2trash you didn't intend to delete, you can later restore it from the recycle bin.**

Cont..

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a')   # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

Cont..

- In general, you should always use the **send2trash.send2trash()** function to delete files and folders.
- But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does.
- If you want your program to free up disk space, use the **os** and **shutil** functions for deleting files and folders.

Walking a Directory Tree

- Say you want to rename every file in some folder and also every file in every subfolder of that folder.
- That is, you want to walk through the directory tree, touching each file as you go.
- Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Eg.

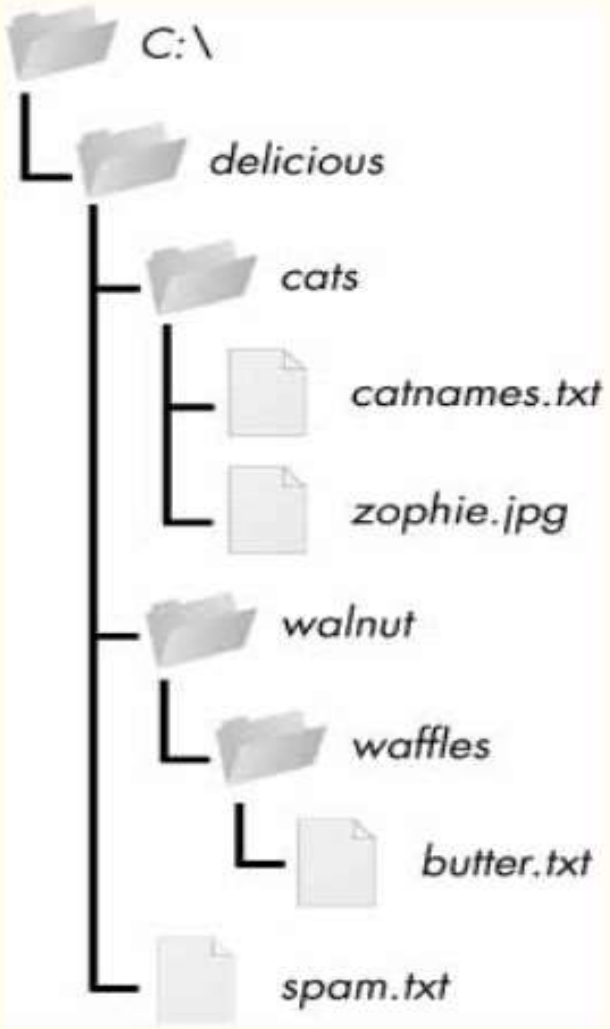


Figure 10-1: An example folder that contains three folders and four files

Eg1..code

```
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('')
```

Cont..

- The **os.walk()** function is passed a single string value: the path of a folder.
- You can use **os.walk()** in a for loop statement to walk a directory tree, much like how you can use the **range()** function to walk over a range of numbers.
- Unlike range(), the **os.walk()** function will return three values on each iteration through the loop:

Cont..

- A string of the current folder's name
- A list of strings of the folders in the current folder
- A list of strings of the files in the current folder

Eg.

```
The current folder is C:\delicious

SUBFOLDER OF C:\delicious: cats

SUBFOLDER OF C:\delicious: walnut

FILE INSIDE C:\delicious: spam.txt
```

Cont..

```
The current folder is C:\delicious\cats

FILE INSIDE C:\delicious\cats: catnames.txt

FILE INSIDE C:\delicious\cats: zophie.jpg
```

Cont..

```
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.
```

Cont..

- Since **os.walk()** returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops.

- Replace the **print()** function calls with your own custom code. (Or if you don't need one or both of them, remove the for loops.)

COMPRESSING FILES WITH THE ZIPFILE MODULE

- The ZIP files (with the *.zip* file extension), can hold the compressed contents of many other files.
- Compressing a file reduces its size, which is useful when transferring it over the internet.
- And since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one.

Cont..

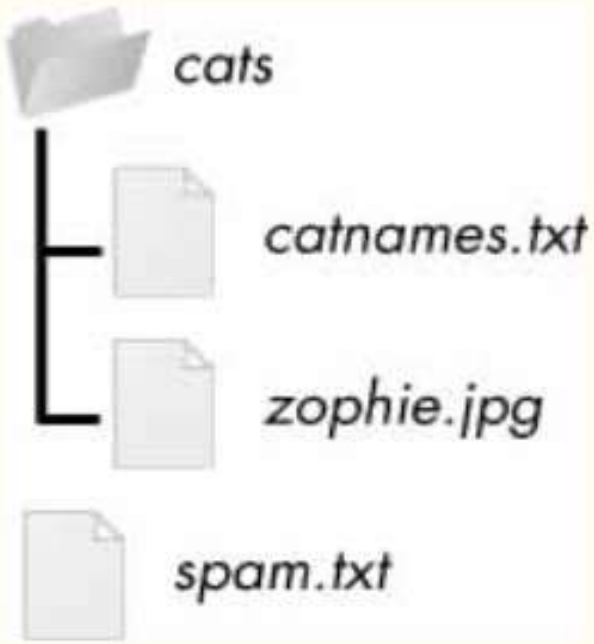


Figure 10-2: The contents of example.zip

Reading ZIP Files

- To read the contents of a ZIP file, first you must create a **ZipFile** object (note the capital letters *Z* and *F*).

- **ZipFile** objects are conceptually similar to the File objects you saw returned by the open() function : they are values through which the program interacts with the file.

- To create a **ZipFile** object, call the **zipfile.ZipFile()** function, passing it a string of the ***.ZIP*** file's filename.

Cont..

```
>>> import zipfile, os


>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
```

Cont..

```
>>> spamInfo.compress_size
3828
>>> f'Compressed file is {round(spamInfo.file_size / spamInfo
.compress_size, 2)}x smaller!'
)
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

Cont..

- A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file.
- These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file.
- ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively.

Extracting from ZIP Files

- The **extractall()** method for **ZipFile** objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

Cont..

- After running this code, the contents of ***example.zip*** will be extracted to C:\. Optionally, you can pass a folder name to **extractall()** to have it extract the files into a folder other than the current working directory.

- If the folder passed to the **extractall()** method does not exist, it will be created.

- The extract() method for ZipFile objects will extract a single file from the ZIP file.

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

Creating and Adding to ZIP Files

- To create your own compressed ZIP files, you must open the ZipFile object in *write mode* by passing 'w' as the second argument.

- (This is similar to opening a text file in write mode by passing 'w' to the open() function.)

Cont..

- When you pass a path to the **write()** method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file.
- The write() method's first argument is a string of the filename to add.
- The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to **zipfile.ZIP_DEFLATED.**
- (This specifies the *deflate* compression algorithm, which works well on all types of data.)

Eg.

```
>>> import zipfile

>>> newZip = zipfile.ZipFile('new.zip', 'w')

>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)

>>> newZip.close()
```

Cont..

- This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.
- Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file.
- If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in *append mode*.

Project: Renaming Files with American-Style Dates to European-Style Dates

- Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY).

- This boring task could take all day to do by hand! Let's write a program to do it instead.

Cont..

Here's what the program does:

1. It searches all the filenames in the current working directory for American-style dates.
2. When one is found, it renames the file with the month and day swapped to make it European-style.

Cont..

This means the code will need to do the following:

1. Create a regex that can identify the text pattern of American-style dates.
2. Call `os.listdir()` to find all the files in the working directory.
3. Loop over each filename, using the regex to check whether it has a date.
4. If it has a date, rename the file with `shutil.move()`.

Step 1: Create a Regex for American-Style Dates

- The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates.

- The to-do comments will remind you what's left to write in this program. Typing them as TODO makes them easy to find using Mu editor's CTRL-F find feature.

Step 2: Identify the Date Parts from the Filenames

- Next, the program will have to loop over the list of filename strings returned from **os.listdir()** and match them against the regex.

- Any files that do not have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables.

Step 3: Form the New Filename and Rename the Files

- As the final step, concatenate the strings in the variables made in the previous step with the European-style date: the date comes before the month.

Project: Backing Up a Folder into a ZIP File

- **Note: This Project was covered in Lab session itself.**

DEBUGGING

- Debugging in Python refers to the process of identifying and fixing errors, bugs, or issues in your code.
- Python provides several tools and techniques to help you debug your code effectively.

Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code.
- Raising an exception is a way of saying, “Stop running the code in this function and move the program execution to the except statement.”
- Exceptions are raised with a raise statement.

➢ In code, a raise statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

Cont..

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

Cont..

- **If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception's error message.**

Eg.

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
      ❶ raise Exception('Symbol must be a single character string.')
    if width <= 2:
      ❷ raise Exception('Width must be greater than 2.')
    if height <= 2:
```

Cont..

```
print(symbol * width)

for i in range(height - 2):

    print(symbol + (' ' * (width - 2)) + symbol)

print(symbol * width)
```

Cont..

```
for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
❹ except Exception as err:
    ❺ print('An exception happened: ' + str(err))
```

Cont..

- Here we've defined a **boxPrint()** function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height.

- This box shape is printed to the screen.

GETTING THE TRACEBACK AS A STRING

- When Python encounters an error, it produces a treasure trove of error information called the ***traceback***.
- The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error.
- This sequence of calls is called the *call* ***stack***.

Cont..

```
def spam():
    bacon()


def bacon():
    raise Exception('This is the error message.')


spam()
```

The error of the program:

- From the traceback, you can see that the error happened on line 5, in the **bacon() function**.
- This particular call to bacon() came from line 2, in the spam() function, which in turn was called on line 7.
- In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

ASSERTIONS

- An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements.

- If the sanity check fails, then an **AssertionError** exception is raised. In code, an assert statement consists of the following:

Cont..

- The `assert` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A comma
- A string to display when the condition is `False`

Cont..

- In plain English, an assert statement says, "I assert that the condition holds true, and if not, there is a bug somewhere, so immediately stop the program.

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]

>>> ages.sort()

>>> ages

[15, 17, 22, 26, 47, 54, 57, 73, 80, 92]

>>> assert

ages[0] <= ages[-1] # Assert that the first age is <= the last age.
```

Cont..

- The assert statement here asserts that the first item in ages should be less than or equal to the last one.
- This is a sanity check; if the code in sort() is bug-free and did its job, then the assertion would be true.
- Because the ages[0] <= ages[-1] expression evaluates to True, the assert statement does nothing.

LOGGING

- If you've ever put a print() statement in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code.
- Logging is a great way to understand what's happening in your program and in what order it's happening.
- Python's logging module makes it easy to create a record of custom messages that you write.

Cont..

- **These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time.**

- **On the other hand, a missing log message indicates a part of the code was skipped and never executed.**

Using the logging Module

```
import logging

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

s -  %(message)s')
```

Cont..

- **import logging:** This line imports the built-in logging module, which provides functionality to create and manage log messages.
- **logging.basicConfig(...)**: This line sets up the basic configuration for logging.
- **level=logging.DEBUG**: This sets the logging level to DEBUG, which means all messages with a severity level of DEBUG and higher will be displayed. Other possible levels include INFO, WARNING, ERROR, and CRITICAL.

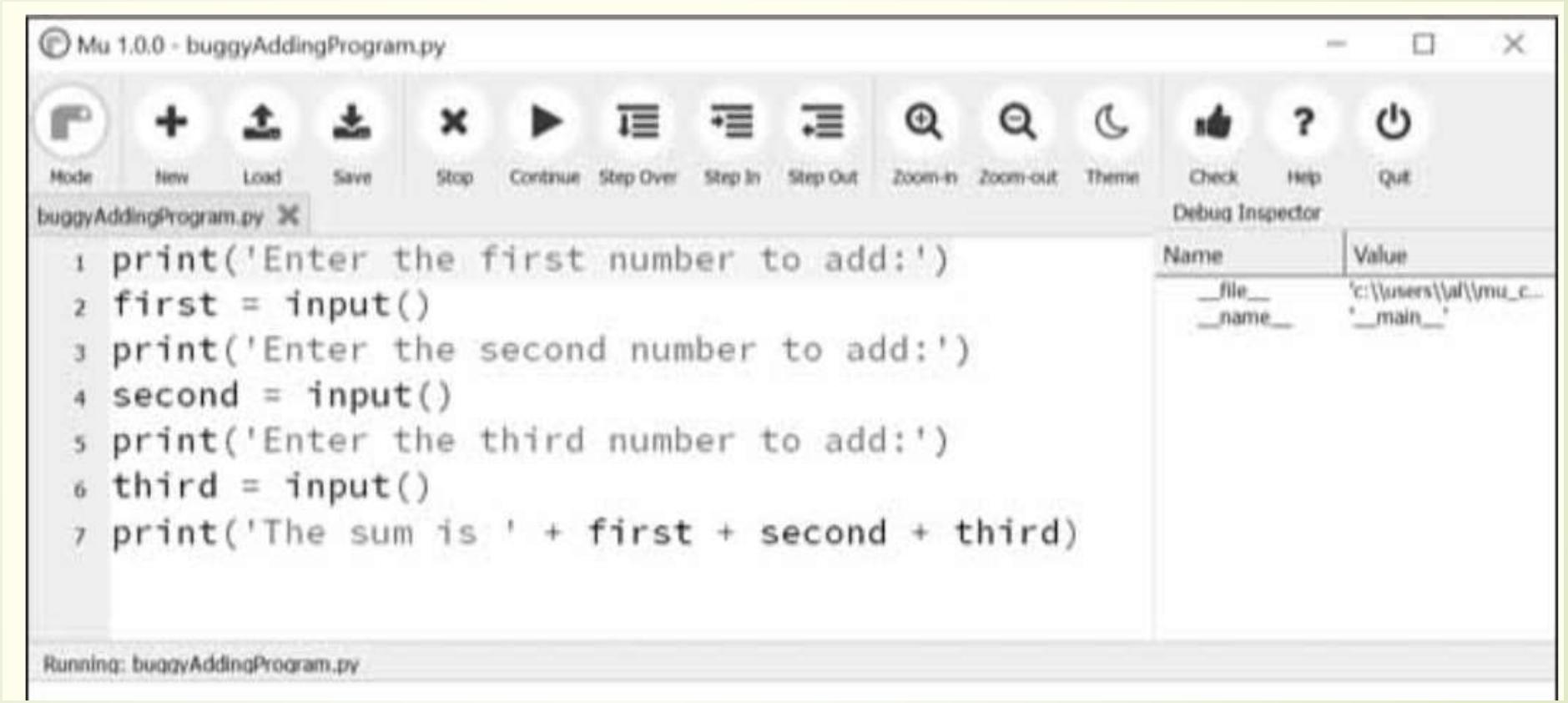
DEBUGGER

- The *debugger* is a feature of the Mu editor, IDLE, and other editor software that allows you to execute your program one line at a time.
- The debugger will run a single line of code and then wait for you to tell it to continue.
- By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime.
- This is a valuable tool for tracking down bugs.

Cont..

- To run a program under Mu's debugger, click the Debug button in the top row of buttons, next to the Run button.
- Along with the usual output pane at the bottom, the Debug Inspector pane will open along the right side of the window.
- This pane lists the current value of variables in your program.
- In Figure 11-1, the debugger has paused the execution of the program just before it would have run the first line of code.

Mu running a program under the debugger



Continue

- Clicking the Continue button will cause the program to execute normally until it terminates or reaches a *breakpoint*.

- If you are done debugging and want the program to continue normally, click the Continue button.

Step In

- Clicking the Step In button will cause the debugger to execute the next line of code and then pause again.
- If the next line of code is a function call, the debugger will “step into” that function and jump to the first line of code of that function.

Step Over

- Clicking the Step Over button will execute the next line of code, similar to the Step In button.
- However, if the next line of code is a function call, the Step Over button will "step over" the code in the function.
- The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns.

Step Out

- Clicking the Step Out button will cause the debugger to execute lines of code at full speed until it returns from the current function.

- If you have stepped into a function call with the Step In button and now simply want to keep executing instructions until you get back out, click the Out button to “step out” of the current function call.

Stop

- If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Stop button.
- The Stop button will immediately terminate the program.

IDLE, which stands for Integrated Development and Learning Environment

- **IDLE, which stands for Integrated Development and Learning Environment, is Python's default integrated development environment that comes bundled with the standard Python distribution.**
- **It includes a debugger that allows you to step through your Python code and analyze its execution, helping you identify and fix errors.**
- **The debugger in IDLE is a useful tool for debugging your code interactively.**

Here's a basic overview of how to use IDLE's debugger for Python:

1. Starting the Debugger:

- **Open IDLE and load the Python script you want to debug.**
- **Click on the "Debug" menu at the top of the IDLE window.**
- **Choose "Debugger" from the dropdown menu, or you can use the keyboard shortcut Ctrl+F5.**

2. Setting Breakpoints:

- Set breakpoints in your code by clicking in the left margin of the code editor next to the line number where you want the debugger to pause.
- Breakpoints indicate where the debugger should halt the execution so you can inspect variables and step through the code.

3. Debugging Controls:

- Once the debugger starts, you'll see a "Debug Control" window.
- This window provides buttons like "Step", "Over", "Out", and "Continue" to control the execution flow of your code.
- The "Step" button allows you to move to the next line of code, "Over" moves to the next line without stepping into functions, and "Out" finishes executing the current function and returns to the caller.
- The "Continue" button lets your code run until the next breakpoint is encountered.

4. Inspecting Variables:

- **You can view the values of variables in the "Variables" window while debugging.**
- **The "Variables" window displays the current state of variables and their values.**

5. Interactive Console:

- **You can interact with the Python console while debugging.**
- **This enables you to execute code snippets and check variable values in real-time.**

6. Exiting Debug Mode:

- **When your code execution reaches the end of the script or when you manually stop debugging, the debugger will exit, and you'll return to the IDLE interface.**