### CHAPTER 1-ORGANISING FILES

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

The `shutil` module in Python is a standard library module that provides various functions for file operations. It offers high-level file operations and utilities, making it easier to work with files and directories.

Here are some commonly used functions in the `shutil` module:

**shutil.copy(src, dst)`:**

This function is used to copy the file from the source (`src`) to the destination (`dst`).

Example:

python

import shutil

# Copy a file

shutil.copy('source\_file.txt', 'destination\_file.txt')

**shutil.copy2(src, dst)`:**

Similar to `shutil.copy()`, this function copies the file from the source to the destination, preserving more metadata such as timestamps.

Example:

python

import shutil

# Copy a file with metadata

shutil.copy2('source\_file.txt', 'destination\_file.txt')

**shutil.move(src, dst)`:**

This function is used to move the file or directory from the source to the destination. It can also be used to rename a file or directory.

Example:

python

import shutil

# Move a file

shutil.move('old\_file.txt', 'new\_file.txt')

**shutil.rmtree(path)`:**

This function is used to delete a directory and all its contents recursively.

Example:

python

import shutil

# Delete a directory and its contents

shutil.rmtree('my\_directory')

**`shutil.make\_archive(base\_name, format, root\_dir)`:**

This function creates an archive file (such as a zip or tar file) from a directory.

Example:

python

import shutil

# Create a zip archive of a directory

shutil.make\_archive('my\_archive', 'zip', 'my\_directory')

**`shutil.rename(src, dst)`:**

The `shutil.rename()` function is used to rename a file or directory. It takes the source path (`src`) and the destination path (`dst`) as arguments. It effectively moves the file or directory to the new destination with the new name.

Example:

python

import shutil

# Rename a file

shutil.rename('old\_file.txt', 'new\_file.txt')

**`shutil.copytree(src, dst)`:**

The `shutil.copytree()` function is used to recursively copy an entire directory and its contents from the source path (`src`) to the destination path (`dst`). It creates a new directory at the destination and copies all the files and subdirectories from the source to the destination.

Example:

python

import shutil

# Copy a directory and its contents

shutil.copytree('source\_directory', 'destination\_directory')

* 1. **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.

import os

for filename in os.listdir():

if filename.endswith('.rxt'):

os.unlink(filename)

* If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted

# import os

for filename in os.listdir():

# if filename.endswith('.rxt'): #os.unlink(filename) print(filename)

* 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.

### 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 send2trash from a Terminal window
* 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.

>>> import send2trash

>>> baconFile = open('bacon.txt', 'a') # creates the file

## >>> baconFile.write('Bacon is not a vegetable.') 25

>>> baconFile.close()

## >>> send2trash.send2trash('bacon.txt')

### Walking a Directory Tree

* 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.

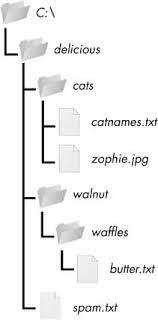


Figure: An example folder that contains three folders and four files

#### 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('')

* 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:

1. A string of the current folder’s name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

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

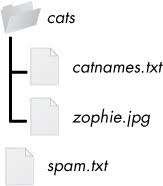
FILE INSIDE C:\delicious\cats: catnames.txt FILE INSIDE C:\delicious\cats: zophie.jpg

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.

### Compressing Files with the zipfile Module

* + Compressing a file reduces its size, which is useful when transferring it over the Internet.
  + since a ZIP file can also contain multiple files and subfolders, it’s a handy way to package several files into one.
  + This single file, called an *archive file*, can then be, say, attached to an email.



*Figure : 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

##### For example

>>> import zipfile, os

>>> os.chdir('C:\\') # move to the folder with example.zip

>>> exampleZip = zipfile.ZipFile('example.zip')

>>> exampleZip.namelist()

['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']

>>> spamInfo = exampleZip.getinfo('spam.txt')

>>> spamInfo.file\_size 13908

>>> spamInfo.compress\_size 3828

>>> 'Compressed file is %sx smaller!' % (round(spamInfo.file\_size / spamInfo

.compress\_size, 2))

'Compressed file is 3.63x smaller!'

>>> exampleZip.close()

* + - 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

>>> os.chdir('C:\\') # move to the folder with example.zip

>>> exampleZip = zipfile.ZipFile('example.zip')

>>> exampleZip.extractall()

>>> exampleZip.close()

* The extract() method for ZipFile objects will extract a single file from the ZIP file. Continue the interactive shell example

>>> 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.
    - 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;

>>> import zipfile

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

>>> newZip.write('spam.txt', compress\_type=zipfile.ZIP\_DEFLATED)

>>> newZip.close()

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

The program does:

* It searches all the filenames in the current working directory for American-style dates.
* When one is found, it renames the file with the month and day swapped to make it European-style.
* This means the code will need to do the following:
* Create a regex that can identify the text pattern of American-style dates.
* Call os.listdir() to find all the files in the working directory.
* Loop over each filename, using the regex to check whether it has a date.
* If it has a date, rename the file with shutil.move().

**Develop a program to backing Up a given Folder (Folder in a current working directory) into a ZIP File by using relevant modules and suitable methods**

import os import zipfile

zf = zipfile.ZipFile("myzipfile.zip", "w")

for dirname, subdirs, files in os.walk("mydirectory"): zf.write(dirname)

for filename in files: zf.write(os.path.join(dirname, filename))

zf.close()

**Develop a Python program to traverse the current directory by listing subfolders and files**

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('')

**Python program to read and print the contents of a text file:**

python

def read\_and\_print\_file(file\_path):

with open(file\_path, 'r') as file:

content = file.read()

print(content)

file\_path = 'example.txt'

read\_and\_print\_file(file\_path)

**Write an algorithm for implement multi clipboard functionality.**

If you’ve responded to a large number of emails with similar phrasing, you’ve probably had to do a lot of repetitive typing. Maybe you keep a text document with these phrases so you can easily copy and paste them using the clipboard. But your clipboard can only store one message at a time, which isn’t very convenient. Let’s make this process a bit easier with a program that stores multiple phrases.

# mclip.py - A multi-clipboard program.

TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",

'busy': """Sorry, can we do this later this week or next week?""",

'upsell': """Would you consider making this a monthly donation?"""}

import sys, pyperclip

if len(sys.argv) < 2:

print('Usage: py mclip.py [keyphrase] - copy phrase text')

sys.exit()

keyphrase = sys.argv[1] # first command line arg is the keyphrase

if keyphrase in TEXT:

pyperclip.copy(TEXT[keyphrase])

print('Text for ' + keyphrase + ' copied to clipboard.')

else:

print('There is no text for ' + keyphrase)

Now that the key phrase is stored as a string in the variable keyphrase, you need to see whether it exists in the TEXT dictionary as a key. If so, you want to copy the key’s value to the clipboard using pyperclip.copy(). (Since you’re using the pyperclip module, you need to import it.) Note that you don’t actually need the keyphrase variable; you could just use sys.argv[1] everywhere keyphrase is used in this program. But a variable named keyphrase is much more readable than something cryptic like sys.argv[1].

**Develop a Python program to traverse the current directory by listing subfolders and files**

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('')

**CHAPTER -2 DEBUGGING**

1. **Explain the role Raising Exceptions with suitable example**
   * Python raises an exception whenever it tries to execute invalid code.
   * *the `raise` statement is used to manually raise an exception during program execution. It allows you to handle exceptional conditions and control the flow of your program based on specific situations.*
   * *When an exception is raised, it interrupts the normal execution of the program and transfers control to the nearest exception handler that can handle the raised exception.*
   * *Here's a small example to illustrate the role of `raise` statement in Python:*

*python*

*def divide(a, b):*

*if b == 0:*

*raise ZeroDivisionError("Cannot divide by zero")*

*return a / b*

*try:*

*result = divide(10, 0)*

*print("Result:", result)*

*except ZeroDivisionError as e:*

*print("Error:", e)*

* + 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
  + For example

>>> 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

* + 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..

def boxPrint(symbol, width, height): if len(symbol) != 1:

raise Exception('Symbol must be a single character string.') if width <= 2:

raise Exception('Width must be greater than 2.') if height <= 2:

raise Exception('Height must be greater than 2.') print(symbol \* width)

for i in range(height - 2):

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

for sym, w, h in (('\*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):

try:

boxPrint(sym, w, h) except Exception as err:

print('An exception happened: ' + str(err))

* + This program uses the except Exception as err form of the except statement If an Exception object is returned from boxPrint()
  + This except statement will store it in a variable named err. The Exception object can then be converted to a string by passing it to str() to produce a userfriendly error message

**Output**

\*\*\*\*

\* \*

\* \*

\*\*\*\* OOOOOOOOOOOOOOOOOOOO O O

O O

O O

OOOOOOOOOOOOOOOOOOOO

An exception happened: Width must be greater than 2.

An exception happened: Symbol must be a single character string.

### 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.
  + when an exception occurs, a traceback is generated to provide information about the sequence of function calls that led to the exception. The traceback includes the line numbers and file names of the code that was executing when the exception occurred.

You can capture the traceback as a string using the `traceback` module in Python. The `traceback` module provides functions to format and manipulate tracebacks.

Here's an example that demonstrates how to get a traceback as a string in Python:

import traceback

def divide(a, b):

return a / b

try:

result = divide(10, 0)

print("Result:", result)

except:

traceback\_str = traceback.format\_exc()

print("Traceback:", traceback\_str)

### Assertions

* + *Python Assertions in any programming language are the debugging tools that help in the smooth flow of code. Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code so that failure of these doesn’t allow the code to execute further.*
  + ***Assert Keyword in Python***
  + *In simpler terms, we can say that assertion is the boolean expression that checks if the statement is True or False. If the statement is true then it does nothing and continues the execution, but if the statement is False then it stops the execution of the program and throws an error.*
  + 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.
  + assert statement consists of the following:
  + 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

*Example:*

*# initializing number*

*a = 4*

*b = 0*

*# using assert to check for 0*

*print("The value of a / b is : ")*

*assert b != 0*

*print(a / b)*

*output:*

*The value of a / b is :*

*---------------------------------------------------------------------------*

*AssertionError Traceback (most recent call last)*

*Input In [19], in <cell line: 10>()*

*8 # using assert to check for 0*

*9 print("The value of a / b is : ")*

*---> 10 assert b != 0*

*11 print(a / b)*

*AssertionError:*

###### Using an Assertion in a Traffic Light Simulation

* + The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively.
  + The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

market\_2nd = {'ns': 'green', 'ew': 'red'}

mission\_16th = {'ns': 'red', 'ew': 'green'}

* + To start the project, you want to write a switchLights() function, which will take an intersection dictionary as an argument and switch the lights.
  + At first, you might think that switchLights() should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'.

##### Program:

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' switchLights(market\_2nd)

* + while writing switchLights() you had added an assertion to check that at least one of the lights is always red,
  + include the following at the bottom of the function:

assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)

* + With this assertion in place, your program would crash with this error message:

Traceback (most recent call last):

File "carSim.py", line 14, in <module> switchLights(market\_2nd)

File "carSim.py", line 13, in switchLights

assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight) u AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}

* + The important line here is the AssertionError
  + Neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program’s execution

### Logging

Python has a built-in module logging which allows writing status messages to a file or any other output streams. The file can contain information on which part of the code is executed and what problems have arisen.

*The logging module in Python provides a flexible and customizable way to log messages and track the flow of a program. Some benefits of using the logging module include:*

*- Ability to control the verbosity of log messages.*

*- Different log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize log messages based on their importance.*

*- Logging messages to different outputs such as console, file, or network.*

*- Customizable log message format.*

import logging

logging.debug('This is a debug message')

logging.info('This is an info message')

logging.warning('This is a warning message')

logging.error('This is an error message')

logging.critical('This is a critical message')

The output of the above program would look like this:

WARNING:root:This is a warning message

ERROR:root:This is an error message

CRITICAL:root:This is a critical message

# importing module

import logging

# Create and configure logger

logging.basicConfig(filename="newfile.log",

format='%(asctime)s %(message)s',

filemode='w')

# Creating an object

logger = logging.getLogger()

# Setting the threshold of logger to DEBUG

logger.setLevel(logging.DEBUG)

# Test messages

logger.debug("Harmless debug Message")

logger.info("Just an information")

logger.warning("Its a Warning")

logger.error("Did you try to divide by zero")

logger.critical("Internet is down")

**Program For Factorial**

import logging

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

* %(message)s') logging.debug('Start of program') def factorial(n):

logging.debug('Start of factorial(%s%%)' % (n)) total = 1

for i in range(n + 1):

total \*= i

logging.debug('i is ' + str(i) + ', total is ' + str(total)) logging.debug('End of factorial(%s%%)' % (n))

return total print(factorial(5)) logging.debug('End of program')

* + debug() function will call basicConfig(), and a line of information will be printed.
  + This information will be in the format we specified in basicConfig() and will include the messages we passed to debug().

##### Output:

2015-05-23 16:20:12,664 - DEBUG - Start of program 2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5) 2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0

2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0

2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0

2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0

2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0

2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0

2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)

0

2015-05-23 16:20:12,684 - DEBUG - End of program

* + The factorial() function is returning 0 as the factorial of 5, which isn’t right.
  + The for loop should be multiplying the value in total by the numbers from 1 to 5. But the log messages displayed by logging.debug() show that the i variable is starting at 0 instead of 1.
  + Since zero times anything is zero, the rest of the iterations also have the wrong value for total
  + Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.
  + Change the for i in range(n + 1): line to for i in range(**1,** n + 1):, and run the program again

**Output**

2015-05-23 17:13:40,650 - DEBUG - Start of program 2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5) 2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1

2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2

2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6

2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24

2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120

2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)

120

2015-05-23 17:13:40,666 - DEBUG - End of program

###### Logging Levels

* + Logging levels provide a way to categorize your log messages by importance. There are five logging levels
  + Messages can be logged at each level using a different logging function.

|  |  |  |
| --- | --- | --- |
| **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 l | logging.warning() | Used to indicate a potential problem that |
|  |  | doesn’t prevent the program from |
|  |  | working but might do so in the future. |
| 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. |

Table 10-1: Logging Levels in Python

>>> import logging

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

#### %(levelname)s - %(message)s')

>>> logging.debug('Some debugging details.')

#### 2015-05-18 19:04:26,901 - DEBUG - Some debugging details.

>>> logging.info('The logging module is working.')

#### 2015-05-18 19:04:35,569 - INFO - The logging module is working.

>>> logging.warning('An error message is about to be logged.')

#### 2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.

>>> logging.error('An error has occurred.')

#### 2015-05-18 19:05:07,737 - ERROR - An error has occurred.

>>> logging.critical('The program is unable to recover!')

#### 2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!

##### Disabling Logging

* + The logging.disable() function disables these so that you don’t have to go into your program and remove all the logging calls by hand.
  + pass logging.disable() a logging level, and it will suppress all log messages at that level or lower

>>> import logging

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

%(message)s')

>>> logging.critical('Critical error! Critical error!')

2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!

>>> logging.disable(logging.CRITICAL)

>>> logging.critical('Critical error! Critical error!')

>>> logging.error('Error! Error!')

* + Since logging.disable() will disable all messages after it, you will probably want to add it near the import logging line of code in your program

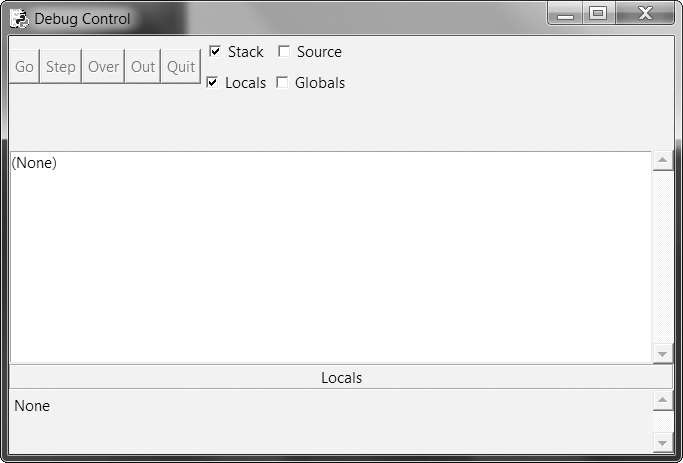
***Logging to a File***

* + Instead of displaying the log messages to the screen, you can write them to a text file. The logging.basicConfig() function takes a filename keyword argument,

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

### IDLE ’s Debugger

* + The *debugger* is a feature of IDLE 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
  + To enable IDLE’s debugger, click Debug4Debugger in the interactive shell window.
  + When the Debug Control window appears, select all four of the Stack, Locals, Source, and Globals checkboxes so that the window shows the full set of debug information
  + While the Debug Control window is displayed, any time you run a program from the file editor
  + debugger will pause execution before the first instruction and display the following:
  + The line of code that is about to be executed
  + A list of all local variables and their values
  + A list of all global variables and their values



#### Figure: The Debug Control window

* + You’ll notice that in the list of global variables there are several variables you haven’t defined, such as builtins , doc , file , and so on. These are variables that Python automatically sets whenever it runs a program.
  + The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

###### Go

* + Clicking the Go 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 **Go**

button.

###### Step

* + Clicking the Step button will cause the debugger to execute the next line of code and then pause again
  + The Debug Control window’s list of global and local variables will be updated if their values change.
  + 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.

###### Over

* + Clicking the Over button will execute the next line of code, similar to the Step button.
  + The 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.
  + For example, if the next line of code is a print() call, you don’t really care about code inside the built-in print() function; you just want the string you pass it printed to the screen.

###### Quit

* + If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Quit button
  + The Quit button will immediately terminate the program. If you want to run your program normally again, select Debug4Debugger again to disable the debugger.

###### Debugging a Number Adding Program

print('Enter the first number to add:') first = input()

print('Enter the second number to add:') second = input()

print('Enter the third number to add:') third = input()

print('The sum is ' + first + second + third)

Save it as *buggyAddingProgram.py* and run it first without the debugger enabled.

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

* + The program hasn’t crashed, but the sum is obviously wrong. Let’s enable the Debug Control window and run it again, this time under the debugger
  + When you press F5 or select **Run**4**Run Module** (with **Debug**4**Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1.
  + The debugger will always pause on the line of code it is about to execute.

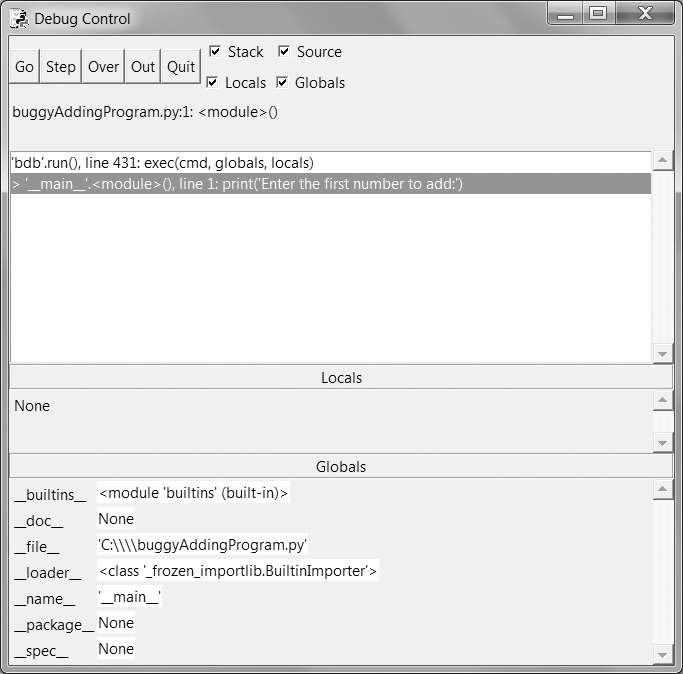


Figure The Debug Control window when the program first starts under the debugger

* + Click the **Over** button once to execute the first print() call. You should use Over instead of Step here, since you don’t want to step into the code for the print() function.
  + The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted.

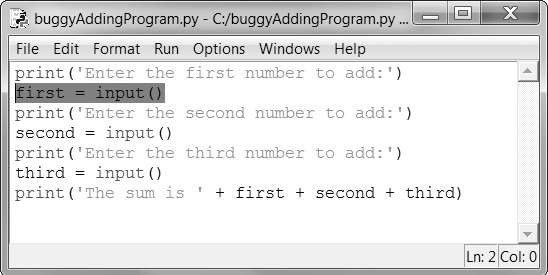
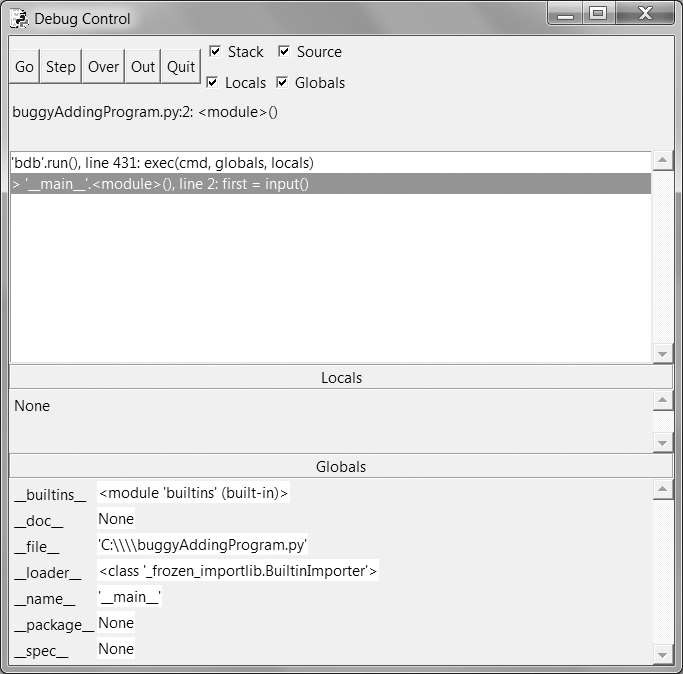


Figure: The Debug Control window after clicking Over



* + Click Over again to execute the input() function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the input() call into the interactive shell window.
  + Enter **5** and press Return. The Debug Control window buttons will be reenabled.
  + Keep clicking Over, entering 3 and 42 as the next two numbers, until the debugger is on line 7, the final print() call in the program
  + Globals section that the first, second, and third variables are set to string values '5', '3', and '42' instead of integer values 5, 3, and 42.
  + When the last line is executed, these strings are concatenated instead of added together, causing the bug.

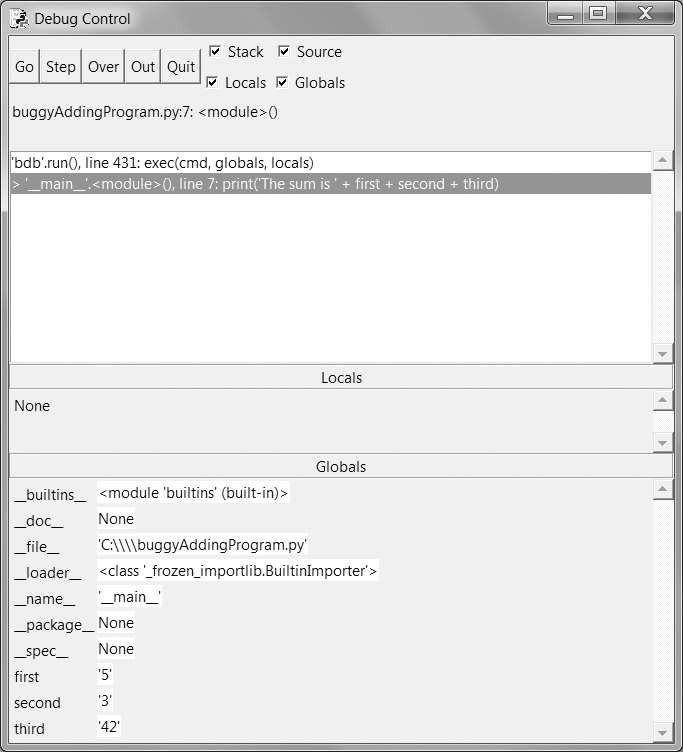


Figure The Debug Control window on the last line. The variables are set to strings, causing the bug.

### Breakpoints

* + A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line.
  + Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times.

import random heads = 0

for i in range(1, 1001):

if random.randint(0, 1) == 1:

heads = heads + 1 if i == 500:

print('Halfway done!')

print('Heads came up ' + str(heads) + ' times.')

* + The random.randint(0, 1) call u will return 0 half of the time and 1 the other half of the time.
  + This can be used to simulate a 50/50 coin flip where 1 represents heads.

**Output:**

Halfway done!

Heads came up 490 times.

* + If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated.
  + If you were interested in the value of heads at the halfway point of the program’s execution, when 500 of 1000 coin flips have been completed, you could instead just set a breakpoint on the line print('Halfway done!')
  + To set a breakpoint, right-click the line in the file editor and select Set Breakpoint,

***explain the program development concept “protototype and patch”with suitable example?***

The "prototype and patch" is a program development concept that involves rapidly prototyping a solution and then refining it through a series of patches or incremental changes. This approach allows for quick iterations and improvements based on feedback and testing. Here's an explanation with a suitable example:

1. Prototype:

- In the prototyping phase, a basic version or a proof-of-concept of the program is developed quickly.

- The focus is on implementing the core functionality and testing it for feasibility and effectiveness.

- The prototype may not have all the desired features or a polished user interface.

Example:

Let's say you are developing a weather application. In the prototyping phase, you might create a basic version that fetches weather data for a single location and displays it in a simple text format. The prototype may not have additional features like multiple locations, graphical representation, or user customization options. The goal is to quickly validate the basic functionality of fetching and displaying weather data.

2. Patch:

- After the prototype is developed, it is tested and evaluated for any issues or limitations.

- Based on the feedback and requirements, incremental changes or "patches" are made to improve the program.

- Patches can include bug fixes, adding new features, enhancing the user interface, or optimizing the code.

Example:

In the weather application example, after the prototype is tested, you receive feedback from users requesting additional features like a graphical representation of temperature trends and the ability to add multiple locations. In the patching phase, you would make incremental changes to implement these features. This may involve updating the code to fetch data for multiple locations, designing a visual representation of temperature trends using charts or graphs, and enhancing the user interface to allow users to customize their preferred settings.

The patching phase continues iteratively as the program evolves based on user feedback, bug reports, and new requirements. Each patch improves the program's functionality, usability, and performance.

The "prototype and patch" approach allows for flexibility and quick iterations in program development. It helps in validating ideas, getting early user feedback, and gradually refining the program to meet the desired goals.

***What is logging? how this would be used to debug the python program?***

*Logging is a best way to understand what’s happening in the program and in what order it’s happening.*

*Python provides logging module that makes it easy to create a record of custom messages that you write. 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.*

*To enable the logging module to display log messages on the screen as program runs, add the following code at the beginning of the program.*

*import logging*

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

*The Python log on an event, it creates a LogRecord object that holds information about that event. The logging module’s basicConfig() function lets you specify what details about the LogRecord object you want to see and how you want those details displayed.*

*Sample program to demonstrate use of logging module*

*import logging*

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

*- %(message)s')*

*logging.debug('Start of program')*

*def factorial(n):*

*logging.debug('Start of factorial(%s%%)' % (n))*

*total = 1*

*for i in range(n + 1):*

*total \*= i*

*logging.debug('i is ' + str(i) + ', total is ' + str(total))*

*logging.debug('End of factorial(%s%%)' % (n))*

*return total*

*print(factorial(5))*

*logging.debug('End of program')*

*The logging.debug() function is used when we want to print log information. This debug() function will call basicConfig(), and a line of information will be printed. This information will be in the format we specified in basicConfig() and will include the messages we passed to debug().*

**Develop a function called Print\_time that takes a time object & print it in**

**the form of hour:minute:second**

ANS:

python

def Print\_time(time):

print(f"{time.hour}:{time.minute}:{time.second}")

# Example usage:

class Time:

def init(self, hour, minute, second):

self.hour = hour

self.minute = minute

self.second = second

time\_obj = Time(9, 30, 45)

Print\_time(time\_obj) # Output: 9:30:45

**Explain the benefits of using logging module with an example**

ANS:

1. Easy Configuration: The logging module allows easy configuration of log levels, output destinations, and formatting.

2. Centralized Logging: It provides a unified interface for logging across different components of an application, making it easier to analyze and troubleshoot issues.

3. Log Level Control: Different log levels (e.g., DEBUG, INFO, WARNING, ERROR) provide granular control over the verbosity of log messages.

4. Timestamps and Formatting: The logging module supports timestamps and customizable log message formatting for better readability and analysis.

5. Multiple Output Destinations: Logs can be directed to console, files, sockets, or custom destinations.

6. Exception Handling: The logging module can capture and log exceptions, including stack traces, aiding in debugging.

Example:

import logging

# Configure logging

logging.basicConfig(filename='app.log', level=logging.DEBUG,

format='%(asctime)s - %(levelname)s - %(message)s')

def divide(a, b):

try:

result = a / b

logging.info(f"Division successful: {a} / {b} = {result}")

except ZeroDivisionError as e:

logging.error(f"Division by zero error: {e}")

except Exception as e:

logging.exception(f"An error occurred: {e}")

# Usage

divide(10, 2) # Logs: "Division successful: 10 / 2 = 5.0"

divide(10, 0) # Logs: "Division by zero error: division by zero"

divide("10", 2) # Logs: "An error occurred: unsupported operand type(s) for /: 'str' and 'int'"

**Write an algorithm for implement multi clipboard functionality.**

If you’ve responded to a large number of emails with similar phrasing, you’ve probably had to do a lot of repetitive typing. Maybe you keep a text document with these phrases so you can easily copy and paste them using the clipboard. But your clipboard can only store one message at a time, which isn’t very convenient. Let’s make this process a bit easier with a program that stores multiple phrases.

# mclip.py - A multi-clipboard program.

TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",

'busy': """Sorry, can we do this later this week or next week?""",

'upsell': """Would you consider making this a monthly donation?"""}

import sys, pyperclip

if len(sys.argv) < 2:

print('Usage: py mclip.py [keyphrase] - copy phrase text')

sys.exit()

keyphrase = sys.argv[1] # first command line arg is the keyphrase

if keyphrase in TEXT:

pyperclip.copy(TEXT[keyphrase])

print('Text for ' + keyphrase + ' copied to clipboard.')

else:

print('There is no text for ' + keyphrase)

Now that the key phrase is stored as a string in the variable keyphrase, you need to see whether it exists in the TEXT dictionary as a key. If so, you want to copy the key’s value to the clipboard using pyperclip.copy(). (Since you’re using the pyperclip module, you need to import it.) Note that you don’t actually need the keyphrase variable; you could just use sys.argv[1] everywhere keyphrase is used in this program. But a variable named keyphrase is much more readable than something cryptic like sys.argv[1].

|  |  |  |
| --- | --- | --- |
| 8c) | **Explain the support for Logging with logging module in Python**  **Logging**   * Logging is a great way to understand what’s happening in your program and in what order its happening. * Python’s logging module makes it easy to create a record of custom messages that you write. * 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**   * To enable the logging module to display log messages on your screen as your program runs, * when Python logs an event, it creates a LogRecord object that holds information about that event. * The logging module’s basicConfig() function lets you specify what details about the LogRecord object you want to see and how you want those details displayed.   With the logging module imported, you can use something called a “logger” to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:  DEBUG INFO WARNING ERROR CRITICAL  The logging module provides you with a default logger that allows you to get started without needing to do much configuration. The corresponding methods for each level can be called as shown in the following example: | 08 |

|  |  |  |
| --- | --- | --- |
|  | import logging  logging.debug('This is a debug message') logging.info('This is an info message') logging.warning('This is a warning message') logging.error('This is an error message') logging.critical('This is a critical message') |  |