Module 4:

Organizing Files and Debugging

ORGANIZING FILES

- ➢Your programs can also organize preexisting files on hard drive.
- ➢Maybe you’ve had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand.
- ➢Or consider tasks such as these:
  - ✓ Making copies of all PDF files (and only the PDF files) in every subfolder of a folder
  - ✓ Removing the leading zeros in the filenames for every file in a folder of hundreds of files named spam001.txt, spam002.txt, spam003.txt, and so on
  - ✓ Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

ORGANIZING FILES

- All this boring stuff is just begging to be automated in Python.
- By programming your computer to do these tasks, you can transform it intoa quick-working file clerk who never makes mistakes.
- As you begin working with files, you may find it helpful to be able to quickly see what the extension (.txt, .pdf, .jpg, and so on) of a file is.
- With macOS and Linux, your file browser most likely shows extensions automatically.
- With Windows, file extensions may be hidden by default.
- To show extensions, go to Start ▸ Control Panel ▸ Appearance and Personalization ▸ Folder Options.
- On the View tab, under Advanced Settings, uncheck the Hide extensions for known file types checkbox.

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.

Copying Files and Folders

➢Enter the following to see how shutil.copy() works:

```
>>> 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')
```

➢ 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 ❶, the original spam.txt filename is used for the new, copied file's filename.

➢ The second shutil.copy() call ❷ also copies the file at C:\Users\Al\eggs.txt to folder C:\Users\Al\some_folder but gives the copied file the name eggs2.txt.

Copying Files and Folders

- While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it.
- Calling shutil.copytree(source, destination) will copy folder at path source, along with all of its files and subfolders, to the folder at the path destination.
- The source and destination parameters are both strings.
- The function returns a string of the path of the copied folder.

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
>>> shutil.copytree(p / 'spam', p / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

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.

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

- 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().

Moving and Renaming Files and Folders

- The destination path can also specify a filename.
- In the following example, the source file is moved and renamed:

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

- This line says, “Move C:\bacon.txt into the folder C:\eggs, and while you’re at it, rename that bacon.txt file to new_bacon.txt.”
- Both of the previous examples worked under the assumption that there was a folder eggs in the C:\ directory. But if there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

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

Moving and Renaming Files and Folders

- Both of the previous examples worked under the assumption that there was a folder eggs in the C:\ directory. But if there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

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

- Here, move() can’t find a folder named eggs in the C:\ directory and so assumes that destination must be specifying a filename, not a folder.
- This can be a tough-to-spot bug in your programs since the move() call can happily do something that might be quite different from what you were expecting.
- This is yet another reason to be careful when using move().

Moving and Renaming Files and Folders

- Finally, the folders that make up the destination must already exist, or else Python will throw an exception.
- Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
--snip--
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\
eggs\\ham'
```

- Python looks for eggs and ham inside the directory does_not_exist.
- It doesn’t find the nonexistent directory, so it can’t move spam.txt to the path you specified.

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.

➢ 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.

Permanently Deleting Files and Folders

- Be careful when using these functions in your programs!
- It's often a good idea to first run your program with these calls commented out and with print() calls added to show the files that would be deleted.
- Here is a Python program that was intended to delete files that have the .txt file extension but has a typo (highlighted in bold) that causes it to delete .rxt files instead:

```
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.

Permanently Deleting Files and Folders

- 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)
```

- 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.
- Once you are certain the program works as intended, delete theprint(filename) line and uncomment the os.unlink(filename) line.
- Then run the program again to actually delete the 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.
- 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.
- (See Appendix A for a more in-depth explanation of how to install third party modules.)
- 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.

Safe Deletes with the send2trash Module

- After you have installed send2trash, enter the following into interactive shell:

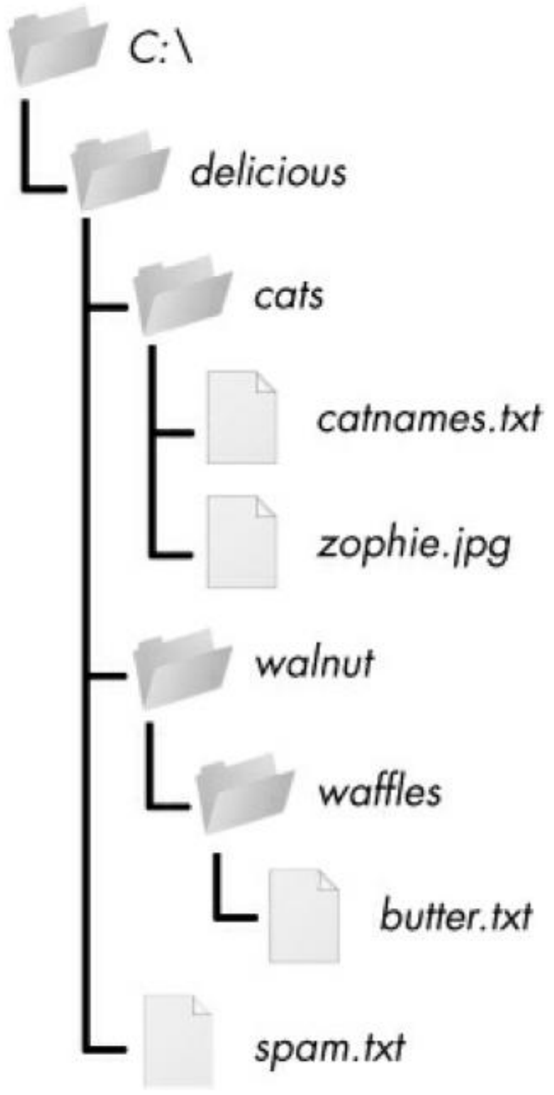
```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

- Note that the send2trash() function can only send files to the recycle bin; it cannot pull files out of it.

- 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.
- An example folder that contains three folders and four files is shown in figure.

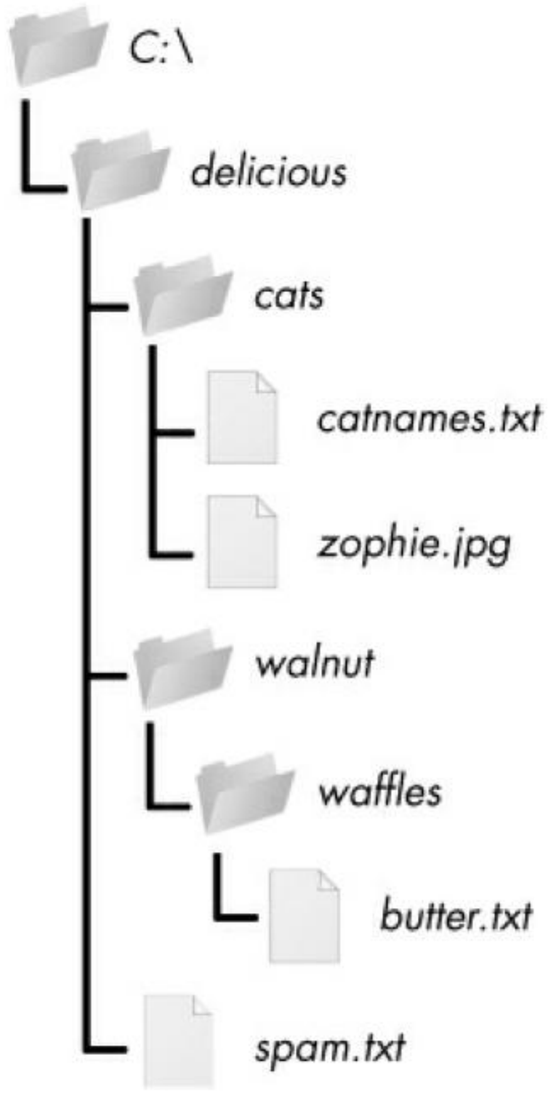


Walking a Directory Tree

- Here is an example program that uses the os.walk() function on the directory tree shown here:

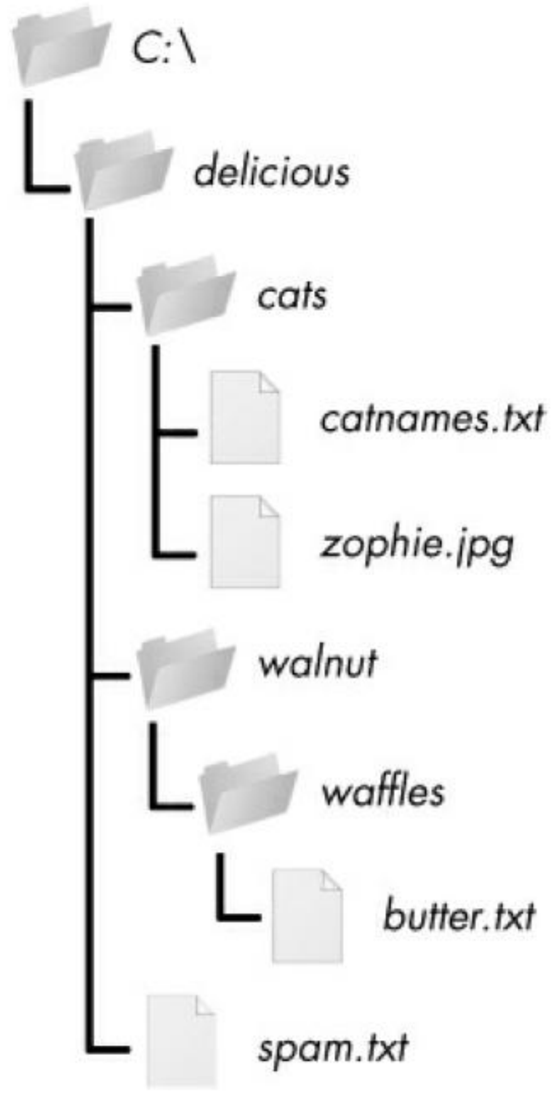
```
import os
for folderName, subfolders, filenames in os.walk('C:\\delicious'):
print('The current folder is ' + folderName)
for subfolder in subfolders:
print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
for filename in filenames:
print('FILE INSIDE ' + folderName + ': '+ filename)
print('')
```

- 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.



Walking a Directory Tree

- Unlike range(), the os.walk() function will return three values on each iteration through the loop:
  - ✓ A string of the current folder's name
  - ✓ A list of strings of the folders in current folder
  - ✓ A list of strings of the files in the current folder
- (By current folder, I mean the folder for the current iteration of the for loop. The current working directory of the program is not changed by os.walk().)
- Just like you can choose the variable name i in the code for i in range(10):, you can also choose the variable names for the three values listed earlier. I usually use the names foldername, subfolders, and filenames.



Walking a Directory Tree

➢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.
```

```
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('')
```

Since os.walk() returns lists of strings for the subfolder and filename variables, 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

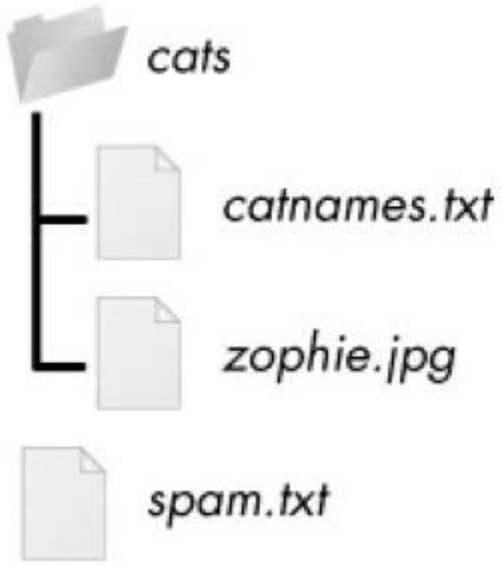
Compressing Files with the zipfile Module

- You may be familiar with ZIP files (with the .zip file extension), which 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.
- This single file, called an archive file, can then be, attached to an email.
- Your Python programs can create and open (or extract) ZIP files using functions in the zipfile module. Say you have a ZIP file named example.zip

Reading ZIP Files

- To read the contents of a ZIP file, first 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 in the previous chapter: 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.
- Note that zipfile is the name of the Python module, and ZipFile() is the name of the function.

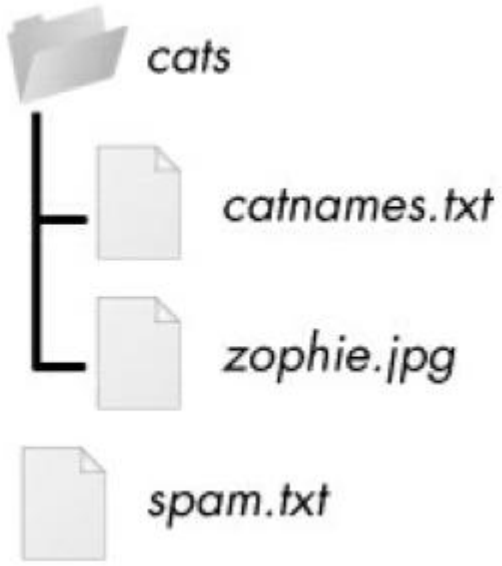
The contents of example.zip



Reading ZIP Files

```
>>> 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
13908
>>> spamInfo.compress_size
3828
❶ >>> f'Compressed file is {round(spamInfo.file_size / spamI
.compress_size, 2)}x smaller!'
)
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

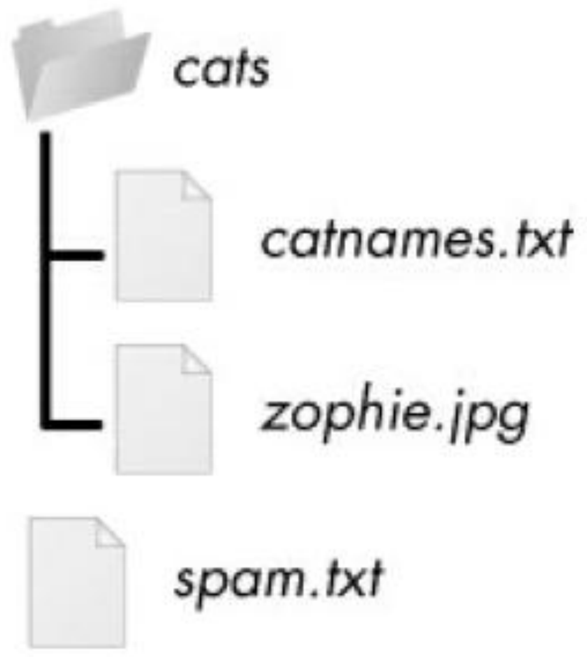
The contents of example.zip



Reading ZIP Files

- 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.
- While a ZipFile object represents an entire archive file, a ZipInfo object holds useful information about a single file in the archive

The contents of example.zip



Reading ZIP Files

- The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.
- After running this code, the contents of example.zip will be extracted toC:\.
- 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 folder passed to the extractall() method does not exist, it will be created.
- For instance, if you replaced the call at ❶ with exampleZip.extractall('C:\\delicious'), the code would extract the files from example.zip into a newly created C:\delicious folder.

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

Reading ZIP Files

- 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()
```

- If this second argument is a folder that doesn’t yet exist, Python will create folder
- The value that extract() returns is the absolute path to which the file was extracted.

- The string you pass to extract() must match one of the strings in thelist returned by namelist().
- Optionally, you can pass a second argument to extract() to extract the file into a folder other than the current working directory.

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.)
- 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.)

Creating and Adding to ZIP Files

- Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip',
'w')
>>> newZip.write('spam.txt',
compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

- This code will create a new ZIP file named new.zip that has the compressed contents of spam.txt.
- Note 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

- Suppose thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to Europeanstyle dates (DD-MM-YYYY).
- This boring task could take all day to do by hand! Let's write a program to do it instead. Here's what the program does:

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

- 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.

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. Make your

```
#! python3
# renameDates.py - Renames filenames with American MM-
DD-YYYY date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date
format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
```

Step 1: Create a Regex for American-Style Dates

- The shutil.move() function can be used to rename files: its arguments are name of file to rename & new filename.
- Because this function exists in the shutil module, you must import that module ❶.
- But before renaming files, identify which files to rename. Filenames with dates such as spam4-4-1984.txt & 01-03-2014eggs.zip should be renamed, while filenames without dates such as littlebrother.epub can be

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY
date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

Step 1: Create a Regex for American-Style Dates

- You can use a regular expression to identify this pattern.
- After importing the re module at the top, call re.compile() to create a Regex object
- ❷. Passing re.VERBOSE for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY
date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

Step 1: Create a Regex for American-Style Dates

- The regular expression string begins with ^(.*?) to match any text at the beginning of the filename that might come before date.
- The ((0|1)?\d) group matches the month. The first digit can be either 0 or 1, so the regex matches 12 for December but also 02 for February.
- This digit is also optional so that the month can be 04 or 4 for April.

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY
date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

Step 1: Create a Regex for American-Style Dates

- The group for the day is ((0|1|2|3)?\d) and follows similar logic; 3, 03, and 31 are all valid numbers for days. (Yes, this regex will accept some invalid dates such as 4- 31-2014, 2-29-2013, and 0-15-2014.
- Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY
date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

Step 1: Create a Regex for American-Style Dates

- While 1885 is a valid year, you can just look for years in the 20th or 21st century.
- This will keep your program from accidentally matching nondate filenames with a date-like format, such as 10-10-1000.txt.
- The (.*?)$ part of the regex will match any text that comes after the date.

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY
date format
# to European DD-MM-YYYY.
❶ import shutil, os, re
# Create a regex that matches files with American date format.
❷ 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.VERBOSE❸)
# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

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. Fill in the first three TODOs in yur program with following

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.
--snip--
# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
mo = datePattern.search(amerFilename)
# Skip files without a date.
❶ if mo == None:
❷ continue
❸ # Get the 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)
--snip--
```

Step 2: Identify the Date Parts from the Filenames

- If the Match object returned from the search() method is None ❶, then the filename in amerFilename does not match the regular expression.
- Continue statement ❷ will skip rest of loop and move on to next filename.
- Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart ❸.
- The strings in these variables will be used to form the European-style filename in the next step
- To keep the group numbers straight, try reading the regex from the beginning, and count up each time you encounter an opening parenthesis.
- Without thinking about code, just write an outline of regular expression.
- This can help you visualize the groups.

Step 2: Identify the Date Parts from the Filenames

- Ex:

```
datePattern = re.compile(r"""^(1) # all text before the date
(2 (3) )- # one or two digits for the month
(4 (5) )- # one or two digits for the day
(6 (7) ) # four digits for the year
(8)$ # all text after the date
""", re.VERBOSE)
```

- Here, the numbers 1 through 8 represent the groups in the regular expression you wrote.
- Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form 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: date comes before month.
- Fill in the three remaining TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
--snip--
# Form the European-style filename.
❶ euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart + afterPart
# Get the full, absolute file paths.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)
# Rename the files.
❷ print(f'Renaming "{amerFilename}" to "{euroFilename}"...')
❸ #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

Step 3: Form New Filename and Rename the Files

- Store the concatenated string in a variable named euroFilename ❶.
- Then, pass the original filename in amerFilename and the new euroFilename variable to the shutil.move() function to rename the file ❸.
- This program has the shutil.move() call commented out and instead prints the filenames that will be renamed ❷.
- Running the program like this first can let you double-check that the files are renamed correctly.
- Then you can uncomment the shutil.move() call and run the program again to actually rename the files.

Backing Up a Folder into a ZIP File

- Say you’re working on a project whose files you keep in a folder named C:\AlsPythonBook.
- You’re worried about losing your work, so you’d like to create ZIP file “snapshots” of the entire folder.
- You’d like to keep different versions, so you want the ZIP file’s filename to increment each time it is made; for example, AlsPythonBook_1.zip, AlsPythonBook_2.zip, AlsPythonBook_3.zip, and so on.
- You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files’ names.
- It would be much simpler to run a program that does this boring task for you.
- For this project, open a new file editor window and save it as backupToZip.py.

Backing Up a Folder into a ZIP File

- The code for this program will be placed into a function named backupToZip()
- This will make it easy to copy & paste the function into other Python programs that need this functionality
- At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#! python3
# backupToZip.py - Copies an entire folder &its contents into
# a ZIP file whose filename increments.
❶ import zipfile, os
def backupToZip(folder):
# Back up the entire contents of "folder" into a ZIP file.
folder = os.path.abspath(folder) # make sure folder is absolute
# Figure out the filename this code should use based on
# what files already exist.
❷ number = 1
❸ while True:
zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
if not os.path.exists(zipFilename):
break
number = number + 1
❹ # TODO: Create the ZIP file.
# TODO: Walk the entire folder tree and compress the files in each folder
print('Done.')
backupToZip('C:\\delicious')
```

Backing Up a Folder into a ZIP File

- The first part, naming the ZIP file, uses the base name of the absolute path of folder.
- If the folder being backed up is C:\delicious, the ZIP file's name should be delicious_N.zip, where N = 1 is the first time you run the program, N = 2 is the second time, and so on.
- You can determine what N should be by checking whether delicious_1.zip already exists, then checking whether delicious_2.zip already exists, and so on.
- Use a variable named number for N ❷, and keep incrementing it inside the loop that calls os.path.exists() to check whether the file exists ❸.
- The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Backing Up a Folder into a ZIP File

- Create the ZIP file, using:
- Now that new ZIP file's name is stored in the zipFilename variable, you can call zipfile.ZipFile() to actually create ZIP file ❶.
- Be sure to pass 'w' as second argument so that the ZIP file is opened in write mode.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.
--snip--
while True:
zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
if not os.path.exists(zipFilename):
break
number = number + 1
# Create the ZIP file.
print(f'Creating {zipFilename}...')
❶ backupZip = zipfile.ZipFile(zipFilename, 'w')
# TODO: Walk the entire folder tree and compress the files in each folder.
print('Done.')
backupToZip('C:\\delicious')
```

Step 3: Walk the Directory Tree & Add to ZIP File

- Now you need to use the os.walk() function to do the work of listing every file in the folder and its subfolders.
- You can use os.walk() in a for loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, & filenames in that

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.
--snip--
# Walk the entire folder tree and compress the files in each folder.
❶ for foldername, subfolders, filenames in os.walk(folder):
print(f'Adding files in {foldername}...')
# Add the current folder to the ZIP file.
❷ backupZip.write(foldername)
# Add all the files in this folder to the ZIP file.
❸ for filename in filenames:
newBase = os.path.basename(folder) + '_'
if filename.startswith(newBase) and filename.endswith('.zip'):
continue # don't back up the backup ZIP files
backupZip.write(os.path.join(foldername, filename))
backupZip.close()
print('Done.')
backupToZip('C:\\delicious')
```

Step 3: Walk the Directory Tree & Add to ZIP File

- In the for loop, the folder is added to the ZIP file ❷.
- The nested for loop can go through each filename in the filenames list ❸.
- Each of these is added to the ZIP file, except for previously made backup ZIPs.
- When you run this program, it will produce output that will look like this:
- Second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

Output:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

Debugging

Debugging

- This chapter covers some tools and techniques for finding root cause of bugs in program
- They help to fix bugs faster and with less effort.
- Computer will do only what you tell it to do; it won't read our mind and do what we intended it to do.
- Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.
- Second, you will look at how to use the debugger.
- The debugger is a feature of Mu that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track how the values change over the course of your program.
- This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be

Raising Exceptions

- ➢ Python raises an exception whenever it tries to execute invalid code.
- ➢ In Chapter 3, you read about how to handle Python’s exceptions with try and except statements so that your program can recover from exceptions that you anticipated.
- ➢ How raise your own exceptions in your 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 raise keyword
  - ✓ A call to the Exception() function
  - ✓ A string with a helpful error message passed to the Exception() function

Ex:

```
>>> 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.
```

Raising Exceptions

- 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.
- Often it’s the code that calls the function, rather than the function itself, that knows how to handle an exception.
- That means you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.
- For example, open a new file editor tab, enter the following code, and save the program as boxPrint.py:

Raising Exceptions

```
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:
❸ 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)
❹ except Exception as err:
❺ print('An exception happened: ' + str(err))
```

- A boxPrint() function is defined that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width & height.
- This box shape is printed to the screen.
- Say we want the character to be a single character, and the width and height to be greater than 2.
- Add if statements to raise exceptions if these requirements aren't satisfied.
- Later, when we call boxPrint() with various arguments, try/except will

Raising Exceptions

```
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:
❸ 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)
❹ except Exception as err:
❺ print('An exception happened: ' + str(err))
```

- This program uses the except Exception as err form of the except statement ❹.
- If an Exception object is returned from boxPrint() ❶ ❷ ❸, this except statement will store it in a variable named err.
- We can then convert the Exception object to a string by passing it to str() to produce a user friendly error message ❺.
- When you run this boxPrint.py, the output will look like this:

Raising Exceptions

```
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:
❸ 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)
❹ except Exception as err:
❺ print('An exception happened: ' + str(err))
```

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.
```

Raising Exceptions

```
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:
❸ 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)
❹ except Exception as err:
❺ print('An exception happened: ' + str(err))
```

Output:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
```

- Using the try and except statements, you can handle errors more gracefully instead of letting entire program crash

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.
- Consider an example: Example.py:

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

Output:

```
Traceback (most recent call last):
File "errorExample.py", line 7, in <module>
spam()
File "errorExample.py", line 2, in spam bacon()
File "errorExample.py", line 5, in bacon
raise Exception('This is the error message.')
Exception: This is the error message.
```

Getting the Traceback as a String

- Consider an example: Example.py:

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

Output:

```
Traceback (most recent call last):
File "errorExample.py", line 7, in <module>
spam()
File "errorExample.py", line 2, in spam bacon()
File "errorExample.py", line 5, in bacon
raise Exception('This is the error message.')
Exception: This is the error message.
```

- From 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, call stack can help you determine which call led to the error.

Getting the Traceback as a String

- Consider an example: Example.py:

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

Output:

```
Traceback (most recent call last):
File "errorExample.py", line 7, in <module>
spam()
File "errorExample.py", line 2, in spam bacon()
File "errorExample.py", line 5, in bacon
raise Exception('This is the error message.')
Exception: This is the error message.
```

- Python displays the traceback whenever a raised exception goes unhandled.
- But you can also obtain it as a string by calling traceback.format_exc().
- This function is useful if you want the information from an exception’s traceback but also want an except statement to gracefully handle the exception.
- You will need to import Python’s traceback module before calling this function.

Getting the Traceback as a String

- For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a text file and keep your program running.
- You can look at the text file later, when you're ready to debug your program. Enter the following into the interactive shell:

```
>>> import traceback
>>> try:
... raise Exception('This is the error message.')
except:
... errorFile = open('errorInfo.txt', 'w')
... errorFile.write(traceback.format_exc())
... errorFile.close()
... print('The traceback info was written to errorInfo.txt.')
111
The traceback info was written to errorInfo.txt.
```

- The 111 is the return value from the write() method, since 111 characters were written to file.
- The traceback text was written to errorInfo.txt.
- Traceback (most recent call last):
  File "<pyshell#28>", line 2, in
   <module>
  Exception: This is the error
message.

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:

  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

Assertions

- 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.”
- Ex:

```
>>> 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.
```

- 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 code in sort() is bug-free & did its job, then assertion would be true.
- Because the ages[0] <= ages[-1] expression evaluates to True, the assert
- statement does nothing. Because the ages[0] <= ages[-1] expression evaluates to True, the assert statement does nothing.

Assertions

- However, let’s pretend we had a bug in our code.
- Say we accidentally called the reverse() list method instead of the sort() list method.
- When we enter the following in the interactive shell, the assert statement raises an
- AssertionError:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.reverse()
>>> ages
[73, 47, 80, 17, 15, 22, 54, 92, 57, 26]
>>> assert ages[0] <= ages[-1] # Assert that the first age is <= the last age.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AssertionError
```

Assertions

- Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash.
- By “failing fast” like this, you shorten the time between the original cause of the bug and when you first notice the bug.
- This will reduce amount of code you will have to check before finding bug’s cause.
- Assertions are for programmer errors, not user errors.
- Assertions should only fail while the program is under development; a user should never see an assertion error in a finished program.
- For errors that your program can run into as a normal part of its operation (such as a file not being found or the user entering invalid data), raise an exception

Assertions

- You shouldn’t use assert statements in place of raising exceptions, because users can choose to turn off assertions.
- If you run a Python script with python -O myscript.py instead of python myscript.py, Python will skip assert statements.
- Users might disable assertions when they’re developing a program and need to run it in a production setting that requires peak performance. (Though, in many cases, they’ll leave assertions enabled even then.)
- Assertions also aren’t a replacement for comprehensive testing.
- For instance, if the previous ages example was set to [10, 3, 2, 1, 20], then the assert ages[0] <= ages[-1] assertion wouldn’t notice that the list was unsorted, because it just happened to have a first age that was less than or equal to the last age, which is the only thing the assertion checked for.

Using an Assertion in a Traffic Light Simulation

- Say you’re building a traffic light simulation program.
- 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'}
```

- These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street.
- To start the project, you want to write a switchLights() function, which will take an intersection dictionary as an argument

Using an Assertion in a Traffic Light Simulation

- 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'.
- The code to implement this idea might look like this:

```
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)
```

- You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it.
- When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Using an Assertion in a Traffic Light Simulation

- Since you’ve already written the rest of the program, you have no ideawhere the bug could be.
- Maybe it’s in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the switchLights() function.
- But if while writing switchLights() you had added an assertion to check that at least one of the lights is always red, you might have included following at 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)
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

Using an Assertion in a Traffic Light Simulation

- The important line here is the AssertionError ❶.
- While your program crashing is not ideal, it immediately points out that a sanity check failed: 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, you can save yourself a lot of future debugging effort.

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

Logging

- If you've ever put a print() statement in your code to output some variable's value while program is running, you've used a form of logging to debug 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.
- 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, copy the following to the top of your program (but under the #! python shebang line):

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

- You don’t need to worry too much about how this works, but basically, when Python logs an event, it creates a LogRecord object that holds information about that event.
- The logging module’s basicConfig() function lets to specify what details about LogRecord object you want to see and how you want those details displayed.

Using the logging Module

- Say you wrote a function to calculate the factorial of a number.
- In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4× 5 × 6 × 7, or 5,040.
- Open a new file editor tab and enter the following code.
- It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as factorialLog.py.

```
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')
```

Using the logging Module

- Here, we use the logging.debug() function when we want to print loginformation.
- 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().
- print(factorial(5)) call is part of original program, so result is displayed even if logging messages

```
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')
```

Using the logging Module

- Output:

```
2019-05-23 16:20:12,664 - DEBUG - Start of program
2019-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2019-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2019-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2019-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2019-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2019-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2019-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2019-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2019-05-23 16:20:12,684 - DEBUG - End of program
```

- The factorial() function is returning 0 as factorial of 5, which isn't right. for loop should be multiplying value in total by numbers from 1 to 5.

```
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')
```

Using the logging Module

- Output:

```
2019-05-23 16:20:12,664 - DEBUG - Start of program
2019-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2019-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2019-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2019-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2019-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2019-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2019-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2019-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2019-05-23 16:20:12,684 - DEBUG - End of program
```

- But 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.

```
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')
```

Using the logging Module

- Change the for i in range(n + 1): line to for i in range(1, n + 1):, and run the program again.
- The output will look like this:

```
2019-05-23 17:13:40,650 - DEBUG - Start of program
2019-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2019-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2019-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2019-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2019-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2019-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2019-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2019-05-23 17:13:40,666 - DEBUG - End of program
```

- The factorial(5) call correctly returns 120. The log messages showed what was going on inside the loop, which led straight to bug.

- You can see that logging.debug() calls printed out not just the strings passed to them but also a timestamp and the word DEBUG.

Don’t Debug with the print() Function

- Typing import logging and logging.basicConfig(level=logging.DEBUG, format='% (asctime)s - %(levelname)s - %(message)s') is somewhat unwieldy.
- You may want to use print() calls instead, but don’t give in to this temptation!
- Once you’re done debugging, you’ll end up spending a lot of time removing print() calls from your code for each log message.
- You might even accidentally remove some print() calls that were being used for nonlog messages.
- The nice thing about log messages is that you’re free to fill your program with as many as you like, and you can always disable them later by adding a single logging.disable(logging.CRITICAL) call.
- Unlike print(), the logging module makes it easy to switch between showing and hiding log messages.

Don’t Debug with the print() Function

- Log messages are intended for the programmer, not the user.
- The user won’t care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that.
- For messages that the user will want to see, like File not found or Invalid input, please enter a number, you should use a print() call.
- You don’t want to deprive the user of useful information after you’ve disabled log messages.

Logging Levels

- Logging levels provide a way to categorize your log messages by importance. There are five logging levels, least to most important.
- Messages can be logged at each level using a different logging function.

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 program from working but might do so in future.

Logging Levels

Logging Levels in Python

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.

Logging Levels

- Your logging message is passed as a string to these functions.
- The logging levels are suggestions.
- Ultimately, it is up to you to decide which category your log message falls into. Enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format='
%(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2019-05-18 19:04:26,901 - DEBUG - Some debugging
details.
>>> logging.info('The logging module is working.')
2019-05-18 19:04:35,569 - INFO - The logging module is
working.
>>> logging.warning('An error message is about to be
logged.')
2019-05-18 19:04:56,843 - WARNING - An error message
is about to be logged.
>>> logging.error('An error has occurred.')
2019-05-18 19:05:07,737 - ERROR - An error has
```

Logging Levels

- Benefit of logging levels is that you can change what priority of logging message you want to see.
- Passing logging.DEBUG to basicConfig() function’s level keyword argument will show messages from all the logging levels (DEBUG being lowest level)
- But after developing program some more, you may be interested only in errors.
- In that case, you can set basicConfig()’s level argument to logging.ERROR.
- This will show only ERROR and CRITICAL messages and skip the DEBUG, INFO, and WARNING messages.

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format='
%(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2019-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2019-05-18 19:04:35,569 - INFO - The logging module is
working.
>>> logging.warning('An error message is about to be
logged.')
2019-05-18 19:04:56,843 - WARNING - An error message is
about to be logged.
>>> logging.error('An error has occurred.')
2019-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2019-05-18 19:05:45,794 - CRITICAL - The program is unable
```

Disabling Logging

- After you've debugged your program, you probably don't want all these log messages cluttering the screen.
- 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.
- You simply pass logging.disable() a logging level, and it will suppress all log messages at that level or lower.
- So if you want to disable logging entirely, just add logging.disable(logging.CRITICAL) to to your program.
- Ex: enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format='
%(asctime)s - %(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2019-05-22 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.
- This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

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, like so:

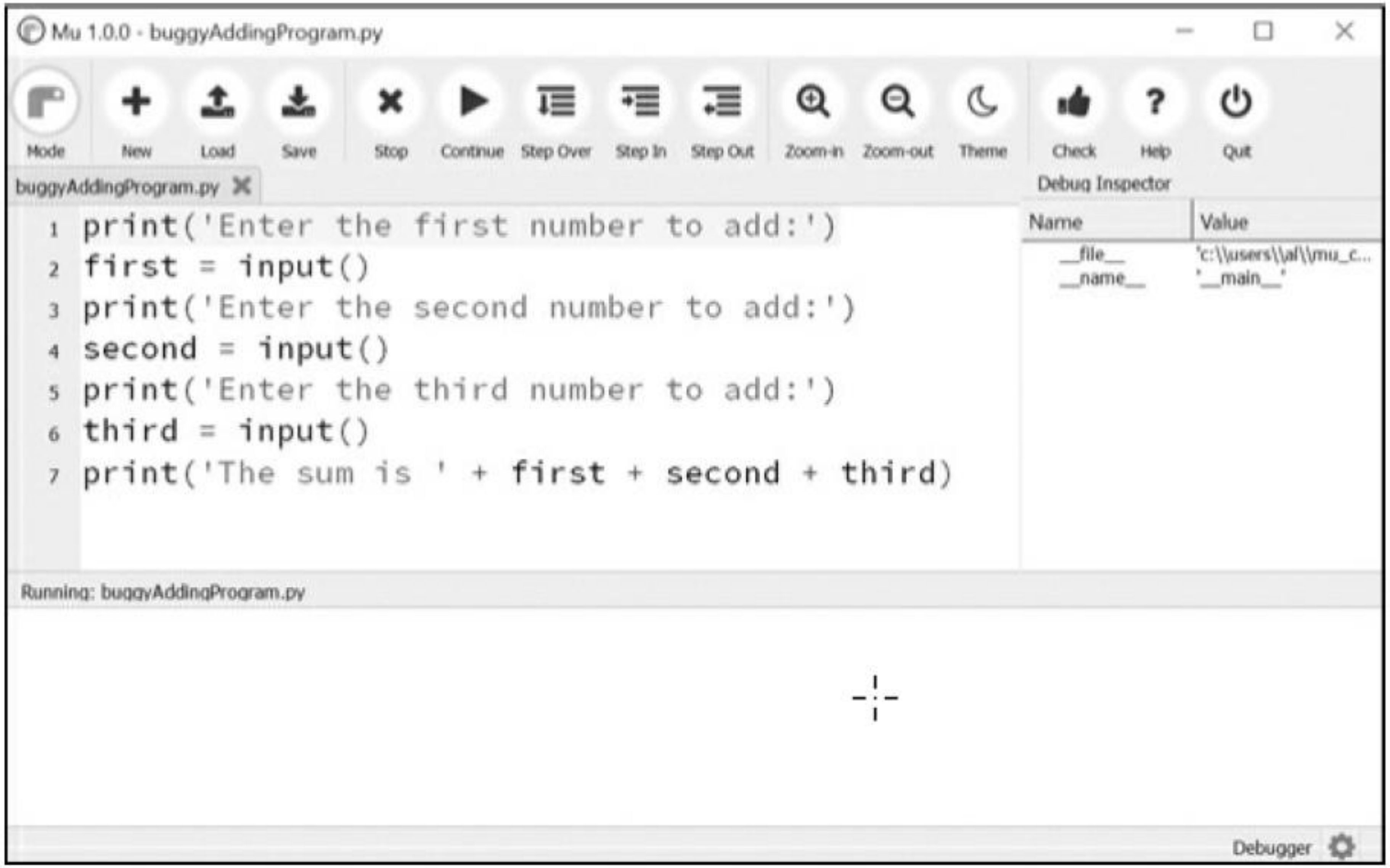
```
import logging
logging.basicConfig(filename='myProgramLog.txt',
level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

- The log messages will be saved to myProgramLog.txt.
- While logging messages are helpful, they can clutter your screen and make it hard to read the program’s output.
- Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program.
- You can open this text file in any text editor, such as Notepad or TextEdit.

Mu's 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 values in variables at any given point during the program's lifetime.
- This is a valuable tool for tracking down bugs.
- 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. You can see this line highlighted in the file editor.

Mu running a program under the debugger



Mu 1.0.0 - buggyAddingProgram.py

buggyAddingProgram.py

```python
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)
```

Debug Inspector

Name	Value
__file__	'c:\\\\users\\\\al\\\\mu_c...
__name__	'__main__'

Running: buggyAddingProgram.py

Mu's Debugger

- Debugging mode also adds the following new buttons to the top of the editor: Continue, Step Over, Step In, and Step Out.
- The usual Stop button is also available.
- Continue:
  - ✓ Clicking the Continue button will cause the program to execute normally until it terminates or reaches a breakpoint.
  - ✓ (I will describe breakpoints later in this chapter.) 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 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.
- For example, if the next line of code calls a spam() function but you don't really care about code inside this function, you can click Step Over to execute the code in the function at normal speed, and then pause when the function returns.
- For this reason, using the Over button is more common than using the Step In button.

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.

Debugging a Number Adding Program

- Open a new file editor tab and enter the following code:
- Save it as buggyAddingProgram.py & run it first without debugger enabled.

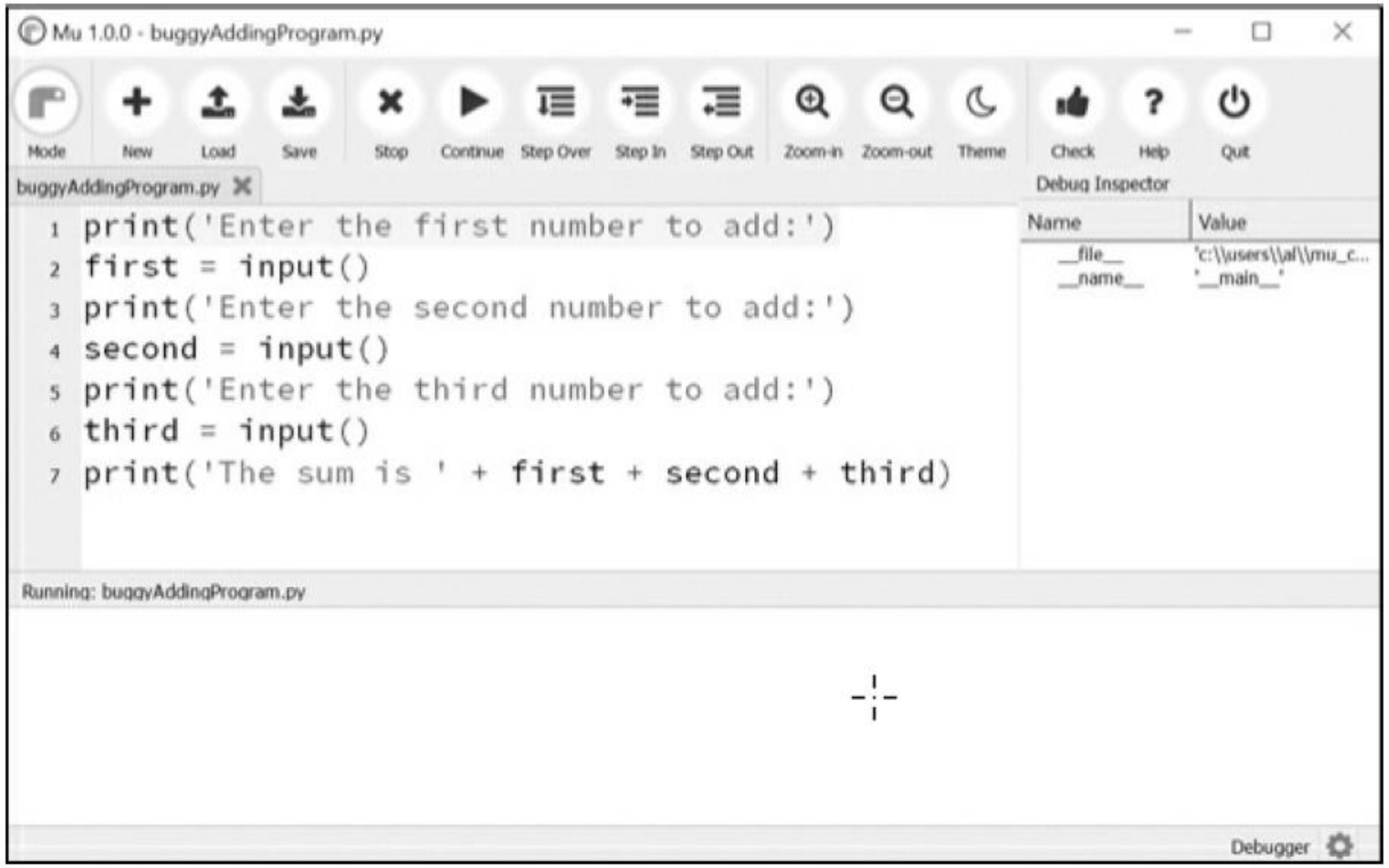
```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second +
third)
```

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
```

Debugging a Number Adding Program

- The program hasn't crashed, but the sum is obviously wrong.
- Run the program again, this time under the debugger.
- When you click the Debug button, the program pauses on line 1, which is the line of code it is about to execute.



```
1 print('Enter the first number to add:')
2 first = input()
3 print('Enter the second number to add:')
4 second = input()
5 print('Enter the third number to add:')
6 third = input()
7 print('The sum is ' + first + second + third)
```

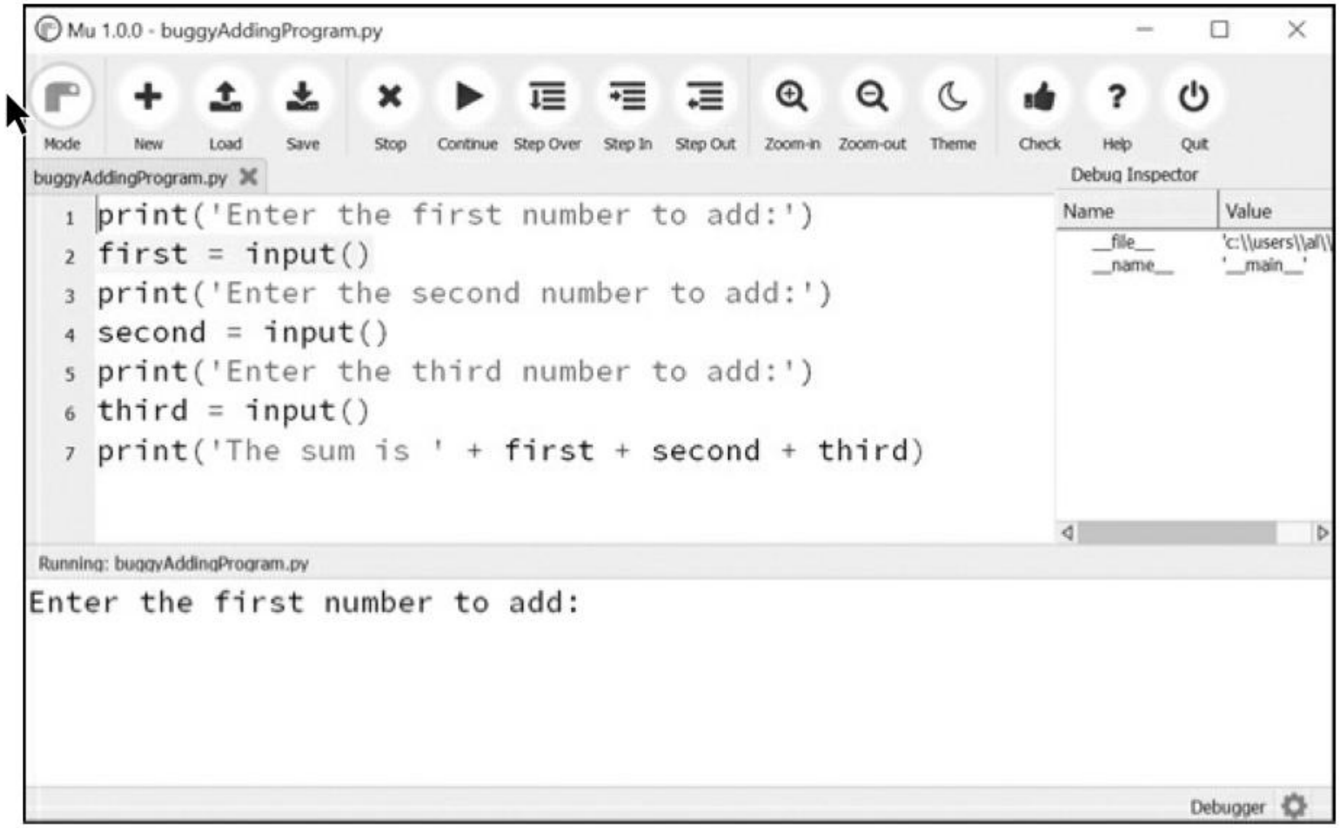
Name	Value
__file__	'c:\\users\\al\\mu_c...
__name__	'__main__'

Running: buggyAddingProgram.py

Debugging a Number Adding Program

- ➢ Click Step Over button once to execute first print() call.
- ➢ Use Step Over instead of Step In here, since you don't want to step into the code for the print() function. (Although Mu should prevent the debugger from entering Python's built-in functions.)

The Mu editor window after clicking Step Over



```
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)
```

Name	Value
__file__	'c:\\users\\al\\
__name__	'__main__'

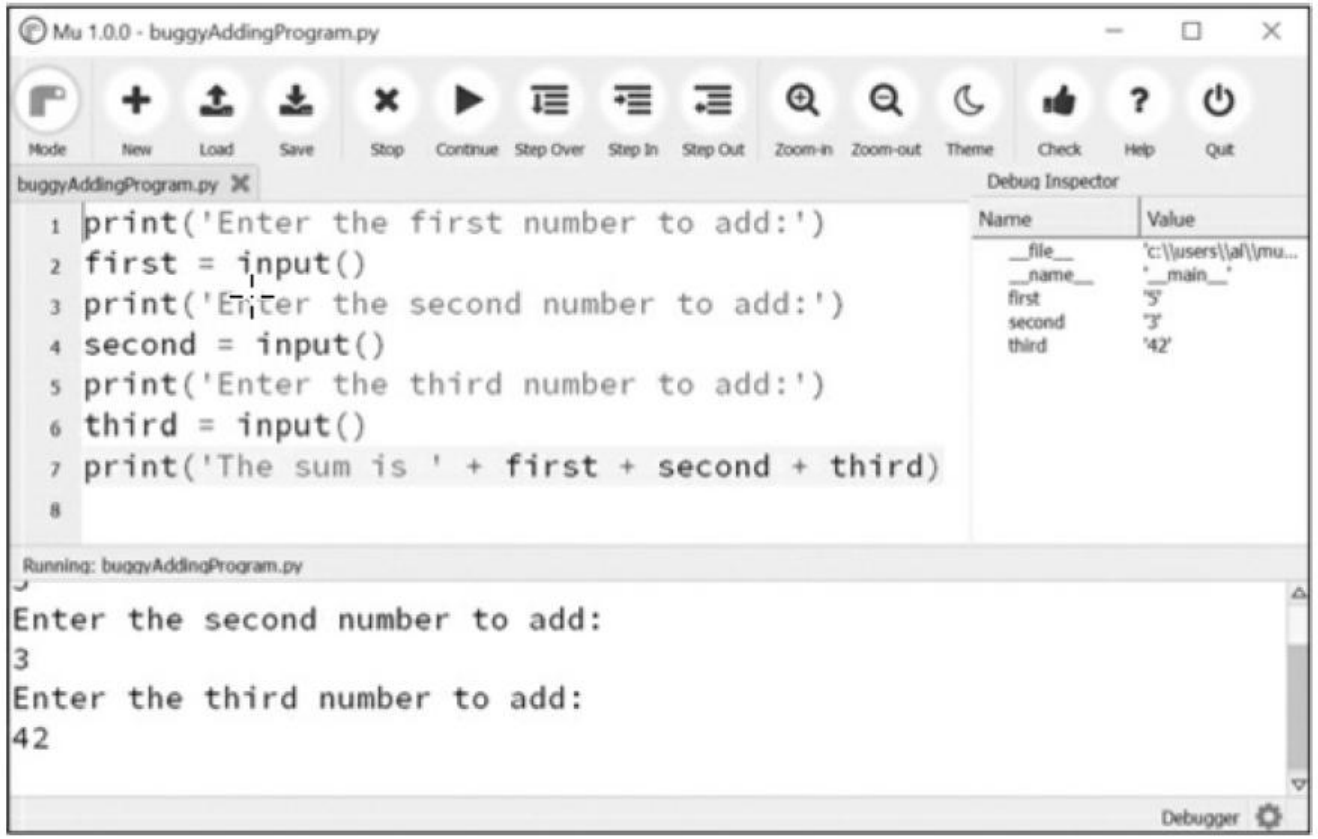
```
Enter the first number to add:
```

- ➢ The debugger moves on to line 2, and highlights line 2 in the file editor, as shown in Figure. This shows where program execution currently is.

Debugging a Number Adding Program

- Click Step Over again to execute input() function call.
- The highlighting will go away while Mu waits for you to type something for the input() call into the output pane.
- Enter 5 and press ENTER.
- The highlighting will return.
- Keep clicking Step Over, & enter 3 and 42 as next two numbers.
- When debugger reaches line 7, final print() call in program, Mu editor window should look like Figure:

The Debug Inspector pane on the right side shows that the variables are set to strings instead of integers, causing the bug.



```
1 print('Enter the first number to add:')
2 first = input()
3 print('Enter the second number to add:')
4 second = input()
5 print('Enter the third number to add:')
6 third = input()
7 print('The sum is ' + first + second + third)
8
```

Name	Value
__file__	'c:\\users\\al\\mu...
__name__	'__main__'
first	'5'
second	'3'
third	'42'

```
Enter the second number to add:
3
Enter the third number to add:
42
```

Debugging a Number Adding Program

- In the Debug Inspector pane, you should see 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, Python concatenates these strings instead of adding the numbers together, causing the bug.
- Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

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 tab and enter the following program, which simulates flipping a coin 1,000 times. Save it as coinFlip.py.

```
import random
heads = 0
for i in range(1, 1001):
❶ if random.randint(0, 1) == 1:
          heads = heads + 1
if i == 500:
          ❷ print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

- The random.randint(0, 1) call ❶ 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.

Breakpoints

- When you run this program without the debugger, it quickly outputs something like the following:

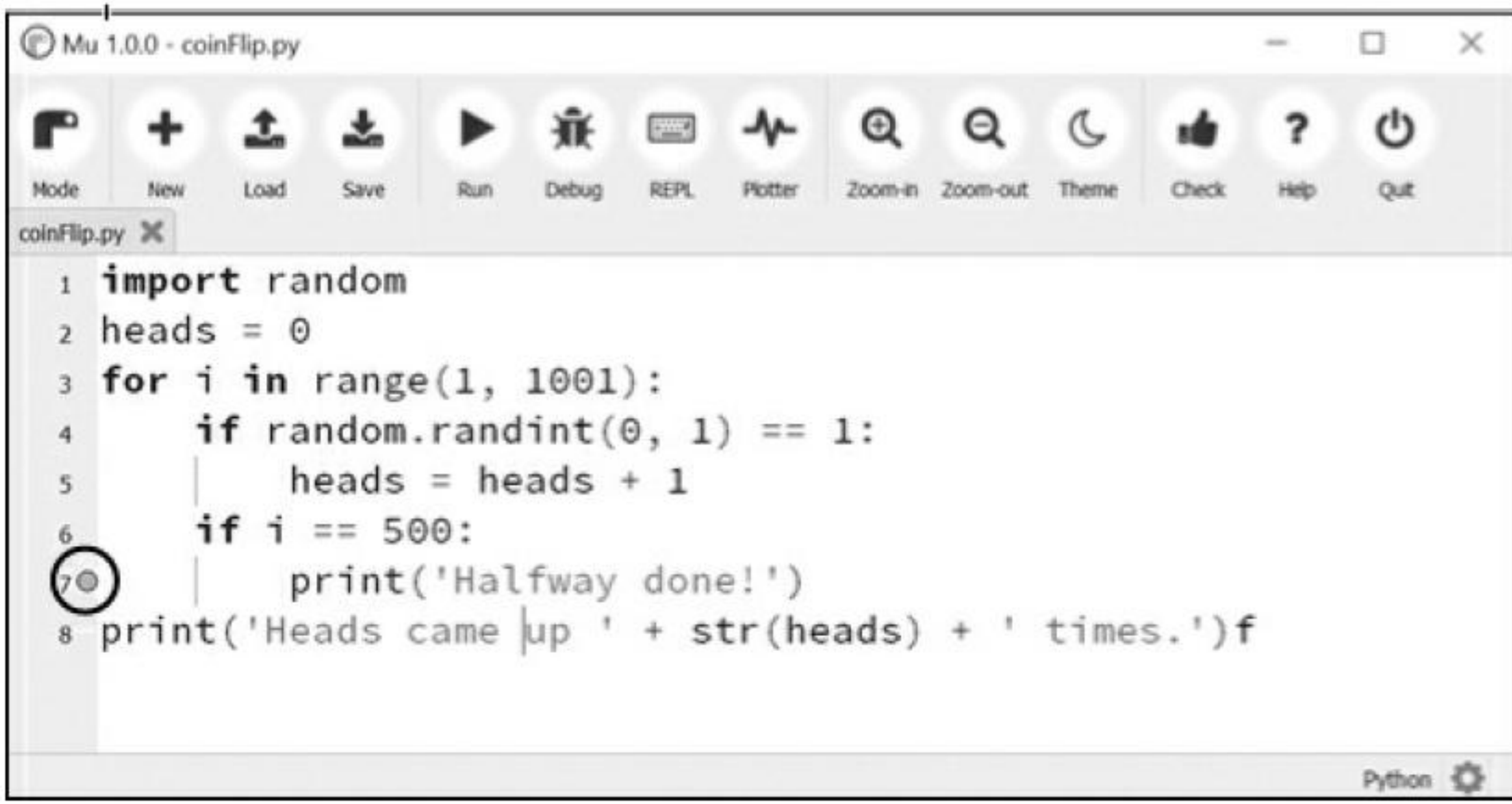
```
Halfway done!
Heads came up 490 times.
```

- If you ran this program under the debugger, you would have to click the Step 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 1,000 coin flips have been completed, you could instead just set a breakpoint on the line print('Halfway done!') ➋.
- To set a breakpoint, click the line number in the file editor to cause a red

Breakpoints

- To set a breakpoint, click the line number in the file editor to cause a red dot to appear, marking the breakpoint like in Figure:

Setting a breakpoint causes a red dot (circled) to appear next to the line number.



```
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.')f
```

Breakpoints

- You don’t want to set a breakpoint on the if statement line, since if statement is executed on every single iteration through loop.
- When you set the breakpoint on the code in the if statement, the debugger breaks only when the execution enters the if clause.
- The line with the breakpoint will have a red dot next to it.
- When you run the program under the debugger, it will start in a paused state at the first line, as usual.
- But if you click Continue, the program will run at full speed until it reaches the line with the breakpoint set on it.
- Then click Continue, Step Over, Step In, or Step Out to continue as normal.
- If you want to remove a breakpoint, click the line number again.
- The red dot will go away, and debugger will not break on that line in future.