# 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

### 1.1 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.
- If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

**Program:**

```
>>> import shutil, os
>>> os.chdir('C:\\')
u>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
 >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

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

**Program:**

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

- The shutil.copytree() call creates a new folder named *bacon_backup* with the same content as the original *bacon* folder

## 1.2 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() calls says, "Move C:\bacon.txt into the folder C:\eggs."
- The *destination* path can also specify a filename

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

- The folders that make up the destination must already exist, or else Python will throw an exception

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
    File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
    'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
    File "<pyshell#29>", line 1, in <module>
        shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
    File "C:\Python34\lib\shutil.py", line 533, in move
        copy2(src, real_dst)
    File "C:\Python34\lib\shutil.py", line 244, in copy2
        copyfile(src, dst, follow_symlinks=follow_symlinks)
    File "C:\Python34\lib\shutil.py", line 108, in copyfile
        with open(dst, 'wb') as fdst:
    FileNotFoundError: [Errno 2] No such file or directory:
'c:\\does_not_exist\\eggs\\ham'
```



## 1.3 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.



## 1.4 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')
```

## 2. 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.

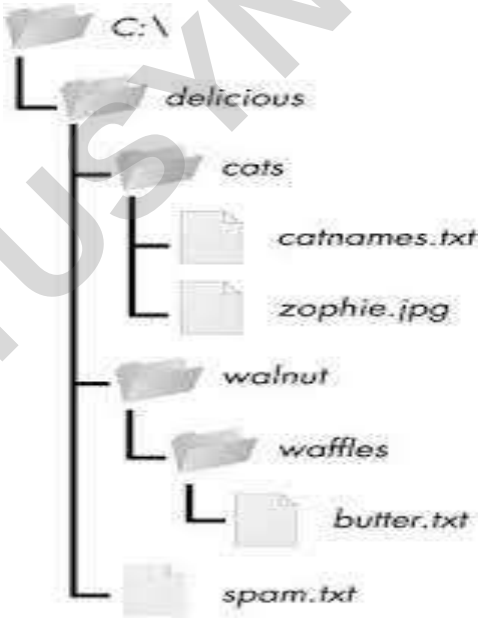


Figure: An example folder that contains three folders and four files

```
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.
```

## 3. 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.

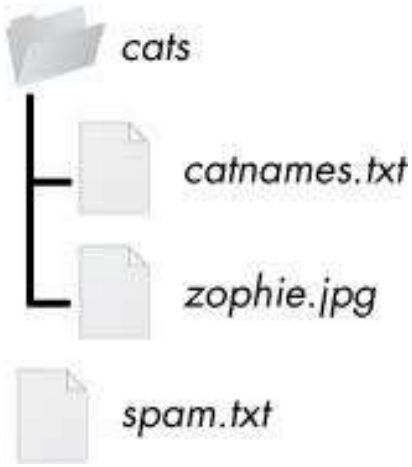


*Figure : The contents of example.zip*

### 3.1 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.

## 3.2 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()
```



### *3.3 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()
```

### 4. 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().

#### *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.

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.
 import shutil, os, re

# Create a regex that matches files with the American date format.
 datePattern = re.compile(r"""^(.*?)     # all text before the date
                ((0|1)?\d)-     # one or two digits for the month
((0|1|2|3)?\d)-                 # one or two digits for the day
((19|20)\d\d)                   # four digits for the year
(.*?)$                           # all text after the date
""", re.VERBOSE)

# 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.
```

- After importing the re module at the top, call re.compile() to create a Regex object
- Passing re.VERBOSE for the second argument w will allow whitespace and comments in the regex string to make it more readable.

***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.

- The matched text will be stored in several variables

```
#! 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.
     if mo == None:
    continue

 # 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--
```

- 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 the code, just write an outline of the regular expression.

```
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)
```

### *Step 3: Form the New Filename and Rename the Files*

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

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date
format
# to European DD-MM-YYYY.

--snip--
        # Form the European-style filename.
        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.
print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
 #shutil.move(amerFilename, euroFilename)          # uncomment after testing
```

## Project: Backing Up a Folder into a ZI P File

### *Step 1: Figure Out the ZIP File's Name*

- The code for this program will be placed into a function named backupToZip(). This will make it easy to copy and paste the function into other Python programs that need this functionality.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

import zipfile, os
    def backupToZip(folder):
    # Backup 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.
     number = 1
    while True:
    zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipFilename):
            break
    number = number + 1
 # TODO: Create the ZIP file.
# TODO: Walk the entire folder tree and compress the files in each folder.
print('Done.')
backupToZip('C:\\delicious')
```

- Add the shebang (#!) line, describe what the program does, and import the zipfile and os modules
- Define a backupToZip() function that takes just one parameter, folder. This parameter is a string path to the folder whose contents should be backed up.

- The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder folder, and each of the subfolders and files to the ZIP file.

### *Step 2: Create the New ZIP File*

```
#! 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('Creating %s...' % (zipFilename))
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 and Add to the 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

```
#! 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.
 for foldername, subfolders, filenames in os.walk(folder):
        print('Adding files in %s...' % (foldername))
# Add the current folder to the ZIP file.
backupZip.write(foldername)
# Add all the files in this folder to the ZIP file.
         for filename in filenames:
                newBase / os.path.basename(folder) + '_'
                if filename.startswith(newBase) and filename.endswith('.zip')
                continue                        # don't backup the backup ZIP files
        backupZip.write(os.path.join(foldername, filename))
backupZip.close()
print('Done.')

backupToZip('C:\\delicious')


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.
```

# CHAPTER -2 DEBUGGING

## 1. Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code.
- Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the except statement."
- Exceptions are raised with a raise statement. In code, a raise statement consists of the following:
- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function
- 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.
```

### 2. 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.

```
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.
```



- The traceback is displayed by Python 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.
- For example,

```
>>> 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.')
116
The traceback info was written to errorInfo.txt.
```

- The 116 is the return value from the write() method, since 116 characters were written to the file.

```
Traceback (most recent call last):
File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```



## 3. 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.
- 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
- For example,

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that."
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

- set podBayDoorStatus to 'open', so from now on, we fully expect the value of this variable to be 'open'
- In a program that uses this variable, we might have written a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work as we expect. So we add an assertion to make sure we're right to assume podBayDoorStatus is 'open'
- Here, we include the message 'The pod bay doors need to be "open".' so it'll be easy to see what's wrong if the assertion fails.

### *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

## 4. 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,

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

- 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

**Program:**

```
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')
```

## 5. 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.

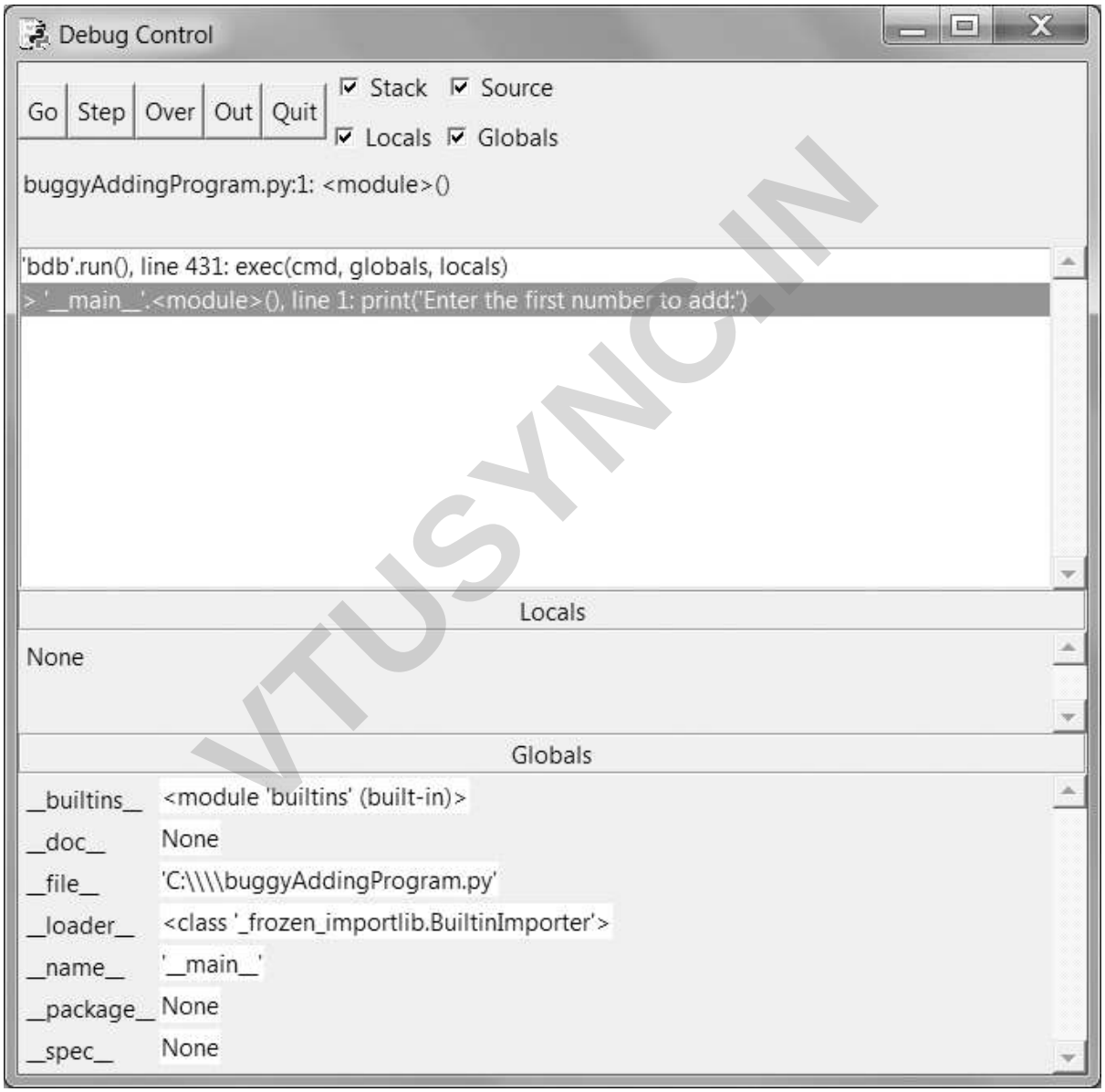


Figure The Debug Control window when the program first starts under the debugger

- 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.

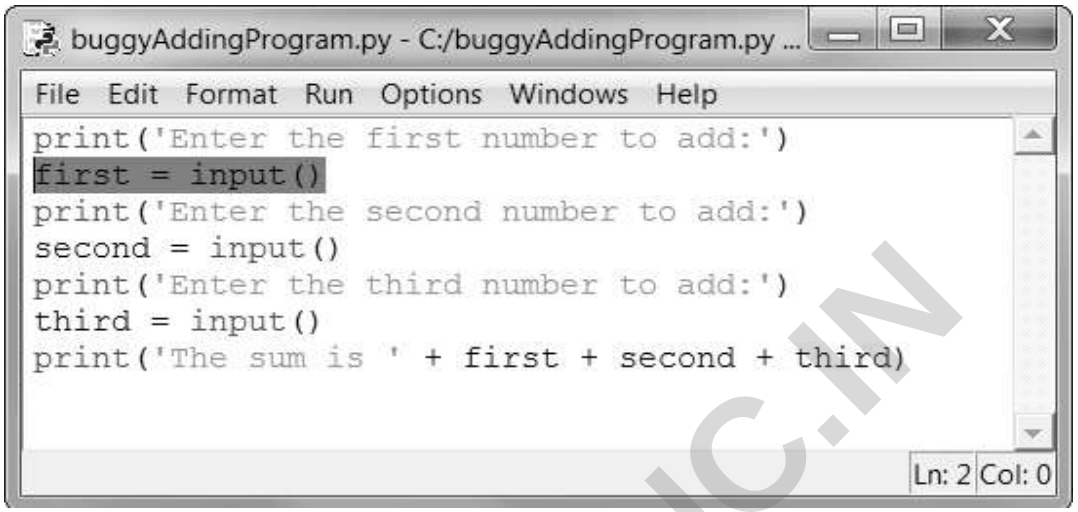


Figure: The Debug Control window after clicking Over

- 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.

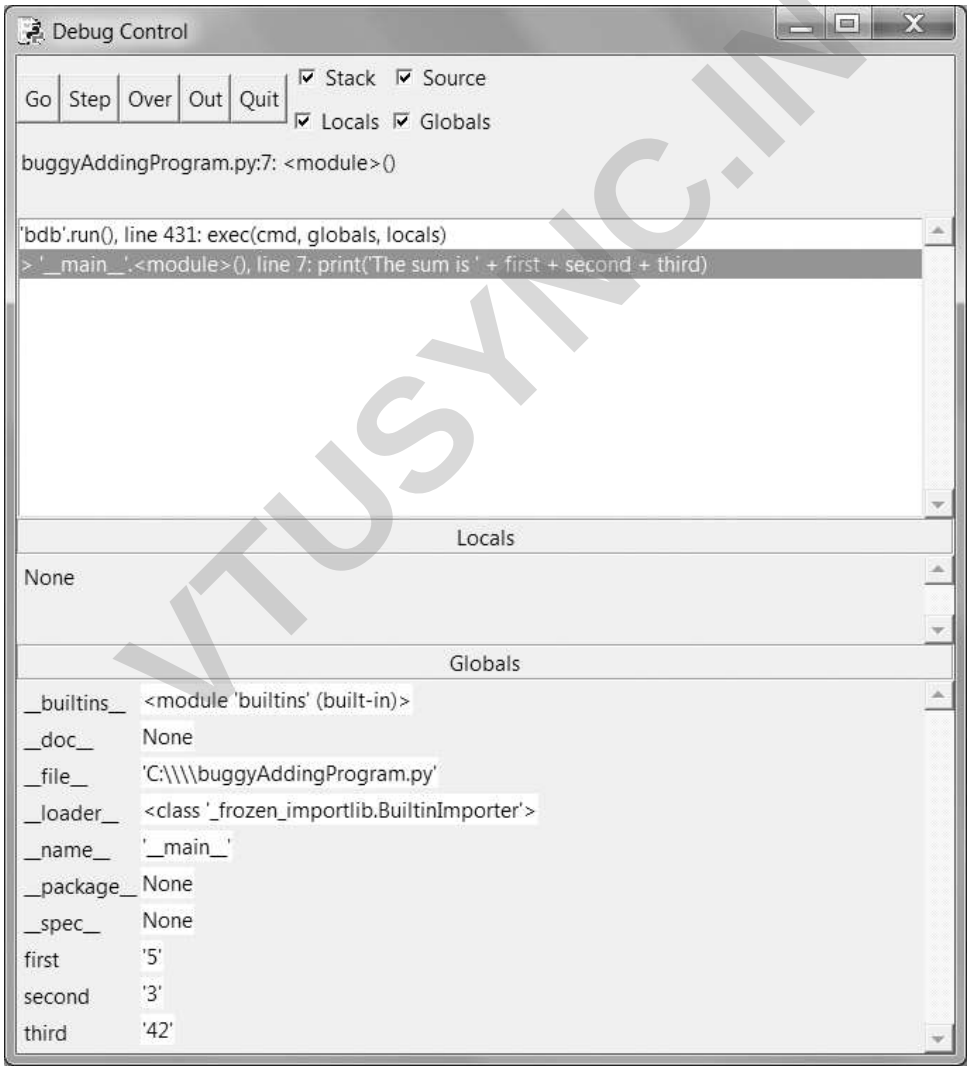


Figure The Debug Control window on the last line. The variables are set to strings, causing the bug.

## Breakpoints

- A *breakpoint* can be set on a specific line of code and forces the debugger to mpause 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,

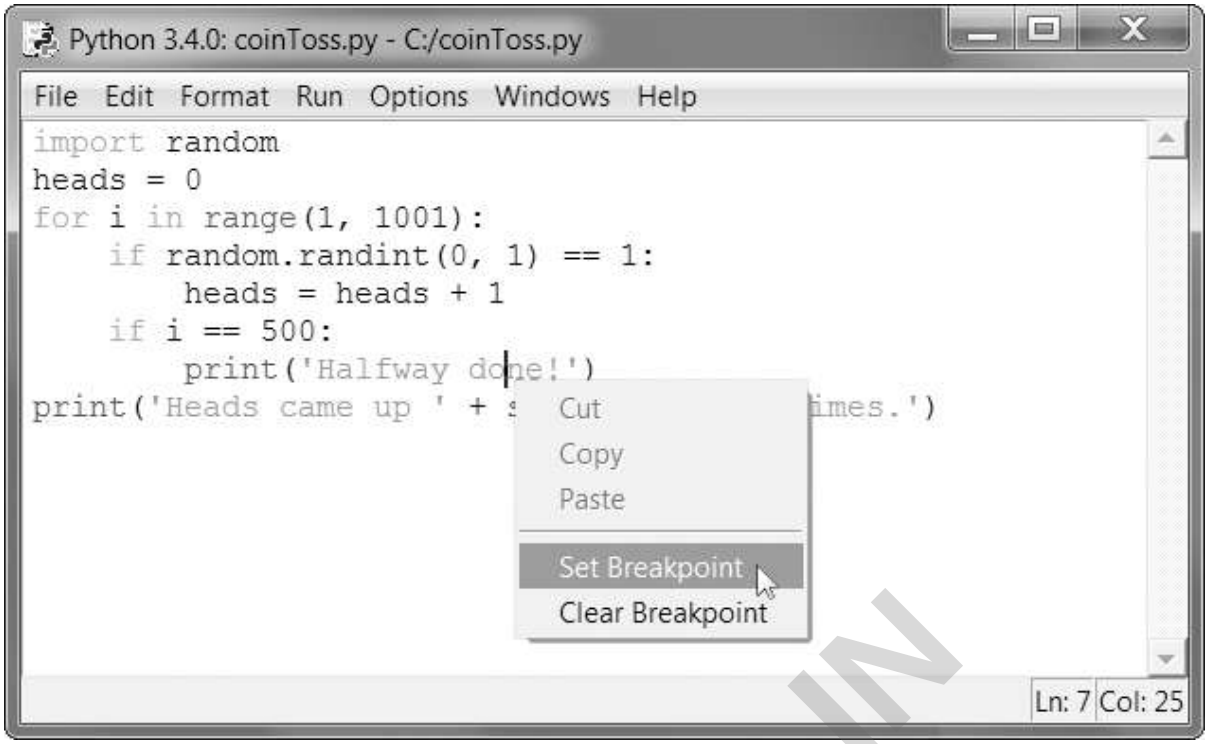


Figure : Setting a breakpoint