# Files

# Introduction

- primarily focus on reading and writing text files such as those we create in a text editor.
- Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.
- Only means to store data permanently.

# Opening files

- When we want to read or write a file (say on your hard drive), we first must *open the file.*
- *Opening the file communicates with your operating system, which* knows where the data for each file is stored.
- When you open a file, you are asking the operating system to find the file by name and make sure the file exists.
- In this example, we open the file mbox.txt, which should be stored in the same folder that you are in when you start Python.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252‘>
```

- If the open is successful, the operating system returns us a *file handle.*
- *The file* handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data.
- You are given a handle if the requested file exists and you have the proper permissions to read the file.

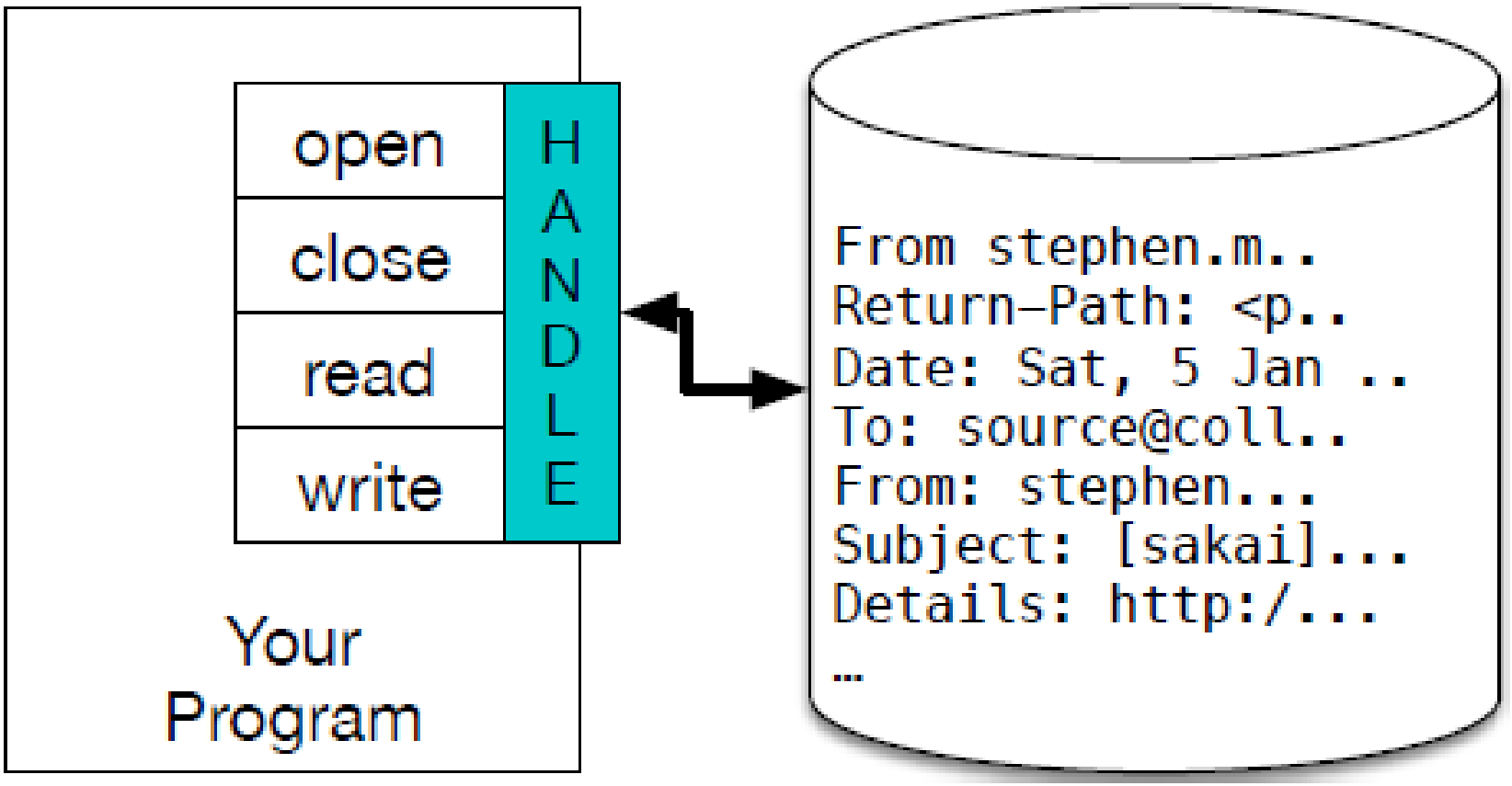



Figure 7.2: A File Handle

- If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt‘
```

- Later we will use try and except to deal more gracefully with the situation where we attempt to open a file that does not exist.

# Python open() Function

- The open() function opens a file, and returns it as a file object.

open(*file, mode*)

# Open Parameter Description

- *File:* The path and name of the file
- *Mode:* defines which mode you want to open the file in:
  - "r" - Read - Default value. Opens a file for reading, error if the file does not exist
  - "a" - Append - Opens a file for appending, creates the file if it does not exist
  - "w" - Write - Opens a file for writing, creates the file if it does not exist
  - "x" - Create - Creates the specified file, returns an error if the file exist

# Other Modes

- In addition you can specify if the file should be handled as binary or text mode
  - "t" - Text - Default value. Text mode
  - "b" - Binary - Binary mode (e.g. images)

# Example 1.py

- Write a Python program to read an entire text file.

# 1.py

- def file_read(fname):
- txt = open(fname)
- print(txt.read())

- file_read(input('Enter the File Name with path'))

# Text files and lines

- A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters.
- For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Return-Path: <postmaster@collab.sakaiproject.org>

Date: Sat, 5 Jan 2008 09:12:18 -0500

To: source@collab.sakaiproject.org

From: stephen.marquard@uct.ac.za

Subject: [sakai] svn commit: r39772 - content/branches/

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772

...

# Example 2.py

- Write a Python program to count the number of lines in the file.

# 2.py

```
def linecount(fname):
    fhand = open(fname)
    count = 0
    for line in fhand:
        count = count + 1
    print('Line Count:', count)

linecount(input("Enter the File PathName:"))
```

# Example 3.py

- Write a Python program to read specified number of lines from the specified file.

# 3.py

```
def file_read_from_head(fname, nlines):
        from itertools import islice
        f=open(fname)
        for line in islice(f, nlines):
            print(line)
fname=input("Enter the Path Name of the FILE:")
nlines=input("Enter the number of lines to read:")

file_read_from_head(fname,int(nlines))
```

# 4.py

- Write a Python program to find the size of the specified file in terms of bytes.

# 4.py

```
def fsize(fname):
    fhand = open(fname)
    data=fhand.read()
    size=len(data)
    print('File size in bytes=', size)
    close(fhand)
fsize(input("Enter the File PathName:"))
```

# Example 5.py

.Write a Python program to read a file line by line store it into an array

# 5.py

```
def file_read(fname):
        content_array = []
        with open(fname) as f:
#Content_list is the list that contains the read lines.
                for line in f:
                        content_array.append(line)
                print(content_array)

file_read(input("Enter the file Name:"))
```

# Searching through a file

- if we wanted to read a file and only print out lines which started with the prefix “From:”,
- we could use the string method *startswith to select only those* lines with the desired prefix:

# Example 6.py

- Write a Python program to read only the lines starting with some given pattern.

# 6.py

```
def linematch(fname, pname):
    fhand = open(fname)
    count = 0
    for line in fhand:
        if line.startswith(pname):
            count=count+1
            print(line)
    print("Total Number of Lines:",count)

fname=input("Enter the File Name:")
pname=input("Enter the patter to search:")
linematch(fname,pname)
```

# Strip()

- 1. strip()-- strip spaces (left+right)
- 2.rstrip()– strip spaces (right)
- 3.lstrip()—strip spaces(left)

# Example 7.py

- Write a python program to strip the spaces on both the sides.

# 7.py

```
def file_read(fname):
        txt = open(fname)
        for line in txt:
            print(line.strip())
file_read(input('Enter the File Name with path'))
```

# Example 8.py

- Write a python program to strip the spaces on right sides.

# 8.py

```
def file_read(fname):
        txt = open(fname)
        for line in txt:
            print(line.rstrip())
file_read(input('Enter the File Name with path'))
```

# Example 9.py

- Write a python program to strip the spaces on left sides.

# 9.py

```
def file_read(fname):
        txt = open(fname)
        for line in txt:
            print(line.lstrip())
file_read(input('Enter the File Name with path'))
```

# Using try, except, and open

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
    count = count + 1
print('There were', count, 'subject lines in', fname)
```

# Writing files

- To write a file, you have to open it with mode "w" as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt'
  mode='w' encoding='cp1252'>
```

# Example 11.py

- Write a Python program to write a list content to a file.

# 11.py

- color = ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']
- with open('abc.txt', "w") as myfile:
- for c in color:
- myfile.write("%s\n" % c)

- content = open('abc.txt')
- print(content.read())

# Assess if a file is closed or not.

- Close() – Closes the file
- Closed() – returns true if closed

# Example 12.py

- Write a python program to assess whether the file closed or not

# 12.py

- f = open('abc.txt','r')
- print(f.closed)
- f.close()
- print(f.closed)

# Zip() function

- **The zip() function take iterators (can be zero or more).**
- **Makes an iterator that aggregates elements based on the iterators passed, and returns an iterator of tuples.**

# zip() Parameters

- The zip() function takes:
- **iterables** - can be built-in iterables (like: list, string, dict), or user-defined iterables (object that has __iter__ method).

# Python Iterators

- **Iterators are objects that can be iterated upon.**
- **Here, you will learn how iterator works and how you can build your own iterator using __iter__ and __next__ methods.**

# What are iterators in Python?

- Iterators are everywhere in Python. They are elegantly implemented within for loops but hidden in plain sight.
- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- Technically speaking, Every Python **iterator object** must implement two special methods, __iter__()and __next__(), collectively called the **iterator protocol**.
- An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.
- The iter() function (which in turn calls the __iter__() method) returns an iterator from them.
-

# Example
# on
# **Iterating Through an Iterator in Python**

```
# define a list
my_list = [4, 7, 0, 3]
# get an iterator using iter()
my_iter = iter(my_list)
## iterate through it using next()
#prints 4
print(next(my_iter))
#prints 7
print(next(my_iter))
## next(obj) is same as obj.__next__()
#prints 0
print(my_iter.__next__())
#prints 3
print(my_iter.__next__())
## This will raise error, no items left
next(my_iter)
```

# A more elegant way of automatically iterating

- -- Using for loops

```
My_list = [4, 7, 0, 3]
for element in my_list:
 ... print(element)
...
 4 7 0 3
```

# Actual Implementation

- **For loop with an iterator is actually implemented as :**

```
for element in iterable:
    # do something with element
```

Internally is as follows:

```
# create an iterator object from that iterable
iter_obj = iter(iterable)
# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop break
        break
```

# Return Value from zip()

- The zip() function returns an iterator of tuples based on the iterable object.
- If no parameters are passed, zip() returns an empty iterator
- If a single iterable is passed, zip() returns an iterator of 1-tuples. Meaning, the number of elements in each tuple is 1.
- If multiple iterables are passed,

  Suppose, two iterables are passed; one iterable containing 3 and other containing 5 elements. Then, the returned iterator has 3 tuples. It's because iterator stops when shortest iterable is exhausted.

# How zip() works in Python?

```
numberList = [1, 2, 3]
strList = ['one', 'two', 'three']
# No iterables are passed
result = zip()
# Converting iterator to list
resultList = list(result)
print(resultList)
# Two iterables are passed
result = zip(numberList, strList)
# Converting iterator to set
resultSet = set(result)
print(resultSet)
```

# Different Number of Elements in Iterables Passed to zip()

```
numbersList = [1, 2, 3]
strList = ['one', 'two']
numbersTuple = ('ONE', 'TWO', 'THREE', 'FOUR')

result = zip(numbersList, numbersTuple)

# Converting to set
resultSet = set(result)
print(resultSet)

result = zip(numbersList, strList, numbersTuple)

# Converting to set
resultSet = set(result)
print(resultSet)
```

# Unzipping the Value Using zip()

- The * operator can be used in conjunction with zip() to unzip the list.

zip(*zippedList)

**Example:**

```
coordinate = ['x', 'y', 'z']
value = [3, 4, 5, 0, 9]

result = zip(coordinate, value)
resultList = list(result)
print(resultList)

c, v =  zip(*resultList)
print('c =', c)
print('v =', v)
```

# Example 13.py

- Write a Python program to combine each line from first file with the corresponding line in second file and write to the third file

# 13.py

```
with open('abc.txt') as fh1, open(‘xyz.txt') as fh2, open('merged','w') as fw:
    for line1, line2 in zip(fh1, fh2):
        # line1 from abc.txt, line2 from test.txt
        print(line1+line2)
        fw.write(line1+line2+"\n")
```

# Example 14.py

- Write a Python program to remove newline characters from a file.

# 14.py

```
def remove_newlines(fname):
    flist = open(fname)
    for line in flist:
        print(line.rstrip("\n"))

print(remove_newlines("abc.txt"))
```

# Example 15.py

- Write a Python program to combine each line from first file with the corresponding line in second file by eliminating new line and write to the third file

# 15.py

```
with open('abc.txt') as fh1, open('abc.txt') as fh2, open('d:\Python\merged.txt','w') as fw:
    for line1, line2 in zip(fh1, fh2):
        # line1 from abc.txt, line2 from test.txtg
        print(line1.rstrip('\n')+ ' ' + line2.rstrip('\n'))
        fw.write(line1.rstrip('\n')+ ' ' + line2.rstrip('\n')+"\n")
```

# The current file position

- Every file maintains a ***current file position***.
- It is the current position in the file, and indicates what the file will read next

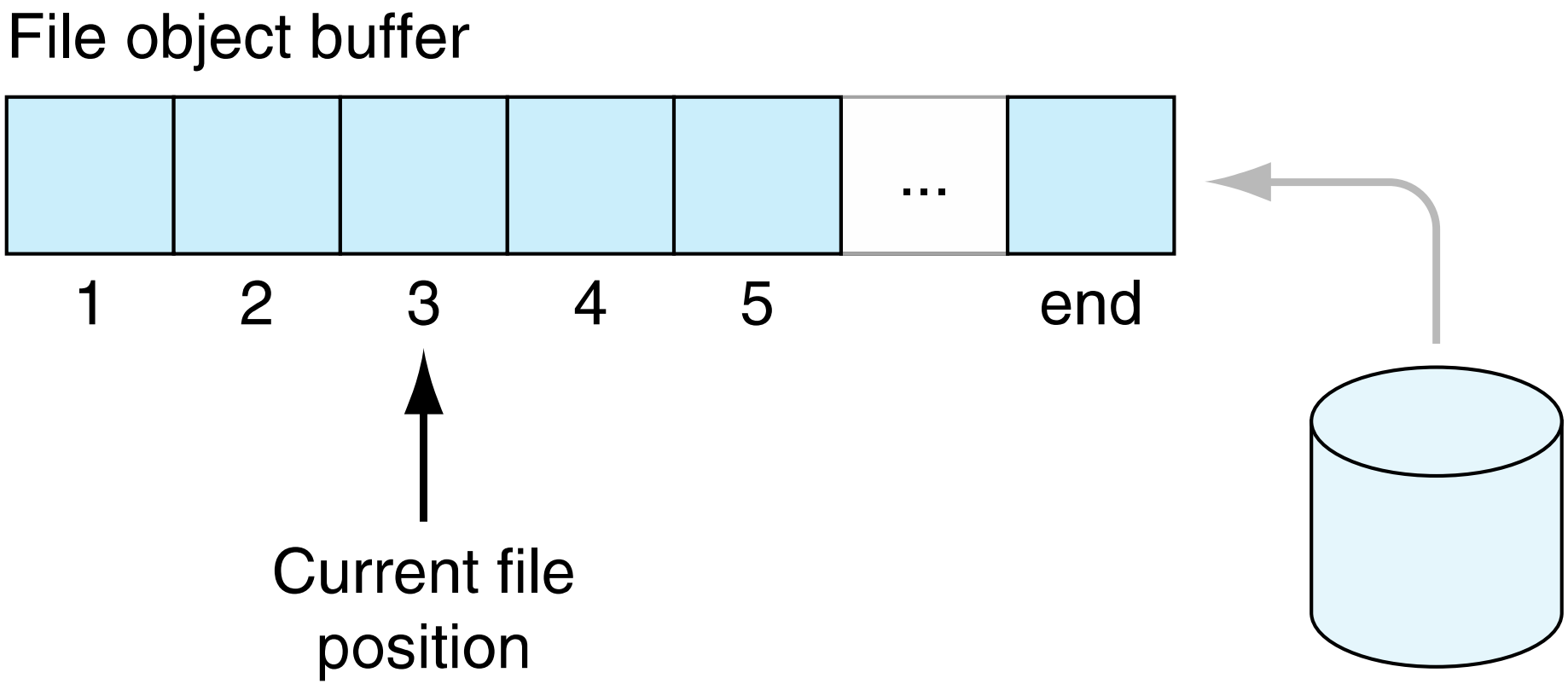



**FIGURE 14.1** Current file position.

# the tell() method

- The `tell()` method tells you the current file position
- The positions are in from the beginning of the file.

my_file=open(‘abc.txt’, ‘r’)

My_file.read(42)

my_file.tell() = 42

# the seek() method

- the `seek()` method updates the current file position to a new file index (in bytes offset from the beginning of the file)
- fd.seek(0) # to the beginning of the file
- fd.seek(100) # 100 bytes from beginning

# counting bytes is a pain

- counting bytes is a pain
- `seek` has an optional argument set:
  - 0: count from the beginning
  - 1: count for the current file position
  - 2: count from the end (backwards)

# Every read moves current pos forward

- every read/readline/readlines moves the current pos forward
- when you hit the end, every read will just yield `''` (empty string), since you are at the end
  - no indication of end-of-file this way!
- you need to seek to the beginning to start again (or close and open, seek is easier)

# example file

We'll work with a file called `temp.txt` which has the following file contents

```
First Line
Second Line
Third Line
Fourth Line
```

```
>>> test_file = open('temp.txt','r')
>>> test_file.tell()            # where is the current file position?
0
>>> test_file.readline()        # read first line
'First Line\n'
>>> test_file.tell()            # where are we now?
11
>>> test_file.seek(0)           # go to beginning
0
>>> test_file.readline()        # read first line again
'First Line\n'
>>> test_file.readline()        # read second line
'Second Line\n'
>>> test_file.tell()            # where are we now?
23
>>> test_file.seek(0,2)         # go to end
46
>>> test_file.tell()            # where are we now?
46
>>> test_file.readline()        # try readline at end of file: nothing there
''
>>> test_file.seek(11)          # go to the end of the first line (see tell above)
11
>>> test_file.readline()        # when we read now we get the second line
'Second Line\n'
>>> test_file.close()
>>> test_file.readline()        # Error: reading after file is closed
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    test_file.readline()
ValueError: I/O operation on closed file.
>>>
```

# Advantage of opening files using with

File is closed automatically when the suite ends

```
>>> with open('temp.txt') as temp_file:
...     temp_file.readlines()
...
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']
>>>
```

# Working with CSV Files

# Spread Sheets

- The spreadsheet is a very popular, and powerful, application for manipulating data
- Its popularity means there are many companies that provide their own version of the spreadsheet
- It would be nice if those different versions could share their data

# CSV, basic sharing

- A basic approach to share data is the comma separated value (CSV) format
  - it is a text format, accessible to all apps
  - each line (even if blank) is a row
  - in each row, each value is separated from the others by a comma (even if it is blank)
  - cannot capture complex things like formula

# Spread sheet and corresponding CSV file

| Name | Exam1 | Exam2 | Final Exam | Overall Grade |
|---|---|---|---|---|
| Bill | 75.00 | 100.00 | 50.00 | **75.00** |
| Fred | 50.00 | 50.00 | 50.00 | **50.00** |
| Irving | 0.00 | 0.00 | 0.00 | **0.00** |
| Monty | 100.00 | 100.00 | 100.00 | **100.00** |
| | | | | |
| Average | | | | **56.25** |

**FIGURE 14.2** A simple spreadsheet from Microsoft Excel 2008.

```
Name,Exam1,Exam2,Final Exam,Overall Grade
Bill,75.00,100.00,50.00,75.00
Fred,50.00,50.00,50.00,50.00
Irving,0.00,0.00,0.00,0.00
Monty,100.00,100.00,100.00,100.00

Average,,,,56.25
```

# Even CSV isn't universal

- As simple as that sounds, even CSV format is not completely universal
  - different apps have small variations
- Python provides a module to deal with these variations called the csv module
- This module allows you to read spreadsheet info into your program

# csv reader

- import the csv module
- open the file as normally, creating a file object.
- create an instance of a csv reader, used to iterate through the file just opened
  - you provide the file object as an argument to the constructor
- iterating with the reader object yields a row as a list of strings

# Example

```
import csv
workbook_file = open('Workbook1.csv','r')
workbook_reader = csv.reader(workbook_file)

for row in workbook_reader:
    print(row)

workbook_file.close()
```

```
>>>
['Name', 'Exam1', 'Exam2', 'Final Exam', 'Overall Grade']
['Bill', '75.00', '100.00', '50.00', '75.00']
['Fred', '50.00', '50.00', '50.00', '50.00']
['Irving', '0.00', '0.00', '0.00', '0.00']
['Monty', '100.00', '100.00', '100.00', '100.00']
[]
['Average', '', '', '', '56.25']
>>>
```

# things to note

- Universal new line is working by default
  - needed for this worksheet
- A blank line in the CSV shows up as an empty list
- empty column shows up as an empty string in the list

# Working Example

```
import csv #in built module


with open("data2.csv","r") as myfile: #
    #wr=csv.writer(myfile,dialect="excel")
    #wr.writerow([name,gender,email])
    reader=csv.reader(myfile)
    for row in reader:
        print(row)
```

# csv writer

much the same, except:

- the opened file must have write enabled
- the method is writerow, and it takes a ***list of strings*** to be written as a row
- A dialect is a class of csv module which helps to define parameters for reading and writing CSV. It allows you to create, store, and re-use various formatting parameters for your data.

# Working Example

```
import csv #in built module
with open("data1.csv","w") as myfile:
    wr=csv.writer(myfile,dialect="excel")

    wr.writerow(["Harish","Male","harish.bitcse82@gmail.
    com"])

    wr.writerow(["abch","Male","abc.bitcse82@gmail.com
    "])

    wr.writerow(["akkc","Male","harish.bitcse82@gmail.co
    m"])
```

# Working Example-2

```
import csv #in built module
with open("data2.csv","a”,newline=“”) as myfile:
    wr=csv.writer(myfile,dialect="excel")

    wr.writerow(["Harish","Male","harish.bitcse82@gmail.
    com"])

    wr.writerow(["abch","Male","abc.bitcse82@gmail.com
    "])

    wr.writerow(["akkc","Male","harish.bitcse82@gmail.co
    m"])
```

- Write a python program to read Name, Gender and Email and write to the csv file as a row.

```
import csv #in built module

name=input("Name:")
gender=input("Gender:")
email=input("Email:")
with open("data2.csv","a", newline="") as myfile:
    wr=csv.writer(myfile,dialect="excel")
    wr.writerow([name,gender,email])
```

# Different delimiter

```
import csv
f = open('items.csv', 'r‘)
    reader = csv.reader(f, delimiter="|")
    for row in reader:
        for e in row:
            print(e)
```

# Registering the dialect

- import csv
- csv.register_dialect("hashes", delimiter="#")
- With open('items3.csv', 'w') as f:

```
    writer = csv.writer(f, dialect="hashes")
    writer.writerow(("pens", 4))
    writer.writerow(("plates", 2))
    writer.writerow(("bottles", 4))
    writer.writerow(("cups", 1))
```

# OS MODULE

- **PATHS are built using Backslash on Windows and Forward Slash on OS X and Linux.**

- If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.
- Fortunately, this is simple to do with the os.path.join() function.
- If you pass it the string values of individual file and folder names in your path,

os.path.join()

- will return a string with a file path using the correct path separators.

- >>> **import os**
- >>> **os.path.join('usr', 'bin', 'spam')**

- 'usr\\bin\\spam'
- I’m running these interactive shell examples on Windows, so os.path.join('usr','bin', 'spam') returned 'usr\\bin\\spam'.
- (Notice that the backslashes are doubledbecause each backslash needs to be escaped by another backslash character.)

# create strings for filenames

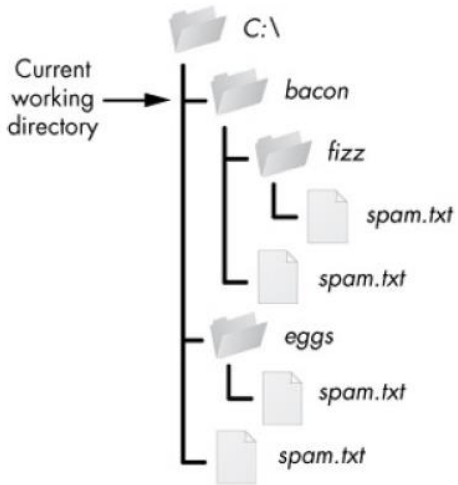
```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

# The Current Working Directory

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32‘
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

# Absolute vs. Relative Paths

- There are two ways to specify a file path.
- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory




| | Relative Paths | Absolute Paths |
|---|---|---|
| C:\ | ..\ | C:\ |
| bacon | .\ | C:\bacon |
| fizz | .\fizz | C:\bacon\fizz |
| spam.txt | .\fizz\spam.txt | C:\bacon\fizz\spam.txt |
| spam.txt | .\spam.txt | C:\bacon\spam.txt |
| eggs | ..\eggs | C:\eggs |
| spam.txt | ..\eggs\spam.txt | C:\eggs\spam.txt |
| spam.txt | ..\spam.txt | C:\spam.txt |

*Figure 8-2. The relative paths for folders and files in the working directory C:\bacon*

# Creating New Folders with os.makedirs()

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

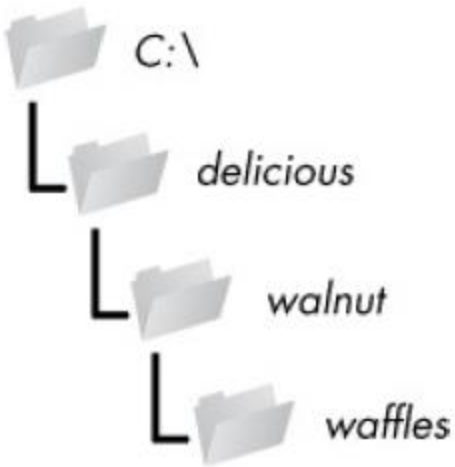



*Figure 8-3. The result of* `os.makedirs('C:\\delicious \\walnut\\waffles')`

# Handling Absolute and Relative Paths

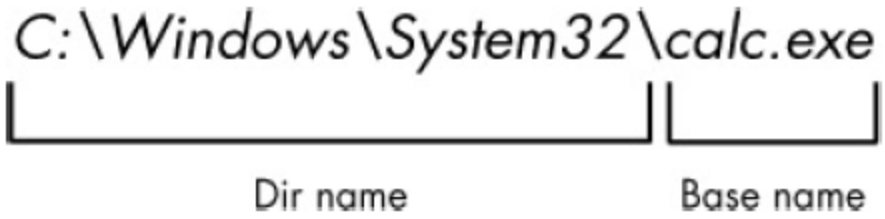
Calling os.path.abspath(*path*) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.

Calling os.path.isabs(*path*) will return True if the argument is an absolute path and False if it is a relative path.

Calling os.path.relpath(*path, start*) will return a string of a relative path from the *start* path to *path*. If *start* is not provided, the current working directory is used as the start path.

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
```

# os.path.basename(*path*) and os.path.dirname(*path*)




```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

# os.path.split()

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

# Finding File Sizes and Folder Contents

- Calling os.path.getsize(*path*) will return the size in bytes of the file in the *path* argument.
- Calling os.listdir(*path*) will return a list of filename strings for each file in the *path* argument. (Note that this function is in the os module, not os.path.)

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

# Checking Path Validity

Calling os.path.exists(*path*) will return True if the file or folder referred to in the argument exists and will return False if it does not exist.

Calling os.path.isfile(*path*) will return True if the path argument exists and is a file and will return False otherwise.

Calling os.path.isdir(*path*) will return True if the path argument exists and is a folder and will return False otherwise.

# Examples

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

# Saving Variables with the shelve Module

- You can save variables in your Python programs to binary shelf files using the shelve module.
- This way, your program can restore data to variables from the hard drive.
- The shelve module will let you add Save and Open features to your program.
- For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

# Creating the shelf file

```
import shelve
shelfFile = shelve.open('mydata')
cats = ['Zophie', 'Pooka', 'Simon']
shelfFile['cats '] = cats
shelfFile.close()
```

# Reading the shelf file

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

# Saving Variables with the pprint.pformat() Function

- Pretty Printing that the pprint.pprint() function will “pretty print” the contents of a list or dictionary to the screen.
- But the pprint.pformat() function will return this same text as a string instead of printing it.
- Not only is this string formatted to be easy to read, but it is also syntactically correct Python code.
- Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use.
- Using pprint.pformat() will give you a string that you can write to *.py* file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

# Example

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

# Importing *myCats.py* in another program

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

# Organizing Files

- You learned how to create and write to new files in Python.
- Your programs can also organize preexisting files on the hard drive.
- consider tasks such as these:

- ❑ Making copies of all PDF files (and *only* the PDF files) in every sub-folder 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)

# 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

- shutil.copy(*source, destination*) will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* are strings.)
- 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.

# Example

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

- 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:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt*.

# shutil.copytree()

- This will copy an entire folder and every folder and file contained in it.
- Calling shutil.copytree(*source, destination*) will copy the folder at the 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.

# Example

```
>>> 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.
- You have now safely backed up your precious, precious bacon.

# Moving and Renaming Files and Folders

- shutil.move(*source, destination*)
- This 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.

# Example

```
>>> 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*.”
- 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().

# Example-2

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

'C:\\eggs\\new_bacon.txt‘

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

- 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*.”

- Previous two 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.
- So the *bacon.txt* text file is renamed to *eggs*

# Exception.

- 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'
```

# Permanently Deleting Files and Folders

- To delete a single file or a single empty folder with functions in the os module,
- 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.

# Example

- for filename in os.listdir():

  if filename.endswith('.**r**xt'):

  os.unlink(filename)

- import os
- for filename in os.listdir():

  if filename.endswith('.rxt'):

  #os.unlink(filename)

  print(filename)

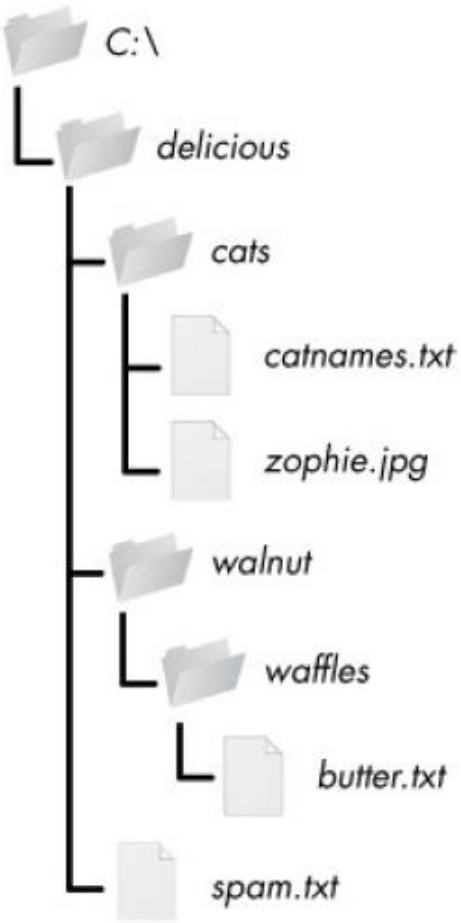
# Safe Deletes with the send2trash Module

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

# Walking a Directory Tree

- 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.
- Python provides a function to handle this process for you.

# Example


C:\
delicious
cats
catnames.txt
zophie.jpg
walnut
waffles
butter.txt
spam.txt

# os.walk() function

- 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

# Example

```
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('')
```

# Output

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

# Compressing Files with the zipfile Module

- Compressing a file reduces its size, which is useful when transferring it over the Internet. 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, say, attached to an email.
- Your Python programs can both create and open (or *extract*) ZIP files using functions in the zipfile module.

# 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 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.

# 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
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

- A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file.

- These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file.

- ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively.

# Extracting from ZIP Files

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

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

- The contents of *example.zip* will be extracted to *C:*. Optionally, you can pass a folder name to extractall() to have it extract the files into a folder other than the current working directory.
- If the folder passed to the extractall() method does not exist, it will be created. 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.

# The extract() method

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

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

- The string you pass to extract() must match one of the strings in the list 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.
- If this second argument is a folder that doesn't yet exist, Python will create the folder. The value that extract() returns is the absolute path to which the file was extracted.

# 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.)

# Example

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

# Debugging

- There are a few tools and techniques to identify what exactly your code is doing and where it's going wrong.
- First, you will look at logging and assertions, two features that can help you detect bugs early.
- Second, you will look at how to use the debugger. The debugger is a feature of IDLE 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.

# Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code.
- We know to handle Python's exceptions with try and except statements so that your program can recover from exceptions that you anticipated.
- But you can also 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 following:

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

```
>>> 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.

# Example

```
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))
```