# READING, WRITING AND ORGANIZING FILES

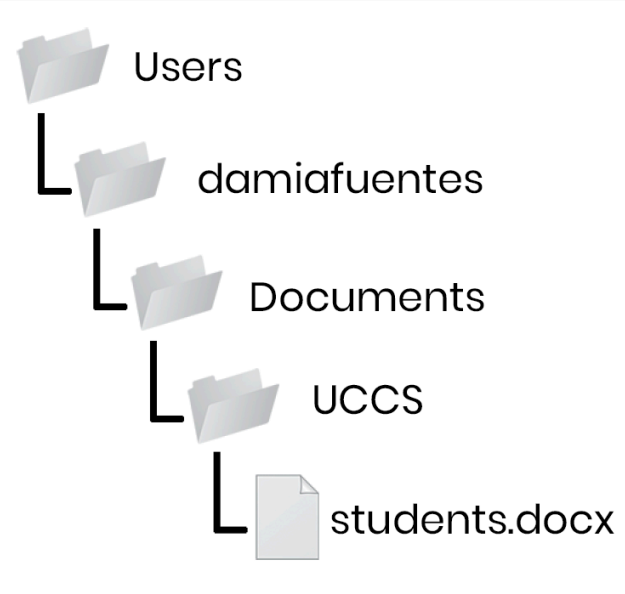
CS 3080: Python Programming

UCCS University of Colorado Colorado Springs

# Reading and writing files

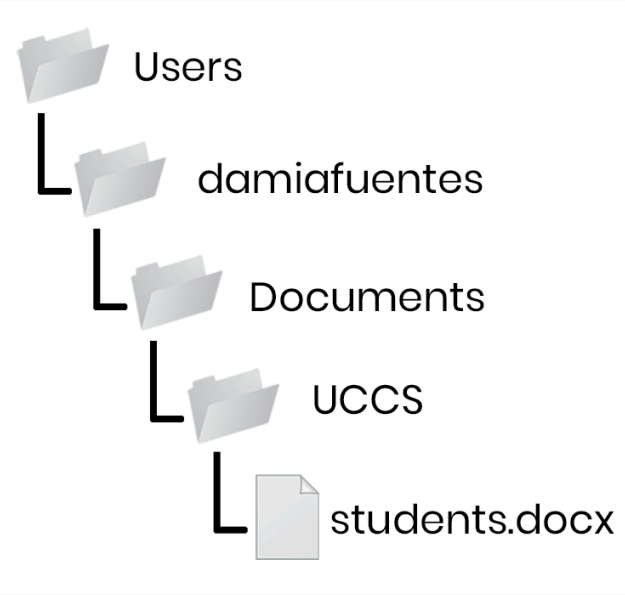
- Variables are a fine way to store data while your program is running.
- But if you want your data to persist even after your program has finished, you need to save it to a file.

# File and file paths




- File has two (very basic) properties:
  - *Filename*
    - The part of the filename after the last period is called the file's extension and tells you a file's type
    - students.docx
  - *Path*
    - Specifies the location of the file on the computer.
    - /Users/johndoe/Documents/UCCS/ (Mac, forward slash)
    - \Users\johndoe\Documents\UCCS\ (Windows, backward slash)
    - Users, johndoe, Documents and UCCS are folders, also called directories

# File and file paths




- File has two (very basic) properties:
  - *Filename*
    - The part of the filename after the last period is called the file's extension and tells you a file's type
    - students.docx
  - *Path*
    - Specifies the location of the file on the computer.
    - **/Users/johndoe/Documents/UCCS/ (OS X and Linux, forward slash)**
    - **\Users\johndoe\Documents\UCCS\ (Windows, backward slash)**
    - Users, johndoe, Documents and UCCS are folders, also called directories

*How to handle Windows and OS X and Linux at the same time?*

# os.path.join() function

```
import os

print(os.path.join('Users','johndoe','UCCS'))
# Mac output: 'Users/johndoe/UCCS'
# Windows output: 'Users\\johndoe\\UCCS'
# Notice the escaped backslashes in Windows.

print(os.path.sep)
# Correct folder-separating slash for the computer running the program
# / for OS X, Linus
# \\ for Windows
```

# The current working directory

- Every program that runs on your computer has a current working directory, or cwd.
- And we can change it!

```
import os

print(os.getcwd())
# /Users/johndoe/Documents/UCCS/CS_3030_Python/Lectures

os.chdir(os.path.join(os.path.sep, 'Users','johndoe', 'Downloads'))
print(os.getcwd())
# /Users/johndoe/Downloads
```

# The current working directory

```
os.chdir(os.path.join(os.path.sep, 'Users','nonExistingUser', 'Downloads’))
# FileNotFoundError: [Errno 2] No such file or directory: '/Users/nonExistingUser/
Downloads'
```

# Absolute vs. Relative Paths

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

- For relative paths we can use the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path.
  - *A **single period** ("dot") for a folder name is shorthand for "this directory."*
  - ***Two periods** ("dot-dot") means "the parent folder."*

# Absolute vs. Relative Paths

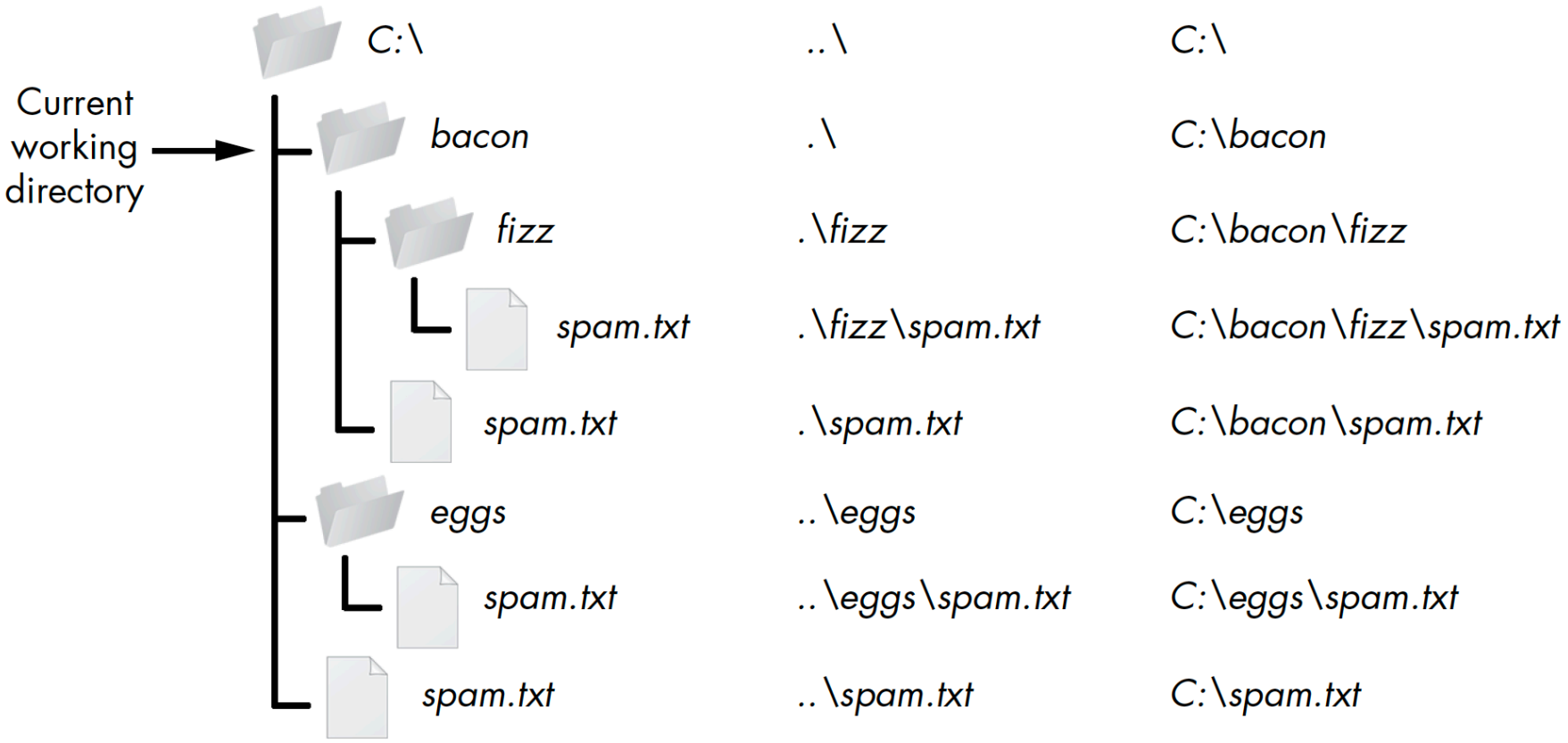



*Figure 8-2: The relative paths for folders and files in the working directory* C:\bacon

# Absolute vs. Relative Paths

- The .\ (or ./) at the start of a relative path is optional.
- For example, .\spam.txt (or ./spam.txt) and spam.txt refer to the same file.

# Creating New Folders

- Your programs can create new folders (directories) with the **os.makedirs()** function
- os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists

```
import os


os.makedirs(os.path.join('.', 'lectures', 'lecture10', 'testfolder', 'subfolder'))
```

# 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.
- Calling **os.path.dirname(path)** will return a string of everything that comes before the last slash in the path argument.
- Calling **os.path.basename(path)** will return a string of everything that comes after the last slash in the path argument.
- Calling **os.path.split()** will return a tuple with the dir name and the base name.

# Finding File Sizes and Folder Contents

- Calling **os.path.getsize(path)** will return the size in bytes of the file in the path argument.
  - *Note this will return the size of a file, not a directory*
- 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.*

# 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.

# The File Reading/Writing Process

- There are three steps to reading or writing files in Python.
    1. Call the **open()** function to return a File object.
    2. Call the **read()** or **write()** method on the File object.
    3. Close the file by calling the **close()** method on the File object.

# open(path)

```
path = os.path.join('.', 'hello.txt')

if os.path.exists(path) and os.path.isfile(path):
    helloFile = open(path)
    print(helloFile)

# <_io.TextIOWrapper name='lectures/lecture11/hello.txt' mode='r'
encoding='UTF-8'>
```

# open(path)

- The path can be either an absolute or relative path.

# open(path)

- The path can be either an absolute or relative path.
- The open() function returns a File object.
  - *A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries.*

# open(path)

- The path can be either an absolute or relative path.
- The open() function returns a File object.
  - *A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries.*
- open(path) will open the file in read-only mode.
  - *When a file is opened in read-only mode, Python lets you only read data from the file; you can't write or modify it in any way.*

# open(path)

- The path can be either an absolute or relative path.
- The open() function returns a File object.
  - *A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries.*
- open(path) will open the file in read-only mode.
  - *When a file is opened in read-only mode, Python lets you only read data from the file; you can't write or modify it in any way.*
- You can explicitly specify the mode by passing the string value 'r' as a second argument to open().
  - *So **open(somePath, 'r')** and open(somePath) do the same thing.*

# Reading the Contents of Files

hello.txt

Hello world! This is a text file.
And this is a second line.
Third line.

```
helloFile = open('hello.txt')
```

- If you want to read the entire contents of a file as a string value:
  - *helloFile.read()*
- If you want a list of string values from the file, one string for each line of text:
  - *helloFile.readlines()*
- If you want to iterate through the lines, one each time:
  - *helloFile.readline()*

# Writing to files

- **open(path, ‘w’)** for write mode.
- **open(path, ‘a’)** for append mode, which will append text to the end of the existing file.
- If the filename passed to open() does not exist, both write and append mode will create a new, blank file.
- openedFile.write(‘Some text’) will write to the file.
- After reading or writing a file, call the **close()** method before opening the file again.

# 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 and save to variables from the disk.
  - *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.*

# Saving Variables with the shelve Module

```
import shelve

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

They are saved like a key-value pair

On Windows you will see three new files in the current working directory: mycats.bak, mycats.dat, and mycats.dir.
On OS X, only a single mycats.db file will be created.

# Saving Variables with the shelve Module

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

# Saving Variables with the shelve Module

```
shelfFile = shelve.open('mydata’)
list(shelfFile.keys())
# ['cats’]
list(shelfFile.values())
# [['Zophie', 'Pooka', 'Simon’]]
shelfFile.close()
```

# Shelve vs plaintext

- Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit,
- But if you want to save data from your Python programs, use the shelve module.

# Saving Variables with the pprint.pformat() Function

```
import pprint

filepath = os.path.join('.', 'lectures', 'lecture11', 'myCats.py')

cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka',
'desc': 'fluffy'}]

print(pprint.pformat(cats))

# "[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy',
'name': 'Pooka'}]"

fileObj = open(filepath, 'w')

fileObj.write('cats = ' + pprint.pformat(cats) + '\n')

fileObj.close()
```

# Saving Variables with the pprint.pformat() Function

```
import lectures.lecture11.myCats as myCats
print(myCats.cats)
# [{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy',
'name': 'Pooka'}]
print(myCats.cats[0])
# {'desc': 'chubby', 'name': 'Zophie'}
print(myCats.cats[0]['name'])
# 'Zophie'
```

# pprint.format() vs shelve module

- The benefit of creating a .py file (as opposed to saving variables with the shelve module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor.
- Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text.
- File objects, for example, cannot be encoded as text. In that case, we will use the shelve module.

# Organizing files

- 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.

# 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)
  - *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.*

- shutil.copytree(source, destination)
  - *copy the folder at the path source, along with all of its files and subfolders, to the new folder at the path destination.*
  - *If destination exists, we'll get FileExistsError*

# Moving and Renaming Files and Folders

- shutil.move(source, destination)
  - *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 there is a file with the same name in destination, it would be overwritten.* ***Since it's easy to accidentally overwrite files in this way, you should take some care when using move().***
  - *The destination path can also specify a filename.*
  - *But if there is no destination folder, then move() will rename the source file to the destination name.*

# Permanently Deleting Files and Folders

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

BE CAREFUL WHEN USING THESE FUNCTIONS!!
They permanently delete the files and folders.

# Permanently Deleting Files and Folders

```
import os


for filename in os.listdir():
    if filename.endswith('.txt'):
        os.unlink(filename)
```

# Permanently Deleting Files and Folders

```
import os


for filename in os.listdir():
    if filename.endswith('.txt'):
        print(filename)
        #os.unlink(filename)
```

Do this before unlink, to see which files are going to be deleted.

# Safe Deletes with the send2trash Module

- Using send2trash is 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.
- You may have to install the module
  - *pip install send2trash*
- It cannot pull files out of the trash.

# Safe Deletes with the send2trash Module

```
import send2trash

baconFile = open('bacon.txt', 'a’) # creates the file
baconFile.write('Bacon is not a vegetable.’)
baconFile.close()

send2trash.send2trash('bacon.txt')
```

# Walking a Directory Tree with os.walk(path)

```
import os

for folderName, subfolders, filenames in os.walk('.'):
    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('')
```

# 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.
- Your Python programs can both create and open (or extract) ZIP files using functions in the zipfile module.

# Compressing Files with the zipfile Module - Reading

```
import zipfile, os

path = os.path.join('.')
os.chdir(path) # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip')
print(exampleZip.namelist())       # ['example/', 'example/folder2/',
                                   # 'example/folder1/', 'example/image1.png',
                                   # 'example/image2.png', 'example/image3.png']
spamInfo = exampleZip.getinfo('example/image1.png')
print(spamInfo.file_size)          # 215872 bytes
print(spamInfo.compress_size)      # 190101 bytes
exampleZip.close()
```

# Compressing Files with the zipfile Module - Extracting

```
import zipfile, os


path = os.path.join('.')
os.chdir(path) # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip')
exampleZip.extractall()
exampleZip.close()
```

# Compressing Files with the zipfile Module - Extracting

```
import zipfile, os


path = os.path.join('.')
os.chdir(path) # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip')
# Extract specific file
exampleZip.extract('example/image1.png')
# Extract to specific location
newPath = os.path.join('..','lecture11')
exampleZip.extract('example/image1.png', newPath)
exampleZip.close()
```

# Compressing Files with the zipfile Module - Compressing

```
import zipfile


newZip = zipfile.ZipFile('ziptest.zip', 'w’)
newZip.write('test.py', compress_type=zipfile.ZIP_DEFLATED)
newZip.close()
```

# Compressing Files with the zipfile Module - Compressing

```
import zipfile


newZip = zipfile.ZipFile('ziptest.zip', 'w’)
newZip.write('test.py', compress_type=zipfile.ZIP_DEFLATED)
newZip.close()
```

Notice 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 .