Module 3:

MANIPULATING STRINGS

Reading and Writing Files

Working with Strings

- Text is one of the most common forms of data your programs will handle.
- How to concatenate two string values together with + operator?
- What else you can do with strings??
- You can
  - ✓ extract partial strings from string values,
  - ✓ add or remove spacing,
  - ✓ convert letters to lowercase or uppercase
  - ✓ check that strings are formatted correctly
  - ✓ access the clipboard for copying and pasting text

String Literals

- String values in Python begin and end with a single quote.
- How can you use a quote inside a string?
- **Ex: 'That is Alice's cat.'**
- won’t work, because Python thinks the string ends after Alice, and rest (s cat.') is invalid Python code.
- There are multiple ways to type strings.

Double Quotes

- Strings can begin and end with double quotes
- Benefit of using double quotes : string can have a single quote character in it.
- >>> spam = "That is Alice's cat.“
- Since the string begins with a double quote, Python knows that single quote is part of the string and not marking end of string.
- What to do, if need to use both single quotes and double quotes in the string???
- **use escape characters.**

Escape Characters

- An escape character lets to use characters that are otherwise impossible to put into a string.
- An escape character consists of a backslash (\) followed by the character you want to add to the string.
- Ex: the escape character for a single quote is \'.
- Use this inside a string that begins and ends with single quotes.

Escape Characters

- >>> spam = 'Say hi to Bob\'s mother.‘
- Python knows that since the single quote in Bob\'s has a backslash, it is not a single quote meant to end string value.
- The escape characters \‘ and \" let to put single quotes and double quotes inside strings, respectively.
- Ex: >>> **print("Hello there!\nHow are you?\nI\'m doing fine.")**

Hello there!
How are you?

Escape Characters

Escape character	Prints as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)
`\\`	Backslash

Raw Strings

- You can place an **r** before the beginning quotation mark of a string to make it a raw string.
- A raw string completely ignores all escape characters and prints any backslash that appears in the string.
- Ex: >>> print(r'That is Carol\'s cat.')

That is Carol\'s cat.

- Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character.

Raw Strings

- Raw strings are helpful while typing string values that contain many backslashes, such as the strings used for Windows file paths like r'C:\Users\Al\Desktop' or regular expressions

Multiline Strings with Triple Quotes

- Use multiline strings instead of using the \n escape character to put a newline into a string.
- A multiline string in Python begins and ends with either three single quotes or three double quotes.
- Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string.
- Python’s indentation rules for blocks do not apply to lines inside a multiline string.

Multiline Strings with Triple Quotes

- Consider a program : catnapping.py

- print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,

Bob''')

Note that single quote character in Eve's does not need to be escaped.

Output:

Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,

Bob

Multiline Strings with Triple Quotes

- Note: single quote character in Eve's does not need to be escaped.
- Escaping single and double quotes is optional in multiline strings.
- print('''Dear Alice,
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
Sincerely,
Bob''')
- The following print() call would print identical text but doesn't use a multiline string:

print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat burglary, and extortion.\n\nSincerely,\nBob')

Multiline Comments

- While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.
- Ex:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com
This program was designed for Python 3, not Python 2.
"""

def spam():
        """This is a multiline comment to help
        explain what the spam() function does."""
print('Hello!')
```

Indexing and Slicing Strings

- Strings use indexes and slices (the same way lists do)
- think of the string 'Hello, world!' as a list and each character in the string as an item with a corresponding index.
- The space and exclamation point are included in the character count, so 'Hello, world!' is 13 characters long, from H at index 0 to ! at index 12.

'	H	e	l	l	o	,		w	o	r
l		d	!	'						
0	1	2	3	4	5	6	7	8	9	
10	11	12	13							

Indexing and Slicing Strings

- If you specify an index, you'll get the character at that position in the string.
- If you specify a range from one index to another, the starting index is included and the ending index is not.
- That's why, if spam is 'Hello, world!', spam[0:5] is 'Hello'.
- The substring got from spam[0:5] will include everything from spam[0] to spam[4], leaving out comma at index 5 &space at

```
>>> spam = 'Hello, world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[7:]
'world!'
```

Indexing and Slicing Strings

- Note that slicing a string does not modify the original string
- You can capture a slice from one variable in a separate variable.
- By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

```
>>> spam = 'Hello, world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

Putting Strings Inside Other Strings

- Strings can be put inside other strings using the + operator and string concatenation
- This requires a lot of tedious typing. A simpler approach is to use string interpolation.
- In this the %s operator inside the string acts as a marker to be replaced by values following the string.
- Benefit: Need not call str() to convert values into string.

```
>>> name = 'Al'
>>> age = 4000
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
```

Putting Strings Inside Other Strings

- Another method is to use f-strings, which is similar to string interpolation except that braces are used instead of %s, with the expressions placed directly inside the braces.
- Like raw strings, f-strings have an f prefix before the starting quotation mark.
- Available Python 3.6 and onwards
- Include the f prefix; otherwise, braces and their contents will be a part of string value:

```
>>> name = 'Al'
>>> age = 4000
>>> f 'My name is {name}. Next year I will be {age + 1}.'
'My name is Al. Next year I will be 4001.'
```

Useful String Methods

- Several string methods analyze strings or create transformed string values.

- **The upper(), lower(), isupper(), and islower() Methods:**
  - **The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.**

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
```

  - Nonletter characters in the string remain unchanged.

Useful String Methods: The upper(), lower()

- Note that these methods do not change the string itself but return new string values.
- **If you want to change the original string, you have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored.**

This is why use spam = spam.upper()
to change the string in spam instead of simply spam.upper().
This is just like if a variable eggs contains value 10. Writing eggs + 3 does not change value of eggs, but eggs = eggs + 3 does.)

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
```

Useful String Methods: The upper(), lower()

- The upper() and lower() methods are helpful if you need to make a case insensitive comparison.
- **Ex: the strings 'great' and 'GREat' are not equal to each other.**
- **But in the following program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to**

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
print('I feel great too.')
else:
print('I hope the rest of your day is good.')
```

Useful String Methods: The upper(), lower()

- When you run this program, the question is displayed, and entering a variation on great, such as GREat, will still give the output I feel great too.

- **Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less**

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
print('I feel great too.')
else:
print('I hope the rest of your day is good.')
```

Output:
How are you?
GREat
I feel great too.

Useful String Methods: The upper(), lower()

- The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns False.

```
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Useful String Methods: The upper(), lower()

- The upper() and lower() string methods themselves return strings.
- you can call string methods on those returned string values as well. Expressions that do this will look like a chain of method calls.

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

Useful String Methods: The isX() Methods

- Along with islower() and isupper(), there are several other string methods that have names beginning with the word **is**.
- **These methods return a Boolean value that describes the nature of the string.**

Useful String Methods: The isX() Methods

Common isX string methods:

- isalpha() Returns True if string consists only of letters and isn't blank
- isalnum() Returns True if string consists only of letters and numbers and is not blank
- isdecimal() Returns True if the string consists only of numeric characters and is not blank
- isspace() Returns True if the string consists only of spaces, tabs, and newlines and is not blank
- istitle() Returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Useful String Methods: The isX() Methods

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

Useful String Methods: The isX() Methods

- The isX() string methods are helpful to validate user input.
- Ex: This program repeatedly asks users for their age and a password until they provide valid input.
- validateInput.py:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')
while True:
        print('Select a new password (letters and numbers only)
        password = input()
        if password.isalnum():
                break
        print('Passwords can only have letters and numbers
```

Useful String Methods: The isX() Methods

- First while loop, asks the user for their age and store their input in age. If age is a valid (decimal) value, first while loop break out of this and move on to the second, which asks for a password.
- Otherwise, it informs the user that they need to enter a number and again ask them to enter their age.

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')
```

Useful String Methods: The isX() Methods

- Second while loop, asks for a password, store the user's input in password, and break out of the loop if the input was alpha numeric.
- If it wasn't, it tells the user the password needs to be alphanumeric and again ask them to enter a password.

```
while True:
        print('Select a new password (letters and numbers
only):')
        password = input()
        if password.isalnum():
                break
        print('Passwords can only have letters and numbers.')
```

Useful String Methods: The isX() Methods

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')
while True:
        print('Select a new password (letters and numbers only):')
        password = input()
        if password.isalnum():
                break
        print('Passwords can only have letters and numbers.')
```

O/P:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

Useful String Methods: The isX() Methods

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')
while True:
        print('Select a new password (letters and numbers only):')
        password = input()
        if password.isalnum():
                break
        print('Passwords can only have letters and numbers.')
```

O/P:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

- Calling isdecimal() and isalnum() on variables, help to test whether the values stored in those variables are decimal or not, alphanumeric or not.
- These tests help to reject the input forty two but accept 42, and reject secr3t! but accept secr3t

Useful String Methods: startswith() & endswith() Methods

- The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method;
- otherwise, they return False
- These methods are useful alternatives to the == equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello, world!')
True
>>> 'Hello, world!'.endswith('Hello, world!')
True
```

Useful String Methods: The join() and split() Methods

- The join() method is useful when you have a list of strings that need to be joined together into a single string value.
- The join() method is called on a string, gets passed a list of strings, and returns a string.
- The returned string is the concatenation of each string in the passed-in list.
- Ex:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Note that string join() calls on is inserted between each string of list argument.

Ex: when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.

Useful String Methods: The join() and split() Methods

- join() is called on a string value and is passed a list value.
- The split() method does the opposite: It's called on a string value and returns a list of strings.

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

- By default, the string 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found.
- These whitespace characters are not included in strings in the returned list.
- Pass a delimiter string to the split() method to specify a different string to split upon.

Ex:

```
>>>'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

Useful String Methods: The join() and split() Methods

- A common use of split() is to split a multiline string along the newline characters.

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment."
Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the fridge', 'that is labeled "Milk Experiment."', '', 'Please do not drink it.', 'Sincerely,', 'Bob']
```

Passing split() with the argument '\n' helps to split the multiline string stored in spam along the newlines and return a list in which each item corresponds to one line of the string.

Splitting Strings with the partition() Method

- The partition() string method can split a string into the text before and after a separator string.

- This method searches the string it is called on for the separator string it is passed, and returns a tuple of three substrings for the “before,” “separator,” and “after” substrings.

```
>>> 'Hello, world!'.partition('w')
('Hello, ', 'w', 'orld!')
>>> 'Hello, world!'.partition('world')
('Hello, ', 'world', '!')
```

If the separator string you pass to partition() occurs multiple times in the string that partition() calls on, the method splits the string only on the first occurrence:

```
>>> 'Hello, world!'.partition('o')
('Hell', 'o', ', world!')
```

Splitting Strings with the partition() Method

- If the separator string can't be found, the first string returned in the tuple will be the entire string, and the other two strings will be empty:

```
>>> 'Hello, world!'.partition('XYZ')
('Hello, world!', '', '')
```

- You can use the multiple assignment trick to assign the three returned strings to three variables:

```
>>> before, sep, after = 'Hello, world!'.partition(' ')
>>> before
'Hello,'
>>> after
'world!'
```

The partition() method is useful for splitting a string whenever you need the parts before, including, and after a particular separator string.

Justifying Text with the rjust(), ljust(), and center() Methods

- rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- First argument to both methods is an integer length for the justified string.

```
>>> 'Hello'.rjust(10)
‘     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello, World'.rjust(20)
'         Hello, World'
>>> 'Hello'.ljust(10)
'Hello      ‘
```

'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

Justifying Text with the rjust(), ljust(), and center() Methods

- An optional second argument to rjust() and ljust() will specify a fill character other than a space character.

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

These methods are especially useful when it is needed to print tabular data that has correct spacing.

Justifying Text with the rjust(), ljust(), and center() Methods

- picnicTable.py:

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Justifying Text with the rjust(), ljust(), and center() Methods

- In this program, a printPicnic() method is defined that will take in a dictionary of information and use center(), ljust(), and rjust() to display that information in a neatly aligned tablelike format.
- The dictionary that was passed to printPicnic() is picnicItems.
- picnicItems, has 4 sandwiches, 12 apples, 4 cups, and 8,000 cookies.
- This information needs to be organized into two columns, with the name of the item on the left and the quantity on the right.

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Justifying Text with the rjust(), ljust(), and center() Methods

- The printPicnic() function takes in a dictionary, a leftWidth for the left column of a table, and a rightWidth for the right column.
- It prints a title, PICNIC ITEMS, centered above the table.
- Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Justifying Text with the rjust(), ljust(), and center() Methods

- After defining printPicnic(), the dictionary picnicItems is defined and printPicnic() is called twice, passing it different widths for the left and right table columns
- When you run this program, the picnic items are displayed twice.
- The first time the left column is 12 characters wide, and the right column is 5 characters wide.
- The second time they are 20 and 6 characters wide.

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Justifying Text with the rjust(), ljust(), and center() Methods

- When this program is run, the picnic items are displayed twice.
- The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide respectively.

```
---PICNIC ITEMS--
sandwiches.. 4
apples..... 12
cups........ 4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.......... 4
apples.............. 12
cups................ 4
cookies............. 8000
```

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Justifying Text with the rjust(), ljust(), and center() Methods

- Using rjust(), ljust(), and center() ensure that strings are neatly aligned, even if it is not sure how many characters long strings are.

```
---PICNIC ITEMS--
sandwiches.. 4
apples..... 12
cups........ 4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.......... 4
apples.............. 12
cups................ 4
cookies............. 8000
```

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

Removing Whitespace with the strip(), rstrip(), and lstrip() Methods

- Above methods are used to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string.
- The strip() string method will return a new string without any whitespace characters at the beginning or end.
- The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends
- The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends

```
>>> spam = ' Hello, World '
>>> spam.strip()
'Hello, World'
>>> spam.lstrip()
'Hello, World '
>>> spam.rstrip()
' Hello, World'
```

Removing Whitespace with the strip(), rstrip(), and lstrip() Methods

- a string argument will specify which characters on the ends should be stripped.

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

- Passing strip() the argument 'ampS' will tell it to strip occurrences of a, m, p, and capital S from the ends of the string stored in spam.
- The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

Numeric Values of Characters with ord() and chr() Functions

- Computers store information as bytes—strings of binary numbers, which means we need to be able to convert text to numbers.
- Because of this, every text character has a corresponding numeric value called a Unicode code point.
- Ex: numeric code point is 65 for 'A', 52 for '4', and 33 for '!'.

Use the ord() function to get the code point of a one character string, and the chr() function to get the one-character string of an integer code point.

```
>>> ord('A')
65
>>> ord('4')
52
>>> ord('!')
33
>>> chr(65)
'A'
```

Numeric Values of Characters with ord() and chr() Functions

These functions are useful when you need to do an ordering or mathematical operation on characters:

```
>>> ord('B')
66
>>> ord('A') < ord('B')
True
>>> chr(ord('A'))
'A'
>>> chr(ord('A') + 1)
'B'
```

Copying and Pasting Strings with the pyperclip Module

- The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard.
- Sending the output of a program to the clipboard will make it easy to paste it into an email, word processor, or some other software.
- The pyperclip module does not come with Python. To install it, from third-party modules.

```
>>> import pyperclip
>>> pyperclip.copy('Hello, world!')
>>> pyperclip.paste()
'Hello, world!'
```

- If something outside of your program changes the clipboard contents, the paste() function will return it.

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called paste(), it would look like this:'
```

READING AND WRITING FILES

- Variables are a fine way to store data while program is running, but if once the program has finished running, if data is needed it needs to be saved to a file.

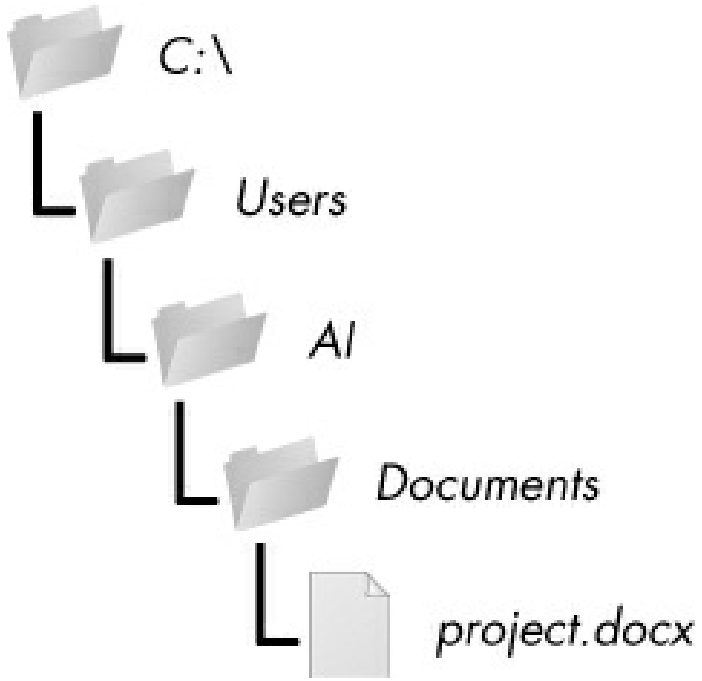
- How to use Python to create, read, and save files on the hard drive.

Files and File Paths

- A file has two key properties: a filename and a path.
- The path specifies the location of a file on the computer.
- Ex: there is a file on my Windows laptop with the filename project.docx in the path C:\Users\Al\Documents.
- The part of the filename after the last period is called the file's extension and tells file's type.
- The filename project.docx is a Word document, and Users, Al, and Documents all refer to folders (=directories).
- Folders can contain files and other folders

Files and File Paths

- Ex: project.docx is in the Documents folder, which is inside the Al folder, which is inside the Users folder



A file in a hierarchy of folders

- The C:\ part of the path is the root folder, which contains all other folders. On Windows, the root folder is named C:\ and is also called C: drive.
- Additional volumes, such as a DVD drive or USB flash drive, will appear differently on different operating systems.
- On Windows, they appear as new, lettered root drives, such as D:\ or E:\.
- Note that folder names and filenames are not case-sensitive on Windows

Backslash on Windows & Forward Slash on macOS and Linux

- On Windows, paths are written using backslashes (\) as the separator between folder names.
- The macOS and Linux operating systems, use the forward slash (/) as their path separator
- If you want programs to work on all operating systems, you will have to write your Python scripts to handle both cases.
- To do this simply use Path() function in the pathlib module.

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

If you pass it the string values of individual file and folder names in your path, Path() will return a string with a file path using the correct path separators.

Backslash on Windows & Forward Slash on macOS and Linux

- Note that the convention for importing pathlib is to run from pathlib import Path, since otherwise we'd have to enter pathlib.Path everywhere Path shows up in code
- Path('spam', 'bacon', 'eggs') returned a WindowsPath object for the joined path, represented as WindowsPath('spam/bacon/eggs').
- Even though Windows uses backslashes, the WindowsPath representation in interactive shell displays them using forward slashes, since open source software developers have historically favored Linux operating system.

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

- pass it to the str() function, which in returns 'spam\\bacon\\eggs'.
- Note that the backslashes are doubled because each backslash needs to be escaped by another backslash character.)

Backslash on Windows & Forward Slash on macOS and Linux

- This Path object, WindowsPath can be passed to the file-related functions.

- Ex: following code joins names from a list of filenames to the end of a folder's name:

```
>>> from pathlib import Path
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
print(Path(r'C:\Users\Al', filename))
C:\Users\Al\accounts.txt
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

Backslash on Windows & Forward Slash on macOS and Linux

- On Windows, backslash separates directories, so can’t be used in filenames.
- Backslashes can be used in filenames on macOS and Linux.
- Path(r'spam\eggs') refers to two separate folders (or a file eggs in a folder spam) on Windows, the same command would refer to a single folder (or file) named spam\eggs on macOS and Linux.
- For this reason, it’s usually a good idea to always use forward slashes in Python code.
- The pathlib module will ensure that it always works on all operating systems.

Using the / Operator to Join Paths

- The / operator used for division can also combine Path objects and strings.
- This is helpful for modifying a Path object after creating it with the Path() function.
- Ex:

Using the / operator with Path objects makes joining paths just as easy as string concatenation.
It's also safer than using string concatenation or join() method.

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Using the / Operator to Join Paths

- A script that uses this code isn't safe, because its backslashes would only work on Windows.
- Use if statement that checks sys.platform (which contains a string describing the computer's operating system) to decide what kind of slash to use.

But applying this custom code everywhere it's needed can be inconsistent and bug-prone.

```
>>> homeFolder = r'C:\Users\Al'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\\Users\\Al\\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\\Users\\Al\\spam'
```

Using the / Operator to Join Paths

- The pathlib module solves these problems by reusing the / math division operator to join paths correctly, no matter what operating system code is running on.
- The following example uses this strategy to join the same paths as in the previous example:

when using the / operator for joining paths is that one of the first two values must be a Path object.

Else Python will give an error

```
>>> homeFolder = Path('C:/Users/Al')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/Al/spam')
>>> str(homeFolder / subFolder)
'C:\\Users\\Al\\spam'
```

Using the / Operator to Join Paths

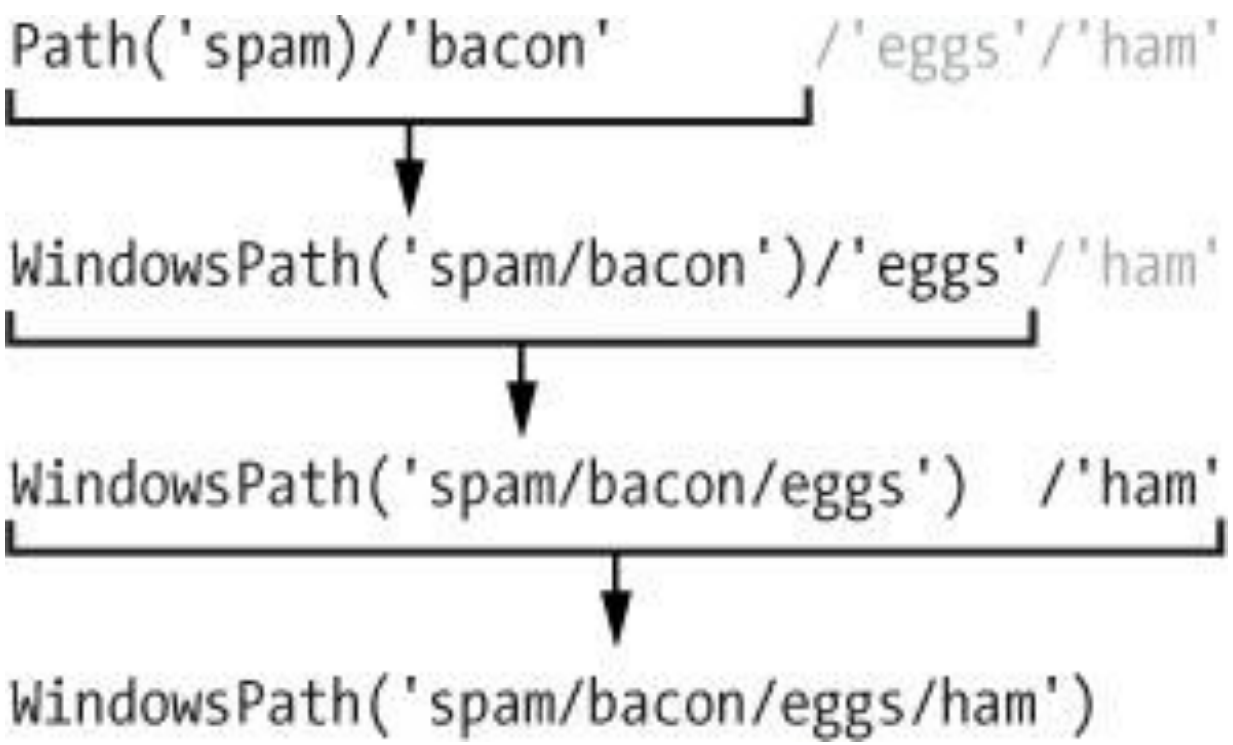
- when using the / operator for joining paths is that one of the first two values must be a Path object.
- Else Python will give an error
- Ex:

```
>>> 'spam' / 'bacon' / 'eggs'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s)
for /: 'str' and 'str'
```

Python evaluates the / operator from left to right and evaluates to a Path object, so either the first or second leftmost value must be a Path object for entire expression to evaluate to a Path object.

Using the / Operator to Join Paths

- Here's how the / operator and a Path object evaluate to the final Path object. Ex:



If you see the TypeError: unsupported operand type(s) for /: 'str' and 'str' error message shown previously, you need to put a Path object on the left side of the expression.

The / operator replaces the older os.path.join() function, which you can learn more about from https://docs.python.org/3/library/os.path.html#os.path.join.

The Current Working Directory

- Every program running on computer has a current working directory, or cwd.
- Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.
- get the current working directory as a string value with the Path.cwd() function and change it using os.chdir().

Here, the current working directory is set to C:\Users\Al\AppData\Local\Programs\Python\Python37,
so the filename project.docx refers to C:\Users\Al\AppData\Local\Programs\Python\Python37\project.docx.
When cwd is changed to C:\Windows\System32, filename project.docx is interpreted as C:\Windows\System32\project.docx.

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')'
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

The Current Working Directory

- Python will display an error if you try to change to a directory that does not exist.
- There is no pathlib function for changing the working directory, because changing the current working directory while a program is running can often lead to subtle bugs.

The os.getcwd() function is the older way of getting the current working directory as a string.

```
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

The Home Directory

- All users have a folder for their own files on the computer called home directory or home folder.
- Get a Path object of the home folder by calling Path.home():

```
>>> Path.home()
WindowsPath('C:/Users/Al')
```

- The home directories are located in a set place depending on your operating system:
  - ✓On Windows, home directories are under C:\Users.
  - ✓On Mac, home directories are under /Users.
  - ✓On Linux, home directories are often under /home.

Your scripts will almost certainly have permissions to read and write the files under home directory, so it's an ideal place to put the files that your Python programs will work with.

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 program's cwd.
- There are also the dot (.) and dot-dot (..) folders.
- These are not real folders but special names 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."

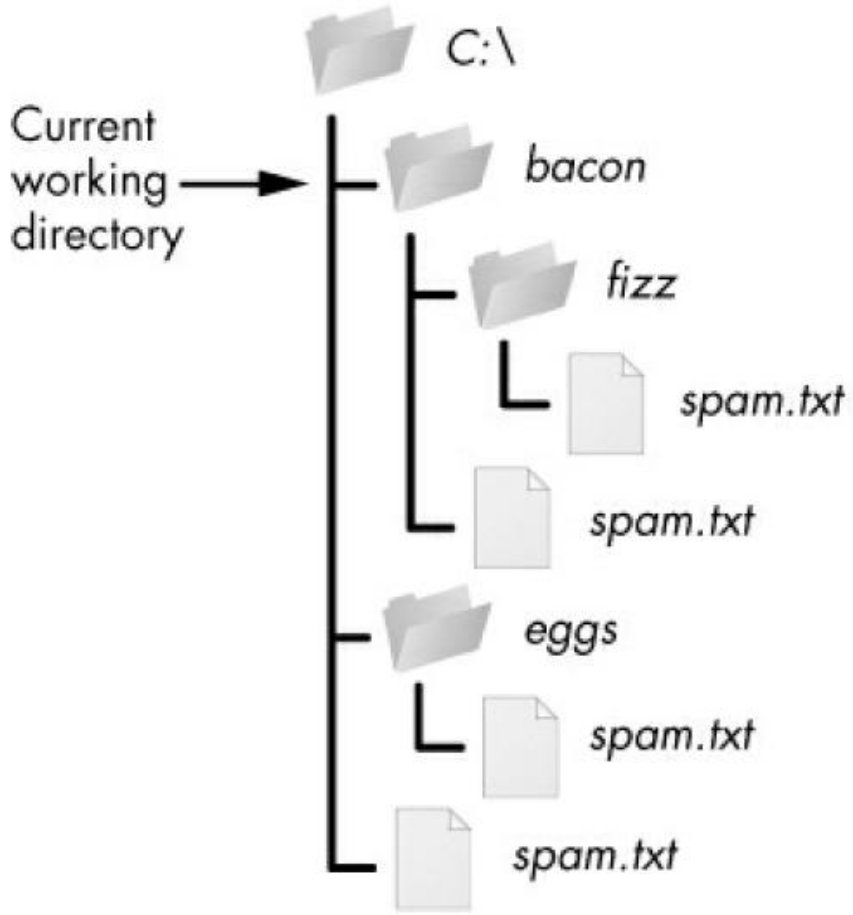
Absolute vs. Relative Paths: Ex

- When cwd is set to C:\bacon, relative paths for other folders & files are set as :

- The .\ at start of a relative path is optional.

- Ex:

.\spam.txt and spam.txt refer to same file.

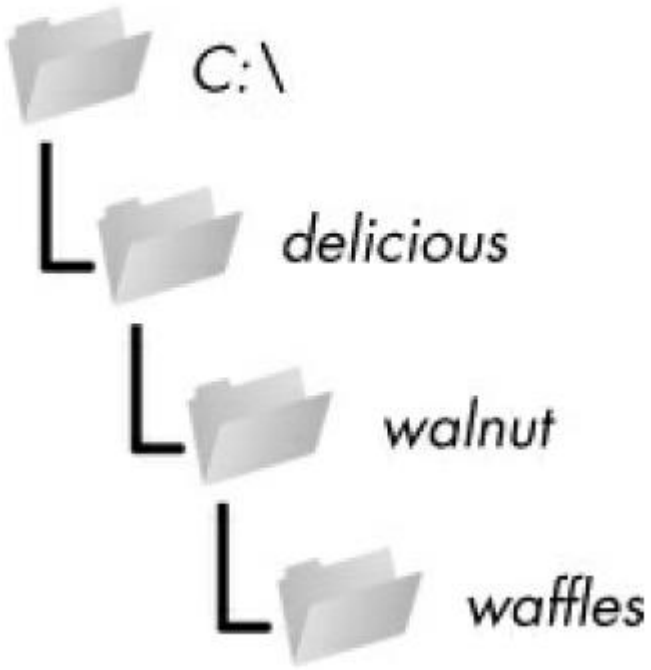


	Relative paths	Absolute paths
C:\	..\	C:\
bacon	.\	C:\bacon
fizz	.\fizz	C:\bacon\fizz
spam.txt	.\fizz\spam.txt	C:\bacon\fizz\spam.txt
spam.txt	.\spam.txt	C:\bacon\spam.txt
eggs	..\eggs	C:\eggs
spam.txt	..\eggs\spam.txt	C:\eggs\spam.txt
spam.txt	..\spam.txt	C:\spam.txt

Creating New Folders Using os.makedirs() Function

- programs can create new folders (directories) with os.makedirs() function

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



- This will create not just the C:\delicious folder but also a walnut folder inside C:\delicious and a waffles folder inside C:\delicious\walnut.
- os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists.

Creating New Folders Using os.makedirs() Function

- To make a directory from a Path object, call the mkdir() method.
- Ex: this code will create a spam folder under home folder on computer:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\Al\spam').mkdir()
```

- Note that mkdir() can only make one directory at a time;
- it won't make several subdirectories at once like os.makedirs().

Handling Absolute and Relative Paths

- The pathlib module provides methods for checking whether a given path is an absolute path and returning the absolute path of a relative path.
- Calling the is_absolute() method on a Path object will return True if it represents an absolute path or False if it represents a relative path.
- Ex:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().is_absolute()
True
>>> Path('spam/bacon/eggs').is_absolute()
False
```

Handling Absolute and Relative Paths

- To get an absolute path from a relative path, you can put Path.cwd() / in front of the relative Path object.
- “relative path,” mean a path that is relative to the current working Directory
- Ex:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37/my/relative/ path')
```

Handling Absolute and Relative Paths

- If relative path is relative to another path besides the current working directory, just replace Path.cwd() with that other path instead
- Ex: It gets an absolute path using the home directory instead of the current working directory

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home() / Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

Handling Absolute and Relative Paths

- The os.path module also has some useful functions related to 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.

Handling Absolute and Relative Paths

➢Ex:

```
>>> os.path.abspath('.')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.path.abspath('.\\Scripts')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Handling Absolute and Relative Paths

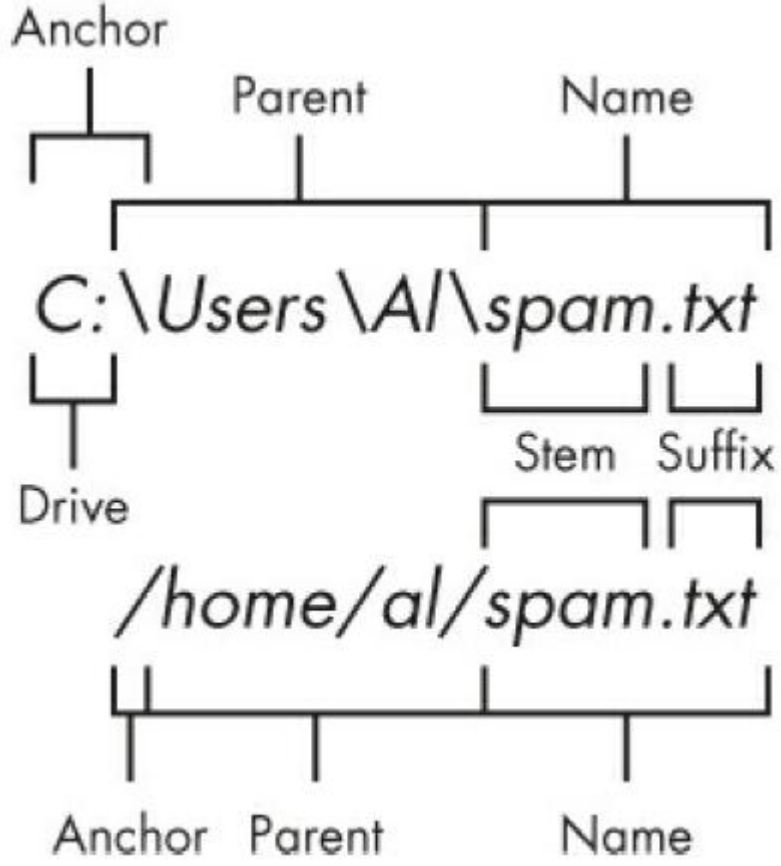
- Since C:\Users\Al\AppData\Local\Programs\Python\Python37 was the working directory when os.path.abspath() was called, the “single-dot” folder represents the absolute path
- 'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'.

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows‘
```

- When the relative path is within the same parent folder as the path, but is within subfolders of a different path, such as 'C:\\Windows' and 'C:\\spam\\eggs', use the “dot-dot” notation to return to the parent folder.

Getting the Parts of a File Path

- Given a Path object, extract the file path's different parts as strings using several Path object attributes.
- Given a Path object, you can extract the file path's different parts as strings using several Path object attributes.
- These can be useful for constructing new file paths based on existing ones.
- The parts of a Windows (top) and macOS/Linux (bottom) file path



Getting the Parts of a File Path

- The parts of a file path include the following:
  - ✓ The anchor, which is the root folder of the file system
  - ✓ On Windows, the drive, which is the single letter that often denotes
  - ✓ a physical hard drive or other storage device
- Note that Windows Path objects have a drive attribute, but macOS and Linux Path objects don't.
- The drive attribute doesn't include the first backslash.
- To extract each attribute from file path:

```
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\\'
>>> p.parent # This is a Path object, not a string.
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

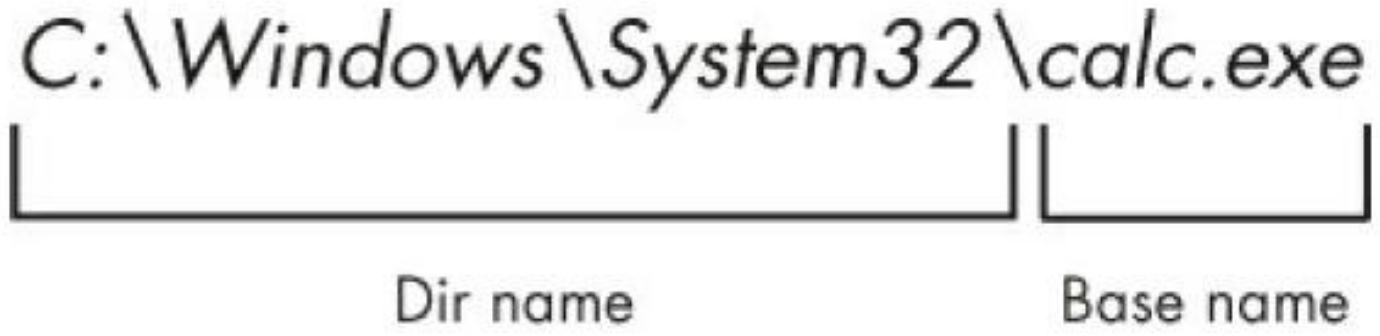
Getting the Parts of a File Path

- These attributes evaluate to simple string values, except for parent, which evaluates to another Path object.
- The parents attribute (which is different from the parent attribute)
- evaluates to the ancestor folders of a Path object with an integer index:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users/Al/AppData/Local/Programs')
>>> Path.cwd().parents[2]
WindowsPath('C:/Users/Al/AppData/Local')
>>> Path.cwd().parents[3]
WindowsPath('C:/Users/Al/AppData')
>>> Path.cwd().parents[4]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[5]
WindowsPath('C:/Users')
>>> Path.cwd().parents[6]
WindowsPath('C:/')
```

Getting the Parts of a File Path

- The older os.path module also has similar functions for getting the different parts of a path written in a string value.
- Calling os.path.dirname(path) will return a string of everything that comes before the last slash in the path argument.
- Ex: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.



Getting the Parts of a File Path

- Ex:

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

- If you need a path’s dir name and base name together, you can just call os.path.split() to get a tuple value with these two strings, like:

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

Getting the Parts of a File Path

- Note that you could create the same tuple by calling os.path.dirname() and os.path.basename() and placing their return values in a tuple:

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

- But os.path.split() is a nice shortcut if you need both values.

- Note that os.path.split() does not take a file path and return a list of strings of each folder.
- For that, use the split() string method and split on the string in os.sep.

Getting the Parts of a File Path

- The os.sep variable is set to the correct folder-separating slash for the computer running the program, '\\' on Windows and '/' on macOS and Linux, and splitting on it will return a list of the individual folders.
- Ex:

```
>>> calcFilePath.split(os.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

- This returns all the parts of the path as strings.
- On macOS and Linux systems, the returned list of folders will begin with a blank string, like this:

```
>>> '/usr/bin'.split(os. sep)
['', 'usr', 'bin']
```

The split() string method will work to return a list of each part of path.

Finding File Sizes and Folder Contents

- The os.path module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.
- 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')
27648
>>> 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']
```

calc.exe program on computer is 27,648 bytes in size, and there are lot of files in C:\Windows\system32.

Finding File Sizes and Folder Contents

- If I want to find the total size of all the files in this directory, I can use os.path.getsize() and os.listdir() together.
- As I loop over each filename in the C:\Windows\System32 folder, the totalSize variable is incremented by the size of each file.
- Notice how when I call os.path.getsize(), I use os.path.join() to join the folder name with the current filename.
- The integer that os.path.getsize() returns is added to value of totalSize.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
totalSize = totalSize +
os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
>>> print(totalSize)
2559970473
```

After looping through all the files, I print totalSize to see the total size of C:\Windows\System 32 folder.

Modifying a List of Files Using Glob Patterns

- To work on specific files, the glob() method is simpler to use than listdir().

```
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at 0x000002A6E389DED0>
>>> list(p.glob('*')) # Make a list from the generator.
[WindowsPath('C:/Users/Al/Desktop/1.png'), WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/Al/Desktop/cat.jpg'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

- Path objects have a glob() method for listing the contents of a folder according to a glob pattern.
- Glob patterns are like a simplified form of regular expressions often used in command line commands.
- The glob() method returns a generator object that you'll need to pass to list() to easily view in the interactive shell:

Modifying a List of Files Using Glob Patterns

- The asterisk (*) stands for “multiple of any characters,” so p.glob('*') returns a generator of all files in the path stored in p.
- Like with regexes, you can create complex expressions:
- The glob pattern '*.txt' will return files that start with any combination of characters as long as it ends with the string '.txt', which is the text file extension.

```
>>> list(p.glob('*.txt') # Lists all text files.
[WindowsPath('C:/Users/Al/Desktop/foo.txt'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Modifying a List of Files Using Glob Patterns

- In contrast with the asterisk, the question mark (?) stands for any single character:

```
>>> list(p.glob('project?.docx')
[WindowsPath('C:/Users/Al/Desktop/project1.docx'),
WindowsPath('C:/Users/Al/
Desktop/project2.docx'),
--snip--
WindowsPath('C:/Users/Al/Desktop/project9.docx')]
```

- The glob expression 'project?.docx' will return 'project1.docx' or 'project5.docx', but it will not return 'project10.docx', because ? Only matches to one character—so it will not match to the two-character string

Modifying a List of Files Using Glob Patterns

- combine the asterisk and question mark to create even more complex glob expressions

```
>>> list(p.glob('*.?x?')
[WindowsPath('C:/Users/Al/Desktop/calc.exe'),
WindowsPath('C:/Users/Al/
Desktop/foo.txt'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

- The glob expression '*.?x?' will return files with any name and any three-character extension where the middle character is an 'x'.
- By picking out files with specific attributes, the glob() method lets you easily specify the files in a directory you want to perform some operation on.

Modifying a List of Files Using Glob Patterns

- combine the asterisk and question mark to create even more complex glob expressions

- Use a for loop to iterate over the generator that glob() returns:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> for textFilePathObj in p.glob('*.txt'):
... print(textFilePathObj) # Prints the Path object as a
string.
... # Do something with the text file.
...
C:\Users\Al\Desktop\foo.txt
C:\Users\Al\Desktop\spam.txt
C:\Users\Al\Desktop\zzz.txt
```

If you want to perform some operation on every file in a directory, use either os.listdir(p) or p.glob('*').

Checking Path Validity

- Many Python functions will crash with an error if you supply them with a path that does not exist.
- Path objects have methods to check whether a given path exists and whether it is a file or folder.
- Assuming that a variable p holds a Path object, you could expect the following:
  - ✓ Calling p.exists() returns True if path exists or returns False if it doesn't exist.
  - ✓ Calling p.is_file() returns True if the path exists and is a file, or returns False otherwise.
  - ✓ Calling p.is_dir() returns True if the path exists and is a directory, or returns False otherwise.

Checking Path Validity

```
>>> winDir = Path('C:/Windows')
>>> notExistsDir = Path('C:/This/Folder/Does/Not/Exist')
>>> calcFile = Path('C:/Windows
/System32/calc.exe')
>>> winDir.exists()
True
>>> winDir.is_dir()
True
>>> notExistsDir.exists()
False
>>> calcFile.is_file()
True
>>> calcFile.is_dir()
False
```

- You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the exists() method.
- For instance, if I wanted to check for a flash drive with the volume named D:\ on my Windows computer, I could do that with the following:

Checking Path Validity

```
>>> dDrive = Path('D:/')
>>> dDrive.exists()
False
```

- Oops! It looks like I forgot to plug in my flash drive.
- The older os.path module can accomplish the same task with the os.path.exists(path), os.path.isfile(path), and os.path.isdir(path) functions, which act just like their Path function counterparts.
- As of Python 3.6, these functions can accept Path objects as well as strings of the file paths.

The File Reading/Writing Process

- The functions covered in the next few sections will apply to plaintext files.
- Plaintext files contain only basic text characters and do not include font, size, or color information.
- Text files with the .txt extension or Python script files with the .py extension are examples of plaintext files.
- These can be opened with Windows's Notepad or macOS's TextEdit application.
- Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

The File Reading/Writing Process

- Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs.
- If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like :

The Windows calc.exe program opened in Notepad

The File Reading/Writing Process

- The pathlib module's read_text() method returns a string of the full contents of a text file. Its write_text() method creates a new text file (or overwrites an existing one) with the string passed to it.

```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

The read_text() call reads and returns the contents of our new file as a string: 'Hello, world!'.

- These method calls create a spam.txt file with the content 'Hello, world!'.
- The 13 that write_text() returns indicates that 13 characters were written to file.

The File Reading/Writing Process

- The more common way of writing to a file involves using the open() function and file objects.

- 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.

Opening Files with the open() Function

- To open a file with the open() function, you pass it a string path indicating the file you want to open;
- It can be either an absolute or relative path.
- The open() function returns a File object.
- creating a text file named hello.txt using Notepad or TextEdit.
- Type Hello, world! as the content of this text file and save it in user home folder.
- Then enter: >>> helloFile = open(Path.home() / 'hello.txt')

Opening Files with the open() Function

- The open() function can also accept strings.
- If you're using Windows, enter:

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

- If you're using macOS, enter:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

- Make sure to replace your_home_folder with your computer username.
- Ex: my username is Al, so I'd enter 'C:\\Users\\Al\\hello.txt' on Windows.

Opening Files with the open() Function

- Note that the open() function only accepts Path objects as of Python 3.6.
- In previous versions, you always need to pass a string to open().
- Both these commands will open the file in “reading plaintext” mode, or read mode for short.
- When a file is opened in read mode, Python lets you only read data from the file; you can’t write or modify it in any way.
- Read mode is the default mode for files you open in Python.
- But if you don’t want to rely on Python’s defaults, you can explicitly specify the mode by passing the string value 'r' as a second argument to open().
- So open('/Users/Al/hello.txt', 'r') and open('/Users/Al/hello.txt') do the same thing.

Opening Files with the open() Function

- The call to open() 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 you're already familiar with.
- In the previous example, you stored the File object in the variable helloFile.
- Now, whenever you want to read from or write to the file, you can do so by calling methods on the File object in helloFile.

Reading the Contents of Files

- If you want to read the entire contents of a file as a string value, use the File object's read() method.
- Let's continue with the hello.txt File object you stored in helloFile.
- Ex:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello, world!'
```

If you think of the contents of a file as a single large string value, the read() method returns the string that is stored in the file.

Reading the Contents of Files

- Alternatively, use the readlines() method to get a list of string values from the file, one string for each line of text.
- Create a file named sonnet29.txt in the same directory as hello.txt and write the following text in it:

When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,

Reading the Contents of Files

- Make sure to separate the four lines with line breaks. Then enter the following into the interactive shell:

```
>>> sonnetFile = open(Path.home() / 'sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone
 beweep my outcast state,\n', And trouble deaf heaven with my
 bootless cries,\n', And look upon myself and curse my fate,']
```

- Note that, except for the last line of the file, each of the string values ends with a newline character \n.
- A list of strings is often easier to work with than a single large string value.

Writing to Files

- Python allows you to write content to a file in a way similar to how the print() function “writes” strings to the screen.
- You can’t write to a file you’ve opened in read mode, though. Instead, you need to open it in “write plaintext” mode or “append plaintext” mode, or write mode and append mode for short.
- Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable’s value with a new value.
- Pass 'w' as the second argument to open() to open the file in

Writing to Files

- Append mode, on the other hand, will append text to the end of existing file.
- You can think of this as appending to a list in a variable, rather than overwriting the variable altogether.
- Pass 'a' as the second argument to open() to open file in append mode.
- If the filename passed to open() does not exist, both write and append mode will create a new, blank file.
- After reading/writing a file, call close() method before opening file

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello, world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a
vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello, world!
Bacon is not a vegetable.
```

Writing to Files

- Note that the write() method does not automatically add a newline character to the end of the string like the print() function does.
- You will have to add this character yourself.
- As of Python 3.6, you can also pass a Path object to the open() function instead of a string for the filename.

Writing to Files

- First, open bacon.txt in write mode.
- Since there isn’t a bacon.txt yet, Python creates one.
- Calling write() on the opened file and passing write() the string argument 'Hello, world! /n' writes the string to the file and returns the number of characters written, including the newline.
- Then close the file.
- To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode.
- We write 'Bacon is not a vegetable.' to the file and close it.
- Finally, to print file contents to the screen, open file in its default read mode, call read(), store resulting File object in content, close file, and print content.

Writing to Files

- Note that the write() method does not automatically add a newline character to the end of the string like the print() function does.
- You will have to add this character yourself.
- As of Python 3.6, you can also pass a Path object to the open() function instead of a string for the filename.

Saving Variables with the shelve Module

- You can save variables in your Python programs to binary shelf files using the shelve module.
- This way, program can restore data to variables from the hard drive.
- The shelve module will let to add Save and Open features to program.
- Ex: 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.

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

Saving Variables with the shelve Module

- To read and write data using the shelve module, you first import shelve.
- Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable.
- You can make changes to the shelf value as if it were a dictionary.
- When you're done, call close() on the shelf value.
- Here, our shelf value is stored in shelfFile.
- We create a list cats and write shelfFile['cats'] = cats to store the list in shelfFile as a value associated with the key 'cats' (like in a dictionary).
- Then we call close() on shelfFile.
- Note that as of Python 3.7, you have to pass the open() shelf method

Saving Variables with the shelve Module

- After running the previous code on Windows, you will see three new files in the current working directory: mydata.bak, mydata.dat, and mydata.dir.

- On macOS, only a single mydata.db file will be created.
- These binary files contain the data you stored in your shelf.
- The format of these binary files is not important; you only need to know what the shelve module does, not how it does it.
- The module frees you from worrying about how to store your program's data to a file.

Saving Variables with the shelve Module

- Your programs can use the shelve module to later reopen and retrieve the data from these shelf files.
- Shelf values don't have to be opened in read or write mode—they can do both once opened.
- Here, we open the shelf files to check that our data was stored correctly.
- Entering shelfFile['cats'] returns the same list that we stored earlier, so we kno that the list is correctly stored, and we call close().

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

Saving Variables with the shelve Module

- Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf.
- Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form.

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

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

- pprint.pprint() function will “pretty print” the contents of a list or dictionary to the screen, while 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
- Ex: 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 a .py file.
- This file will be your very own module that you can import whenever you want to use the variable stored in it.

Saving Variables with the shelve Module

- Import pprint to let us use pprint.pformat(). We have a list of dictionaries, stored in a variable cats.
- To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string.

```
>>> 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()
```

- Once we have the data in cats as a string, it's easy to write the string to a

file, which we'll call myCats.py

Saving Variables with the shelve Module

- The modules that an import statement imports are themselves just Python scripts.
- When the string from pprint.pformat() is saved to a .py file, the file is a module that can be imported just like any other.

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

- since Python scripts are themselves just text files with the .py file extension, your Python programs can even generate other Python programs.
- You can then import these files into scripts.

Saving Variables with the 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.

- For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. For most applications, however, saving data using the shelve module is the preferred way to save variables to a file.

- 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.

Project: Generating Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

Project: Generating Random Quiz Files

1. Creates 35 different quizzes
2. Creates 50 multiple-choice questions for each quiz, in random order
3. Provides the correct answer and three random wrong answers for
each question, in random order
4. Writes the quizzes to 35 text files
5. Writes the answer keys to 35 text files

Project: Generating Random Quiz Files

This means the code will need to do the following:

1. Store the states and their capitals in a dictionary
2. Call open(), write(), and close() for the quiz and answer key text files
3. Use random.shuffle() to randomize the order of the questions and multiple-choice options

Project: Generating Random Quiz Files

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data:

Create a file named randomQuizGenerator.py, and make it look like following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
❶ import random
# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany',
```

Project: Generating Random Quiz Files

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data:

Create a file named randomQuizGenerator.py, and make it look like following:

```
'North Carolina': 'Raleigh', 'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City'
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}
# Generate 35 quiz files.
❸ for quizNum in range(35):
# TODO: Create the quiz and answer key files.
# TODO: Write out the header for the quiz.
# TODO: Shuffle the order of the states.
# TODO: Loop through all 50 states, making a question for each.
```

Project: Generating Random Quiz Files

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data:
Create a file named randomQuizGenerator.py, and make it look like following:

Since this program will be randomly ordering the questions and answers, you'll need to import the random module
❶ to make use of its functions.
The capitals variable ❷ contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with TODO comments for now) will go inside a for loop that loops 35 times
❸. (This number can be changed to generate any number of quiz files.)

Step 2: Create the Quiz File and Shuffle the Question Order

- Now it's time to start filling in those TODOs.
- The code in the loop will be repeated 35 times—once for each quiz— so you have to worry about only one quiz at a time within the loop. First
- you'll create the actual quiz file.
- It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period.
- Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Step 2: Create the Quiz File and Shuffle the Question Order

- Add the following lines of code to randomQuizGenerator.py:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
--snip--
# Generate 35 quiz files.
for quizNum in range(35):
# Create the quiz and answer key files.
```

Step 2: Create the Quiz File and Shuffle the Question Order

➢ Add the following lines of code to randomQuizGenerator.py:

```
❶ quizFile = open(f'capitalsquiz{quizNum + 1}.txt', 'w')
❷ answerKeyFile = open(f'capitalsquiz_answers{quizNum + 1}.txt', 'w')
# Write out the header for the quiz.
❸ quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
quizFile.write((' ' * 20) + f'State Capitals Quiz (Form{quizNum + 1})')
quizFile.write('\n\n')
# Shuffle the order of the states.
states = list(capitals.keys())
❹ random.shuffle(states)
# TODO: Loop through all 50 states, making a question for each.
```

Step 3: Create the Answer Options

- Now you need to generate the answer options for each question, which will be multiple choice from A to D.
- You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz.
- Then there will be a third for loop nested inside to generate the
- multiple-choice options for each question.
- Make your code look like the following:

Step 3: Create the Answer Options

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
--snip--
# Loop through all 50 states, making a question for each.
for questionNum in range(50):
# Get right and wrong answers.
❶ correctAnswer = capitals[states[questionNum]]
❷ wrongAnswers = list(capitals.values())
❸ del wrongAnswers[wrongAnswers.index(correctAnswer)]
❹ wrongAnswers = random.sample(wrongAnswers, 3)
❺ answerOptions = wrongAnswers + [correctAnswer]
❻ random.shuffle(answerOptions)
# TODO: Write the question and answer options to the quiz file.
# TODO: Write the answer key to a file.
```

Step 3: Create the Answer Options

- The correct answer is easy to get—it's stored as a value in the capitals dictionary ❶.
- This loop will loop through the states in the shuffled stateslist, from states[0] to states[49], find each state in capitals, and store that state's corresponding capital in correctAnswer.
- The list of possible wrong answers is trickier.
- You can get it by duplicating all the values in the capitals dictionary ❷, deleting the correct answer ❸, and selecting three random values from this list ❹.

Step 3: Create the Answer Options

- The random.sample() function makes it easy to do this selection.
- Its first argument is the list you want to select from; the second argument is the number of values you want to select.
- The full list of answer options is the combination of these three wrong answers with the correct answers ❺.
- Finally, the answers need to be randomized ❻ so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

- Write the question to quiz file and the answer to answer key file.
- Make your code look like the following:

```
#! python3
# randomQuizGenerator.py – Creates  quizzes with questions and answers in
# random order, along with the answer key.
--snip--
# Loop through all 50 states, making a question for each.
for questionNum in range(50):
--snip--
```

Step 4: Write Content to the Quiz and Answer Key Files

- Write the question to quiz file and the answer to answer key file.
- Make your code look like the following:

```
# Write the question and the answer options to the quiz file.
quizFile.write(f'{questionNum + 1}. What is the capital of
{states[questionNum]}?\n')
❶ for i in range(4):
❷ quizFile.write(f" {'ABCD'[i]}. { answerOptions[i]}\n")
quizFile.write('\n')
# Write the answer key to a file.
❸ answerKeyFile.write(f"{questionNum + 1}.
{'ABCD'[answerOptions.index(correctAnswer)]}")
quizFile.close()
answerKeyFile.close()
```

Step 4: Write Content to the Quiz and Answer Key Files

- A for loop that goes through integers 0 to 3 will write the answer options in the answerOptions list ❶.
- The expression 'ABCD'[i] at ❷ treats the string 'ABCD' as an array and will evaluate to 'A','B', 'C', and then 'D' on each respective iteration through the loop.
- In the final line ❸, the expression answerOptions.index (correctAnswer) will find the integer index of the correct answer in the randomly orderedanswer options, and 'ABCD'[answerOptions.index(correctAnswer)] will evaluate to the correct answer's letter to be written to the answer key file.

Step 4: Write Content to the Quiz and Answer Key Files

- After you run the program, this is how your capitalsquiz1.txt file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your random.shuffle() calls:

Name:
Date:
Period:
State Capitals Quiz (Form 1)
1. What is the capital of West Virginia?
A. Hartford
B. Santa Fe
C. Harrisburg
D. Charleston
2. What is the capital of Colorado?
A. Raleigh
B. Harrisburg
C. Denver
D. Lincoln
--snip--

Step 4: Write Content to the Quiz and Answer Key Files

- The corresponding capitalsquiz_answers1.txt text file will look like this:

```
1. D
2. C
3. A
4. C
--snip--
```

Project: Updatable Multi-Clipboard

- Let's rewrite the "multi-clipboard" program from Chapter 6 so that it uses the shelve module.
- The user will now be able to save new strings to load to the clipboard without having to modify the source code.
- We'll name this new program mcb.pyw (since "mcb" is shorter to type than "multi-clipboard").
- The .pyw extension means that Python won't show a Terminal window when it runs this program.
- (See Appendix B for more details.)

Project: Updatable Multi-Clipboard

- The program will save each piece of clipboard text under a keyword.
- Ex: when you run py mcb.pyw save spam, the current contents of the clipboard will be saved with the keyword spam.
- This text can later be loaded to the clipboard again by running py mcb.pyw spam.
- And if the user forgets what keywords they have, they can run py mcb.pyw list to copy a list of all keywords to the clipboard.

Project: Updatable Multi-Clipboard

- Here's what the program does:

1. The command line argument for the keyword is checked.
2. If the argument is save, then the clipboard contents are saved to the keyword.
3. If the argument is list, then all the keywords are copied to the clipboard.
4. Otherwise, the text for the keyword is copied to the clipboard.

Project: Updatable Multi-Clipboard

- This means the code will need to do the following:
  1. Read the command line arguments from sys.argv.
  2. Read and write to the clipboard.
  3. Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run... window by creating a batch file named mcb.bat with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

- Make a skeleton script with some comments and basic setup.

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
# py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
# py.exe mcb.pyw list - Loads all keywords to clipboard.
❷ import shelve, pyperclip, sys
❸ mcbShelf = shelve.open('mcb')
# TODO: Save clipboard content.
# TODO: List keywords and load content.
mcbShelf.close()
```

Step 1: Comments and Shelf Setup

- It's common practice to put general usage information in comments at top of file ❶.
- If you ever forget how to run your script, you can always look at these comments for a reminder.
- Then you import your modules ❷. Copying and pasting will require pyperclip module, and reading the command line arguments will require sys module.
- The shelve module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file.
- Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program.

Step 2: Save Clipboard Content with a Keyword

- The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case.
- Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--
# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
❷ mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
❸ # TODO: List keywords and load content.
mcbShelf.close()
```

Step 2: Save Clipboard Content with a Keyword

```
#! python3
# mcb.pyw - Saves and loads
pieces of text to the clipboard.
--snip--
# Save clipboard content.
❶ if len(sys.argv) == 3 and
sys.argv[1].lower() == 'save':
❷ mcbShelf[sys.argv[2]] =
pyperclip.paste()
elif len(sys.argv) == 2:
❸ # TODO: List keywords and
load content.
mcbShelf.close()
```

- If the first command line argument (which will always be at index 1 of the sys.argv list) is 'save' ❶, the second command line argument is the keyword for the current content of the clipboard.
- The keyword will be used as the key for mcbShelf, and the value will be the text currently on the clipboard ❷.
- If there is only one command line argument, you will assume it is either 'list' or a keyword to load content onto clipboard.
- You will implement that code later. For now, just put a TODO comment there ❸.

Step 3: List Keywords and Load a Keyword's Content

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--
# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
# List keywords and load content.
❶ if sys.argv[1].lower() == 'list':
❷ pyperclip.copy(str(list(mcbShelf.keys())))
elif sys.argv[1] in mcbShelf:
❸ pyperclip.copy(mcbShelf[sys.argv[1]])
mcbShelf.close()
```

- Finally, let's implement the two remaining cases: the user wants to load clipboard text in from a keyword, or they want a list of all available keywords.
- Make your code look like the following:

Step 3: List Keywords and Load a Keyword's Content

```
#! python3
# mcb.pyw - Saves and loads pieces of
text to the clipboard.
--snip--
# Save clipboard content.
if len(sys.argv) == 3 and
sys.argv[1].lower() == 'save':
mcbShelf[sys.argv[2]] =
pyperclip.paste()
elif len(sys.argv) == 2:
# List keywords and load content.
❶ if sys.argv[1].lower() == 'list':
❷
pyperclip.copy(str(list(mcbShelf.keys())))
elif sys.argv[1] in mcbShelf:
❸ pyperclip.copy(mcbShelf[sys.argv[1]])
```

- If there is only one command line argument, first check whether it's 'list' ❶.
- If so, a string representation of the list of shelf keys will be copied to clipboard ❷.
- The user can paste this list into an open text editor to read it.
- Otherwise, you can assume the command line argument is a keyword.
- If this keyword exists in the mcbShelf shelf as a key, you can load the value onto the clipboard ❸.
- And that's it! Launching this program has different steps depending on what operating system your computer uses.
- See Appendix B for details.

Step 3: List Keywords and Load a Keyword's Content

```
#! python3
# mcb.pyw - Saves and loads pieces of
text to the clipboard.
--snip--
# Save clipboard content.
if len(sys.argv) == 3 and
sys.argv[1].lower() == 'save':
mcbShelf[sys.argv[2]] =
pyperclip.paste()
elif len(sys.argv) == 2:
# List keywords and load content.
❶ if sys.argv[1].lower() == 'list':
❷
pyperclip.copy(str(list(mcbShelf.keys())))
elif sys.argv[1] in mcbShelf:
❸ pyperclip.copy(mcbShelf[sys.argv[1]])
```

- Recall the password locker program you created in Chapter 6 that stored the passwords in a dictionary.
- Updating the passwords required changing the source code of the program.
- This isn't ideal, because average users don't feel comfortable changing source code to update their software.
- Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs.
- By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.