Module – 3

MANIPULATING STRINGS

Working with Strings

String Literals

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing `'That is Alice's cat.'` won't work, because Python thinks the string ends after `Alice`, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An escape character lets you 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.

```
>>> 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 the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

String Literals

Can we use a quote inside a string?
'That is Alice's cat.'

Double quotes
"That is Alice's cat."

Escape character
'Say hi to Bob\'s mother.'

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

```
'That is Alice's cat'
```

```
  File "<ipython-input-8-01a84325dd8d>", line 1
    'That is Alice's cat'
                        ^
SyntaxError: unterminated string literal (detected at line 1)
```

SEARCH STACK OVERFLOW

```
"That is Alice's cat"
```

```
'That is Alice's cat'
```

```
print('Hello there!\nHow are you?\nI\'m doing fine')
```

```
Hello there!
How are you?
I'm doing fine
```

```
print('Hello there!\nHow are you?\nI\'m doing fine')
print("Hello there!\nHow are you?\nI'm doing fine")
print("Hello there!\tHow are you?\tI'm doing fine")
print('Hello there!\\How are you?\\I\'m doing fine')
```

```
Hello there!
How are you?
I'm doing fine
Hello there!
How are you?
I'm doing fine
Hello there!	How are you?	I'm doing fine
Hello there!\How are you?\I'm doing fine
```

print( ) using escape characters & raw strings

```
Hello there!

How are you?

I'm doing fine.
```

print("Hello there!\nHow are you?\nI\'m doing fine.")

```
print(r'Hello there!\nHow are you?\nI\'m doing fine')
```

```
Hello there!\nHow are you?\nI\'m doing fine
```

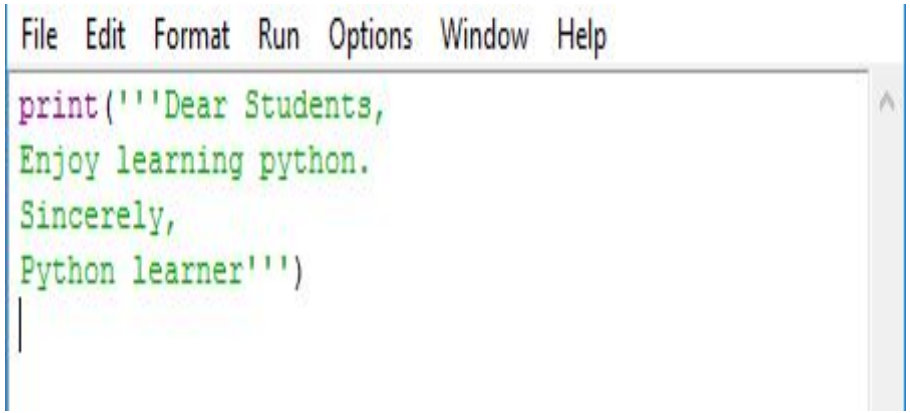
A raw string completely ignores all escape characters and prints any backslash that appears in the string.

print(r'That is Carol\'s cat.')

```
That is Carol\'s cat.
```

Multiline string with three single quotes

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.




```
>>>
=== RESTART: C:/Users/Lenovo/AppData/Local/Pro
grams/Python/Python311/nulti.py ==
Dear Students,
Enjoy learning python.
Sincerely,
Python learner
```

Python’s indentation rules for blocks do not apply to lines inside a multiline string.

```
print('''Dear Alice,

      Eve's cat has been arrested for catnapping, cat burgarly, and extrtion.
      Sincerely,
      Bob''')
```

```
Dear Alice,

      Eve's cat has been arrested for catnapping, cat burgarly, and extrtion.
      Sincerely,
      Bob
```

 # Multiline string using single quote

```
print("""Dear Alice,

      Eve's cat has been arrested for catnapping, cat burgarly, and extrtion.
      Sincerely,
      Bob""")
```

```
Dear Alice,

      Eve's cat has been arrested for catnapping, cat burgarly, and extrtion.
      Sincerely,
      Bob
```

 # Multiline string using double quote

Without using multiline string

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

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burgarly, and extrtion.

Sincerely,
Bob
```

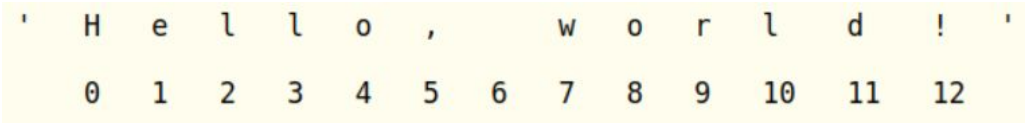
Multiline Comments

```
"""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!')
```

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.

Indexing and Slicing Strings

'Hello, world!'



Each string value can be thought of as a list and each character in the string as an item with a corresponding index

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

-ve index count from the end

Slicing string

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

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

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.

The in and not in Operators with Strings

```
>>> 'Hello' in 'Hello, World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello, World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

The in and not in operators can be used with strings just like with list values.

An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

Putting Strings Inside Other Strings

```
>>> 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.'
```

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' % (name, age)
'My name is Al. I am 4000 years old.'
```

string interpolation using %s

```
>>> 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.'
```

f-strings, which is similar to string interpolation except that braces are used instead of %s

Built-in methods for string manipulation

Useful String Methods

upper(), lower(), isupper(), and islower() Methods

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

```
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

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.

```
'abcABC'.islower()
```

```
False
```

```
'abcABC'.isupper()
```

```
False
```

```
'ABC'.isupper()
```

```
True
```

```
'1234ABC'.isupper()
```

```
True
```

```
'1234abc'.islower()
```

```
True
```

upper( ) and lower( ) chain

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

Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well.

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

```
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.')
```

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 likely to fail.

```
How are you?
GREat
I feel great too.
```

Nonletter characters in the string remain changed or unchanged

Asw: Unchanged

- **String 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() and lower() on the string and then assign the new string to the variable where the original was stored.

```
spam='Hello world!'
spam.upper()
spam
```

Output:

Hello world!

Guess the output:

```
spam='Hello world!'
spam.upper()
spam
```

Output:
Hello world!

```
spam='Hello world!'
spam=spam.upper()
Spam
```

Output:
HELLO WORLD!

```
for i in range(2,2):
  print(i)
```

Output:
It will not print anything

When do we use upper ()and lower() methods?
These methods are helpfull when you do a case-insensitive comparison.

isX() Methods

- isalpha() Returns True if the string consists only of letters and isn't blank
- isalnum() Returns True if the 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

isX( ) examples

```
>>> '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
```

```
' '.isalpha()
False
'helloHello'.isalpha()
True
'hello123'.isalnum()
True
'01234'.isdecimal()
True
'01234c'.isdecimal()
False
'hello123!'.isalnum()
False
'123'.isalnum()
True
'abcAbc'.isalnum()
True
'This is String Methods'.istitle()
False
'This Is String Methods'.istitle()
True
'This Is String METHODS'.istitle()
False
```

‘ ’.isspace()

True

‘\n’.isspace()

True

‘ My name ’.isspace()

False

‘ \n ’.isspace()

True

‘ \n hi ’.isspace()

False

The **isX methods** are helpful when you need to **validate user input**.

Example:

The following program **repeatedly asks user for their age and a password until they provide valid input.**

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

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

In the second while loop, we ask for a password, store the user's input in password, and break out of the loop if the input was alphanumeric.
If it wasn't, we're not satisfied so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

```
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, we're able to test whether the values stored in those variables are decimal or not, alphanumeric or not. Here, these tests help us reject the input forty two and accept 42, andreject secr3t! and accept secr3t.

startswith() and endswith()

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.

```
>>> '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
```

These methods are useful alternatives to the ‘==‘ operator.

'Hello World!'.startswith()

Error

'Hello World!'.startswith('Hello')

True

'Hello World!'.startswith('HEllo')

False

'Hello World!'.endswith('Hello')

False

'Hello World!'.endswith(‘World')

False

'My name is smitha'.endswith('smitha')

True

join() method

The join() method is useful when you have a list of strings that need to be joined together into a single string value.

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

This method is called on a string and passed a list of string

split ( ) method

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

It does opposite to join()

It is called on a string value and return a list of strings.

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 the strings in the returned list.

split ( ) using a delimiter

```
>>> 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']
```

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

```
'My name is simon'.split('m')
```

Output:

['My na', 'e is si', 'on‘]

```
spam='''Dear Alice
 How   are you doing?
 Please take care of your health'''
spam.split()
```

Output:

['Dear', 'Alice', 'How', 'are', 'you', 'doing?', 'Please', 'take', 'care', 'of', 'your', 'health']

```
spam='''Dear Alice
How  are you doing?
Please take care of your health'''
spam.split('\n')
```

Output:

['Dear Alice', 'How are you doing?', 'Please take care of your health']

Splitting Strings with the partition() Method

```
>>> '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:

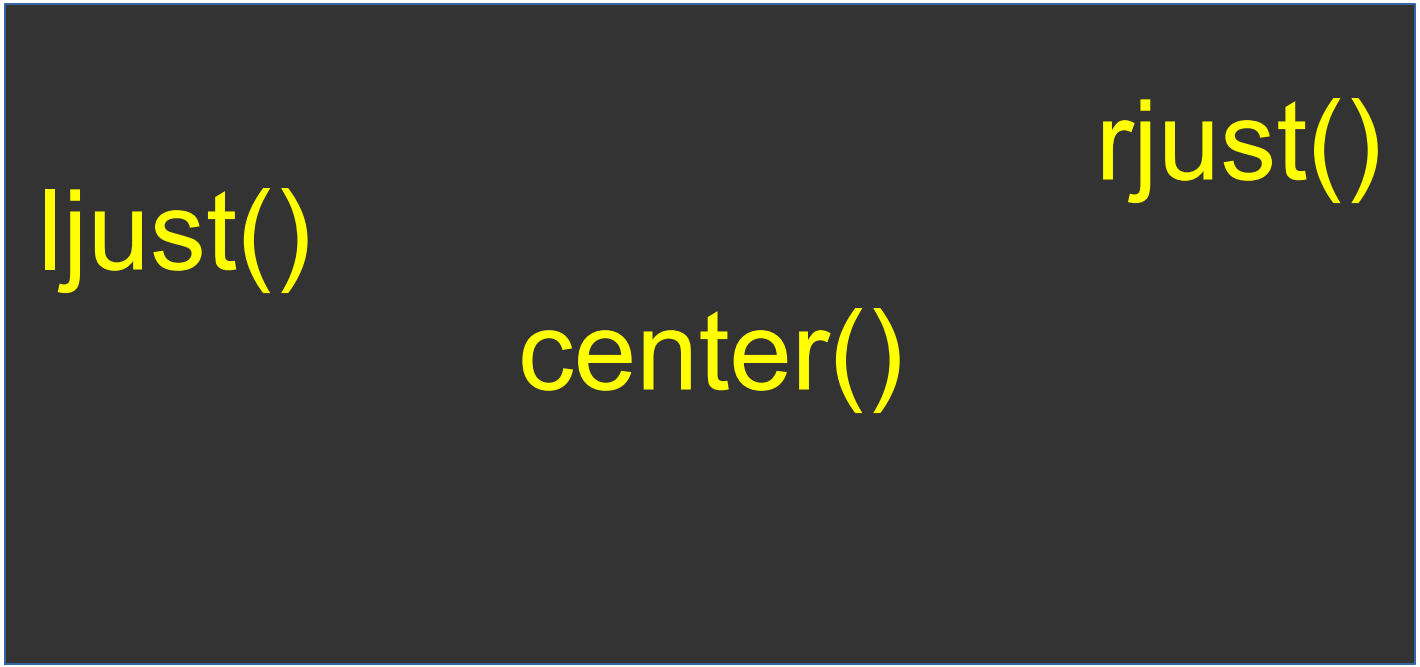
```
>>> 'Hello, world!'.partition('o')
('Hell', 'o', ', world!')
```

Using partition( ) for multiple assignment

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

```
>>>fruit,sep,vegetable='Apple Carrot'.partition(‘ ')
>>>vegetable
‘Carrot’
>>>fruit
‘Apple’
```

Justifying Text



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

- **The rjust() and ljust()** string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- The **first argument** to both methods is an integer length for the justified string.
- An optional **second argument** to rjust() and ljust() will specify a fill character other than a space character.

Justifying text

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

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

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

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)
```

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

In this program, we define a printPicnic() method that will take in a dictionary of information and use center(), ljust(), and rjust() to display that information in a neatly aligned table-like format.

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.

In picnicItems, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

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

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

- whitespace characters (space, tab, and newline)
- 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

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

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

Passing `strip()` the argument `'ampS'` will tell it to strip occurences 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 the ord() and chr() Functions

Every text character has a corresponding numeric value called a Unicode code point. For example, the numeric code point is 65 for 'A'

```
>>> ord('A')
65
>>> ord('4')
52
```

The ord() function can be used 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

```
>>> chr(65)
'A'
```

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

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

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 your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

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

NOTE: pyperclip module should be available in the local environment

Generate N prime number sequence

```
list=[]
for i in range(2,101):
  flag=0
  for j in range(2,i):
    if i%j==0:
     flag=1
     break
  if flag==0:
    list.append(i)
print('The prime numbers are:',list)
```

Output:

The prime numbers are: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Project

Reading and Writing Files

File Handling in Python

Files and File Paths

- A file has two key properties: a filename (usually written as one word) and a path.
- The path specifies the location of a file on the computer. For example: there is a file on my Windows 7 laptop with the filename projects.docx in the path C:\Users\asweigart\Documents.

  The part of the filename after the last period is called the file's extension and tells you a file's type.

- project.docx is a Word document, and Users, asweigart, and Documents all refer to folders/ directories
- Folders can contain files and other folders.

  For example, project.docx is in the Documents folder, which is inside the asweigart folder, which is inside the Users folder.
- 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 the C: drive.
- On OS X and Linux, the root folder is /.

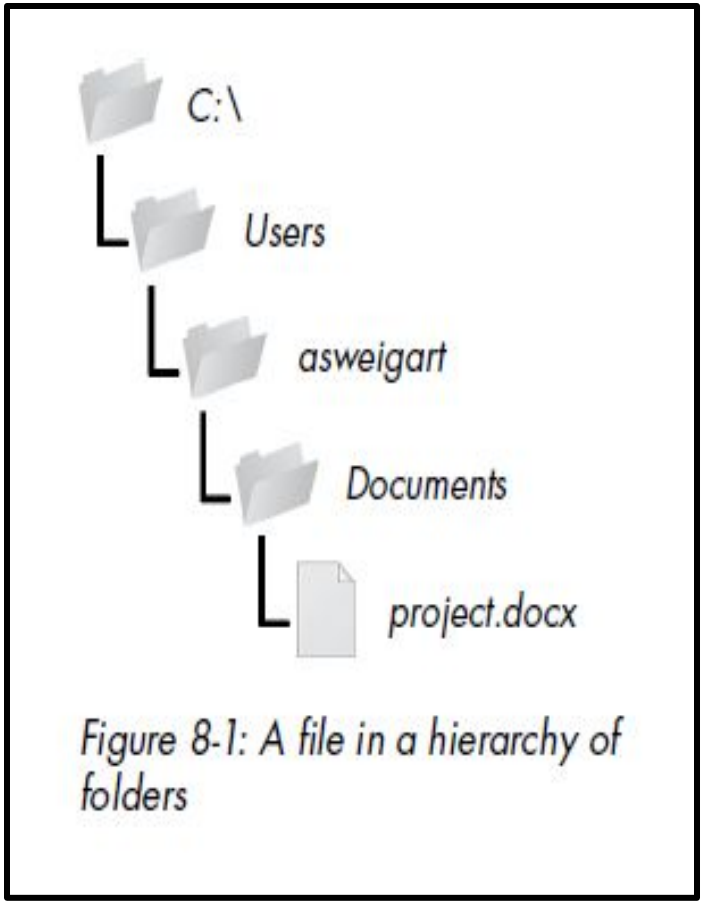


Figure 8-1: A file in a hierarchy of folders

Backslash on Windows and Forward Slash on OS X and Linux

- On Windows, paths are written using backslashes (\) as the separator between folder names.
- OS X and Linux, however, use the forward slash (/) as their path separator.
- If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.
- This is simple to do with the os.path.join() function.
- If you pass it the string values of individual file and folder names in your path, os.path.join() will return a string with a file path using the correct path separators.

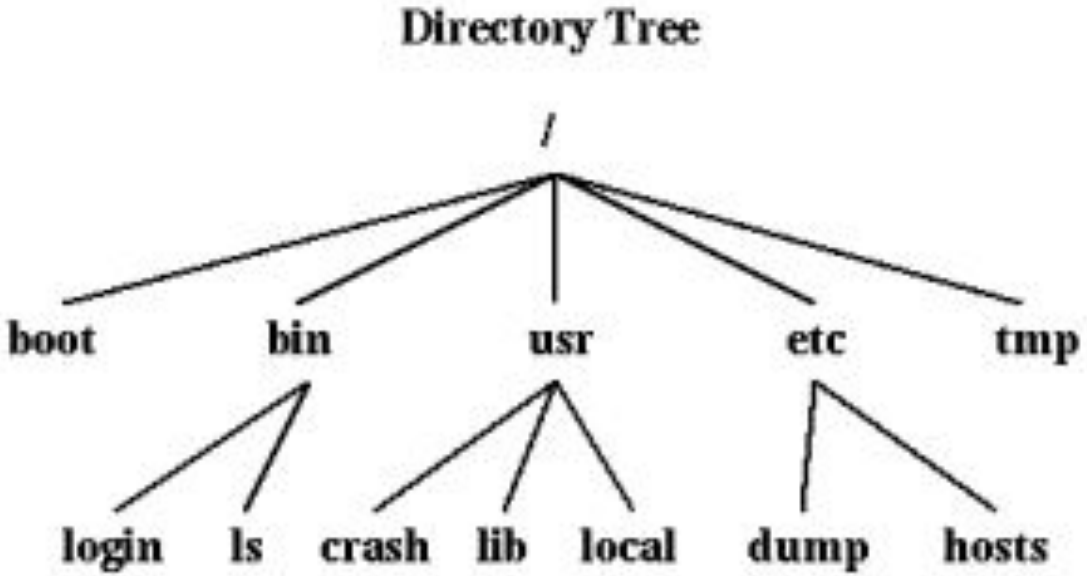
- os.path.join('usr', 'bin', 'spam') returned 'usr\\bin\\spam‘ in windows.
- If I had called this function on OS X or Linux, the string would have been 'usr/bin/spam'.
- The os.path.join() function is helpful if you need to create strings for filenames.

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

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

example joins names from a list of filenames to the end of a folder’s name

Filesystem in UNIX



File handling commands in UNIX

pwd : present working directory

ls : list files

Home Directory (/)

cd : change directory

cd / : change to root directory

mkdir : make directory / create directory

Copy and move commands: cp and mv

pathlib module in python

A built-in module called pathlib is available in python 3.4 onwards for handling files.

File management includes creating, moving, copying, and deleting files and directories, checking if a file or directory exists,and so on.

The Current Working Directory

- Every program that runs on your 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.
- You can get the current working directory as a string value with the os.getcwd() function and change it with os.chdir().

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Python will display an error if you try to change to a directory that does not exist.

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

Absolute vs. Relative Paths

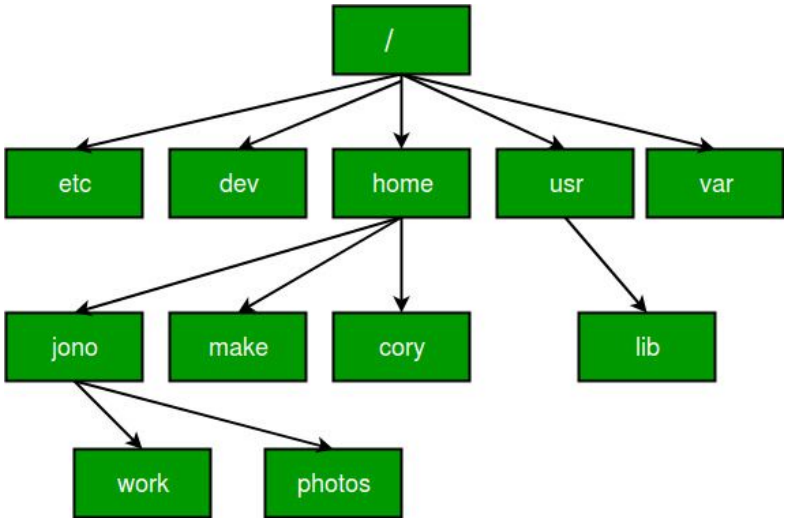
There are two ways to specify a file path:

An **absolute path**, which always begins with the root folder

A **relative path**, which is relative to the program's current working directory

- There are also 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.**"

(dot) and (dot dot) operators




```
$pwd
/home/kt/abc
$cd ..                  ***moves one level up***
$pwd
/home/kt
```

```
$pwd
/home/kt/abc            ***moves two level up***
$cd ../..
$pwd
/home
```

Demo in Terminal

. (this directory)
.. (parent directory)

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

Creating New Folders with os.makedirs()

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. That is, os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists.

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

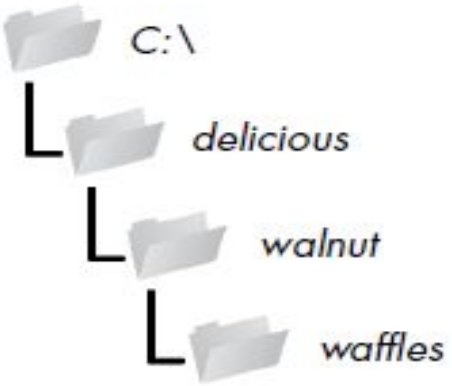


Figure 8-3: The result of `os.makedirs('C:\\delicious\\walnut\\waffles')`

The os.path Module

The `os.path` module contains many helpful functions related to filenames and file paths. For instance, you've already used `os.path.join()` to build paths in a way that will work on any operating system. Since `os.path` is a module inside the `os` module, you can import it by simply running `import os`. Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section. The full documentation for the `os.path` module is on the Python website at *http://docs.python.org/3/library/os.path.html*.

Handling Absolute and Relative Paths

The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

- Calling os.path.abspath(path) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path.
- Calling os.path.relpath(path, start) will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path.

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

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

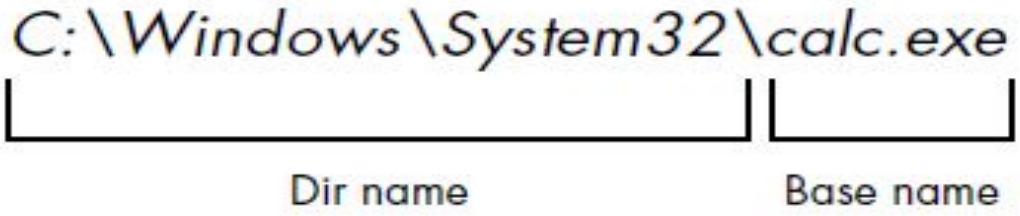


Figure 8-4: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

Finding File Sizes and Folder Contents

Calling os.path.getsize(path) will return the size in bytes of the file in the path argument.

Calling os.listdir(path) will return a list of filename strings for each file in the path argument

```
>>> 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']
```

dirname and basename

```
>>> path='c:\\windows\\system32\\calc.exe'
>>> import os
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'c:\\windows\\system32'
```

os.path.split()

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 so:

```
>>> path='c:\\windows\\system32\\calc.exe'
>>> os.path.split(path)
('c:\\windows\\system32', 'calc.exe')
```

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
```

Notice 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')
```

os.path.sep()

For example, enter the following into the interactive shell:

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

On OS X and Linux systems, there will be a blank string at the start of the returned list:

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

The `split()` string method will work to return a list of each part of the path. It will work on any operating system if you pass it `os.path.sep`.

Finding File Sizes and Folder Contents

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

```
>>> os.listdir('C:\\users')
['All Users', 'Default', 'Default User', 'desktop.ini', 'Public', 'Smitha']
>>> os.path.getsize('C:\\users\\smitha')
8192
```

To find the total size of all the files in this directory, we can use os.path.getsize() and os.listdir() together.

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

>>> print(totalSize)
1117846456
```

Checking Path Validity

Calling os.path.exists(path) returns True if the path exists or returns False if it doesn’t exist.

Calling os.path.isfile(path) returns True if the path exists and is a file, or returns False otherwise.

Calling os.path.isdir(path) returns True if the path exists and is a directory, or returns False otherwise.

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

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `os.path.exists()` function. 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:

```
>>> os.path.exists('D:\\')
False
```

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.
- **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 in

Three steps to reading or writing files in Python

Call the open() function to return a File object.

Call the read() or write() method on the File object.

Close the file by calling the close() method on the File object.

Binary Files

Opening a calc.exe in windows using notepad

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.

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

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

open('/Users/asweigart/hello.txt', 'r')

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.

use the **readlines()** method to get a *list of string* values from the file, one string for each line of text.

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

```
>>> sonnetFile = open('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,']
```

Read the content of the **file**

```
[2] f=open('dict.txt')
    f.read()
```

0s

```
'VTU Belgaum\nMy name is Smitha\nGraduate under VTU\nBelgaum'
```

To read the content of the file line by line there are 3 commands:

Read few characters

```
f=open('dict.txt')
print(f.read(5))
```

```
VTU B
```

dict.txt

```
1 VTU Belgaum
2 My name is Smitha
3 Graduate under VTU
4 Belgaum
```

readline()

```
[14] f=open('dict.txt')
     print(f.readline())
```

```
VTU Belgaum
```

readlines()

```
[15] f=open('dict.txt')
     print(f.readlines())
```

```
['VTU Belgaum\n', 'My name is Smitha\n', 'Graduate under VTU\n', 'Belgaum']
```

Writing to Files

- Similar to how the print() function “writes” strings to the screen.
- Pass 'w' as the second argument to open() to open the file in write mode.
- Append mode, on the other hand, will append text to the end of the existing file. Pass 'a' as the second argument to open() to open the file in append mode.

```
>>> 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.
```

Write to the file

```
f=open('new.txt','w')
print(f.write('hello'))
f.close()
```

```
5
```

Before

new.txt

```
1 VTU Belgaum
2 My name is Smitha
3 graduate under VTU
4 Belgaum
```

After

new.txt

```
1 hello
```

append content to a file

```
[23] f=open('new.txt','a')
     print(f.write(' Welcome'))
     f.close()
```

```
8
```

Before

new.txt

```
1 VTU Belgaum
2 My name is Smitha
3 graduate under VTU
4 Belgaum
```

After

new.txt

```
1 VTU Belgaum
2 My name is Smitha
3 graduate under VTU
4 Belgaum Welcome
```

5. Develop a program to print 10 most frequently appearing words in a text file. [Hint: Use dictionary with distinct words and their frequency of occurrences. Sort the dictionary in the reverse order of frequency and display dictionary slice of first 10 items]

```
fname = input('Enter the file name: ')
fhand = open(fname)
counts = {}
for line in fhand:
   words = line.split()
   for word in words:
      if word in counts:
         counts[word] += 1
      else:
        counts[word] = 1
print('dictionary before sorting:',dict(counts))
print('sorted dictionary:',dict(sorted(counts.items())))
print('First 10 items:',dict(list(sorted(counts.items()))[0:10]))
```

OUTPUT

```
Enter the file name: prgm.txt
dictionary before sorting: {'VTU': 7, 'Belgaum': 6, 'My': 3, 'name': 3, 'is': 3, 'Smitha': 3, 'Graduate': 3, 'under': 3, 'Welcome': 2, 'to': 2, 'world': 1, 'of': 1, 'NIE': 1}
sorted dictionary: {'Belgaum': 6, 'Graduate': 3, 'My': 3, 'NIE': 1, 'Smitha': 3, 'VTU': 7, 'Welcome': 2, 'is': 3, 'name': 3, 'of': 1, 'to': 2, 'under': 3, 'world': 1}
First 10 items: {'Belgaum': 6, 'Graduate': 3, 'My': 3, 'NIE': 1, 'Smitha': 3, 'VTU': 7, 'Welcome': 2, 'is': 3, 'name': 3, 'of': 1}
```