Python Programming

INFO.2820.061

Instructor: Andy Garside

Strings and Text Files

Introduction

Text is used more than any other form of data. Text appears in documents, web pages, databases, and many other locations. In this lecture we will discuss some of Python's text capabilities, including methods for manipulating strings, and the tools needed to read and write text to files.

Strings

The Structure of Strings

We have seen and used strings a number of times in our lectures. For example,

```
>>> "Hello World!"
'Hello World!'
>>> ""
''
>>> 'A'
'A'
```

Recall that you can use either double quotes ("), or single quotes (') to define a string. In our textbook and the lecture notes we use single quotes when working with single character strings, and we use double quotes for all other strings.

From a user's perspective, a Python string is simple – it appears as a sequence of characters. From the Python language's perspective, it is more complex.

For each string, Python keeps not only the sequence of characters that make up the string, but other elements designed to help it efficiently work with the string. In short, a Python string is a **data structure**, a compound unit consisting of several other pieces of data. More generically, we refer to a Python string as a **string object**.

Note: A side effect of a string being an object is that even the simplest of strings, the empty string "", takes space. In fact, in Python, the empty string takes around 40 bytes.

In Python, a string is **immutable**, meaning it cannot be changed or mutated. Once a string is created, the characters of the string can be accessed (read) but they cannot be replaced, inserted or removed from the string object.

Note: Of course, there are times when programmers want to replace, insert and remove characters from a string. Since Python strings are immutable, we will see that this is effectively accomplished behind the scenes in Python by the creation of a second string that contains the characters of the original string with any desired modifications.

Sequence Types – Strings and Lists

A string in Python is a specific example of what is known as a **sequence data type** – in this case, a sequence of characters. There are a number of **sequence data types** in Python and they all share some characteristics.

A Python **list** is another sequence data type. We will cover lists in detail in a future lecture but it's useful to have a basic understanding of them now.

While a Python string is a sequence of characters, a Python **list** is a sequence of objects, for example integers, floating-point values, strings, or even other lists. (The elements in a list do **not** have to all be of the same type.)

We have seen that you can create a Python **string** by enclosing a sequence of characters in quotes (single or double quotes). For example, the first statement below creates the string "Hello World!" and assigns it to the variable *saying*. The string is then displayed using the **print** function.

```
>>> saying = "Hello World!"
>>> print(saying)
Hello World!
```

In a similar manner, you can create a Python **list** by enclosing a comma-separated sequence of objects within square brackets, [ ]. For example, the first line below creates a **list** containing three strings and assigns it to the variable *file_list*. The second line prints the elements in the **list** using the **print** function.

```
>>> file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
>>> print(file_list)
['smith_gcd.py', 'cheng_gcd.py', 'garside_gcd.py']
```

Why don't we need a loop to print all the items in a list? If you're coming from a language like C, you'd likely expect to use a loop to print the items in a **list**. In Python, a **list** is a built-in data type, so functions like **print** are designed to work with them, much like the C function **printf** knows how to print a character string without needing loop.

We will make use of **lists** in some of our examples in this lecture.

The len function

The Python built-in function, **len**, returns the length of an object's data. In the case of a string, the **len** function returns the number of characters in the string. For example,

```
>>> len("Hello World!")
12
>>> len('A')
1
>>> len("")
0
```

When working with strings, each character of the string is referenced by its numeric position, called an **index** or **subscript**. In Python, the character in the first position (from the left) is given index 0 and each subsequent character is given the next numeric index. The index of the last character of the string is, effectively, the length of the string minus 1.

For example, the string “Hello World!” has 12 characters (including a space and the exclamation point) and the indices of the characters range from 0 to 11.

H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

The Subscript Operator

A string is a sequence of characters. As such, it is commonly desirable to access individual characters of the string, or to access a substring of the string. To do this in Python, we use the **subscript operator**.

Accessing Characters in a String

The **subscript operator** allows you to access one or more characters in a string using the string **indices** (or **subscripts**).

The simplest form of the **subscript operator** has the following syntax where <*string*> is the string whose characters you want to access and <*integer expression*> is any integer expression that produces an integer **index** (**subscript**) in the range of the string.

<*string*>**[**<*integer expression*>**]**

For example,

```
>>> saying = "Hello World!"    # Create a string

>>> saying[0]      # return the first char (index 0)
'H'

>>> saying[6]      # return the 7th char (index 6)
'W'

>>> saying[len(saying)]    # len(saying) = 12 so returns an error
Traceback (most recent call last):
  File "<pyshell#117>", line 1, in <module>
    saying[len(saying)]
IndexError: string index out of range
>>> saying[len(saying) - 1]   # len(saying) – 1 is the last char
'!'
```

As a shorthand notation, you can provide negative subscripts to access characters starting from the right-hand side of a string.

```
>>> saying[-1]    # return the last char in the string
'!'
>>> saying[-2]    # return the second to last char in the string
'd'
```

There are times when a program might make use of both the string indices and the values found at those indices. For example, the following count-controlled loop uses the **range** built-in to generate a list of **indices** from 0 to the length of the string "Hello World!" minus 1. The print **statement** in the body of the **for** loop prints the index, a space, and the character found at that index in the string.

```
saying = "Hello World!"
for index in range(len(saying)):
    print(index, saying[index])
```

```
0 H
1 e
2 l
3 l
4 o
5  
6 W
7 o
8 r
9 l
10 d
11 !
```

Accessing Substrings within a String

When working with strings, it is often useful to be able to obtain a subsection of the string, also known as a **substring**. For example, given a string containing a file name of the form *<user>_<program>.py* (for example, "garside_gcd.py"), it might be useful to extract the user name, or the program name.

You can use Python's **subscript** operator to obtain any substring through a process called **slicing** (the term **slicing** reflects the idea of obtaining a **slice** of the string). To obtain a **slice** (**substring**) using the **subscript** operator, you insert a **colon** within the square brackets along with a ***start*** index and an ***end*** index. The ***start*** index is the starting index of the substring you want to obtain, and the ***end*** index is the index of the end of the substring **plus** 1.

The general syntax is as follows,

<string>[*start*:*end*]

If you omit the ***start*** index, the beginning of the string is assumed (index 0).
If you omit the ***end*** index, the end of the string is assumed (last index + 1)

Below are some examples of obtaining substrings. The string being used, "**garside_gcd.py**", is 14 characters long. The first character, '**g**', is at index 0. The last character, '**y**', is at index 13.

```
>>> name = "garside_gcd.py"

>>> name[:]      # start and end omitted. Return the whole string
'garside_gcd.py'

>>> name[0:1]    # start at index 0, end at index 0
'g'

>>> name[0:7]   # start at index 0, end at index 6
'garside'

>>> name[:7]   # start omitted. Start at index 0, end at index 6.
'garside'
```

The following three examples all return the same substring.

This example explicitly extracts the substring starting at index 8 and ending at index 13 (14 – 1).

```
>>> name[8:14]  # return substring from index 8 to 13 (14 - 1)
'gcd.py'
```

This example uses the **len** function to calculate the end of the string.

```
>>> name[8:len(name)]  # return substring from index 8 to length
'gcd.py'
```

This final example omits the *end* subscript, which defaults to the end of the string.

```
>>> name[8:]    # return substring from index 8 to end of string
'gcd.py'
```

Negative integers may be used to indicate subscripts starting from the right-hand side of the string. For example, we can obtain the filename extension from the string as follows,

```
>>> name[-3:]   # return substring from index 12 to end of line
'.py'
```

As a larger example, consider the following code. The first line creates a **list** called *file_list* containing three file names.

The **for** loop header on the second line iterates over all the entries in *file_list* and assigns each file name in the **list** to the loop variable *file_name,* a different one on each pass.

The body of the **for** loop consists of a single **print** statement that prints from the beginning of the file_name string to the length of the file_name string minus 7.

The resulting output is the list of user names contained within the file_name strings, as shown below.

```
file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
for file_name in file_list:
    print(file_name[:len(file_name) - 7])

smith
cheng
garside
```

The “`in`” Operator – Testing for a Substring within a String

Sometimes it is useful simply to know whether a substring exists within a string. The Python **`in`** operator, which we have used with **for** loops in earlier lectures, can also be used to determine if a substring exists in a string.

When using the **`in`** operator this way, the general format is

<*substring*> **`in`** <*string*>

where the operator returns **True** if <*substring*> exists in <*string*>. Otherwise, it returns **False**.

The example below demonstrates the use of the **`in`** operator to test for the existence of the substring "_gcd.py" in a **list** of strings.

The first line creates a **list** of file names and assigns the list to the variable *file_list*. Notice that only two of the file names contain the substring “_gcd.py”.

The **for** loop header in the second line loops through the strings in *file_list* assigning them to the variable *file_name*, a different one on each pass.

The **if** on the third line employs the **`in`** operator to test if the string “_gcd.py” exists in the string held by *file_name*. When the test is **True**, the **print** statement in the body of the **if** prints the file name.

As seen below, the output consists of only the file names that contain the substring “_gcd.py”

```python
file_list = ["smith_gcd.py", "cheng_circle.py", "garside_gcd.py"]
for file_name in file_list:
    if "_gcd.py" in file_name:
        print(file_name)
```

```
smith_gcd.py
garside_gcd.py
```

Since a string is a sequence of characters, the characters can also be accessed with a simple **for** loop. For example, the following **for** loop iterates through the characters of the string **"Hello"**, assigning successive letters of the string to the loop variable *char*. The body of the **for** loop consists of a single **print** statement that prints the character, *char*, followed by a newline.

```
>>> for char in "Hello!":
        print(char)

H
e
l
l
o
!
```

String Methods

Python is an **object oriented** language. We will learn more about what this means in future lectures but for now, suffice it to say that an object oriented language is designed around the concept of creating objects (for example, a string object) and a set of **methods** to operate on those objects (for example, a method to convert the characters of a string to upper-case). You can think of a **method** somewhat like you think of a **function**, but there are important differences.

One difference between a function and a method is that a **method** is defined for a particular kind of object and a **method** has access to the **data** of that object without needing to pass the data as an argument.

A second difference is that you call a function by giving the function's name along with any function arguments enclosed in parenthesis. To call a **method**, you give the name of the object, a **dot** (period), and then the name of the **method** followed by any arguments enclosed in parenthesis. The general syntax is,

<object>.<method-name>(<argument1>,<argument2>,…)

For example, a **string object** in Python has a **method** called **upper**() which returns a copy of the string object with the data (characters) converted to upper case. Since the **upper method** takes no arguments, you can call it by entering the name of the string object, a **dot**, and then the text **upper()**, as shown below.

```
>>> s = "Hello World!"
>>> s.upper()
'HELLO WORLD!'
```

A **literal string** is also a string object and you can call the same **methods** on **literal** strings as you would on a string variable. The only difference is that a **literal** string doesn't have a variable name assigned to it.

```
>>> "Hello World!".upper()
'HELLO WORLD!'
```

Below is a table containing a number of Python's string **methods** and their descriptions.

Remember, in Python, it's not possible to modify a string because Python strings are immutable. Once they are created, they can be read or deleted, but they can't be changed. Because of this, many of the methods below return a modified copy of the original string.

String Method	Description
s.center(width)	Returns a copy of **s** centered within a column **width** wide
s.count(sub *[*, **start** *[*, **end***]]***)**	Returns the number of non-overlapping occurrences of substring **sub** in **s**. Use optional arguments **start** and **end** to limit the search within **s**. The indices **start** and **end** are interpreted as they are with the **subscript operator**.
s.startswith(sub)	Returns **True** if **s** starts with sub. **False** otherwise.
s.endswith(sub)	Returns **True** if **s** ends with sub. **False** otherwise.
s.lower()	Returns a copy of **s** converted to **lowercase**.
s.upper()	Returns a copy of **s** converted to **uppercase**.
s.isalpha()	Returns **True** if **s** contains only **letters**. **False** otherwise. (Note: spaces and punctuation are not considered letters.)
s.isdigit()	Returns **True** if **s** contains only digits. **False** otherwise.
s.find(sub *[*, **start** *[*, **end***]]***)**	Returns the lowest index in **s** where substring **sub** is found. Use optional arguments **start** and **end** to limit the search within **s**. The indices **start** and **end** are interpreted as they are with the **subscript operator**.
s.replace(old, new *[*, **count***]***)**	Returns a copy of **s** with all occurrences of substring **old** replaced by **new**. If optional argument **count** is given, replace only the first **count** occurrences.

s.join(*string sequence*)	Returns a string that is the concatenation of the strings in ***string sequence*** separated by the string **s**. (Note: A Python **list** containing strings is a common string sequence.)
s.split(*[separator string]*)	Returns a Python **list** containing the words in **s**. By default, **whitespace** (tabs, spaces, newlines) is used to separate the words in **s**. Use the optional ***separator string*** to choose what is used to separate the words.
s.strip(*[strip-string]*)	Returns a copy of **s** with leading and trailing whitespace (tabs, spaces, newlines) removed. Use optional argument **strip-string** to remove leading and trailing **strip-string**, instead.

Below are some examples using the **methods** described in the table above along with details of their execution. The output resulting from the statements is shown immediately below each example.

```
s = "To be, or not to be, that is the question."

# Display the string, s, centered within a column of 80 characters.
print(s.center(80))
                   To be, or not to be, that is the question.

# Display the number of times the string “be” occurs in string s.
print(s.count("be"))
2

# Same as above but consider only the text between the two commas,
# inclusive. (Remember, indices start at 0, so the first comma is at
# index 5 and the second is at index 19.)
print(s.count("be", 5, 20))
1

# Returns False (because there is a period at the end of string s).
print(s.endswith("question"))
False
```

```
# Returns True because string s ends with "question."
print(s.endswith("question."))
True

# Returns True because string s starts with "To be"
print(s.startswith("To be"))
True

# Returns a copy of string s converted to lowercase.
print(s.lower())
to be, or not to be, that is the question.
```

Note: there is a string method called **casefold()** which can be used similarly to **lower()**. The difference is that **casefold()** also works with many non-Latin (e.g. non-English) Unicode characters. However, you will see **lower()** used most frequently when working with languages such as English.

```
# Returns a copy of string s converted to uppercase.
print(s.upper())
TO BE, OR NOT TO BE, THAT IS THE QUESTION.

# Returns False since s contains characters other than letters.
# (It includes spaces and punctuation)
print(s.isalpha())
False

# Returns False since s has characters other than digits.
print(s.isdigit())
False

# Returns 14 since the only instance of "to be" starts at index 14.
# Note: Python strings are case sensitive, so "To be" and "to be"
# are different strings.
print(s.find("to be"))
14

# Returns 3 since "be" first occurs at index 3 in string s.
print(s.find("be"))
3
```

```
# Returns 17 since find is asked to locate the string after index 5
# and the first occurrence of “be” after index 5 is at index 17.
print(s.find("be",5))
17

# Replaces all instances of the string “be” with “go”.  (Note that if
# the word “before” were in the string, it would be changed to
# “gofore”, so you need to be careful.)
print(s.replace("be","go"))
To go, or not to go, that is the question.

# The join method takes a sequence as an argument. In this case, the
# argument is the string s, which contains a sequence of characters.
# Below, join effectively joins each item of the sequence with the
# string “-“. The result is a new string where a hyphen “-“
# separates each character of the string s.
hyphen_separated_string = "-".join(s)
print(hyphen_separated_string)
T-o- -b-e-,- -o-r- -n-o-t- -t-o- -b-e-,- -t-h-a-t- -i-s- -t-h-e- -q-u-e-s-t-i-o-n-.
```

```
# The following is perhaps a more typical use of join.
# When the sequence passed to join is a list of strings, each item in
# the sequence (each string) is separated by the joining string. In
# this case, a new string is created containing each item in the list
# sequence (each string) separated by “ – “.
print(" - ".join([s,"William Shakespeare"]))
To be, or not to be, that is the question. - William Shakespeare
```

Note: For performance reasons, it’s better to put strings together with **join** rather than the **+** operator. Because Python strings are immutable (i.e. they cannot be changed), appending each string with **+** is inefficient because the entire string must be copied on each append. It is much more efficient to accumulate the strings into a **list** (which is mutable) and then **join** the strings together.

```
# Returns a list of strings. Each string is a substring of s obtained
# by splitting the string s at every comma, “,”.
print(s.split(","))
['To be', ' or not to be', ' that is the question.']
```

```
# By default, the split method splits a string into substrings on
# whitespace boundaries. In this example, the result is a list where
# each item is a word in the string, including any punctuation.
print(s.split())
['To', 'be,', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question.']

# If each item in string s is separated by a single space, splitting
# the string and rejoining it with a space will create a copy of the
# original string.
#
# Below, as always, items in parenthesis are evaluated first. String s
# is first split into a list containing the words in the string (as in
# the example above). The list is then passed to join and the
# substrings (words) in the list are each joined by a single space.
print(" ".join(s.split()))
To be, or not to be, that is the question.
```

Now that we've discussed some string methods, we can generalize one of the examples we created earlier. Recall the program below which extracts usernames from a list of filenames.

```
file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
for file_name in file_list:
    print(file_name[:len(file_name) - 7])

smith
cheng
garside
```

The problem with this program is that it will produce incorrect results if the character sequence following the "_" is lengthened or reduced. For example, the output would be more than just the user names if the program were given the following definition of *file_list*,

```
file_list = ["smith_circle.py", "cheng_circle.py", "garside_circle.py"]

smith_ci
cheng_ci
garside_ci
```

We can make the program more general by splitting each of the strings within the list at the underscore, “_“, and just printing the first element in the resulting list.

The first two lines in the program below are the same as before. It loops through each string in *file_list* and assigns it to the variable *file_name*.

The third line takes the current *file_name* and splits it into elements using an underscore, “_“, as the separator, and assigns the result to a variable called *items*. For example, the first filename, **“smith_gcd.py”**, would be split into a list that looks as follows,

['smith', 'gcd.py']

The last line of the program then prints the first element of the list.

Note: In Python, you access **list** elements using the **subscript** operator, **[]**, just as you would access the characters of a string. For example, the first element in a **list** is referenced as **[0]**. The **last** element in a **list** is referenced as **[-1]**.

```
file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
for file_name in file_list:
    items = file_name.split("_")
    print(items[0])
```

```
smith
cheng
garside
```

In the next example, we modify the program to consolidate the last two lines to condense the source code.

The **print** statement in this example first calls the ***split(“_“)*** method on the string contained in *file_name*, creating a two element **list**. The first element in the list, the user name, is then obtained using the **subscript** operator **[0]** and the resulting string is passed to the **print** function as an argument.

Question: How do we know that the **split(“_“)** method is called to create a list **before** the subscript operator **[0]** tries to access the list? We know because a **method** call has higher preference than other operators, including the **subscript operator**.

```
file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
for file_name in file_list:
    print(file_name.split("_")[0])
```

```
smith
cheng
garside
```

Text Files

All of the programs we have seen to this point gather their input directly from the user via a keyboard and display the output to the screen. While this is common practice, entering input via a keyboard limits the amount of data that can be entered and how quickly it can be entered. In a similar manner, writing to a screen, while good for immediate output, is limited by how quickly the user can process the output at interactive rates. Also, the fact that the output is not permanent means the data can't be view and/or processed at a later time.

For many applications, it is desirable to write the results of program computations into a file or to read information that has been previously stored from a file. We cover these process in this section.

Writing Text to a File

As we've seen repeatedly, most things in Python create and use objects. This is also true when working with files.

Opening a File for Writing

To write to a file, you can open the file with the built-in function **open**. The **open** function takes a *filename* as an argument along with a **mode string** indicating how the file should be opened. When opening the file for writing, the **mode string** is '**r**'.

The **open** function opens a connection to the file on disk, and returns a **file object**. If the file does **not** exist, it is created with the given *filename*. If the file already exists, Python opens it.

The general syntax of the **open** statement is,

file_object = open(*<filename>*,*<mode>*)

For example, the following statement opens a file named ***myfile.txt*** for writing.

```
f = open("myfile.txt", 'w')
```

Where is “myfile.txt”? When **open** opens a file, it does so in your **current directory/folder**. The current directory will be the location from which you’re running your Python script. If you wish to open a file in a different location, you can give **open** the full **pathname** to the filename. For example, the following statement opens up the file **myfile.txt** for writing in the folder C:\Users\Garside\Documents.

```
f = open("C:\\Users\\Garside\\Documents\\myfile.txt", 'w')
```

Notice that the backslashes, “\”, in the filename path are escaped with a preceding backslash so they are not interpreted as special characters.

Writing verses appending: When a file is opened for writing using **mode** ‘**w**’, any existing data in the file is **lost!** Please be sure to make a backup copy of any file you intend to write to if you want to retain the previous contents. If you want to **append** to an existing file, you can **open** the file with the **mode** ‘**a**’

Writing to an Opened File

Once a file is opened for writing, you can write data to the file using the file object’s **write** method. Like other methods, you call the **write** method by specifying the file object, a **dot** (**.**), and then the name of the method.

The file object’s **write** method takes a **single** string as an argument. The string is written to the file exactly as specified. If you want the string to end with a **newline**, you must include the **newline**. If you want to write multiple lines to the file, you can write one long string containing multiple newlines.

The following statement writes two lines to the file ***myfile.txt*** opened earlier. Notice that the write method is called from the file object, **f**. The method’s argument is a single string containing two **newlines**, **\n**.

```
f.write("Line one.\nLine two.\n")
```

The lines written to the file look as follows. If the file previously existed, any previously existing data is lost.

```
Line one.
Line two.
```

Note: **You can only write strings**. When writing to a file using the **write** method, all data must be in string format. If you have numeric data, like integers or floats, you must first convert them to strings before writing them.

Closing a file

When you're done writing to a file you should close it. Technically, when your program finishes it will close any open files, however, there are times when a program does more processing after a file has been written, and it's best to close the file as soon as you're done writing to avoid any accidental changes or potential loss of data should the system unexpectedly crash.

You use the **close** method to close a file opened with the **open** function. The **close** method takes no arguments.

The following statement closes the file we've been working with,

```
f.close()
```

Putting it all together

The example below combines our previous program for extracting usernames from filenames with code to write the results to a file named "usernames.txt".

```
f = open("usernames.txt", 'w')
file_list = ["smith_gcd.py", "cheng_gcd.py", "garside_gcd.py"]
for file_name in file_list:
    f.write(file_name.split("_")[0] + "\n")
f.close()
```

Notice that the **print** statement has been replaced with a call to the **write** method. Also notice that a **newline**, **\n**, has been explicitly concatenated to each extracted username. Unlike the **print** function, the **write** method does not automatically add a **newline** to the data it writes to the file.

Writing Numbers to a File

The argument to the **write** function must be a string. **write** does not support numeric data types, like **int** and **float**. The built-in conversion function, **str**, will convert most data types to a string. You can then write the resulting value to a file using white-space or a newline as a separator.

The following code contains a **for** loop that writes 500 random numbers to the file *integers.txt*. The program imports the **random** module and makes use of the **randint** method to generate the pseudo-random numbers. The **str** function is used to convert each integer value to a string and the **+** operator then appends a newline. The converted string is passed to the **write** method and the value is written to the file *integers.txt*.

```
import random

f = open("integers.txt", 'w')
for count in range(500):
    number = random.randint(1, 500)
    f.write(str(number) + '\n')
f.close()
```

Reading Text from a File

Reading text from a file is similar to writing text to a file, with a few notable differences.

Opening a file for Reading

You open a file for reading using the **mode string** ‘r’, instead of the mode string ‘w’ used for writing. For example,

```
f = open("myfile.txt", 'r')
```

If the file doesn’t exist, an error is raised.

Reading the whole file with one call

There are several ways to read in a file's contents after it is opened. The simplest and fastest way is with the file object's **read** method. The **read** method reads the entire contents of the file into a single string. If the file contains multiple lines of text, the newline characters at the end of each line will be embedded in the string. For example, the following line will read the contents of a file opened up by the **open** command and put all the contents into a string named *text*, newlines and all.

```
text = f.read()
```

If you were to call the **read** method again, it would return an empty string, indicating the end of file has been reached.

If you want to reread the file with **read**, you need reopen the file by issuing the **open** call again to "rewind" the file for another input process.

Reading the file line by line

There are times when it's more convenient to read a file line by line than as one single string. Since a file object is an **iterator** (that is, Python can iterate over the content of the object), a very succinct way to do this is with a **for** loop and the **in** operator.

The code below opens the file *myfile.txt* for reading. The **for** loop then reads in the content of the file one line at a time, each time assigning the text of the line to the variable *line*. The **print** statement then prints the line.

Notice that the **print** statement includes the optional argument **end=""**. We have seen this before. It is used to tell the **print** statement **not** to add a **newline** to the end of the text that it prints. We do this here because the line of text that was read from the file **already** contains a **newline**. If we don't do this, a second newline will be printed and there will be a blank line printed between each line of the output, effectively making the output double spaced.

```
f = open("myfile.txt", 'r')
for line in f:
    print(line, end="")
```

If you want to read the file line by line but, possibly, not read all of the file, you can used the file object's method, **readline**(). The **readline** method consumes a line of input from the file and returns it as a string, including the **newline**. When **readline** encounters the end of the file it returns an empty string.

The code below does the same thing as the previous **for** loop but, in this case, we use the **readline** method and a **while True** loop to iterate over every line in the file.

The first line in the loop body reads a line in the file. The **if** statement then checks if the line is empty (signaling the end of the file) and, if so, the **break** statement breaks out of the loop, effectively terminating the loop.

If the line is not empty, the **print** statement prints the line. Notice, once again, that the **print** statement includes **end=””** so as to not add an additional newline.

```
f = open("myfile.txt", 'r')
while True:
    line = f.readline()
    if line == "":
        break
    print(line, end="")
```

When you **open** a file for **reading**, you do not need to explicitly **close** the file with the **close**() method. You can do so, if you wish, but Python will close the file automatically when the program ends and, since you’re not modifying the file (because you’re only reading from it), there’s no danger of corruption if the system crashes before the program ends.

Reading numbers from a File

When a file is opened for writing using **open**, the objects written must be strings. In a similar manner, when a file is opened for reading using **open**, only strings can be read from the file.

If the strings that are read from a file are intended to be something else, like an integer of floating-point number, the strings must be converted on input to match the desired data type.

Consider the file *integers.txt* that we created earlier. The contents of the file is a set of random integers, converted to strings. Let’s say we want to **open** and read the file *integers.txt*, and sum up the integers.

The code below opens the file *integers.txt* for reading, ‘r’, and initializes the *total* sum of the integers to 0.

The **for** loop reads in each line of the file, assigning the text read to the string variable *line* (*line* is created as a string variable because it is being assigned the string read from file object **f**). The first line of the loop body strips any leading or trailing whitespace from the string using the method **.strip()** and the **int** function converts the string to an integer. The result is assigned to the variable *number*.

The last line in the loop body adds the value of *number* to the variable *total*, thus creating a running total of the sum of the numbers.

The final **print** statement, which is outside the loop body, prints the final *total*.

```
f = open("integers.txt", 'r')
total = 0
for line in f:
    number = int(line.strip())
    total += number
print("The sum is", total)
```

```
The sum is 123892
```

Note: In the code above, if you mistakenly indent the last print statement to the level of the loop body, it will be printed each time through the loop, effectively printing a running total. This is a case where incorrect indentation doesn't produce an error but, rather, produces an unexpected result.

The table below summarizes the file operations we have discussed and their meanings. Note that all the operations are methods except for **open**, which is a function. **open** is used to return a file object and then methods are used to operate on the file object. In the table, it is assumed that the file object is named 'f'.

Method	Meaning
Open(filename, mode)	Opens filename and returns a file object. mode can be one of 'r' = read 'w' = write 'rw' = both read and write 'a' = append

f.close()	Closes a file. Not needed when a file is opened only for reading, ‘r’.
f.write(string)	Writes a string to the file. You must include a newline if you don’t want all the lines to run together.
f.read()	Reads the entire file and returns it as a single string. Returns the empty string, “”, when the end of file is reached.
f.readline()	Reads one line of the file, including the newline, and returns it as a string. Returns the empty string, “”, when the end of file is reached.

Accessing and Manipulating Files and Directories (Folders) on Disk

When working with files it is often useful to be able to manipulate their names and locations from within a Python program. It may also be necessary to create a new directory (folder) to hold a file, or to rename a file or directory. In addition, is important to perform error checking when working with files, for example, to check if a file exists before attempting to read from it. Python includes a couple of modules, which you can import, that help you with these tasks.

The **os** module includes a wide variety of functions for working with files and directories, including functions to change, list, create and remove directories, and functions to list, remove and rename files.

The following table contains some of the functions in the **os** module and their meanings.

os Module Function	Meaning
chdir(**path**)	Changes the current working directory to **path**.
getcwd()	Returns the path of the current working directory
listdir(**path**)	Returns a Python **list** of the names in the directory named **path**.
mkdir(**path**)	Creates a new **directory** named **path** and places it in the current working directory.

remove(**path**)	Removes (deletes) the **file** named **path** from the current working directory.
rename(**old**, **new**)	Renames the file or **directory** from **old** to **new**.
rmdir(**path**)	Removes (deletes) the **directory** named **path** from the current working directory.
sep	**sep** is a variable that holds the separator character ('/' or '\') of the current file system.

The following example imports the os module and uses the function **listdir** to read the names of the entries in the directory *C:\Users\Garside\Documents* and print them one line at a time.

```
import os

file_list = os.listdir("C:\\Users\\Garside\\Documents")
for file in file_list:
    print(file)
```

Notice that the **path** passed to **listdir** has an additional backslash, '\', in front of each backslash in the **path** so that the following characters are not treaded as escape sequences.

Note: **os.listdir** looks exactly like a method call, but it is not. The reason is that **os** is not an object, it's the name of a module, so the **dot** notation in this case is simply used to identify the function to be called within the module. So remember, you will see the same **dot** notation used for two things (1) as a method call when the first item is an object (like a string object, as in **"APPLE".lower()**) and (2) as a function call when the first item is a module name.

The module **os.path** includes additional useful functions for operating on files and directories. Some of the more commonly used functions are listed below along with their meanings.

os.path Module Function	Meaning
exists(**path**)	Returns **True** if **path** exists. **False** otherwise.

isdir(**path**)	Returns **True** if **path** is a **directory** (**folder**). False otherwise.
isfile(**path**)	Returns **True** if **path** is a **file**. False otherwise.
getsize(**path**)	Returns the **size** of the object named by **path** in **bytes**.
normcase(**path**)	Converts **path** to a pathname appropriate for the current file system. For example, converts '/' to '\' and letters to lowercase on a Windows system.

Conclusion

In this lecture we've looked in-depth at **strings**, from their structure to their **methods**, including accessing **substrings** and indexing **character ranges** with the **subscript operator**. We have also looked at how **text files** are manipulated using Python, including **reading** and **writing** text files, and working with both **string and numeric data types**. Further, we've presented a number of new functions for **manipulating files and directories on disk**. Given the prevalence of text-based processing in information systems today, these capabilities add greatly to what we can do using Python.