Data Preparation in Python

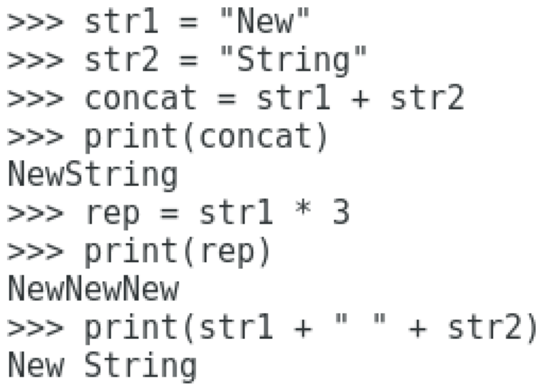
Python offers an ideal environment for processing text files that contain well (and not so well) formatted data. That is, it has many useful built in String functions, an extremely flexible List data structure, and its file I/O functions are quite powerful. To perform file I/O on data, three things are needed: facilities for manipulating strings, a data structure for storing or retrieving data, and a file to read from or write to. You may choose to use either interactive or command line Python for the String Manipulation and Introduction to Lists sections, but you must use write a Python script for the File I/O section.

String Manipulation

Previously, you learned how to access individual elements of Strings using the [] operator and specifying either an index ([i]) or range of indices ([a:b]), respectively. Python has many useful built in String functions in addition to these that make text processing much easier than in other languages.

Concatenation and Repetition

Strings may be concatenated (literally added together) using the (+) operator or repeated with the repetition operator (*).



```
>>> str1 = "New"
>>> str2 = "String"
>>> concat = str1 + str2
>>> print(concat)
NewString
>>> rep = str1 * 3
>>> print(rep)
NewNewNew
>>> print(str1 + " " + str2)
New String
```

The most common cases of “string building” through concatenation are in creating more complex strings, such as forming file names to access, or in print statements, as shown above.

String Conversion Functions

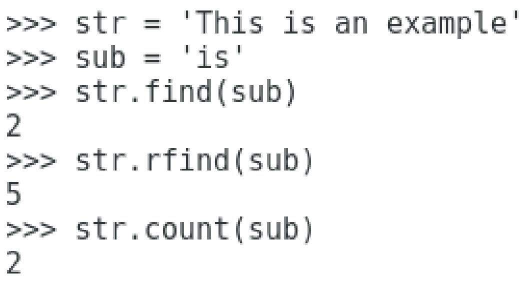
Python offers several built-in functions for changing the case of letters in a String object. To call any of them, simply use the format **str.function()** where str is a variable assigned to a String and function is one of the functions in the list below.

- **capitalize() – capitalizes the first character in a String (if it’s a letter character) and converts any remaining letter characters to lowercase.**
- **lower() – converts all letter characters to lowercase.**
- **upper() – converts all letter characters to uppercase.**
- **swapcase() – changes the case of all letter characters to their opposing case (lower to upper and upper to lower).**

String Find Functions

To find the existence of a substring within a string and to find out how many times a substring occurs within a string, the following functions may be used. The format is **str.function(sub, start, end)**, where sub is a substring being searched for, start is the starting index to start searching at and end is the last index to search at. Both start and end are optional parameters. You may choose to specify just a start index or both a start and end index.

- **find – returns the starting index of the first occurrence of the substring within this String.**
- **rfind – returns the starting index of the last occurrence of the substring within this String.**
- **count – returns the number of occurrences of the substring within this String.**



```
>>> str = 'This is an example'
>>> sub = 'is'
>>> str.find(sub)
2
>>> str.rfind(sub)
5
>>> str.count(sub)
2
```

The substring ‘is’ occurs twice within the String str, once at Index 2 and another time at Index 5.

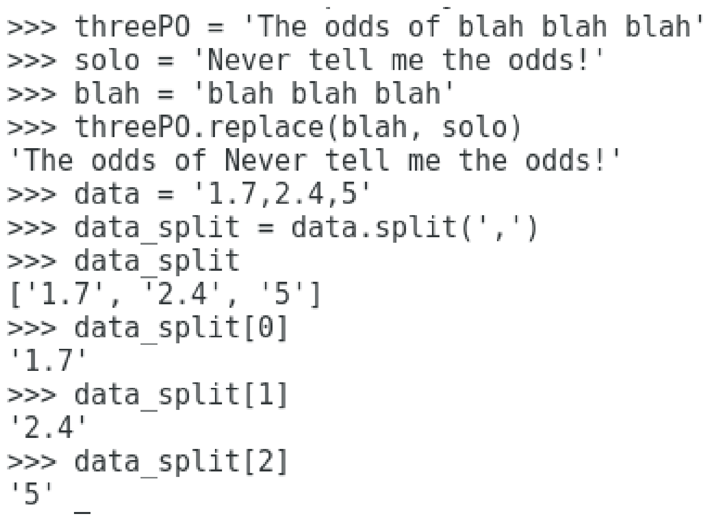
Practice Exercise

Add a start (and end?) so that the **find** function only finds the second occurrence of ‘is’ in the String. What do find, rfind, and count return if the substring does not exist in the String?

String Replacement

These functions include actual string replacement along with performing splits of strings using a delimiter. Split will become especially important in reading data from files. The format of these functions is **str.function()** where each function has their own unique arguments, as shown below.

- **replace(old, new) –** replace all occurrences of the substring old in String str with the substring new.
- **split(delimeter) –** returns a List of Strings after splitting the String str using the delimeter.



```
>>> threePO = 'The odds of blah blah blah'
>>> solo = 'Never tell me the odds!'
>>> blah = 'blah blah blah'
>>> threePO.replace(blah, solo)
'The odds of Never tell me the odds!'
>>> data = '1.7,2.4,5'
>>> data_split = data.split(',')
>>> data_split
['1.7', '2.4', '5']
>>> data_split[0]
'1.7'
>>> data_split[1]
'2.4'
>>> data_split[2]
'5' _
```

In the call to the replace function, we replaced the occurrence of the substring “blah” with the string “solo”. In the call to split, the data is split into a list of strings using the commas to determine where each resultant String should begin. Notice that even though we have numeric data separated by commas in the String data, the result of splitting is a List of Strings, not floats and/or integers.

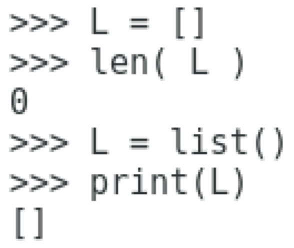
However, as you’ll see in the next section, elements of a List can be of any type, even other Lists!

Introduction to Lists

A Python List is a collection of objects of any type. A List may contain objects of one or more type (integers, floats, Strings, Lists). Lists are indexed beginning at element 0 and each element may be accessed using the [i] operator for a single index or [a:b] for a range of indices. Their contents may be changed at any time through many different ways. This section shows a non-comprehensive guide to using Python Lists in the context of data processing.

List Creation

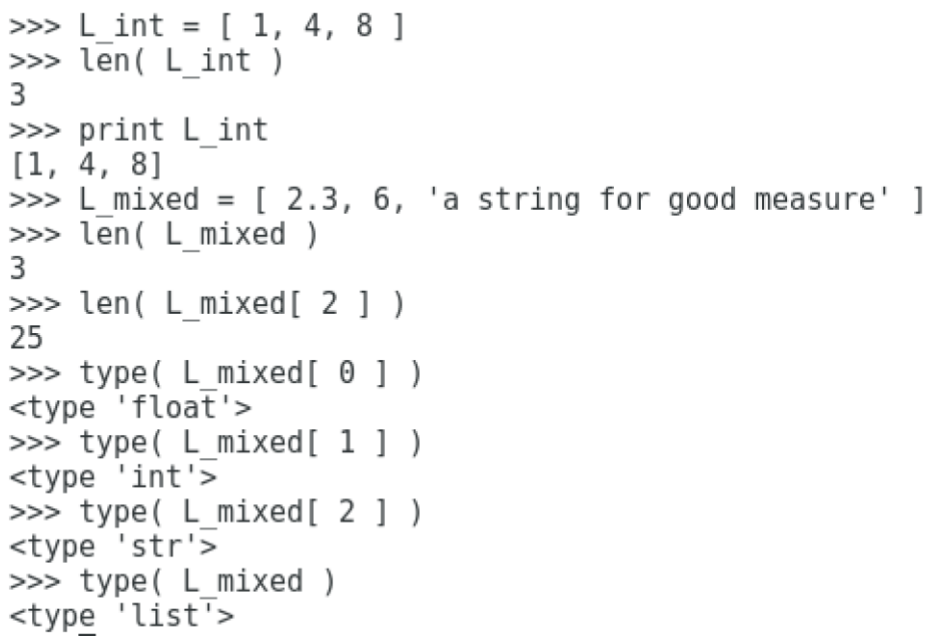
An empty List may be created using L = [] or L = list().



```
>>> L = []
>>> len( L )
0
>>> L = list()
>>> print(L)
[]
```

The contents of a List can be output to the console using print and as with any collection in Python, len outputs the length (number of items) of the collection.

A list can also be initialized with elements specified. The elements may be of one type or a mix of types.

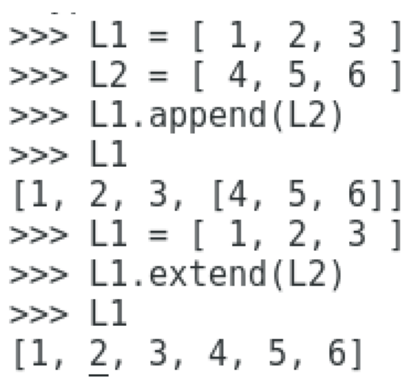


```
>>> L_int = [ 1, 4, 8 ]
>>> len( L_int )
3
>>> print L_int
[1, 4, 8]
>>> L_mixed = [ 2.3, 6, 'a string for good measure' ]
>>> len( L_mixed )
3
>>> len( L_mixed[ 2 ] )
25
>>> type( L_mixed[ 0 ] )
<type 'float'>
>>> type( L_mixed[ 1 ] )
<type 'int'>
>>> type( L_mixed[ 2 ] )
<type 'str'>
>>> type( L_mixed )
<type 'list'>
```

List L_int contains integers 1, 4, and 8 and has a length of 3 elements, while L_mixed has a float, int, and String for a total of 3 elements. To determine the length of the String element of L_mixed, the len function is called with Element 2 of L_mixed as an argument. We are able to confirm the individual types of each element in L_mixed as well as show that L_mixed is, in fact, a List.

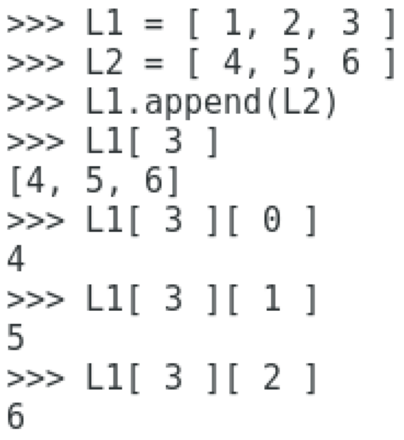
List Extension – Append and Extend

Lists may be extended in many ways, two of which are with the **append** and **extend** functions. The append function adds an element to the end of the List, while extend concatenates two Lists together. There are some subtle differences between append and extend that we will bring out by example. The append function is going to be used extensively when reading data from a file into a List.



```
>>> L1 = [ 1, 2, 3 ]
>>> L2 = [ 4, 5, 6 ]
>>> L1.append(L2)
>>> L1
[1, 2, 3, [4, 5, 6]]
>>> L1 = [ 1, 2, 3 ]
>>> L1.extend(L2)
>>> L1
[1, 2, 3, 4, 5, 6]
```

When L2 is “appended” to L1, the result is that List L1 now has 4 elements. The final element is a List with the integers 4, 5, and 6 in it. However, when L1 is “extended” by L2, the elements of L2 are added *individually* to L1 so that List L1 now has 6 elements total. There are certainly use cases for both methods of extending a List.

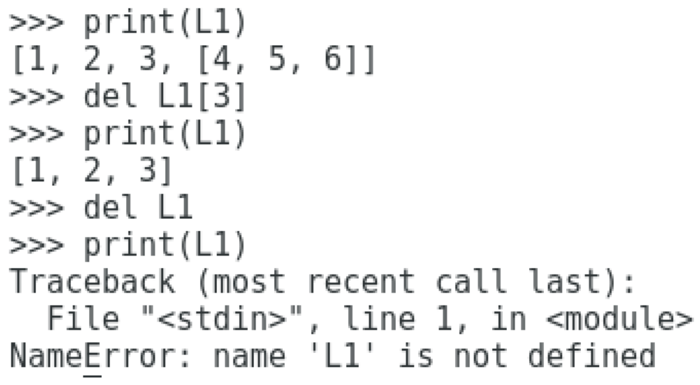


```
>>> L1 = [ 1, 2, 3 ]
>>> L2 = [ 4, 5, 6 ]
>>> L1.append(L2)
>>> L1[ 3 ]
[4, 5, 6]
>>> L1[ 3 ][ 0 ]
4
>>> L1[ 3 ][ 1 ]
5
>>> L1[ 3 ][ 2 ]
6
```

Looking deeper into the append function, we come back to L2 appended to L1. L1 contains 1 at Element 0, 2 at Element 1, 3 at Element 2 and the List [4, 5, 6] at Element 3. Essentially, Element 3 is "two dimensional" while the remaining elements of List L1 are singular integer objects. To access the individual elements of the List stored in Element 3, the [i][j] notation is needed, where i is the element of L1 and j is the element of the List at index L1[i]. We can now access the individual elements of the List at Element 3 using [3][0], [3][1], and [3][2].

Removing List Elements

The **del** function is used to remove either individual elements from a List or deleting the entire List object. To remove individual elements of a List, the format "**del L[i]**" is used, while deleting the entire List only requires "**del L**".



```
>>> print(L1)
[1, 2, 3, [4, 5, 6]]
>>> del L1[3]
>>> print(L1)
[1, 2, 3]
>>> del L1
>>> print(L1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'L1' is not defined
```

File I/O

Python offers simple, yet powerful, file I/O. The general steps for performing file I/O are as follows:

- Create a file handle by opening a file in some mode (read, write, append)
- Use the file handle to read from or write to the file
- Close the file handle

Opening a File

Files are opened using <gasp> the **open** function! The open function format is:

f = open(filename, mode)

Where filename is a String with the full path to the file and mode is read (r), write(w), or append (a). There are many other modes, including binary (b) and read/write (+). Technically, mode is optional (read is the default mode if a mode is not specified, but we prefer that you specify a mode every time you open a file.

For this section, we'll be using iris.txt, which is in the Python section of the Content area on myCourses. If you would like to view this file in a text editor in Windows, Notepad++ is recommended. The file contents are:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
```

Each record (line) of the file contains four floats and a String that identifies which type of iris flower each record is. For ease of use, place iris.txt in a convenient location since you will need to access it with your Python scripts.

```
>>> f = open('iris.txt', 'r')
>>> print(f)
<_io.TextIOWrapper name='iris.txt' mode='r' encoding='UTF-8'>
>>> f.closed
False
>>> f.mode
'r'
>>> f.name
'iris.txt'
```

The file handle f is now pointing to the file iris.txt that has been opened in read mode. When printing the file handle, we see that the file is, in fact, open, the name of the file that was opened, its mode, and where in memory the file handle is.

Individual file attributes may also be accessed:

- **f.closed – if the file is closed (True) or open (False)**
- **f.mode – the mode the file is opened with**
- **f.name – the file name the file handle is pointing to**

Closing a File

The file may be closed using the **close()** function.

```
>>> f.close()
>>> print(f)
<_io.TextIOWrapper name='iris.txt' mode='r' encoding='UTF-8'>
>>> f.closed
True
```

It’s that easy! Call the close function on the file handle and you can confirm the file is closed using the closed function.

Reading from a File

There are many ways to read data from a file, ranging from reading bytes at a time from the file (read) to reading an entire file’s contents into a String (readlines). For the purposes of this course, we’re going to stick with reading a single line at a time from a file using **readline**. Its format is simply **f.readline()**.

Create a file called file_reader.py in a text editor. We’re going to write a script that reads iris.txt line by line until there is no more data to read.

```
# open iris.txt for reading
f = open('iris.txt', 'r')

# read the first line of the file
line = f.readline().strip()

# keep reading until there are no
# more lines in the file
while line:
   print(line)
   # read the next line
   line = f.readline().strip()

f.close()
```

Rather than trying to detect the end of file (EOF), this code continues reading lines from the file until there are no more lines to read. Try printing out the contents of line after the loop to see why the while loop is exited when it reaches EOF.

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
```

Notice in the output that we avoid an “extra” new line by using .strip() in tandem with readline(). We use the strip function because there is a “\n” new line character at the end of each line in the file and we would like to avoid printing that character in addition to the new line created by using a print statement.

Writing Data to a File

Writing data to a file merely requires that a file be opened in write mode and the **write** function is used to write data to the file. The format is **f.write(data)**.

Create a file called file_writer.py in a text editor. The following script illustrates how to write data to a file.

```python
# open iris.txt for writing
f = open('new_iris.txt', 'w')

# create some data
L = [ [ 5.1, 3.5, 1.4, 0.2, 'Iris-setosa' ], [ 7.0, 3.2, 4.7, 1.4, 'Iris-versicolor' ] ]

# loop over each element of L and build a string to write to the output file
for iter in L :
    line = str( iter[ 0 ] ) + "," + str( iter[ 1 ] ) + "," + str( iter[ 2 ] ) + "," + \
           str( iter[ 3 ] ) + "," + iter[ 4 ] + "\n"
    f.write(line)

f.close()
```

The List L contains data for two of the records in iris.txt. In the for loop that iterates over the elements of L, a "line" is built by converting each float to a String and concatenating the items in each List element with commas in between. Note that the continuation operator (\) is used because the "string building" statement is so long.

```
[student@localhost ~]$ python3 file_writer.py
[student@localhost ~]$ cat new_iris.txt
5.1,3.5,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
```

After running the script, the output file is created as expected. Try opening new_iris.txt in file_reader.py to ensure that it can be read properly.

Programming Exercise (*this exercise will be part of Assignment 1*)

Write a Python script called iris_formatter.py that takes two command line arguments: the name of an input file and the name of an output file. The purpose of the script is to create an output file that can eventually be used in our data mining software, Weka, which uses an ARFF file format. For the purposes of the "iris" data set, which we'll be using as an example throughout the semester, Weka will expect an iris data file to begin with the following lines:

```
@RELATION iris

@ATTRIBUTE sepallength  REAL
@ATTRIBUTE sepalwidth   REAL
@ATTRIBUTE petallength  REAL
@ATTRIBUTE petalwidth   REAL
@ATTRIBUTE class    {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
```

RELATION defines the name of the data set, the ATTRIBUTE lines describe what the expected data will look like for a given data attribute, and DATA indicates where the data begins in the file.

Your Python script should do the following:

1. Read in the data from either a .ARFF file or a .CSV file.

2. Save the data to a List.
3. Create a new .ARFF output file by writing out the contents of the List to the file.

You will need to write two reading functions, one that can read in iris.arff (call it read_arff) and one that can read in iris.csv (call it read_csv). Both of these files are on myCourses in Exercise1.zip. Each function should take the file name as an argument and will read the data into a List, then return the List. Hint: in order to call the correct function, you will have to parse the input file name!

When reading in iris.arff, you will find that there is a lot of excess information that you're not interested in. What you want to do is only read in the "data" from the file and skip over everything else. The data begins after the @DATA line in the file. In all, your List at the end of the function should contain 150 data "records".

Reading in iris.csv presents a different problem. The data is relatively easy to read because all 150 lines in the file represent a data record. However, the final item in each line is a number (either 1, 2, or 3), instead of the name of a type of iris flower. What you will need to do is convert the number into a String so that the output ARFF file can be properly read by Weka. This means that if you see a 1 at the end of a line, you should replace it with "Iris-setosa", 2 with "Iris-versicolor", and 3 with "Iris-virginica".

Once the reading function returns the List back to where you called the function from, you should call a function called write_data that takes the List and the output file name as arguments. The function will open the output file, write the exact lines shown in the image above and then write out the data contained in the List in the same format that it was read in. Regardless of your input file, your output file should look very much like iris.arff, except it won't have all the excess information in it.