Sequences: Strings, Lists and Files

Table Data: Rows and Fields

Tables consist of *records* (rows) and *fields* (column values).

Tabular text files are organized into rows and columns.

comma-separated values file (CSV)

```
19260701,0.09,0.22,0.30,0.009
19260702,0.44,0.35,0.08,0.009
19270103,0.97,0.21,0.24,0.010
19270104,0.30,0.15,0.73,0.010
19280103,0.43,0.90,0.20,0.010
19280104,0.14,0.47,0.01,0.010
```

space-separated values file

```
19260701    0.09    0.22    0.30   0.009
19260702    0.44    0.35    0.08   0.009
19270103    0.97    0.21    0.24   0.010
19270104    0.30    0.15    0.73   0.010
19280103    0.43    0.90    0.20   0.010
19280104    0.14    0.47    0.01   0.010
```

Our job for this lesson is to *parse* (separate) these values into usable data.

Table Data in Text Files

Text files are just sequences of characters. *Newline* characters separate text files into lines. Python reads text files *line-by-line* by separating lines at the newlines.

If we print a CSV text file, we may see this:

```
    19260701,0.09,0.22,0.30,0.009
    19260702,0.44,0.35,0.08,0.009
    19270103,0.97,0.21,0.24,0.010
    19270104,0.30,0.15,0.73,0.010
    19280103,0.43,0.90,0.20,0.010
    19280104,0.14,0.47,0.01,0.010
```

However, here's what a text file really looks like under the hood:

```
19260701,0.09,0.22,0.30,0.009\n19260702,0.44,0.35,0.08,
0.009\n19270103,0.97,0.21,0.24,0.010\n19270104,0.30,0.15,
0.73,0.010\n19280103,0.43,0.90,0.20,0.010\n19280104,0.14,
0.47,0.01,0.010
```

The newline character separates the **records** in a CSV file. The *delimeter* (in this case, a comma) separates the fields.

When displaying a file, your computer will translate the newlines into a line break, and drop down to the next line. This makes it seem as if each line is separate, but in fact they are only separated by newline characters.

Goals for this Unit

These objectives encapsulate the core, more central approach to processing data in Python.

The Summing Average

Can we sum a column of values from a table? Our data analysis will resemble that of **excel**: it will read a column of value from a tabular file and *sum up* the values in that column.

Here's how it works:

- We set a summing float variable to 0.
- We call function **open()** with a filename, returning a **file object**.

- We loop through the file by using **for** with the file object.
- Inside the **for** block, each line is a string. We **split()** the string on the delimeter (here, a comma), returning a **list of strings**.
- We **subscript** the list to select the field value we want -- this is a string object with a numeric value.
- We convert the string to a float object.
- We add the float object to the summing float variable.
- The loop continues until the file is exhausted. **The Counting** *Can we parse and count the lines, words and characters in a file?* We will emulate the work of the Unix **wc** (word count) utility, which does this work. Here's how it works:
- We call function **open()** with a filename, returning a file object.
- We call **read()** on the file object, returning a string object containing the entire file text.
- We call **splitlines()** on the string, returning a list of strings. **len()** will then tell us the number of lines.
- We call **split()** on the same string, returning a list of strings. **len()** will then tell us the number of words.
- We call **len()** on the string to count the number of characters.

Summary: File Object

3 ways to read strings from a file.

for: read line-by-line

```
fh = open('../python_data/students.txt')  # file object allows looping thr
                                          # series of strings
for my_file_line in fh:                   # my_file_line is a string
    print my_file_line                    # prints each line of students.t

fh.close()                                # close the file
```

read(): read entire file as a single string

```
fh = open('../python_data/students.txt')  # file object allows reading
text = fh.read()                          # read() method called on file o
fh.close()                                # close the file

print text
```

The above prints:

```
jw234,Joe,Wilson,Smithtown,NJ,2015585894
ms15,Mary,Smith,Wilsontown,NY,5185853892
pk669,Pete,Krank,Darkling,NJ,8044894893
```

readlines(): read as a list of strings

```
fh = open('../python_data/students.txt')
file_lines = fh.readlines()              # file.readlines() returns a lis
fh.close()                               # close the file
print file_lines
```

The above prints:

```
['jw234,Joe,Wilson,Smithtown,NJ,2015585894\n', 'ms15,Mary,Smith,Wilsontowr
NY,5185853892\n', 'pk669,Pete,Krank,Darkling,NJ,8044894893\n']
```

Summary: String Object

Strings: 4 ways to manipulate strings from a file.

split() a string into a list of strings

```
mystr = 'jw234,Joe,Wilson,Smithtown,NJ,2015585894'
elements = mystr.split(',')
print elements                # ['jw234', 'Joe', 'Wilson', 'Smithtown', 'N
```

slice a string

```
mystr = '2014-03-13 15:33:00'
year =  mystr[0:4]            # '2014'
month = mystr[5:7]            # '03'
day =   mystr[8:10]           # '13'
```

strip() a string

```
xx = 'this is a line with a newline at the end\n'

yy = xx.rstrip()             # return a new string without the newline

print yy                     # 'this is a line with a newline at the end'
```

splitlines() a multiline string

```
fh = open('../python_data/students.txt')  # open the file, return a file c
text = fh.read()                          # read the entire file into a st
                                          # (of course this includes newli

lines = text.splitlines()                 # returns a list of strings
                                          # (similar to fh.readlines(),
                                          # except without newlines)
```

Summary: List Object

Lists: selecting individual elements of a list.

A list is a sequence of objects of any type:

initialize a list: lists are initalized with square brackets and comma-separated objects.

```
aa = ['a', 'b', 'c', 3.5, 4.09, 2]
```

subscript a list: using the list name, square brackets an an element *index*, starting at 0

```
elements = ['jw234', 'Joe', 'Wilson', 'Smithtown', 'NJ', '2015585894']

var = elements[0]              # 'jw234'
var2 = elements[4]             # 'NJ'
var3 = elements[-1]            # '2015585894' (-1 means last index)
```

Summary: len() function for string and list length

len() can be used to measure lists as well as strings.

```
mystr = 'hello'
mylist = [1.3, 1.9, 0.9, 0.3]

lms = len(mystr)               # 5  (number of characters in mystr)
lml = len(mylist)              # 4  (number of elements in mylist)
```

Because it can measure lists or strings, **len()** can also measure files (when rendered as a list of strings or a whole string).

Summary: repr() function for "true" representations of strings

repr() takes any object and shows a more "true" representation of it. With a string, **repr()** will show us the newlines at the end of each line

```
aa = open('../python_data/small_db.txt')  # open a file, returns a file ob

xx = aa.read()                # read() on a file object, returns a single 

print repr(xx)                # the string with newlines visible:  '101:Ac
```

Reading a file: options

(Note that the remaining slides repeat some of the same material, but from a more practical perspective.)

<u>**for** loop: loop line-by-line</u>

The **for** loop repeats execution of its block until the file is completely read.

Note that the **for** block is very similar to the **while**. The difference is that **while** relies on a test to continue executing, but **for** continues until it reaches the end of the file.

```
fh = open('../python_data/students.txt')    # file object allows looping t
                                            # series of strings

for xx in fh:                               # xx is a string

    print xx                                # prints each line of students

fh.close()                                  # close the file
```

"my_file_line" is called a *control variable*, and it is *automatically reassigned* each line in the file as a string.

break and **continue** work with **for** as well as **while** loops.

readlines(): work with the file as a list of string lines

To capture the entire file into a list of lines, use the file **readlines()** method:

```
fh = open('../python_data/students.txt')

lines = fh.readlines()

for line in lines:
    line = line.rstrip()
    print line

print lines[0]              # the first line from the file

print len(lines)            # the number of lines in the file
```

We can then loop through the list, or perform other operations (select a single line or slice, get the number of lines with **len()** of the list, etc.)

read() with **splitlines()**: an easy way to drop the newlines

A handy trick is to **read()** the file into a string, then call **splitlines()** on the string to split on newlines.

```
fh = open('../python_data/students.txt')

text = fh.read()
lines = text.splitlines()

for line in lines:
    print line
```

This has the effect of delivering the entire file as a list of lines, but with the newlines removed (because the string was split on them with **splitlines()**).

Table *Records* Are Read from a file as String Objects

As Python reads files line-by-line, it handles each line as a string object.

```
fh = open('../python_data/students.txt')   # file object allows looping th
                                           # series of strings
for bb in fh:
     print type(bb)                        # <type 'str'>
```

Again, the control variable **bb** is *reassigned for each iteration of the loop*. This means that if the file has 5 lines, the loop executes 5 times and **bb** is reassigned a new value 5 times.

Stripping a file line with rstrip()

When reading a file line-by-line, we should strip off the newline with the string method **rstrip()**.

```
fh = open('../python_data/students.txt')    # file object allows looping t
                                            # series of strings
for xx in fh:                         # xx is a string

    xx = xx.rstrip()                  # remove "whitespace" from end of the line
                                      # and return a new string with the string

    print xx                          # prints each line of students.txt

fh.close()                            # close the file
```

String slicing

A string can be *sliced* by position: we specify the start and end position of the slice.

Indices start at 0; the "upper bound" is *non-inclusive*

```
mystr = '19320805    3.62   -2.38    0.08   0.001'
year =  mystr[0:4]              # '1932'
month = mystr[4:6]              # '08'
day = mystr[6:8]                # '05'
```

To slice to the end, omit the upper bound

```
mystr = '19320805    3.62   -2.38    0.08   0.001'

rf_val = mystr[32:]             # '   0.001'
```

Table *Fields* Are Parsed from File Line Strings into *Lists of Strings*

The string **split()** method returns a *list of strings*, each string a field in a single record (row or line from the table).

The *delimeter* tells Python how to split the string. Note that the delimeter does *not* appear in the list of strings.

```
line_from_file = 'jw234:Joe:Wilson:Smithtown:NJ:2015585894\n'

xx = line_from_file.split(':')

print xx                           # ['jw234', 'Joe', 'Wilson', 'Smithtown',
```

If no delimeter is supplied, the string is split *on whitespace*:

```
gg = 'this is a file    with    some     whitespace'

hh = gg.split()                    # splits on any "whitespace character"

print hh                           # ['this', 'is', 'a', 'file', 'with', 'so
```

Table Fields Are Selected from a List Using *List Subscripts*

Each table record (row or line from the table) when rendered as a list of strings (from **split()**) is *addressable* by *index*.

The index starts at 0. A *negative index* (-1, -2, etc.) will count from the end.

```
gg = '2016:5.0:5.3:5.9:6.1'

hh = gg.split(':')            # splits on any "whitespace character"

print hh                      # ['2016', '5.0', '5.3', '5.9', '6.1']

kk = hh[0]                    # '2016'   (index starts at 0)

mm = hh[1]                    # '5.0'

zz = hh[-1]                   # '6.1'    (negative index selects from the

yy = hh[-2]                   # '5.9'
```

Portions of a string can be "sliced" using *string slicing*

Special *slice syntax* lets us specify a *substring* by position.

split() separates a string based on a delimeter, but some strings have no delimeter but must be parsed by position:

```
mystr = '20140313'
year =  mystr[0:4]             # '2014'  (the 0th through 3rd index)
month = mystr[4:6]             # '03'    (the 4 and 5 index values)
day =   mystr[6:]              # '13'    (note that no upper index means s
```

Note that the upper index is *non-inclusive*, which means that it specifies the index *past* the one desired.

stride and *negative stride*

A third value, the *stride* or *step* value, allows skipping over characters (every 2nd element every 3rd element, etc.)

```
mystr = '20140303'

skipper = mystr[0:7:2]        # '2100'
```

The negative stride actually *reverses* the string (when used with no other index):

```
mystr = '20140303'

reverser = mystr[::-1]      # '0304102'
```