Files, Strings, and Dictionaries

Mojtaba Alaei

November 9, 2013

Content of the course

Reading a file Line by Line

Consider the following text file:

```
18.3
92.0
34.0
31.0
12.4
42.1
10.4
```

To read a file, we first need to open the file. This action creates a file object, here stored in the variable infile:

```
infile = open('data1.txt', 'r')
```

The string 'r', tells that we want to open the file for reading.
The basic recipe for reading the file line by line applies a for loop like this:

```
for line in infile:
    # do something with line
```

Reading a file Line by Line

Instead of reading one line at a time, we can load all lines into a list of strings (lines) by:

```
liens = infile.readlines()
```

This statement is equivalent to:

```
lines = []
for line in infile:
    lines.append(line)
```

example: Compute the average of the numbers in the file

Try the following program:

```
infile = open('data1.txt', 'r')
lines = infile.readlines()

mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

example: Compute the average of the numbers in the file

Try the following program:

```
infile = open('data1.txt', 'r')
lines = infile.readlines()

mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

gives an error message:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The reason is that lines holds each line (number) as a string, not a float or int that we can add to other numbers.
A fix is to convert each line to a float:

```
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
```

example: Compute the average of the numbers in the file

An alternative implementation is to load the lines into a list of float objects directly:

```
infile = open('data1.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

While Loop over Lines

When infile.readline() returns an empty string, the end of the file is reached and we must stop further reading. The following while loop reads the file line by line using infile.readline():

```
while True:
    line = infile.readline()
    if not line:
        break
    # process line
```

Computing the average of the numbers in the data1.txt file can now be done in yet another way:

```
infile = open('data1.txt', 'r')
mean = 0
n = 0
while True:
    line = infile.readline()
    if not line:
        break
    mean += float(line)
    n += 1
mean = mean/float(n)
print mean
```

Reading a File into a String

The call infile.read() reads the whole file and returns the text as a string object:

```
>>> infile = open('data1.txt', 'r')
>>> filestr = infile.read()
>>> filestr
'18.3\n92.0\n34.0\n31.0\n12.4\n42.1\n10.4\n'
>>> print filestr
18.3
92.0
34.0
31.0
12.4
42.1
10.4
```

split() will split the string into words:

```
>>> words = filestr.split()
>>> words
['18.3', '92.0', '34.0', '31.0', '12.4', '42.1', '10.4']
>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)
>>> print mean
34.3142857143
```

A more compact program looks as follows:

```
infile = open('data1.txt', 'r')
numbers = [float(w) for w in infile.read().split()]
mean = sum(numbers)/len(numbers)
```

Reading a Mixture of Text and Numbers

Many data files contain a mix of text and numbers. The file rainfall.dat provides an example:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan 81.2
Feb 63.2
Mar 70.3
Apr 55.7
May 53.0
Jun 36.4
Jul 17.5
Aug 27.5
Sep 60.9
Oct 117.7
Nov 111.0
Dec 97.9
Year 792.9
```

Reading a Mixture of Text and Numbers

How can we read the rainfall data in this file and make a plot of the values?

Reading a Mixture of Text and Numbers

How can we read the rainfall data in this file and make a plot of the values?

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = []
    for line in infile:
        words = line.split()
        number = float(words[1])
        numbers.append(number)
    infile.close()
    return numbers

values = extract_data('rainfall.dat')
from pylab import plot, show
month_indices = range(1,13)
plot(month_indices, values[:-1])
show()
```

Reading a Mixture of Text and Numbers

We can condense the for loop over lines in the file, if desired, by using a list comprehension:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = [float(line.split()[1]) for line in infile]
    infile.close()
    return numbers
```

Dictionaries

Suppose we need to store the temperatures from three cities: Oslo, London, and Paris. For this purpose we can use a list,

```
temps = [13, 15.4, 17.5]
```

But it is better to make a dictionary as follows:

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
# or
temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Additional text-value pairs can be added when desired. We can, for instance, write

```
>>> temps['Madrid'] = 26.0
>>> print temps
{'Paris': 17.5, 'Oslo': 13, 'London': 15.4, 'Madrid': 26.0}
```

Dictionary Operations

The string "indices" in a dictionary are called keys. To loop over the keys in a dictionary d, one writes for key in d: and works with key and the corresponding value d[key] inside the loop:

```
>>> for city in temps:
...     print 'The temperature in %s is %g' % (city, temps[city])
...
The temperature in Paris is 17.5
The temperature in Oslo is 13
The temperature in London is 15.4
The temperature in Madrid is 26
```

We can check if a key is present in a dictionary by the syntax if key in d:

```
>>> if 'Berlin' in temps:
...     print 'Berlin:', temps['Berlin']
... else:
...     print 'No temperature data for Berlin'
...
No temperature data for Berlin
```

Dictionary Operations

Writing key in d yields a standard boolean expression, e.g.:

```
>>> 'Oslo' in temps
True
```

The keys and values can be extracted as lists from a dictionary:

```
>>> temps.keys()
['Paris', 'Oslo', 'London', 'Madrid']
>>> temps.values()
[17.5, 13, 15.4, 26.0]
```

A key-value pair can be removed by del d[key]:

```
>>> del temps['Oslo']
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps) # no of key–value pairs in dictionary
3
```

Dictionary Operations

Sometimes we need to take a copy of a dictionary:

```
>>> temps_copy = temps.copy()
>>> del temps_copy['Paris']
# this does not affect temps
>>> temps_copy
{'London': 15.4, 'Madrid': 26.0}
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

Note that:

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0 # change t1
>>> temps # temps is also changed
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

Example: Polynomials as Dictionaries

Consider the polynomial

$$p(x) = -1 + x^2 + 3x^7. \tag{1}$$

A dictionary can be used to map a power to a coefficient:

```
p = {0: -1, 2: 1, 7: 3}
```

A list can, of course, also be used, but in this case we must fill in all the zero coefficients too, since the index must match the power:

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

The following function can be used to evaluate a polynomial represented as a dictionary:

```
def poly1(data, x):
    sum = 0.0
    for power in data:
        sum += data[power]*x**power
    return sum
```

A more compact implementation:

```
def poly1(data, x):
    return sum([data[p]*x**p for p in data])
```

Example: File Data in Dictionaries

Consider densities.dat file:

```
air                 0.0012
gasoline            0.67
ice                 0.9
pure water          1.0
seawater            1.025
human body          1.03
limestone           2.6
granite             2.7
iron                7.8
silver              10.5
mercury             13.6
gold                18.9
platinium           21.4
Earth mean          5.52
Earth core          13
Moon                3.3
Sun mean            1.4
Sun core            160
proton              2.8E+14
```

Example: File Data in Dictionaries

Solution:

```
def read_densities(filename):
    infile = open(filename, 'r')
    densities = {}
    for line in infile:
        words = line.split()
        density = float(words[-1])

        if len(words[:-1]) == 2:
            substance = words[0] + ' ' + words[1]
        else:
            substance = words[0]



        densities[substance] = density
    infile.close()
    return densities
```

Common Operations on Strings

Substring Specification:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s[8:]     # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]   # index 8, 9, 10 and 11 (not 12!)
'18.4'
```

A negative upper index counts, as usual, from the right such that s[-1] is the last element, s[-2] is the next last element, and so on:

```
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

Searching for Substrings. The call s.find(s1) returns the index where the substring s1 first appears in s. If the substring is not found, -1 is returned.

```
>>> s.find('Berlin')  # where does 'Berlin' start?
0
>>> s.find('pm')
20
>>> s.find('Oslo')  # not found
-1
```

Common Operations on Strings

To just check if a string is contained in another string:

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False
```

Two other convenient methods for checking if a string starts with or ends with a specified string are startswith and endswith:

```
>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False
```

Substitution. The call s.replace(s1, s2) replaces substring s1 by s2 everywhere in s:

```
>>> s.replace(' ', '_')
'Berlin:_18.4_C__at_4_pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

A nice example:

```
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

Common Operations on Strings

String Splitting. The call s.split() splits the string s into words separated by whitespace (space, tabulator, or newline):

```
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

Splitting a string s into words separated by a text t can be done by s.split(t). For example, we may split with respect to colon:

```
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
```

With s.splitlines(), a multi-line string is split into lines (very useful when a file has been read into a string and we want a list of lines):

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
2nd line
3rd line
>>> t.splitlines()
['1st line', '2nd line', '3rd line']
```

Common Operations on Strings

Upper and Lower Case. s.lower() transforms all characters to their lower case equivalents, and s.upper() performs a similar transforma- tion to upper case letters:

```
>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'
```

Strings Are Constant. A string cannot be changed, i.e., any change always results in a new string. Replacement of a character is not pos- sible:

```
[18] = 5
...
TypeError: 'str' object does not support item assignment
```

If we want to replace s[18], a new string must be constructed, for example by keeping the substrings on either side of s[18] and inserting a '5' in between:

```
>>> s[:18] + '5' + s[19:]
'Berlin: 18.4 C at 5 pm'
```

Common Operations on Strings

Strings with Digits Only. One can easily test whether a string contains digits only or not:

```
>>> '214'.isdigit()
True
>>> ' 214 '.isdigit()
False
>>> '2.14'.isdigit()
False
```

Whitespace. We can also check if a string contains spaces only by call- ing the isspace method. More precisely, isspace tests for whitespace, which means the space character, newline, or the TAB character:

```
>>> '     '.isspace() # blanks
True
>>> ' \n'.isspace()   # newline
True
>>> ' \t '.isspace()  # TAB
True
>>> ''.isspace()      # empty string
False
```

Common Operations on Strings

Stripping off leading and/or trailing spaces in a string is sometimes useful:

```
>>> s = '    text with leading/trailing space    \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip()    # left strip
'text with leading/trailing space    \n'
>>> s.rstrip()    # right strip
'    text with leading/trailing space'
```

Joining Strings. The opposite of the split method is join:

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> t = ', '.join(strings)
>>> t
'Newton, Secant, Bisection'
```

Example: Reading Pairs of Numbers

Make a file (for example read_pairs1.dat) with following format:

```
(1.3,0)   (0,1)   (0,-0.01)
(-1,2)   (3,-1.5)   (1,0)
(1,1)    (10.5,-1) (2.5,-2.5)
```

We want to o read this text into a nested list pairs such that pairs[i] holds the pair with index i, and this pair is a tuple of two float objects.
Solution:

```
lines = open('read_pairs1.dat', 'r').readlines()
pairs = []  # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1] # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1); n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair) # add 2-tuple to last row
```

Example: Reading Pairs of Numbers

How to read the following data ('xyz.dat'):

```
x=-1.345      y= 0.1112       z= 9.1928
x=-1.231      y=-0.1251       z= 1001.2
x= 0.100      y= 1.4344E+6    z=-1.0100
x= 0.200      y= 0.0012       z=-1.3423E+4
x= 1.5E+5     y=-0.7666       z= 1027
```

Example: Reading Pairs of Numbers

How to read the following data ('xyz.dat'):

```
x=-1.345      y= 0.1112       z= 9.1928
x=-1.231      y=-0.1251       z= 1001.2
x= 0.100      y= 1.4344E+6    z=-1.0100
x= 0.200      y= 0.0012       z=-1.3423E+4
x= 1.5E+5     y=-0.7666       z= 1027
```

Solution:

```
infile = open('xyz.dat', 'r')
coor = [] # list of (x,y,z) tuples
for line in infile:
    words = line.split('=')
    x = float(words[1][:-1])
    y = float(words[2][:-1])
    z = float(words[3])
    coor.append((x, y, z))
infile.close()
import numpy as np
coor = np.array(coor)
print coor.shape, coor
```

Writing Data to File

outfile.write(s), which writes a string s to a file handled by the file object outfile.
Writing to a file demands the file object f to be opened for writing:

```
# write to new file, or overwrite file:
outfile = open(filename, 'w')
# append to the end of an existing file:
outfile = open(filename, 'a')
```

An example:

```
data = [[ 0.75,        0.29619813, -0.29619813, -0.75      ],
        [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
        [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
        [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```