Introductory Python 4

We have already met the basic data structures that Python offers:

- lists,
- tuples, and
- dictionaries.

A lot of data science involves munging data obtained from various sources, such as the web or Excel CSV files. These are typically not in the format we need for the analysis, so we must process these strings and extract the useful information. This lecture will concentrate on different aspects of this problem.

- string processing
- Input/Output
- regular expressions

Strings

As we've seen earlier, strings share many of the attributes of lists.

In [1]:
```
fruit = 'banana'
print(fruit[1])
```

```
a
```

The expression fruit[1] treats fruit as a list, with the character at index 1 being picked out (recall that list indices start at 0). Similarly, the *len* function works with strings too.

In [2]:
```
len(fruit)
```

Out[2]:
```
6
```

In [3]:
```
# Ditto with loops
for char in fruit:
    print(char)
```

```
b
a
n
a
n
a
```

In [4]:
```
# and slices
fruit[1:3] # recall that this selects items from index 1 up to, but not
```

Out[4]:
```
'an'
```

However, the **difference from lists** is that a string cannot be modified.

In [5]:
```
# fruit[2] = 'X'  # Not allowed!
```

Example:

Suppose I want to write the following function: take a string, and a list of indices, and return a string with those indices blanked out.

> blank_out('abcdef', [1,3,5]) should return 'a_c_e_'

How do we do this?

In [6]:
```
def blank_out(orig_string, indices):
    # Assemble blanked out string character by character
    blanked_out_string = ''  # empty string

    for index, char in enumerate(orig_string):
        if index in indices:
            blanked_out_string += '_'   # recall: x += y is shorthand f
        else:
```

```
            blanked_out_string += char

    return blanked_out_string

print(blank_out('abcdef', [1, 3, 5])       )
```

```
a_c_e_
```

Another option is to:

1. convert orig_string, which cannot be modified, into a list, which *can* be modified,
2. modify the list at the given indices, and finally,
3. convert the list back into a string.

In [7]:
```
def blank_out_2(orig_string, indices):
    orig_list = list(orig_string)  # this converts the string into a li

    for index in indices:
        orig_list[index] = '_'  # change the list

    blanked_out_string = ''.join(orig_list) # join convert list into st
    return blanked_out_string

print(blank_out_2('ABCDEF', [1, 3, 5]))
```

```
A_C_E_
```

We used the *join* method, which concatenates a list of items using a delimiter.

```
delimiter_string.join([item1, item2, item3, ...])
```

Here, we called *join* on the empty string '' since we didn't want any extra stuff between the character list.

Alongside *join()*, we will need three common string manipulation functions:

> *string.lstrip(char)*

This strips away character char, if present, from the beginning of the string

In [8]:
```
'AAA Car Insurance'.lstrip('A')
```

Out[8]:
```
' Car Insurance'
```

In [9]:
```
'AAA Car Insurance'.lstrip('B')
```

Out[9]:
```
'AAA Car Insurance'
```

> *string.rstrip(char)*

Same as lstrip but from the end of the string (lstrip for left, rstrip for right)

In [10]:
```
'AAA Car Insurance'.rstrip('e')
```

Out[10]:
```
'AAA Car Insuranc'
```

In [11]:
```
'AAA Car Insurance'.rstrip('E')
```

Out[11]:
```
'AAA Car Insurance'
```

> *string.split(string2)*

This breaks up a string into pieces separated by string2

In [12]:
```
"A told B, and B told C, I'll race you to the top, of the coconut tree"
```

Out[12]:
```
['A told B', 'and B told C', "I'll race you to the top", 'o
f the coconut tree']
```

In [13]:
```
"A told B, and B told C, I'll race you to the top, of the coconut tree"
```

```
['A',
```

Out[13]:
```
 'told',
 'B,',
 'and',
 'B',
 'told',
 'C,',
"I'll",
 'race',
 'you',
 'to',
 'the',
 'top,',
 'of',
 'the',
 'coconut',
 'tree']
```

Example:

Let's do another common use-case: parsing a string. Suppose we are given the string '{12.4, 3.4, 4, 36}' and we want to extract all the numbers into a list of floating point numbers.

How do we do it?

(As an aside, when dealing with Excel CSV files, we need something very similar).

In [14]:
```
orig_string = '{12.4, 3.4, 4, 36}'

# First step: make a copy of this string; we will operate on the copy
vec = orig_string

# First step: remove the { and }
# Use lstrip(), which strips away matching characters from the beginnir
vec = vec.lstrip('{') # Store the resulting string in vec again
vec = vec.rstrip('}')
print('vec after stripping beginning and end:', vec)

# Now we need to split the string by the ', ' delimiter
nums = vec.split(', ')  # Note: don't forget the space in ', '
print('nums =', nums)
```

```
# nums is a list of strings, but we need a list of floating point numbe
float_nums = [float(x) for x in nums]    # list-comprehension
print('floating point list is', float_nums)
```

```
vec after stripping beginning and end: 12.4, 3.4, 4, 36
nums = ['12.4', '3.4', '4', '36']
floating point list is [12.4, 3.4, 4.0, 36.0]
```

Formatting and printing

We have want to combine several variables to form one string.

In [15]:
```
name = 'Deepayan'
ID = 123
```

Suppose I want to combine them into a string:

```
"Hello Citizen Deepayan ID 123"
```

In [16]:
```
### won't work since ID is integer!
# greeting = 'Hello Citizen ' + name + ' ID ' + ID
```

The *format* function allows us to create a string from a *template*.

In [17]:
```
# The {} spaces are filled in successively by the parameters of the for
greeting = 'Hello Citizen {} ID {}'.format(name, ID)
print(greeting)
```

```
Hello Citizen Deepayan ID 123
```

In [18]:
```
# The same parameter can be used repeatedly in the template
greeting = '{name}! Hey {name}! Can you hear me, {name}?'.format(name='
print(greeting)
```

```
Deepayan! Hey Deepayan! Can you hear me, Deepayan?
```

Summary of strings

- Strings mostly act like lists

    - Get a list of its characters by list("string")
- **split()** a string into a list, or **join()** a list into a string
- **lstrip()** and **rstrip()** to remove beginning and trailing characters
- Many more string functions: run help()

Input/Output

We will often have to deal with reading input from files, and writing results out to files. The approach is simple:

1. Open the file, specifying whether you want to read, write, or append
2. Do what you want with it
3. Close the file

Let's do this with an Excel CSV file.

In [19]:
```
# A super-simple car
!cat Intro_4_data/CarParts.csv
```

```
Part name,Number of units,Price per unit,Total price
Wheels,4,500,2000
Doors,4,200,800
Windows,4,100,400
Engine,1,2000,2000
Body,1,5000,5000
```

Three types of operations

- Open the file
- Read in one line at a time
- Close the file

In [20]:
```
# Opening the file
fp = open('Intro_4_data/CarParts.csv', 'r')
```

- fp stands for "file pointer". Once the file has been opened, all our interactions with that file (e.g., reading line by line) will go through this file pointer.
- The 'r' stands for 'read'; Python will not allow us to write to this file pointer.

In [21]:
```
first_line = fp.readline()  # Reads in one line
print(first_line)
```

```
Part name,Number of units,Price per unit,Total price
```

Let's say we want to load all the parts information, compute the total cost, and write out the total cost to another file.

How do we do this?

- Read in one line at a time
- Split it into its parts
- Keep a running total over part prices
- At the end, close the file and report the results.

In [22]:
```
# Let us iterate over the lines.
# readline() will return the empty string when the end of file is reach
price_of_car = 0

line = fp.readline()
while line is not '':
    # the line can have some trailing characters, like 'newline'
    line = line.rstrip()

    # split it into parts, to get the total price
    part_name, num_units, price_per_unit, total_price_of_part = line.sp

    # add this part's price to the total price of the car
    # remember: everything we get from the split() is a string.
    price_of_car += int(total_price_of_part)

    # print a running total
```

```
    print('After adding', part_name, 'running total =', price_of_car)

    # read in a new line
    line = fp.readline()

print('Total price of car =', price_of_car)
```

```
After adding Wheels running total = 2000
After adding Doors running total = 2800
After adding Windows running total = 3200
After adding Engine running total = 5200
After adding Body running total = 10200
Total price of car = 10200
```

In [23]:

```
# Finally, close the file pointer
fp.close()
```

Reading all lines in a file can also be done in a *for loop*.

In [24]:

```
price_of_car = 0

fp = open('Intro_4_data/CarParts.csv', 'r')
line = fp.readline()  # Ignore first line (the "header" line)

for line in fp:  # <--- Iterate over lines from a file
    line = line.rstrip()
    part_name, num_units, price_per_unit, total_price_of_part = line.sp
    price_of_car += int(total_price_of_part)

fp.close()
print('Total price of car =', price_of_car)
```

```
Total price of car = 10200
```

Now lets write out this total price to a CSV file. Again, open a file pointer, write it to file using this file pointer, and close the file pointer.

In [25]:

```
# Open file for writing
fp = open('Result.csv', 'w')  # the 'w' option says we want to write to

# Write out the header for the CSV
```

```
print('Type of purchase,Price', file=fp)

# Write out the values for Car
# price_of_car is a number, which must be converted to a string before
print('Car,{}'.format(price_of_car), file=fp)

# Close the file pointer
fp.close()
```

In [26]:

```
# Check the results
!cat Result.csv
```

```
Type of purchase,Price
Car,10200
```

Regular expressions

Often, the string we must deal with are far uglier. Regular expressions help find complex patterns in strings. Let's see some examples.

In [27]:

```
import re  # This is the regular expressions module. We must import thi
```

In [28]:

```
string_list = ['no numbers', 'That costs $2,000!', 'Beverly Hills 90210

# Find strings that contain numbers
for one_string in string_list:
    matches = re.findall('[0-9]', one_string)
    if len(matches) > 0:
        print('The matches in "'+ one_string + '" are:', matches)
    else:
        print('No matches in "' + one_string + '"')
```

```
No matches in "no numbers"
The matches in "That costs $2,000!" are: ['2', '0', '0',
'0']
The matches in "Beverly Hills 90210" are: ['9', '0', '2',
'1', '0']
```

The basic approach to regular expressions is that you give

- a searcher (here, *re.findall()*)
- a pattern (here, '*[0-9]*')
- and a string in which to search (here, *one_string*).

In this case:

- The pattern '*[0-9]*' means: any character between '0' and '9', i.e., any one of '0', '1', '2', ..., '9'
- The searcher *re.findall()* tries to find all match of the pattern to the string (*one_string*).

The re module has many other specialized searchers.

Let's look a bit more at some commonly used patterns.

- **[...]** means match any character within the square brackets
- **[^...]** means match anything *except* the characters within the brackets

> [abcd]

means match any **one** of 'a', 'b', 'c', or 'd'

> [a-z]

means match anything lower-case character

> [a-zA-Z]

means match any lower-case or upper-case character

> [0-9]

means any number

> **.** (the "full stop" sign)

means match any one character, whatever it is (this is called a *wild card*)

> **\.**

means match the full-stop character

There are several useful shorthands as well:

- **\w** is shorthand for [a-zA-Z0-9_]
- **\d** is shorthand for [0-9]
- **\s** is shorthand for a space or a tab, often used as delimiter

Example (Find phone numbers):

Let's try to find phone numbers in text. We will only consider the forms

```
123.456.7890
```

or

```
123-456-7890
```

and we will assume there is a space/tab both before and after the number.

In [29]:
```
def find_phone_number(s):
    """Given a string, find a phone number in it.
       Consider only two forms:
       123.456.7890 or
       123-456-7890"""
    print(re.findall('\s\d\d\d[\.-]\d\d\d[\.-]\d\d\d\d\s', s))

find_phone_number(' Call me at 512-232-1234 or 888.291.2135 ASAP')
```

```
[' 512-232-1234 ', ' 888.291.2135 ']
```

Example (Find Twitter handles):

Let's say we want to detect twitter handles. These will start with the '@' character, followed by a combination of letters, numbers, and the underscore _.

```
@BarackObama
@taylorswift13
@Harry_Styles
```

We have one problem: we don't know how many characters there will be after the '@' sign. For this, we need to understand pattern *repetitions*.

- **\w** matches one English character
- **\w?** matches at most one English character (e.g., 'https?' matches both 'http' and 'https')
- **\w*** matches zero or more English characters (e.g., both '' and 'Henry')
- **\w+** matches one or more English characters (e.g., 'Henry', but not '')

In [30]:
```
def find_twitter_handles(s):
    print(re.findall('@[a-zA-Z0-9_]+', s))

blatantly_false_string = """
I'm gonna have the MOST followers! More tha @BarackObama, bigger
that @katyperry, gonna top @taylorswift13, snuff out @Harry_Styles,
and all you punks out there! That's right, contact me at
bigdaddy@utexas.edu while I set up my account.
"""

find_twitter_handles(blatantly_false_string)
```

```
['@BarackObama', '@katyperry', '@taylorswift13', '@Harry_St
yles', '@utexas']
```

Worked, apart from the piece of the email address '@utexas'

showing up. Let's add the requirement that real Twitter handles in text will have a space before them.

In [31]:
```
def find_twitter_handles_2(s):
    print(re.findall('\s@[a-zA-Z0-9_]+', s))

find_twitter_handles_2(blatantly_false_string)
```

```
[' @BarackObama', ' @katyperry', ' @taylorswift13', ' @Harr
y_Styles']
```

Example (Find email addresses):

We will search for email addresses like deepay@utexas.edu. How?

First cut:

1. some characters (not @ or space),
2. followed by @,
3. followed by some more characters (not @ or space).

In [32]:
```
def find_emails(s):
    print(re.findall('[^@\s]+@[^@\s]+', s))

find_emails('Hello from csev@umich.edu to cwen@iupui.edu about the meet
```

```
['csev@umich.edu', 'cwen@iupui.edu']
```

In [33]:
```
# More complicated setup: extract emails from email headers.
header_string = """
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
"""

find_emails(header_string)
```

```
['stephen.marquard@uct.ac.za', '<postmaster@collab.sakaipro
ject.org>', '<source@collab.sakaiproject.org>;', 'apache@lo
calhost)', 'stephen.marquard@uct.ac.za']
```

Some of the extracted emails have '<' and ')'. Let's add the requirement that

- the email address must contain only [a-z] or [A-Z] or '.', and
- it must start and end with [a-z] or [A-Z].

In [34]:
```
def find_emails_2(s):
    print(re.findall('[a-zA-Z][a-zA-Z\.]*@[a-zA-Z\.]*[a-zA-Z]', s))

find_emails_2('Hello from csev@umich.edu to cwen@iupui.edu about the me
find_emails_2(header_string)
```

```
['csev@umich.edu', 'cwen@iupui.edu']
['stephen.marquard@uct.ac.za', 'postmaster@collab.sakaiproj
ect.org', 'source@collab.sakaiproject.org', 'apache@localho
st', 'stephen.marquard@uct.ac.za']
```

Example (detect URLs):

Let us try to detect URLs within webpages. Webpages are often written in HTML, and will encode URLs within a special *tag*, in the following format:

```
...<a href="http://mccombs.utexas.edu">McCombs</a>...
...<a href="https://gmail.com">Gmail</a>...
```

Here, the pattern is:

```
<a href="http://..."> or
<a href="https://...">
```

In [35]:
```
def find_URLs(s):
    print(re.findall('<a href="https?://[^"]+">', s))

test_string_1 = '...<a href="http://mccombs.utexas.edu">McCombs</a>...'
test_string_2 = '...<a href="https://gmail.com">Gmail</a>...'

find_URLs(test_string_1)
find_URLs(test_string_2)
```

```
['<a href="http://mccombs.utexas.edu">']
['<a href="https://gmail.com">']
```

That's nice, except we didn't really want the 'a href' part in the output, though we need it to be part of the pattern.

To fix this, we can put parentheses around the part we want as the result.

In [36]:

```
def find_URLs_2(s):
        print(re.findall('<a href="(https?://[^"]+)">', s))

find_URLs_2(test_string_1)
find_URLs_2(test_string_2)
```

```
['http://mccombs.utexas.edu']
['https://gmail.com']
```