Semester 2, 2019: Lab 5

S2 2019

Lab 5

Note: If you do not have time to finish all exercises (in particular, the programming problems) during the lab time, you should continue working on them later. You can do this at home (if you have a computer with python set up), in the CSIT lab rooms outside teaching hours, or on one of the computers available in the university libraries or other teaching spaces.

If you have any questions about or difficulties with any of the material covered in the course so far, ask your tutor for help during the lab.

Objectives

The purpose of this week's lab is to:

- review two of python's built-in sequence types (strings and lists) and NumPy array;
- iterate over sequences using the `for` loop; and
- investigate some of the built-in methods for strings.

The string type

Strings (type `str`) is python's type for storing text. A string is a sequence, but unlike other sequence types (`list`, `tuple`, and `numpy.ndarray`, for example) a string can only contain characters.

The type of a value ("object") determines what we can do with it: For example, we can do arithmetic on numbers, and we can ask for the length of a sequence (using the `len` function), but we cannot ask for the length of a number, or do much arithmetic with strings. Some operations can be done on several value types, but have different effects for different types. For example, using the `+` operator on two strings concatenates them: thus, `"1.234" + "1.234"` evaluates to `"1.2341.234"`, whereas adding two floating point numbers `1.234 + 1.234` evaluates to `2.468`.

Writing string literals

Recall that a literal is an expression that represents a constant value - like a `1` represents the integer one, or `"abc"` represents the three-letter string abc. String literals in python are written by enclosing them in either single, double or triple quotation marks:

```
In [1]: str1 = 'a few words'

In [2]: str2 = "a few words"

In [3]: str3 = '''a few words'''

In [4]: str1 == str2
Out [4]: ...

In [5]: str2 == str3
Out [5]: ...
```

(Note that the triple quote is written by typing three single quotes.) There is no difference between single- and double-quoted strings. The triple quote has the special ability that it can stretch over several lines:

```
In [1]: many_line_string = '''This is line 1.
This is the second line.
And this is line three.'''

In [2]: many_line_string
Out [2]: ...

In [3]: print(many_line_string)
...
```

Note the difference between how `many_line_string` is displayed when you evaluate the expression and when you print it. What is the `'\n'` character?

Remember that a string enclosed in one type of quotation marks can not contain the same type of quotation marks (unless they are escaped - see the lecture slides on strings), but can contain the other types.

Exercise 0

Write string literals for each of the following sentences:

- The possessive form of ‘it’ is ‘its’ - “it’s” is an abbreviation of “it is”.
- A ‘” is three ‘but two’ do not make a “.

Note that the first sentence should be on two lines.

Use the `print` function to verify that your strings are displayed correctly. Try writing them both with and without using triple quotes.

Character encoding

A string is a sequence (ordered collection) of characters.

Characters (like every other type of information stored in a computer) are represented by numbers. Interpreting a number as a character requires an *encoding*; python 3 uses the unicode standard for encoding characters in strings.

Python provides the `ord` and `chr` functions for translating between numbers and the characters they represent: ord(a_character) returns the corresponding character code (as an `int`) while chr(an_integer) returns the character that the integer represents.

Exercise 1

Try the following:

```
In [1]: ord('a')
Out [1]: ...

In [2]: ord('A')
Out [2]: ...

In [3]: chr(91)
Out [3]: ...

In [4]: chr(92)
Out [4]: ...

In [5]: chr(93)
Out [5]: ...

In [6]: chr(20986)+chr(21475)
Out [6]: ...

In [7]: chr(5798) + chr(5794) + chr(5809) + chr(5835) + chr(5840) + chr(5830) + chr(5825) + chr(5823)
Out [7]: ...
```

Remember that characters outside the ASCII range (unicode numbers above 255) may not display properly, if the computer you are using does not have a font for showing those characters. Also, many of the characters below number 32 are so called “non-printable” control characters, which may not be displayed.

Sequences

Remember that:

- `str`, `list` and `numpy.ndarray` are sequence types.
- A string (value of type `str`) can only contain characters, while a list or NumPy array can contain elements of any type - including a mix of elements of different types.

Exercise 2

Operations on python's built-in sequence types are summarised in this section of the python library reference.

To remind yourself what you can do with a sequence, run the following in the python shell:

```
In [1]: aseq = "abcd"

In [2]: type(aseq)          # 1. what type of sequence is this?
Out [1]: ...

In [3]: aseq + aseq         # 2. concatenation
Out [1]: ...

In [4]: aseq * 4            # 3. repetition
Out [1]: ...

In [5]: aseq[0]             # 4. Indexing
Out [1]: ...

In [6]: type(aseq[0])
Out [1]: ...

In [7]: aseq[-2]
Out [1]: ...

In [8]: aseq[1:-2]          # 5. Slicing
Out [1]: ...

In [9]: aseq[1:2]           # 5b.
Out [1]: ...

In [10]: bseq = "abdc"

In [11]: aseq < bseq        # 6. Comparison
Out [1]: ...
```

```
In [12]: for elem in aseq:    # 7. Iteration, using a for loop.
             print(elem)

In [13]: min(aseq)         # 8. built-in functions: min, max, sorted
Out [1]: ...

In [14]: max(aseq)
Out [1]: ...

In [15]: sorted(aseq)
Out [1]: ...
```

Next, try the operations above with a different sequence type, this time an array:

```
In [1]: aseq = numpy.array([1,2,3,4])

In [2]: bseq = numpy.array([1,3,2,4])

In [3]: type(aseq)        # 1. what type of sequence is this?
Out [1]: ...
                          # 2 - 8 as above
```

This experiment will show you some important differences, but also some similarities, between strings and arrays:

- The type of elements in a string (what is returned by indexing) is also `str`, while the type of the elements in an array is not an array (compare the results of test 4).
- Slicing a string returns a string, and slicing an array returns an array (compare the results of test 5).
- Arithmetic operations and comparison on arrays work element-wise (compare the results of tests 2, 3, and 6). For more about what you can do with arrays, read the section on slicing and array operations in Lab 4.

The built-in functions `min` and `max` and `sorted` are applicable to any sequence type (in fact, to any iterable type). Look them up using python's `help` function. What happens if you apply them to an empty sequence?

Iteration over sequences

Lab 4 introduced you to python's two kinds of loops: the `while` loop, which repeatedly executes a suite as long as a condition is true, and the `for` loop, which executes a suite once for every element of a sequence.

Exercise 3(a)

Write a function called `count_capitals` that takes a string argument and returns the number of capital (upper case) letters in the string. The function should look very similar to the one you wrote for Exercise 3(a) in Lab 4.

For this problem, you will need to determine if a letter is a capital. It will be helpful to know that in the unicode character encoding, the capital letters (of the English alphabet) are ordered sequentially; that is `ord('A') + 1 == ord('B')`, `ord('A') + 2 == ord('C')`, etc, up to `ord('A') + 25 == ord('Z')`. (Alternatively, have a look at the documentation of python's string methods; there are several that help you do things with letter case.)

Exercise 3(b) (*advanced*)

Is it possible to write a general function `count` that takes a sequence and some property X and counts how many elements in it have that property?

Hint: Functions are values in python. You can pass a function as an argument to another function.

Slicing

Python's built-in sequence types (which include type `str`) provide a mechanism, called slicing, to select parts of a sequence (that is, substrings if the sequence happens to be a string). It is done using the notation `sequence[start:end]`. There is also an extended form of slicing, which takes three arguments, written `sequence[start:end:step]`.

(Punch & Enbody's book has a detailed description of slicing, including its extended form, in Section 4.1.5 (page 183). Downey's book discusses slicing in Section "String Slices" in Chapter 8; the extended form of slicing is only briefly mentioned in Exercise 8-3.)

Exercise 4(a)

To make sure you understand what the arguments in a slicing expression mean, run through the following examples. For each expression, try to work out what the output will be before you evaluate the expression.

```
In [1]: my_string = "Angry Public Swamp Methods"

In [2]: L = len(my_string)

In [3]: my_string[1:L]
Out [3]: ...

In [4]: my_string[0:L - 1]
Out [4]: ...
```

```
In [5]: my_string[0:L:2]
Out [5]: ...

In [6]: my_string[L:0:-1]
Out [6]: ...

In [7]: my_string[6:6+6]
Out [7]: ...

In [8]: my_string[11:11-6:-1]
Out [8]: ...

In [9]: my_string[2*L]
Out [9]: ...

In [10]: my_string[0:2*L]
Out [10]: ...
```

("Angry Public Swamp Methods" - Why such a strange, non-sense string? It's designed to make it easier for you to see what happens when you try different slicing expressions. Can you see what's special about it? Hint: Remember from the earlier exercise that `ord('S')` is not equal to `ord('s')` - 'S' and 's' are different characters.)

Exercise 4(b)

The statement

```
In [1]: my_list = [i + 1 for i in range(26)]
```

assigns `my_list` a list of integers of the same length as the string used in the previous exercise. Try the slicing expressions on `my_list` instead of `my_string`. Is the result of all expressions what you expect?

String methods

A "method" is the same thing as a function, but it uses a slightly different call syntax. A method is always called on a particular object (value). A method call,

```
object.method(...)
```

can be thought of as "on this object, perform that method".

For now, we will just explore some of the rich set of methods that python provides for performing operations on built-in data types, such as strings. You can find the documentation of pythons string methods at http://docs.python.org/3/library/stdtypes.html#text-sequence-type-str, or by using the built-in help function in the python shell.

Try the following:

```
In [1]: sentence = "the COMP1730 lectures are boring, but the labs are great"

In [2]: sentence.capitalize()
Out [2]: ...

In [3]: sentence.swapcase()
Out [3]: ...

In [4]: sentence.count("e")
Out [4]: ...

In [5]: sentence.find("lectures")
Out [5]: ...

In [6]: sentence.find("exciting")
Out [6]: ...

In [7]: sentence.split(" ")
Out [7]: ...

In [8]: sentence.upper().find("LECTURES")
Out [8]: ...

In [9]: sentence.find("LECTURES").upper()
Out [9]: ...
```

After each method call, try `print(sentence)`. Did any of the methods modify the initial string?

Exercise 5

(From Punch & Enbody, Chapter 4: Question 14, on page 221.) There are five string methods that modify case: `capitalize`, `title`, `swapcase`, `upper` and `lower`.

- Look up each of these methods in the python on-line documentation or using the built-in `help` function. (Note: To find the documentation of a string method using help, you must prefix the method name with `str.`; that is, use `help(str.capitalize)` instead of `help(capitalize)`.)
- Based on your understanding, can you predict what will be the effect of each of these methods on the following strings:

```
s1 = "turner"
s2 = "north lyneham"
s3 = "AINSLIE"
s4 = "NewActon"?
```

To check if your understanding is correct, write python code to perform the operation, run it and see.

More debugging problems

Learning to read, understand and debug code is a very important skill, so here are some more debugging exercises.

Exercise 6(a)

Here is a function that is meant to count the number of repetitions of a substring in a string. For example, if the string is `'abcdabcf'` and the substring is `'ab'`, the count should be 2. (Of course, for any of the substrings `'a'`, `'b'`, `'c'`, `'ab'`, `'bc'` or `'abc'`, the count should be 2.)

```
def count_repetitions(string, substring):
    '''
    Count the number of repetitions of substring in string. Both
    arguments must be strings.
    '''
    n_rep = 0
    # p is the next position in the string where the substring starts
    p = string.find(substring)
    # str.find returns -1 if the substring is not found
    while p >= 0:
        n_rep = n_rep + 1
        # find next position where the substring starts
        p = string[p + 1:len(string) - p].find(substring)
    return n_rep
```

However, the function is not correct. For some inputs it will return the wrong answer, and for some inputs it will get stuck in an infinite loop. Find examples of arguments that cause these errors to happen.

Among the many string methods, there is one that does what this function is meant to do: `'abcdabcf'.count('abc')` will return 2. Can you fix the function above *without* using the string `count` method, i.e., keeping the idea of the original function and only correcting the error?

Exercise 6(b)

Here is another string function. This function is meant to remove all occurrences of a substring from a string, and return the result. For example, if the string is `'abcdabcf'` and the substring is `'bc'`, we want the function to return `'adaf'`.

```
def remove_substring_everywhere(string, substring):
    '''
    Remove all occurrences of substring from string, and return
    the resulting string. Both arguments must be strings.
    '''
    p = string.find(substring)
    lsub = len(substring) # length of the substring
    while p >= 0:
        string[p : len(string) - lsub] = string[p + lsub : len(string)]
        p = string.find(substring)
    return string
```

This function is also not correct, and should cause a runtime error on almost any input. Like in the previous problem, try to make the function do what it is supposed to while keeping the idea of the original function Can you also find a string method that does (or can be made to do) what this function is intended to do?

Programming problems

Note: We don't expect everyone to finish all these problems during the lab time. If you do not have time to finish these programming problems in the lab, you should continue working on them later (at home, in the CSIT labs after teaching hours, or on one of the computers available in the university libraries or other teaching spaces).

Ceasar cipher

A Ceasar cipher is based on simply shifting the letters in the alphabet by a fixed amount. For example we might do the following:

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   DEFGHIJKLMNOPQRSTUVWXYZABC
```

So each 'A' in the message is replaced by a 'D', each 'M' by a 'P', and so on. That is, there is a shift of 3 letters. Note that the alphabet wraps around at the end: An 'X' (third from the end) is replaced by an 'A', etc.

(a) Write a function that takes two arguments, a string to encrypt and a shift value (an integer) to use for encrypting it, and returns the encrypted string. Apply the same shift to both lower and upper case letters. Do not alter the non-alphabetical characters (like space, comma etc).

Examples:

- Encrypting "Et tu, Brutus!" with a shift of 3 should return "Hw wx, Euxwxv!".

  (Wikipedia has a long list of Latin phrases if you want to encrypt more examples in Latin.)

- Encrypting “IBM” with a shift of -1 should return “HAL”.

To decrypt a string, you can call the same function with the negative of the shift that was used to encrypt it. Also note that the function is invariant of the shift modulo 26: that is, a shift of 29 is the same as a shift of 3 (`3 == 29 % 26`), and a shift of `-1` is the same as a shift of 25 (`25 == -1 % 26`).

(b) To break the Caesar cipher you just need to guess the shift. Try the following three approaches:

- There are only 25 possible different shifts to decrypt the code. Write a function that takes an encrypted string and prints out the first five or so words decrypted using successively larger shifts (up to 25). See if you can guess based on this which is the right shift value. (How many words do you need to check to tell the right shift value from the others?)
- Repeat the above, but this time decrypt the entire message using increasing shift values. For each shift value search the decrypted message to find how many of the following 40 “common” three letter words exist:

  ```
  the,and,for,are,but,not,you,all,any,can,
  her,was,one,our,out,day,get,has,him,his,
  how,man,new,now,old,see,two,way,who,boy,
  did,its,let,put,say,she,too,use,dad,mom
  ```

  Return the shift value that gives the highest number of different three letter words. Does this automatic process succeed in breaking the cipher? (How did you deal with upper and lower case letters?)
- Next, consider the occurrence of individual letters. The most common letter in English is “e”, which occurs nearly 13% of the time. For the encrypted text, determine the most frequently occurring letter, assume it should be an “e”, from this determine the shift, and decrypt the message. Does this automatic process correctly break the cipher? (Again - how did you deal with upper and lower case?)

Tests Here are some encrypted strings that you can try your decryption methods on:

- `"’Awnhu pk neoa wjz awnhu pk xaz Iwgao w iwj dawhpdu, xqp okyewhhu zawz"’`

  (Note the use of triple quotes because the string contains a line break.)

-
  ```
  ’"Jcstghipcsxcv xh iwpi etctigpixcv fjpaxin du zcdlatsvt iwpi vgdlh ugdb iwtdgn,
  egprixrt, rdckxrixdc, phhtgixdc, tggdg pcs wjbxaxipixdc." (Gjat 7:  Jht p rdadc
  puitg pc xcstetcstci rapjht id xcigdsjrt p axhi du epgixrjapgh, pc peedhxixkt, pc
  pbeaxuxrpixdc dg pc xaajhigpixkt fjdipixdc.  Ugdb Higjcz & Lwxit, "Iwt Tatbtcih du
  Hinat".)’
  ```

  (Note that the string is enclosed in single quotes because it contains double quotes.)

-
  ```
  "Cywo cmsoxdscdc gybu cy rkbn drobo sc xy dswo vopd pyb cobsyec drsxusxq.  (kddbsledon
  dy Pbkxmsc Mbsmu)"
  ```

You should also make up your own tests (use the encryption function you wrote to encrypt sentences, then test if your cipher-breaking functions find the correct shift!)

Pig Latin

Pig Latin is a game of alterations played on words. To translate an English word into Pig Latin, the initial consonant sound is transposed to the end of the word and an “ay” is affixed. Specifically, there are two rules:

- If a word begins with a vowel, append “yay” to the end of the word.
- If a word begins with a consonant, remove all the consonants from the beginning up to the first vowel and append them to the end of the word. Finally, append “ay” to the end of the word.

For example,

- dog => ogday
- scratch => atchscray
- is => isyay
- apple => appleyay

Write a function that takes one argument, a string, and returns a string with each word in the argument translated into Pig Latin.

Hints:

- The `split` method, when called with no additional arguments, breaks a string into words, and returns the words in a list.
- Slicing is your friend: it can pick off the first character for checking, and you can slice off pieces of a string and use string concatenation (the `+` operator) to make a new word.
- Making a string of vowels allows use of the `in` operator: `vowels="aeiou"` (how do you make this work with both upper and lower case?)
- Test your function with a diverse range of examples. Your tests should cover all cases (for example, test words beginning with a vowel and words beginning with a consonant). Pay particular attention to edge cases (for example, what happens if the word consists of just one vowel, like “a”? what happens if the string is empty?).

Floating point error analysis (*advanced*)

In the lecture on functions (in week 2) we mentioned a way of approximating the derivative of a function f at a point x, by the slope of a straight line through f(x - d) and f(x + d). More precisely, the formula is (f(x + d) - f(x - d)) / (2 * d).

As the distance d tends to zero, we expect this approximation to grow closer to the real derivative f’(x). However, this fails to take into account the floating point round-off error in the calculation of f(x + d) and f(x - d), which may become larger relative to the size of 2 * d.

To measure this effect, we can compare the approximation with the true derivative, for cases where the latter is known. For example, if f(x) = ex (which is available in python as `math.exp`), we know that the derivative is f'(x) = f(x).

(a) Write a function that computes the approximation of f'(x), with a parameter for the distance d. As python allows you to pass functions as arguments, you can write a (simple) function that does this calculation for any function f and point x.

(b) Write another function that calculates the error as the absolute difference between the approximate and true derivative, for given values of x and d, using the exponential function as f.

Generate a series of diminishing values for d, from, say, 0.1 down to 10-15. You can do this using NumPy as follows:

```
ds = np.power(10.0, np.arange(-1,-16,-1))
```

Alternatively, you can also do the same thing with list comprehension and the `range` function:

```
ds = [10 ** -i for i in range(1,16)]
```

Calculate and plot the error for each d-value in this range. What can you observe?

(c) Try the same exercise with some other functions. f(x) = x2 is an interesting case to test because its derivative is a linear function.