Lab 5

S2 2017

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 introduce the third built-in sequence type, tuples.
- Contrast a reference to an object with a copy of an object, in the context of lists.

Sequence types: strings and lists

Remember that:

- `str` and `list` are sequence types.
- A string (value of type `str`) can only contain characters, while a list can contain elements of any type - including a mix of elements of different types in the same list.
- Strings are immutable, while list is a mutable type. (What does mutable mean? This is explained in Section Section 7.6 (pages 310 to 321) of Punch & Enbody's book, and Chapter 10 in Downey's book.)

Exercise 0

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]: ...

In [16]: aseq[-1] = 'z'     # 9. Element assignment

In [17]: aseq
Out [1]: ...
```

Next, try operations 1-9 above with a different sequence type, this time a list:

```
In [1]: aseq = [1,2,3,4]

In [2]: bseq = [1,3,2,4]

In [3]: type(aseq)        # 1. what type of sequence is this?
Out [1]: ...
                          # 2 - 9 as above
```

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

- Slicing a list returns a list (test 5b).
- Strings are immutable, but lists are mutable (test 9).
- Ordering comparisons are defined the same way for lists as for strings.

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?

Exercise 1

There are several ways to create lists. For each of the following, write a small piece of python code to create that list. Try finding at least two different ways of making each of the lists.

- Create a list containing 100 elements that all have the same value, e.g. all are 1. Try to find three different approaches. Discuss your three alternatives with your neighbour - did you find the same three, or different ones?
- Create a list of 100 integers whose value is the same as their index (position in the list) plus 1, that is, `mylist[0] == 1`, `mylist[1] == 2`, `mylist[2] == 3`, etc.
- Create a list of 10 lists of increasing length. The first list should be empty, i.e., `len(mylist[0]) == 0`; the second list should have one element, so `len(mylist[1]) == 1`; and so on.

Just like strings, the list data type has several methods that can be helpful in solving these problems. For example, read `help(list.append)`, `help(list.insert)` and `help(list.extend)` to see the documentation of some of the methods that modify lists. Modifying methods and operations for mutable sequence types (i.e., list) are summarised in this section of the python documentation.

Extra programming problem (optional)

Write three functions that *return* each of the three different lists. Your functions should take as argument the length of the list to be constructed in each case. Important: Every time you call the function it should return a "fresh" list, that does not share any references with the other lists.

List comprehension (optional)

Python has a mechanism for writing compact expressions that create lists, called *list comprehension*. The general form of a comprehension is

iterable_exp should be an expression that evaluates to an iterable type (for example, a sequence), and the comprehension creates a list whose elements are the result of evaluating element_exp for each element in the iterable value. The variable varname is assigned each value from the iterable in turn, and can be used in the element_exp. For example,

```
[ 2 ** k for k in [0, 1, 2, 3, 4] ]
```

will create the list with elements `2 ** 0`, `2 ** 1`, etc, up to `2 ** 4`, i.e, the list `[ 1, 2, 4, 8, 16 ]` and

```
[ ord(char) for char in "Hello!" ]
```

will create a list whose elements are the numbers corresponding to the encoding of the characters in the string `"Hello!"`.

You can find out more about them in Section "List comprehensions" of Chapter 19 in Downey's book, or Section 7.10 in Punch & Enbody's book.

Try writing expressions to generate the lists in the exercise above using comprehensions.

Mutable objects and references

In python, every value computed by the interpreter is an *object*. An object in python has:

- An identity: This is a number assigned by the python interpreter when an object is created. You can access this number using the `id` function - you saw some examples of this in Exercise 1 of lab 2. Examining the identity of an object is typically not useful in the programs you write, but it will sometimes help you to understand what is happening.

- Some attributes: This is information about the object. An attribute that every object has is a *type*, which tells us (and the python interpreter) what kind of object it is. Other attributes tell us something about the "content" of the object (for example, if the object is of type `int`, that is, an integer, what integer it is).

When we assign a value to variable, the variable name is associated with a *reference* to the object; that is, essentially, the objects identity. Several variables can refer to the same objects.

A *mutable* object is one that can be modifed. This means that we can change the object's attributes. The object's identity remains unchanged.

Downey's book describes mutable objects (specifically, lists) and aliasing (multiple names referring to the same object) in Chapter 10. Punch & Enbody's book describes it in Section 7.6 (pages 310 to 321).

Exercise 2(a)

The following code attempts to construct a table containing the first three rows of the periodic table. Run the following commands in the python shell:

```
In [1]: row1=["H","He"]
In [2]: row2=["Li","Be","B","C","N","O","F","Ar"]
In [3]: row3=["Na","Mg","Al","Si","P","S","Cl","Ne"]
In [4]: ptable=row1
In [5]: ptable.extend(row2)
In [6]: ptable.extend(row3)
```

- What is the value of `ptable` now?
- What is the value of `row1` now? Is this what you expected?
- Correct the error in `row2` (Ar should be Ne) by executing a command of the form `row2[?] = ?`. (The question mark means it is for you to figure out what is the right index to change. You should *not* write a literal `?`.) What happens to `ptable` as a result of this assignment?
- Correct the error in `row3` (Ne should be Ar) by executing a command of the form `ptable[?] = ?`. What happens to `row3` as a result of this assignment?

To help explain what is happening, use the `id` function to see the identities of the objects referenced by each of the variables:

```
In [7]: id(row1)
Out [7]: ...
In [8]: id(row2)
Out [8]: ...
In [9]: id(row3)
Out [9]: ...
In [10]: id(ptable)
Out [10]: ...
```

You should find that `row1` and `ptable` both reference the same object (a list), but that the contents of `row2` and `row3` were copied into `ptable`.

If you are uncertain about any of the above, ask your tutor. This may also be a good time to try stepping through and visualising the example using the on-line tool pythontutor.com: Copy the code you want to test into the text box and click the "Visualize Execution" button. Remember to make sure you have selected "Python 3.x" in the menu above where you enter the code.

Exercise 2(b)

Now execute the following commands (make sure you redefine the rows):

```
In [1]: row1 = ["H","He"]
In [2]: row2 = ["Li","Be","B","C","N","O","F","Ar"]
In [3]: row3 = ["Na","Mg","Al","Si","P","S","Cl","Ne"]
In [4]: ptable = [row1]
In [5]: ptable.append(row2)
In [6]: ptable.append(row3)
```

- What is the value of ptable now? How does this differ from what you had in Exercise 2(a)?
- What is the value of `row1` now? How does it differ from what you had in Exercise 2(a)?
- To get the first element of the first row from `ptable`, you would use the expression `ptable[0][0]`. Write the expressions to get the sixth element from the second row ("O") and the second element from the third row ("Mg"). Use only the `ptable` variable, not `row2` or `row3`.
- Correct the error in row 2 (Ar should be Ne) by executing a command of the form `row2[?] = ?`. Does this also change `ptable`?
- Correct the error in row 3 (Ne should be Ar) by executing a command of the form `ptable[?][?] = ?`. Does this change `row3`?

Again, you can use the `id` function to see the identities of the objects involved (try both `id(ptable)` and `id(ptable[0])`). You should find that each element in `ptable` is a reference to one of the row lists.

Again, if you want to visualise what is going on, try pythontutor.com.

Shallow and deep copies

In Exercise 2(a) you assigned `ptable` the list referenced by `row1` via the statement `ptable = row1`. Subsequent operations showed that `ptable` and `row1` referenced the same object (list). How do we actually make a copy of a list?

Exercise 2(c)

Execute the following commands:

```
In [1]: row2=['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ar']
In [2]: ptable1=["H","Xe",row2]
In [3]: ptable2=ptable1[:]
```

- Is `ptable1` a list, a list of lists or a mix of both? (Print it out!)
- Correct element "Xe" in `ptable2` to be "He". Is the change propagated to `ptable1`?
- Correct element "Ar" in `ptable2` to be "Ne". Is the change propagated to `ptable1` and to `row2`?

You should find that after executing `ptable2=ptable1[:]` the first two elements of `ptable2` are copies of those in `ptable1` but the third element is a reference to `row2`. The effect of adding `[:]` to the `ptable2=ptable1` assignment is to create a so-called shallow copy of `ptable1`.

Exercise 2(d)

Execute the following commands

```
In [1]: row2=['Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ar']
In [2]: ptable1=["H","Xe",row2]
In [3]: import copy
In [4]: ptable2=copy.deepcopy(PT1)
```

- Now correct the "Ar" to "Ne" by assigning the right element of `ptable2`. Inspect to see whether the elements of `ptable1` or `row2` have changed. (You should find no changes.)

As mentioned last week's lecture, you should never use deepcopy: it is much slower and consumes more memory than shallow copy; it may also cause problems with circular references (such as list containing a reference to itself). Any problems with shared references can always be avoided by thinking more carefully about how you have structured your code.

Introducing tuples

As mentioned at the start of the lab, python has a third built-in sequence type, called `tuple`. Like a list, tuples can contain any type of value, including a mix of value types. The difference between type `list` and type `tuple` is that tuples are immutable.

To create a tuple, you only need to write its elements separated by commas:

```
In [1]: aseq = 1,2,3,4

In [2]: type(aseq)
Out [2]: ...
```

It is common practice to write a tuple in parentheses, like this:

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

In [2]: type(aseq)
Out [2]: ...
```

and the python interpreter will usually print tuples enclosed in parentheses. However, remember that it is the comma that creates the tuple, not the parentheses! To see this, try the following:

```
In [1]: x = (1)

In [2]: type(x)
Out [2]: ...

In [3]: x = 1,

In [4]: type(x)
Out [4]: ...
```

There is one exception: To create an empty tuple (a tuple with length zero), you have to put a comma between nothing and nothing! Typing `x = ,` does not work. Instead, use an empty pair of parentheses to create an empty tuple, like this: `x = ()`.

Exercise 3

Now, retry the tests you did in Exercise 0, with tuple values:

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

In [2]: bseq = 1,3,2,4

In [3]: type(aseq)        # 1. what type of sequence is this?
Out [3]: ...
                          # 2 - 9 as before.
```

Element assignment (test 9) should give you an error, because the tuple is immutable.

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).

Pop and slice

The list method `pop(position)` removes the element in the given position from the list (read `help(list.pop)` or the on-line documentation).

(a) Write a function `allbut(a_list, index)`, which takes as arguments a list and an index, and returns a *new list* with all elements of the argument list (in the order they were) except for the element at `index`. The argument list should *not be modified* by the function.

Example:

```
In [1]: my_list = [1,2,3,4]

In [2]: my_short_list = allbut(my_list, 2)

In [3]: print(my_short_list)
[1, 2, 4]

In [4]: print(my_list)
[1, 2, 3, 4]
```

(b) A slice expression, `a_list[start:end]`, returns a new list with the elements from `start` to `end - 1` of the list.

Write a function `slice_in_place(a_list, start, end)`, which takes as arguments a list and two indices, and modifies the argument list so that it is equal to the result of the slice expression `a_list[start:end]`. The function should *not* return any value.

Example:

```
In [1]: my_list = [1, 2, 3, 4]

In [2]: slice_in_place(my_list, 1, 3)

In [3]: print(my_list)
[2, 3]
```

Advanced: Make your `slice_in_place` function work with both positive and negative indices (like slicing does). Add default values for the `start` and `end` parameters such that the function behaves the same as the slicing expresssion when only one of the two parameters is given.

List shuffle

Sorting a list puts the elements in order; shuffling it puts them in (more or less) disorder. The python module `random` implements a function called `shuffle` which randomly shuffles a list. Here, we will look at a deterministic (non-random) shuffle of a list. A "perfect shuffle" (also known by many other names) cuts

the list in two parts, as evenly sized as possible, then interleaves the elements of the two parts to form the shuffled list.

(a) Write a function `perfect_shuffle(a_list)` which takes as argument a list and returns the perfect shuffle of the list. The function should *not* modify the argument list.

Example:

```
In [1]: my_list = [1, 2, 3, 4, 5, 6]

In [2]: my_shuffled_list = perfect_shuffle(my_list)

In [3]: print(my_shuffled_list)
[1, 4, 2, 5, 3, 6]

In [4]: print(my_list)
[1, 2, 3, 4, 5, 6]
```

(b) Write a function `perfect_shuffle_in_place(a_list)` which takes as argument a list and performs the perfect shuffle on the list. The function should modify the list, and *not* return any value. After the function call, the argument list should have the same length and contain the same elements; only the order of them should change.

Example:

```
In [1]: my_list = [1, 2, 3, 4, 5, 6]

In [2]: perfect_shuffle_in_place(my_list)

In [3]: print(my_list)
[1, 4, 2, 5, 3, 6]
```

Advanced: Write a program that repeatedly shuffles a list using this function and counts how many shuffles are done before the list becomes equal to the original list.

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!)

Nested lists

A list that contains lists is sometimes called *nested*. We define the *nesting depth* of a list as follows:

- A list that does not contain any list has a nesting depth of zero
- A list that contains lists has a nesting depth equal to the maximum nesting depth of the lists it contains, plus one.

Note that "a list that contains lists" can also contain values that are not lists.

For example, the nesting depth of `[[1,2], [2,4]]` is 1, while the nesting depth of `[1, [2], [[3], [[4], 5]]]` is 3.

Write a function that takes as argument a list and returns its nesting depth.

What does your function return when called with the list `[[[]]]`? (and is that what it should return?)

2-dimensional NumPy arrays

A *diagonal* matrix is a square (n x n) matrix `M` such that only elements on the diagnonal (where row equals column, that is, `M[i,i]`) are non-zero. This can be generalised: Say a matrix `M` has *diagonality* d if `M[i,j] != 0` only in positions such that the absolute difference between i and j is strictly less than d. (Thus, a diagonal matrix has diagonality 1.) For example, the matrix in Figure 1 has diagonality 3.

(a) Write a function `diagonality(matrix)` which takes as argument a matrix and returns the smallest d such that `matrix` has diagonality d. (Assume that `matrix` is a square 2-dimensional NumPy array.)

(b) The diagonality of a matrix can sometime be decreased by rearranging the order of rows and columns. For example, the matrix

```
array(1, 2, 0, 0,
      0, 0, 5, 6,
      3, 4, 0, 0,
      0, 0, 7, 8)
```

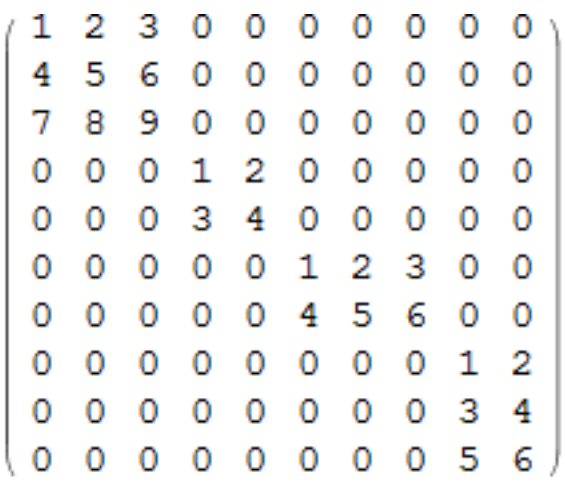

Figure 1: a matrix with diagonality 3

has diagonality 3, but swapping the two middle rows results in the matrix

```
array(1, 2, 0, 0,
      3, 4, 0, 0,
      0, 0, 5, 6,
      0, 0, 7, 8)
```

with diagonality 2.

First, work out how to swap a pair of rows, or a pair of columns in a matrix. (Hint: You can use slicing and slice assignment. Remember that NumPy arrays allow for more types of indexing and slicing than other python sequence types.) You probably need to use some temporary variable to hold one of the vectors being swapped. Remember that slicing NumPy arrays does not (always) return a copy, even a shallow one.

Next, write a function that attempts to minimise the diagonality of a given matrix by repeatedly swapping rows or columns. A simple idea is to look at the result of every possible swap and check if has lower diagonality. However, this may not always be enough, as several swaps may be necessary to reduce the diagonality.