Lab 02: Sequences and Iteration

Contents

Objectives

By the end of this lab you will understand:

- The difference between strings and lists
- How iteration works

By the end of this lab you will be able to:

- Perform various operations on strings and lists
- Iterate through sequences
- Handle program errors

1 Strings

Last week, we saw strings as a basic type that could be printed; e.g., using `print 'hello, world!'`. As a reminder, strings can start and end with either single quotes `''` or double quotes `""`. The individual components of strings are called *characters*. Longer strings can also be spread across multiple lines, in which case they must start and end with triple quotes `'''''`, as in the following example:

```
long_str = '''line 1
line 2
line 3'''
```

If you enter this command in a file `long_quote.py` followed by `print long_str`, and then run the file, you'll see it print in an expected format; i.e., you'll see:

```
> python long_quote.py
line 1
line 2
line 3
```

If instead you enter it into the interactive interpreter, and then retrieve the value of `long_str`, you'll see that it looks a bit different:

```
>>> long_str = ‘‘‘line 1
... line 2
... line 3’’’
>>> long_str
‘line 1\nline 2\nline 3’
```

The special character `\n` is used to create a new line (the equivalent of pressing Enter in a regular text editor), which is why printing this string looks the way it does. Another special character you may be interested in is `\t`, which is used to create a tab.

While printing strings is already a useful feature, it turns out there are many other built-in functions and operators one can perform on a string `mystr`. These include:

- `len(mystr)`: returns the length of the string; i.e., the number of characters it contains. For example, `len(‘this’)` returns 4.
- `mystr1 + mystr2`: returns the *concatenation* of the two strings; i.e., a new string consisting of `mystr1` followed by `mystr2` (with no space in between). This operator is not limited to two strings: it can be used to concatenate an arbitrary number of them. To concatenate the same string `n` times with itself, you can use `mystr * n`. For example, `‘hello, ’ + ‘world!’` returns `‘hello, world!’`
- `mystr[i]`: returns the character at the i-th position in `mystr`. In Python, as in most other programming languages, this type of *index* starts at 0 (meaning the first character in the string is at index 0, not 1) and ends at `len(mystr)-1`. For example, `‘this’[0]` returns `‘t’` and `‘this’[-2]` returns `‘i’`.
- `x in mystr`: returns a boolean representing whether or not the character `x` is contained within the string `mystr`. For example, `‘e’ in ‘this’` returns `False`.
- `mystr[i:j]`: returns the *slice* (or subsection) of the string starting at index i and ending at index $j-1$. Among other things, this means that `mystr[0:len(str)]` returns the whole string. For example, `‘hello, world’[1:5]` returns `‘ello’`.
- `mystr.index(x)`: returns the index of the *first* occurrence of the character `x` within `mystr`. For example, `‘hello, world’.index(‘o’)` returns 4.
- `mystr.count(x)`: returns the number of times `x` appears within `mystr`. For example, `‘hello, world’.count(‘o’)` returns 2.

Task 1. Create a file `strings.py` and write code to try out these six operators. In particular, think about edge cases that may return somewhat unusual answers (e.g., `len(‘’)`), concatenate several different numbers of strings, retrieve the characters or slices at many different indices, and count characters in the cases where they're both in and not in the string. Continue the good practices of commenting your code, and printing both the expected and the actual values (you may find that the `\t` character is helpful in lining these two up, for a quick comparison) when testing.

Task 2. Using slices, write a program `clip.py` that defines a string `mystr` and prints a new string consisting of the first three and last three characters of `mystr` (as always, test it on several edge cases). Then define two strings `mystr1` and `mystr2`, and create and print a string consisting of the two strings concatenated, with a space in between, but with the first three characters of each string swapped (e.g., if `mystr1 = ‘abcd’` and `mystr2 = ‘wxyz’` then the string you print should be `‘wxyd abcz’`).

2 Lists

Another Python type that in some ways behaves a lot like a string is a *list*, which has type `list`. Syntactically, lists are denoted using square brackets `[]` and every element in a list must be separated using a comma. For example, a simple list is `[1,2,3,4]`, and `[]` is the empty list. Lists can be assigned to variables just as any other type.

2.1 The basics

Task 3. To see the similarities between strings and lists, create a file `lists.py` and test out the operations introduced above, just as you did with strings.

Unlike strings, lists are a *container* type, which is one of the most useful objects in Python (or really in any programming language). The main thing that sets lists and other containers apart from strings is *mutability*: like integers and all the other types we've seen so far, strings are meant to be static or *immutable* objects that do not change over time. So, once you assign a variable to a string, as in `x = 'hello'`, the only way to change the value of `x` is to re-assign it (again, just as with an integer or other numeric type).

Containers, on the other hand, are *mutable*, as they represent a collection of objects, and that collection is allowed to change over time. This gives rise to several additional operations that change the object itself and thus work on lists (and other container types we'll see later on in the module) but not strings. Examples of these are:

- `mylist[i] = x` replaces the value that was previously at `mylist[i]` with `x`. This is an example of mutating `mylist`, as it now has a different value after this *item assignment*.
- `mylist.append(x)` adds `x` to the end of `mylist`.
- `mylist.remove(x)` removes the first occurrence of `x` within `mylist`.
- `mylist.extend(mylist2)` extends `mylist` by adding all the entries in `mylist2` to the end of it. This operation is strictly different from concatenation, which returns a new list, as again it actually mutates the value of `mylist`.
- `mylist.reverse()` reverses the order of the elements in `mylist`; i.e., the element that was first becomes last, etc.
- `mylist.sort()` sorts the elements in `mylist` according to some built-in comparator in Python.

In contrast to all of the functions we've seen so far, all of these functions have one thing in common, which is that they do not return any value. Just like an assignment statement, they affect the binding of a variable, rather than creating and returning a new value that is unbound to any variable.

Task 4. Extend the file `lists.py` to test out these additional operators, keeping in mind that they do not return any value and instead result in successive changes to the list. As always, think about edge cases that may return somewhat unusual answers, combine lists containing items of different types, and repeat operations. Continue the practice of testing by printing both the expected and actual results.

To confirm that these operations do not work on strings, try a few of them out, either by extending the `strings.py` file or by using the interactive interpreter.

2.2 Lists of many types

Python is different from most programming languages in the sense that lists can contain objects of many different types, such as `[1,'a',2.5,True]`; in other languages with more rigid type systems, lists must contain a single type. It is recommended, however, that lists do contain objects of a single type, and we will see later in the module a container class that is more typically used in this *heterogenous* way.

If you think of all the types you're aware of so far, there is one you may not have thought of as something you could put in a list: another list! Lists of lists are in fact commonly used in Python, and can be thought of as providing multiple dimensions.

Using lists of lists allows us to index several times, as in

```
>>> list_of_lists = [[1,2,3],[4,5,6],[7,8,9]]
>>> list_of_lists[1]
[4, 5, 6]
>>> list_of_lists[1][2]
6
```

This would also work with lists of strings, since strings are also indexable. So, casting `str` as first `list` and then `int`, we'd get similar answers if we ran

```
>>> list_of_strs = [‘123’,‘456’,‘789’]
>>> list(list_of_strs[1])
[‘4’, ‘5’, ‘6’]
>>> int(list_of_lists[1][2])
6
```

In general, you can use `list(mystr)` to turn the string `mystr` into a list of its individual characters. To go back, you can use the function `mystr.join(mylist)`, which returns a concatenation of the elements in `mylist`, separated by `mystr` (as long as every element in `mylist` is of type `str`). For example,

```
>>> ‘+’.join([‘1’,‘2’,‘3’])
‘1+2+3’
```

In particular, `‘’.join(mylist)` can turn a list of strings into a single string (with no separation between the characters).

2.3 Lists of numbers

While lists can contain arbitrary types, it turns out that if all of the objects in a list are of a numeric type (`int`, `float`, etc.), Python has several built-in functions to operate on these lists. Some examples are:

- `sum(mylist)` returns the sum of all the numbers in `mylist`.
- `min(mylist)` returns the minimum number in `mylist`.
- `max(mylist)` returns the maximum number in `mylist`.

Task 5. Using `sum`, create a file `average.py` in which you print the average of the values in a list. Experiment with multiple lists, and as always incorporate good practices (testing, edge cases, comments, etc.).

Another special type of numeric list is a *range*. The `range` function is an easy way to specify a numeric range, so `range(1,5)` returns the list `[1, 2, 3, 4]`, and in general `range(x,y)` returns a list containing all numbers greater than or equal to `x` and (strictly) less than `y`. If you want to skip numbers, by (for example) including only every other number in the range, you can provide an optional third argument to the function; e.g., `range(1,10,2)` returns `[1, 3, 5, 7, 9]`.

In Python3, `range` is not a list, but rather an *iterable* (an object that we'll learn more about later); this is done for efficiency. So, when you evaluate something like `range(1,5)`, you just see `range(1,5)` rather than the expanded list, but the way you interact with ranges is the same.

Checkpoint 1. You have reached a checkpoint! Please call the PGTA or lecturer over to check what you've done so far.

3 For Loops

If the main thing separating strings and lists is the idea of mutability, then what is it that makes them so similar? Both of them are examples of *sequences*, or *iterable* types. In both `‘1234’` and `[1,2,3,4]`, for example, there is a well defined number of objects and an ordering over those objects.

We have already seen that indexing and slicing allow us to pull out subsequences of both of these types. To walk through the sequence object by object, however, we need to use something called a *loop*. Loops are one of the most fundamental components of all non-functional programming languages, and allow you to repeatedly execute a series of statements.

3.1 Syntax

We'll see another example of loops next week, but for this week we're going to focus solely on `for` loops. A `for` loop has three main parts: the variable name `v`, the sequence `seq`, and the *body*. If `seq` is a variable name, then it must have been assigned already to a sequence before the loop, but `v` should be undefined, in which case it takes on meaning only within the body. The syntax is then:

```
for v in seq:
    <body line 1>
    <body line 2>
    ...
```

Again, `for` loops are most useful for walking through sequences of objects, and executing the same lines of code (the body) every time. For example, to print all elements in a list, we would run:

```
>>> for x in [1,2,3]:
...     print x
...
1
2
3
```

Effectively, this loop goes through the sequence `[1,2,3]` one step at a time, re-assigning `x` to the next item in the list at each iteration, and then running the statement in the body on `x`.

3.2 Exercises

Task 6. Create a file `len_sum.py`. Using a `for` loop, recreate the built-in Python functions `len` and `sum` by writing code to print out the length of both lists and strings (test on both) and print out the sum of a list of numbers. Hint: when writing something like `x = x + 1`, Python allows you to write `x += 1` as shorthand (or *syntactic sugar*) instead.

Task 7. To iterate through a list `mylist`, we often use the syntax `for x in mylist`, as you probably did in the previous task (with maybe a different variable name). To iterate through a list `mylist` in a slightly different way, add more code to `len_sum.py` that defines a range associated with `mylist`, and then repeat the process of computing the sum by iterating through `mylist` using that range. Extend this process to compute the sum of every second element in `mylist` too.

4 Error Handling

We already saw in lectures (and possibly during labs) how easy it is to encounter errors while programming; e.g., if we type `10 / 0` then we get a `ZeroDivisionError`, and if we type `x` before it is defined we get a `NameError`. Indeed, if you've been doing this lab properly by testing out many edge cases, you should already have encountered many additional errors yourselves. For example, we get:

- `IndexError` if we try to index past the end of a list or string (try `'abc'[100]`);
- `ValueError` if we try to find the index of something that isn't there (try `'abc'.index('d')`) or remove something that isn't there (try `[1,2,3].remove(4)`);
- `TypeError` if we try to concatenate objects of different types (try `[1,2,3] + 'abc'`);
- `AttributeError` if we try to mutate a string using a list operation (try `'abc'.append('d')`); and
- `IndentationError` if we don't indent the body of a `for` loop.

While our goal as good programmers is ultimately to avoid these errors by writing good programs, it is helpful to know how to *handle* them anyway, just in case we do encounter them. This is particularly helpful when we want to test out many different edge cases (some of which will return errors) without terminating the program.

Error handling is relatively simple in Python, and involves the keywords `try` and `except`. The code in the `try` block (which should be indented) is the code you want to run, which should behave as normal until (or *except*) it encounters an error. At this point, the program will run the code in the `except` block (which should again be indented), and then keep going. For example, if we run the following program

```
try:
    mylist = [1,2,3,4]
    for x in range(0,6):
        print mylist[x]
except:
    print 'error: went past end of list'

print 'hello, world!'
```

then we'll get

```
1
2
3
4
error: went past end of list
hello, world!
```

This type of error handling thus allows us to still tell the programmer that an error occurred, but also allows the program to continue running. In fact, the code that gets run can depend on the error, as in:

```
>>> try:
...     mylist = [1,2,3,4]
...     for x in range(0,6):
...             print mylist[x]
...     mylist.remove(5)
... except IndexError:
...     print 'index error: went past end of list'
... except ValueError:
...     print 'value error: tried to remove value not in list'
...
1
2
3
4
index error: went past end of list
```

In this case, the code in the `try` block had both types of errors, so the program ran the code of the `except` block it found first. The ordering of the `except` blocks can thus be used to prioritize one type of error over another, and the inclusion of the error type means you can let the program terminate on some types of errors and continue on others.

Task 8. Go back to `strings.py` and `lists.py` and add in error handling for any edge cases that caused errors (or, if you didn't have any before, add some in now).

5 Putting It All Together

Until relatively recently, sequencing the human genome was an enormously computationally expensive process, and was thus done rarely and behind closed doors. Today, advances in computing

have made this process significantly cheaper, and as a result DNA sequences are now made available via publicly accessible databases. There are many opportunities this provides to researchers studying genetics and related areas, including to crime scientists interested not only in the use of DNA as forensic evidence but also in understanding if specific genes can act as indicators of specific patterns of behavior.

As you might know, a single strand of DNA consists of four nucleobases: adenine (A), thymine (T), guanine (G), and cytosine (C), and is structured as a double helix, or as two strands. Strands of DNA are oriented in opposite directions (i.e., they are *antiparallel*), and due to their molecular structure each base binds with one other base: A binds with T, and G binds with C (and vice versa). The other base with which one base binds is called its *complement*.

In this lab, we'll be taking a strand of DNA and computing the antiparallel strand (the reverse), the strand with which it binds (the complement), and the antiparallel strand with which it binds (the reverse complement).

Task 9. Create a file `dna.py` and create in it a variable `dna` that represents some strand of DNA (feel free to start with something relatively short, for testing purposes, but eventually it would be good to use something long enough to illustrate the speed of the program).

First, write code to compute `reverse_dna`, or the reverse strand. **Do not** do this using `mylist.reverse()`; instead, find another way to reverse the string. Don't get discouraged if you find this difficult! Hint: You will probably still find it easier to use lists (because they are mutable), to index into the lists using negative indices (e.g., using `dna[-1]` to access the last item in `dna`), and to use ranges. To test your code, you can check it against `dna[::-1]` (one bonus point if you can explain, in a comment, why this reverses `dna`).

Next, write code to compute `comp_dna`, or the complementary strand. Here you will need to find some way to represent the mapping from A to T and from G to C (and vice versa). There is no easy way to compute the expected result, so you'll have to do it by hand (or leave out the expected results if your strands get too long).

Finally, combine the techniques you used for the previous two components to write code to compute `reverse_comp_dna`, or the reverse complementary strand. Again, there is no easy way to compute the expected results, but rather than doing it by hand you can use one of the many online tools available for this (e.g., the one at `reverse-complement.com/`) to check your answers.

As an example of a successful run, if you define `dna = 'atgcaat'`, then running your program should print out the following:

```
reverse should be   taacgta
reverse is          taacgta
------------
complement should be    tacgtta
complement is           tacgtta
------------
reverse complement should be    attgcat
reverse complement is           attgcat
```

6 Handing In

As you did last week, feel free to play around some more with all of the concepts we've covered in this lab. Once you are done, add all `.py` files into a `lab02` directory; this should mean adding:

- `strings.py`
- `clip.py`
- `lists.py`
- `average.py`
- `len_sum.py`
- `dna.py`

Push these files, along with `partner.txt` indicating who you worked with, to the remote repository, and you're done! Please check with one of us on your way out to ensure that you've done this correctly.