Semester 2, 2019: Lab 4

Lab 4

Note: This lab has more problems than we expect everyone to finish during the lab time. If you do not have time to finish all problems, you can continue working on them later (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 across campus), or return to them in a later lab.

Objectives

The purpose of this week's lab is to:

- understand the indexing of (1-dimensional) sequences;
- do some computations over sequences that require iterating through them using loops; and
- practice reading and debugging code.

Exercise 0: Reading and debugging code

The following are attempts to define a function that takes three (numeric) arguments and checks if any one of them is equal to the sum of the other two. For example, `any_one_is_sum(1, 3, 2)` should return `True` (because `3 == 1 + 2`), while `any_one_is_sum(0, 1, 2)` should return `False`.

(a) All of the functions below are incorrect. For each of them, find examples of arguments that cause it to *return* the wrong answer.

Function 1

```
def any_one_is_sum(a,b,c):
    sum_c=a+b
    sum_b=a+c
    sum_a=b+c
    if sum_c == a+b:
        if sum_b == c+a:
            if sum_a == b+c:
                return True
    else:
        return False
```

Function 2

```
def any_one_is_sum(a,b,c):
    if b + c == a:
        print(True)
    if c + b == a:
        print(True)
    else:
        print(False)
    return False
```

Function 3

```
def any_one_is_sum(a, b, c):
    if a+b==c and a+c==b:
        return True
   else:
        return False
```

(b) For each of the three functions above, can you work out how they are *intended* to work? That is, what was the idea of the programmer who wrote them? What comments would be useful to add to explain the thinking? Is it possible to fix them by making only a small change to each function?

Exercise 1: Debugging loops

Below are two attempted solutions to the problems of summing the odd and even digits in a number (from Lab 3), respectively. Both of them, however, have some problems: For some arguments, they may return the wrong answer, or not return at all because the loop never ends.

(a) For each of the two functions, find arguments that cause it to return an incorrect answer, and arguments that cause it to get stuck in an infinite loop (either or both may be possible). Arguments to the function must (of course!) be non-negative integers.

Hint: Add `print` calls *inside* the loop to see what is happening. Print the variables that appear in the loop condition, so you can see if they are changing or not (if they are not, then the loop is stuck).

```
def sum_odd_digits(number):
    dsum = 0
    # only count odd digits
    while number % 2 != 0:
        # add the last digit to the sum
        digit = number % 10
        dsum = dsum + digit
        # divide by 10 (rounded down) to remove the last digit
        number = number // 10
    return dsum

def sum_even_digits(number):
    m = 1 # the position of the next digit
```

```
    dsum = 0 # the sum
    while number % (10 ** m) != 0:
        # get the m:th digit
        digit = (number % (10 ** m)) // (10 ** (m - 1))
        # only add it if even:
        if digit % 2 == 0:
            dsum = dsum + digit
        m = m + 1
    return dsum
```

(b) Like in the previous problem, can you work out how the functions are *intended* to work? That is, what was the idea of the programmer who wrote them? What comments would be useful to add to explain the thinking? Is it possible to fix the errors you uncovered in your testing by making only a small change to each function?

(c) (*advanced*) Here is a more complex function:

```
def mystery(m):
    assert type(m) is int and m >= 0, "m must be a non-negative integer"
    while m > 1:
        i = 2
        while i < m:
            while m % i == 0:
                m = m // i
            i = i + 1
    return i
```

(It is called `mystery` because it's not very obvious what it does.)

The mystery function is meant to work on any non-negative integer input, but some arguments will cause it to get stuck in an infinite loop. Can you find examples of arguments that will cause this?

Learning to read and debug code is a very important skill (and it will also show you the value of good naming and commenting!). There are more debugging exercises towards the end of the lab.

Sequence types

We have already seen a number of times that all values in python have a *type*, such as `int`, `float`, `str`, etc. To determine the type of a value we can use the function `type(_some expression_)`. python has three built-in sequence types: lists (type `list`), strings (type `str`) and tuples (type `tuple`). These sequence types are used to represent different kinds of ordered collections.

To write a list literal, write its elements, separated by commas, in a pair of square brackets:

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

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

The elements that you write can be expressions. These are evaluated, and the resulting values become the elements of the list:

```
In [3]: my_list = [2, 2 + 1, 2 * 2, 2 + 3]

In [4]: my_list
Out [4]: ...
```

The NumPy array type

The NumPy library provides a type for represening n-dimensional arrays of values (usually numbers, but also other types, such as Booleans), and functions for doing calculations with arrays.

NumPy is not part of python's standard library, but it is installed on the CSIT lab computers, and other computers across campus. (If you want to be able to use your own python setup, you have to ensure that you have NumPy installed. The Anaconda distribution includes NumPy, SciPy and matplotlib by default, which is one reason why we recommend it. Read the guide to setting up python for more information.)

Like any library (module) in python, to use NumPy you must first import it:

```
In [1]: import numpy
```

Remember that the names of all functions in the imported module are prefixed with the module name. That is, you have to write `numpy.linspace(...)` instead of just `linspace(...)`. When you import a module, you can give it a shorthand name. For example, if your write

```
In [1]: import numpy as np
```

the functions in the NumPy module will be prefixed with just `np` instead of `numpy`. In all the examples below, we will assume you have imported NumPy with the abbreviation `np`.

Information about the functions that NumPy provides is available through the built-in help system. However, you can also find documentation of NumPy and SciPy on-line; the on-line documentation can be easier to navigate. You can also find some tutorials at numpy.org.

We can create an array from a list:

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

In [2]: my_array = np.array(my_list)

In [3]: my_array
Out [3]: ...

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

or simply:

```
In [5]: np.array([3, 1.2, -2])
Out [5]: ...
```

As shown in the lectures, there are also several functions for creating arrays with specific contents:

```
In [6]: np.zeros(10)
Out [6]: ...

In [7]: np.ones(10)
Out [7]: ...

In [8]: np.linspace(-2, 2, 21)
Out [8]: ...

In [9]: np.arange(4,9)
Out [9]: ...
```

`linspace(from, to, num)` returns an array of `num` floating point numbers evenly spaced between `from` and `to`.

`arange(from,to)` returns an array of consecutive integers, starting with `from` and ending at `to - 1`. If you provide just one argument, as in `arange(to)`, the starting number defaults to 0.

Indexing sequences

Both `list` and NumPy's `ndarray` are called sequence data types. Every element in a sequence has an *index* (position). The first element is at index 0. The *length* of a sequence is the number of elements in the sequence. The index of the last element is the length minus one. The built-in function `len` returns the length of any sequence.

Indexing a sequence selects a single element from the sequence (for example, a character if the sequence is a string). Python also allows indexing sequences from the end, using negative indices. That is, `-1` also refers to the last element in the sequence, and `-len(seq)` refers to the first.

Exercise 2(a)

This exercise is to play with the `list` sequence type. Execute the following in the python shell. For each expression, try to work out what the output will be before you evaluate the expression.

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

In [2]: my_list[1]
Out [2]: ...

In [3]: my_list[4]
Out [3]: ...

In [4]: my_list[-1]
Out [4]: ...

In [5]: L = len(my_list)
```

```
In [6]: my_list[L - 1]
Out [6]: ...

In [7]: my_list[1 - L]
Out [7]: ...
```

They should all run without error. Is the result of each expression what you expected?

Exercise 2(b)

As we saw above, you can turn a list into a NumPy array:

```
In [1]: my_array =  np.array(my_list)
```

You can now try the indexing expressions that you did with `my_list` above on `my_array` instead. Is there any difference in the result?

Exercise 2(c)

What does the following statement do?

```
In [1]: my_array = np.arange(1,7)
```

Iteration over sequences

Python has two kinds of loop statements: 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. (To be precise, the `for` loop works not only on sequences but on any type that is *iterable*. All sequences are iterable, but later in the course we will see examples of types that are iterable but not sequences.)

Both kinds of loop can be used to iterate over a sequence. Which one is most appropriate to implement some function depends on what the function needs to do with the sequence. The `for` loop is simpler to use, but only allows you to look at one element at a time. The `while` loop is more complex to use (you must initialise and update an index variable, and specify the loop condition correctly) but allows you greater flexibility; for example, you can skip elements in the sequence (increment the index by more than one) or look at elements in more than one position in each iteration.

In the lectures so far, we have only used `while` loops, and they are sufficient to solve all the problems in this lab. The syntax and execution of the `for` loop is described in the text books (Downey: Section "Traversal with a for loop" in Chapter 8; Punch & Enbody: Sections 2.1.4 and 2.2.13).

Exercise 3(a)

The following function takes one argument, a sequence, and counts the number of elements in it that are negative. It is implemented using a `while` loop.

```
def count_negative(sequence):
    count = 0
    index = 0
    while index < len(sequence):
```

```
        if sequence[index] < 0:
            count = count + 1
        index = index + 1
    return count
```

Note that the function will work on any sequence type (e.g., both `list` and `array`), as long as it contains only numbers.

Rewrite this function so that it uses a `for` loop instead.

To test your function, you can use the following inputs:

- `[-1, 0, -2, 1, -3, 2]` (3 negative numbers)
- `np.linspace(2, -2, 5)` (2 negative numbers)
- `np.arange(5) - 3` (3 negative numbers)
- `np.sin(np.linspace(0, 4*np.pi, 50))` (25 negative numbers)
- `np.zeros(10)` (0 negative numbers)

You can create more test cases by making variations of these, or using other array-creating functions.

Exercise 3(b)

Write a function called `is_increasing` that takes a sequence (of numbers) and returns `True` iff the elements in the array are in (non-strict) increasing order. This means that every element is less than or equal to the next one after it. For example,

- for `[1, 5, 9]` the function should return `True`
- for `[3, 3, 4]` the function should return `True`
- for `[3, 4, 2]` the function should return `False`

Is it best to use a `for` loop or a `while` loop for this problem? (Note: Downey describes different solutions to a very similar problem in Section "Looping with Indices" in Chapter 9.)

Test your function with the examples above, and with the examples you used for exercise 3(a).

Also test your function on an empty sequence (that is, a list or array with no elements). An empty list can be created with the expression `[]` (and an empty array with `np.array([])`). Does your function work? Does it work on a sequence with one element?

Exercise 3(c)

The average (or mean) of a sequence of numbers is the sum of the numbers divided by the length of the sequence. You can calculate the average of a sequence of numbers using python's built-in function `sum` (which works on any sequence type, as long as it contains numbers), using the NumPy function `np.mean` (if you convert the sequence to an array first), or writing your own function using a loop over the sequence (as was shown in the lecture).

Write a function `most_average(numbers)` which finds and returns the number in the input that is *closest* to the average of the numbers. (You can assume that the argument is a sequence of numbers.) By closest, we mean the one that has the smallest absolute difference from the average. You can use the built-in function `abs` to find the absolute value of a difference. For example, `most_average([1, 2, 3,`

`4, 5])` should return `3` (the average of the numbers in the list is `3.0`, and `3` is clearly closest to this). `most_average([3, 4, 3, 1])` should also return `3` (the average is `2.75`, and `3` is closer to `2.75` than is any other number in the list).

More debugging problems

Exercise 4

Here is a function that is meant to return the position (index) of a given element in a sequence; if the element does not appear in the sequence, it returns the length of the sequence. For example, `find_element([3,2,1,4], 1)` should return `2`, since that is the index where we find a `1`.

```
def find_element(sequence, element):
    i = 0
    while sequence[i] != element:
        if i < len(sequence):
           i = i + 1
        i = i + 1
    return i
```

However, the function is not correct. For some inputs it will cause a runtime error. Find an example of arguments that cause an error to occur. Can you correct the error without introducing another?

Programming problems

Note: These are more substantial programming problems. We do not expect that everyone will finish them within the lab time. If you do not have time to finish them during 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).

Closest matches

(a) Write two functions, `smallest_greater(seq, value)` and `greatest_smaller(seq, value)`, that take as argument a sequence and a value, and find the smallest element in the sequence that is greater than the given value, and the greatest element in the sequence that is smaller than the given value, respectively.

For example, if the sequence is `[13, -3, 22, 14, 2, 18, 17, 6, 9]` and the target value is `4`, then the smallest greater element is `6` and the greatest smaller element is `2`.

- You can assume that all elements in the sequence are of the same type as the target value (that is, if the sequence is an array of numbers, then the target value is a number).
- You can *not* assume that the elements of the sequence are in any particular order.
- You should not assume that the sequence is of any particular type; it could be, for example, a NumPy array, a list, or some other sequence type. Use only operations on the sequence that are valid for all sequence types.
- What happens in your functions if the target value is smaller or greater than all elements in the sequence?

(b) Same as above, but assume the elements in the sequence are sorted in increasing order; can you find an algorithm that is more efficient in this case?

Counting duplicates

If the same value appears more than once in a sequence, we say that all copies of it except the first are *duplicates*. For example, in `array(-1, 2, 4, 2, 0, 4)`, the second 2 and second 4 are duplicates; in the string "Immaterium", the 'm' is duplicated twice (but the 'i' is not a duplicate, because 'I' and 'i' are different characters).

Write a function `count_duplicates(seq)` that takes as argument a sequence and returns the number of duplicate elements (for example, it should return 2 for both the sequences above). Your function should work on any sequence type (for example, both arrays and strings), so use only operations that are common to all sequence types. For the purpose of deciding if an element is a duplicate, use standard equality, that is, the `==` operator.

Putting stuff in bins

A *histogram* is way of summarising (1-dimensional) data that is often used in descriptive statistics. Given a sequence of values, the range of values (from smallest to greatest) is divided into a number of sections (called "*bins*") and the number of values that fall into each bin is counted. For example, if the sequence is `array(2.09, 0.5, 3.48, 1.44, 5.2, 2.86, 2.62, 6.31)`, and we make three bins by placing the dividing lines at 2 and 4, the resulting counts (that is, the histogram) will be the sequence `2, 4, 2`, because there are 2 elements less than 2, 4 elements between 2 and 4, and 2 elements > 4.

(a) Write a function `count_in_bin(values, lower, upper)` that takes as argument a sequence and two values that define the lower and upper sides of a bin, and counts the number of elements in the sequence that fall into this bin. You should treat the bin interval as open on the lower end and closed on the upper end; that is, use a strict comparison `lower < element` for the lower end and a non-strict comparison `element <= upper` for the upper end.

(b) Write a function `histogram(values, dividers)` that takes as argument a sequence of values and a sequence of bin dividers, and returns the histogram as a sequence of a suitable type (say, an array) with the counts in each bin. The number of bins is the number of dividers + 1; the first bin has no lower limit and the last bin has no upper limit. As in (a), elements that are equal to one of the dividers are counted in the bin below.

For example, suppose the sequence of values is the numbers 1,..,10 and the bin dividers are `array(2, 5, 7)`; the histogram should be `array(2, 3, 2, 3)`.

To test your function, you can create arrays of random values using NumPy's random module:

```
In [1]: import numpy.random as rnd
In [2]: values = rnd.normal(0, 1, 50)
```

This creates an array of 50 numbers drawn according to the normal distribution with mean 0 and standard deviation 1. The following creates 10 evenly sized bins covering the range of values:

```
In [1]: import numpy as np
In [2]: range = np.max(values) - np.min(values)
```

```
In [3]: dividers = (np.arange(1, 10) * (range / 10)) + np.min(values)
```

As you increase the size of the value array, you should find that the histogram becomes more symmetrical and more even.

You can also test your function by comparing it with the histogram function provided by NumPy (see `help(numpy.histogram)`).

(*Advanced*) Slicing and array operations

Python's built-in sequence types and NumPy's `array` provide a mechanism, called slicing, to select parts of a sequence. 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.) We will come back to slicing in future labs, so if you do not have time to cover it this week, that's ok.

To understand what the arguments in a slicing expression mean, you can try the following examples:

```
In [1]: my_array = np.arange(0,20)

In [2]: L = len(my_array)

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

In [4]: my_array[0:L - 1]
Out [4]: ...

In [5]: my_array[0:L:2]
Out [5]: ...

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

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

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

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

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

All of the slice expressions above should work the same on a list with the same elements.

NumPy arrays also support two generalised forms of indexing:

- If `i` is an array of integers, `a[i]` returns an array with the elements of `a` at the indices in `i`. All values in `i` must be valid indices for `a`, that is, between `0` and `len(a) - 1`.
- If `i` is an array of Boolean values `a[i]` returns an array with the elements of `a` at positions where the value in `i` is `True`. The length of `i` must be equal to that of `a`.

Note that these forms of indexing do not work on other python sequence types (such as lists).

Arithmethic operators (`+`, `-`, `*`, `**`, `/`, `//`, `%`), comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) and bit-wise logical operators (`&` for "and", `|` for "or" and `~` for "not") can all be applied to NumPy arrays. They all perform their operation *element-wise*. That is, if `a` and `b` are two arrays, `c = a + b` is another array such that `c[i] = a[i] + b[i]` for `i` in the range `0` to `len(a) - 1`. If `a` is an array and `b` is a non-array value (such as an `int` or a `float`), the operation is done between each element of `a` and `b`; again, the result is an array. For the operators that take two arguments (all except `~` and unary `-`) if they are applied to two arrays, these have to be of the same length. Comparison operators ('==', `!=`, `<`, `>`, `<=`, `>=`) applied to arrays return arrays of Boolean values.

Exercise 5(a) (*advanced*)

Use the `arange` function, together with array operations, to write expresssions that construct the following arrays:

- An array with integers `0`, .., `n`, followed by `n - 1`, .., `0`.
- An array of `n` alternating `1`'s and `-1`'s, starting with a `1`.
- An array of `n` alternating `1`'s and `-1`'s, starting with a `-1`.

It may be helpful to know that:

- When an arithmetic operation (such as `+` or `*`) is done on an array of Boolean values, these are converted to integers (`False == 0` and `True == 1`).
- If `a` and `b` are two arrays, the call `np.concatenate((a, b))` returns an array that contains the elements of `a` followed by the elements of `b` (and thus its length is `len(a) + len(b)`). Note that there is an extra pair of parentheses around `a,b` in the function call; this is not a typo.
- The function `np.repeat` can also be useful. Look it up using the help system!

Exercise 5(b) (*advanced*)

As was demonstrated in last week's lecture, with NumPy arrays it is sometimes possible to write complex operations very compactly. For example, counting negative values (as in Exercise 3(a) above) can be done with

```
def count_negative(sequence):
    return np.sum(np.array(sequence) < 0)
```

Can you use slicing, array operations and other NumPy functions to implement the `is_increasing` and `most_average` functions (on arrays) in a similar way?