Lab 4

S2 2017

Lab 4

Note: There may be more exercises in this lab than you can finish during the lab time. If you do not have time to finish all exercises (in particular, the programming problems), you can 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.

Objectives

The purpose of this week's lab is to:

- understand the indexing of (1-dimensional) sequence types, such as arrays and strings;
- write functions using loops (`for` and `while`) to solve problems that require iterating through sequences;
- investigate some of the functions available (in python and in NumPy) for doing things with different sequence types.

The string type

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_)`. 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 string (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`.

Exercise 0

Run the following in the python shell:

```
In [1]: string1 = "1.234"
In [2]: print(string1)
...

In [3]: type(string1)
Out [4]: ...

In [5]: string2 = string1 + string1

In [6]: print(string2)
...

In [7]: len(string1)
Out [7]: ...

In [8]: len(string2)
Out [8]: ...

In [9]: float1 = float(string1)

In [10]: type(float1)
Out [10]: ...

In [11]: float1 + float1
Out [11]: ...
```

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 containt the same type of quotation marks (unless they are escaped - see the lecture slides on strings), but can contain the other types. (If you happen to like video tutorials better than lecture slides, here are two elementary ones on strings: http://www.youtube.com/watch?v=jECazLigEH8 and http://www.youtube.com/watch?v=yjRkEX4OTyc. The second one also looks at some string methods, which show up a little later in this lab.)

Exercise 1

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.

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(21475)+chr(20986)
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.

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.

As shown in the lectures, there are several functions for creating arrays:

```
In [1]: np.array([3, 1.2, -2])
Out [1]: ...

In [2]: np.zeros(10)
Out [2]: ...

In [3]: np.ones(10)
Out [3]: ...

In [4]: np.linspace(-2, 2, 21)
Out [4]: ...

In [5]: np.arange(4,9)
Out [5]: ...
```

`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

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)

Run the following examples. For each expression, try to work out what the output will be before you evaluate the expression.

```
In [1]: my_string = "as if on"

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

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

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

In [5]: L = len(my_string)

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

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

Exercise 2(b)

The statement

```
In [1]: my_array = np.arange(1,9)
```

assigns the variable `my_array` an array of the integers 1,2,...,8. (Thus, the length of this array is the same as the length of the string used in the previous exercise.) Try the indexing expressions from the previous exercise on `my_array` instead of `my_string`. They should all work. Is the result of all expressions what you expect?

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.

The syntax and execution of `while` and `for` loops is described in the lectures, and in the text books (Downey: Sections "The while Statement" in Chapter 7 and "Traversal with a for loop" in Chapter 8; Punch & Enbody: Sections 2.1.4, and 2.2.10 to 2.2.13).

Exercise 3(a)

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

```
def count_negative(array):
    count = 0
    index = 0
    while index < len(array):
        if array[index] < 0:
            count = count + 1
        index = index + 1
    return count
```

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

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

- `np.linspace(-2, 2, 4)` (2 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 `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 the previous exercise.

To do this, 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(c)

Write a function called `is_increasing` that takes an array (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 `np.array([1, 5, 9])` the function should return `True`
- for `np.array([3, 3, 4])` the function should return `True`
- for `np.array([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 array (that is, an array with no elements). An empty array can be created with the expression `np.array([])`. Does your function work? Does it work on an array with one element?

Try calling your function with a string argument (for example, `"a B c D"`). Does it work? Does it return a correct answer? (Remember that character comparison is based on the corresponding unicode number: `a < B` is the same as `ord('a') < ord('B')`.)

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 previous exercise that `ord('S')` is not equal to `ord('s')` - 'S' and 's' are different characters.)

Exercise 4(b)

As in Exercise 2(b) above, use the statement

```
In [1]: my_array = np.arange(1,27)
```

to assign `my_array` an array of integers of the same length as the string used in the previous exercise, and try the slicing expressions on `my_array` instead of `my_string`. Is the result of all expressions what you expect?

Operations on arrays

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.

Run the following:

```
In [1]: a = np.linspace(0, 2, 5)

In [2]: a
Out [2]: ...

In [3]: b = np.ones(5)

In [4]: b
Out [4]: ...

In [5]: a - b
Out [5]: ...

In [6]: a - 1
Out [6]: ...

In [7]: b / a
Out [7]: ...

In [8]: 1 / a
Out [8]: ...

In [9]: i = np.arange(5)

In [10]: i
Out [10]: ...

In [11]: i < 3
Out [11]: ...

In [12]: i[::-1] < 3
Out [12]: ...
```

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`. (Of course, 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 strings).

Exercise 5

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!

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)`. Which of the methods modify the string? Why?

Exercise 6

(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.

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 "qazrfvujm" and the target value is 'h', the smallest greater element is 'f' and the greatest smaller element is 'j'.

- You can assume that all elements in the sequence are of the same type as the target value (that is, if the sequence is a string, then the target value is a letter; 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 string, 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)`).

A statistical test for correlation.

(*Note*: This problem has a lot of background material, some of which is quite mathematical. If it is too much for you, you can always skip it and take the next problem.)

In statistics, a paired sample is a set of samples where each sample measures two different quantities ("variables", in stats terminology) of the same "individual". For example:

- The individuals may be students, and the two quantities measures could be their final grade in two different courses.

- The two time series (IR range sensor and tachometer) that were used in the example in last week's lecture can be seen as a paired sample: the "individuals" are points in time, and the two quantities that are measured for each individual are the two sensor values.
- The "individuals" could be instances of a computational problem and the two quantities measured can be the running times of two different algorithms that both solve the problem.

A statistical test for correlation is used to determine if the sample provides any evidence for a relationship between the quantities. That is, does the sample "prove" (in a statistical sense) something like the statement that "students who do well in subject X are more likely to do well in subject Y"?

One of the very simplest statistical tests for correlation is the binomial test. It is based on dividing the space of possible measurement points into four quadrants, and counting how many samples fall in each quadrant.

Let's call the two measured values `x` and `y`, and assume they are numbers. That way, we can look at them as points on a plane. For example, if our sample is

```
(0, 2.09), (1, 0.5), (2, 3.48), (3, 1.44), (4, 5.2), (5, 2.86), (6, 2.62), (7, 6.31)
```

we will represent it with the two 1-dimensional arrays

```
x = np.array([0, 1, 2, 3, 4, 5, 6, 7])
y = np.array([2.09, 0.5, 3.48, 1.44, 5.2, 2.86, 2.62, 6.31])
```

and use the indices 0 through 7 as the names of the "individuals".

(Note: This assumes you have executed `import numpy as np`, as described earlier in the lab.)

The boundaries of each quadrant are defined by the *median* of the two sets of measurements. The median of a list of values is (roughly speaking) a value `M` such that half the elements in the list are less than `M` and half are greater than `M`. (When finding the median of a list of numbers, it is usually taken to be one of the numbers in the list, or the average of the two numbers that appear in the middle when the list is sorted.) NumPy has a function that computes the median of an array (of numbers):

```
median_of_x = np.median(x)
```

For the example above, `np.median(x)` returns 3.5 and `np.median(y)` returns 2.74. Plotting the data points and the median lines, we have:

The quadrants are numbered 1 to 4, counter-clockwise, starting from the bottom left. (This numbering is arbitrary.) This means an individual `i` belongs to quadrant 1 if `x[i]` is less than the x-median and `y[i]` is less than the y-median, to quadrant 2 if `x[i]` is greater than the x-median and `y[i]` is less than the y-median, and so on.

If there is no correlation between the two quantities, we would expect the points to be scattered roughly evenly over all four quadrants (because the boundaries are chosen so that half the points are on either side of the x-median and half the points are on either side of the y-median). If, on the other hand, there

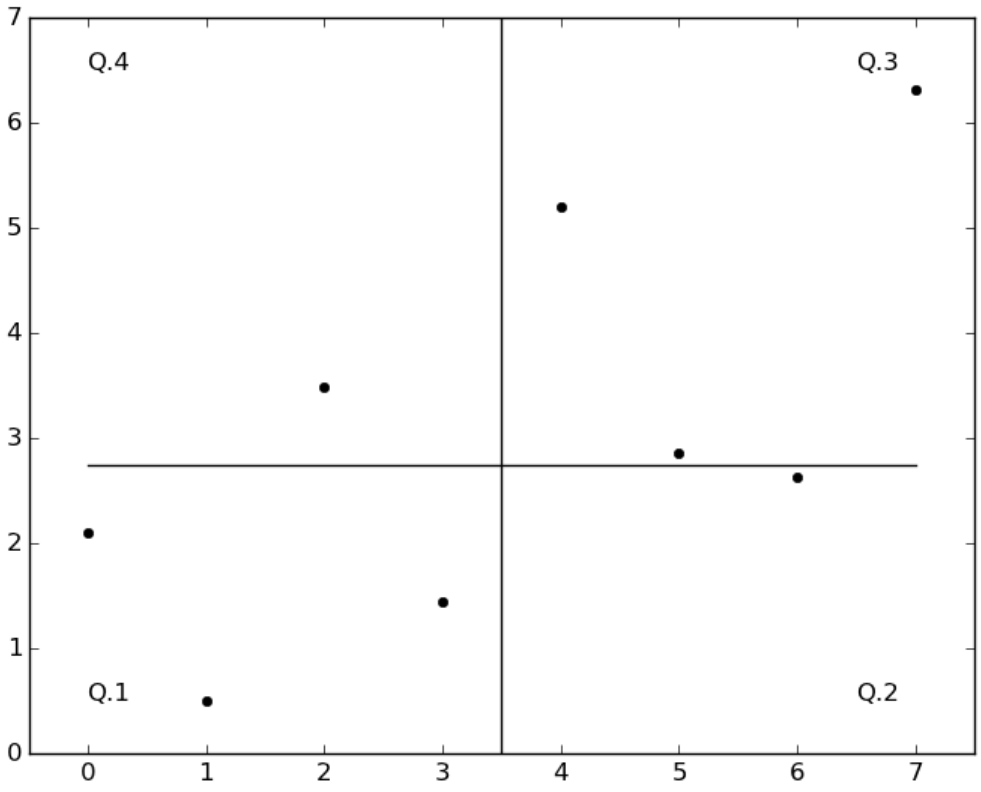

Figure 1: example data plotted, showing quadrants

is a positive correlation (that is, large x-values tend to go together with large y-values more often than with small y-values), the sum of the number of points in quadrants 1 and 3 (the positive diagonal) should outweigh the sum of the number of points in quadrants 2 and 4 (the negative diagonal); if there is a negative correlation (that is, large x-values tend to go together with small y-values more often than with large y-values), the number of points in quadrants 2 and 4 should outweigh the number of points in quadrants 1 and 3.

(a) Write a function `count_by_quadrants(x, y)` that takes as argument two arrays (of the same length) containing the two measurements of a paired sample, and returns the number of sample points (individuals) in quadrants 1 and 3. For example, called with the two arrays `x` and `y` in the example above, your function should return 6.

You can then obtain the number of sample points in quadrants 2 and 4 by subtracting the function value from the size of the sample (the length of the arrays); in the example, it is 2.

(b) The probability of the observed outcome under the hypothesis that there is no correlation is given by the binomial probability mass function with parameters n equal to the size of the sample and p = 0.5.

Specifically, to compute this probability, compute the function

$$\left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{(n-k)}$$

, where

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

. The factorial function (n!) is defined as n * (n - 1) * (n - 2) * ... * 2 * 1. The k in the formula is the number of sample points in quadrants 1 and 3 (that is, the value computed by your function above).

If this probability is very small, we can say it is highly unlikely that there is no correlation between the two quantities, and therefore that it is more likely that there is a correlation. In the example above, we have the values `k = 6` and `n = 8`, resulting in a probability of approximately 0.11. So it's unlikely, but not impossible, that there is no correlation.

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. (Lists were only briefly mentioned in last week's lecture. The list is another built-in sequence type in python. This means you can do all the things you normally do with a sequence also on a list; in particular, you can index it, and iterate over it with a `for` loop.)
- 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?).

Text segmentation

The `str.split` method mentioned in the previous problem does not consider punctuation. For example, applied to the text in the last bullet point of the list of hints above, it will return the list

```
['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?).']
```

Note how punctuation marks, parentheses and quotation marks have "stuck" to the words.

Write a function that takes as argument a string and returns the string with everything that is not letters removed. Your function should leave whitespace (spaces, `' '`, tabs, `'\t'` and newlines, `'\n'`) in place, so that applying the `split` method to the resulting string returns the right list of words.