Lists: Heart Rate Example

Now that you have seen the basic mechanics of lists, let us take a look at an example of using lists, and some of the things we might do. We are going to work a few examples and problems around the idea of heart rate data over time. Here, we have time series data, where someone's heart rate is measured and recorded every second. We are going to start with some examples related to working out, but you might also want to analyze heart rate data for medical diagnoses. Of course, heart rate data is not the only kind of time series data—you can see many examples of it in a variety of settings. One very common and major example is stock prices in the stock market.

Let's get started with our heart rate time series data in this video.

Heart Rate Introduction

As the video asked you to, please take a moment to work out Steps 2, 3, and 4 for the "time in heart rate zone" problem. In particular we want to develop an algorithm that has three inputs:

1. ***data:*** a list of heart rate measurements, one per second
2. ***low:*** the lower bound of the range (inclusive)
3. ***high:*** the upper bound of the range (exclusive)

The algorithm should return an answer that is how many seconds of the data were within the specified range. When you have finished steps 2 and 3, try your algorithm out on these inputs:

1. ***data:*** [140, 145, 150, 155, 152, 148, 140]
2. ***low:*** 145
3. ***high:*** 152

Did your algorithm give you an answer of 3? If so, then you can be more confident in it! If not, go back and see if you can figure out what went wrong and how to fix it.

Now that we have an algorithm in English, we want to turn it into Python. I am going to show you how to turn my algorithm into Python. If you have a different algorithm that works, that is ok—there are many different ones that are right!

My algorithm is:

> Start with `count` being 0
> For each element x in `data`
> If x is greater than or equal to `low` and x is less than `high`
> Increment `count`
> Your answer is `count`

As usual, we'll start with the function definition, and put our algorithm as comments

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   # For each element x in data
        # If x is greater than or equal to low and x is less than high
               # Increment count
   #Your answer is count
```

Hopefully the first line is familiar by now: we want to keep track of a value with a name, so we use a variable:

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   count = 0
   # For each element x in data
        # If x is greater than or equal to low and x is less than high
               # Increment count
   #Your answer is count
```

The next line asks us to do something for each element of data. You recently learned how to do that with "for x in data"

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   count = 0
   # For each element x in data
 for x in data:
        # If x is greater than or equal to low and x is less than high
               # Increment count
   #Your answer is count
```

The next line is an if statement. We want to make sure both of two conditions are true: (x >= low) and (x < high). In Python, we just use the word "and" for this:

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   count = 0
   # For each element x in data
 for x in data:
        # If x is greater than or equal to low and x is less than high
    if (x >= low) and (x < high):
               # Increment count
   #Your answer is count
```

Then we want to increment count, which is just count = count + 1. We'll also put a pass statement to indicate the end of our `if`, and also the end of our `for`. Remember that while these pass statement are not strictly needed, it is good practice to add them and they can help you avoid a lot of pain if you change your code later:

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   count = 0
   # For each element x in data
 for x in data:
        # If x is greater than or equal to low and x is less than high
    if (x >= low) and (x < high):
               # Increment count
       count = count + 1
       pass
    pass
   #Your answer is count
```

Now we just need to give back our answer with a return statement:

```
def count_time_in_range(data, low, high):
   # Start with count being 0
   count = 0
   # For each element x in data
 for x in data:
        # If x is greater than or equal to low and x is less than high
    if (x >= low) and (x < high):
               # Increment count
       count = count + 1
       pass
    pass
   #Your answer is count
  return count
```

Now, you should go try this out in Python. You can test it with:

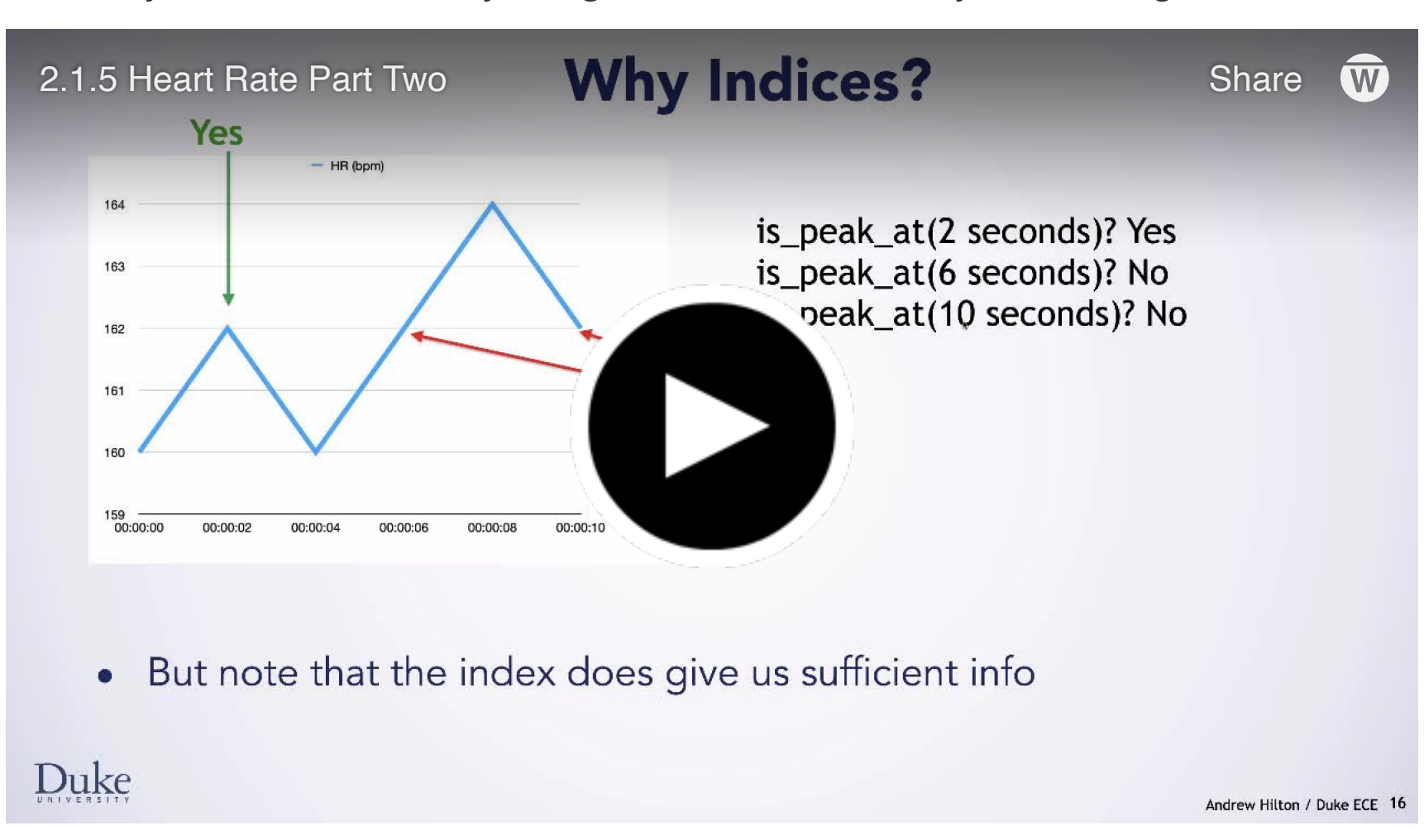
`count_time_in_range([149, 151, 152, 150, 153, 155, 157, 164, 165, 167, 169], 150,165)`

Heart Rate Lists, Part One

Next, let's see a slightly different example. We want to count how many times the person working out went up and down the stairs. We can use the heart rate patterns to tell when they were going up (heart rate increasing) and down (heart rate decreasing).

Now take a moment to work out steps 2 and 3 to come up with your algorithm for the problem of counting how many trips up and down the stairs, by counting peaks in the data. Remember that we aren't going to do `is_peak_at` yet, we will just assume that we will write it correctly. This algorithm will take one input, `data`, which is the list of heart rate data as before.

Now that you have done devised your algorithm, let's turn it into Python code together:



Do you notice any similarities between this problem and the `count_time_in_range` problem we did earlier? Take a moment to reflect on similarities in the algorithm and code between the two problems. What is similar about them and what is different? Why are they so similar? Can you think of a general class of problems that would all have the same basic structure?

Heart Rate Lists, Part Two

Our last task here is to write the `is_peak_at` function. Let's take a look at what is involved in that, then you will go write and test that function as an assignment.

You will want to make use of list indexing for this problem. We discussed list indexing in an earlier video, but if you need a quick refresher, you can do `data[num]` where num is some numerical expression. This could be, for example `data[0]`, which is the first element of the list, or `data[i]`, where `i` is some variable whose value is a number, or an arithmetic expression like `data[i+1]` or `data[i+j]` if `i` and `j` are both variables with numerical values.

Day 6, Part 2

Now it is time to complete **10_hr_peaks** on the Mastery Learning Platform. Login to the course server, go to **10_hr_peaks** in your git repository, and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.

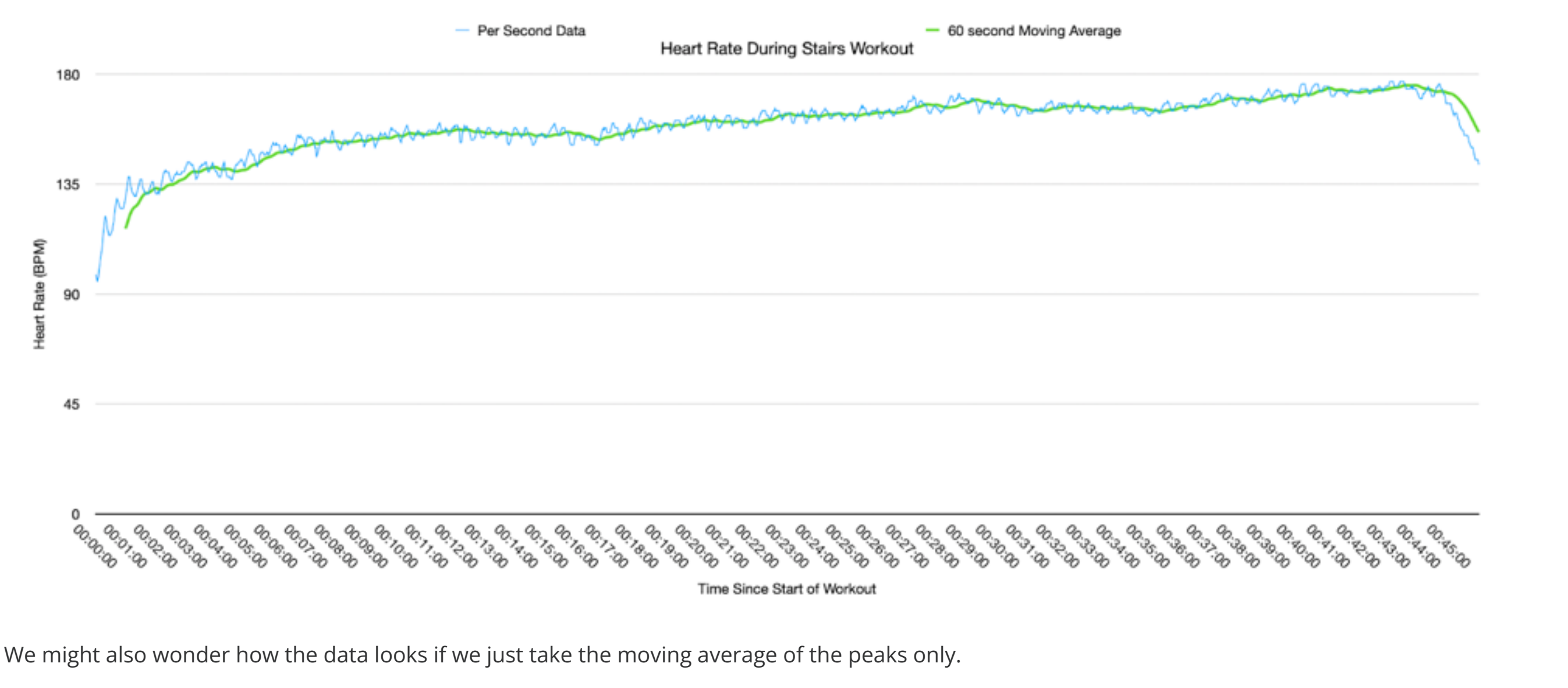
A Quick Note: Moving Averages

Our previous problem (counting trips up and down the stairs) not only gave us practice with lists, but also with breaking a larger problem down into two smaller problems. Breaking larger problems down into smaller problems is quite an important skill in programming as it is the only way to reasonably deal with problems of substantial size. Keep this lesson in mind not only on your upcoming assignments, but also throughout whatever other programming you might do.

We are going to wrap up this example of heart rate data for lists with one more problem: moving averages. The N-second (or N-day, or any other unit of time) moving average of time series data is another sequence of time series data where each point is the average over the past N points. The earliest points can be handled in a couple different ways: either using fewer points in the average, or by creating a shorter output series and dropping the first N-1 points. We will take the later approach, resulting in a shorter output than input. For example, if our input is [1,5,8,12,15,12,4,3] and we wanted to take the 4-second moving average, we would get [6.5, 10, 11.75, 10.75, 8.5]. We arrive at this by:

- (1+5+8+12)/4 = 6.5
- (5+8+12+15)/4 = 10
- (8+12+15+12)/4 = 11.75
- (12+15+12+4)/4 = 10.75 , and
- (15+12+4+3)/4 = 8.5.

The moving average of a time series of data gives a "smoother" result than the input data—local variations are flattened out, giving the broader trend. For our workout data, the smoother moving average shows the trend of workout intensity. The graph below shows the original data (blue) and the 60 second moving average of the heart rate data:



We might also wonder how the data looks if we just take the moving average of the peaks only.

Note that if we did this, we would have applied two transformations to the data: filtering it down to just the peaks, and taking the moving average of the peaks. For your next programming problem, you are going to write three functions: one to filter the data down to only the peaks, one to take the N-second moving average, and one that gives the N-second moving average of just the peaks. Note that the last of these should be very short as it makes use of the prior two.

We also encourage you to take a moment to observe how abstracting "get the moving average" out into its own function serves three benefits. First, as you have seen before, this makes for a smaller, self-contained problem which is easier to solve. Second, you can test it separately from the other functions, giving you confidence that it works when you use it with something else (and making it easier to debug if something does go wrong!). Third, you can use this function separately when you want to. If you just want to compute the moving average of the original data, you can do that. If you want the moving average of the peaks, you can put them together. We aren't going to delve into software engineering principles here, but do briefly note that this idea is called the "Single Responsibility Principle"—make each function do "one thing" so you can put them together in different ways easily.

Day 6, Part 3

Now it is time to do a few assignments on the MLP.

Start with completing **11_hr_moving_avg** on the Mastery Learning Platform. Login to the course server, and navigate to **11_hr_moving_avg** and read the readme for directions.

When you've completed that assignment, complete **12_tests_subseq** on the Mastery Learning Platform. Login to the server, go to **12_tests_subseq** in your git repository and read the README for directions.

Finally, once you've completed that, complete **13_subseq** on the Mastery Learning Platform. Login to the server, go to **13_subseq** and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.