Week 2 Coding Lecture 1: For Loops

In this lecture, we will discuss loops: A programming concept that allows you to repeate a block of code many times.

Suppose that we want to make the row vector $x = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix}$ as an array in python. We know at least two very easy ways to do so. We could write it out like this:

```
import numpy as np
x = np.array([[0, 1, 2, 3, 4]])
print(x)
```

```
    [[0 1 2 3 4]]
```

or we could use the `arange` function like this:

```
x = np.arange(0, 5)
x = np.reshape(x, (1, -1))
print(x)
```

```
    [[0 1 2 3 4]]
```

(Note that the reshape command is necessary because we wanted a 1×5 row vector instead of a 1D array.)

But suppose that we did not know about the `arange` function and did not want to type out the whole vector. (I will mostly stick to short vectors in these notes because they don't fill the entire page when printed, but you can imagine why we wouldn't want to type out the entire vector if there were 1,000 or 1,000,000 entries instead of 5.) Another approach would be to make an "empty vector" x and then fill in the correct entries. By "empty vector", I mean a vector of the appropriate size but without the correct values.

There are a few commands in the `numpy` package to construct arrays. Two of the most useful are `zeros` and `ones`. The syntax for these functions is fairly straightforward. For instance, `np.zeros((m, n))` produces a 2D array with `m` rows and `n` columns where all of the entries are zero. `np.zeros(m)` produces a 1D array with `m` entries, all of which are zero.

```
print(np.zeros((3, 4)))
```

```
    [[0. 0. 0. 0.]
     [0. 0. 0. 0.]
     [0. 0. 0. 0.]]
```

```
print(np.zeros(5))
```

```
    [0. 0. 0. 0. 0.]
```

The `ones` command works exactly the same way, but it produces an array whose entries are all ones.

```
print(np.ones((3, 4)))
```

```
    [[1. 1. 1. 1.]
     [1. 1. 1. 1.]
     [1. 1. 1. 1.]]
```

```
print(np.ones(5))
```

```
    [1. 1. 1. 1. 1.]
```

If we want to make an empty vector of the same size as `x`, we can therefore use

```
x = np.zeros((1, 5))
print(x)
```

```
    [[0. 0. 0. 0. 0.]]
```

Now that we have an empty vector to work with, we can fill in the entries we want.

Side Note: I will call this an "empty" vector in this class, but that name is not quite right. There is another command in the `numpy` package called `empty` that creates an array of a given size without filling in any values. The entries of the resulting array will be determined by whatever happened to be in memory before. The `empty` function is faster than either `zeros` or `ones`, and so it is often used in highly optimized code. However, it tends to cause problems that are a nightmare to debug. I do not recommend that you use it here.

In particular, we can use

```
x[0, 0] = 0
x[0, 1] = 1
x[0, 2] = 2
x[0, 3] = 3
x[0, 4] = 4
print(x)
```

```
    [[0. 1. 2. 3. 4.]]
```

We have now created exactly the same vector as in the previous section, albeit with much more typing. The step where we make an empty vector is called *initialization* and we say that we *initialized* the vector `x`. This tells python to set aside a block of memory large enough to hold a 1×5 vector.

It's worth noting that this initialization step is not always necessary in other languages (particularly MATLAB), but it is required here. The reason is that we can't change the size of a numpy array once it is created, so we have to make sure that we have an array of the correct size. (There is a significantly worse alternative: Instead of setting each entry of `x`, we could repeatedly use the `append` function to add entries to the end of `x`. Note that this involves repeatedly copying the array.)

As you might imagine, code like that in the above cell is not very practical. This is already a tedious amount of typing with only 5 entries and would be much worse with 1,000 or 1,000,000. In addition, it is easy to make mistakes with this much code and harder to hunt them down if we find out later that we made the wrong vector. It is also tedious to make changes to this code. If we later decide that each entry of `x` should be twice as big, we would have to change 5 lines of code (or 1,000 or 1,000,000 if our vector were larger).

However, these five lines are almost exactly the same. This suggests that we should be able to avoid most of this typing. If we could tell python the pattern for one of these lines, then we could hopefully just tell python to repeat the pattern five times.

This is exactly what a `for` loop is for. If you want to repeat the same block of code several times, you can use the python `for` keyword. For example, to repeat some code five times, we could write the code

```
for k in range(5):
    # Some code to be repeated
    pass
```

(The word pass is not part of the syntax, but I had to include it because for loops are not allowed to only have comments in them. The keyword `pass` just means "do nothing".) In general, the first line should say

```
for variable_name in range(number_of_steps): other code
```

Note that the words `for` and `in` are required, as is the `:` at the end of the first line. It is also very important to indent the code after the loop. This indentation is what tells python which lines of code are supposed to be repeated. Other languages like MATLAB or C use other keywords or symbols to show the end of a block of code, but python uses levels of indentation. This can take a little getting used to. PyCharm will handle the indentation for you, but it is important to be consistent. If one line of code is indented by 1 space and another line is indented by 2 spaces, the code will either not run or not do what you want. Moreover, different operating systems and text editors handle tabs differently, so a tab may or may not be the equivalent of some number of spaces. I recommend that you go into PyCharm settings, then in the menu go to Editor->Code Style->Python and make sure that "Use tab character" is *un*checked. This will make PyCharm automatically convert tabs into spaces.

The word `for` is a reserved word that tells python this is a for loop. The variable `k` is called a *loop counter* or a *loop index* or a *loop variable*, depending on context. You can think of it as keeping track of what step you are on. When you run this code, it will behave exactly as if you had instead typed

```
k = 0
# Some code to be repeated
k = 1
# Some code to be repeated
k = 2
# Some code to be repeated
k = 3
# Some code to be repeated
```

```
k = 4
# Some code to be repeated
```

As you can see, the code is repeated five times and the variable `k` tells us which step we are on (counting from 0, jut like indices). For a particularly silly example, the following code prints out the number 8 ten times:

```
for k in range(10):
    print(8)
```

```
8
8
8
8
8
8
8
8
8
8
```

We can also use the loop variable in the code to be repeated. This is useful for making the code do slightly different things each time. As another example, the following code prints out the first four perfect squares:

```
for k in range(4):
    print(k ** 2)
```

```
0
1
4
9
```

There is nothing special about the name `k`. You can call your loop index whatever you want. I will usually stick to things like `i`, `j`, `k`, `m` or `n` because those are common index variables in mathematics, but some people prefer more descriptive names like `counter` or `index` or `loop_var`. This is really just a matter of taste.

Getting back to our previous example, after initializing `x` we want to repeat the code `x[0, some_number] = some_number` over and over again. The value of `some_number` starts at 0, then increases to 1, then 2, then 3 and then 4. This is exactly what our loop counter did. We can rewrite our original code as

```
x = np.zeros((1, 5))
for j in range(5):
    x[0, j] = j

print(x)
```

```
[[0. 1. 2. 3. 4.]]
```

Remember, what python is really doing when we run this is

```
x = np.zeros((1, 5))
j = 0
x[0, j] = j
j = 1
x[0, j] = j
j = 2
x[0, j] = j
j = 3
x[0, j] = j
j = 4
x[0, j] = j

print(x)
```

```
[[0. 1. 2. 3. 4.]]
```

which is exactly what we wanted.

This is an incredibly common coding construct, and we will use it many times in this course. First, you initialize a variable that will be used for storing your results. Next, you use a loop to make many related calculations. The result of each of these calculations gets stored in one of the entries of your original variable. In our case, the "calculation" was just finding the value of `j`, but in real problems it will often be substantially more complicated.

As another example, suppose that we wanted to create the vector

$$x = \begin{pmatrix} 1 \\ 1.1 \\ 1.2 \\ 1.3 \\ 1.4 \end{pmatrix}.$$

Of course, we already know a shortcut with the `arange` or `linspace` functions, but suppose that we didn't have those options. We could do almost the same thing as before. (Notice that `x` is a column vector this time.)

```
x = np.zeros((5, 1))
x[0, 0] = 1
x[1, 0] = 1.1
x[2, 0] = 1.2
x[3, 0] = 1.3
x[4, 0] = 1.4

print(x)
```

```
[[1. ]
 [1.1]
 [1.2]
 [1.3]
 [1.4]]
```

Once again, we are repeating almost the same line of code five times, so we should be able to use a loop to make this simpler. However, this problem as an extra wrinkle because there are two different numbers that change in each line: The row index of `x` (which takes the values 0, 1, 2, 3 and 4) and the value being assigned (which takes the values 1, 1.1, 1.2, 1.3 and 1.4). A loop only keeps track of one loop variable for us. This means that we will need to deal with the other value on our own. Probably the easiest way to approach this problem is to write something like:

`x = np.zeros((5, 1)) for k in range(5): x[k, 0] = # Some value`

This means that the loop will keep track of the index of `x` (we get to use `k` as the index), but we will still have to figure out what "some value" should be at each step. A standard way to do this is to start by defining a variable as whichever value we want in the first step, then incrementing this variable at every step of the loop. That is,

```
x = np.zeros((5, 1))
x_value = 1;
for k in range(5):
    x[k, 0] = x_value
    x_value = x_value + 0.1

print(x)
```

```
[[1. ]
 [1.1]
 [1.2]
 [1.3]
 [1.4]]
```

The line `x_value = 1` is also called *initialization*. Here we are initializing the variable `x_value`. The line `x_value = x_value + 0.1` increments this value by 0.1 after each step. The first time through the loop, `x_value` is 1 (as desired). The second time through the loop it is 1.1, then 1.2, then 1.3, then 1.4.

This is another very common pattern in programming. If you have a variable that changes at each step in a loop, you can initialize it to some useful starting value, then update it at the end of each loop step.

The incrementing approach described above is a standard approach and should almost always be your first choice. In this case, however, we have another interesting option: We can find a formula for `x_value` interms of `k`. In particular, we can use

```
x = np.zeros((5, 1))
```

```
for k in range(5):
    x[k, 0] = 1 + 0.1 * k

print(x)
```

```
[[1. ]
 [1.1]
 [1.2]
 [1.3]
 [1.4]]
```

You should convince yourself of why this makes exactly the same vector as in the previous example.

It is not always easy to find a formula like this, but if you happen to notice one then this is often a good approach. It tends to be much harder to think up and possibly a little bit slower, but often doesn't accumulate as much rounding error. As with many things, unless you are extremely concerned about efficiency, which method you choose is largely a matter of taste.

It is worth noting that the `range(5)` part of our `for` loops can be generalized quite a bit. For example, we can replace it with the syntax `range(start, stop, step)`, like this:

```
for k in range(2, 12, 3):
    print(k)
```

```
2
5
8
11
```

This syntax makes the loop variable begin at `start` and increase in increments of `step` until just before it reaches `stop`.

Actually, the `range` part of our `for` loops is not a part of the general python `for` loop. We can replace `range` with a wide variety of different statements. A discussion of what exactly `range` does or what statements are allowed to replace it is well beyond the scope of this class. That said, it is useful to know that we can replace the `range` statement with a 1D numpy array. (Technically, you can also replace it with a 2D array, but the results are probably not what you would expect.) As an example, we could write the code

```
x = np.exp(np.arange(3, 10))
print(x)
```

```
[  20.08553692   54.59815003  148.4131591   403.42879349 1096.63315843
 2980.95798704 8103.08392758]
```

```
for k in x:
    print(k)
```

```
20.085536923187668
54.598150033144236
148.4131591025766
403.4287934927351
1096.6331584284585
2980.9579870417283
8103.083927575384
```

As you can see, when we use a 1D numpy array instead of `range`, the loop variable takes on the value of each entry of the array in turn.

We could therefore have used the `np.arange` function in place of `range` in all of our previous loops. As an example,

```
x = np.zeros((1, 5))
for j in np.arange(5):
    x[0, j] = j

print(x)
```

```
[[0. 1. 2. 3. 4.]]
```

This is actually usually a bad idea, but explaining why is beyond the scope of these notes. For now, just use `range` in your `for` loops whenever possible.

Nested loops

Now let's try something slightly more complicated. Intead of filling out the entries of a vector, we will fill out the entries of a matrix. For starters, we will try to create the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{pmatrix}.$$

If we do not want to type the whole matrix out (and once the matrix gets large that would be very impractical) then we can use a similar approach to the last section. For instance, we could type

```
A = np.zeros((4, 3))
A[0, :] = 1
A[1, :] = 2
A[2, :] = 3
A[3, :] = 4

print(A)
```

```
[[1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
```

This initializes the matrix (creates a zero matrix of the right size), then sets each row in turn. As in our previous examples, we have almost exactly repeated the same line of code several times, which means that this is a prime candidate for a loop. Following the same logic as above, we can write

```
A = np.zeros((4, 3))
for i in range(4):
    A[i, :] = i + 1

print(A)
```

```
[[1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
```

This produces the desired matrix.

Now suppose that we want to make the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}.$$

Following a similar strategy, we could initialize our matrix and then fill in each value one at a time.

```
A = np.zeros((4, 3))
A[0, 0] = 1
A[0, 1] = 2
A[0, 2] = 3

A[1, 0] = 4
A[1, 1] = 5
A[1, 2] = 6

A[2, 0] = 7
A[2, 1] = 8
A[2, 2] = 9

A[3, 0] = 10
A[3, 1] = 11
A[3, 2] = 12

print(A)
```

```
[[ 1.  2.  3.]
```

```
 [ 4.  5.  6.]
 [ 7.  8.  9.]
 [10. 11. 12.]]
```

Once again, this seems like a good candidate for a loop, but this time we have three values that are changing: The row index, the column index and the value on the right hand side. One approach might be to have one `for k in range(12)` loop and perform each of these twelve steps, but it would end up being pretty messy to keep track of the necessary variables. A better method is to think of each block of three commands as a single piece of code. We really repeat each of those blocks three times. This means that we want something like

`A = np.zeros((4, 3)) for i in range(4): A[i, 0] = some_number A[i, 1] = some_other_number A[i, 2] = some_different_number`

Look at the three lines inside this loop. They are really just the same line of code repeated three times with slightly different numbers, which makes them another good candidate for a `for` loop. We can therefore write

`A = np.zeros((4, 3)) for i in range(4): for j in range(3): A[i, j] = some_number`

Now all we need to do is keep track of `some_number`. Here is one of the more straightforward methods (very similar to one of the examples from the single `for` loop section):

```
A = np.zeros((4, 3))
A_value = 1
for i in range(4):
    for j in range(3):
        A[i, j] = A_value
        A_value = A_value + 1

print(A)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]
 [10. 11. 12.]]
```

Notice the indentation - it is very important. All the code in the first `for` loop is indented once (in this notebook, one indentation is four spaces). All the code in the second `for` loop is indented an additional time. When the loops are over, we return to the original indentation.

You should carefully study this example until you can clearly see why it creates the matrix we are looking for and what order everything happens in. In particular, it may help to have python print out the values of `i`, `j` and `A_value` at every step.

As you can see, code inside a `for` loop does not have to be just one line; it can be arbitrarilly complicated. In particular, we are allowed to put loops inside other loops. These are called *nested for loops*. The loop involving `i` (the one where the `for` is not indented) is called the *outer loop*. The loop involving `j` (the one where the `for` is indented once) is called the *inner loop*. We will often need two loops nested together like this example, but it is probably wise to avoid going any deeper than this. Python doesn't have any limit on how many loops you can nest (well, it does, but the limit is very, very large), but you will find that your code becomes both very slow and very hard to understand once you have too many nested loops.

Fibonacci numbers

Now let's try a more interesting mathematical example. We will try to calculate some Fibonacci numbers. The Fibonacci numbers F_n are defined by the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

where $F_1 = F_2 = 1$. That is, the nth Fibonacci number is the sum of the previous two. We will start by calculating the first 20 Fibonacci numbers and saving them in a vector. This is the first example where we really do need a loop: There is no python shortcut like `np.arange` for calculating Fibonacci numbers. (There is actually a relatively simple formula for F_n, but it is not easy to derive.)

We will use the same basic approach as before: Initialize an array for our Fibonacci numbers, then use a loop to fill in each entry of the array in turn. The basic skeleton is

```
fib = np.zeros(20) for n in some_range: fib(n) = # calculate the nth Fibonacci number
```

(Note that I chose to use a 1D array instead of a row or column vector for `fib`. Any of those choices would be fine.) There are quite a few variations we could use to write this code, but probably the most straightforward way is to essentially copy the definition into our loop like so:

`fib = np.zeros(20) fib[0] = 1 fib[1] = 1 for n in some_range: fib[n] = fib[n - 1] + fib[n - 2]`

Notice that we included the first two Fibonacci numbers in the initialization step. This is just part of the definition of the problem. We can now figure out what the loop variable is supposed to be. We ahve already filled out the first two values, so the index `n` should start at 2, and we want to fill out everything up to index 19 (the 20th entry), so `n` should end at 19. We therefore need

```
fib = np.zeros(20)
fib[0] = 1
fib[1] = 1

for n in range(2, 20):
    fib[n] = fib[n - 1] + fib[n - 2]

print(fib)
```

```
[1.000e+00 1.000e+00 2.000e+00 3.000e+00 5.000e+00 8.000e+00 1.300e+01
 2.100e+01 3.400e+01 5.500e+01 8.900e+01 1.440e+02 2.330e+02 3.770e+02
 6.100e+02 9.870e+02 1.597e+03 2.584e+03 4.181e+03 6.765e+03]
```

(As a side note, the `e` in the printout is scientific notation. It stands for "times ten to the power of", so something like `6.765e+03` means 6.765×10^3.)

Suppose that we decide to calculate the first 200 numbers instead. We can use almost exactly the same code, but we have to make two changes: We need to change the range from `range(2, 20)` to `range(2, 200)` and we need to change the initialization line so that `fib` is the right size.

```
fib = np.zeros(200)
fib[0] = 1
fib[1] = 1

for n in range(2, 200):
    fib[n] = fib[n - 1] + fib[n - 2]

print(fib[-1])
```

```
2.8057117299251016e+41
```

I only printed out the last value of `fib` because the entire array takes a lot of space.

Note that forgetting to change either 20 to a 200 causes problems. If we forget to change the 20 in the `range` function, then the loop only calculates the first 20 Fibonacci numbers and the rest of the array is left as zeros. If we forget to change the 20 in the initialization step, then python will throw an error when `n` gets to 20 and we try to set the value of `fib[20]`, because `fib` does not have a 21st entry.

Since both of these numbers are always supposed to be the same, it is a good idea to make a variable for the number of Fibonacci numbers that we are calculating.

```
total_numbers = 200
fib = np.zeros(total_numbers)
fib[0] = 1
fib[1] = 1

for n in range(2, total_numbers):
    fib[n] = fib[n - 1] + fib[n - 2]

print(fib[-1])
```

```
2.8057117299251016e+41
```

This way, we only need to change the value of `total_numbers` and the rest of the code will work as desired.

We have successfully calculated the first 20 (or 200, or any number) Fibonacci numbers. In fact, this code is pretty close to the best solution we can get for this problem. However, imagine that we modify the problem slightly. Suppose that instead of the first 20 numbers, we want to find all the Fibonacci numbers that are less than 1,000,000. Now our solution doesn't work. The issue is that we don't know beforehand how many numbers we will need, so we don't know what our loop variable should be.

To solve this problem, we want to start calculating Fibonacci numbers like before, but tell python to stop when `fib[n]` reaches 1,000,000. To write this in code, we will need a new concept that tells python "if `fib[n]` is bigger than 1,000,000, stop the loop." We will talk about this concept in the next lecture.