

▼ Week 1 Coding Lecture 2: Arrays and Vectors

Naive Example

Suppose that you are responsible for calculating the total grades for a class. As an example, suppose that the class has homework assignments worth a total of 190 points, tests worth a total of 200 points and quizzes worth a total of 50 points. To calculate a student's final grade, you need to add up their homework, test and quiz scores and divide by the total number of possible points (440). For example, suppose we had four students with the following grades:

Name	Homework	Quizzes	Tests
Alice	170	45	160
Bob	163	40	195
Carol	188	50	138
Dan	150	36	172

Here is one way we could calculate all of the students' final grades in python: First, we enter all of the data into python variables.

```
hw1 = 170
hw2 = 163
hw3 = 188
hw4 = 150

quiz1 = 45
quiz2 = 40
quiz3 = 50
quiz4 = 36

test1 = 160
test2 = 195
test3 = 138
test4 = 172
```

Next, we use the formula $\text{final} = (\text{hw} + \text{quiz} + \text{test}) / 440$ to calculate each student's final grade.

```
total_possible = 440
final1 = (hw1 + quiz1 + test1) / total_possible
final2 = (hw2 + quiz2 + test2) / total_possible
final3 = (hw3 + quiz3 + test3) / total_possible
final4 = (hw4 + quiz4 + test4) / total_possible
```

If you want to know a student's grade, you would just have to print out the corresponding variable. For instance, if Carol asked to see her grade you could use

```
print("Carol's grade is: ")
print(final3)
```

```
Carol's grade is:
0.8545454545454545
```

As you have probably noticed, this is not a very good method. For one thing, this involved a lot of typing (and imagine how much worse it would be with hundreds of students instead of four!) In addition, we had to make many variables with similar names like hw1, hw2, hw3 and hw3. It would be very easy to make a mistake with one of these variable names somewhere in the code, and it would take a lot of time to find and fix the problem. Another issue that might not seem obvious is that this code is difficult to change. For example, if we decided to change the formula for calculating final grades then we would have to make identical changes to our code in four different places. This creates more unnecessary typing and more opportunities for error.

▼ Array Example

A much better solution is to group grades together under just a few variable names. For example, we could put all the homework grades together into one variable, all the quiz grades together in another, etc. There are many ways to accomplish this task in python, but the tool we will use in this class is called a numpy array. You can create one with the following code (remember that you only have to import numpy once at the beginning of your script):

```
import numpy as np
hw = np.array([170, 163, 188, 150])
```

If you want to see this array, you can print it just like any other variable:

```
print(hw)

[170 163 188 150]
```

Similarly, we can make arrays for quiz grades and test grades:

```
quiz = np.array([45, 40, 50, 36])
test = np.array([160, 195, 138, 172])
```

These variables have a different type than any we have seen before:

```
print(type(hw))

<class 'numpy.ndarray'>
```

They are called nd arrays. The "nd" stands for n-dimensional, because it is possible to make arrays with any number of dimensions. All the arrays we created are 1-dimensional. That is, they have a length (the number of entries) but not a height. Arrays (along with almost all python variables) have various properties that you can access. In particular, arrays have a property called `ndim` that tells you the number of dimensions. You can access `ndim` with the following code:

```
print(hw.ndim)

1
```

Similarly, arrays have a property called `shape` that tells you how many entries there are in each dimension. In our case, these arrays only have one dimension and the shape just tells you how long the array is (i.e., how many elements are in the array). You can access the shape with the following code:

```
print(hw.shape)

(4,)
```

The last two lines of code are examples of very general python syntax. If you want to access a given property of a variable you can use the syntax `variable_name.property_name`. The type of a variable determines which properties it has. There is no need to memorize lists of property names. PyCharm will list all available properties as soon as you type `variable_name`. You can also look up the documentation for a type online to find out more about the properties. For instance, [here](#) is the documentation for the `ndarray` type.

Almost any operation that makes sense when applied to one number also makes sense when applied to an array. For example,

```
print(hw * 2)

[340 326 376 300]

print(test / 10)

[16.  19.5 13.8 17.2]

print(quiz + 5)

[50 45 55 41]

print(quiz ** 2)

[2025 1600 2500 1296]
```

Moreover, most of these operations also make sense when applied to two arrays of the same shape. For example, you can add two arrays of the same shape like this:

```
print(hw + test)

[330 358 326 322]
```

This operation is called *elementwise* addition, because python is adding corresponding elements of the two arrays. In other words, the first element of `hw` is added to the first element of `test`, then the second element of `hw` is added to the second element of `test`, etc.

The same thing works for other common mathematical operations. You can have elementwise subtraction, multiplication and division:

```
print(hw - test)

[ 10 -32  50 -22]

print(hw * test)

[27200 31785 25944 25800]

print(hw / test)

[1.0625      0.83589744 1.36231884 0.87209302]
```

Most of the usual mathematical functions also can be used on arrays in an elementwise fashion. For instance, you can take the `sin` of every element of `hw` with the code

```
print(np.sin(hw))

[ 0.34664946 -0.35491018 -0.47552367 -0.71487643]
```

Note that it only makes sense to combine arrays in this manner if they have the same shape. In particular, if we make two 1D arrays with different lengths and try to combine them, then we will usually get a somewhat cryptic error message.

```
x = np.array([1, 2, 3, 4])
y = np.array([1, 2, 3, 4, 5])
print(x + y)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-91-6e9322dd1dc7> in <module>
      1 x = np.array([1, 2, 3, 4])
      2 y = np.array([1, 2, 3, 4, 5])
----> 3 print(x + y)

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

SEARCH STACK OVERFLOW

```
print(x * y)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-92-1ef76cb9d707> in <module>
----> 1 print(x * y)

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

SEARCH STACK OVERFLOW

We won't talk about broadcasting in this class, so you can generally just assume that the error message "could not be broadcast together" means that you tried to use elementwise operations on arrays of different shapes.

Getting back to our example, remember that our original goal was to calculate the final scores for each student. We can do this with the code

```
final = (hw + quiz + test) / total_possible
print("The final scores are: ")
print(final)

The final scores are:
[0.85227273 0.90454545 0.85454545 0.81363636]
```

There is a lot of extraneous example code in between, but if you look back you'll see that we reduced 17 lines of code to 5. (In fact, this code also runs faster than the previous version. It is both more convenient and more efficient to keep all of your data in arrays.)

▼ Array Access

While we cleaned up our code quite a bit by using arrays, it does seem like we might have lost something important. Suppose that Carol comes to you and asks for her grade. Before, you could simply type

```
print(final[3])

0.8545454545454545
```

but that won't work here. We could of course just type

```
print(final)

[0.85227273 0.90454545 0.85454545 0.81363636]
```

to display all of the grades and tell Carol that hers is the third number, but we don't want to share everyone else's grade with one student. Fortunately, python has simple shortcuts to access individual entries in an array. If we want the third entry in `final`, we can just type

```
print(final[2])

0.8545454545454545
```

This syntax might look a little strange. The general syntax is `array_name[index]`, where `array_name` is the name of your array and `index` is the element number you want to access. It is very important to remember that python starts counting indices at 0, so index 0 is the first element of an array, index 1 is the second element, index 2 is the third element, etc. That is why we used `final[2]` and not `final[3]`.

```
print(final[0])
```

```
0.8522727272727273
```

```
print(final[1])
```

```
0.9045454545454545
```

```
print(final[2])
```

```
0.8545454545454545
```

```
print(final[3])
```

```
0.8136363636363636
```

Python does not allow you to access entries beyond the last one. Since the array `final` only has four entries, the last index is 3. If you try to access the element at index 4 (i.e., the fifth element of the array) then you will get an error:

```
print(final[4])
```

IndexError

Traceback (most recent call last)

You also cannot use decimal numbers for indices. An index must be a whole number. For instance:

```
print(final[1.5])
```

IndexError

Traceback (most recent call last)

```
<ipython-input-102-e99c969f5e7b> in <module>
```

```
----> 1 print(final[1.5])
```

IndexError: only integers, slices (``:``), ellipsis (``...``), `numpy.newaxis` (``None``) and integer or boolean arrays are valid indices

[SEARCH STACK OVERFLOW](#)

Notice that python is checking the type of the index, not the numerical value. This means that the following code will also fail, even though it makes sense mathematically:

```
print(final[1.0])
```

IndexError

Traceback (most recent call last)

```
<ipython-input-103-e7320f64aae9> in <module>
```

```
----> 1 print(final[1.0])
```

IndexError: only integers, slices (``:``), ellipsis (``...``), `numpy.newaxis` (``None``) and integer or boolean arrays are valid indices

[SEARCH STACK OVERFLOW](#)

In other words, only numbers with the type `int` are allowed as indices.

Python does allow a useful shortcut with indices. Negative indices count from the last element of the array. In other words, `-1` is the

```
print(final[-1])
```

```
0.8136363636363636
```

```
print(final[-2])
```

```
0.8545454545454545
```

```
print(final[-3])
```

```
0.9045454545454545
```

```
print(final[-4])
```

```
0.8522727272727273
```

You should think of this array access syntax as being a special variable name. `final[2]` is the name of the third entry (second index) of the array `final`. You can use this syntax anywhere a variable name would be appropriate. In particular, you can use it in calculations like

```
print(final[2] * final[3] - hw[0])
```

```
-169.30471074380165
```

Perhaps more importantly, you can also use this notation on the left side of the `=` operator to modify the contents of an array. For example, suppose that we missed one of Dan's homework assignments and his homework grade is supposed to be 170. We can use the code

```
hw[3] = 170
```

to change his homework grade. If we print out the array `hw`, we can see that the last entry has changed to 170.

```
print(hw)

[170 163 188 170]
```

However, there is an important caveat to remember when modifying the entries of an array. You are not allowed to assign a value to an element of the array if the index is out of bounds. That is, you can't assign a new value to an entry past the end of the array. For instance, if you try to change the 10th entry (index 9) of the array `hw`, you will get the following error:

```
hw[9] = 150
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-111-03dd8a3eed4a> in <module>
----> 1 hw[9] = 150

IndexError: index 9 is out of bounds for axis 0 with size 4
```

SEARCH STACK OVERFLOW

If you need to add additional entries to an array, you must make a new copy of the array with the additional element. In other words, once an array is made, you cannot make it bigger.

It is particularly common to want to add one more element to an array. To do this, you need to make a new array that is one longer than the original and then copy all of the elements of the old array into the appropriate places in the new array. This is such a common operation that numpy has a pre-defined function to do it for you:

```
print("The original hw array is: ")
print(hw)
print("We can add one more element with the append command: ")
print(np.append(hw, 150))
```

```
The original hw array is:  
[170 163 188 170]  
We can add one more element with the append command:  
[170 163 188 170 150]
```

The syntax is `np.append(name_of_vector, value_of_new_element)`. It is very important to remember that the `append` command makes an entirely new array. Very often, we don't want the old array anymore, so it is common to assign this new array to the name of the old one. For instance,

```
x = np.array([1, 2, 3, 4])  
x = np.append(x, 5)  
print(x)
```

```
[1 2 3 4 5]
```

It is also worth noting that copying large arrays is quite slow. We will talk about this more once we start writing more complicated code, but it is a good idea to minimize the use of functions like `append` when your arrays are too long.

▼ Slice Syntax

We saw above how to access and modify elements of an array one at a time. It is also possible to access or modify entire sections of an array. You can do so using something called a `slice`. The most basic version of a slice uses the following syntax:

`name_of_array[start:stop]`, where `start` and `stop` are indices. Here are a few examples:

```
x = np.array([3, 8, 5, 2, 1, 6])  
print(x[1:4])
```

```
[8 5 2]
```

```
print(x[0:2])
```

```
[3 8]
```

```
print(x[-3:-1])
```

```
[2 1]
```

In general, `x[start:stop]` gives you an array containing the elements of `x` from the index `start` up to **but not including** the index `stop`. This is why `x[1:4]` contains index 1 (the second element), index 2 (the third element) and index 3 (the fourth element), but not index 4 (the fifth element).

You can also exclude either `start` or `stop` from this syntax. If you exclude `start`, then python includes every element from the beginning of the array up to (but not including) the index `stop`. For instance,

```
print(x[:4])
```

```
[3 8 5 2]
```

If you exclude `stop`, then python includes every element from the index `start` up to the end of the array. For instance,

```
print(x[2:])
```

```
[5 2 1 6]
```

▼ Creating Evenly Spaced Arrays

It is often useful to create evenly spaced arrays. For instance, when we plot a function $f(x)$, we will often want an array of x -values to use for data points on our plot. Python has several different ways to create these evenly spaced arrays:

linspace

The first method is a function from the numpy package named `linspace`. It can be used like this:

```
print(np.linspace(1, 5, 10))
```

```
[1.          1.44444444 1.88888889 2.33333333 2.77777778 3.22222222
 3.66666667 4.11111111 4.55555556 5.          ]
```

This created an array with 10 elements, evenly spaced between 1 and 5. The general syntax is `np.linspace(start, stop, number)`. This creates an array with `number` evenly spaced elements between `start` and `stop`. Unlike the slice syntax from before, notice that the array created by `linspace` **does include stop**.

▼ arange

A similar command from the numpy package is called `arange`. It can be used like this:

```
print(np.arange(1, 5, 0.5))

[1.  1.5 2.  2.5 3.  3.5 4.  4.5]
```

This created an array with elements 0.5 apart, starting at 1 and ending just before 5. The general syntax is `np.arange(start, stop, step)`. This creates an array whose first element is `start` and whose last element is just smaller than `stop`. The difference between each element is `step`. In other words, the first element is `start`, and then the elements increase by `step` over and over until just before they reach `stop`.

We will use slice syntax and the `linspace` and `arange` functions very often throughout this class. It is easy to get confused and forget which of these includes the final value and which do not or to forget that slices start counting from index 0. These will become easier to remember as you practice, but in the meantime it is always a good idea to print out the arrays you are making and check that they have the correct number of elements. This will save you a lot of headaches chasing down errors later on in your code.

▼ 2D Arrays and Vectors

So far we have only made 1D arrays. These are very useful for holding lists of data, but they are not the most mathematically relevant object. In mathematics (particularly in linear algebra) we care a lot about whether a vector is a column (i.e., arranged vertically) or a

row (i.e., arranged horizontally). For example, in mathematics the following are not the same thing:

$$\mathbf{x} = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

In other words, we don't just care about the length of the vector; we care about its length and height. The vector \mathbf{x} is called a 1×4 vector because it has one row and four columns. It is called a *row vector* because it only has one row. The vector \mathbf{y} is called a 4×1 vector because it has four rows and only one column. It is called a *column vector* because it only has one column.

To make python understand this distinction, we need to make our arrays two-dimensional instead of one-dimensional. There are several ways to do this, but for now we will focus on one: the `reshape` command. As an example,

```
x = np.array([1, 2, 3, 4])
print("This is the array x:")
print(x)
print("x has this many dimensions: ")
print(x.ndim)
print("And x has this shape: ")
print(x.shape)
```

```
This is the array x:
[1 2 3 4]
x has this many dimensions:
1
And x has this shape:
(4,)
```

```
x_row = np.reshape(x, (1, 4))
print("This is the array x_row:")
print(x_row)
print("x_row has this many dimensions: ")
print(x_row.ndim)
```

```
print("And x_row has this shape: ")
print(x_row.shape)
```

```
This is the array x_row:
[[1 2 3 4]]
x_row has this many dimensions:
2
And x_row has this shape:
(1, 4)
```

```
x_col = np.reshape(x, (4, 1))
print("This is the array x_col:")
print(x_col)
print("x_col has this many dimensions:")
print(x_col.ndim)
print("And x_col has this shape: ")
print(x_col.shape)
```

```
This is the array x_col:
[[1]
 [2]
 [3]
 [4]]
x_col has this many dimensions:
2
And x_col has this shape:
(4, 1)
```

The general syntax is `name_of_new_array = np.reshape(name_of_old_array, (number_of_rows, number_of_columns))`. To make a row vector, `number_of_rows` should be 1 and `number_of_columns` should be the length of the old array. To make a column vector, `number_of_columns` should be 1 and `number_of_rows` should be the length of the old array.

One nuisance with this syntax is that you have to remember the length of your original array (in this case 4). Fortunately, python has a shortcut for this. You can replace the 4 with a -1 and python will replace it with the length of the original array.

```
x_row = np.reshape(x, (1, -1))
print("This is the array x_row:")
print(x_row)
```



```
print(x_row)
print("x_row has this many dimensions: ")
print(x_row.ndim)
print("And x_row has this shape: ")
print(x_row.shape)
```

```
This is the array x_row:
[[1 2 3 4]]
x_row has this many dimensions:
2
And x_row has this shape:
(1, 4)
```

```
x_col = np.reshape(x, (-1, 1))
print("This is the array x_col:")
print(x_col)
print("x_col has this many dimensions:")
print(x_col.ndim)
print("And x_col has this shape: ")
print(x_col.shape)
```

```
This is the array x_col:
[[1]
 [2]
 [3]
 [4]]
x_col has this many dimensions:
2
And x_col has this shape:
(4, 1)
```

Important note: In this class, we will refer to 2D arrays with only one row as "row vectors" and 2D arrays with only one column as "column vectors". If we don't need to specify row/column, we will just call them "vectors". This is not strictly correct python - there is another type of object called a vector in python, but we will not be using it. Our nomenclature matches the mathematical version, as well as the MATLAB version.

Homework Guidelines: In the homework, I will almost always specify that you should save your final answer as a row vector or column vector rather than a 1D array. This means that you have to reshape your answer once you are done. As an example, I might ask you to make a row vector with the entries 1, 2 and 3 and save that vector in a variable named `A1`. You could make it like this (among many other methods):

```
A1 = np.arange(1, 4)
A1 = np.reshape(A1, (1, 3))
print("This is A1:")
print(A1)
```


```
This is A1:
[[1 2 3]]
```

The reshape command is necessary here. You will not get credit from Gradescope if you just use a 1D array.

▼ Transpose

It is very common to want to reshape a row vector into a column vector or vice versa. You could just use the reshape command. For example, if we wanted the vector `A1` from above to be a column vector instead, we could write

```
A1 = np.reshape(A1, (3, 1))
print("This is the new A1: ")
print(A1)
```

```
 This is the new A1:
[[1]
 [2]
 [3]]
```

However, this operation is so common (and so mathematically useful) that it has its own name. Switching from a row to column or vice versa is called *transposing* a vector, and you can implement it in python with the `transpose` function. For instance:

```
print("A1 is currently a column vector:")
print(A1)
```

```
print("We can make it into a row vector with the transpose function:")
A1 = np.transpose(A1)
print(A1)
print("And then we can make it back into a column vector:")
A1 = np.transpose(A1)
print(A1)
```

A1 is currently a column vector:

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

We can make it into a row vector with the transpose function:

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

And then we can make it back into a column vector:

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

If \mathbf{x} is a vector, then we denote the transpose of \mathbf{x} by \mathbf{x}^T . That is, \mathbf{x}^T has the same elements as \mathbf{x} , but it is flipped from a row to a column or vice versa.

