Authors: Kevin Hsu and Erika Mack

Table of Contents

Intro

Data Types

Understanding Names

Functions

Iteration

Examples of Breaking Down Long Code Expressions

Intro

Coding is an essential part of Data 8 that many of you will continue to use even beyond this course. This skill enables you to process and draw insights from large amounts of data that you could never do by hand. **However, coding is still just a tool used for data science, and not its final goal; the most important parts of data science are performed without a computer, such as thinking critically about the quality of data, choosing a regression model for a dataset, analyzing visualizations to form an opinion, or considering the ethical implications of a study.** Having an understanding of both coding foundations and data science principles is necessary to succeed in data science.

This guide discusses some basic coding concepts like data types, names, and functions, as well as how you can use these ideas to digest long complicated lines of code, or write your own.

Data Types

There are three main categories of data types used in Data 8:

Single Values	Single values can be thought of as a single point of data. They come in the following three varieties: 1. Boolean: These are true or false values. They can be created by typing in True or False in Python, or by performing logical comparisons with ==, >, etc. 2. String: These are text data that have their own corresponding suite of functions like `.replace` and features like concatenation with a + operator. They are specified by putting single or double quotes around text. a. Example: “Hello”, or ‘world!’ 3. Number: These come in two varieties, integer and floating point. Integers

	have up to the ones place of precision, while floating point numbers can represent numbers with greater precision. a. Float example: 2.0 b. Integer example: 3
Arrays	An array is an ordered collection of zero or more single values (eg. an array of numbers created by `make_array`, `np.arange`, or an array of strings returned by `.column`). Checkout the array guide!
Tables	A Table can be thought of as a sequence of named columns (arrays) that each describe a single attribute of the data for all entries in the dataset, or alternatively a sequence of rows that each contain all information about a single entry in the dataset. Check out the guide on working with tables!

There are several ways to go back and forth between these classes of data, such as with the `make_array` function to transform a group of single values into an array, or the tbl.column function to extract an array from a Table. When evaluating a line of code, it is always important to think about what data types one is working with and verify that they match up. Here's some common mistakes for data type mismatches:

Data Type Mismatches Common Mistakes:

- You cannot use array operations on a Table or Table operations on an array (for example, trying to call `.num_rows` on an array will NOT work nor will calling `.item` on a Table; instead, you can first convert a Table to an array with `.column` or from array(s) to a Table with `.with_columns`).
- Remember that the output of `.select` is a Table and not an array! Even when you pass a single column name into `.select`, the output will still be a Table with just one column, so you cannot perform array operations with the output.
- When trying to convert a datatype like an array or a table into a single number with an operation like `np.mean` or `.num_rows`, and then performing additional arithmetic operations on the result, make sure that the order of datatype operations is correct (ie. convert to a single number first, and then use the result for more operations).
    - For example, suppose you don't have access to np.mean, but you still want to take the average of an array `nums`. One way that you could do this is by summing all the elements and then dividing it by the length of the array. Your code might look something like `sum(nums) / len(nums)`. Despite the numerator and denominator of this fraction involving an array, both `sum(nums)` and `len(nums)` collapse the array into single numbers, so we are dividing a number and number. Trying to perform arithmetic operations with an array will apply the same operation to every single element, and arithmetic operations with a Table will generally result in an error.
- Be wary of mismatches between single values like trying to add a string and a number together. For example, '1' + '2' is '12', 1 + 2 is 3, but '1' + 2 is a type mismatch error.

- Note that some operations also work between different classes of data types, such as multiplying an array with a number together to create a scaled version of the array.

Understanding Names

You can keep track of a specific value by assigning it to a specific name, which allows you to refer to that value by the name instead of the actual value itself (ie. if you define a name `x = 5`, you can use `x` to refer to 5). Named values are essential for many reasons, including writing more concise, readable code and creating more complex logic.

To create a name and assign value to it, you use a single equals sign. The syntax looks like:

```
value_name = ...
```

The name is always on the left hand side, and the right hand side must be an expression that evaluates to a value (single value, array, Table). Note that in these assignment statements, the right hand side is always evaluated before the left hand side. For example, consider the following code snippet:

```
x = 10
x = x + 20
```

The first line simply assigns the number 10 as the value of `x`, but the second line incorporates `x` on the right hand side while reassigning a new value to `x`. The second line might seem jarring from a mathematical standpoint, but is valid when we consider the order that Python processes the statement. We first evaluate the right hand side of the assignment to be `x + 20`, or `10 + 20`. After simplifying this to just a single number (30), we assign `x` to this new value.

To evaluate an expression with a name inside it, simply replace every instance of a name with its value. For example, if you have the following code:

```
fahrenheit_temps = make_array(60, 70, 80)
celsius_temps = 5/9 * (fahrenheit_temps - 32)
```

To figure out what value is assigned to `celsius_temps` on the second line, replace the `fahrenheit_temps` with an array of 60, 70, and 80 and perform the arithmetic operations.

Here's some common mistakes with names:

- Notice the difference between strings and names. Names don't have quotes around them, so only put quotation marks around text that you want to put into a string. For example, `"hello"` is a string whereas `hello` is a name. Putting `hello` without quotes in a separate cell would output the value of the name `hello`, but putting `"hello"` in a cell would output the string `"hello"` back. Another common mistake with names and strings is how to refer to a specific

column in a Table. Suppose you have a Table with the column “Cities”. If you want to refer to it with a function like `.column` or `.where`, you have to use the string `“Cities”`. Note that this is different from a name `Cities`, since the name could store any possible value while the Table function requires the specific string column name.
- Don’t reassign builtin functions or modules. When Python first starts up and we import datascience and numpy into the notebook, some functions are already predefined, including `print`, `sum`, `max`, and `make_array`. Jupyter also highlights builtin functions in green. However, if you treat one of these names as a variable and write a line of code like `print = 5`, you will be unable to call the `print` function normally anymore. Trying to call `print(“hello”)` would simply error, because you’re now trying to call the number `5`. To fix this, delete the line of the code that reassigns the preexisting function and restart the kernel.
- Make sure you actually use the names you assign. If you assign a value to a name (that’s not the final solution name in a problem) but never use it, you should try to think about why you created that name in the first place and what goal it accomplishes. You will eventually find that you’re either able to write the same code without creating that name in the first place, or you’ll be able to find a use for that name.

Functions

In math, a function called on a variable x is denoted as f(x). Here f is defined as some procedure for an undefined value x, once x is assigned to some value.

For example: f(x) = x + 2 means for any value x that is passed in to f, the number 2 will be added. f(2) = 2 + 2, f(5) = 5 + 2, etc. Python functions work the same way in that any arguments passed in to the function act as a variable which is used for some procedure as defined in the function. From the example above, we can create a python function called `add_two`:

Whenever we want to define a function in Python we use a def statement:

```
def add_two(x):
      y = x + 2
      return y
```

`def` defines a function of the name you choose (here we chose the name `add_two`) and has as many arguments as you pass into the parentheses (here we had one argument named `x`). Your procedure is what your function does, and that is defined by the lines of code indented beneath the `def` statement (here we have two lines of code: one where we add 2 to the value passed in for `x` and assign that result to a variable `y`, and a second where we return `y`). Finally, you need a `return` statement. Whatever you want your function to spit back out at you is what should be in the `return` statement (here we are returning the variable `y`).

Common mistakes in creating functions:

Example of hardcoding variables:

When creating a function it can be helpful to think of a specific use case and generalize your function from there. Usually if you find yourself copying and pasting blocks of code, that's an indication you should be making a function. For example, if we wanted a function that could sort any table by its first column in descending order, we might want to work with a specific table first to see how we'd code it. Pretend we have a table named `example_table`, we know the code to sort `example_table` by its first column in descending order would be:

```
example_table.sort(0, descending = True)
```

Now we can use this process to create a more general function that can be used for any table by replacing example_table for a variable that is passed into the function as an argument:

```
def sort_descending_first_column(tbl):
            return tbl.sort(0, descending = True)
```

This function is correct and would work to sort any table passed in as an argument. However, a common mistake is to "hard code" a variable into the function so that the function only works for a specific value. In the function above that might look like this:

```
def sort_descending_first_column(tbl):
            return example_table.sort(0, descending = True)
```

Since we hard coded the example table into the function, no matter what table we pass in to `sort_descending_first_column`, the result returned is a copy of `example_table` sorted in descending order by its first column. This is something to watch out for. **In general, you want to make sure every argument you pass in to your function shows up as a variable somewhere in your intended code.**

Passing in wrong data types as arguments:

When working with functions, it's important to remember that the code inside your function still needs to follow all the rules of Python. That means anything that will error outside a function will also error inside. This is why it's really important to pay attention to your data types. Any arguments passed in to your function will be copy and pasted into your code wherever the variable for that argument is used.

As an example let's use the `sort_descending_first_column` function we defined in the last example. It takes a table as an argument. If you were to do something like this:

```
sort_descending_first_column(example_table.column(0))
```

Then you would be passing in an array as the tbl argument. Then when the `sort_descending_first_column` function got to the return line it would be trying to call the

`.sort` method on the array `example_table.column(0)` and it would cause an error. If you are ever unsure of the data types of a function you would like to use that you did not create, you can check the Python reference sheet.

Passing an incorrect number of arguments into a function:

Each function will take the number of arguments that it is built to. When using a function, if you are unaware of the number of arguments it takes check the python reference sheet or the documentation. Some functions take zero arguments and are executed by calling `function_name()`. It is important to note that even if a function takes zero arguments, if you want to use it and have returned whatever is defined in the function's return statement, you must call it by using the empty parentheses.

Some functions may have optional arguments, like the function `np.arange`. `np.arange` takes 1, 2, or 3 arguments and will error if zero or four or more arguments are passed in. Some arguments like `sum`, `max`, and `min` take an arbitrary number of arguments i.e. `max`(2, 4, 7) has three arguments and `max(make_array`(2, 4, 7)) has one argument that's an array. The functions you will be creating in this class will take the exact number of arguments you passed in to the line of your `def` statement when creating your function.

Variables created inside your functions don't exist outside of your function:

Let's look at the `add_two` function we wrote earlier. The variable `y` is used inside the function, but does not exist as an assigned variable in the rest of our notebook. If we were to call `add_two(3)` the number 5 would be assigned to the variable y within the function, then 5 would be returned from the return line. However, if we tried to access `y` outside of our function, it would cause an undefined variable error. This note is a little outside of scope for the class so if you don't fully understand, don't worry. The main takeaway is to not use variables defined inside your functions anywhere else in your notebook unless you redefine the variable elsewhere.

Forgetting a return statement:

If your function seems to be working and all the logic is sound but for some reason your function isn't returning anything, you might be forgetting a return statement. Alternatively, it's important to remember that even if you are using a return statement, you might be using a function like `print()` or `.show()` that doesn't return anything, but simply visualizes the result. If you're trying to use `.show()` in a function, the table method `.take()` is probably what you are looking for, and if you're using `print()`, you can just delete the print statement and use the values you were trying to print.

Not storing return values of functions:

Finally, while not all functions have to have return values in order to work, all builtin functions and functions you write in Data 8 have a return value. If you don't capture the return value of a function by assigning it to a name, then the work the function does won't be saved anywhere. For example, let's consider `np.append`, which adds an element to the end of an array. In the code snippet below, we want to add the number 5 to the end of our array.

```
arr = np.arange(5)
np.append(arr, 5)
print(arr)
```

This print statement will show `array([0, 1, 2, 3, 4])` instead of `array([0, 1, 2, 3, 4, 5])`, since np.append's return value of the updated array was not assigned to anything. We can fix this bug by making the following change:

```
arr = np.arange(5)
arr = np.append(arr, 5)
print(arr)
```

This will produce the correct output of `array([0, 1, 2, 3, 4, 5]).`

Iteration

In this class you will often want to repeat a certain block of code a specific number of times, or do something with every element of an array. This is called iteration and is done with a for loop. The structure of a for loop is as follows:

```
for <variable> in <array>:
      <code you want repeated>
```

When you write a for loop, your code will automatically assign your `<variable>` to the first value in your `<array>`, then all of the code in the indented block will be executed. Once your code reaches the end of the indented block, the for loop will start over and your `<variable>` will be assigned to the second element in your `<array>`. This will continue until your variable has been assigned to every value in the `<array>` and your indented code block will have been executed as many times as there are elements in the `<array>`.

A common notation is to assign the letter `i` (short for index) as your `<variable>`, but you can make your variable anything you want. For example, if you have an array `colors` defined as `make_array('red', 'green', 'blue', 'yellow')` and you are using a for loop to print each color, you may choose to write the following code:

```
for color in colors:
      print(color)
```

Where color is your variable and will be assigned to each color in the colors array in order. The function would print the following output:

```
'red'
'green'
'blue'
'yellow'
```

We can 'unroll' for loops to further investigate exactly how they operate. The previous for loop, unrolled would look like this:

```
color = colors.item(0) # 'red'
print(color)
color = colors.item(1) #'green'
print(color)
color = colors.item(2) #'blue'
print(color)
color = colors.item(3) #'yellow'
print(color)
```

Finally, another use case is to perform iteration while you don't actually care about the value of the variable and want to simply repeat an action n times. For example, suppose you want to create a 10 element long array where each element is just 0. Here's one way you could create such an array:

```
zeros = make_array()
for unused in np.arange(10):
      zeros = np.append(zeros, 0)
```

This will assign zeros to a length 10 array where each element is just 0. One note is that the `<variable>` is never used inside the body of the loop, so it doesn't matter what the value of `unused` is during a specific iteration.

We can 'unroll' that for loop as well:

```
zeros = make_array()
i = np.arange(10).item(0) # 0
zeros = np.append(zeros, 0) # array([0])
i = np.arange(10).item(1) # 1
zeros = np.append(zeros, 0)# array([0, 0])
i = np.arange(10).item(2) # 2
zeros = np.append(zeros, 0)# array([0, 0, 0])
```

```
i = np.arange(10).item(3) # 3
zeros = np.append(zeros, 0)# array([0, 0, 0, 0])

… # rest of lines omitted
# note we didn’t use i anywhere besides assigning it!
```

Examples of Breaking Down Long Code Expressions

Now that we’ve discussed data types, variables, and functions, we can apply these skills to evaluate longer lines of code. Similar to mathematical expressions, the goal of breaking down longer lines of code is to always simplify each expression as far as possible until you reach a single output (which falls in one of our data type classes of single value, array, or Table).

As a brief analogy, think about simplifying the following math expression:

$$f(x)*e^{y} + 3*h(g(z + 4*x - y)) + cos(2*pi*z)$$

How would you start evaluating this expression? The first step you might take is to substitute in the variables x, y, and z into the expression. Then, you would evaluate each of the functions on their input. Note that some functions like g have an expression as its input that we need to evaluate first, and h even has another function as its input. As a result, we know that we would first need to evaluate g on its input before we can begin to evaluate h, as we need g’s output for h’s input. Finally, after you’ve reduced all functions and variables into just numbers, you can simply add the numbers together.

How to evaluate code

Example 1: Working through chained table functions

When evaluating a line of code, you have a similar goal of wanting to reduce all functions and names down to a single output (single value, array, or Table). Let’s walk through an example of evaluating a long line of code:

tbl.where(‘Column1’, are.above(10)).sort(‘Column2’, descending = True).column(0).item(1)

Step 1:

<u>**tbl**</u>.where(‘Column1’, are.above(10)).sort(‘Column2’, descending = True).column(0).item(1)

>> <u>**The underlined code**</u> is a table

Step 2:

tbl<u>.where(‘Column1’, are.above(10))</u>.sort(‘Column2’, descending = True).column(0).item(1)

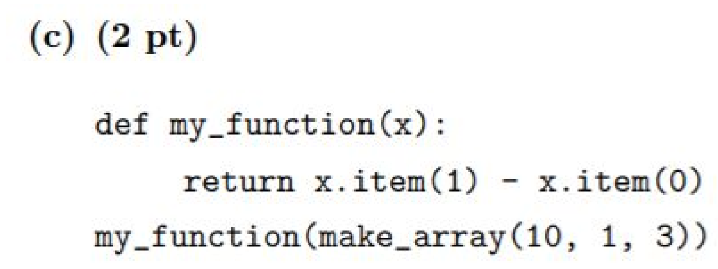
>> <u>**The underlined code**</u> returns a <u>**table**</u> filtered to only include rows from the original table tbl where values in ‘Column 1’ are above 10

Step 3:
tbl.where('Column1', are.above(10)).sort('Column2', descending = True).column(0).item(1)
>>**The underlined code** returns a **table** where the rows from the **table** in step 3 are shown here in **bold text that's not underlined** have been sorted in descending order by values in 'Column 2'

Step 4:
tbl.where('Column1', are.above(10)).sort('Column2', descending = True).column(0).item(1)
>> **The underlined code** returns an **array** composed of values from the first column of the **table** resulting from the **bolded and not underlined text**.

Step 5:
tbl.where('Column1', are.above(10)).sort('Column2', descending = True).column(0).item(1)
>> **The underlined text** would result in a **single value data type** (string, int, bool, etc) of the same type as the values stored in the **array** produced by the code in the **bolded and not underlined text**.

Example 2: Working through a function call
Here's a second example where we're working through a question from Fall 2019's midterm. The goal of this question is to determine what the output of this code snippet is.


(c) (2 pt)

```
def my_function(x):
    return x.item(1) - x.item(0)
my_function(make_array(10, 1, 3))
```

We first notice that the output of this code snippet is the output of calling the function `my_function`. To start calling the function, we first need to pay attention to the input of `my_function`.

```
def my_function(x):
     return x.item(1) - x.item(0)
my_function(make_array(10, 1, 3))
```

The bolded section is the input, which is assigned to `x`, the argument `my_function` takes in. I can then make a mental note that `x = array([10, 1, 3])` within the body of the function. Next, I start evaluating the body of the function with the argument we are passing, the array.

```
def my_function(x):
     x = array([10, 1, 3])# Evaluate with the argument
     return x.item(1) - x.item(0)
my_function(array([10, 1, 3]))
```

The function's body only has one line, which is also the return value of the function. We can break this line of code down by substituting `x` with its value (the array `[10, 1, 3]`) and evaluating each term individually.

```
def my_function(x):
     x = array([10, 1, 3])# Evaluate with the argument
     return array([10, 1, 3]).item(1) - array([10, 1, 3]).item(0)
my_function(array([10, 1, 3]))
```

This substitution makes the term `x.item(1)` equivalent to `[10, 1, 3].item(1)`, which is just the second item of the array (just the number 1). Similarly, the term `x.item(0)` is equal to `[10, 1, 3].item(0)`, which is the first item of the array (just the number 10).

```
def my_function(x):
     x = array([10, 1, 3])# Evaluate with the argument
     return 1 - 10
my_function(array([10, 1, 3]))
```

Subtracting `x.item(0)` from `x.item(1)` can then be evaluated as 1 - 10 = -9, which is the function's return value.

```
def my_function(x):
     x = array([10, 1, 3])# Evaluate with the argument
     return 1 - 10 # -9
my_function(array([10, 1, 3]))

-9
```

Finally, we conclude that the entire last bolded line evaluates to the number -9. Since there's no more work to be done, we also say that -9 is the output of the entire code snippet itself.