# 6 Writing Custom Functions

## 6.1 Introduction

A **function** is a set of actions that we group together and name. Throughout this course, you've already used a bunch of different functions in python that are built into the language or added through packages: `mean`, `ggplot`, `merge`, etc. In this chapter, we'll be writing our own functions.

## 6.2 Defining a function

### 6.2.1 When to write a function?

If you've written the same code (with a few minor changes, like variable names) more than twice, you should probably write a function instead of copy pasting. The motivation behind this is the "don't repeat yourself" (DRY) principle. There are a few benefits to this rule:

1. Your code stays neater (and shorter), so it is easier to read, understand, and maintain.
2. If you need to fix the code because of errors, you only have to do it in one place.
3. You can re-use code in other files by keeping functions you need regularly in a file (or if you're really awesome, in your own package!)
4. If you name your functions well, your code becomes easier to understand thanks to grouping a set of actions under a descriptive function name.

Consider the following data analysis task done in python:

```
df = pd.DataFrame({'a':np.random.normal(1,1,10), 'b':np.random.normal(2,2,10),'c

df['a'] = (df['a'] - min(df['a']))/(max(df['a']) - min(df['a']))

df['b'] = (df['b'] - min(df['a']))/(max(df['b']) - min(df['b']))

df['c'] = (df['c'] - min(df['c']))/(max(df['c']) - min(df['c']))

df['d'] = (df['d'] - min(df['d']))/(max(df['d']) - min(df['d']))
```

**Check In**

What does this code do?

**Example**

In theory, the code rescales a set of variables to have a range from 0 to 1. But, because of the copy-pasting, the code's author made a mistake and forgot to change an `a` to `b`!

Writing a function will help us avoid these subtle copy-paste errors.

## 6.2.2 Building up the function

To write a function, we first analyze the code to determine how many **inputs** it has

```
df['a'] = (df['a'] - min(df['a']))/(max(df['a']) - min(df['a']))
```

This code has only one input: `df['a']`.

**Check In**

What is the **object structure** of this input?

Now we choose an **argument name** for our new input. It's nice if the argument name reminds the user of what type or structure of object is expected.

In this case, it might help to replace `df$a` with `vec`.

```
vec = df['a']

(vec - min(vec))/(max(vec) - min(vec))
```

```
0    0.000000
1    0.526701
2    1.000000
3    0.797135
4    0.687501
5    0.554046
6    0.570852
7    0.632162
8    0.422267
9    0.451399
Name: a, dtype: float64
```

Then, we make it a bit easier to read, removing duplicate computations if possible (for instance, computing `min` two times) or separating steps to avoid nested parentheses (for instance, computing `max` first).

```
vec = df['a']
min_vec = min(vec)
max_vec = max(vec)

(vec - min_vec)/(max_vec - min_vec)
```

```
0    0.000000
1    0.526701
2    1.000000
3    0.797135
4    0.687501
5    0.554046
6    0.570852
7    0.632162
8    0.422267
9    0.451399
Name: a, dtype: float64
```

Finally, we turn this code into a function with the `def` command:

```
def rescale_vector(vec):

  min_vec = min(vec)
  max_vec = max(vec)

  rescaled_vec = (vec - min_vec)/(max_vec - min_vec)

  return rescaled_vec
```

- The name of the function, `rescale_vector`, describes what the function does - it rescales a vector (pandas `Series` or numpy `array`).
- The function takes one **argument**, named `vec`; any references to this value within the function will use `vec` as the name.
- The code that actually does what your function is supposed to do goes in the **body** of the function, after the `:`. It is important for the body of the function to be *indented*.
- The function **returns** the computed object you want to hand back to the user: in this case, `rescaled_vec`.

> **Note**
>
> Some people prefer to create a final object and then `return:` that object, as we have done above with `rescaled_vec`.
>
> Others prefer fewer objects and longer lines of code, i.e.,
>
> ```
> def rescale_vector(vec):
>
>   min_vec = min(vec)
>   max_vec = max(vec)
>
>   return (vec - min_vec)/(max_vec - min_vec)
> ```
>
> These two approaches will work identically; it's a matter of personal preference.

The process for creating a function is important: first, you figure out how to do the thing you want to do. Then, you simplify the code as much as possible. Only at the end of that process do you create an actual function.

Now, we are able to use our function to avoid the repetition:

```
df['a'] = rescale_vector(df['a'])
df['b'] = rescale_vector(df['b'])
df['c'] = rescale_vector(df['c'])
df['d'] = rescale_vector(df['d'])
```

You probably notice there is still a little bit of repetition here, with `df['a']` appearing on both the left and right side of the `=`. But this is good repetition! When we **assign** or **reassign** an object or column, we want that to be an obvious and deliberate choice.

It's also possible that you might have preferred to keep your original column untouched, and to make new columns for the rescaled data:

```
df['a_scaled'] = rescale_vector(df['a'])
df['b_scaled'] = rescale_vector(df['b'])
df['c_scaled'] = rescale_vector(df['c'])
df['d_scaled'] = rescale_vector(df['d'])
```

### 6.2.3 Documenting your function

When you want to use a function in `python`, but you can't quite remember exactly how it works, you might be in the habit of typing `?fun_name` or `help(fun_name)` to be able to see the **documentation** for that function.

When you write your own function - whether for yourself or for others - it's important to provide reminders of what the function is for and how it works.

We do this by adding text in a very specific structure into the *body* of our function:

```
def rescale_vector(vec):

  """
  Rescales a numeric vector to have a max of 1 and min of 0

  Parameter
  ---------
  vec : num
    A list, numpy array, or pandas Series of numeric data.

  Returns
  -------
  array
    A numpy array containing the rescaled values.
  """

  min_vec = min(vec)
```

```
    max_vec = max(vec)

    rescaled_vec = (vec - min_vec)/(max_vec - min_vec)

    return rescaled_vec
```

A few important aspects of the above to note:

- The use of three quotation marks, `"""` is necessary for this text to be recognized as official documentation.
- The exact structure of the *Parameters* and *Returns* sections, with underlines, is important for consistency with other documentation. There are a few other formatting styles that are generally accepted; we'll stick with the one in this example in this course.
- When listing the *arguments*, a.k.a. *parameters* of our function, we specify the name of the argument, the data type that is expected, and a brief description.
- When listing the *returns* of our function, we specify what object structure is being returned, and a brief description.
- The blank line after the final `"""` is important!

**Check In**

Define the function using the code above, then run `help(rescale_vector)`. Pretty satisfying, right?

## 6.3 Scope

In the previous example, you might have expected

```
rescale_vector(df['a'])
```

to change the column of the `df` object automatically - but it does not, unless you explicitly reassign the results! This is because everything that happens "inside" the function cannot change the status of objects *outside* the function.

This critical programming concept - determining when objects are changed and when they can be accessed - is called **scope**.

In general, an object is only available within the **environment** where it was created. Right now, as you work in your Jupyter notebook or Quarto document, you are interacting with your **global environment**. Run the `globals()` function to see all the objects you have created and libraries you have loaded so far.

Just as someone sitting next to you on a different computer can't access an object in your own global environment, the *body* of a function is its own *function environment*.

Anything that is created or altered in the function environment does not impact the global environment - locally, you only "see" what the function *returns*.

**Just our opinion...**


Like Las Vegas, what happens in a function stays in that function

## 6.3.1 Name Masking

Scope is most clearly demonstrated when we use the same variable name inside and outside a function.

Note that this is 1) bad programming practice, and 2) fairly easily avoided if you can make your names even slightly more creative than `a`, `b`, and so on. But, for the purposes of demonstration, I hope you'll forgive my lack of creativity in this area so that you can see how name masking works.

Consider the following code:

```
a = 10

def myfun(a):
  a = a + 10
  return a


myfun(a)
```

```
a + 5
```

**Check In**

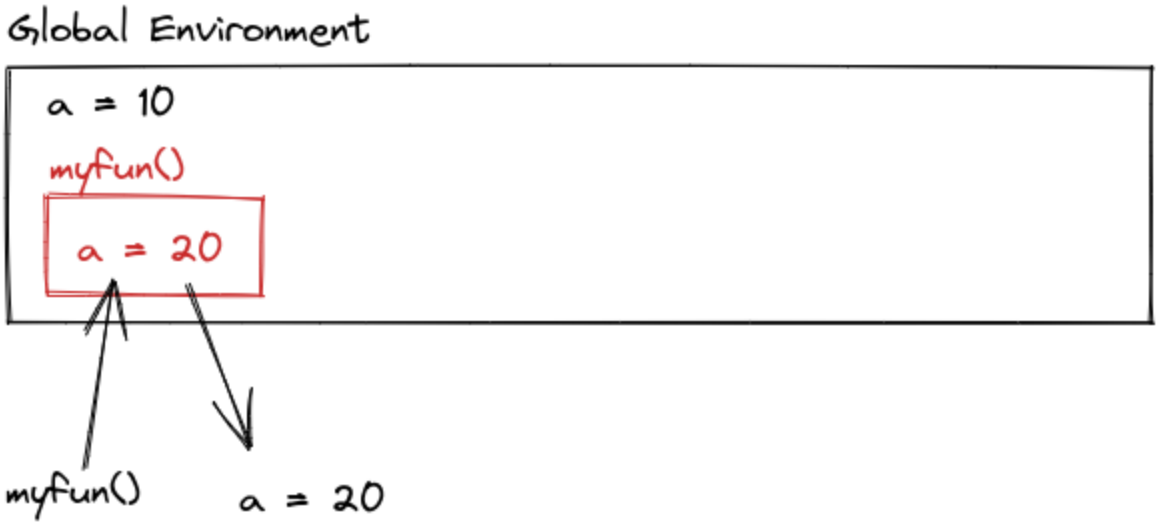
**Without running the code,** what do you think will be printed out by the last two lines of code?

**Check In**

The object named `a` within the *function environment*, i.e. the **parameter** of the function, was altered to be equal to `20`.

Then, the function *returned* the value of its parameter `a`, which was `20`.

Here is a sketch of that idea:




However, the object named `a` in our *global environment* is not impacted. The code `a + 5`, outside the function, still refers to the object in the global environment, which is equal to `10`.

## 6.3.2 Nested environments and scope

One thing to notice is that each time the function is run, it creates a new *local environment*. That is, previous running of a function can't impact future runs of that function.

For example, this code gives the same answer over and over; it does not continue to add `10` to a copy of `a`, because it never alters the object `a` in the global environment **or** the parameters of the other functions' local environments.

```
def myfun(a):
  a = a + 10
  return a
myfun(a)
```

```
myfun(a)
myfun(a)
```

```
NameError: name 'a' is not defined
```

However, *all* of the local environments are considered to be **inside** of your global environment. That is, while they cannot *change* objects in the global environment, they can "see" those objects.

Notice in the below code that we don't pass *any* arguments to `myfun()`. But it is still able to compute `b + 1` and return an answer!

```
b = 10

def myfun():
  return b + 1

myfun()
```

```
11
```

A function will always look in its *local* environment first; then check the *global* for backups:

```
b = 10

def myfun():
  b = 20
  return b + 1

myfun()
```

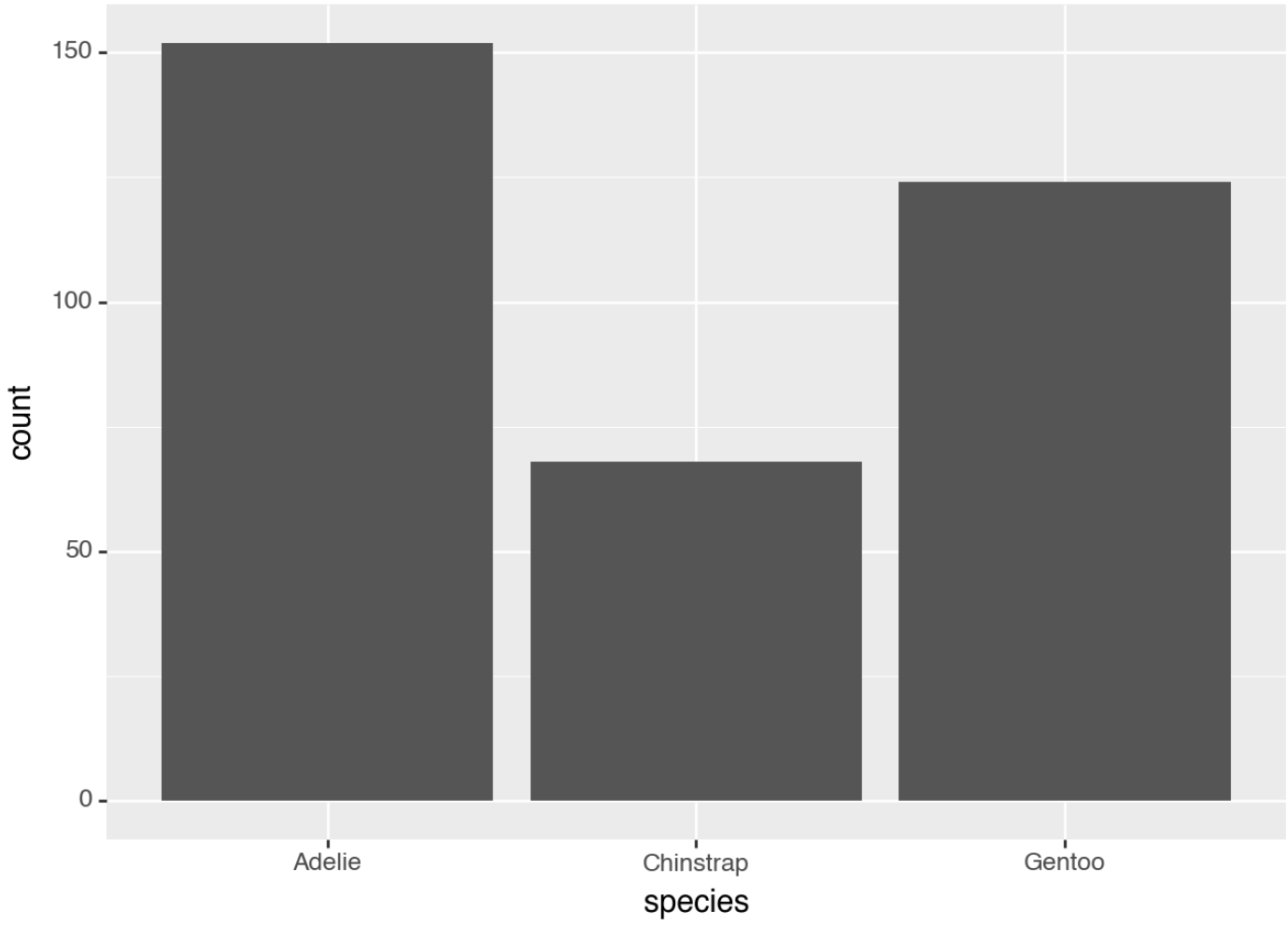
```
21
```

> ⚠ **Warning**
>
> Writing a function that relies on global objects is in general **terrible programming practice**!!!
>
> What if you accidentally change or delete that global object? Suddenly the *exact same code*, i.e. `myfun()` runs differently or not at all.

Alas, although this is bad practice, we quite often "cheat" in Data Science and use global references to our dataset in function shortcuts, e.g.

```
penguins = load_penguins()

def plot_my_data(cat_var):

  plot = (ggplot(penguins, aes(x = cat_var)) + geom_bar())

  return plot
```

```
plot_my_data('species')
```




This “trick” is sometimes called **Dynamic Lookup**. You should mostly avoid it; but when you use it carefully and deliberately in the context of a specific data analysis, it can be a convenient approach.

**Check In**

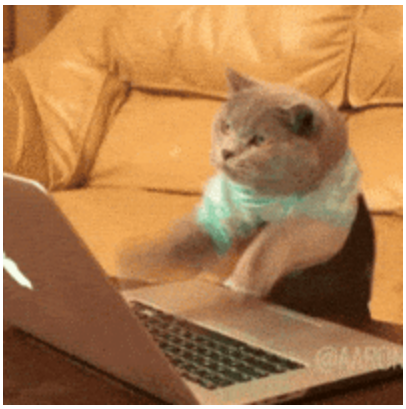
Write a custom function that does the following:

1. Limit the `penguins` dataset to a user-chosen species and/or island.
2. Makes a scatterplot of the penguin bill length and depth.

Try writing a version of this function using *dynamic lookup*, and a version where everything the function needs is passed in directly.

## 6.4 Unit tests

So: You have now written your first custom function. Hooray!



Now, before you move on to further analysis, you have an important job to do. You need to make sure that your function will work - or will break in a helpful way - when users give unexpected input.

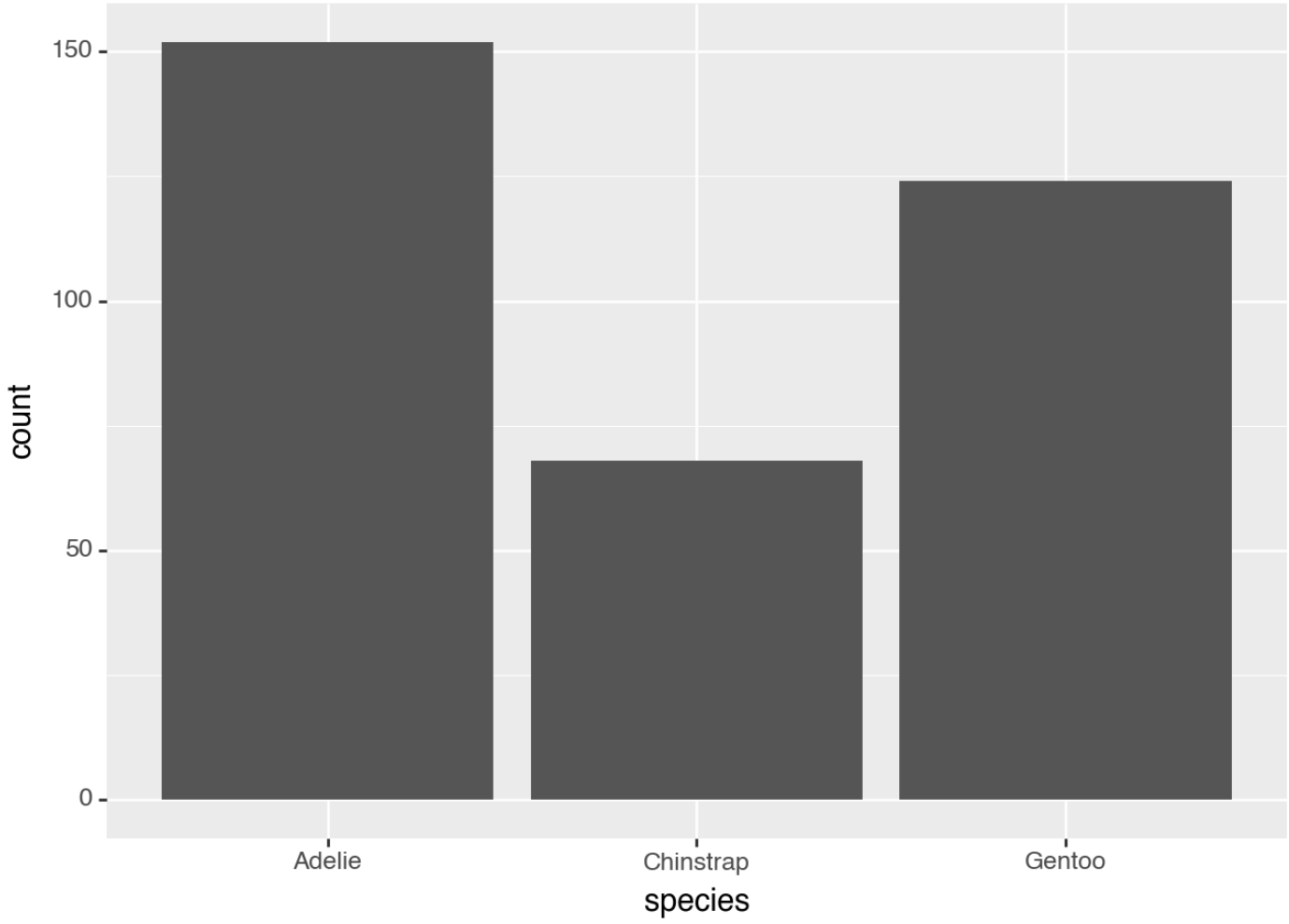
### 6.4.1 Unit testing

The absolute most important thing to do after defining a function is to run some **unit tests**.

This refers to snippets of code that will try your function out on both "ordinary" user input, and strange or unexpected user input.

For example, consider the `plot_my_data` function defined above. We immediately *unit tested* it by running

```
plot_my_data('species')
```




But what if someone tried to enter the name of a variable that is not categorical? Or a variable that doesn't exist in the `penguins` dataset? Or an input that is not a variable name (string)? Or no input at all?

```
plot_my_data('bill_length_mm')

plot_my_data('name')

plot_my_data(5)

plot_my_data(True)

plot_my_data()
```

```
TypeError: plot_my_data() missing 1 required positional argument: 'cat_var'
```

**Example**

Are all of these outputs what you expected? Why or why not? Can you explain why the unexpected behavior happened?

## 6.4.2 Input Validation

When you write a function, you often assume that your parameters will be of a certain type. But as we saw above, you can't guarantee that the person using your function knows that they need a certain type of input, and they might be confused by how your function handles that input. In these cases, it's best to **validate** your function input.

#### 6.4.2.1 `if-else` and `sys.exit`

Generally your approach to validation will be to check some conditions, and **exit** the function if they are not met. The function `exit()` from the `sys` library is a good approach. You want to make sure you write an *informative* error statement in your exit, so that the user knows how to fix what went wrong.

> **Learn More**
>
> This article provides a short guide to writing informative error messages.

The most common condition checking is to make sure the object type is correct - i.e., that the user inputs a string, and that the string refers to a **categorical** (a.k.a. "object") variable in the `penguins` dataset.

Recall that you can "reverse" a boolean (`True` or `False`) value using the `not` statement. Sometimes, it is easier to check if a condition is **not** met than to list all the "good" conditions.

Putting these together, we can check if our user input to `plot_my_data` is what we expect:

```
from sys import exit

def plot_my_data(cat_var):

  if not isinstance(cat_var, str):
    exit("Please provide a variable name in the form of a string.")

  if not (cat_var in penguins.columns):
    exit("The variable provided is not found in the penguins dataset.")

  if not penguins[cat_var].dtypes == 'object':
    exit("Please provide the name of a categorical (object type) variable.")

  plot = (ggplot(penguins, aes(x = cat_var)) + geom_bar())

  return plot
```

> ⚠ **Warning**
>
> Notice that we have used the `isinstance` base python function to check that the user inputted a string; but we have used the `dtype` method to check the data type of the column of data in a `pandas` dataframe.
>
> When checking types and structures, be careful about "special" object types from packages, like `pandas` data frames or `numpy` arrays - they each unfortunately have their own type checking functions *and* their own names for types.

Now, let's retry our unit tests:

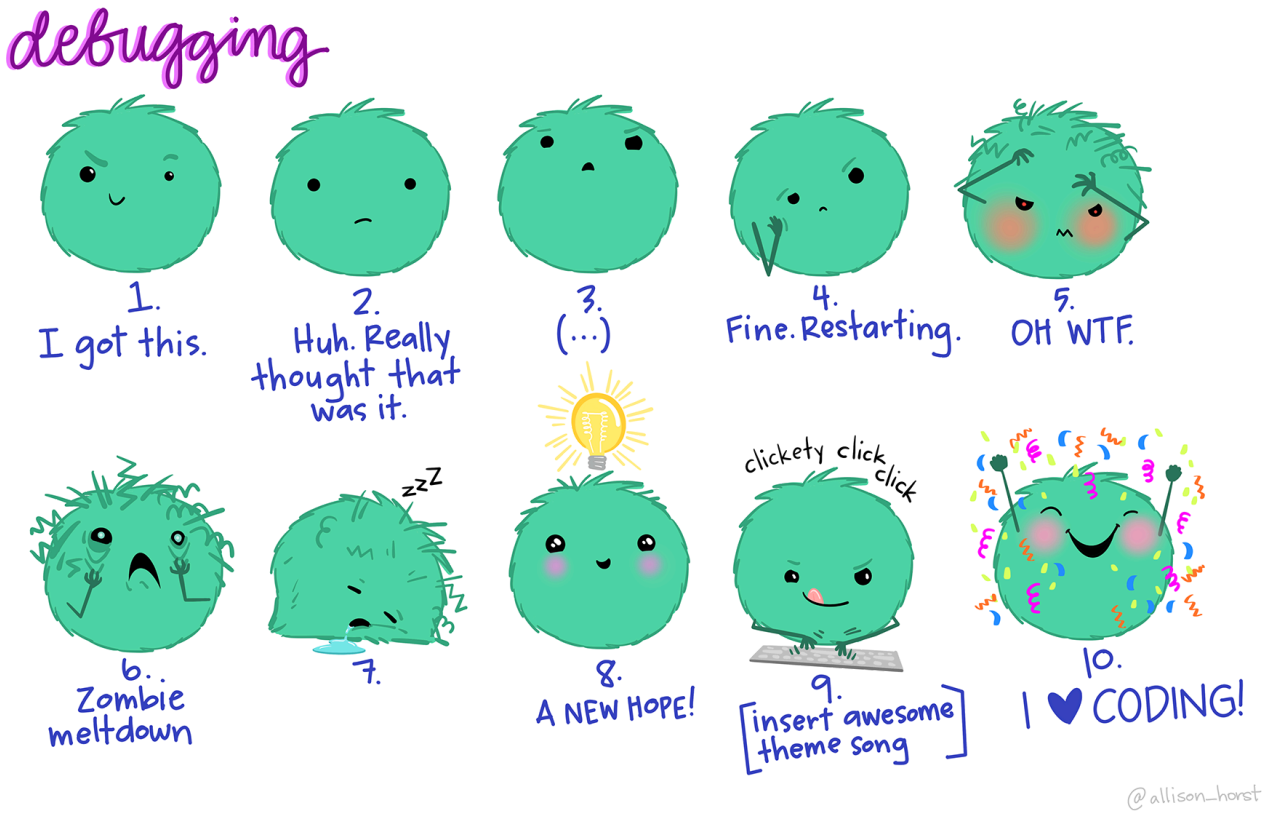
```
plot_my_data('bill_length_mm')

plot_my_data('name')

plot_my_data(5)

plot_my_data(True)

plot_my_data()
```

```
SystemExit: Please provide the name of a categorical (object type) variable.
```

Input validation is one aspect of **defensive programming** - programming in such a way that you try to ensure that your programs don't error out due to unexpected bugs by anticipating ways your programs might be misunderstood or misused. If you're interested, Wikipedia has more about defensive programming.

## 6.5 Debugging

Now that you're writing functions, it's time to talk a bit about debugging techniques. This is a lifelong topic - as you become a more advanced programmer, you will need to develop more advanced debugging skills as well (because you'll find newer and more exciting ways to break your code!).




The faces of debugging (by Allison Horst)

Let's start with the basics: print debugging.

### 6.5.1 Print Debugging

This technique is basically exactly what it sounds like. You insert a ton of print statements to give you an idea of what is happening at each step of the function.

Let's try it out on the `rescale_vector` function.

```
def rescale_vector(vec):

  min_vec = min(vec)
  max_vec = max(vec)

  rescaled = (vec - min_vec)/(max_vec - min_vec)

  return rescaled
```

Suppose we try the following:

```
my_vec = [-1,0,1,2,3]

my_vec = np.sqrt(my_vec)

rescale_vector(my_vec)
```

```
array([nan, nan, nan, nan, nan])
```

You probably have spotted the issue here, but what if it wasn't obvious, and we wanted to know why our function was returning an array of `nan` values.

Is the culprit the min? The max?

```
def rescale_vector(vec):

  min_vec = min(vec)
  print("min: " + str(min_vec))

  max_vec = max(vec)
  print("max: " + str(max_vec))

  rescaled = (vec - min_vec)/(max_vec - min_vec)

  return rescaled
```

```
rescale_vector(my_vec)
```

```
min: nan
max: nan

array([nan, nan, nan, nan, nan])
```

> ⓘ **Note**
>
> Notice how the `print()` statements cause information to get printed out as the function ran, but did not change the `return` value of the function!

Hmmm, both the min and the max were `nan`. This explains why our rescaling introduced all missing values!

So, the issue must be with the user input itself. Let's rewrite our function to take a look at that.

```
def rescale_vector(vec):

  print(vec)

  min_vec = min(vec)
  max_vec = max(vec)

  rescaled = (vec - min_vec)/(max_vec - min_vec)

  return rescaled
```

```
rescale_vector(my_vec)
```

```
[       nan 0.         1.         1.41421356 1.73205081]
```

```
array([nan, nan, nan, nan, nan])
```

Ah-ha! The first value of the input vector is a `nan`.

Ideally, the user would not input a vector with missing values. But it's our job to make sure the function is prepared to handle them.

> **Check In**
>
> Add code to the `rescale_vector` function definition to check if the vector has any `nan` values, and give an informative error message if so.

> **Check In**
>
> Think of other options for handling `nan`s in user input in this function. What are the pros and cons of writing functions that are **opinionated** - i.e., that give errors unless the user input is perfect - versus functions that try to work with imperfect input?

## 6.5.2 Beyond print statements: breakpoints

While `print()` statements work fine as a quick-and-dirty debugging strategy, you will soon get tired of using them, since you have to change your function and reload it every time you want to check something.

A more elegant - and ultimately easier - approach is to "dive in" to the environment of the function itself, where you can interact with the *parameters* in the *local environment* the same way you might interact with your

*global environment*.

To do this, we will set a **breakpoint** in our function. This will cause the function to run until that point, and then stop and let us play around in the environment.

```
def rescale_vector(vec):

  breakpoint()

  min_vec = min(vec)
  max_vec = max(vec)

  rescaled = (vec - min_vec)/(max_vec - min_vec)

  return rescaled
```

**Check In**

Set a breakpoint in your `rescale_vector` code, as above, and then run the function on a vector.

Play around with the interface of the local environment until you get used to this.

### 6.5.3 `pdb` ("python de-bugger")

Although setting breakpoints can be much cleaner and more convenient than several `print()` statements, using `breakpoint()` still required us to modify and reload the function.

The most advanced and clean approach to debugging is to use the `pdb` library to dive straight in to the local environment.

```
def rescale_vector(vec):

  min_vec = min(vec)
  max_vec = max(vec)

  rescaled = (vec - min_vec)/(max_vec - min_vec)

  return rescaled
```

```
import pdb
pdb.run("rescale_vector(my_vec)")
```

**Check In**

Try the above debugging approach.

## 6.6 General Debugging Strategies

> Debugging: Being the detective in a crime movie where you are also the murderer.

The overall process of addressing a bug is:

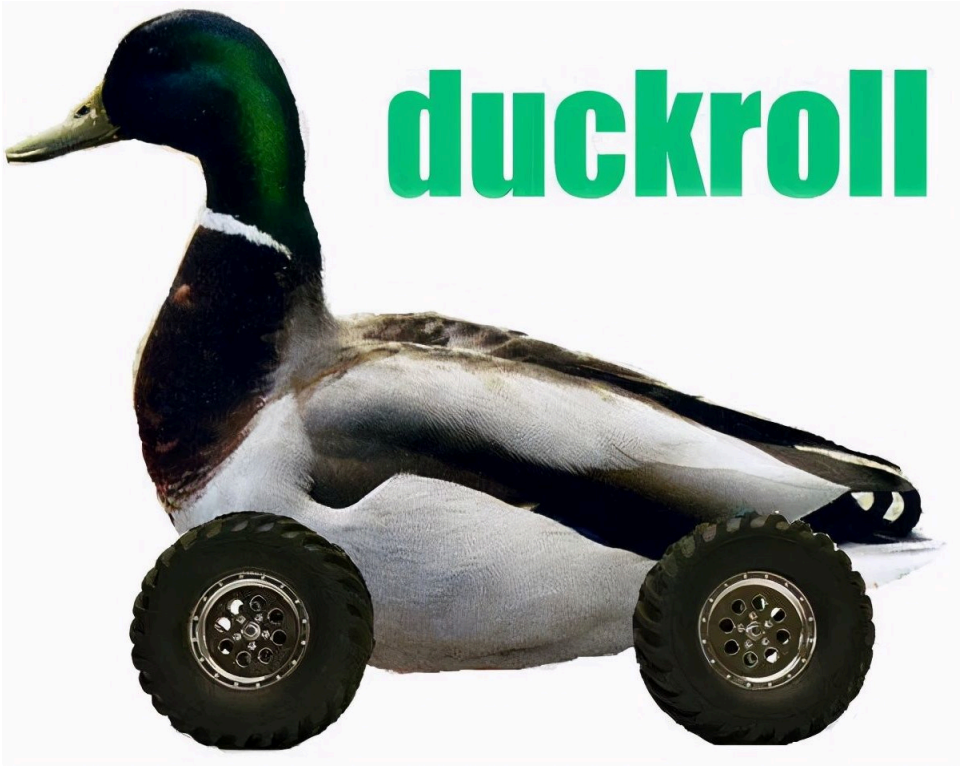
0. **Realize that you have a bug**
1. **Google!** Generally Googling the error + the programming language + any packages you think are causing the issue is a good strategy.
2. **Make the error repeatable**: This makes it easier to figure out what the error is, faster to re-try to see if you fixed it, and easier to ask for help. Unit tests are perfect for this.
3. **Figure out where it is**. Print statements and debuggers help you dig into the function to find the problem area.
4. **Fix it and test it**. The goal with tests is to ensure that the same error doesn't pop back up in a future version of your code. Generate an example that will test for the error, and add it to your documentation.

### 6.6.1 Rubber Duck debugging

Have you ever called a friend or teacher over for help with an issue, only to find that by explaining it to them, you solved it yourself?

Talking through your code out loud is an extremely effective way to spot problems. In programming, we call this **Rubber Duck Debugging**, because coders will sometimes keep a small toy like a rubber duck by their computer, and talk to it while they are alone.

**Just our opinion...**



> This is the original RickRoll. Yes, really.

**Learn More**

A more thorough explanation of rubber duck debugging can be found at gitduck.com.

## 6.6.2 Refactoring your code

> "Divide each difficulty into as many parts as is feasible and necessary to resolve it." -René Descartes, Discourse on Method

In programming, as in life, big, general problems are very hard to solve effectively. Instead, the goal is to break a problem down into smaller pieces that may actually be solveable.

When we redesign our functions to consist of many smaller functions, this is called **refactoring**. Consider the following function:

```
def rescale_all_variables(df):

  for col in df.columns:

    min_vec = min(df[col])
    max_vec = max(df[col])

    df[col] = (df[col] - min_vec)/(max_vec - min_vec)


  return df
```

A much cleaner and easier to read way to use this function would be to use the smaller function `rescale_vector` *inside of* `rescale_all_variables`

```
def rescale_all_variables(df):

  for col in df.columns:
    df[col] = rescale_vector(df[col])

  return df
```

This not only makes the code more readable to humans, it also helps us track down whether the error is happening in the outside function (`rescale_all_variables`) or the inside one (`rescale_vector`)

## 6.6.3 Taking a break!

Do not be surprised if, in the process of debugging, you encounter new bugs. This is a problem that's so well-known it has an xkcd comic.

At some point, getting up and going for a walk may help!

**Practice Activity**

Click here to answer a few questions about function code.

Edit this page