8.3.4 Revisit Functions

Now that you've filtered out bad data, you need to clean up each movie entry so it's in a standard format. If you can make one process broad enough to handle every movie entry, you can apply that process repeatedly for every movie entry. For this task you will create a function.

To keep things organized, we are going to make a **function** that performs that cleaning process.

REWIND

Remember, functions are blocks of code within a script or algorithm that perform a specific task. There are four basic parts to a function:

1. Name
2. Parameters

3. Code block
4. Return value

We're going to expand on some trickier aspects of functions before we make our movie cleaning function.

First, we need to talk about **scope**. Inside the code block of a function, we can use variables that were created outside the function and initialize new variables inside the function.

This is called the "scope" of the variables:

- Variables created outside the function are called **global** variables.
- New variables created inside the function are **local** variables.
- The hierarchy of variables is called the **scope**.

IMPORTANT

> The scope of local variables only works inside the function in which they are created. If we try to access a local variable outside the function in which it's defined, Python will raise a `NameError` because it won't be able to find the function.

Keep this trick in mind with the scope of variables: We can create a local variable with the same name as a global variable. Inside the function, the local variable takes precedence, but the value of the global variable will be unchanged. For example, consider the code below:

```
x = 'global value'

def foo():
    x = 'local value'
```

```
    print(x)

foo()
print(x)
```

The output of this code would be:

```
local value
global value
```

Outside the function, `x` has the value `global value`. Although we define a new function after assigning a value to the global variable `x`, the `x` inside the function is a new variable whose scope is local to `foo()`. Even after calling the `foo()` function, the global variable `x` keeps its original value.

However, we have to be careful with variables we send to a function. Any data structure can be sent as a parameter to a function, including dicts, lists, and tuples, but we must be careful with mutable objects.

CAUTION

> When passing mutable objects like a dict or list as parameters to a function, the function can change the values inside the object.

For example, consider the code below:

```
my_list = [1,2,3]
def append_four(x):
    x.append(4)
append_four(my_list)
print(my_list)
```

The output would be:

```
[1, 2, 3, 4]
```

It changed `my_list` to `[1,2,3,4]`, even outside the function. To leave the original object unchanged, we need to make a copy. For lists, we make a copy with the `list` function; for dicts, we make a copy with the `dict` function, like so:

```
new_list = list(old_list)
new_dict = dict(old_dict)
```

Lambda Functions

There's a special function we can make in Python called **lambda**, which is the most stripped-down kind we can make.

Lambda functions are written in one line and automatically return a value without using the return keyword. Lambda functions have no name and are also known as "anonymous functions."

So, how do we call a function with no name? We don't. There are functions that expect other functions to be sent to them as a parameter, and lambda functions are a way to quickly create a concise function to send as a parameter to another function. We'll return to lambda functions in a later section, but the basic syntax to a lambda function follows:

```
lambda arguments: expression
```

This function will take in an argument and will return the expression. Even though lambda functions are supposed to be anonymous, just this once we'll create a lambda function and assign a name so that we can see how they work. A lambda function that squares a value looks like the following:

```
lambda x: x * x
```

Here, `x` is the argument, and `x * x` is the expression. Let's assign this to a name so that we can use it:

```
square = lambda x: x * x
square(5)
```

The output will be:

```
25
```

If this seems strange, don't worry. The main benefit of lambda functions is that they can be used as one-time-use functions. We'll talk more about lambda functions when we have a more natural use case for them—they'll make more sense at that time.

NOTE

> There are many esoteric topics related to functions, including defining functions within functions, recursion, and functions that create more functions. These are part of the **functional programming** paradigm. Functional programming has its place, but it also has a well-deserved reputation for being confusing and a bit niche. It's good to be aware of functional programming, but we won't rely on its concepts for this module.

Functions are powerful, but enough talk—let's put them to work.