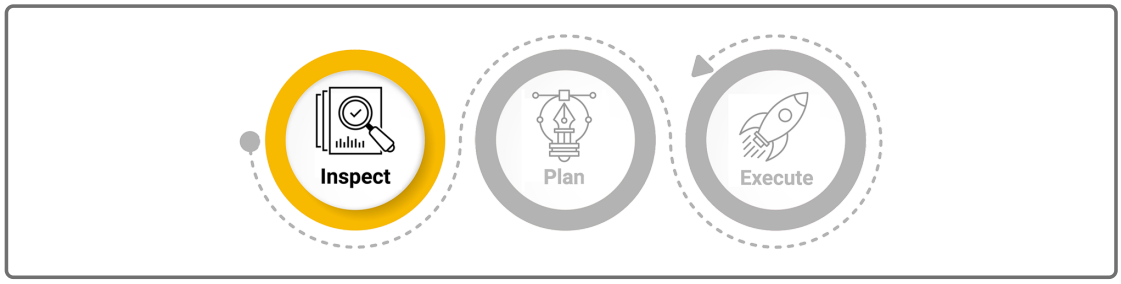
8.3.8 Make a Plan to Convert and Parse the Data

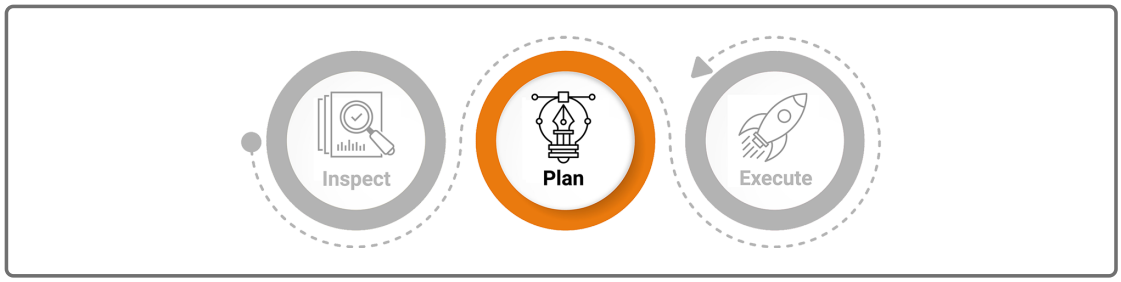
The Wikipedia data is now structured in tabular form, but Britta needs it to have the right data types once it's in the SQL table. For example, we can't do analysis on numeric data if it's stored as a string in the SQL table—it needs to be stored in numeric format. Some of the data also has numeric information written out (like the word "million"). To convert those columns to numbers, the data needs to be parsed.

We've got our DataFrame columns trimmed down to just the ones we want, but some of the columns have data stored as text when it should be a different data type, such as numeric data or datetimes. To convert those columns, we need to understand how to use regular expressions.

Regular expressions are ridiculously powerful, but they can be intimidating at first. There are even seasoned programmers who still don't feel entirely comfortable with regular expressions, but don't worry. We'll break down each step, and before you know it, you'll be rocking regular expressions like a pro.



First, we need to identify which columns need to be converted. `wiki_movies_df.dtypes` will display the data type for each column.



The apply function takes and applies another function to every single value of a Pandas series. Looking through the data, column by column, we see that:

- **Box office** should be numeric.
- **Budget** should be numeric.
- **Release date** should be a date object.
- **Running time** should be numeric.

The box office and budget amounts aren't written in a consistent style, so we're going to need a powerful way to parse their data correctly. We've already dipped a toe into regular expressions; now it's time to dive all the way in.

We'll start on the box office data, which should give us code that we can reuse and tweak for the budget data since they're both currencies. It will be helpful to only look at rows where box office data is defined, so first we'll make a data series that drops missing values with the following:

```
box_office = wiki_movies_df['Box office'].dropna()
```

PAUSE

As a habit, always check the number of data points that exist after you drop any data. Here we have 5,485 movies with box office data. Does that seem like a reasonable amount? (Take a minute to think about it before peeking at the answer.)

Show Answer

Regular expressions only work on strings, so we'll need to make sure all of the box office data is entered as a string. By using the `map()` method, we can see which values are not strings. First, make a `is_not_a_string()` function:

```
def is_not_a_string(x):
    return type(x) != str
```

Then add the following:

```
box_office[box_office.map(is_not_a_string)]
```

The output would be:

```
34                          [US$, 4,212,828]
54      [$6,698,361 (, United States, ), [2]]
74                    [$6,488,144, (US), [1]]
126               [US$1,531,489, (domestic)]
```

```
130                          [US$, 4,803,039]
                        ...
6980                [$99.6, million, [4], [5]]
6994                    [$365.6, million, [1]]
6995                          [$53.8, million]
7015                     [$435, million, [7]]
7048                    [$529.3, million, [4]]
Name: Box office, Length: 135, dtype: object
```

Having to create a new function every time we want to use the `map()` method is cumbersome and interrupts the readability of our code. What we want is a stripped-down, one-line way of writing our functions. Also, we don't need to use it ever again outside of our `map()` call, so we don't need to give it a name. If you think we're talking about types of functions that will work here, you're right.

REWIND

Remember, this is what lambda functions are made for. Instead of creating a new function with a block of code and the `def` keyword, we can create an anonymous lambda function right inside the `map()` call.

Remember, lambda functions don't have a name (because they don't need one) and automatically return a variable. They use the following syntax:

```
lambda arguments: expression
```

So the lambda function version of `is_not_a_string()` is:

```
lambda x: type(x) != str
```

We can update our `map()` call to use the lambda function directly instead of using `is_not_a_string()`:

```
box_office[box_office.map(lambda x: type(x) != str)]
```

The output would be:

```
34                            [US$, 4,212,828]
54      [$6,698,361 (, United States, ), [2]]
74                     [$6,488,144, (US), [1]]
126                [US$1,531,489, (domestic)]
130                           [US$, 4,803,039]
                       ...
6980               [$99.6, million, [4], [5]]
6994                   [$365.6, million, [1]]
6995                         [$53.8, million]
7015                    [$435, million, [7]]
7048                   [$529.3, million, [4]]
Name: Box office, Length: 135, dtype: object
```

From the output, we can see that there are quite a few data points that are stored as lists. There is a `join()` string method that concatenates list items into one string; however, we can't just type `join(some_list)` because the `join()` method belongs to string objects. We need to make a separator string and then call the `join()` method on it. For example, the code would be:

```
some_list = ['One','Two','Three']
'Mississippi'.join(some_list)
```

The outputs would be:

```
'OneMississippiTwoMississippiThree'
```

We'll use a simple space as our joining character and apply the `join()` function only when our data points are lists. The code looks like the following:

```
box_office = box_office.apply(lambda x: ' '.join(x) if type(x) == list else
```

Looking through the data, many of the box office numbers are written either like "\$123.4 million" (or billion) or "\$123,456,789." We'll use regular expressions to find out just how many of each style are in our data.

There is a built-in Python module for regular expressions: `re`. We'll need to import that library, so add the line below to the first cell, with the other import statements, and rerun the cell.

```
import re
```

NOTE

> Python best practices recommend putting all of your import statements at the beginning of your program. If you realize that you need to import another module, it's better to add it to the top rather than have import statements scattered throughout your program, especially when using Jupyter notebooks. Because cells can be run out of order, when import statements are used in other cells, it's easy for them to get lost, and for a cell to have code that depends on another module end up before the cell that imports the module.

Now let's start writing some regular expressions.