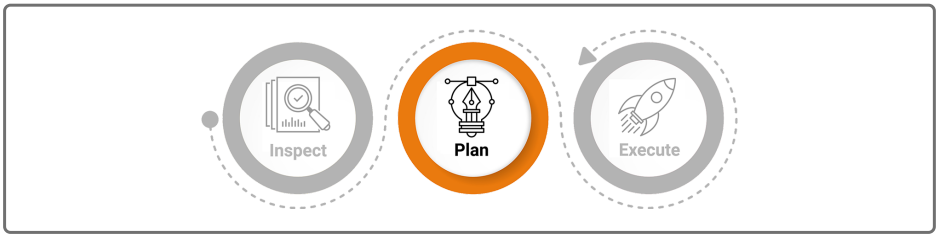
8.3.10 Parse the Box Office Data

Now that you have added regular expressions to your analyst toolbelt, it's time to apply this tool to the box office data.

Remember, there are two main forms the box office data is written in: "$123.4 million" (or billion), and "$123,456,789." We're going to build a regular expression for each form, and then see what forms are left over.

Create the First Form



For the first form, our pattern match string will include six elements in the following order:

1. A dollar sign
2. An arbitrary (but non-zero) number of digits
3. An optional decimal point
4. An arbitrary (but possibly zero) number of more digits
5. A space (maybe more than one)
6. The word "million" or "billion"

We'll translate those rules into a regular expression, step by step.

Step 1: Start with a dollar sign.

The dollar sign is a special character in regular expressions, so we'll need to escape it.

Step 2: Add an arbitrary (but non-zero) number of digits.

We'll add the `\d` character to specify digits only, and the `+` modifier to capture one or more digits. Our regular expression string now appears as `"\$\d+"`.

Step 3: Add an optional decimal point.

Remember, the decimal point is a special character, so it needs to be escaped with a backslash. Since the decimal point is optional, add a question mark modifier after it. Our regular expression string now appears as `"\$\d+\.?"`.

Step 4: Add an arbitrary (but possibly zero) number of more digits.

Once again, we'll use the `\d` character to specify digits only, but now with the `*` modifier because there may be no more digits after the decimal point. Our regular expression string now appears as `"\$\d+\.?\d*"`.

Step 5: Add a space (maybe more than one).

Now we're going to use the `\s` character to match whitespace characters. To be safe, we'll match any number of whitespace characters with the `*` modifier. Our regular expression string now appears as `"\$\d+\.?\d*\s*"`.

Step 6: Add the word "million" or "billion."

Since "million" and "billion" only differ by one letter, we can match it with a character set for the first letter. We specify character sets with square brackets, so we'll add `"[mb]illion"` to the end of our string. Our finished regular expression string now appears as `"\$\d+\.?\d*\s*[mb]illion"`.

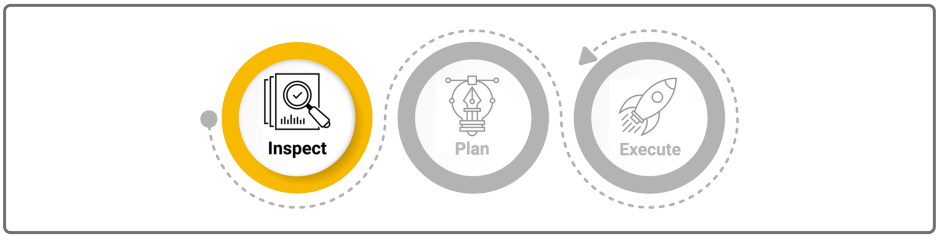
Before moving on, a note about regex playgrounds. Sites like **https://regex101.com (https://regex101.com)** let you test your regex expressions on texts. Try playing around with some of the examples that you have gone over. You will find it helpful to use such a tool when writing regular expressions.

Create a variable `form_one` and set it equal to the finished regular expression string. Because we need the escape characters to remain, we need to preface the string with an `r`.

```
form_one = r'\$\d+\.?\d*\s*[mb]illion'
```

NOTE

> You might be wondering if we're going to miss any box office values that have uppercase letters. Don't worry—when we use the `contains()` method, we will specify an option to ignore case.



Now, to count up how many box office values match our first form. We'll use the `str.contains()` method on `box_office`. To ignore whether letters are uppercase or lowercase, add an argument called `flags`, and set it equal to `re.IGNORECASE`. In case the data is not a string, we'll add the `na=False` argument to parse the non-string data to `False`. Finally, we can

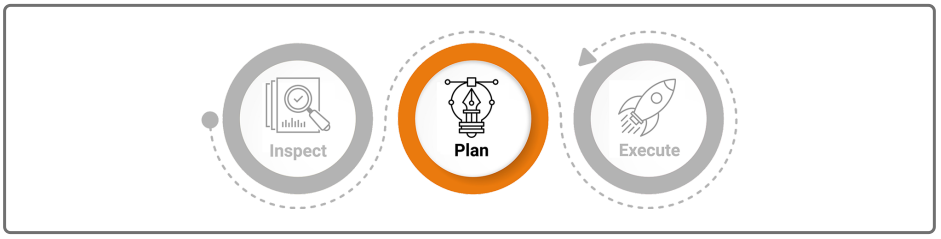
call the `sum()` method to count up the total number that return True. Your code should look like the following:

```
box_office.str.contains(form_one, flags=re.IGNORECASE, na=False).sum()
```

FINDING

There are 3,896 box office values that match the form "$123.4 million/billion."

Create the Second Form



Next, we'll match the numbers of our second form, "$123,456,789." In words, our pattern match string will include the following elements:

1. A dollar sign
2. A group of one to three digits
3. At least one group starting with a comma and followed by exactly three digits

Step 1: Start with a dollar sign.

Once again, we need to escape the dollar sign for it to match. Our regular expression string starts like this: `"\$"`.

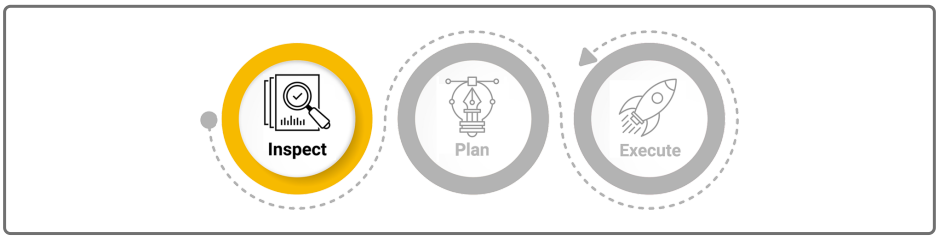
Step 2: Add a group of one to three digits.

We'll use the `\d` character for digits, but this time, we'll modify it with curly brackets to only match one through three repetitions. Our regular expression string now appears as `"\$\d{1,3}"`.

Step 3: Match at least one group starting with a comma and followed by exactly three digits.

To match a comma and exactly three digits, we'll use the string `",\d{3}"`. To match any repetition of that group, we'll put it inside parentheses, and then put a plus sign after the parentheses: `"(,\d{3})+"`. We'll add one more modification to specify that this is a non-capturing group by inserting a question mark and colon after the opening parenthesis: `"(?:,\d{3})+"`. The use of a non-capturing group isn't strictly necessary here, but it eliminates an unwanted warning message in Jupyter Notebook. Our finished regular expression string now appears as `"\$\d{1,3}(?:,\d{3})+"`.

Create another variable `form_two` and set it equal to the finished regular expression string. Don't forget to make it a raw string so Python keeps the escaped characters.



Now count up the number of box office values that match this pattern. Don't forget to put an `r` before the string and set the flags option to include `re.IGNORECASE`.

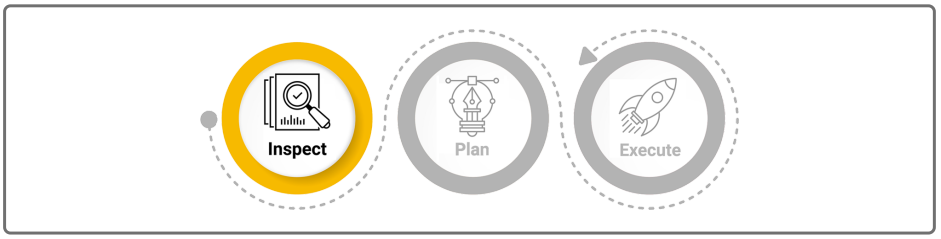
Your code should look like this:

```
form_two = r'\$\d{1,3}(?:,\d{3})+'
box_office.str.contains(form_two, flags=re.IGNORECASE, na=False).sum()
```

FINDING

There are 1,544 box office values that match the form "$123,456,789."

Compare Values in Forms



Most of the box office values are described by either form. Now we want to see which values aren't described by either. To be safe, we should see if any box office values are described by both.

To make our code easier to understand, we'll create two Boolean Series called `matches_form_one` and `matches_form_two`, and then select the box office values that don't match either. First, create the two Boolean series with the following code:

```
matches_form_one = box_office.str.contains(form_one, flags=re.IGNORECASE, na
matches_form_two = box_office.str.contains(form_two, flags=re.IGNORECASE, na
```

Recall the Python logical keywords "not," "and," and "or." Try the following code to see which values in `box_office` don't match either form.

```
# this will throw an error!
box_office[(not matches_form_one) and (not matches_form_two)]
```

The code above will give you a `ValueError` with the explanation "The truth value of a Series is ambiguous." (Unfortunately, the meaning of that error is also ambiguous.)

Instead, Pandas has element-wise logical operators:

- The element-wise negation operator is the tilde: ~ (similar to "not")
- The element-wise logical "and" is the ampersand: &
- The element-wise logical "or" is the pipe: |

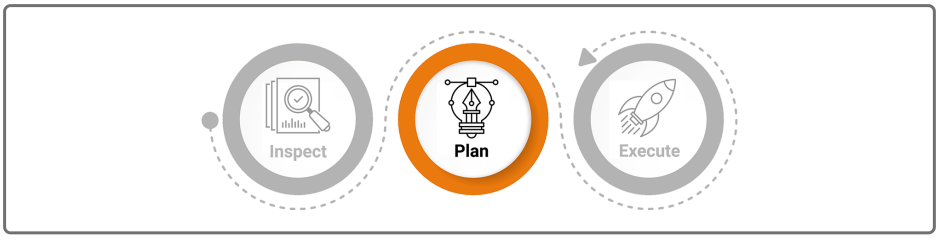
The code we want to use is as follows:

```
box_office[~matches_form_one & ~matches_form_two]
```

We can now see which entries do not fit the expected formatting. Your results should look like the following:

```
34                        US$ 4,212,828
79                             $335.000
110                   $4.35-4.37 million
130                        US$ 4,803,039
600                           $5000 (US)
731                         $ 11,146,270
957                              $ 50,004
1070                          35,254,617
1147    $ 407,618 (U.S.) (sub-total) [1]
1446                        $ 11,829,959
1480                          £3 million
1611                           $520.000
1865                         ¥1.1 billion
2032                                 N/A
2091                                $309
2130               US$ 171.8 million [9]
2257                   US$ 3,395,581 [1]
2263            $ 1,223,034 ( domestic )
2347                           $282.175
2638            $ 104,883 (US sub-total)
```

Fix Pattern Matches



We can fix our pattern matches to capture more values by addressing these issues:

1. Some values have spaces in between the dollar sign and the number.
2. Some values use a period as a thousands separator, not a comma.
3. Some values are given as a range.
4. "Million" is sometimes misspelled as "millon."

1. Some values have spaces in between the dollar sign and the number.

This is easy to fix. Just add `\s*` after the dollar signs. The new forms should look like the following:

```
form_one = r'\$\s*\d+\.?\d*\s*[mb]illion'
form_two = r'\$\s*\d{1,3}(?:,\d{3})+'
```

2. Some values use a period as a thousands separator, not a comma.

This is slightly more complicated, but doable. Simply change `form_two` to allow for either a comma or period as a thousands separator. We'd ordinarily do that by putting the comma and period inside straight brackets [,.], but the period needs to be escaped with a slash [,\.]. The code should match the following:

```
form_two = r'\$\s*\d{1,3}(?:[,\.]\d{3})+'
```

The results will also match values like 1.234 billion, but we're trying to change raw numbers like $123.456.789. We don't want to capture any values like 1.234 billion, so we need to add a negative lookahead group that looks ahead for "million" or "billion" after the number and rejects the match if it finds those strings. Don't forget the space! The new form should look like this:

```
form_two = r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)'
```

3. Some values are given as a range.

To solve this problem, we'll search for any string that starts with a dollar sign and ends with a hyphen, and then replace it with just a dollar sign using the `replace()` method. The first argument in the `replace()` method is the substring that will be replaced, and the second argument in the `replace()` method is the string to replace it with. We can use regular expressions in the first argument by sending the parameter `regex=True`, as shown below.

```
box_office = box_office.str.replace(r'\$.*[---](?![a-z])', '$', regex=True)
```

CAUTION

> Always be wary of parsing dashes. The character you can type on standard keyboards is a hyphen, but some editors will convert them in certain situations to em dashes and en dashes. That is why you are seeing three different types of dashes in the regex expression above. To learn more than you've ever wanted to know about dashes, see the **Wikipedia page for "Dash." (https://en.wikipedia.org/wiki/Dash)** We'll need to put all three into a character set in our replace regular expression.

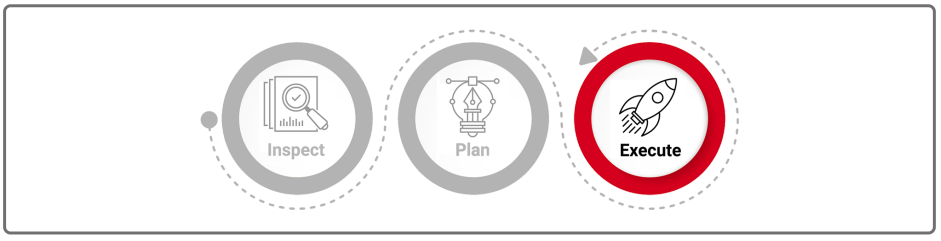
4. "Million" is sometimes misspelled as "millon."

This is easy enough to fix; we can just make the second "i" optional in our match string with a question mark as follows:

```
form_one = r'\$\s*\d+\.?\d*\s*[mb]illi?on'
```

The rest of the box office values make up such a small percentage of the dataset and would require too much time and effort to parse correctly, so we'll just ignore them.

We're finished writing our regular expressions for the box office values. The hard part is over. Save your work and take a quick break if you need one—you earned it!



Extract and Convert the Box Office Values

Now that we've got expressions to match almost all the box office values, we'll use them to extract only the parts of the strings that match. We do this with the `str.extract()` method. This method also takes in a regular expression string, but it returns a DataFrame where every column is the data that matches a capture group. We need to make a regular expression that captures data when it matches either `form_one` or `form_two`. We can do this easily with an f-string.

The f-string `f'{form_one}|{form_two}'` will create a regular expression that matches either `form_one` or `form_two`, so we just need to put the whole thing in parentheses to create a capture group. Our final string will be `f'({form_one}|{form_two})'`, and the full line of code to extract the data follows:

```
box_office.str.extract(f'({form_one}|{form_two})')
```

Now we need a function to turn the extracted values into a numeric value. We'll call it `parse_dollars`, and `parse_dollars` will take in a string and return a floating-point number. We'll start by making a skeleton function with comments explaining each step, and then fill in the steps with actual code.

```
def parse_dollars(s):
    # if s is not a string, return NaN

    # if input is of the form $###.# million

        # remove dollar sign and " million"

        # convert to float and multiply by a million

        # return value

    # if input is of the form $###.# billion

        # remove dollar sign and " billion"

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###

        # remove dollar sign and commas

        # convert to float

        # return value

    # otherwise, return NaN
```

Since we're working directly with strings, we'll use the `re` module to access the regular expression functions. We'll use `re.match(pattern, string)` to see if our string matches a pattern. To start, we'll make some

small alterations to the forms we defined, splitting the million and billion matches from form one.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " million"

        # convert to float and multiply by a million

        # return value

    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flags=r

        # remove dollar sign and commas

        # convert to float

        # return value

    # otherwise, return NaN
    else:
        return np.nan
```

Next, we'll use `re.sub(pattern, replacement_string, string)` to remove dollar signs, spaces, commas, and letters, if necessary.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):
```

```
        # remove dollar sign and " million"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a million

        # return value

    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a billion

        # return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flags=r

        # remove dollar sign and commas
        s = re.sub('\$|,','', s)

        # convert to float

        # return value

    # otherwise, return NaN
    else:
        return np.nan
```

Finally, convert all the strings to floats, multiply by the right amount, and return the value.

```
def parse_dollars(s):
    # if s is not a string, return NaN
    if type(s) != str:
        return np.nan

    # if input is of the form $###.# million
    if re.match(r'\$\s*\d+\.?\d*\s*milli?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " million"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a million
        value = float(s) * 10**6

        # return value
        return value
```

```
    # if input is of the form $###.# billion
    elif re.match(r'\$\s*\d+\.?\d*\s*billi?on', s, flags=re.IGNORECASE):

        # remove dollar sign and " billion"
        s = re.sub('\$|\s|[a-zA-Z]','', s)

        # convert to float and multiply by a billion
        value = float(s) * 10**9

        # return value
        return value

    # if input is of the form $###,###,###
    elif re.match(r'\$\s*\d{1,3}(?:[,\.]\d{3})+(?!\s[mb]illion)', s, flags=r

        # remove dollar sign and commas
        s = re.sub('\$|,','', s)

        # convert to float
        value = float(s)

        # return value
        return value

    # otherwise, return NaN
    else:
        return np.nan
```

Now we have everything we need to parse the box office values to numeric values.

First, we need to extract the values from `box_office` using `str.extract`. Then we'll apply `parse_dollars` to the first column in the DataFrame returned by `str.extract`, which in code looks like the following:

```
wiki_movies_df['box_office'] = box_office.str.extract(f'({form_one}|{form_tw
```

If you then run wiki_movies_df['box_office'], the output should look like this:

```
0       2.140000e+07
1       2.700000e+06
2       5.771809e+07
3       7.331647e+06
```

```
4       6.939946e+06
5                NaN
...
7071    4.190000e+07
7072    7.610000e+07
7073    3.840000e+07
7074    5.500000e+06
7075             NaN
```

We no longer need the Box Office column, so we'll just drop it:

```
wiki_movies_df.drop('Box office', axis=1, inplace=True)
```