8.3.11 Parse Budget Data

Luckily, you already did a lot of the heavy lifting for parsing the budget data when you parsed the box office data. You'll use the same pattern matches and see how many budget values are in a different form.

Luckily, we've already done a lot of the heavy lifting for parsing the budget data when we parsed the box office data. We'll use the same pattern matches and see how many budget values are in a different form. First, we need to preprocess the budget data, just like we did for the box office data.

Create a budget variable with the following code:

```
budget = wiki_movies_df['Budget'].dropna()
```

Convert any lists to strings:

```
budget = budget.map(lambda x: ' '.join(x) if type(x) == list else x)
```

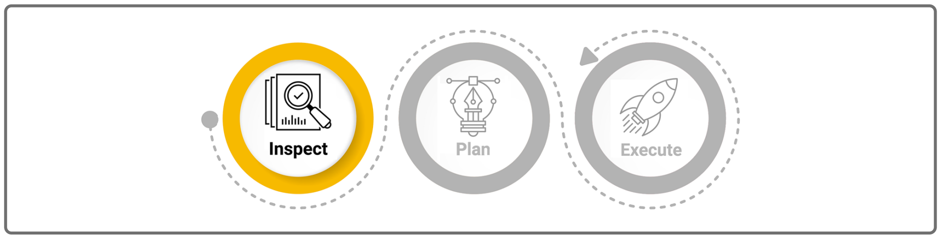
Then remove any values between a dollar sign and a hyphen (for budgets given in ranges):

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

Now test your skills in the following Skill Drill.

SKILL DRILL

Use the same pattern matches that you created to parse the box office data, and apply them without modifications to the budget data. Then, look at what's left.



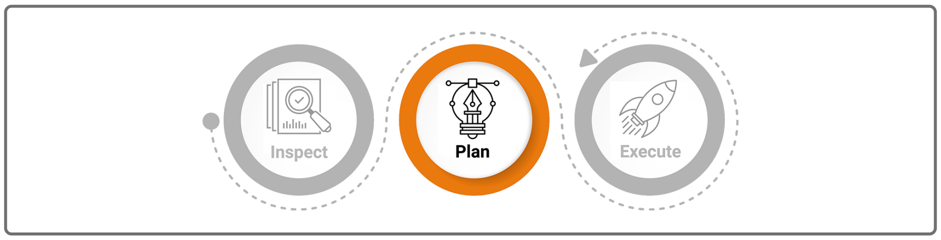
Your code should look like this:

```
matches_form_one = budget.str.contains(form_one, flags=re.IGNORECASE, na=Fal
matches_form_two = budget.str.contains(form_two, flags=re.IGNORECASE, na=Fal
budget[~matches_form_one & ~matches_form_two]
```

The output should look like this:

```
136                          Unknown
204       60 million Norwegian Kroner
478                          Unknown
973                 $34 [3] [4] million
1126                    $120 [4] million
1226                         Unknown
1278                             HBO
1374                      £6,000,000
1397                      13 million
1480                     £2.8 million
1734                    CAD2,000,000
1913        PHP 85 million (estimated)
1948                     102,888,900
1953                    3,500,000 DM
1973                      £2,300,874
2281                      $14 milion
2451                      £6,350,000
3144                     € 40 million
3360                    $150 [6] million
3418                         $218.32
```

Not bad! That parsed almost all of the budget data. However, there's a new issue with the budget data: citation references (the numbers in square brackets).



We can remove those fairly easily with a regular expression.

What regular expression will match a number within square brackets?

- ○ `“[0-9]”`
- ○ `“[\d+]”`
- ○ `“\[\d\]”`
- ○ `“\[\d+\]”`

Check Answer

Finish ▶

Remove the citation references with the following:

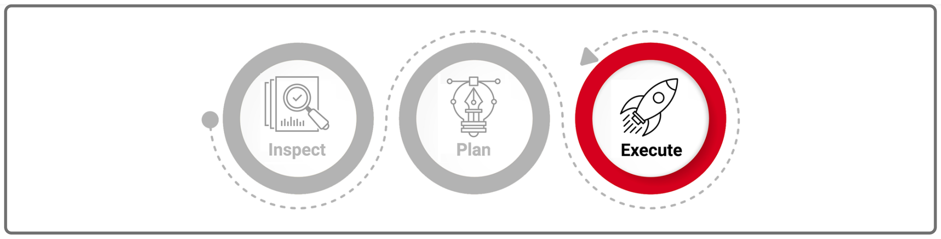
```
budget = budget.str.replace(r'\[\d+\]\s*', '')
budget[~matches_form_one & ~matches_form_two]
```

There will be 30 budgets remaining.

PAUSE

Is it worth our time to try and parse what we can out of these remaining 30 budget values, or should we just drop them?

Show Answer



Everything is now ready to parse the budget values. We can copy the line of code we used to parse the box office values, changing "box_office" to "budget":

```
wiki_movies_df['budget'] = budget.str.extract(f'({form_one}|{form_two})', fl
```

We can also drop the original Budget column.

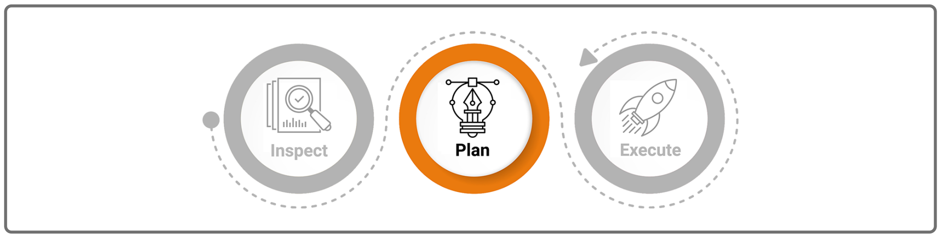
```
wiki_movies_df.drop('Budget', axis=1, inplace=True)
```

Parse Release Date

Parsing the release date will follow a similar pattern to parsing box office and budget, but with different forms.

First, make a variable that holds the non-null values of Release date in the DataFrame, converting lists to strings:

```
release_date = wiki_movies_df['Release date'].dropna().apply(lambda x: ' '.j
```



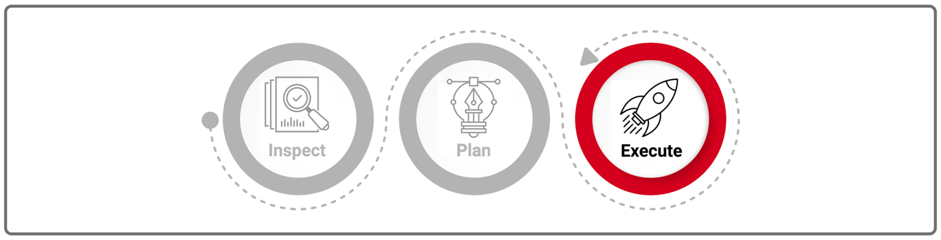
The forms we'll be parsing are:

1. Full month name, one- to two-digit day, four-digit year (i.e., January 1, 2000)

2. Four-digit year, two-digit month, two-digit day, with any separator (i.e., 2000-01-01)
3. Full month name, four-digit year (i.e., January 2000)
4. Four-digit year

SKILL DRILL

Try to figure out the regular expressions for each form before moving on. Test them in your Jupyter Notebook with some test strings.



One way to parse those forms is with the following:

```
date_form_one = r'(?:January|February|March|April|May|June|July|August|Septe
date_form_two = r'\d{4}.[01]\d.[0123]\d'
date_form_three = r'(?:January|February|March|April|May|June|July|August|Sep
date_form_four = r'\d{4}'
```

Of the four regular expressions, the first matches the `month, dd, yyyy` format. The second matches these two formats, for example: `yyyy-mm-dd` and `yyyy/mm/dd`. The third matches `month yyyy`. The fourth matches `yyyy`. For longer regex expressions, you might consider using the re.VERBOSE option, which allows you to comment on each component of a regex. See **this Stack Overflow discussion (https://stackoverflow.com/questions/13851794/how-to-implement-a-verbose-regex-in-python)** for an example.

And then we can extract the dates with:

```
release_date.str.extract(f'({date_form_one}|{date_form_two}|{date_form_three
```

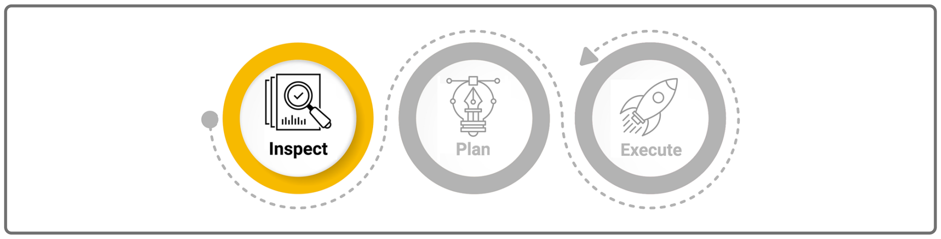
Instead of creating our own function to parse the dates, we'll use the built-in `to_datetime()` method in Pandas. Since there are different date formats, set the `infer_datetime_format` option to `True`. The date formats we've targeted are among those that the to_datetime() function can recognize, which explains the `infer_datetime_format=True` argument below.

```
wiki_movies_df['release_date'] = pd.to_datetime(release_date.str.extract(f'(
```

Parse Running Time

First, make a variable that holds the non-null values of Release date in the DataFrame, converting lists to strings:

```
running_time = wiki_movies_df['Running time'].dropna().apply(lambda x: ' '.j
```



It looks like most of the entries just look like "100 minutes." Let's see how many running times look exactly like that by using string boundaries.

```
running_time.str.contains(r'^\d*\s*minutes$', flags=re.IGNORECASE, na=False)
```

The above code returns 6,528 entries. Let's get a sense of what the other 366 entries look like.

```
running_time[running_time.str.contains(r'^\d*\s*minutes$', flags=re.IGNORECA
```

The output should look like this:

```
9                                             102 min
26                                             93 min
28                                            32 min.
34                                            101 min
35                                             97 min
                                 ...
6500       114 minutes [1] 120 minutes (extended edition)
6643                                         104 mins
6709    90 minutes (theatrical) [1] 91 minutes (unrate...
7057    108 minutes (Original cut) 98 minutes (UK cut)...
7075                Variable; 90 minutes for default path
Name: Running time, Length: 366, dtype: object
```

Let's make this more general by only marking the beginning of the string, and accepting other abbreviations of "minutes" by only searching up to the letter "m."

```
running_time.str.contains(r'^\d*\s*m', flags=re.IGNORECASE, na=False).sum()
```

That accounts for 6,877 entries. The remaining 17 follow:

```
running_time[running_time.str.contains(r'^\d*\s*m', flags=re.IGNORECASE, na=
```

The output should look like the following.

```
668                   UK:84 min (DVD version) US:86 min
727                     78-102 min (depending on cut)
840                     Varies (79 [3] -84 [1] minutes)
1347                                          25 : 03
1442    United States: 77 minutes Argentina: 94 minute...
1498                                        1hr 35min
1550                                           varies
1773                  Netherlands:96 min, Canada:95 min
1776                                    approx. 14 min
2272                                       1 h 43 min
2990                                           1h 48m
3919                                          4 hours
4418    US domestic version: 86 minutes Original versi...
4959    Theatrical cut: 97 minutes Unrated cut: 107 mi...
5416                  115 [1] /123 [2] /128 [3] minutes
5439                                  1 hour 32 minutes
7038                Variable; 90 minutes for default path
```

We can capture some more of these by relaxing the condition that the pattern has to start with at the beginning of the string, but the entries with hours and minutes listed separately will give erroneous data.

What is the new regular expression that relaxes the condition of patterns starting at the beginning of the string?

- ○ '^\d*\s*m'
- ○ r'\d*\s*m'
- ○ r'^\d\s*m'
- ○ r'^\d*\sm'

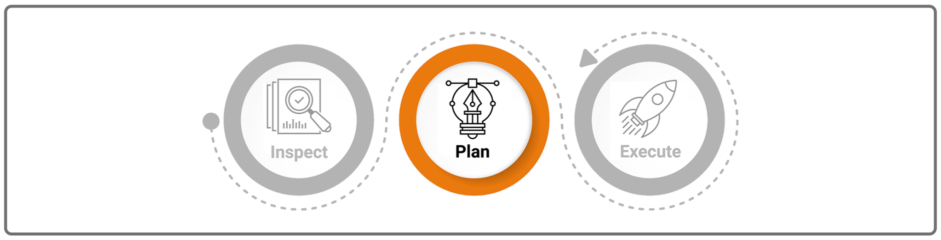
Check Answer

Finish ▶

Even though it's a very small number of entries, it's not too hard to parse, so we'll go ahead and parse those, too.

NOTE

> This is another judgment call. It's only 17 entries out of almost 7,000, so it's highly unlikely that our analysis will be affected by just ignoring these data points. In a time crunch, it would be perfectly acceptable to just move on. However, it's not very difficult to parse these new forms, and we'll have more flexible code if we do. If we decide to do another, larger scrape of Wikipedia data, it's entirely possible that a significant portion have their runtime formatted this way.

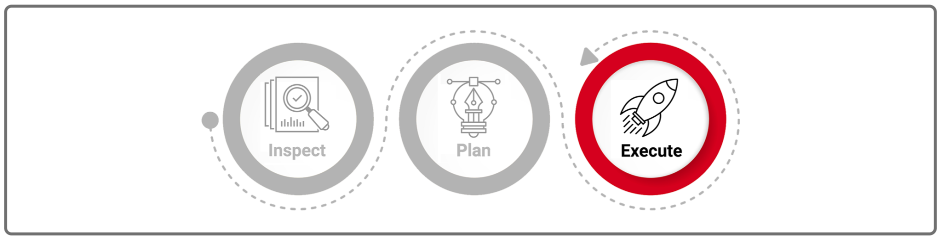


We can match all of the hour + minute patterns with one regular expression pattern. Our pattern follows:

1. Start with one or more digits.
2. Have an optional space after the digit and before the letter "h."

3. Capture all the possible abbreviations of "hour(s)." To do this, we'll make every letter in "hours" optional except the "h."
4. Have an optional space after the "hours" marker.
5. Have an optional number of digits for minutes.

As a pattern, this looks like `"\d+\s*ho?u?r?s?\s*\d*"`.



With our new pattern, it's time to extract values. We only want to extract digits, and we want to allow for both possible patterns. Therefore, we'll add capture groups around the `\d` instances as well as add an alternating character. Our code will look like the following.

```
running_time_extract = running_time.str.extract(r'(\d+)\s*ho?u?r?s?\s*(\d*)|
```

Unfortunately, this new DataFrame is all strings, we'll need to convert them to numeric values. Because we may have captured empty strings, we'll use the `to_numeric()` method and set the errors argument to `'coerce'`. Coercing the errors will turn the empty strings into Not a Number (NaN), then we can use `fillna()` to change all the NaNs to zeros.

```
running_time_extract = running_time_extract.apply(lambda col: pd.to_numeric(
```

Now we can apply a function that will convert the hour capture groups and minute capture groups to minutes if the pure minutes capture group is zero, and save the output to `wiki_movies_df`:

```
wiki_movies_df['running_time'] = running_time_extract.apply(lambda row: row[
```

Finally, we can drop `Running time` from the dataset with the following code:

```
wiki_movies_df.drop('Running time', axis=1, inplace=True)
```

The Wikipedia dataset has been cleaned! Save your notebook and give yourself a pat on the back—you just did some hard work.