Data 100 Debugging Guide

Yash Dave Lillian Weng Wesley Zheng Emrie Loh

Table of contents

About

This text offers pointers for keyboard shortcuts or common mistakes that accompany the coursework in the Summer 2025 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

Inspiration for this guide was taken from the UC San Diego course DSC 10: Principles of Data Science and their debugging guide.

If you spot any typos or would like to suggest any changes, please email us at **data100.instructors@berkeley.edu**.

1 Jupyter 101

> **Note**
>
> If you're using a MacBook, replace `ctrl` with `cmd`.

1.1 Shortcuts for Cells

For the following commands, make sure you're in command mode. You can enter this mode by pressing `esc`.

- `a`: create a cell above
- `b`: create a cell below
- `dd`: delete current cell
- `m`: convert a cell to markdown (text cell)
- `y`: convert a cell to code

1.2 Running Cells

For individual cells,

- `ctrl` + `return`: run the current cell
- `shift` + `return`: run the current cell and move to the next cell

To run all cells in a notebook:

- In the menu bar on the left, click `Run`. From here, you have several options. The ones we use most commonly are:
  - `Run All Above Selected Cell`: this runs every cell above the selected cell
  - `Run Selected Cell and All Below`: this runs the selected cell and all cells below
  - `Run All`: this runs every cell in the notebook from top-to-bottom

1.3 Saving your notebook

Jupyter autosaves your work, but there can be a delay. As such, it's a good idea to save your work as often as you remember and especially before submitting assignments. To do so, press `ctrl` + `s`.

1.4 Restarting Kernel

In the menu bar on the left, click `Kernel`. From here, you have several options. The ones we use most commonly are:

- `Restart Kernel...`
- `Restart Kernel and Run up to Selected Cell`
- `Restart Kernel and Run All Cells`

1.5 Automatically Closing Brackets

Many IDEs like VSCode have a functionality that automatically closes brackets. For example, pressing `(`, `{`, or `[` would automatically add the second bracket at the other end `)`, `}`, and `]`, respectively. Datahub does not have this functionality turned on by default, but you can do so by going into `Settings` -> `Auto Close Brackets`. If you see a check mark to the left of `Auto Close Brackets`, then it's enabled.

2 Jupyter / Datahub

2.1 My kernel died, restarted, or is very slow

Jupyterhub connects you to an external container to run your code. That connection could be slow/severed because:

1. you haven't made any changes to the notebook for a while
2. a cell took too much time to run
3. a cell took up too many resources to compute

When you see a message like this:

1. Either press the "Ok" button or reload the page
2. Restart your kernel
3. Rerun your cells

Note that you may lose some recent work if your kernel restarted when you were in the middle of editing a cell. As such, we recommend saving your work as often as possible.

If this does not fix the issue, it could be a problem with your code, usually the last cell that executed before your kernel crashed. Double check your logic, and feel free to make a private post on Ed if you're stuck!

2.2 I can't edit a cell

We set some cells to read-only mode prevent accidental modification. That said, do not modify the given cells unless instructed to do so. If instructed, to make the cell writeable,

1. Click the cell
2. Click setting on the top right corner
3. Under "Common Tools", you can toggle between "Editable" (can edit the cell) and "Read-Only" (cannot edit the cell)

2.3 My text cell looks like code

If you double-click on a text (markdown) cell, it'll appear in its raw format. To fix this, simply run the cell. If this doesn't fix the problem, check out the commonly asked question below.

2.4 My text cell changed to a code cell / My code cell changed to a text cell

Sometimes, a text (markdown) cell was changed to a code cell, or a code cell can't be run because it's been changed to a text (markdown) or raw cell. To fix this, toggle the desired cell type in the top bar.

2.5 Why does running a particular cell cause my kernel to die?

If one particular cell seems to cause your kernel to die, this is likely because the computer is trying to use more memory than it has available. For instance: your code is trying to create a gigantic array. To prevent the entire server from crashing, the kernel will "die". This is an indication that there is a mistake in your code that you need to fix.

2.6 I accidentally deleted something in a cell that was provided to me – how do I get it back?

Suppose you're working on Lab 5. One solution is to go directly to DataHub and rename your lab05 folder to something else, like lab05-old. Then, click the Lab 5 link on the course website again, and it'll bring you to a brand-new version of Lab 5. You can then copy your work from your old Lab 5 to this new one, which should have the original version of the assignment.

Alternatively, you can access this public repo and navigate to a blank copy of the assignment you were working on. In the case of Lab 5 for example, the notebook would be located at `lab/lab05/lab05.ipynb`. You can then check and copy over the contents of the deleted cell into a new cell in your existing notebook.

2.7 "Click here to download zip file" is not working

When this happens, you can download the zip file through the menu on the left.

Right click on the generated zip file and click "Download".

2.8 Autograder could not locate my PDF

Sometimes when running the `grader.export(run_tests=True)` cell at the end of the notebook, you run into an error where the autograder could not locate the PDF:

To fix this, make sure you have not accidentally converted any cells to `raw` type. All cells should either be `markdown` or `code` cells. You can easily convert cells by following these instructions earlier in the debugging guide.

2.9 I can't export my assignment as a PDF due to a `LatexFailed` error

Occasionally when running the `grader.export(run_tests=True)` cell at the end of the notebook, you run into an error where the PDF failed to generate:

Converting a Jupyter notebook to a PDF involves formatting some of the markdown text in LaTeX. However, this process will fail if your free response answers have (unresolved) LaTeX characters like `\n`, `$`, or `$$`. There are several ways to resolve this:

1. **Export the notebook as a PDF**: In the upper left hand menu, go to `File` -> `Save and Export Notebook As` -> `PDF`. Upload this file to Gradescope under the "Submit PDF" option.
2. **Print the notebook from HTML**: In the upper left hand menu, go to `File` -> `Save and Export Notebook As` -> `HTML`. In the new tab that will open up, print the website by typing `ctrl` + `p` (Windows) or `cmd` + `p` (Mac).
3. **Take screenshots**: If you're short on time, your best bet is to take screenshots of your free response answers. When submitting to Gradescope, choose the "Submit Images" options instead of the "Submit PDF" option.
4. **Removing special LaTeX characters**: If you have more time and would like the Datahub-generated PDF, please remove any special LaTeX characters from your free response answers.

If you use an alternate form of submission listed above, you don't need to worry if you can't select pages or if the selection doesn't align. We'll manually look through your submission when grading, and will account for that.

2.10 I can't open Jupyter: `HTTP ERROR 431`

If this happens, try clearing your browser cache or opening Datahub in an incognito window.

2.11 Datahub is not loading

If your link to Datahub is not loading, go to https://data100.datahub.berkeley.edu/hub/home and restart your server.

3 Autograder and Gradescope / Pensieve

Citation

Many of these common questions were taken and modified from the UC San Diego course DSC 10: Principles of Data Science and their debugging guide.

3.1 Autograder

3.1.1 Understanding autograder error messages

When you pass a test, you'll see a nice message and a cute emoji!

When you don't, however, the message can be a little confusing.

The best course of action is to find the test case that failed and use that as a starting point to debug your code.

In the example above, we see that the test case in green, `max_swing in set(bus['name'])`, is not passing. The actual output (in blue) is often hard to parse, so the best course of action is to:

1. Make a new (temporary) cell after the `grader.check(...)` cell. Please do not make a new cell in between the given code cell and the `grader.check(...)` cell, as it could mess with the results.
2. Copy and paste the failing test case into your temporary cell and run it.
    a. If it's giving you an error like in the example above, look at the last line of the error and use the Debugging Guide's search functionality in the top left menu to find the corresponding guide.
    b. If it's not giving you an error, it'll likely give you an output like `False`. This means that your code does not cause an error (yay!), but it returns an incorrect output. In these casses, inspect each individual element of the test case. The example above checks if `max_swing` is in `set(bus['name'])`, so it might be a good idea to display both variables and do a visual check.
    c. If you're still having issues, post on Ed!
3. After your `grader.check(...)` passes, feel free to delete the temporary cell.

3.1.2 Why do I get an error saying "`grader is not defined`"?

If it has been a while since you've worked on an assignment, the kernel will shut itself down to preserve memory. When this happens, all of your variables are forgotten, including the grader. That's OK. The easiest way to fix this is by restarting your kernel and rerunning all the cells. To do this, in the top left menu, click `Kernel` -> `Restart and Run All Cells`.

3.1.3 I'm positive I have the right answer, but the test fails. Is there a mistake in the test?

While you might see the correct answer displayed as the result of the cell, chances are your solution isn't being stored in the answer variable. Make sure you are assigning the result to the answer variable and that there are no typos in the variable name. Finally, restart your kernel and run all the cells in order: `Kernel` -> `Restart and Run All Cells`.

3.1.4 Why does the last `grader.export` cell fail if all previous tests passed?

This can happen if you "overwrite" a variable that is used in a question. For instance, say Question 1 asks you to store your answer in a variable named `stat` and, later on in the notebook, you change the value of `stat`; the test right after Question 1 will pass, but the test at the end of the notebook will fail. It is good programming practice to give your variables informative names and to avoid repeating the same variable name for more than one purpose.

3.1.5 Why does a notebook test fail now when it passed before, and I didn't change my code?

You probably ran your notebook out of order. Re-run all previous cells in order, which is how your code will be graded.

3.2 Gradescope/Pensieve

When submitting to Gradescope or Pensieve, there are often unexpected errors that make students lose more points than expected. Thus, it is imperative that you **stay on the submission page until the autograder finishes running**, and the results are displayed.

3.2.1 Why did a Gradescope/Pensieve test fail when all the Jupyter notebook's tests passed?

This can happen if you're running your notebook's cells out of order. The autograder runs your notebook from top-to-bottom. If you're defining a variable at the bottom of your notebook and using it at the top, the Gradescope/Pensieve autograder will fail because it doesn't recognize the variable when it encounters it.

Another scenario where this can happen is if you defined a variable in an earlier cell but later changed its name or deleted it. While your current session's kernel will "remember" the variable, the Gradescope/Pensieve autograder will not, because it runs the notebook from top to bottom with no variables pre-stored. As a result, it will not recognize the variable if it has been deleted or renamed, and will throw an error.

The third scenario where this can happen is if you do not save your notebook before running the final export cell, which generates a zip file containing your notebook and the necessary files. If you run the export cell without saving, the autograder will not have access to the most recent version of your notebook, which can lead to unexpected errors. It will rely on the state of the notebook at the time of the last save, so any changes made afterward will not be reflected in the autograder's execution.

This is why we recommend first saving and then going into the top left menu and clicking `Kernel` -> `Restart` -> `Run All`. The autograder "forgets" all of the variables and runs the notebook from top-to-bottom like the Gradescope/Pensieve autograder does. This will highlight any issues.

Find the first cell that raises an error. Make sure that all of the variables used in that cell have been defined above that cell, and not below.

3.2.2 Why do I get a `NameError: name ___ is not defined` when I run a grader check?

This happens when you try to access a variable that has not been defined yet. Since the autograder runs all the cells in-order, if you happened to define a variable in a cell further down and accessed it before that cell, the autograder will likely throw this error. Another reason this could occur is because the notebook was not saved before the autograder tests are run. When in doubt, it is good practice to restart your kernel, run all the cells again, and save the notebook before running the cell that causes this error.

3.2.3 Why do I see multiple `NameError: name ___ is not defined` errors for all questions after a certain point on Gradescope/Pensieve?

This can happen if you import external packages when you are not instructed to do so. The base image on Gradescope/Pensieve includes only a specific list of pre-installed packages, defined by the course staff in the autograder. If you import a package that is not on this list, it will not be available to the autograder, even if you download the package correctly on your local Jupyter notebook environment. As a result, it will first encounter a `ModuleNotFoundError` when trying to import the package (though this error may not be shown to you), and then a `NameError` for every question afterward. Although this behavior is unintuitive, it's a limitation of the autograder—only course staff can view certain output messages from your notebook to prevent cheating. If you encounter this issue, review your notebook's imports and ensure you are not using any packages not included in the autograder environment.

3.2.4 My autograder keeps running/timed out

If your Gradescope submission page has been stuck running on this page for a while:

or if it times out:

it means that the Gradescope autograder failed to execute in the expected amount of time. TThis could be due to an inefficiency in your code or an issue on Gradescope's end, so we recommend resubmitting and allowing the autograder to rerun. If the issue persists after a few attempts, you may need to investigate your code for inefficiencies.

For example, if you rerun all cells in your Jupyter notebook and notice that some cells take significantly longer to run than others, you might need to optimize those cells. Keep in mind that the time it takes to run all tests in your Jupyter notebook is not equivalent to the time it takes on Gradescope/Pensieve. However, it is proportional—if your notebook runs slowly on Datahub, it will likely run even more slowly on Gradescope/Pensieve, as the difference in runtime is amplified.

It is your responsibility to ensure that the autograder runs properly, and if it still fails, you should follow up by making a private Ed post.

4 Pandas

4.1 Understanding `pandas` errors

`pandas` errors can look red, scary, and very long. Fortunately, we don't need to understand the entire thing! The most important parts of an error message are at the **top**, which tells you which line of code is causing the issue, and at the **bottom**, which tells you exactly what the error message is.

This note is (mostly) structured around the error messages that show up at the bottom.

4.2 My code is taking a really long time to run

It is normal for a cell to take a few seconds – sometimes a few minutes – to run. If it's is taking too long, however, you have several options:

1. Try restarting the kernel. Sometimes, Datahub glitches or lags, causing the code to run slower than expected. Restarting the kernel should fix this problem, but if the cell is still taking a while to run, it is likely a problem with your code.
2. Scrutinize your code. Am I using too many for loops? Is there a repeated operation that I can substitute with a `pandas` function?

4.3 Why is it generally better avoid using loops or list comprehensions when possible?

In one word: performance. `NumPy` and `pandas` functions are optimized to handle large amounts of data in an efficient manner. Even for simple operations, like the elementwise addition of two arrays, `NumPy` arrays are much faster and scale better (feel free to experiment with this yourself using `%%time`). This is why we encourage you to **vectorize** your code (ie. using `NumPy` arrays, `Series`, or `DataFrames` instead of Python lists) and use in-built `NumPy`/`pandas` functions wherever possible.

4.4 KeyErrors

4.4.1 `KeyError: 'column_name'`

This error usually happens when we have a `DataFame` called `df`, and we're trying to do an operation on a column `'column_name'` that does not exist. If you encounter this error, double check that you're operating on the right column. It might be a good idea to display `df` and see what it looks like. You could also call `df.columns` to list all the columns in `df`.

4.5 TypeErrors

4.5.1 `TypeError: '___' object is not callable`

This often happens when you use a default keyword (like `str`, `list`, `range`, `bool`, `sum`, or `max`) as a variable name, for instance:

```
sum = 1 + 2 + 3
```

These errors can be tricky because they don't error on their own but cause problems when we try to use the name `sum` (for example) later on in the notebook.

To fix the issue, identify any such lines of code (Ctrl+F on "`sum =`" for example), change your variable names to be something more informative, and restart your notebook.

Python keywords like `str` and `list` appear in green text, so be on the lookout if any of your variable names appear in green!

4.5.2 `TypeError: could not convert string to a float`

This error often occurs when we try to do math operations (ie. `sum`, `average`, `min`, `max`) on a `DataFrame` column or `Series` that contains strings instead of numbers (note that we can do math operations with booleans; Python treats `True` as 1 and `False` as 0).

Double check that the column you're interested in is a numerical type (`int`, `float`, or `double`). If it looks like a number, but you're still getting this error, you can use `.astype(...)` (documentation) to change the datatype of a `DataFrame` or `Series`.

4.5.3 `TypeError: Could not convert <string> to numeric`

Related to the above (but distinct), you may run into this error when performing a numeric aggregation function (like `mean` or `sum` functions that take integer arguments) after doing a `groupby` operation on a `DataFrame` with non-numeric columns.

Working with the `elections` dataset for example,

```
elections.groupby('Year').agg('mean')
```

would error because `pandas` cannot compute the mean of the names of presidents. There are three ways to get around this:

1. Select only the numeric columns you are interested in before applying the aggregation function. In the above case, both `elections.groupby('Year')['Popular Vote']` or `elections['Popular vote'].groupby('Year')` would work.
2. Setting the `numeric_only` argument to `True` in the `.agg` call, thereby applying the aggregation function only to numeric columns. For example, `elections.groupby('Year').agg('mean', numeric_only=True)`.
3. Passing in a dictionary to `.agg` where you specify the column you are applying a particular aggregation function to. Continuing the same example, this looks like `elections.groupby('Year').agg({'Popular vote' : 'mean')`.

4.5.4 `TypeError: 'NoneType' object is not subscriptable` / `AttributeError: 'NoneType' object has no attribute 'shape'`

This usually occurs as you assign a `None` value to a variable, then try to either index into or access some attribute of that variable. For Python functions like `append` and `extend`, you do not need to do any variable assignment because they mutate the variable directly and return `None`. Assigning `None` tends to happen as a result of code like:

```
some_list = some_list.append(element)
```

In contrast, an operation like `np.append` does not mutate the variable in place and, instead, returns a copy. In these cases, (re)assignment is necessary:

```
some_array = np.append(some_array, element)
```

4.5.5 `TypeError: 'int'/'float' object is not subscriptable`

This occurs when you try and index into an integer or other numeric `Python` data type. It can be confusing to debug amidst a muddle of code, but you can use the error message to identify which variable is causing this error. Using `type(var_name)` to check the data type of the variable in question can be a good starting point.

4.6 IndexErrors

4.6.1 `IndexError: invalid index to scalar variable.`

This error is similar to the last `TypeError` in the previous section. However, it is slightly different in that scalar variables come up in the context of `NumPy` data types which have slightly different attributes.

For a concrete example, if you defined

```
numpy_arr = np.array([1])
```

and indexed into it twice (`numpy_arr[0][0]`), you would get the above error. Unlike a Python integer whose type is `int`, `type(numpy_arr[0])` returns the `NumPy` version of an integer, `numpy.int64`. Additionally, you can check the data type by accessing the `.dtype` attribute of `NumPy` array (`numpy_arr.dtype`) or scalar variable (`numpy_arr[0].dtype`).

4.6.2 `IndexError: index _ is out of bounds for axis _ with size _`

This error usually happens when you try to index a value that's greater than the size of the array/list/`DataFrame`/`Series`. For example,

```
some_list = [2, 4, 6, 8]
```

`some_list` has a length of 4. Trying `some_list[6]` will error because index 6 is greater than the length of the array. Note that `some_list[4]` will also cause an `IndexError` because Python and `pandas` uses zero indexing, which means that the first element has index 0, the second element has index 1, etc.; `some_list[4]` would grab the fifth element, which is impossible when the list only has 4 elements.

4.7 ValueErrors

4.7.1 `ValueError: Truth value of a Series is ambiguous`

This error could occur when you apply Python logical operators (`or`, `and`, `not`), which only operate on a single boolean values, to `NumPy` arrays or `Series` objects, which can contain multiple values. The fix is to use bitwise operators `|`, `&`, `~` , respectively, to allow for element-wise comparisons between values in arrays or `Series`.

Alternatively, these errors could emerge due to overwriting Python keywords like `bool` and `sum` that may be used in the autograder tests, similar to what's described here. You should follow a similar procedure of identifying the line of code erroring, checking if you've overwritten any Python keywords using Ctrl+F, and renaming those variables to something more informative before restarting your kernel and running the erroring tests again.

4.7.2 `ValueError: Can only compare identically-labeled Series objects`

As the message would suggest, this error occurs when comparing two `Series` objects that have different lengths. You can double check the lengths of the `Series` using `len(series_name)` or `series_name.size`.

4.7.3 `ValueError: -1 is not in range` / `KeyError: -1`

This error occurs when you try and index into a `Series` or `DataFrame` as you would a Python list. Unlike a list where passing an index of -1 gives the last element, `pandas` interprets `df[-1]` as an attempt to find the row corresponding to index -1 (that is, `df.loc[-1]`). If your intention is to pick out the last row in `df`, consider using integer-position based indexing by doing `df.iloc[-1]`. In general, to avoid ambiguity in these cases, it is also good practice to write out both the row and column indices you want with `df.iloc[-1, :]`.

5 RegEx

RegEx syntax can be incredibly confusing, so we highly encourage using sources like the Data 100 Exam reference sheet (you can find this under the "Exam Resources" section on our Resources page) or websites like regex101.com to help build your understanding.

5.1 How to Interpret regex101

Regex101 is a great tool that helps you visually interact with RegEx patterns. Let's take a look at its components with a simple example.

5.1.1 Example 1: Basic

0. **Flavor**: Regular expressions work slightly differently depending on the programming language you use. In Data 100, we only use the `Python` flavor. By default, regex101 opens on the PCRE2 flavor, so make sure to change to `Python` before experimenting.
1. **Regular Expression**: This is where the RegEx expression goes. For this example, our pattern is `Data 100`. In `Python`, we denote it as a string `r"Data 100"` with the prefix `r` to indicate that this is a RegEx expression, not a normal `Python` string. In regex101, because we changed to the `Python` flavor, we don't need to type out the `r"` at the start or the `"` at the end, as that's already set up for us.
2. **Explanation**: This portion of the website explains each component of the pattern above. Since it does not contain any special characters, `Data 100` will match any portion of a string containing `Data 100`.
3. **Test String**: This is where you can try out different inputs and see if they match the RegEx pattern. Of the 4 example sentences, we see that only the first sentence contains characters that match the pattern, highlighted in blue. (Note that while sentence 3 does contain `data 100`, RegEx is sensitive to capitalization. `d` and `D` are different characters)
4. **Match Information**: Each match between the RegEx expression and test strings is shown here.

5.1.2 Example 2: Greedy

For this example, let's replace the `100` in our original expression with `\d+` so that our pattern is `Data \d+`

`\d` and `+` are both special operators, and the explanation on the top right (boxed in red) tells us what they do:

- `\d` matches digits, or any number between 0 and 9. It's equivalent to `[0-9]`.
- `+` matches the previous token ≥ 1 times. It is a *greedy operation*, meaning it will match as many characters as possible.

Altogether, the expression `\d+` will match any digit one or more times. Look at each match under "Match Information". Can you see why they align with `Data \d+`?

5.1.3 Example 3: Capturing Groups

Let's say we're given a body of text with dates formatted as `DD/Month/YYYY` (ie. 04/Jan/2014), and we're interested in extracting the dates. An expression like `r"\d+\/\w+\/\d+"` would match any string with the `DD/Month/YYYY` format:

- the first `\d+` matches `DD` patterns (ie. `04`)
- `\/` matches the `/` separator. Since `/` is a special operator in RegEx, we need to escape it with `\` to get the literal character.
- `\w+` in the middle matches `Month` patterns we're interested in (ie. `Jan`, `January`)
- lastly, `\d+` matches `YYYY` patterns (ie. `2014`)

That's great! This pattern will match the entirety of `DD/Month/YYYY`, but what if we want to access `DD` individually? What about `YYYY`? This is where **capturing groups** comes in handy. Capturing groups are RegEx expressions surrounded by parenthesis `()` that are used to remember the text they match so that it can be referenced later. Putting capturing groups around `\d+` and `\w+` to get `r"(\d+)\/(\w+)\/(\d+)"`gives us the following:

- The "Explanation" section now shows an explanation for each of the 3 capturing groups.
- In our test strings, the portion matching the RegEx expression is highlighted in blue per usual. Additionally, each capturing group is highlighted with a particular color: green, orange, and purple.
- These colored highlights correspond to their match/group under "Match Information". "Match #" (light blue) shows the entire portion that matches the expression while "Group #" shows the match per group.

5.1.3.1 How do I access captured groups?

To access each group, we use the following syntax:

```
target_string = "Today's date is 01/March/2024."
result = re.search(r"(\d+)\/(\w+)\/(\d+)", target_string)

result # re.Match object
result.groups() # all captured groups: ('01', 'March', '2024')
result.group(0) # '01/March/2024', the full match
result.group(1) # '01', the first captured group
result.group(2) # 'March', the second captured group
result.group(3) # '2024', the third captured group
```

5.2 RegEx Misconceptions & General Errors

5.2.1 I'm certain my RegEx pattern in `.str.replace` is correct, but I'm not passing the grader check.

Here's the skeleton from the exam reference sheet:

`s.str.replace(pat, repl, regex=False)`

Notice how the `regex=` argument has a default value of `False`, causing `pandas` to treat `pat` like a normal `Python` string. Make sure to set `regex=True` if you're using RegEx!

5.2.2 My RegEx pattern matches the test cases on regex101, but is not working in `pandas` with `.str.findall` / `.str.extractall` / `.str.extract`.

The most likely reason for this is forgetting to include the `r` before the string with your regular expression. Without including the `r` in `.str.findall(r".*")`, `pandas` will not interpret your pattern as a regex expression and will only match it literally.

5.2.3 `Value Error: pattern contains no capture groups`

These errors usually occur when using `s.str.extract` or `s.str.extractall`. Read more about it in the RegEx course notes. This error means that your RegEx pattern does not match anything in the given `Series` of strings. To debug this, try putting your pattern into regex101.com and use example strings from the Series as test cases.

5.2.4 When do I need to escape characters?

The special characters in RegEx are: `. ^ $ * + ? ] [ \ | ( ) { } }`

If you want to match exactly those characters in a RegEx expression, you need to "escape" them by preceding them with a backslash `\`. However, the rules around this can change in the context of character classes.

For example, the pattern `r"[.]"` matches `'.'`, the literal period. In this context, it is not treated as a special character. The hyphen, while not included in the list of special characters, also changes its behavior depending on its position in a character class. It can be used to specify a range of characters (e.g. `r"[0-9]"`) based on their Unicode values, or match a literal `'-'` if it does not have two adjacent characters (e.g. `r"[-09]"` matches `-`, `0`, `9`). To be on the safer side, you could escape `-` like in `r"[0\-9]"` to achieve the same result.

Finally, it's generally good practice to escape both single and double quotes for greater readability. Technically, patterns like `r"'(.\*)'"` and `r'"(.\*)"'` do work as you'd expect, but you can already see how confusing it is to decipher what's going on. Escaping the quotes inside the pattern does not affect what matches you get, but makes it easier to figure out what the intended match was.

5.2.5 The three uses of ^

The `^` character can be tricky to wrap your head around given how its function changes depending on the context:

1. If used at the start of a pattern, like in `r"^\w"`, it means that a lowercase letter must begin the string in order for a match to occur.
2. If included at the start of a character class, like in `r"[^abc]"`, it negates all the characters in that class and will match with any other character – in the above example, any character that is not `a`, `b`, `c`.
3. Finally, if escaped as in `r"\^"` it is treated as a literal and will match any instance of `^`.

5.2.6 What's the difference between all the `re` functions?

The exam reference sheets give a few `re` functions, but how can you determine which one to use?

`re.match` and `re.search` only return *one* instance of a match between string and pattern (or None if there's no match) - `re.match` only considers characters at the beginning of a string - `re.search` considers characters anywhere in the string - For example: "' pattern = r"Data 100" example1 = "Data 100 is the best!" example2 = "I love Data 100!"

re.match(pattern, example1).group(0) # matches “Data 100” re.match(pattern, example2) # does not match “Data 100” because it’s not at the beginning of a string; returns None

re.search(pattern, example1).group(0) # matches “Data 100” re.search(pattern, example2).group(0) # matches “Data 100” “‘

If, instead, you’re interested in finding *all* matches between the given string and pattern, `re.findall` will find them all, returning the matches in a list.

```
re.findall(r'\d+', 'Data 100, Data 8, Data 101')
# returns a list of strings: ['100', '8', '101']

re.findall(r'\d+', 'Data science is great')
# no matches found, returns empty list: []
```

`re.sub` will find them all *and replace it* with a string of your choice.

```
re.sub(r'\d+', 'panda', 'Data 100, Data 8, Data 101')
# returns 'Data panda, Data panda, Data panda'

re.sub(r'\d+', 'panda', 'Data science is great')
# no matches found, returns the original string "Data science is great"
```

5.2.7 What’s the difference between `re` functions and `pd.Series.str` functions?

Generally, all the `pd.Series.str` functions are used when you want to apply a Python or RegEx string function to a *Series of strings*. In contrast, `re` functions are applied to string objects. The reference sheet gives a great overview of the different use cases of each of the `pd.Series.str` functions.

6 Visualizations

Visualizations are how data scientists use to communicate their insights to the world. In the process of making a good visualization, be it while adding a legend or setting the x-axis label, we may run into some errors. Let's take a look at how to resolve them below!

6.0.1 My legend's labels don't match up / my legend isn't displaying properly

If you simply add `plt.legend()` after your plotting line of code, you should see a legend; seaborn will sometimes automatically populate the legend. However, if you're plotting multiple lines or sets of points on a single plot, the labels in the legend may not correctly line up with what's shown.

Make sure to pass in the `label` argument into the `sns` plotting function with the label you want associated with that individual plot. For example,

```
sns.histplot(means_arr, label = 'simulated values') # informative label name
plt.title('Simulated values')
plt.plot(original, 10, 'bo', label = 'original test statistic') # informative label name
plt.legend(loc = 'upper left') # can specify location of legend
```

6.1 The y-axis of my `histplot` shows the count, not the density

Look into the `sns.histplot` documentation and see what arguments the `stat` parameter takes in. By default, `stat=count`, and the number of elements in each histogram bin is the y axis. But if you wanted to normalize the distribution such that the total area is 1, consider passing `stat=density` into the plot function.

6.2 I'm having trouble labeling the axes/title of my graph

To label the axes and title of a graph, we use the following syntax:

```
plt.xlabel("x axis name")
plt.ylabel("y axis name")
plt.title("graph title")
```

Where `plt.xlabel`, `plt.ylabel`, and `plt.title` are matplotlib functions that we call.

However, we often see students use the following incorrect syntax to try and label their plot:

```
plt.xlabel = "x name"
plt.ylabel = "y name"
plt.title = "graph title"
```

Now, instead of `plt.xlabel`, `plt.ylabel`, and `plt.title` being functions, they are strings. Trying to call one of the labelling function using the correct syntax in the next few cells (ie.`plt.xlabel("x name")`) will result in a `TypeError: str object is not callable`. If this happens to you, comb through your notebook and look for places when you used the incorrect syntax. After fixing it, restart your kernel and rerun your cells.

6.3 My `sns.lineplot` has an unwanted shaded region around the solid lines.

Note: the following examples are taken from `sns.lineplot`'s documentation.

`sns.lineplot` gives us a clean line when each `x` value has one `y` value. For example, the table

Year	May
1948	120
1949	122
1950	123
⋮	⋮

will give us a clean plot because each year corresponds to a single "Number of Flights in May" value.

When each x value has multiple y values, `sns.lineplot` will automatically plot a shaded region around the solid line, where the solid line is the mean of the y values for that x value (think of a groupby on x aggregated by `.mean()` on y) and the shaded region is the 95% confidence interval (read more about confidence intervals in the Data 8 textbook). For example, the table

Year	May
1948	115
1948	120
1948	125
1949	118
1949	122
1949	126
⋮	⋮

will plot a lineplot with a shaded region.

If you do not want the shaded region, aggregate the data such that there is only one y-value for a given x-value; then, make the plot.

7 Project A1 Common Questions

7.1 Question 6

7.1.1 `TypeError: could not convert string to float: 'SF'`

Type errors like these usually stem from applying a numeric aggregation function to a non-numeric column as described in the `pandas` section of the debugging guide.

Aggregation functions like `np.median` and `np.mean` are only well-defined for columns with numeric types like `int` and `float`. Your code is likely trying to aggregate across all columns in `training_data`, including those of type `str`. Instead of aggregating across the entire `DataFrame`, try just selecting the relevant columns.

7.1.2 `TypeError: unhashable type: 'Series'`

This error can occur if you try and use Python's `in` to check whether values in a `Series` are contained in a list. If you're trying to perform boolean filtering in this manner, you should look into the `.isin` (documentation) function as introduced in HW 2.

7.2 Question 7

7.2.1 I'm not sure how to use `sklearn` to do One Hot Encoding

A good starting point is to revisit the One Hot Encoding question in Lab 7. It's recommended you look through this portion of the walkthrough, so you have a good understanding of how to use the `OneHotEncoder` object. Pay attention to what each variable represents and the expected outputs of the functions used. Can you map the logic from the lab to this project? A nice way to start is to make a new cell and experiment with examples from the documentation.

7.2.2 My OHE columns contain a lot of `NaN` values

This may happen if you try and merge the OHE columns with the `training_data` table without making sure both `DataFrame` have the *same index values*. Look into the `pd.merge` documentation for ways to resolve this.

8 Project A2 Common Questions

8.1 Questions 5d and 5f

8.1.1 General Debugging Tips

Question 5 is a challenging question that mirrors a lot of data science work in the real world: cleaning, exploring, and transforming data; fitting a model, working with a pre-defined pipeline and evaluating your model's performance. Here are some general debugging tips to make the process easier:

- Separate small tasks into helper functions, especially if you will execute them multiple times. For example, a helper function that one-hot encodes a categorical variable may be helpful as you could perform it on multiple such columns. If you're parsing a column with RegEx, it also might be a good idea to separate it to a helper function. This allows you to verify that you're not making errors in these small tasks and prevents unknown bugs from appearing.
- Feel free to make new cells to play with the data! As long as you delete them afterward, it will not affect the autograder.
- The `feature_engine_final` function looks daunting at first, but start small. First, try and implement a model with a single feature to get familiar with how the pipeline works, then slowly experiment with adding one feature at a time and see how that affects your training RMSE.

8.1.2 My training RMSE is low, but my validation/test RMSE is high

Your model is likely overfitting to the training data and does not generalize to the test set. Recall the bias-variance tradeoff discussed in lecture. As you add more features and make your model more complex, it is expected that your training error will decrease. Your validation and test error may also decrease initially, but if your model is too complex, you end up with high validation and test RMSE.

To decrease model complexity, consider visualizing the relationship between the features you've chosen with the `(Log) Sale Price` and removing features that are not highly correlated. Removing outliers can also help your model generalize better and prevent it from fitting to noise in the data. Methods like cross-validation allow you to get a better sense of where you

lie along the validation error curve. Feel free to take a look at the code used in Lecture 16 if you're confused on how to implement cross-validation.

8.1.3 `ValueError: Per-column arrays must each be 1-dimensional`

If you're passing the tests for question 5d but getting this error in question 5f, then your `Y` variable is likely a `DataFrame`, not a `Series`. `sklearn` models like `LinearRegression` expect `X` to be a 2D datatype (ie. `DataFrame`, 2D `NumPy` array) and `Y` to be a 1D datatype (ie. `Series`, 1D `NumPy` array).

8.1.4 `KeyError: 'Sale Price'/KeyError: 'Log Sale Price'`

`KeyError`s are raised when a column name does not exist in your `DataFrame`. You could be getting this error because:

- The test set does not contain a `"(Log) Sale Price"` as that's what we're trying to predict. Make sure you only reference the `"(Log) Sale Price"` column when working with training data (`is_test_set=False`).
- You dropped the `"Sale Price"` column twice in your preprocessing code.

8.1.5 `Value Error: could not convert string to float`

This error occurs if your final design matric contains non-numeric columns. For example, if you simply run `X = data.drop(columns = ["Log Sale Price", "Sale Price"])`, all the non-numeric columns of `data` are still included in `X` and you will see this error message. The `fit` function of a `lm.LinearRegression` object can take a `pandas DataFrame` as the `X` argument, but requires that the `DataFrame` is only composed of numeric values.

8.1.6 `ValueError: Input X contains infinity or a value too large for dtype('float64')`

The reason why your `X` data contains infinity is likely because you are taking the logarithm of 0 somewhere in your code. To prevent this, try:

- Adding a small number to the features that you want to perform the log transformation on so that all values are positive and greater than 0. **Note that whatever value you add to your train data should also be added to your test data.**
- Removing zeroes before taking the logarithm. Note that this is only possible on the training data as you cannot drop rows from the test set.

8.1.7 `ValueError: Input X contains NaN`

The reason why your design matrix `X` contains `NaN` values is likely because you take the log of a negative number somewhere in your code. To prevent this, try:

- Shifting the range of values for features that you want to perform the logging operation on to positive values greater than 0. **Note that whatever value you add to your train data should also be added to your test data.**
- Removing negative values before taking the log. Note that this is only possible on the training data as you cannot drop rows from the test set.

8.1.8 `ValueError: The feature names should match those that were passed during fit`

This error is followed by one or both of the following:

```
Feature names unseen at fit time:
- FEATURE NAME 1
- FEATURE NAME 2
  ...

Feature names seen at fit time, yet now missing
- FEATURE NAME 1
- FEATURE NAME 2
  ...
```

This error occurs if the columns/features you're passing in for the test dataset aren't the same as the features you used to train the model. `sklearn`'s models expect the testing data's column names to match the training data's. The features listed under `Feature names unseen at fit time` are columns that were present in the *training* data but not the *testing* data, and features listed under `Feature names seen at fit time, yet now missing` were present in the *testing* data but not the *training* data.

Potential causes for this error:

- Your preprocessing for `X` is different for training and testing. Double-check your code in `feature_engine_final`! Besides removing any references to `'Sale Price'` and code that would remove rows from the test set, your preprocessing should be the same.
- Some one-hot-encoded categories are present in training but not in testing (or vice versa). For example, let's say that the feature `"Data100"` has categories "A", "B", "C", and "D". If "A", "B", and "C" are present in the training data, but "B", "C", and "D" are present in the testing data, you will get this error:

```
The feature names should match those that were passed during fit. Feature names unseen a
- Data100_D
  ...

Feature names seen at fit time, yet now missing
- Data100_A
```

8.1.9 `ValueError: operands could not be broadcast together with shapes ...`

This error occurs when you attempt to perform an operation on two NumPy arrays with mismatched dimensions. For example, `np.ones(100000) - np.ones(1000000)` is not defined since you cannot perform elementwise addition on arrays with different lengths. Use the error traceback to identify which line is erroring, and print out the shape of the arrays on the line before using `.shape`.

8.1.10 `TypeError: NoneType is not subscriptable`

This error occurs when a `NoneType` variable is being accessed like a class, for example `None.some_function()`. It may be difficult to identify where the `NoneType` is coming from, but here are some possible causes:

- Check that your helper functions always end with a `return` statement and that the result is expected!
- `pandas`' `inplace=` argument allows us to simplify code; instead of reassigning `df = df.an_operation(inplace=False)`, you can choose to shorten the operation as `df.an_operation(inplace=True)`. Note that any `inplace=True` argument modifies the `DataFrame` and *returns nothing* (read more about it in this stack overflow post). Both `df = df.an_operation(inplace=True)` and `df.an_operation(inplace=True).another_operation()` will result in this `TypeError`.

Check the return type of all functions you end up using. For example, `np.append(arr_1, arr_2)` returns a `NumPy` array. In contrast, `Python`'s `.append` function mutates the iterable and returns `None`. If you are unsure of what data type is being returned, looking up the documentation of the function , adding print statements, using `type(some_function(input)` are all useful ways to debug your code.

8.2 Question 6

8.2.1 I'm getting negative values for the `prop_overest` plot

Note that in the function body, the skeleton code includes:

```
# DO NOT MODIFY THESE TWO LINES
    if subset_df.shape[0] == 0:
        return -1
```

The above two lines of code are included to avoid dividing by 0 when computing `prop_overest` in the case that `subset_df` does not have any rows. When interpreting the plots, you can disregard the negative portions as you know they are invalid values corresponding to this edge case. You can verify this by observing that your `RMSE` plot does not display the corresponding intervals.

8.3 Gradescope/Pensieve

8.3.1 I don't have many Gradescope/Pensieve submissions left

If you're almost out of Gradescope submissions, try using k-fold cross-validation to check the accuracy of your model. Results from cross-validation will be closer to the test set accuracy than results from the training data. Feel free to take a look at the code used in Lecture 16 if you're confused on how to implement cross-validation.

8.3.2 "Wrong number of lines (** instead of **)"

This occurs when you remove outliers when preprocessing the testing data. *Please do not remove any outliers from your test set.* You may only remove outliers in training data.

8.3.3 My code runs locally, but it fails for every question after a certain point

See Why do I see multiple NameError: name `___` is not defined errors for all questions after a certain point on Gradescope/Pensieve?

9 SQL Common Errors

9.1 Common Misconceptions

9.1.1 How to read SQL Errors

Because Jupyter notebooks run Python natively, we need to import extra modules like `sql`, `sqlalchemy`, and `duckdb` in order to run SQL queries within our notebook. This results in slightly more complex error messages than normal Python. Refer to the image below if you need help identifying the most important parts (the red and blue text).

Sometimes, SQL errors may show up as a long string without any red highlight, for example:

```
(duckdb.duckdb.BinderException) Binder Error: Referenced column "some_column"
not found in FROM clause! LINE 1: SELECT some_column                    ^ ...
```

Note that this is the same error as the one above – it's just not formatted nicely. Try your best to parse it; you can even copy the error into a new cell and manually indent sentences if it helps with readability!

9.1.2 Comments in SQL

Unlike `Python`, which uses `#...` or triple quotes `"""..."""` to indicate a comment, SQL comments take the following forms:

```
-- This is a single-line or in-line comment

/*
  This is a
  multi-line
  comment
*/
```

Because you are still coding within a Python notebook, the colors that Jupyter chooses for your SQL queries may not match up to what you expect; using `#...` will be green text like Python comments, but running the code will give you:

`Parser Error: syntax error at or near...`

Using `--` or `/* ... */` may not color-code the text correctly, but SQL will still read it as a comment!

9.1.3 Filtering with multiple conditions

When you are using more than one condition in filtering clauses like `WHERE` or `HAVING`, it's important consider that the ordering of `OR`s and `AND`s will affect your SQL output.

In `SQL`, the operator precedence typically evaluates `AND` operations before `OR` operations, and so incorrectly ordered conditions can lead to unexpected results. To ensure your conditions are evaluated in the intended order, use parentheses to group them explicitly.

For example, consider the following query condition:

```
WHERE A > 500 AND B > 200 OR C < 50
```

By default, is will be evaluated as

```
WHERE (A > 500 AND B > 200) OR C < 50
```

because `AND` has a higher precedence than `OR`. To evaluate it differently, add parentheses to specify the order of operations:

```
WHERE A > 500 AND (B > 200 OR C < 50)
```

9.1.4 Truncated Display Limit

Our Jupyter Notebooks are set to only display the first 10 rows of a table. Otherwise, your notebook would likely crash with the weight of displaying >1000 rows. You'll likely see a

`Truncated to displaylimit of 10`

printed right below your SQL query if your table contains more than 10 rows.

9.2 Common Errors

9.2.1 `NameError: name '[res_q#]' is not defined`

This occurs when you have an error in your SQL code, causing the cell after to fail. Because we're working with Jupyter Notebooks, some SQL errors are not highlighted in red like `Python` errors usually are. Check the output under your `%%sql` cell to see if it ran smoothly. Any text below

`Running query in 'duckdb:///data/{database_name}.db`

is an error message that you can read to debug your code. Note that this error could be formatted as one long string without any indents or newlines, so it may be harder than typical `Python` errors to parse.

9.2.2 `Referenced column "{some_string}" not found`

There are 2 potential causes for this error:

1. **The column you're looking for does not exist in the table.** Make sure that the column names are spelled and capatalized exactly as shown in the assignment. If you are trying to reference a column that is not in the current table, consider using `JOIN` to be able to access these columns. You can reference the SQL II course notes for a refresher on how to use the `JOIN` clause.
2. **You used double quotes `""` around `some_string` instead of single quotes `''`.** Unlike `Python`, SQL differentiates double and single quotes: double quotes indicate something that exists within the database, like tables and column names; single quotes indicate strings.

9.2.3 `Cannot compare values of type __ and type __`

This error often occurs when a string-type column (ie. `CHAR` or `VARCHAR`) is compared with a numeric-type column (ie. `INT`, `BIGINT`, `FLOAT`, `REAL`). If you're interested in learning more about SQL datatypes, check out this link!

To debug, start by checking the type of the column(s) you're working with; you can either scroll up to the cell (or make a new cell) with the code:

```
%%sql
SELECT * FROM sqlite_master WHERE type='table';
```

which outputs the schema for each table. If we're working with the IMDB database, we get the following output:

We can see that the table `Title` contains the columns

- `tconst` of type `BIGINT`
- `titleType` of type `VARCHAR`
- `primaryTitle` of type `VARCHAR`
- and so on...

It's a good idea to double check the types of the columns involved by looking at the schema before performing numerical comparisons and when deciding whether to `CAST` a particular column.

9.2.4 My query runs but I'm not passing the tests

For a lot of questions in the homework, we limit our focus to **movie** titles. Double check that you are filtering your results appropriately. More generally, if you find your queries aren't erroring, but are failing the tests due to some logical error, a good starting point is always reading the question again and making a checklist of all the conditions you need to implement.