# Data 100 Debugging Guide

Yash Dave Lillian Weng

# Table of contents

# About

This text offers pointers for keyboard shortcuts or common mistakes that accompany the coursework in the Spring 2024 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 fill out the Data 100 Content Feedback Form (Spring 2024). Note that this link will only work if you have an @berkeley.edu email address. If you're not a student at Berkeley and would like to provide feedback, 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 loose 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. 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 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 `$$`. If you're short on time, your best bet is to take screenshots of your free response answers and submit them to Gradescope. If you have more time and would like the Datahub-generated PDF, please remove any special LaTeX characters from your free response answers.

## 2.9 I can't open Jupyter: `HTTP ERROR 431`

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

# 3 Autograder and Gradescope

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

When submitting to Gradescope, 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 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 autograder will fail because it doesn't recognize the variable when it encounters it.

This is why we recommend 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 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 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. This could be due to an inefficiency in your code or a problem on Gradescope's end, so we recommend resubmitting and letting the autograder rerun. **It is your responsibility to ensure that the autograder runs properly**, and, if it still fails, to 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 two 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)`.

### 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 $\geq 1$ 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 `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.

### 5.2.3 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.

### 5.2.4 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 `^`.

### 5.2.5 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"
```

### 5.2.6 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.