8.4.1

Introduction to Regular Expressions

Although you and Britta have transformed all the data, Britta wants to go over how to clean and transform data by using Python regular expressions.

A **regular expression**, also known as a **regex**, is a string of characters that defines a search pattern. Although the syntax might be new to you, the concept is one that you're already familiar with the noncoding world.

For example, "MM/DD/YYYY" is a string of characters that defines a pattern for entering dates. It's a regular expression that you can recognize, because it follows a well-defined pattern. Likewise, "(###) ###-####" is a pattern for entering United States phone numbers. Regular expressions just offer a more formal way of defining these kinds of patterns—so that our code can find them. We'll expand on the details of regular expressions in this lesson. For now, just remember that we use them to search for patterns in text.

We use regular expressions to test whether strings exist in a specific format, to test whether strings contain a substring in a specific format, to extract pertinent information from strings while discarding unnecessary information, and to perform complicated replacements of substrings.

Furthermore, we can use regular expressions in almost all general-purpose languages, like Python, JavaScript, Java, and C#. Sometimes, they offer the only viable solution to a problem.

In the previous lessons of this module, we performed some common and straightforward data cleaning tasks. Examples include slicing and splitting strings; merging DataFrames; and formatting, renaming, and dropping columns. We also performed more-challenging tasks, like converting the rows in the `contact_info_df` DataFrame to Python dictionaries.

As a data analyst or data engineer, you might come across data that's harder to clean and transform. And, the conventional methods that you've used until now can both introduce errors and take lots of time to use to fulfill the business requirements. Luckily, Python has a module—`re`, which stands for "regular expression operations"—that we can use for data cleaning by retrieving substrings from strings.

But before we take a deep dive into regular expressions, let's consider some examples.

Example: Converting Single Quotes to Double Quotes

Suppose that the `contact_info_df` DataFrame had each key and string value in single quotes instead of double quotes, as follows:

`data = "{'contact_id': 4661, 'name': 'Cecilia Velasco', 'email': 'cecilia.velasco@rodrigues.fr'}"`

Now, create a new Jupyter Notebook, and then run the following code:

```
# Import the json module.
import json
# Assign the string data to a variable.
data = "{'contact_id': 4661, 'name': 'Cecilia Velasco', 'email': 'cecilia

# Convert the string data to a dictionary.
converted_data = json.loads(data)
# Iterate through the dictionary (row) and get the values.
row_values = [v for k, v in converted_data.items()]
print(row_values)
```

Trying to convert the `data` string to a dictionary results in the following error:

`JSONDecodeError: Expecting property name enclosed in double quotes: line 1 column 2 (char 1)`

This means that we need to have double quotes around our keys.

An easy solution is to replace the single quotes with double quotes by using the following code:

```
data = data.replace("'", '"')
print(data)
```

Running the code puts double quotes instead of single ones around the keys and string values, as follows:

`{"contact_id": 4661, "name": "Cecilia Velasco", "email": "cecilia.velasco@rodrigues.fr"}`

Now, you can use the earlier code to print the values after the string data gets converted to a dictionary. Do this yourself in the following Skill Drill:

SKILL DRILL

Convert the following updated string data to a dictionary, and then print the value of each key:

`dict_data = {"contact_id": 4661, "name": "Cecilia Velasco", "email": "cecilia.velasco@rodrigues.fr"}`

Example: Finding Substrings Without Punctuation as a Guide

Let's consider another example. Suppose that we had the following string data, which contains no punctuation:

`string_data = "contact_id 4661 name Cecilia Velasco email cecilia.velasco@rodrigues.fr"`

We can use list slicing to get the desired values. But, what if we had 1000 similar rows, like the `contact_info_df` DataFrame does? List slicing would still work to extract the four-digit numbers. But, retrieving the names and email addresses would be a time-consuming endeavor, because those values have different lengths.

Instead, we can use regular expressions and the `findall` function to extract the strings that we need. The `findall` function is one of the most powerful functions that works with regular expressions. It searches for all the strings that match a specified pattern. Here's the syntax for using the `findall` function:

`re.findall(pattern, string)`

To find all the four-digit numbers in our string data, for example, we can use the following code:

```
# Import the regular expression module.
import re
# Assign the string data to a variable.
string_data = "contact_id 4661 name Cecilia Velasco email cecilia.velasco
# Extract the four digit number.
```

```
contact_id = re.findall(r'(\d{4})', string_data)
print(contact_id)
```

The output from running the preceding code is a list that holds the four-digit number, as follows:

```
[4661]
```

Let's go over what happens when we use `’(d{4})’` for the pattern.

- `’(d{4})’` — The opening and closing parentheses contain the pattern for the capture group.
- `’(\d{4})’` — The "`\d`" matches a numerical digit.
- `’(\d{4})’` — The "`{4}`" says to match a numerical digit exactly four times.

Regular expressions use the backslash (`\`), which Python also uses for special characters. So, we need to tell Python to treat our regular expressions as raw strings of text. To do so, we put the letter "r" (without the quotation marks) before the `’(d{4})’` pattern. The regular expression within parentheses is called a **capture group**, which will capture or extract the desired substring from the variable, `string_data`. This results in the `r'(\d{4})'` that we used. We need to do this every time that we create a regular expression string.

Example: Finding Substrings in Multiple Rows

What if we need to use a regular expression pattern to extract strings from multiple rows in a DataFrame? We can use the Pandas `str.extract` function.

Download the following CSV file, which contains the contact info without punctuation:

contacts_string_data.csv (https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/v2/module_8/contacts_string_data.csv)

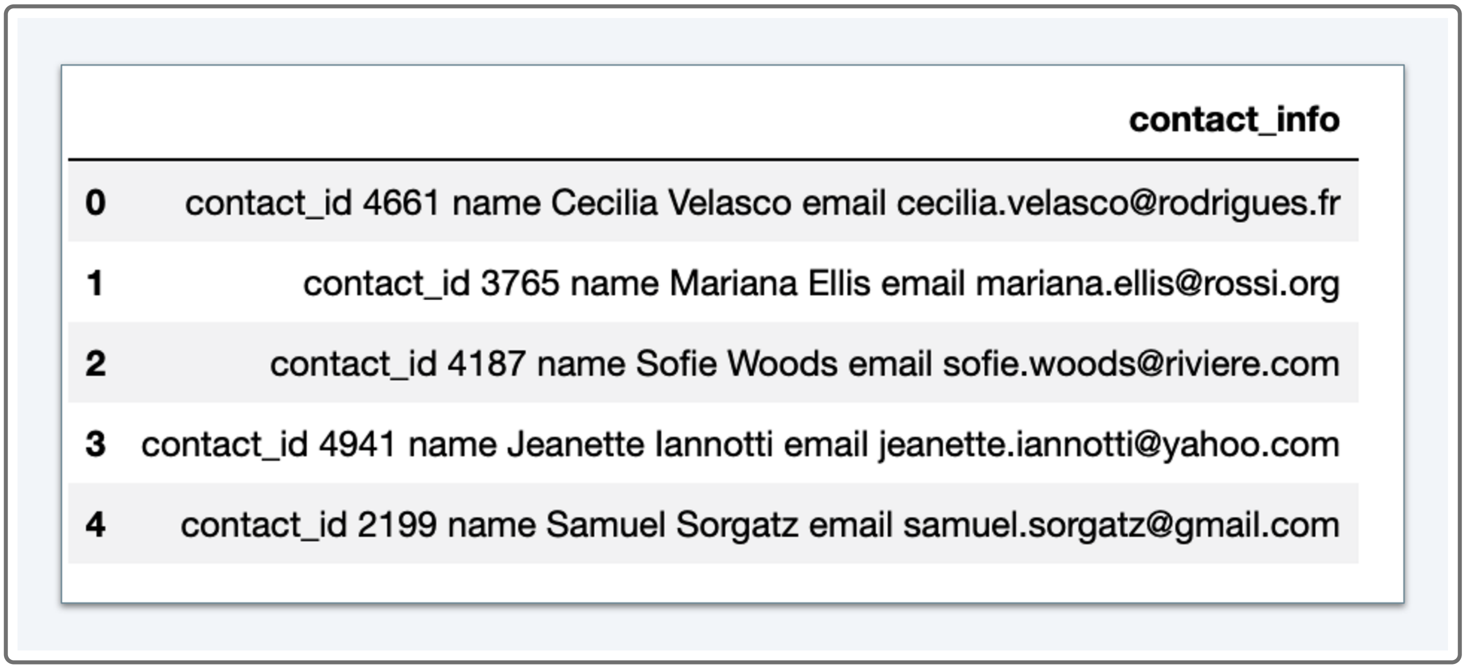
In a new Jupyter notebook file, copy the following code to read in the `contacts_string_data.csv` file into a DataFrame.

```
# Import the Pandas dependency.
import pandas as pd


# Read the contacts string data into a Pandas DataFrame
```

```
contacts_string_df = pd.read_csv("../Resources/contacts_string_data.csv")
contacts_string_df.head()
```

The following image shows the output from running the preceding code:



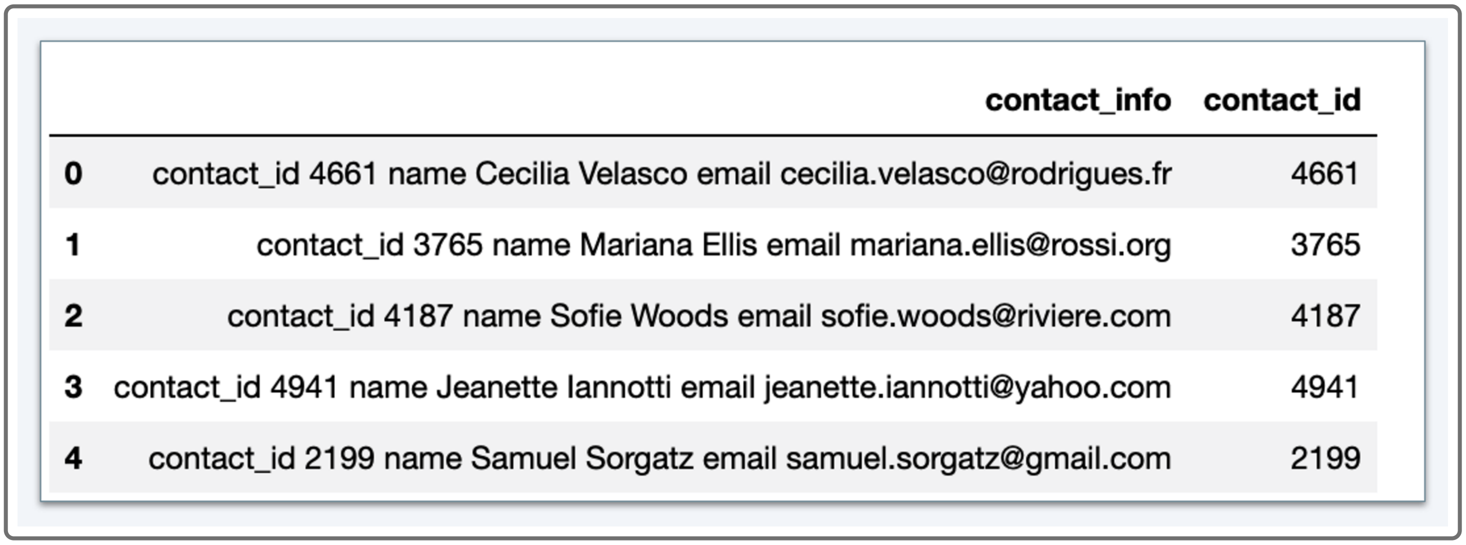
	contact_info
0	contact_id 4661 name Cecilia Velasco email cecilia.velasco@rodrigues.fr
1	contact_id 3765 name Mariana Ellis email mariana.ellis@rossi.org
2	contact_id 4187 name Sofie Woods email sofie.woods@riviere.com
3	contact_id 4941 name Jeanette Iannotti email jeanette.iannotti@yahoo.com
4	contact_id 2199 name Samuel Sorgatz email samuel.sorgatz@gmail.com

In the preceding image, notice that there is one column, "contact_info", and each row contains a Python string data instead of the Python dictionary with three keys—`contact_id`, `name`, and `email`—and values for each key

To extract the four-digit contact identification number of the "contact_id" we use the `str.extract` function, and we add the parameter, `'(\d{4})'` as the capture group. Run the following code to extract the four-digit contact identification number:

```
# Extract the four-digit contact ID number.
contact_string_df['contact_id'] = contact_string_df['contact_info'].str.e
```

Running the preceding code produces a second column in the `contact_df` DataFrame that contains the four-digit contact identification number, as the following image shows:



	contact_info	contact_id
0	contact_id 4661 name Cecilia Velasco email cecilia.velasco@rodrigues.fr	4661
1	contact_id 3765 name Mariana Ellis email mariana.ellis@rossi.org	3765
2	contact_id 4187 name Sofie Woods email sofie.woods@riviere.com	4187
3	contact_id 4941 name Jeanette Iannotti email jeanette.iannotti@yahoo.com	4941
4	contact_id 2199 name Samuel Sorgatz email samuel.sorgatz@gmail.com	2199

Now that we've received an introduction to regular expressions, including some examples, we'll next learn how to write them from scratch.

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.