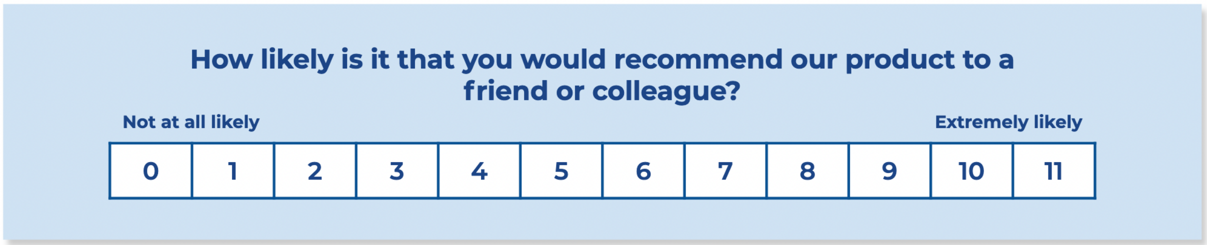
1. DRY: Don't repeat yourself


How likely is it that you would recommend our product to a friend or colleague?
Not at all likely
Extremely likely
0
1
2
3
4
5
6
7
8
9
10
11

Have you ever started your data analysis and ended up with repetitive code? Our colleague Brenda who works as a Product Analyst, has found herself in this situation and has asked us for some help. She's written a script to pull Net Promotor Score (NPS) data from various sources. NPS works by asking *How likely is it that you would recommend our product to a friend or colleague?* with a rating scale of 0 to 10. Brenda has set up this NPS survey in various ways, including emails and pop-ups on the mobile app and website. To compile the data from the different sources, she's written the following code:

```
# Read the NPS email responses into a DataFrame
email = pd.read_csv("datasets/2020Q4_nps_email.csv")
# Add a column to record the source
email['source'] = 'email'

# Repeat for NPS mobile and web responses
mobile = pd.read_csv("datasets/2020Q4_nps_mobile.csv")
mobile['source'] = 'mobile'
web = pd.read_csv("datasets/2020Q4_nps_web.csv")
web['source'] = 'web'

# Combine the DataFrames
q4_nps = pd.concat([email,mobile,web])
```

This results in the DataFrame `q4_nps` that looks like this:

	response_date	user_id	nps_rating	source
0	2020-10-29	36742	2	email
1	2020-11-26	31851	10	email
2	2020-10-27	44299	10	email
...	...	...	...	...

This code works, but it violates the Don't Repeat Yourself (DRY) programming principle. Brenda repeats the same code for email, mobile, and web, except with different variable names and file names. While it's often quicker to copy and paste, it makes it easier to introduce errors. For example, if you need to edit one of those lines, you have to do it in multiple places. Enter functions! Repeated code is a sign that we need functions. Let's write a function for Brenda.

In [1]:
```python
# Import pandas with the usual alias
import pandas as pd

# Write a function that matches the docstring
def convert_csv_to_df(csv_name, source_type):
    """ Converts an NPS CSV into a DataFrame with a column for the source.

    Args:
        csv_name (str): The name of the NPS CSV file.
        source_type (str): The source of the NPS responses.

    Returns:
        A DataFrame with the CSV data and a column, source.
    """
    df = pd.read_csv(csv_name)
    df['source'] = source_type
    return df

# Test the function on the mobile data
convert_csv_to_df("datasets/2020Q4_nps_mobile.csv", "mobile")
```

Out[1]:

	response_date	user_id	nps_rating	source
0	2020-12-29	14178	3	mobile
1	2020-10-29	33221	1	mobile
2	2020-11-01	21127	10	mobile
3	2020-12-07	42894	3	mobile
4	2020-11-26	30501	5	mobile
...	...	...	...	...
1796	2020-12-29	49529	3	mobile
1797	2020-12-24	23671	7	mobile
1798	2020-11-28	39954	7	mobile
1799	2020-12-19	21098	7	mobile
1800	2020-12-23	14919	7	mobile

1801 rows × 4 columns

2. Verifying the files with the "with" keyword

Excellent, we have a function that reads and cleans Brenda's CSVs precisely the way she needs! She can call this function in the future for as many different sources as she wants. Before we combine the NPS DataFrames, we want to add a function that verifies that the files inputted are valid. Each of these NPS dataset files should have three columns: `response_date`, `user_id`, `nps_rating`. Previously, Brenda would check this manually by opening each file.

Let's write a function that uses the **context manager** `with open()` so that we properly close the files we open, even if an exception is raised. If we don't use the `with` keyword with `open()`, we would need to call `close()` after we're done with the file. Even then, it's risky because an error might be raised before the `close()` functions are called.

The function will return `True` if the file contains the right columns. Otherwise, it will return `False`. To test the function, we'll use `datasets/corrupted.csv` to simulate a corrupted invalid NPS file.

In [2]:

```python
# Define a function check_csv which takes csv_name
def check_csv(csv_name):
    """ Checks if a CSV has three columns: response_date, user_id, nps_rating

    Args:
        csv_name (str): The name of the CSV file.

    Returns:
        Boolean: True if the CSV is valid, False otherwise
    """
    # Open csv_name as f using with open
    with open(csv_name) as f:
        first_line = f.readline()
        # Return true if the CSV has the three specified columns
        if first_line == "response_date,user_id,nps_rating\n":
            return True
        # Otherwise, return false
        else:
            return False

# Test the function on a corrupted NPS file
check_csv('datasets/corrupted.csv')
```

Out[2]:

```
False
```

3. Putting it together with nested functions

Alright, we now have one function that verifies that the CSVs are valid and another that converts them into the DataFrame format needed by Brenda. What's left? Looking at the script, this is the last line we haven't covered: `q4_nps = pd.concat([email,mobile,web])`. We could use this line of code, but we'll see more code repetition if we get CSVs from other sources or time periods.

To make sure our code is scalable, we're going to write a function called `combine_nps_csvs()` that takes in a dictionary. Python dictionaries have key:value pairs. In our case, the CSV's name and source type will be the key and value, respectively. That way, we can define a dictionary with as many NPS files as we have and run it through `combine_nps_csvs()`. For each file, we'll check that it's valid using `check_csv()`, and if it is, we'll use `convert_csv_to_df()` to convert it into a DataFrame. At the start of the function, we'll define an empty DataFrame called `combined` and everytime a CSV is succesfully converted, we'll concatenate it to `combined`.

In [3]:

```python
# Write a function combine_nps_csvs() with the arg csvs_dict
def combine_nps_csvs(csvs_dict):
    # Define combine as an empty DataFrame
    combined = pd.DataFrame()
    # Iterate over csvs_dict to get the name and source of the CSVs
    for name, source in csvs_dict.items():
        # Check if the csv is valid using check_csv()
        if check_csv(name):
            # Convert the CSV using convert_csv_to_df() and assign it to temp
            temp = convert_csv_to_df(name, source)
            # Concatenate combined and temp and assign it to combined
            combined = pd.concat([combined, temp])
        # If the file is not valid, print a message with the CSV's name
        else:
            print(name + " is not a valid file and will not be added.")
    # Return the combined DataFrame
    return combined

my_files = {
  "datasets/2020Q4_nps_email.csv": "email",
  "datasets/2020Q4_nps_mobile.csv": "mobile",
  "datasets/2020Q4_nps_web.csv": "web",
  "datasets/corrupted.csv": "social_media"
}

# Test the function on the my_files dictionary
combine_nps_csvs(my_files)
```

```
datasets/corrupted.csv is not a valid file and will not be added.
```

Out[3]:

	response_date	user_id	nps_rating	source
0	2020-11-06	11037	7	email
1	2020-12-24	34434	9	email
2	2020-12-03	49547	8	email
3	2020-10-04	13821	7	email
4	2020-10-23	29407	9	email
...	...	...	...	...
2285	2020-12-25	10656	8	web
2286	2020-11-07	32918	10	web
2287	2020-10-16	15667	10	web
2288	2020-11-20	47153	7	web
2289	2020-10-17	47071	5	web

6043 rows × 4 columns

4. Detractors, Passives, and Promoters

We've summarized our colleague's script into one function: `combine_nps_csvs()` ! Let's move on to analyzing the NPS data, such as actually calculating NPS. As a reminder, NPS works by asking *How likely is it that you would recommend our product to a friend or colleague?* with a rating scale of 0 to 10.

NPS ratings are categorized into three groups. Ratings between 0 to 6 are **detractors**, ratings between 7 to 8 are **passives**, and finally, ratings 9 to 10 are **promoters**. There's more to analyzing NPS, but remember, functions should be small in scope and should just "do one thing". So before we get ahead of ourselves, let's write a simple function that takes an NPS rating and categorizes it into the appropriate group.

In [4]:
```
def categorize_nps(x):
    """
    Takes a NPS rating and outputs whether it is a "promoter",
    "passive", "detractor", or "invalid" rating. "invalid" is
    returned when the rating is not between 0-10.

    Args:
        x: The NPS rating

    Returns:
        String: the NPS category or "invalid".
    """
    # Write the rest of the function to match the docstring
    if not isinstance(x, int):
        return "invalid"
    if x > 10 or x < 0:
        return "invalid"
    elif x > 8:
        return "promoter"
    elif x > 6:
        return "passive"
    else:
        return "detractor"

# Test the function
categorize_nps(8)
```

Out[4]:
```
'passive'
```

5. Applying our function to a DataFrame

So we have a function that takes a score and outputs which NPS response group it belongs to. It would be great to have this as a column in our NPS DataFrames, similar to the `source` column we added. Since we've modularized our code with functions, all we need to do is edit our `convert_cvs_to_df()` function and nest `categorize_nps()` into it. However, the way we'll nest `categorize_nps()` will be different than previous times. The `pandas` library has a handy function called `apply()`, which lets us apply a function to each column or row of a DataFrame.

In [5]:
```
def convert_csv_to_df(csv_name, source_type):
    """ Convert an NPS CSV into a DataFrame with columns for the source and NPS group.

    Args:
        csv_name (str): The name of the NPS CSV file.
        source_type (str): The source of the NPS responses.

    Returns:
         A DataFrame with the CSV data and columns: source and nps_group.
    """
    df = pd.read_csv(csv_name)
    df['source'] = source_type
    # Define a new column nps_group which applies categorize_nps to nps_rating
    df['nps_group'] = df['nps_rating'].apply(lambda x: categorize_nps(x))
    return df

# Test the updated function with mobile data
convert_csv_to_df("datasets/2020Q4_nps_mobile.csv", "mobile")
```

Out[5]:

	response_date	user_id	nps_rating	source	nps_group
0	2020-12-29	14178	3	mobile	detractor
1	2020-10-29	33221	1	mobile	detractor
2	2020-11-01	21127	10	mobile	promoter
3	2020-12-07	42894	3	mobile	detractor
4	2020-11-26	30501	5	mobile	detractor
...	...	...	...	...	...
1796	2020-12-29	49529	3	mobile	detractor
1797	2020-12-24	23671	7	mobile	passive
1798	2020-11-28	39954	7	mobile	passive
1799	2020-12-19	21098	7	mobile	passive
1800	2020-12-23	14919	7	mobile	passive

1801 rows × 5 columns

6. Calculating the Net Promoter Score

If we hadn't broken down our code into functions earlier, we would've had to edit our code in multiple places to add a `nps_group` column, increasing the chance of introducing errors. It also helps that our functions have one responsibility keeping our code flexible and easier to edit and debug.

Now we're in a good place to calculate the Net Promoter Score! This is calculated by subtracting the percentage of detractor ratings from the percentage of promoter ratings, in other words:

$ NPS = \frac{\text{# of Promoter Rating - # of Detractor Ratings}}{\text{Total # of Respondents}} * 100 $

We want to calculate the NPS across all sources, so we'll use `combine_nps_csvs()` from Task 3 to consolidate the source files. As expected, that will output a DataFrame which we'll use as an input for a new function we're going to write, `calculate_nps()`.

In [6]:

```
# Define a function calculate_nps that takes a DataFrame
def calculate_nps(DataFrame):
    # Calculate the NPS score using the nps_group column
    promoter_count = DataFrame[DataFrame['nps_group'] == 'promoter']['nps_group'].count
    detractor_count = DataFrame[DataFrame['nps_group'] == 'detractor']['nps_group'].cou
    total_count = DataFrame['nps_group'].count()
    NPS_score = (promoter_count - detractor_count) / total_count * 100
    # Return the NPS Score
    return NPS_score

my_files = {
  "datasets/2020Q4_nps_email.csv": "email",
  "datasets/2020Q4_nps_web.csv": "web",
  "datasets/2020Q4_nps_mobile.csv": "mobile",
}

# Test the function on the my_files dictionary
q4_nps = combine_nps_csvs(my_files)
calculate_nps(q4_nps)
```

Out[6]:
```
9.995035578355122
```

7. Breaking down NPS by source

Is it good to have an NPS score around 10? The worst NPS score you can get is -100 when all respondents are detractors, and the best is 100 when all respondents are promoters. Depending on the industry of your service or product, average NPS scores vary a lot. However, a negative score is a bad sign because it means you have more unhappy customers than happy customers. Typically, a score over 50 is considered excellent, and above 75 is considered best in class.

Although our score is above 0, it's still on the lower end of the spectrum. The product team concludes that majorly increasing NPS across our customer base is a priority. Luckily, we have this NPS data that we can analyze more so we can find data-driven solutions. A good start would be breaking down the NPS score by the source type. For instance, if people are rating lower on the web than mobile, that's some evidence we need to improve the browser experience.

In [7]:
```
# Define a function calculate_nps_by_source that takes a DataFrame
def calculate_nps_by_source(DataFrame):
    # Group the DataFrame by source and apply calculate_nps()
    nps_by_source = DataFrame.groupby('source').apply(lambda x: calculate_nps(x))
    # Return a Series with the NPS scores broken by source
    return nps_by_source


my_files = {
  "datasets/2020Q4_nps_email.csv": "email",
  "datasets/2020Q4_nps_web.csv": "web",
  "datasets/2020Q4_nps_mobile.csv": "mobile",
}

# Test the function on the my_files dictionary
q4_nps = combine_nps_csvs(my_files)
calculate_nps_by_source(q4_nps)
```

Out[7]:
```
source
email     18.596311
```

```
mobile   -14.714048
web       22.096070
dtype: float64
```

8. Adding docstrings

Interesting! The mobile responses have an NPS score of about -15, which is noticeably lower than the other two sources. There are few ways we could continue our analysis from here. We could use the column `user_id` to reach out to users who rated poorly on the mobile app for user interviews. We could also breakdown NPS by source and date to see if there was a date where NPS started clearly decreasing - perhaps the same time there was a bug or feature realeased. With the functions we created, Brenda is in a good place to continue this work!

The last thing we'll discuss is docstrings. In Task 1, 2, 4 and 5, we included docstrings for `convert_csv_to_df()`, `check_csv()`, and `categorize_nps()`. However, we should include docstrings for all the functions we write so that others can better re-use and maintain our code. Docstrings tell readers what a function does, its arguments, its return value(s) if any, and any other useful information you want to include, like error handling. There are several standards for writing docstrings in Python, including: Numpydoc, Google-style (chosen style in this notebook), and reStructuredText.

To make sure Brenda and other colleagues can follow our work, we are going to add docstrings to the remaining undocumented functions: `combine_nps_csvs()`, `calculate_nps()`, and `calculate_nps_by_source`. It's up to you how you want to write the docstrings - we'll only check that a docstring exists for each of these functions.

In [8]:

```
# Copy and paste your code for the function from Task 3
def combine_nps_csvs(csvs_dict):
    """
    Takes in a dictionary of csv file locations and media source, checks if the csv fil
    If the csv is valid, it is imported as a pandas dataframe with a new column indicat
    All valid dataframes are concatenated and returned as a single dataframe.

    Args:
        csvs_dict (dict): a dictionary of csv file locations and their source.

    Returns:
        pandas.DataFrame
    """
    # Define combine as an empty DataFrame
    combined = pd.DataFrame()
    # Iterate over csvs_dict to get the name and source of the CSVs
    for name, source in csvs_dict.items():
        # Check if the csv is valid using check_csv()
        if check_csv(name):
            # Convert the CSV using convert_csv_to_df() and assign it to temp
            temp = convert_csv_to_df(name, source)
            # Concatenate combined and temp and assign it to combined
            combined = pd.concat([combined, temp])
        # If the file is not valid, print a message with the CSV's name
        else:
            print(name + " is not a valid file and will not be added.")
    # Return the combined DataFrame
    return combined

# Copy and paste your code for the function from Task 6
def calculate_nps(DataFrame):
    """
    Takes a dataframe and returns the nps score resulting from user reviews.

    Args:
        DataFrame (pd.DataFrame): the dataframe to be analyzed.

    Returns:
        float
    """
    # Calculate the NPS score using the nps_group column
    promoter_count = DataFrame[DataFrame['nps_group'] == 'promoter']['nps_group'].count
    detractor_count = DataFrame[DataFrame['nps_group'] == 'detractor']['nps_group'].cou
    total_count = DataFrame['nps_group'].count()
    NPS_score = (promoter_count - detractor_count) / total_count * 100
    # Return the NPS Score
    return NPS_score

# Copy and paste your code for the function from Task 7
def calculate_nps_by_source(DataFrame):
    """
    Takes a dataframe and returns the nps score resulting from user reviews broken down

    Args:
        DataFrame (pd.DataFrame): the dataframe to be analyzed.

    Returns:
        series
    """
    # Group the DataFrame by source and apply calculate_nps()
    nps_by_source = DataFrame.groupby('source').apply(lambda x: calculate_nps(x))
    # Return a Series with the NPS scores broken by source
    return nps_by_source
```