8 Midway project

Congratulations! You've made it halfway through this book. If you've been doing the exercises, I hope you've found your pandas skills improving, little by little. (And if you opened the book to this chapter without doing the exercises first, shame on you!)

Are you forgetting some of the syntax, method names, and parameter names? Are you making frustrating, "stupid" mistakes? That's only natural, and it happens to everyone, no matter how long they've been using pandas or any other large software library. Over time, though, it will become more natural and more obvious, at least when using the functionality that's most common in your work.

The whole point of this book is to gain experience and fluency through practice. Such gains

happen incrementally and over time. But they do happen, even if it doesn't always feel that way.

In this chapter, we're taking a break from exercises that concentrate on particular topics and themes. Instead, I'm going to ask you to do a small project that requires you to use many of the parts of pandas that you've learned about in the last few chapters. I hope this project gives you a chance to integrate the different skills you've learned so far.

We'll look at data from the 2020 Python Developer Survey alongside the 2021 survey from Stack Overflow. The Python survey, which is run by JetBrains (the company behind the popular PyCharm editor for Python, among others), is our best snapshot of the global Python community—who they are and what they do. Separately, the well-known programming Q&A site Stack Overflow runs an annual survey of programmers of all types, including those using Python.

Table 8.1 What you need to

know

Concept	What is it?	Example	To learn more
`pd.MultiIndex.from_tuples`	Returns a multi-index object from a list of tuples	`pd.MultiIndex.from_tuples(a_list)`	**http://mng.bz/ZqnZ**
`str.split`	Breaks strings apart, returns a list, and puts extra items on the *right*	`'abc def ghi'.split(None, 1) # returns ['abc', 'def ghi']`	**http://mng.bz/aR4z**
`str.rsplit`	Breaks strings apart, returns a list, and puts extra items on the *left*	`'abc def ghi'.rsplit(None, 1) # returns ['abc def', 'ghi']`	**http://mng.bz/aR4z**

roblem

Here is what I'd like you to do:

1. Load the CSV file (called **2020_sharing_data_outside.csv**) with results from the Python survey into a data frame. Let's call that `py_df`.

2. Turn the columns into a multi-index. How you do this depends on the column:
    1. Most of the columns have the form `first.second.third`, with two or more words separated by `.` characters. Divide the column name into two parts, one before the final `.` and one after. The multi-index column for this example would then be `('first.second', 'third')`. If there were only two parts, it would be `('first', 'second')`.
    2. In the case of about 20 columns, the top level should be `general`, and the second level should be the original column name. The columns you should treat this way are
    3. `age`,
    4. `are.you.datascientist`,
    5. `is.python.main`,
    6. `company.size`,
    7. `country.live`,
    8. `employment.status`,
    9. `first.learn.about.main.ide`,
    10. `how.often.use.main.ide`,

11. `is.python.main`,
12. `main.purposes`
13. `missing.features.main.ide`
14. `nps.main.ide`,
15. `python.version.most`,
16. `python.years`,
17. `python2.version.most`,
18. `python3.version.most`,
19. `several.projects`,
20. `team.size`,
21. `use.python.most`,
22. `years.of.coding`
23. Use the function `pd.MultiIndex.from_tuples` to create the multi-index, and then reassign it back to `df.columns`. (Hint: A function, along with a Python `for` loop or list comprehension, will come in handy here.)

3. Sort the columns so they're in alphabetical order. (This isn't technically necessary, but it makes the data easier to see and understand.)

4. Answer these questions:
    1. What are the 10 most popular Python IDEs?
    2. Which 10 other programming languages (`other.lang`) are most commonly used by Python developers?
    3. What were the 10 most common countries from which survey participants came?
    4. According to the Python survey, what proportion of Python developers have each level of experience?
    5. Which country has the greatest number of Python developers with 11+ years of experience?
    6. Which country has the greatest *proportion* of Python developers with 11+ years of experience?
5. Load the Stack Overflow data (**so_2021_survey_results.csv**) into a data frame. Let's call that `so_df`.

6. Show the average salary for different types of employment. Contractors and freelancers like to say that they earn more than full-time employees. What does the data here show you?
7. Create a pivot table in which the index contains countries, the columns are education levels, and the cells contain the average salary for each education level per country.
8. Create this pivot table again, only including countries in the OECD subset. In which of these countries does someone with an associate's degree earn the most? In which of them does someone with a doctoral degree earn the most?
9. Remove rows from `so_df` in which `LanguageHaveWorkedWith` is `NaN`.
10. Remove rows from `so_df` in which Python isn't included as a commonly used language (`LanguageHaveWorkedWith`). How many rows remain?

11. Remove rows from `so_df` in which `YearsCode` is `NaN`. How many rows remain?
12. Replace the string value `Less than 1 year` in `YearsCode` with 0. Replace the string value `More than 50 years` with 51.
13. Turn `YearsCode` into an integer column.
14. Create a new column in `so_df`, called `experience`, which will categorize the values in the `YearsCode`. Values can be
    1. Less than 1 year
    2. 1–2 years
    3. 3–5 years
    4. 6–10 years
    5. 11+ years
15. According to the Stack Overflow survey, what proportion of Python developers have each level of experience?

Working it out

This project is all about understanding the world of Python developers better, using data from two different surveys. There are hundreds, if not thousands, of other questions we could ask (and answer) using this data; if you

find this project of interest, I encourage you to continue the analysis on your own.

Load Python survey results into a data frame

We start by loading the data from the Python community survey into a data frame. On the face of it, this shouldn't be too hard:

```
py_filename = '../data/2020_sharing_data_outside.csv'
py_df = pd.read_csv(py_filename)
```

If you load the data in this way, you'll likely get a warning from pandas indicating that some columns had mixed types. We've seen this problem before; pandas does a good job of guessing a column's `dtype`, but that consumes a great deal of memory. We can either explicitly specify the `dtype`s of our columns in our call to `pd.read_csv` or (if we have sufficient memory) let pandas read all the data in and guess.

We won't be using many columns in this project, so the real-life, practical solution is to specify columns with `usecols`. However, I want you to get some practice

creating a multi-index and also have the data available for further exploration after this project is complete. Thus, we read all the data in and tell pandas to use as much memory as it needs to guess the `dtype` correctly:

```
py_filename = '../data/2020_sharing_data_outside.csv'
py_df = pd.read_csv(py_filename, low_memory=False)
```

NOTE I'm assuming that your computer has enough memory to load all the columns. If not, you should indeed pass `usecols` to `read_csv`, specifying only the columns used in this exercise. That will reduce the memory usage enough to let pandas guess correctly without over-burdening your computer.

There's nothing technically wrong with using the data frame as is. However, it contains 264 (!) columns, too many for most people to understand and think about. Moreover, although a CSV file cannot have hierarchical column names, the names were clearly designed to give us a sense of hierarchy. For example, we have

`other.lang.Java`, `other.lang.JavaScript`, `other.lang.C/C++`, and so forth—all of which could fit under an `other.lang` category.

Can we take a flat list of columns and turn it into a multi-index, thus making it easier to think about and work with? Yes, but it will take a little work. Notice that each column name is of the type `first.second.third`. If we break the column name apart at the final `.` character, we can create a multi-index in which the primary column becomes `first.second` and the secondary column `third`. We end up with a top-level column of `other.lang` and second-level columns of `Java`, `JavaScript`, `C/C++`, and so on (figure 8.1).

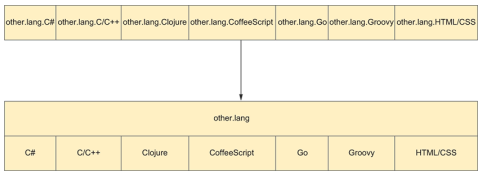

Figure 8.1 Turning a single index row into a multi-index

How can we create such a multi-index and then apply it to our data frame? We can use

`pd.MultiIndex.from_tuples`, a function that pandas provides for precisely this purpose. If we pass a list of tuples to this function, it returns a multi-index object, which we can then assign to a data frame's `index` or `columns` attribute, as appropriate. In our case, we want to assign it to `columns`, replacing the existing index object used on the columns.

First, we create the list of tuples. Each tuple's first element contains the text up to the final `.` in the column name, and the second element is the word following that final `.`. We can do this using Python's `str.rsplit` method, which works similarly to `str.split` but from the right rather than the left. By itself, `str.rsplit` won't make a difference. But if we pass a second, integer argument of 1, it returns a list of two elements split from the final `.`:

```
s = 'abcd.efgh.ijkl'
s.rsplit('.', 1)
```

This code returns `['abcd.efgh', 'ijkl']`, perfect for our purposes. (Except that it's a list, not a tuple.)

However, some of the column names don't belong in an overall category. For those, we give a top-level column of `general`. We define those columns in a list:

```
general_columns = ['age',
                   'are.you.datascientist',
                   'is.python.main',
                   'company.size',
                   'country.live',
                   'employment.status',
                   'first.learn.about.main.ide',
                   'how.often.use.main.ide',
                   'is.python.main',
                   'main.purposes'
                   'missing.features.main.ide'
                   'nps.main.ide',
                   'python.version.most',
                   'python.years',
                   'python2.version.most',
                   'python3.version.most',
                   'several.projects',
                   'team.size',
                   'use.python.most',
                   'years.of.coding'
                  ]
```

We write a function, `column_multi_name`, that takes a single column name (i.e., a string). If the column

name is one of those that gets a `general` top-level column, we return a two-element tuple containing `general` and then the existing column name. In all other cases, we return a two-element tuple based on a list we get back from `str.rsplit`:

```
def column_multi_name(column_name):
    if column_name in general_columns:              ①
        return ('general', column_name)             ②
    else:
        first, rest = column_name.rsplit('.', 1)    ③
        return (first, rest)                        ④
```

① Should this column have a general prefix?

② Returns a two-element tuple starting with general

③ Splits the tuple into a two-element list

④ Returns the elements as a tuple

We invoke this function on each column name in `py_df` and then pass the result to `pd.MultiIndex.from_tuples`. We use a list comprehension:

```
(
    pd
```

```
      .MultiIndex.from_tuples([
       column_multi_name(one_column_name)       ①
       for one_column_name in py_df.columns])   ②
  )
```

① Runs column_multi_name on the current column name

② Goes through each column name in py_df

We then assign the resulting list to `py_df.columns`, replacing the original columns with our multi-index:

```
py_df.columns = (
    pd
    .MultiIndex.from_tuples([
    column_multi_name(one_column_name)
    for one_column_name in py_df.columns ])
)
```

Sort the columns alphabetically

Most of the time, it doesn't matter whether columns are sorted. But I've found that when working with a multi-index, it's often best to sort the column names, if only to make it easier to skim through them. To do this, we take advantage of the fact that we can pass a list of columns to `py_df` to get those columns back. If we sort the list before we apply it, we

can get the columns back in a particular order. Assigning that back to `py_df` will thus sort the columns:

```
py_df = py_df[sorted(py_df.columns)]
```

What are the 10 most popular Python IDEs?

We can determine what IDEs Python developers use most often from the `ide` top-level column and the `main` second-level column, by passing a tuple:

```
py_df[('ide', 'main')]
```

We can count how often each IDE appears using `value_counts`, limiting the output to the 10 top results:

```
(
    py_df[('ide', 'main')]
    .value_counts()
    .head(10)
)
```

Which 10 other programming languages (other.lang) are most

commonly used by Python developers?

I asked you to find what other languages are most commonly used by Python developers. Non-Python languages were listed under the `other.lang` top-level index, with the particular language that each developer uses as a second-level index entry. Asking for `py_df['other.lang']` thus returns all columns under `other.lang` as a data frame—one with 54,462 rows (one for each survey respondent) and 24 columns (one for each non-Python language). Each cell contains either the name of the language or `NaN` (indicating that the survey respondent does not use this language). With this data, how can we calculate the number of people who use each of these languages?

The answer is easier than it may at first appear: the `count` method returns the number of non-`NaN` values in a series. When applied to a data frame, the `count` method returns a series whose indexes are the data frame's columns and whose values are the number

of non-null values in that column. In other words, we can run

```
py_df['other.lang'].count()
```

The result is the following series:

```
Bash / Shell    13793
C#               4460
C/C++           11623
Clojure           361
CoffeeScript      319
Go               3398
Groovy            719
HTML/CSS        15469
Java             8109
JavaScript      16662
Kotlin           1384
None             6402
Objective-C       583
Other            3592
PHP              4060
Perl              886
R                2465
Ruby             1165
Rust             1853
SQL             13391
Scala             927
Swift             854
TypeScript       3717
Visual Basic     1604
dtype: int64
```

With this series, we can sort the values in descending order:

```
(
    py_df['other.lang']
    .count()
    .sort_values(ascending=False)
)
```

Finally, we get the 10 first values:

```
(
    py_df['other.lang']
    .count()
    .sort_values(ascending=False)
    .head(10)
)
```

What are the 10 most common countries from which survey participants came?

This information is in the `general` top-level index and the `country.live` second-level index:

```
py_df[('general', 'country.live')]
```

This returns the country name for each survey respondent. To count the number of times each country appears, we can use `value_counts`:

```
py_df[('general', 'country.live')].value_counts()
```

Because `value_counts` sorts results by descending value, we can use `head(10)` to retrieve the 10 most commonly named countries in the survey:

```
py_df[('general', 'country.live')].value_counts().head(10)
```

What proportion of Python developers have each level of experience?

Once again, we can turn to `value_counts`, passing `normalize=True` to get the percentage for each level:

```
py_df[
    ('general', 'python.years')
    ].value_counts(normalize=True)
```

The greatest proportion of developers have three to five years of experience, followed by those with less than one year, followed by those with between one and two years. All told, about 75% of the respondents to the survey have been using Python for up to five years, and half of them have been using it for less than two years.

Which country has the greatest number of Python developers with 11+ years of experience?

What about the most experienced Python developers? In particular, what countries have the greatest number of Python developers with 11+ years of experience using the language? To find out, we first need to get only those rows of `py_df` in which the experience is 11+ years:

```
py_df[py_df[
    ('general','python.years')] == '11+ years']
```

But wait: we want to group by `country.live`, whose top-level index is `general`—the same as `python.years`. We can thus restrict our query, applying our boolean index only to those columns within `general`:

```
py_df['general'][py_df[
    ('general','python.years')] == '11+ years']
```

Now that we only have the columns in `general`, we can prepare a new query that gives us results on a per-country basis:

```
py_df['general'][py_df[
    ('general','python.years')] == '11+ years'
    ].groupby('country.live')
```

This sets up the grouping query to operate on a per-country basis but doesn't ask any questions. Let's determine how many non-null values each column has for each country:

```
py_df['general'][
    py_df[('general','python.years')] == '11+ years'
    ].groupby('country.live').count()
```

The resulting data frame's rows are country names, and its columns are from `general`. Values are integers indicating how many non-null rows there are for each column for each country. The difference in counts reflects occasional null values. We're interested in finding super-experienced Python developers in each country, allowing us to cut our result down to one column only, `python.years`:

```
(
    py_df['general']
    [py_df[('general','python.years')] == '11+ years']
    ['python.years']
    .groupby('country.live')
```

```
    .count()
)
```

This returns a series in which the index contains country names and the values are integers: the number of 11+ year veterans of Python.

To find which country has the most experienced Python developers, we call `sort_values`, asking for results in descending order. Then we apply `head(1)`, returning the name of the country with the most developers, as well as the number itself:

```
(
    py_df['general']
    [py_df[('general','python.years')] == '11+ years']
    .groupby('country.live')
    ['python.years']
    .count()
    .sort_values(ascending=False)
    .head(1)
)
```

Which country has the greatest proportion of Python developers with 11+ years of experience?

The US, somewhat naturally, has the greatest number of experienced developers. A

more interesting question is which country has the greatest *proportion* of Python developers with 11+ years of experience.

We need to find out how many developers are in each country. To do that, we create a new variable called `country_experience`, taken from `py_df['general']` and consisting of two columns—`country.live` and `python.years`:

```
country_experience = (
    py_df['general']
    [['country.live', 'python.years']]
)

all_per_country = (
    country_experience
    ['country.live']
    .value_counts()
)
```

We also need to get the number of senior Python developers in each country. We did that in a previous part of this exercise, but with `country_experience` in place, we have another method for determining this:

```
expert_per_country = (
    country_experience
    .loc[country_experience['python.years'] == '11+ years',
         'country.live']
    .value_counts()
)
```

We now have two series (`expert_per_country` and `all_per_country`) with matching indexes (country names). We can take advantage of the fact that pandas will use the index when dividing one series by another:

```
(expert_per_country / all_per_country
    ).sort_values(ascending=False).dropna().head(10)
```

In this code, we first divide the number of experts by the total number of Python developers per country. We then sort the values in descending order so we can find the country with the greatest proportions of experienced Python developers. To avoid null values, we use `dropna` on the resulting series, getting

```
Norway            0.265432
Ireland           0.225490
Australia         0.225420
Belgium           0.225108
```

```
Slovenia          0.224490
New Zealand       0.197917
Sweden            0.194030
Finland           0.190141
United Kingdom    0.186486
Austria           0.186170
Name: country.live, dtype: float64
```

Although the United States certainly has a very large number of senior Python developers, it's not in the top 10 when we take country size into account.

But is the data an accurate portrait of the modern Python community? After all, do one quarter of Norwegian Python developers really have more than a decade of experience? Maybe it's just me, but I'm skeptical. I wonder whether the type of person who fills out such a survey is also more enthusiastic than the average Python developer—and thus skews to a more experienced population.

Load the Stack Overflow data into a data frame, so_df

Next, we switch gears and look at the Stack Overflow survey. We load the CSV file into a data frame:

```
so_filename = '../data/so_2021_survey_results.csv'
so_df = pd.read_csv(so_filename, low_memory=False)
```

Once again, we pass `low_memory=False`, telling pandas that it should use as much memory as it needs to guess the `dtype` correctly.

Show the average salary for different types of employment

The Stack Overflow survey includes a great deal of information about people's jobs and salaries. I asked you to verify whether, based on the data collected here, freelancers and contractors earn more than full-time employees, as is often assumed to be the case. To find this out, we take the data frame and run `groupby` on `Employment`:

```
so_df.groupby('Employment')
```

This means whatever query we run, the rows are the distinct values in the `Employment` column. We're interested in the mean annual salary, reported here in dollars as `ConvertedCompYearly`, per

type of employment, which we can calculate as follows:

```
so_df.groupby('Employment')['ConvertedCompYearly'].mean()
```

This is good, but we can make it easier to compare the data points by sorting them:

```
(
    so_df
    .groupby('Employment')['ConvertedCompYearly'].mean()
    .sort_values(ascending=False)
)
```

We can see from these results that according to this survey, people who are employed full time earn the most, followed by retirees, followed by contractors:

```
Employed full-time                                      129913.094086
Retired                                                 120252.500000
Independent contractor, freelancer, or self-employed    111160.260190
I prefer not to say                                      44589.437500
Employed part-time                                       43344.532974
Not employed, and not looking for work                           NaN
Not employed, but looking for work                               NaN
Student, full-time                                               NaN
Student, part-time                                               NaN
Name: ConvertedCompYearly, dtype: float64
```

I find this hard to believe and especially wonder whether it's accurate to say that retirees

are earning almost as much as full-time employees.

There's nothing technically wrong with this result, but it's hard to read. We may want to remove the `NaN` values. Plus, dollar figures can generally be rounded to two digits after the decimal point. And maybe we can add commas before every group of three digits.

Dropping `NaN` is easy with `dropna`. But how can we format floating-point values? I would normally use an f-string:

```
x = 12345.6789
print(f'{x:,.2f}')    ①
```

① Prints 12,345.68

We can't use an f-string directly on each element of a series, but we can use a function to do it for us. In particular, we can use `apply` to run a function on each element and then use `lambda` to create an anonymous function that applies the f-string to each one, thus giving a column of strings:

```
(
    so_df
    .groupby('Employment')['ConvertedCompYearly'].mean()
    .sort_values(ascending=False)
    .dropna()
    .apply(lambda n: f'{n:,.2f}')
)
```

If you're one of the many Python developers who dislike `lambda`, you can hand `str.format` to `apply`:

```
(
    so_df
    .groupby('Employment')['ConvertedCompYearly'].mean()
    .sort_values(ascending=False)
    .dropna()
    .apply('{:,.2f}'.format)
)
```

Notice that we're not invoking the method, but passing it to `apply`, where it is invoked on each value. There isn't any value preceding the `:` in the curly braces; that's because `str.format` implicitly handles positional arguments.

To display all floats this way, we can set `pd.options.display.float_format`:

```
pd.options.display.float_format = '{:,.2f}'.format
```

This does what we did with `apply`, telling pandas to invoke this method whenever it sees a floating-point value.

Now let's ask a different question: rather than looking at average salaries for different types of work, let's instead look at average salaries for different levels of education. Moreover, let's further divide that by country. What I'm asking for, of course, is a pivot table—one in which the index contains country names, the columns contain the distinct values from `EdLevel`, and the cells contain the mean of `ConvertedCompYearly` for each country-education combination:

```
so_df.pivot_table(index='Country',
        columns='EdLevel',
        values='ConvertedCompYearly')
```

Create this pivot table again, only including countries in the OECD subset

Next, I asked you to load the subset of OECD countries into a data frame:

```
oecd_filename = '../data/oecd_locations.csv'
oecd_df = pd.read_csv(oecd_filename,
    header=None, index_col=1,
    names=['abbrev', 'Country'])
```

The data frame we create in this code uses the country name for the index. That's because we're next going to use it in a `join` with `so_df`, so the indexes need to be aligned. The country names will act as indexes.

We join our OECD subset data frame with the Stack Overflow data and then re-create our pivot table. The effect is to reduce the number of rows (i.e., countries) in our output. And indeed, once we run our `join`, we get back only 13 rows, one for each country in the OECD subset:

```
(
    oecd_df
    .join(so_df
          .set_index('Country'))
    .pivot_table(index='Country',
                 columns='EdLevel',
                 values='ConvertedCompYearly')
)
```

Notice that we call `so_df.set_index('Country')` to temporarily set the country

to be the index of `so_df`. That allows us to join it with `oecd_df`—and then create the pivot table, which is our ultimate goal.

Now that we know average salaries in all these countries and for all education levels, we can ask some questions of the data. For example, I asked you to determine in which country someone with an associate's degree can expect to earn the most. We could have stored the pivot table to a variable, but instead we chain the relevant methods together:

```
(
    oecd_df
    .join(so_df.set_index('Country'))
    .pivot_table(index='Country',
                 columns='EdLevel',
                 values='ConvertedCompYearly')
    ['Associate degree (A.A., A.S., etc.)']
    .sort_values(ascending=False)
)
```

After creating the pivot table, we retrieve the column for associate's degrees and sort them from highest to lowest. From the results we see here, it looks like the country that offers the best pay for people with an associate's degree is

Australia, followed by Germany and Israel.

What about PhDs? Do countries that pay well for an associate's degree also pay well if you have a PhD or similar post-graduate degree? We can perform a similar query:

```
(
    oecd_df
    .join(so_df.set_index('Country'))
    .pivot_table(index='Country',
                 columns='EdLevel',
                 values='ConvertedCompYearly')
    ['Other doctoral degree (Ph.D., Ed.D., etc.)']
    .sort_values(ascending=False)
)
```

There does seem to be some overlap; the highest-paying countries for PhDs are Japan, Australia, France, Israel, and Germany.

There may also be some reason to suspect that this data isn't totally accurate; is it really possible that the mean salary in Hungary for someone with an associate's degree is $63,000/year, whereas with a PhD it's only $52,000/year? Or that there is a salary difference of only

$12,000/year between Germans with an associate's degree and a PhD?

My point is that data analysis requires more than just number crunching—you also have to ask whether the numbers make sense. And if they don't, you should ask yourself why that may be the case. For example, perhaps the sample sizes are so small that the data isn't truly representative of the total population.

Remove rows from so_df in which LanguageHaveWorkedWith is NaN

Next we want to analyze Python programmers in the Stack Overflow survey. The `LanguageHaveWorkedWith` column allows us to identify who they are—but that column contains text, with languages separated from one another with `;` characters. So, someone who works with both Python and JavaScript could have a value of `Python;JavaScript`. If we want people who work with Python, we need to find those who have "Python" in that

column. For that, we can use `str.contains` to look inside the string. But there's a problem: some survey respondents didn't fill out this information, which means it's `NaN`. And trying to run `str.contains` on a `NaN` value will result in an error.

We thus need to first remove all rows that contain `NaN` for `LanguageHaveWorkedWith`. We can do that by running `dropna`, telling it to only look at the `LanguageHaveWorkedWith` column:

```
so_df = (
    so_df
    .dropna(subset=['LanguageHaveWorkedWith'])
)
```

Remove rows from so_df in which Python isn't in LanguageHaveWorkedWith

Once we've done that, we can be sure all values in `LanguageHaveWorkedWith` are strings. We apply `str.contains` and look for "Python":

```
so_df = (
    so_df
```

```
    .loc
    [so_df['LanguageHaveWorkedWith'].str.contains('Python')]
)
```

We end up with nearly 40,000 people who use Python—a smaller sample than the 54,000 who responded to the Python survey, but still a substantial sample size. Also, although the survey asked what languages people had used in the last year, we don't know whether they used Python once in the last year, every day, or somewhere in between.

Now that we have found the Python developers from Stack Overflow, we would like to compare them with the respondents to the Python community survey. In particular, we'd like to know if they have similar levels of experience. But in the data's original form, it's not possible to determine that: whereas the Python community survey lumps people into categories (e.g., "Less than 1 year" and "1–2 years"), the Stack Overflow survey asks for a specific number of years of experience.

Remove rows from so_df in which YearsCode is NaN

To do this, we first remove all rows in which `YearsCode` is `NaN`:

```
so_df = so_df.dropna(subset=['YearsCode'])
```

In YearsCode, replace "Less than 1 year" with 0 and "More than 50 years" with 51

I asked you to create a new column called `experience` in the Stack Overflow data frame, which turns the raw year numbers into categories. We know we can use `pd.cut` to accomplish this, but `pd.cut` works only if all values in a column are numeric—and in this case, two options are non-numeric: `Less than 1 year` and `More than 50 years`. Our first task is thus to turn those into numbers:

```
so_df.loc[so_df['YearsCode'] ==
    'Less than 1 year', 'YearsCode'] = 0
so_df.loc[so_df['YearsCode'] ==
    'More than 50 years', 'YearsCode'] = 51
```

With these integer values in place, we can turn `YearsCode` into an integer column:

```
so_df['YearsCode'] = so_df['YearsCode'].astype(int)
```

Create a new column called "experience" in so_df, categorizing values in YearsCode

Now we can use `pd.cut` to re-create the same categories we had in the Python community survey:

```
so_df['experience'] = pd.cut(so_df['YearsCode'],
       bins=[-1, 1, 2, 5, 10, 100],
      labels=['Less than 1 year',
      '1-2 years',
      '3-5 years',
      '6-10 years',
      '11+ years'])
```

Remember that `pd.cut` uses the numbers passed to the `bins` keyword argument as the extreme edges of the bins it defines—which means if we want to give the first label to a number of 0, we should start the bin at –1. And yes, there is the option of including values on the left (or right), but I decided this is easier and ensures that bins don't overlap.

According to the Stack Overflow survey, what proportion of Python

How many developers have each level of experience?

Next, we want to see the distribution of experience levels in the Stack Overflow survey. We again use `value_counts`:

```
11+ years           0.373388
6-10 years          0.318589
3-5 years           0.222530
1-2 years           0.047440
Less than 1 year    0.038054
Name: experience, dtype: float64
```

From this, we can see that Stack Overflow respondents are much more experienced than Python survey respondents. As you may recall, 75% of the Python survey respondents have been using Python for up to five years, whereas in the Stack Overflow survey, the number of new programmers is about 25%. Half of the Python survey respondents have been using it for less than two years, whereas that's true for less than 10% of the Stack Overflow group.

However, we need to think before we say anything too sweeping when comparing

these surveys. After all, the Stack Overflow survey asked about all the experience the respondent has had as a programmer, whereas the Python survey asked how long the person had been programming in Python. The same person, filling out both surveys, might have been programming in Java for 20 years and Python for only 2 and would thus have answered the questions differently on each survey. Making such comparisons and integrating data from different sources can be tricky and requires some thought; just joining two data frames isn't sufficient. That said, it is interesting to see just how heavily the Python survey skewed toward newcomers and how heavily Stack Overflow skewed toward experienced developers. The Python community survey might do well to include an "overall programming experience" question in the future, to help with such analysis and to better understand how much Python plays a role in the members of its community.

olution

```
py_filename = '../data/2020_sharing_data_outside.csv'
py_df = pd.read_csv(py_filename, low_memory=False)

general_columns = ['age', 'are.you.datascientist',
                   'is.python.main', 'company.size',
                   'country.live', 'employment.status',
                   'first.learn.about.main.ide',
                   'how.often.use.main.ide',
                   'is.python.main', 'main.purposes'
                   'missing.features.main.ide'
                   'nps.main.ide',
                   'python.version.most',
                   'python.years',
                   'python2.version.most',
                   'python3.version.most',
                   'several.projects',
                   'team.size',
                   'use.python.most',
                   'years.of.coding'
                  ]

def column_multi_name(column_name):
    if column_name in general_columns:
        return ('general', column_name)
    else:
        first, rest = column_name.rsplit('.', 1)
        return (first, rest)

py_df.columns = pd.MultiIndex.from_tuples(
                  [column_multi_name(one_column_name)
                  for one_column_name in py_df.columns])

py_df = py_df[sorted(py_df.columns)]

py_df[('ide', 'main')].value_counts().head(10)

py_df['ide'].value_counts().head(10)

py_df['other.lang'].count().sort_values(ascending=False).head(10)
```

```
py_df['general', 'country.live'].value_counts().head(10)

py_df[('general', 'python.years')].value_counts(normalize=True)

(
    py_df['general']
    [py_df[('general','python.years')] == '11+ years']
    .groupby('country.live')['python.years'].count()
    .sort_values(ascending=False)
    .head(1)
)

country_experience = (
    py_df['general']
    [['country.live', 'python.years']]
)

all_per_country = (
    country_experience['country.live']
    .value_counts()
)

expert_per_country = (
    country_experience
    .loc[
        country_experience['python.years'] == '11+ years',
        'country.live']
    .value_counts()

(expert_per_country / all_per_country).sort_values(
    ascending=False).dropna().head(10)
```

Is that it? No, not at all! But our printing and formatting system requires that we take a break after 60 lines. And hey, if you've been reading more than 60 lines of code, maybe you should take a break before continuing.

All set? Here's the rest of the code:

```
so_filename = '../data/so_2021_survey_results.csv'
so_df = pd.read_csv(so_filename, low_memory=False)

so_df.pivot_table(index='Country', columns='EdLevel',
    values='ConvertedCompYearly')

oecd_filename = '../data/oecd_locations.csv'
oecd_df = pd.read_csv(oecd_filename, header=None,
    index_col=1, names=['abbrev', 'Country'])

(
    oecd_df
    .join(so_df
          .set_index('Country'))
    .pivot_table(index='Country',
                 columns='EdLevel',
                 values='ConvertedCompYearly')
)

(
    oecd_df
    .join(so_df.set_index('Country'))
    .pivot_table(index='Country',
                  columns='EdLevel',
                  values='ConvertedCompYearly')
    ['Associate degree (A.A., A.S., etc.)']
    .sort_values(ascending=False)
)

(
    oecd_df
    .join(so_df.set_index('Country'))
    .pivot_table(index='Country',
                 columns='EdLevel',
                 values='ConvertedCompYearly')
    ['Other doctoral degree (Ph.D., Ed.D., etc.)']
    .sort_values(ascending=False)
)
```

```
so_df = so_df.dropna(subset=['LanguageHaveWorkedWith'])
so_df = so_df[so_df['LanguageHaveWorkedWith'].str.contains('Python')]

so_df.shape

so_df = so_df.dropna(subset=['YearsCode'])

so_df.shape

so_df.loc[so_df['YearsCode'] ==
    'Less than 1 year', 'YearsCode'] = 0
so_df.loc[so_df['YearsCode'] ==
    'More than 50 years', 'YearsCode'] = 51

so_df['YearsCode'] = so_df['YearsCode'].astype(int)

so_df['experience'] = pd.cut(so_df['YearsCode'],
       bins=[-1, 1, 2, 5, 10, 100],
      labels=['Less than 1 year',
      '1-2 years',
      '3-5 years',
      '6-10 years',
      '11+ years'])

so_df['experience'].value_counts(normalize=True)
```

ummary

Whew! This was a big, long exercise, meant to help you integrate and use many of the ideas and techniques we've discussed in this book so far. Of course, there are many pieces of pandas that we didn't use in this project—but to be honest, it's a rare project that uses all the capabilities pandas has to offer. That said, we did a lot of things here: loading

and cleaning data, joining data frames, analyzing data, and even comparing different data sets and thinking critically about how trustworthy they are. If you felt comfortable with all the techniques in this project, I'd say you're well on your way to internalizing the way pandas does things and using it productively in your projects.