9 Strings

When most people think of pandas or data analysis in general, they think of numbers. And indeed, much of the work that people do with pandas is with numbers. That's why pandas is built on top of NumPy, which takes advantage of C's fast, efficient integers and floats. And that's why so many of the exercises in this book involve working with numbers.

However, we often have to work with textual data—usernames, product names, sales regions, business units, ticker symbols, and company names are just a few examples. Sometimes the text is central to the analysis you're doing—such as when you're preparing data for a text-based machine-learning model—and other times, it's secondary to the numbers and used as a description or categorical data.

It turns out that pandas is also well-equipped to handle text. It does this not by storing string data in NumPy but rather by using fully fledged string objects: either those that come with Python or (more recently) a pandas-specific string class that reduces both ambiguity and errors. (I’ll have more to say about these two string types and when to use each one later in the chapter.) In either case, we can apply a wide variety of string methods to our data.

This is normally done via the `str` accessor, available on every pandas series that contains strings. When we invoke a method via `str`, we get back a new series. The returned series can replace the existing one, be assigned to a new variable, or be assigned as a new column alongside the original one.

In this chapter, you’ll work through exercises that help you identify and understand how to work with textual data and the `str` accessor in pandas. After going through these exercises, you’ll know which string methods are available, feel more comfortable using them, and know how to apply your own custom functions to string columns.

Text data types

For many years, pandas used Python’s internal string type to store text. This was a big improvement over NumPy, which stores characters in C arrays—more efficient than Python strings but with much more limited functionality. To refer to such Python strings, pandas assigned a `dtype` of `object`. The good news is that this worked fairly well, giving great string functionality within pandas. The bad news was that a series could contain *any* type of Python object, not just strings. This led to bugs because we could accidentally store a list, dictionary, or `None` into such a column without noticing. After all, these are all Python objects, so there was no way for pandas to stop us from adding them.

Pandas 1.0.0 added a new `pd.StringDtype` to solve such problems. As the name indicates, it is meant to be used as a `dtype` on a series. Because it’s specific to textual data, we cannot mix it up with other types of objects. Further, the pandas documentation indicates that this will, at some point, be the standard string type for pandas.

But wait—previously, a series with a `dtype` of `object` could be a string and could also be `NaN`. What happens now? After all, `NaN` isn't an instance of `pd.StringDtype` but rather of `float`. The answer is that if you're going to use `pd.StringDtype`, you should also use `pd.NA` instead of `NaN`. You can think of `pd.NA` as a more flexible version of `NaN` that is compatible with all pandas `dtype`s.

Should you use `pd.StringDtype`? As of this writing, the pandas documentation is inconsistent: on the one hand, it lists several benefits of `pd.StringDtype`. On the other hand, it says "`StringDType` is considered experimental. The implementation and parts of the API may change without warning."

In this chapter, I'll assume that you are using the old-fashioned (and definitely stable) `object` type in your columns. However, you will likely need (and want) to switch to `pd.StringDType` in the future.

If all goes well, doing so will mean no changes to your programs other than better checking of your values and potentially even better performance.

Table 9.1 What you need to

know

Concept	What is it?	Example	To learn more
`s.explode`	Returns a new series with each element on its own line	`s.explode()`	http://mng.bz/RxDP
`str.contains`	Returns a series of booleans, indicating which elements of the input series contain the target string	`s.str.contains('a')`	http://mng.bz/2D2X
`str.get_dummies`	Returns a data frame containing 1s and 0s based on a categorical series	`s['country'].get_dummies(sep=';')`	http://mng.bz/1q2g
`str.index`	Returns a series of integers, each indicating where the target string was found in the	`s.str.index('a')`	http://mng.bz/PzEP

	corresponding element of the input series		
`str.len`	Returns a series of integers indicating the length of each element	`s.str.len()`	http://mng.bz/Jg0v
`str.replace`	Returns a series based on an existing series, replacing the first argument with the second	`s.str.replace('a', 'e')`	http://mng.bz/wv6Q
`str.split`	Returns a series in which each element is a list of strings; the argument specifies the delimiter used to perform the split	`s.str.split(';')`	http://mng.bz/qrD2
`str.strip`	Returns a series of Python strings without the argument’s	`s.str.strip('.!?')`	http://mng.bz/7D2y

	characters on either side		
`s.isin`	Returns a boolean series indicating whether a value in `s` is an element of the argument	`s.isin(['A', 'B', 'C'])`	http://mng.bz/mVW2
`i.intersection`	Returns a new index object containing elements in two existing index objects	`i.intersection(i2)`	http://mng.bz/5w21

The str accessor

Traditional Python strings support a large number of methods and operators ranging from search (`str.index`) to replacement (`str.replace`) to substrings (slices) to checks of the string's content (e.g., `str.isdigit` and `str.isspace`). But if we have a series containing strings, how can we invoke such a method on every element?

Experienced Python developers would normally use a `for` loop or perhaps a list comprehension. But in pandas, we do whatever we can to avoid such loops because of their inefficiency. We could use the `apply` method, invoking a function to every element of a series. And indeed, `apply` is needed if we want to use a custom function.

In many cases, though, there's a better way: the `str` accessor. Using `str` gives us access to a variety of string methods—including, but not limited to, standard Python

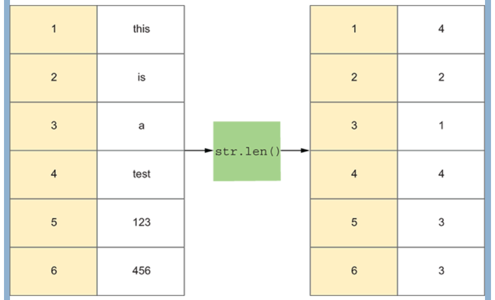
string methods. A method invoked via `str` is applied to every element in the series. It returns a new series of the same length and with the same index, whose values are the results of invoking the method on each element. For example, we can get the length of a string by invoking the `len` method on the `str` accessor:

```
s = Series('this is a test 123 456'.split())
s.str.len()
```

The result is a new series containing the lengths of the values in `s`:

```
0    4
1    2
```

```
2    1
3    4
4    3
5    3
dtype: int64
```



Getting the lengths of all strings in a series with `.str.len()`

What if we want to find all values in `s` that can be turned into integers?

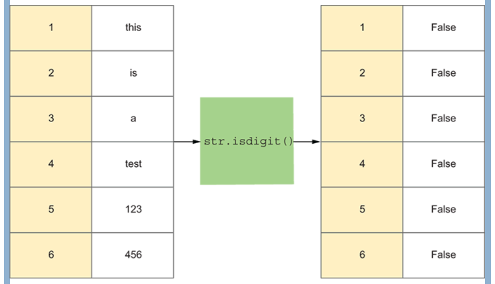
```
s.str.isdigit()
```

The result is a boolean series indicating which values contain only the characters 0–9:

```
0    False
1    False
2    False
3    False
4     True
5     True
dtype: bool
```

Because it contains only booleans and shares an index with `s`, it's suitable for use as a boolean (mask) index on `s` to find numeric values:

```
s.loc[s.str.isdigit()]
```



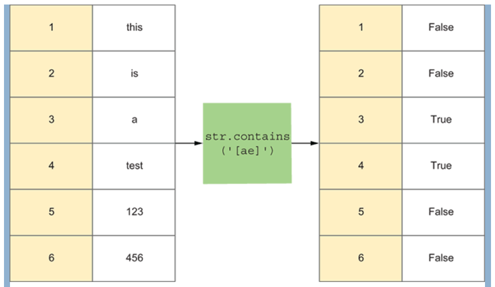
Finding which elements of a series contain only digits

The `str` accessor supports methods beyond those available in Python’s `str` class. For example, we can search in a string using `contains`. However, `contains` allows us to use a regular expression. We can thus find all words containing either *a* or *e*:

```
s.str.contains('[ae]')
```

This query returns the following series:

```
0    False
1    False
2     True
3     True
4    False
5    False
dtype: bool
```



Finding which elements of a series contain either *a* or *e*:

Applied to our original series `s`, we can find all words that contain either *a* or *e*:

```
s.loc[s.str.contains('[ae]')]
```

This results in

```
2       a
3    test
dtype: object
```

Note that although `str.contains` currently (as of this writing) defaults to treating its argument as a regular expression, there are plans for that default value to change. It's thus a good idea to be explicit about your intentions by passing `regex=True` so the string isn't taken literally:

```
s[s.str.contains('[ae]', regex=True)]
```

The `str` accessor makes it easy to use pandas to call string methods and work with textual data. However, you should spend some time reviewing the list of string methods in the pandas documentation to get a good sense of what they are and what they can do.

xercise 36 • Analyzing Alice

In this exercise, we'll look at the famous book *Alice in Wonderland*, the text of which is made freely available via Project Gutenberg and included with the data files for this book. Here is what I'd like you to do:

1. Open the file **alice-in-wonderland.txt**, and read it into a pandas series or data frame such that each word is a separate value. (If you choose to read it as a data frame, that's fine. I'll refer to the "series" or "column" when describing the data in this exercise.)

2. Answer these questions:
    1. What are the 10 most common words in the book?
    2. Does this change if you count the words without regard to case?
    3. Does this change if you remove all the punctuation (as defined in `string.punctuation`) from the beginning and end of each word?
    4. How many capitalized words does the book contain?
    5. If you ignore punctuation and quotes before the start of a word, how many capitalized words does the book contain?
    6. Count the number of vowels (a, e, i, o, and u) in each word. What is the average number of vowels per word?

Working it out

In this exercise, we use the string functionality in pandas in a variety of ways. To begin, I asked you to read the contents of *Alice in Wonderland* into a series. Normally, we don't read text files into pandas—although to be honest, a library such as pandas has so many users and use cases that it's possible people do this on a regular basis. If you were to feed `open(filename)` into `Series`, the series would contain the lines from alice-in-wonderland.txt. Instead of that, I asked you to create a series containing the separate words from the file.

To turn the file into a series of words, we need to do the following:

1. Read the entire file into Python as a string with the `read` method.
2. Break the string into a list of strings with the `str.split` method.
3. Turn the resulting list into a series.

Here's the code we use to do this:

```
filename = '../data/alice-in-wonderland.txt'
s = Series(open(filename).read().split())
```

NOTE The `read` method returns a string containing the contents of the file. What if the file contains several terabytes of data? Unless the IT department at your company is unusually generous, you’ll find yourself running out of memory. Normally, I suggest that people *not* read an entire file into memory at once and instead iterate over its lines. In this particular case, I know the file is small and there won’t be any problems with reading it all at once.

With our series in place, we can analyze the text it contains. First, I asked you to find the most common words. As we’ve seen countless times before, `value_counts` will help us here. Invoking it on our series returns a new series whose index contains our words (i.e., the values from `s`) and whose values (sorted in descending order) are integers describing how many times each word appears in `s`:

```
s.value_counts()
```

Not surprisingly, the most common words are *the*, *and*, *a*, and *to*. But what if these words appeared at the start of a sentence? They would be capitalized and wouldn’t be included in our count. How can we transform all the words to lowercase and then find how common they are? We can use the `str` accessor to run the `lower` method on our series. That returns a new series of strings on which we can run `value_counts`:

```
s.str.lower().value_counts().head(10)
```

But wait a second—because of the way we create our series, using whitespace characters to indicate the boundary between words, it's possible that the words have punctuation marks before or after their letters. I thus asked you to repeat the query for the 10 most common words, but only after removing/ignoring punctuation characters. This turns out to be easier than you may imagine using the `str.strip` method. This method is typically use to remove whitespace from the start or end of a string:

```
s = '   abc   '
s.strip()            ①
```

① Returns 'abc'

But we can also pass a string argument to `str.strip`, removing any characters that appear in that argument from the start or end of the string:

```
s = ':;:;abc:;:;'
s.strip(':;')     ①
```

① Returns 'abc' after removing all occurrences of : and ; from the start and end of s

The `string` module provides a number of predefined strings, including `string.punctuation`, which comes in handy on such occasions:

```
import string
s = ':;:;abc:;:;'
s.strip(string.punctuation)     ①
```

① Returns 'abc'

Given a series `s` containing strings, we can get a new series containing those same strings, but without leading and trailing punctuation, by invoking `split` via the `str` accessor:

```
s.str.strip(string.punctuation)
```

To find the 10 most common words in `s`, ignoring punctuation, we can thus say

```
(
    s
    .str
    .strip(string.punctuation)
    .value_counts()
    .head(10)
)
```

And although I didn't ask you to do this, we could get the 10 most common words, ignoring both case and punctuation:

```
(
    s
    .str
    .lower()
    .str
    .strip(string.punctuation)
    .value_counts()
    .head(10)
)
```

Notice that we use `str` twice here: once to run `lower` on the original series `s` and a second time to run `strip` on the series of strings returned by `str.lower`. We'll see more examples of this as we review the other parts of this exercise.

Next, I asked you to count the number of capitalized words in the book. This means finding all words that begin with a capital letter, from *A* through *Z*. There are several ways to do this, but my favorite is to use a regular expression. Given that the pandas string method `str.contains` supports regular expressions, we can say the following:

```
s.str.contains('^[A-Z]\w*$',      (1)
          regex=True)
```

(1) Words starting with a capital letter followed by zero or more alphanumeric characters

This returns a boolean series with the same index as `s`. The value is `True` whenever the word starts with a capital letter (anchored to the start of the string with `^`) and contains zero or more alphanumeric characters (`\w*`) through the end of the word. (We have to allow for zero or more characters because of single-letter capitalized words such as *A* and *I*.)

With this in hand, we can apply the boolean series to `s`:

```
(
    s
    .loc[s.str.contains('^[A-Z]\w*$', regex=True)]
)
```

Then we can apply the `count` method to find how many values the series contains:

```
(
    s
    .loc[s.str.contains('^[A-Z]\w*$', regex=True)]
    .count()
)
```

But wait: what if the word has a punctuation mark, such as quotes, before the initial capital letter? To get an accurate count, we need to remove punctuation from both ends of the words and look for which are capitalized. Here’s how we can do that:

```
(
    s
    .loc[s.str.strip(string.punctuation )
    .str
    .contains('^[A-Z]\w*$', regex=True)]
    .count()
)
```

Here we first remove punctuation from the start and end of each word and feed the resulting series into `str.contains` with our regular expression. That returns a boolean series we can apply back to `s`, thus finding the total number of capitalized words.

Next, I asked you to calculate the mean number of vowels in each word. This requires first finding a way to calculate the number of vowels in each word and then calculating the mean value. The easiest way to do this is with the `apply` method, which lets us run a function of our choice on each element of the series. We start by writing a function that counts vowels:

```
def count_vowels(one_word):
    total = 0
    for one_letter in one_word.lower():
        if one_letter in 'aeiou':
            total += 1

    return total
```

This is a simple Python function that takes a string as an argument, counts the vowels in it, and returns an integer. We can apply this function to every element of our series `s`, getting a new series back:

```
s.apply(count_vowels)
```

I asked for the mean number of vowels in each word. Because we now have a series of integers, we can get that back with

```
s.apply(count_vowels).mean()
```

Solution

```
filename = '../data/alice-in-wonderland.txt'
s = Series(open(filename).read().split())                      ①

s.value_counts().head(10)                                      ②
s.str.lower().value_counts().head(10)                          ③
s.str.strip(string.punctuation).value_counts().head(10)        ④

(
    s
    .loc[s.str.contains('^[A-Z]\w*$', regex=True)]
)                                                              ⑤

(
    s
    .loc[s.str.contains('^[A-Z]\w*$', regex=True)]
    .count()
)                                                              ⑥

def count_vowels(one_word):                                    ⑦
    total = 0
    for one_letter in one_word.lower():
        if one_letter in 'aeiou':
            total += 1

    return total

s.apply(count_vowels).mean()                                   ⑧
```

① Creates a series based on the words in the file

② Which 10 words appear most often?

③ If we ignore case, which 10 words appear most often?

④ If we ignore punctuation before and after the word, which 10 words appear most often?

⑤ How many capitalized words appear in the book?

⑥ Ignoring leading and trailing punctuation, how many capitalized words appear?

⑦ Defines a function that returns the number of vowels in a given string

⑧ What is the mean number of vowels in our words?

You can explore a version of this in the Pandas Tutor at **http://mng.bz/y82d**.

Beyond the exercise

- What is the mean of all integers in *Alice*?
- What words in *Alice* don't appear in the dictionary? Which are the five most common such words? (For the purposes of this exercise, I used a version of Linux's dictionary file available from **http://mng.bz/MZWB**.)
- What are the minimum and maximum number of words per paragraph?

xercise 37 • Wine words

If you’re like me, you may enjoy having a glass of wine with your dinner. On occasion, you may even read the wine’s description on the back of the bottle, where the winemaker uses flowery language to describe the winemaking process and the flavors you may detect when drinking the wine. I know I’m not the only person who sometimes raises an eyebrow at the words used in these descriptions. I decided to use pandas to better understand what words are used in describing wine and whether we can find any interesting insights from these words.

We looked at the wine-review database earlier, in exercise 35. In this exercise, we’ll examine the words that reviewers use to describe the wines and see if particular words are more likely to occur related to specific provinces and varieties. Along the way, we’ll find ways to use pandas to analyze text in some new ways. Here’s what I want you to do:

1. Open the file **winemag-150k-reviews.csv**, and read it into a data frame. You only need the columns `country,province,description,` and `variety.`

2. Answer these questions:
    1. What are the 10 most common words containing five or more letters in the wine descriptions? Turn all words into lowercase and remove all punctuation and symbols at the start or end of each word for easier comparison. Also remove the words *flavors*, *aromas*, *finish*, *palate*, and *drink*.
    2. What are the 10 most common words for non-California wines?
    3. What are the 10 most common words for French wines?
    4. What are the 10 most common words for white wines? For our purposes, look for Chardonnay, Sauvignon Blanc, and Riesling.
    5. What are the 10 most common words for red wines? For our purposes, look for Pinot Noir, Cabernet Sauvignon, Syrah, Merlot, and Zinfandel.
    6. What are the 10 most common words for rosé wines?

3. Show the 10 most common words for the five most common wine varieties.

Working it out

First, I asked you to create a data frame with the wine information. We only need four columns, so we can load just those:

```
filename = '../data/winemag-150k-reviews.csv'
df = pd.read_csv(filename,
                 usecols=['country','province',
                   'description', 'variety'])
```

Next I wanted to start performing some analysis on the words. But because we'll be running the same type of analysis on different subsets of the data frame, we can benefit from writing a function. What does this function need to do?

- Accept a series of text (i.e., wine descriptions)
- Turn the text into lowercase (for easier comparison)
- Turn that into a series of individual words
- Remove leading and trailing punctuation
- Remove words with fewer than five letters
- Remove common wine-related words

- Find the 10 most commonly occurring words

Fortunately, it’s not difficult to write such a function, which we call `top_10_words`. The function expects to receive one argument, a pandas series of strings, which we call `s`. Each string in the series is assumed to contain multiple words separated by whitespace.

The first thing we want to do is turn all the strings in the series to lowercase for easier counting. We can do that using the `str` accessor and the `lower` method:

```
words = (
    s
    .str
    .lower()
    )
```

We next want to turn our series of sentences into a series of words. That is, instead of having multiple words in each row, we want a single word in each row. This means we’ll create a series that’s larger—potentially *much* larger—than the input series `s`.

If you’re familiar with Python string methods, you won’t be surprised that we use the `split` method here via the `str` accessor. (Note that this means we need to specify `str` a second time so we can run `split` on each element of the series returned from `str.lower()`.) `split` takes a string and breaks it apart wherever it encounters a delimiter such as `:` or `,`. In this case, we don't specify a delimiter, so `split` uses any whitespace—space, tab, newline, carriage return, or vertical tab—to break the strings apart:

```
words = (
         s
         .str
         .lower()
         .str
         .split()
)
```

The good news is that we have separated the words from one another. The bad news is that our series still contains the same number of rows as before. Now each row contains a list of strings rather than a single string.

Fortunately, the `explode` method takes a series containing an iterable of objects (e.g., a list of strings) and returns a new series in which each object has its own row. We can thus get each word in its own row as follows:

```
words = (
        s
        .str
        .lower()
        .str
        .split()
        .explode()
)
```

We could stop there, but let's clean things up a bit more by removing any punctuation characters at the beginning or end of any word. That will avoid problems when counting words that come at the start or end of a sentence; otherwise we might include leading and trailing punctuation. The easiest way to do this is with the Python `str.strip` method. We normally think of `strip` as a method that removes whitespace at the start or end of a string, but that's just the default behavior. We can pass a string containing characters we want to remove from the beginning and end of each string. The result is a new series in which the strings don't have any of these characters at their start or end:

```
words = (
    s
    .str
    .lower()
    .str
    .split()
    .explode()
    .str
    .strip(',$.?!$%')
)
```

We now have in `words` a series containing individual, lowercase words without any leading or trailing punctuation. Next we want to remove words that have fewer than five letters. We can do that using a boolean index based on the output from the `len` method on the `str` accessor:

```
words.loc[(words.str.len()>=5)]
```

But that's not the only filter we want to put on `words`. We also want to remove a number of common words that crop up in nearly every wine description or review. We can use the `isin` method in a series, passing a list of strings as an argument, to determine which rows are and aren't in that list:

```
common_wine_words = ['flavors', 'aromas', 'finish', 'drink', 'palate']
~words.isin(common_wine_words)
```

We can combine these two mask indexes to get only words that contain at least five characters and don't appear in `common_wine_words`:

```
(
    words
    .loc[(words.str.len()>=5) &
         (~words.isin(common_wine_words))]
)
```

Note that we use `~`, the boolean "not" operator in pandas, to flip the boolean index that we get back from `words.isin`.

Our function is called `top_10_words` because it's supposed to return the 10 most common words found in the wine reviews. Given that `words` is now a series of words, we can run `value_counts`, followed by `head(10)`, and return the 10 words most commonly found:

```
return (
    words
    .loc[(words.str.len()>=5) &
         (~words.isin(common_wine_words))]
    .value_counts()
    .head(10)
)
```

With this, we have a complete function, `top_10_words`, that we can apply to any series of words:

```
def top_10_words(s):
    common_wine_words = ['flavors', 'aromas',
            'finish', 'drink', 'palate']

    words = s.str.lower().str.split(
        ).explode().str.strip(',$.?!$%')

    return (
        words
        .loc[(words.str.len()>=5) &
             (~words.isin(common_wine_words))]
        .value_counts()
        .head(10)
    )
```

We can apply our function to all wines in the review database:

```
top_10_words(df['description'])
```

I asked you to find the 10 most common words used in French wine reviews. We need to extract the `description` column for wines made in France:

```
df.loc[df['country'] == 'France', 'description']
```

We can pass the resulting series to `top_10_words`:

```
top_10_words(
    df
```

```
        .loc[df['country'] == 'France',
             'description']
             )
```

Next, I asked you to find the words most commonly associated with wines made outside of California. We need to search on the `province` column and then apply the `!=` operator to find those from outside of that state:

```
top_10_words(
    df
    .loc[df['province'] != 'California',
    'description']
    )
```

Notice that in this data set, you have to pay attention to the `province` column, which is distinct from the `country` column. Additional columns allow you to zero in on a particular region within a country; as you may know, different regions are known for producing not only different types of wines but also distinctive flavors specific to those regions.

Next, I thought it would be interesting to compare the words used most often for white, red, and rosé wines. I gave a (very nondefinitive) list of wines of each type and asked you to find the top 10 words used in each of their descriptions. The queries are identical except for the lists:

```
top_10_words(
    df.loc[df['variety']
           .isin(['Chardonnay',
                  'Sauvignon Blanc',
                  'Riesling']),
           'description'])

top_10_words(
    df.loc[df['variety']
           .isin(['Pinot Noir',
                  'Cabernet Sauvignon',
                  'Syrah', 'Merlot',
                  'Zinfandel']),
          'description'])

top_10_words(
    df.loc[df['variety'] == 'Rosé',
           'description'])
```

Notice how the `isin` method allows us to perform an “or” search—one that we could certainly do with pandas boolean operators and a mask index but that is shorter and more readable with `isin`.

Finally, I asked you to find the 10 most common words for the five most commonly mentioned wine varieties. To do that, we first need to determine the five most-mentioned varieties:

```
(
    df['variety']
    .value_counts()
    .head(5)
    .index
)
```

Here we run `value_counts` on the varieties to determine how common each variety is in the database. We use `head(5)` to find the five most common varieties. We can then find all reviews for one of these varieties using `isin`:

```
(
    df
    .loc[df['variety']
         .isin(df['variety']
                .value_counts()
                .head(5)
                .index),
    'description']
)
```

Notice that we couldn't just use `isin` on the values we got back from `value_counts`, because those would be numbers. Instead, we have to check the index of the resulting series, which contains words.

Finally, we can find the top 10 words used in reviews for these varieties by again applying our function, `top_10_words`:

```
top_10_words(
    df
    .loc[df['variety']
         .isin(df['variety']
               .value_counts()
               .head(5)
               .index),
    'description']
)
```

Solution

```
filename = '../data/winemag-150k-reviews.csv'
df = pd.read_csv(filename,
                 usecols=['country','province',
                     'description', 'variety'])

def top_10_words(s):
    common_wine_words = ['flavors', 'aromas',
            'finish', 'drink', 'palate']

    words = (
        s
        .str.lower()
        .str.split()
        .explode()
        .str.strip(',$.?!$%')
        )

    return (
        words
        .loc[(words.str.len()>=5) &
             (~words.isin(common_wine_words))]
        .value_counts()
        .head(10)
    )
```

```
top_10_words(df['description'])
top_10_words(df.loc[df['country'] ==
    'France', 'description'])

top_10_words(df.loc[df['province'] !=
    'California', 'description'])

top_10_words(
    df.loc[df['variety']
           .isin(['Chardonnay',
                  'Sauvignon Blanc',
                  'Riesling']),
           'description'])

top_10_words(
    df.loc[df['variety']
           .isin(['Pinot Noir',
                  'Cabernet Sauvignon',
                  'Syrah', 'Merlot',
                  'Zinfandel']),
          'description'])

top_10_words(
    df.loc[df['variety'] == 'Rosé',
           'description'])

top_10_words(
    df
    .loc[df['variety']
         .isin(df['variety']
               .value_counts()
               .head(5)
               .index),
    'description']
)
```

You can explore a version of this in the Pandas Tutor at http://mng.bz/aE4m.

Beyond the exercise

- Which country's wines got the highest average score?
- Create a pivot table in which the index contains countries, the columns contain varieties, and the cells contain mean scores. Include only the top 10 varieties.
- What is the correlation between the number of wines offered by a country and the mean score for that country? That is, does a country that submits more wines to competitions get, on average, a higher score than one that submits fewer wines to competitions?

xercise 38 • Programmer salaries

In the Stack Overflow survey we examined in chapter 8, developers indicated which programming languages they're currently using. Unfortunately, the languages are in a single text column separated by semicolons. In this exercise, you'll work with that data, extracting and analyzing it in a variety of ways:

1. Open the file **so_2021_survey_results.csv**, and read it into a data frame. You only need the columns `LanguageHaveWorkedWith`, `LanguageWantToWorkWith`, `Country`, and `CompTotal`.

2. Answer these questions:
    1. What are the different programming languages that developers currently use?
    2. What are the 10 programming languages most commonly used today?
    3. What are the 10 programming languages people most want to use?
    4. What languages are on both top-10 lists?
    5. What languages in the top 10 have people worked with but *don’t* want to work with in the future?
    6. What is the most popular (current) language used by people in each country?
    7. What is the mean number of languages used in the last year?
    8. What is the greatest number of languages people listed as having used in the last year?
    9. How many people chose that largest number?
    10. How many people in the survey claim salaries of $2 million or more?
3. Remove rows in which salaries are less than $2 million.

4. Turn the `LanguageHaveWorkedWith` column into “dummy” columns in `df` such that each language is its own column.
5. Determine what combination is best if you want to maximize your salary and have to choose two languages from Python, JavaScript, and Java.

Working it out

In this exercise, we look at one of the most useful and interesting parts of the Stack Overflow survey: the list of programming languages that participants marked themselves as having used in the last year. The good news is that we have rich data that can give us insights into developers from around the world. The bad news is that these languages are all in a single column of the original CSV, making the data challenging to work with. This exercise uses a number of techniques to work with such data.

To begin with, we load the Stack Overflow data by reading it all into a data frame:

```
filename = '../data/so_2021_survey_results.csv'
df = pd.read_csv(filename,
    usecols=['LanguageHaveWorkedWith',
             'LanguageWantToWorkWith',
             'Country', 'CompTotal'])
```

To reduce memory usage and allow pandas to correctly determine what type of data should be in each column, we specify which columns we want to load into the data frame.

The first question we want to answer is which programming languages programmers currently use. The answers are all in `LanguageHaveWorkedWith`, a text (string) column. However, people answering the survey could provide more than one answer—which explains why this field contains numerous subfields separated by semicolons. For example, here are five rows from the file:

```
0       C++;HTML/CSS;JavaScript;Objective-C;PHP;Swift
9                                           C++;Python
11  Bash/Shell;HTML/CSS;JavaScript;Node.js;SQL;Typ...
12                                C;C++;Java;Perl;Ruby
16                C#;HTML/CSS;Java;JavaScript;Node.js
```

Notice that in the third row, the respondent indicated so many programming languages that pandas doesn't even display all of them by default, ending the string with `...` (an elipsis).

Pandas display options

You can change the maximum width of a column displayed by pandas by setting the `display.max_colwidth` option. For example:

```
pd.set_option('display.max_colwidth', 100)
```

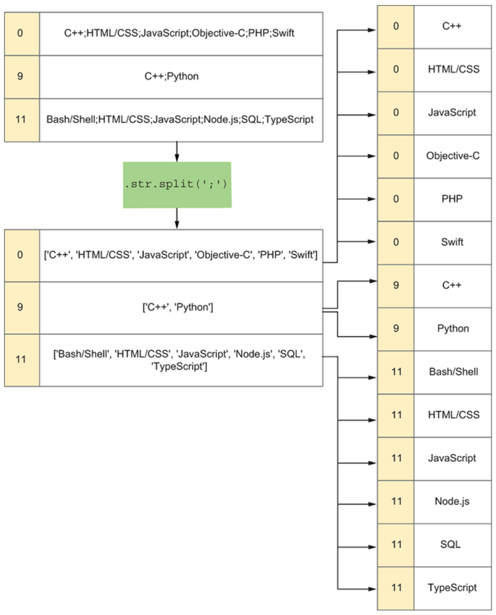
You can set it back to the original value with `pd.reset_option`:

```
pd.reset_option('display.max_colwidth')
```

Full documentation about pandas display options is at **http://mng.bz/gvYv**.

To query the data frame based on which programming language(s) people used, we need to be able to treat these strings as separate fields, not just as large strings. The best way to do that, as we saw in exercise 37, is to first run `split` on our string column (resulting in a series of Python lists) and run the `explode` method on the result (figure 9.1):

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
)
```



0
C++;HTML/CSS;JavaScript;Objective-C;PHP;Swift
9
C++;Python
11
Bash/Shell;HTML/CSS;JavaScript;Node.js;SQL;TypeScript
.str.split(';')
0
['C++', 'HTML/CSS', 'JavaScript', 'Objective-C', 'PHP', 'Swift']
9
['C++', 'Python']
11
['Bash/Shell', 'HTML/CSS', 'JavaScript', 'Node.js', 'SQL', 'TypeScript']
0 C++
0 HTML/CSS
0 JavaScript
0 Objective-C
0 PHP
0 Swift
9 C++
9 Python
11 Bash/Shell
11 HTML/CSS
11 JavaScript
11 Node.js
11 SQL
11 TypeScript

Figure 9.1 We can use `split` and `explode` to turn a series of text into a series of individual words.

The result of this query is a series of strings—all the different strings the `LanguageHaveWorkedWith` column contained. But now, each programming language is in a separate row. This allows us to count them using `value_counts`:

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
    .value_counts()
)
```

This way, we can see how many times each language was mentioned, sorted from the most popular (JavaScript) to the least popular (APL). We’re only interested in the 10 most commonly found languages, so we cut off the result after the top 10:

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
    .value_counts()
    .head(10)
)
```

We’re actually less interested in the numbers than in the names of those languages. We can thus request the index from the returned series:

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
    .value_counts()
    .head(10)
    .index
)
```

Finally, we assign that to a variable, `have_worked_with`, because we'll need these values shortly and it's easier to work with them from a variable than a long, repeated query:

```
have_worked_with = (
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
    .value_counts()
    .head(10)
    .index
)
```

Next, we perform the same query on the column `LanguageWantToWorkWith` containing the answers to the question "What language do you hope to work with in the next year?" Other than the name of the column and the variable to which we assign the results, the query is the same:

```
want_to_work_with = (
    df['LanguageWantToWorkWith']
    .str.split(';')
    .explode()
    .value_counts()
    .head(10)
    .index
)
```

Next, I asked what languages are on both top-10 lists. Because pandas index objects are similar to series, we could run the `isin` method, asking which elements of `want_to_work_with` are in `have_worked_with` and using the resulting boolean index on `want_to_work_with`:

```
(
    want_to_work_with
    .loc[want_to_work_with.isin(have_worked_with)]
)
```

But it turns out that pandas makes it easy to do this with the `intersection` method. Note that this method works on index objects and not on series:

```
want_to_work_with.intersection(have_worked_with)
```

Next, I asked you to determine which languages in the top 10 people have worked with but *don't* want to work with in the coming year. We can again use `isin` to find which elements of `have_worked_with` are in `want_to_work_with`:

```
have_worked_with.isin(want_to_work_with)
```

This returns a boolean index. We can reverse it with `~` to find which elements of `have_worked_with` are *not* in `want_to_work_with`:

```
~have_worked_with.isin(want_to_work_with)
```

Now we can apply the resulting boolean index to `have_worked_with`:

```
(
    have_worked_with
    [~have_worked_with.isin(want_to_work_with)]
)
```

And we discover that despite their current popularity, people aren't excited about working with either shell scripts or C++ in the future. (I understand and agree!)

Next, I asked you to find out which language is most popular in each country. That is, we've already found that JavaScript is the most popular programming language overall—is this universally true? Our data frame has a `Country` column, so it stands to reason that we can use `groupby` to find the most popular language per country. But there's a problem: the languages are all in the `LanguageHaveWorkedWith` column. If we use `explode` to put each language on its own row, the resulting series is a different length than `df`, meaning we cannot add it as a new column.

However, the series we get back from `explode` has the same index as the original series on which it was run. So if the original column had an index of 0 and mentioned both Python and JavaScript, the resulting series has two rows, both with an index of 0, one with Python and the other with JavaScript. This means although we cannot assign the exploded series as a column, we can use `join` to merge the series onto the data frame.

First, let's create a new series, `all_languages`, containing the programming languages. We don't need to do this, but it will make the join easier to understand:

```
all_languages = (
    df
    ['LanguageHaveWorkedWith']
    .str.split(';')
    .explode()
)
```

Then we can perform our join. Note that although `join` is a method on data frames (not series), we can pass either a data frame or a series as the argument to it. In other words, we can say

```
df.join(all_languages)
```

Actually, this code doesn't work: we get an error because the data frame that results from this join would have two columns named `LanguageHaveWorkedWith`. There are several ways to solve this problem: we could set `LanguageHaveWorkedWith.name` to a different value, or we could pass a value to one or both of the `lsuffix` or `rsuffix` parameters, adding a suffix to joined columns from the left or right and thus avoiding a clash. But I think the easiest approach is to realize that we really only care about the `Country` column in the data frame, meaning we can run `join` on it and it alone:

```
df[['Country']].join(all_languages)
```

Notice that we use double square brackets around `'Country'` to ensure that the result is a data frame rather than a series. Now that we've created this new data frame, we can use `groupby` on it:

```
(
    df[['Country']]
    .join(all_languages)
    .groupby('Country')
)
```

This gives us a `groupby` object, but now we have to apply a method. And what aggregation method should we use? The normal choices are `mean`, `count`, and `std`, but here we want the value that appears the most, often known as the *mode*. However, there isn't any `mode` method we can apply—at least, no such method is provided directly. However, we can use the method `pd.Series.mode`, applying it by passing it to the `agg` method on our `groupby` object:

```
(
    df[['Country']]
    .join(all_languages)
    .groupby('Country')
    .agg(pd.Series.mode)
)
```

The result is a one-column data frame whose index contains country names and whose values represent the most popular language in each country. We can even find the relative popularity of different languages with `value_counts`:

```
(
    df[['Country']]
    .join(all_languages)
    .groupby('Country')
    .agg(pd.Series.mode)
    .value_counts()
)
```

Next, I asked you to find the mean number of languages that developers used in the last year. What we can do is break `LanguageHaveWorkedWith` into pieces and then run `len` on that list. That gives us a series of integers on which we can run `mean`:

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .str.len()
    .mean()
)
```

Notice that we have to use the `str` accessor twice here: first to run the `split` method, turning our series of strings into a series of lists, and a second time to run `len` on each element, giving us a series of integers—on which we can run `mean`. And yes, we're using the `str` accessor to run `len` on lists; the accessor will try to run the method on whatever data it has, and because lists also support `len`, we're fine.

Next, I wanted you to determine the greatest number of languages anyone indicated they used in the last year. We can do that by running `max`:

```
(
    df['LanguageHaveWorkedWith']
    .str.split(';')
    .str.len()
    .max()
)
```

At least one person said they worked with 38 different programming languages in the last year—out of the 38 listed on the survey questionnaire. This leads me to wonder if they simply checked all the boxes. Maybe others did the same thing. I asked you to determine how many people marked that same number of languages:

```
(
    df
    .loc[df['LanguageHaveWorkedWith']
         .str.split(';')
         .str.len() == 38,
        'LanguageHaveWorkedWith']
    .count()
)
```

Here we use the length of the post-split list in a comparison, resulting in a boolean index. We apply the boolean index to the column `LanguageHaveWorkedWith` and apply `count` to find out how many rows match.

Next, I asked you to look at developer salaries as reported in the survey. First, how many developers are making more than $2 million/year?

```
(
    df
    .loc[df['CompTotal'] >= 2_000_000]
    ['CompTotal']
    .count()
)
```

Wow—2,369 people reported that kind of salary! Let’s remove them from our data, because otherwise it will be skewed:

```
df = (
    df
    .loc[df['CompTotal'] < 2_000_000]
)
```

We’ll get back to salaries in a moment. Now we take the

`LanguageHaveWorkedWith` column and turn it into multiple columns to so we can analyze the individual languages more easily. Doing this is known as creating *dummy columns*. Instead of a column containing the string `'JavaScript;Python'`, we create one column called `JavaScript` and another called `Python`, putting 1s where the person marked themselves as using JavaScript and 0s where they indicated they did not.

We can create a new data frame of dummy values based on `LanguageHaveWorkedWith` using the `str.get_dummies` method:

```
(
    df['LanguageHaveWorkedWith']
    .str.get_dummies(sep=';')
)
```

But how can we integrate this new data frame into our existing one? The answer is `pd.concat`, which we've used before. The difference is that we want to join the data frames horizontally (i.e., combining them left and right, rather than top and bottom). To tell `pd.concat` this, we need to indicate `axis='columns'`, similar to what we've done with other methods in the past, such as `df.drop`. We can then assign the result of the concatenation back to `df`:

```
df = (
    pd.concat([df,
               df['LanguageHaveWorkedWith']
               .str.get_dummies(sep=';')],
              axis='columns')
)
```

With these dummy columns in place, we can ask questions about salaries and language knowledge. First, what was the average salary of someone who knows Python and JavaScript but not Java?

```
df['CompTotal'][(df['Python'] == 1) &
                (df['JavaScript'] == 1) &
                (df['Java'] == 0)].mean()
```

We get a result of $126,817.

What about someone who knows Python and Java but not JavaScript?

```
df['CompTotal'][(df['Python'] == 1) &
                (df['JavaScript'] == 0) &
                (df['Java'] == 1)].mean()
```

Here we get a result of $162,737.

Finally, what about someone
who knows Java and JavaScript
but not Python?

```
# Java and Javascript, not Python
df['CompTotal'][(df['Python'] == 0) &
                (df['JavaScript'] == 1) &
                (df['Java'] == 1)].mean()
```

This results in $140,867.

Solution

```
filename = '../data/so_2021_survey_results.csv'
df = pd.read_csv(filename,
    usecols=['LanguageHaveWorkedWith',
             'LanguageWantToWorkWith',
             'Country', 'CompTotal'])

df['LanguageHaveWorkedWith'
    ].str.split(';').explode().value_counts().index

have_worked_with = df['LanguageHaveWorkedWith'
    ].str.split(';').explode(
    ).value_counts().head(10).index

want_to_work_with = df['LanguageWantToWorkWith'
    ].str.split(';').explode(
    ).value_counts().head(10).index

want_to_work_with.intersection(have_worked_with)
have_worked_with[~have_worked_with.isin(want_to_work_with)]

all_languages = df['LanguageHaveWorkedWith'
    ].str.split(';').explode()

df[['Country']].join(all_languages).groupby('Country'
    ).agg(pd.Series.mode)

df['LanguageHaveWorkedWith'].str.split(';').str.len().mean()
df['LanguageHaveWorkedWith'].str.split(';').str.len().max()
```

```
df['LanguageHaveWorkedWith'][
    df['LanguageHaveWorkedWith'].str.count(';') == 38].count()

(
    df
    .loc[df['CompTotal'] >= 2_000_000]
    ['CompTotal']
    .count()
)

df = (
    df
    .loc[df['CompTotal'] < 2_000_000]
)

df = (
    pd.concat(
    [df,
    df['LanguageHaveWorkedWith']
    .str.get_dummies(
        sep=';')], axis='columns')

df['CompTotal'][(df['Python'] == 1) &
                (df['JavaScript'] == 1) &
                (df['Java'] == 1)].mean()

df['CompTotal'][(df['Python'] == 1) &
                (df['JavaScript'] == 0) &
                (df['Java'] == 1)].mean()

df['CompTotal'][(df['Python'] == 0) &
                (df['JavaScript'] == 1) &
                (df['Java'] == 1)].mean()
```

You can explore a version of this in the Pandas Tutor at http://mng.bz/4JvR.

Beyond the exercise

- When developers are stuck (as indicated in the column `NEWStuck`), what are the three things they're most likely to do?
- What proportion of the survey respondents marked their gender as `Man`? Does that proportion seem similar to your real-life experiences?
- On average, what proportion of their years coding have been done professionally?

ummary

In this chapter, we looked at various ways pandas lets us work with textual data, especially via the `str` accessor. The combination of Python's rich string methods along with the various ways pandas lets us manipulate series and data frames gives us a great deal of flexibility and lets us ask a wide variety of sophisticated questions that aren't directly numerical. Many data sets, such as the ones we looked at in this chapter, contain a mix of numeric and textual data, and being able to work with the text alongside the numbers is especially useful.