3

CREATING AND MANIPULATING DATAFRAMES AND LISTS

The secret to the Excel pro's success when learning Python is the dataframe, an object provided by the pandas module. Dataframes look and feel like familiar Excel spreadsheets, but they offer a radically different way of solving problems. In fact, they're so essential to working with Excel efficiently that I've dedicated several chapters to them.

This chapter will give you an introductory overview of how they work, without lots of details. You'll import the pandas module and re-create the basic dataframe you saw in the book's introduction; meet the dataframe's close relative, the list; and learn how to perform several common operations on both dataframes and lists. In later chapters, we'll dig deeper into how and why dataframes work the way they do.

What Exactly Is a Dataframe?

Guido van Rossum created Python between 1989 and 1991 to help make programming easier. In 2008, Wes McKinney, a research and analytics associate from the world of finance, came up with an idea to make it easier still by replicating the spreadsheet experience within Python. He created a new module that you can import into your Python script called pandas to automatically link related datasets and manage these relationships, and thus the "pandas dataframe" or simply "dataframe" was born. In terms of lineage, Python is like the parent, the pandas module is the adult child, and the dataframe object is the grandchild. While a dataframe visually resembles an Excel spreadsheet with a structure composed of columns and rows, it is so much more.

Technically, a dataframe is a unique data type or object (akin to a string or an integer) created by the pandas module that is constructed from two native Python objects: series and lists. A *series object* is like a container for a single list object; the series object acts like a wrapper around that list for which each list element is assigned an index label. Each series represents a column in a dataframe and shares a common index. The *index* is the reference column on the far left of the dataframe that labels each row, or record (I use the terms *row* and *record* interchangeably throughout the book). In effect, the index acts as the spine around which all data hangs within a dataframe. A dataframe can have a single column of entries or a hundred, and the index is what holds them together.

By default, pandas assigns a numeric index called a *sequential index* or *range index*, which looks similar to row numbers in Excel. However, whereas Excel starts counting its rows from 1, a pandas index uses *zero-based indexing*, meaning it starts counting from 0. A pandas dataframe index also can be customized to include text-based labels. Index labels do not have to be unique, so you must "know your data" and remove any duplicates in your index if your downstream logic depends on unique values.

How to Create a Dataframe

As with most things Python, there are multiple ways to create a dataframe. This section will cover creating one manually, copying an existing one, and subsetting a source dataframe. A fourth approach, importing an Excel file, is covered on its own in Chapter 10.

Importing the pandas Module and Manually Creating a Dataframe

To work with dataframes, you must first *import* pandas because it, like all modules, extends Python's power but is not native functionality. Thus, the code in Listing 3-1 starts with the `import` keyword followed by the module name. If you have launched IDLE or Spyder from Anaconda, the launch package will include the necessary behind-the-scenes connections to import pandas and other popular modules.

Once imported, you can call upon the module's methods. To manually create a simple dataframe, you use the pandas `DataFrame()` method as the rest of Listing 3-1 demonstrates.

```
>>> import pandas
>>> dfPets = pandas.DataFrame({'Type' : ['Dog', 'Dog', 'Cat'],  'Name' : ['Bounce', 'Holly',
    'Ronnie']})
>>> print(dfPets)
    Type    Name
0  Dog  Bounce
1  Dog   Holly
2  Cat  Ronnie
```

Listing 3-1: Creating a simple dataframe using the `DataFrame()` method

The second line contains an assignment statement consisting of three basic steps. To start the statement, you declare the `dfPets` variable. In the middle, you write the assignment operator (`=`), which assigns whatever you do on the right to whichever variable you declared on the left. Finally, you call the `DataFrame()` method to create the new dataframe using the data you provide between the parentheses. The curly brackets (`{}`) within the parentheses define that data. In this case, you specify the column headers `Type` and `Name` followed by the list of values to populate each column within square brackets (`[]`). You'll learn more about lists later in the chapter. For more on what the different kinds of brackets mean, see the box "Bracketology" on page 41.

NOTE *When Python sees that you are working with a dataframe, it knows to expect certain pandas methods that are prepackaged to do things to and with that dataframe object. (Conversely, applying a pandas method to a non-dataframe object generates an error.) The pandas module contains all of these recipes for operations, doing the work so that you don't have to write the code yourself.*

Now that you've created your dataframe, you can see why it's easy to think of it as a virtual spreadsheet. Figure 3-1 compares how the pet data appears in an Excel spreadsheet and in a dataframe in the IDLE working environment.

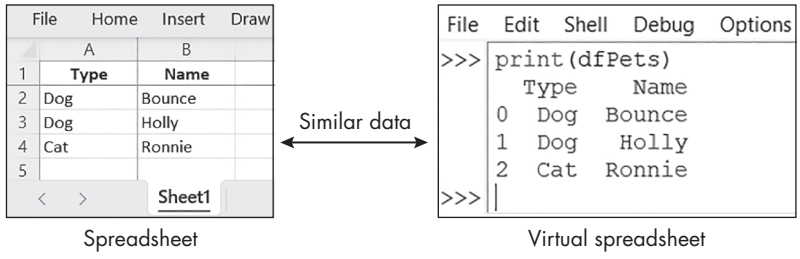

Figure 3-1: Comparing the same data in an Excel spreadsheet and a pandas dataframe

The actual spreadsheet and the virtual spreadsheet have the same row-and-column structure and share the same data. Spreadsheets and dataframes differ in some respects; for example, they label their rows in different ways (as we will explore later in this chapter). But as an Excel pro, you probably recognize how dataframes organize their information. In effect, Excel and Python are simply two ways of maintaining, viewing, and working with the same data in a table-like structure.

Before we explore the details of the code in Listing 3-1, let's clean up the formatting. Listing 3-1 writes out the full `DataFrame()` logic in one continuous line consistent with most online examples. However, I believe this `DataFrame()` method is a bit busy-looking and hard to follow without any spacing or line breaks. When I write code, I prefer to use hard returns to add some whitespace, as well as indentation to line up similar elements such as the two series in this dataframe. Following this convention can be especially helpful when you go back to read a script that you might have written months before. Python ignores the hard returns when it executes, or *compiles*, your code. The parentheses act as "begin" and "end" signals to Python, so you can insert as many hard returns after an open parenthesis as you wish, and Python will know the statement continues until it sees the corresponding close parenthesis. Listing 3-2 shows the code from Listing 3-1 after some cleanup.

```
>>> import pandas
>>> dfPets = pandas.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
>>> print(dfPets)
   Type    Name
0  Dog  Bounce
1  Dog   Holly
2  Cat  Ronnie
```

Listing 3-2: Adding formatting to clean up the code from Listing 3-1

That's much easier to read!

Note that both Listings 3-1 and 3-2 use the same `dfPets` name for their dataframes. While they share the same names and contents, these dataframes are not the same object. In fact, throughout this book I will create many dataframes called `dfPets` to demonstrate examples but they will not all share the same contents. If you create two dataframes with common names in two different scripts, they will be completely independent. If you reuse a dataframe in the same script, however, the second object will overwrite the prior one.

Listings 3-1 and 3-2 demonstrate how your code will look in IDLE, but you can alternatively use Spyder, entering your code in the Editor pane on the left and viewing the output in the Console pane on the right, as demonstrated in Figure 3-2.

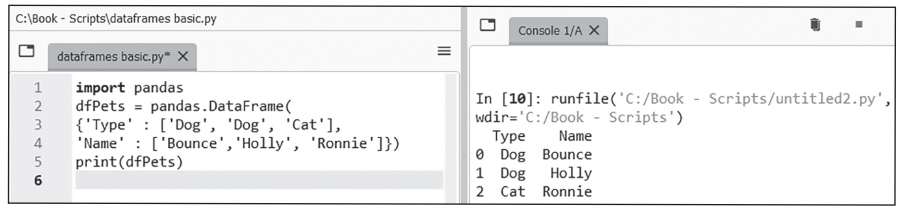

Figure 3-2: Creating and viewing a basic dataframe in Spyder

Compare how the input and the output look in the two working environments. In the listings in IDLE format, the input appears after the >>> prompt and the output appears immediately after the `print()` statement, while Spyder omits the prompt and displays the input and output side by side.

Let's take a closer look at this script and explore some ways to modify it.

Importing the pandas Module with an Accessor

The preceding examples begin with the statement `import pandas`. After importing any module, to distinguish its methods from native Python methods, you must prepend the module name to any of its methods you call. This is why you're using `pandas.DataFrame()` to call the `DataFrame()` method.

Fortunately, you can define a shortcut, known as an *accessor*, so that you don't have to type the full word *pandas* every time you call one of its methods. Following the `import` keyword and the module name, you can use the `as` keyword to declare your desired shortcut accessor as follows.

```
import pandas as pd
dfPets = pd.DataFrame({
    'Type' : ['Dog', 'Dog', 'Cat'],
    'Name' : ['Bounce', 'Holly', 'Ronnie']})
print(dfPets)
  Type    Name
0  Dog  Bounce
1  Dog   Holly
2  Cat  Ronnie
```

Now, because you've imported pandas `as pd`, you can call `pd.DataFrame()` instead of `pandas.DataFrame()` to save yourself some typing.

Naming a Dataframe

When declaring a variable, you can use any name you like so long as it doesn't start with a number or include hyphens. It's a good practice, however, to use an intuitive and standardized naming convention to remind you that the object is a dataframe. In this book, the custom is to prepend `df` to each dataframe variable, as in `dfPets` here. Using a prefix to indicate the data type is based on a convention popularized at Microsoft in the 1980s known as *Hungarian notation*. The Python convention is to use lowercase

variable names. In my Excel experience, however, I've always capitalized my column headers. Since this book is written for Excel users, I'll capitalize most variables.

Next, the script defines two series. You can write them on a single or continuous line, but I like to enter such data on separate lines to see them better. When printed, each series will display as columns sharing the same index (shown on the far left). The `Type` series contains the list `['Dog', 'Dog', 'Cat']`, and the `Name` series contains the list `['Bounce', 'Holly', 'Ronnie']`. Each pet has an associated type and name (for example, `Dog` and `Bounce`) that share a common index label (`0`).

To print the `dfPets` dataframe, you simply insert its name within the parentheses of the `print()` function, just as you passed strings of text, numbers, and variables into `print()` in the previous chapter. If you wanted, you could also pass the dataframe through the `type()` function to confirm that it's a dataframe object. As mentioned previously, printing out the object type is optional, but doing so in your early coding projects can help you keep track of what's what and troubleshoot any problems.

Passing Variables into a Dataframe

You can also pass variables into the series in lieu of fixed values, as shown in this new but similar dataframe (again, with changes highlighted in bold):

```
import pandas as pd
Type_1 = 'Dog'
Name_1 = 'Bounce'
dfPets = pd.DataFrame({
    'Type' : [Type_1, 'Dog', 'Cat'],
    'Name' : [Name_1, 'Holly', 'Ronnie']})
print(dfPets)
  Type    Name
0  Dog  Bounce
1  Dog   Holly
2  Cat  Ronnie
```

Note that you can mix variables with strings. Here you've defined the variables `Type_1` and `Name_1` and set their values to the strings `'Dog'` and `'Bounce'`, respectively, before passing them into each series. The resulting dataframe still contains the values `Dog` and `Bounce` rather than their source variable names.

Creating a Dataframe with Specific Index Labels

Instead of relying on the default pandas index, you can define your own by passing it as an argument to the `index` parameter inside the `DataFrame()` method's parentheses, as in this `dfPets2` dataframe:

```
import pandas as pd
dfPets2 = pd.DataFrame({
    'Type': ['Dog', 'Dog', 'Cat', 'Frog'],
    'Name': ['Bounce', 'Holly', 'Ronnie', 'Kermit']},
    index = ['A', 'Z', 1, 'A'])
```

```
print(dfPets2)
   Type    Name
A   Dog  Bounce
Z   Dog   Holly
1   Cat  Ronnie
A  Frog  Kermit
```

Defining `index` looks a bit different from defining the columns because an index isn't the same as an ordinary column or field. You have a fair degree of freedom when assigning index labels. As demonstrated here, you can jump from `A` to `Z`, mix strings and integers, and even repeat an index label (as in the case of Bounce the dog and Kermit the frog, which both have an index of `A`). Duplicate index labels can be useful when you're grouping related items that share a common category or attribute or if you need to select multiple rows with a single index lookup, for example. However, they can make some operations ambiguous or cause errors that are difficult to troubleshoot, so Python novices should always use unique index labels.

Copying a Dataframe

Instead of creating a new dataframe manually, you can duplicate an existing one using the `copy()` method:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
>>> dfCopy = dfPets.copy()
>>> print(dfCopy)
    Type    Name
0   Dog  Bounce
1   Dog   Holly
2   Cat  Ronnie
```

As this instance of `copy()` demonstrates, you can deploy methods without writing anything inside the parentheses. Python understands that means you want the method to operate based on its default settings.

Subsetting a Dataframe by Column

You can also subset an existing dataframe by copying specific columns or rows. The following example demonstrates how to create a new dataframe by copying a single column:

```
>>> dfNames = dfPets[['Name']].copy()
>>> print(dfNames)
     Name
0  Bounce
1   Holly
2  Ronnie
```

This new `dfNames` dataframe shares the same index as its source `dfPets`, but includes only the selected `Name` column, as indicated by the double square brackets (for more on this syntax, see "Bracketology" on page 41). Using the `copy()` method means these two dataframes will be completely independent of each other. This is but one way to subset a dataframe; many more techniques will be explored in Chapter 6.

FUNCTIONS, METHODS, AND ATTRIBUTES

Before proceeding to more detailed dataframe operations, let's define and distinguish between functions, methods, and attributes, all of which play a distinct role in Python. The following table summarizes the main differences between them.

Term	Description	Examples
Function	Operates on different object types to perform a task and return a result based on the information given to it. Most functions are native to Python. Function calls end with parentheses.	`print()`, `type()`, `len()`
Method	Belongs to a specific object type and performs operations related to that type. Methods are called in the form *`object.method`*`()` (known as *dot notation*) and often modify or provide information about the object they belong to. Many methods are associated with imported modules or libraries. Methods end with parentheses.	`pandas.DataFrame()`
Attribute	Stores information about a particular object. Attributes are also accessed by dot notation (*`object.attribute`*), but unlike methods, they're not followed by parentheses and they hold values instead of performing an action. Attributes may be followed by square brackets containing locational details.	`df.dtype`, `df.shape`, `df.loc`

These definitions are broad and inadequate for a full-time coder, but they should suffice as a starting point for further research for a Python novice.

Common Dataframe Operations

You're probably wondering how to replicate in your virtual spreadsheet the tasks you typically do in Excel with an actual spreadsheet. This section covers a sampling of operations that Python novices might enjoy using while learning to use dataframes. Figure 3-3 shows some of them.

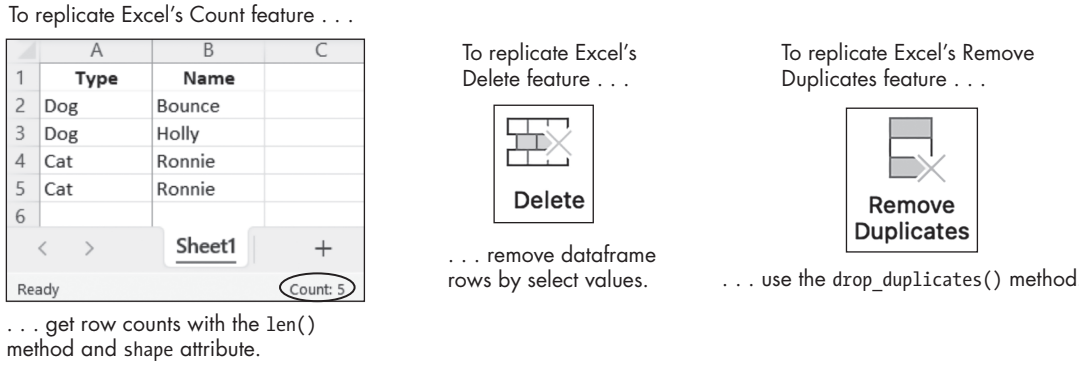

Figure 3-3: A few common Excel operations you'll learn how to replicate in Python

We'll start with the three operations shown, then cover a few more that involve working with more than one dataframe at a time.

Counting Rows Using the len() Function

You can count the number of dataframe rows using the `len()` function (that is, the *len*gth of the dataframe) as follows:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
>>> print(len(dfPets))
    3
```

Here, you use Python's built-in `len()` function directly on the dataframe and print the result immediately, but you can also use `len()` to store the number of rows in a variable:

```
>>> pet_number = len(dfPets)
>>> print(pet_number)
    3
```

Here, you're assigning the length to a variable called `pet_number` and then printing the value of that variable.

These examples demonstrate two different ways to get the same information. In the first case, you simply view the row count without saving it. In the second, you store the data in a variable that you can use again later.

Counting Rows and Columns with the shape Attribute

The `shape` attribute, one of the most important attributes to a Python novice, counts a dataframe's rows and columns, which you can then print:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
>>> print(dfPets.shape)
    (3, 2)
```

Note how the `shape` attribute returns the data within parentheses. This denotes a non-editable list-like object called a *tuple*, which has a fixed order. In the case of the `shape` attribute, the first value is always the number of rows and the second is always the number of columns.

To isolate the row and column counts so you can use them later, you could easily convert these counts into variables, as shown:

```
row_count, col_count = df.shape
```

On the left, `row_count` and `col_count` are user-defined variables (meaning you can name them anything you want) that will record the respective number of rows and columns in the dataframe to the right of the assignment operator. Here's an example:

```
>>> row_count, col_count = dfPets.shape
>>> print('Row Count = ' + str(row_count))
    Row Count = 3
>>> print('Column Count = ' + str(col_count))
    Column Count = 2
```

This code prints out the row and column counts as part of a string you've specified.

When I first encountered the `shape` attribute, I doubted I'd need to use it much. Over time, I realized that it offers powerful information. In particular, I learned to use row and column counts as quality controls to compare dataframes before and after operations such as merging (explored in Chapter 8).

Deleting Rows with a Specific Value

Deleting a dataframe row (or record) with a particular value works very similarly to subsetting, but you apply the technique to an existing dataframe rather than creating a new, smaller one:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
```

```
>>> dfPets = dfPets[dfPets['Name'] != 'Ronnie']
>>> print(dfPets)
      Type    Name
    0  Dog  Bounce
    1  Dog   Holly
```

This code keeps the records for which `Name` is *not equal to* (indicated by the `!=` characters) `Ronnie`. In other words, it deletes the record for Ronnie the cat.

Note that Python does not give you any warning before overwriting a dataframe with a new version based on your script, so proceed with caution.

Identifying and Dropping Duplicated Rows

The pandas module offers a terrific tool for isolating records in which all or some of the field values are duplicated: the `drop_duplicates()` method.

Consider the following example, where the data for Ronnie the cat is duplicated in the source dataframe but removed from a new, subsetted dataframe:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie', 'Ronnie']})
>>> print(dfPets)
      Type    Name
    0  Dog  Bounce
    1  Dog   Holly
    2  Cat  Ronnie
    3  Cat  Ronnie
>>> dfPets_New = dfPets.drop_duplicates()
>>> print(dfPets_New)
      Type    Name
    0  Dog  Bounce
    1  Dog   Holly
    2  Cat  Ronnie
```

The `dfPets` source dataframe maintains two Ronnie the cat records, while the `dfPets_New` subset dataframe contains just one. Alternatively, you could overwrite the original `dfPets` instead of returning a new dataframe by simply running `dfPets = dfPets.drop_duplicates()`.

Concatenating Dataframes

You can concatenate dataframes with the `concat()` method. This method allows you to stack dataframes on top of each other (concatenating by rows) or lay them side by side (concatenating by columns), depending on the arguments you choose.

Concatenation by Row (axis = 0), Same Columns

Consider the case of two dataframes sharing identical columns:

```
>>> import pandas as pd
>>> dfDog = pd.DataFrame({
...     'Type': ['Dog', 'Dog'],
...     'Name': ['Bounce', 'Holly']})
>>> print(dfDog)
     Type    Name
   0  Dog  Bounce
   1  Dog   Holly
>>> dfCat = pd.DataFrame({
...     'Type': ['Cat'],
...     'Name': ['Ronnie']})
>>> print(dfCat)
     Type    Name
   0  Cat  Ronnie
```

To concatenate these two dataframes by rows, or "stack" them together vertically, you pass their names as a list into the `concat()` method, as shown in Listing 3-3.

```
>>> dfPets = pd.concat([dfDog, dfCat], axis = 0)
>>> print(dfPets)
     Type    Name
   0  Dog  Bounce
   1  Dog   Holly
   0  Cat  Ronnie
```

Listing 3-3: Concatenating two dataframes with identical columns by using an explicit `axis = 0` argument

Even though `axis = 0` is the default argument, I recommend always defining it explicitly as shown here. Doing so will help you think in terms of *axes* (that is, rows or columns, stacking vertically or horizontally), which will serve you well as you learn more about pandas and encounter axes in different contexts. That said, omitting the `axis` parameter and simply relying upon the default value produces the same result, as Listing 3-4 demonstrates.

```
>>> dfPets = pd.concat([dfDog, dfCat])
>>> print(dfPets)
     Type    Name
   0  Dog  Bounce
   1  Dog   Holly
   0  Cat  Ronnie
```

Listing 3-4: Concatenating two dataframes with identical columns by relying upon the default `axis` value

You're not limited to concatenating just two dataframes, as you can see in Listing 3-5.

```
>>> dfFrog = pd.DataFrame({
...     'Type': ['Frog'],
...     'Name': ['Kermit']})
>>> print(dfFrog)
    Type    Name
0  Frog  Kermit
>>> dfPets_Plus = pd.concat([dfDog, dfCat, dfFrog], axis = 0)
>>> print(dfPets_Plus)
    Type    Name
0   Dog  Bounce
1   Dog   Holly
0   Cat  Ronnie
0  Frog  Kermit
```

Listing 3-5: Concatenating multiple dataframes

This example defines a third dataframe with the same columns, `dfFrog`, and then concatenates all three pet dataframes into a new one called `dfPets_Plus`. You simply add the `dfFrog` dataframe within the list passed to the `concat()` method. The rows in `dfPets_Plus` are organized based on the order in which you listed each dataframe.

There are a few more points to note about these examples:

- The dataframes to concatenate are always passed in as a list, so they must be enclosed within square brackets (`[]`) and separated by commas. If you forget the brackets, your code will crash. (You'll learn more about lists in the next section; also see "Bracketology" on page 41.)
- All of the dataframes in these examples have the same number of columns, and their column labels are spelled and even capitalized the same way. In the next section, you'll see some examples where the source dataframes have different columns.
- The indexes retain their original values in the returned dataframe, which means that the concatenated dataframe can contain non-unique values. In Listing 3-5, Bounce the dog, Ronnie the cat, and Kermit the frog all have an index label of `0` in `dfPets_Plus`. This is perfectly allowable in pandas. It's up to you whether to leave them as is or change the index. My personal preference is to update the index after every concatenation. For more details on manipulating index labels, see "Working with Dataframe Indexes" on page 109.

Concatenation by Row (axis = 0), Different Columns

When you concatenate dataframes with different columns, pandas will populate the returned dataframe with a special object that looks like Excel's #N/A error message to fill in any resulting gaps, as shown in Listing 3-6.

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Name' : ['Bounce', 'Holly', 'Ronnie'],
...     'Type' : ['Dog', 'Dog', 'Cat']})
>>> print(dfPets)
         Name Type
    0  Bounce  Dog
    1   Holly  Dog
    2  Ronnie  Cat
>>> dfTurtle = pd.DataFrame({
...     'Name'    : ['Bob'],
...     'Type'    : ['Turtle'],
...     'Species' : ['Reptile']})
>>> print(dfTurtle)
      Name    Type  Species
    0  Bob  Turtle  Reptile
>>> dfPets_w_Species = pd.concat([dfPets, dfTurtle], axis = 0)
>>> print(dfPets_w_Species)
         Name    Type  Species
    0  Bounce     Dog      NaN
    1   Holly     Dog      NaN
    2  Ronnie     Cat      NaN
    0     Bob  Turtle  Reptile
```

Listing 3-6: Concatenating dataframes with different columns

In this example, `dfTurtle` has a `Species` column that `dfPets` does not. As a result, in the returned dataframe, the shared `Name` and `Type` columns look nice, but pandas must populate the `Species` column with what are called *NaN* objects for the records from the `dfPets` dataframe. Chapter 5 will fully cover how to work with such NaN objects.

Concatenation by Column (axis = 1), Same Index Values

To concatenate dataframes by column (that is, to place them side by side), you simply add the argument `axis = 1`. In most cases, you'll be concatenating dataframes that share the same index labels but have different columns, as in Listing 3-7.

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Name' : ['Bounce', 'Holly', 'Ronnie'],
...     'Type' : ['Dog', 'Dog', 'Cat']},
...     index = ['A', 'B', 'C'])
```

```
>>> print(dfPets)
        Name Type
   A  Bounce  Dog
   B   Holly  Dog
   C  Ronnie  Cat
>>> dfWeights = pd.DataFrame({'Pounds' : [8, 10, 9]},
...     index = ['A', 'B', 'C'])
>>> print(dfWeights)
      Pounds
   A       8
   B      10
   C       9
>>> dfPets_w_Weight = pd.concat([dfPets, dfWeights], axis = 1)
>>> print(dfPets_w_Weight)
        Name Type  Pounds
   A  Bounce  Dog       8
   B   Holly  Dog      10
   C  Ronnie  Cat       9
```

Listing 3-7: Concatenating dataframes with the same index labels by column

This script updates the `dfPets` dataframe with the weights of each pet. Because the index labels in `dfWeights` directly correspond to those in `dfPets`, the returned `dfPets_w_Weight` dataframe looks clean; the index remains the same, but the `Pounds` column now appears on the right side. (If you'd wanted `Pounds` to be the leftmost column, you would have specified `dfWeights` first in the list you passed to the `concat()` method.)

In this example, you've created a new dataframe, but you could just as easily have overwritten `dfPets` to add the supplemental column by writing `dfPets_w_Weight = pd.concat([dfPets, dfWeights], axis = 1)`.

Concatenation by Column (axis = 1), Different Index Values

Just as it does when concatenating dataframes with different columns, pandas populates the returned dataframe with NaN objects when concatenating dataframes with different index labels, as demonstrated in Listing 3-8.

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Name' : ['Bounce', 'Holly', 'Ronnie'],
...     'Type' : ['Dog', 'Dog', 'Cat']},
...     index = ['A', 'B', 'C'])
>>> print(dfPets)
        Name Type
   A  Bounce  Dog
   B   Holly  Dog
   C  Ronnie  Cat
>>> dfWeights_New = pd.DataFrame({'Pounds' : [77]},
...     index = ['D'])
```

```
>>> print(dfWeights_New)
   Pounds
D      77
>>> dfPets_w_Weight_New = pd.concat([dfPets, dfWeights_New], axis = 1)
>>> print(dfPets_w_Weight_New)
     Name Type  Pounds
A  Bounce  Dog     NaN
B   Holly  Dog     NaN
C  Ronnie  Cat     NaN
D     NaN  NaN    77.0
```

Listing 3-8: Concatenating dataframes with different index labels by column

The record in `dfWeights_New` has a record with the index label `D`, which does not appear in `dfPets`. Likewise, `dfWeights_New` contains neither a `Name` nor a `Type` column. Thus, in the concatenated dataframe, pandas must populate the `Name` and `Type` columns with NaN objects for index `D` and must populate the `Pounds` column with NaN objects for indexes `A`, `B`, and `C`. Know your data if you want to avoid unexpected NaN objects!

Lists: The DNA of Dataframes

Before diving deeper into dataframes, I want to introduce their ancestor—the list object—because it will help you both understand and manipulate virtual spreadsheets. Just as importantly, list objects can play a flexible, important role in code design. It is worth the Python novice's time to know about them in their own right.

At its heart, a *list object* (or simply *list*) in Python is exactly what it sounds like: a container for a group of items. This object is native to Python, which means it existed well before the invention of pandas, but it constitutes a key part of dataframe DNA.

Some key points about lists include the following:

- A list is a single-dimensional set of values enclosed by square brackets (`[]`).
- Unlike tuples, the list-like objects you encountered earlier with the `shape` attribute, lists are editable.
- If you add an index to a list, you get a series. As previously mentioned, a series object is like a single-column dataframe with an index, an optional name, and values.
- You can create an empty dataframe by simply defining a list of column labels. As you'll see in Chapter 4, you can also use lists to permanently rearrange column order and to temporarily print select fields in whatever order you desire.

BRACKETOLOGY

Different kinds of brackets tell Python and pandas what kind of data they're working with and what kind of objects or methods to expect:

Parentheses, ()

Parentheses appear most commonly at the end of method calls. You can add arguments within the parentheses or you can leave them empty, which tells Python to use the default arguments. As mentioned earlier, parentheses are also used to define a tuple, but tuples won't be covered further in this book.

Square brackets, []

Square brackets are used to contain lists, to identify a column or columns in a dataframe, or, as you'll read about in Chapter 5, to define a cell or range using the `loc[]` and `iloc[]` attributes.

Double square brackets, [[ ]]

As Chapter 6 will discuss, you'll often see double square brackets when subsetting a dataframe. Each pair of square brackets plays a different role. In the code `dfSubset = dfSource[['Name', 'Type']]`, the outer pair of square brackets says, "Place a list of columns inside." The inner pair of brackets actually contains the list of column labels.

Curly brackets, {}

Also known as braces, curly brackets are used for defining dictionaries, a Python object type that stores data in pairs of keys (similar to indexes) and values. You've also seen curly brackets used in some `DataFrame()` examples to define the series of data to be entered. (At a fundamental level, dataframes actually contain dictionaries of series that share a common index.) We'll talk more about dictionaries in Chapter 6.

List objects are native to Python, so you don't need to import any additional modules to use them. Early coders relied on lists, but had to do an enormous amount of work wrangling them to produce complex results. Excel does this work behind the scenes, and making these tasks easier with Python was a major motivation behind the invention of pandas.

Creating a List

To create a list, define a variable as the list name and then pass the list items between square brackets as follows:

```
>>> Pet_List = ['Bounce', 'Holly', 'Ronnie']
>>> print(Pet_List)
    ['Bounce', 'Holly', 'Ronnie']
>>> print(type(Pet_List))
    <class 'list'>
```

As with most objects in Python, you can name a list almost anything you want. It's helpful, however, to use the word *list* in the name to remind yourself what it is. When you enclose elements within square brackets, Python recognizes that you are creating a list object. Likewise, when you print the list, Python reminds you that you are dealing with a list data type by displaying the list items within square brackets.

NOTE *As you've seen before, this code includes the* `type()` *function and prints the data type for educational purposes only; you don't have to add this line in your script for it to work.*

You can also analyze a list, such as by using the `len()` function to count the number of elements the list contains:

```
>>> print(len(Pet_List))
    3
```

As demonstrated earlier in this chapter with row counts for dataframes, you can also store the list count in a variable by writing something like `list_count = len(Pet_List)`.

Isolating Unique List Values

If you want to drop duplicate values from a list so that it contains only unique elements, convert the data into a *set*, which includes unordered elements with no duplicates, using Python's native `set()` method:

```
>>> Animals_list = ['Dog', 'Dog', 'Cat']
>>> Animals_set_unique = set(Animals_list)
>>> print(Animals_set_unique)
    {'Dog', 'Cat'}
>>> print(type(Animals_set_unique))
    <class 'set'>
>>> Animals_list_unique = list(Animals_set_unique)
>>> print(Animals_list_unique)
    ['Dog', 'Cat']
>>> print(type(Animals_list_unique))
    <class 'list'>
```

The `set()` function returns the unique values from the list in the form of yet another data type called a set. As shown here, you can then pass that

set into the `list()` function if you want to return to the same data type with which you started.

Appending a Single Item to a List

To add a single item to a list, apply the native Python `append()` function like so:

```
>>> Name_List = ['Bounce', 'Holly', 'Ronnie']
>>> Name_List.append('Kermit')
>>> print(Name_List)
    ['Bounce', 'Holly', 'Ronnie', 'Kermit']
```

The resulting object now includes the `Kermit` item at the end of the list.

To place a new item at a particular location in the list, use the `insert()` function as follows (don't forget to apply zero-based indexing when specifying the insertion location):

```
>>> Name_List = ['Bounce', 'Holly', 'Ronnie']
>>> Name_List.insert(1,'Kermit')
>>> print(Name_List)
    ['Bounce', 'Kermit', 'Holly', 'Ronnie']
```

Passing index `1` and the list item `Kermit` to the `insert()` function adds `Kermit` to `Name_List` in the second position.

Adding a List to a List

To concatenate a list to another list, use the `extend()` function:

```
>>> Name_List = ['Bounce', 'Holly', 'Ronnie']
>>> Add_List = ['Scruffy', 'Barney']
>>> Name_List.extend(Add_List)
>>> print(Name_List)
    ['Bounce', 'Holly', 'Ronnie', 'Scruffy', 'Barney']
```

With the `extend()` method, the list items from the second list (`Add_List` in this example) will always be added to the end of the first list (`Name_List`).

Sorting a List

Use the `sorted()` function to sort a list in alphabetical or numerical order:

```
>>> Pets = ['Dog', 'Turtle', 'Cat']
>>> Pets = sorted(Pets)
>>> print(Pets)
    ['Cat', 'Dog', 'Turtle']
>>> List_Numbers = [1, 5.0, -4]
>>> List_Numbers = sorted(List_Numbers)
>>> print(List_Numbers)
    [-4, 1, 5.0]
```

Keep in mind that `sorted()` requires that the list contain compatible data types. In this example, you can successfully sort a mixture of integers

and floats, but if you'd tried to sort a list with strings and numbers like `['A', 1, 'B']`, you would get an error.

Identifying Minimum, Maximum, and Mean List Values

It's easy to pull the minimum or maximum value from a list using Python's native `min()` and `max()` functions, as shown here:

```
>>> List_Numbers = [1, 5.0, -4]
>>> Minimum_Value = min(List_Numbers)
>>> print(Minimum_Value)
    -4
>>> Maximum_Value = max(List_Numbers)
>>> print(Maximum_Value)
    5.0
```

To calculate a mean from a list, however, you'll need to import the NumPy library (and assigning an accessor as `np`) in order to use its `mean()` function:

```
>>> import numpy as np
>>> List_Numbers = [1, 5, -4]
>>> Mean = np.mean(List_Numbers)
>>> print('Mean = ' + str(Mean))
    Mean = 0.6666666666666666
```

If you don't need to return a result with the default 16-point accuracy shown here, see Chapter 4 to learn how to round float objects.

Removing a List Element

To drop a list item, the `remove()` function will remove the first occurrence of that item in the list:

```
>>> List = ['Dog', 'Turtle', 'Cat']
>>> List.remove('Turtle')
>>> print(List)
    ['Dog', 'Cat']
```

The resulting list no longer contains the `Turtle` element. There is no single function to remove all occurrences of a value; you need to construct logic, such as a loop statement, to accomplish that.

Comparing Lists

You can use logic statements to compare lists using the equality operator (`==`) as follows:

```
>>> List_1 = ['Dog', 'Turtle', 'Cat']
>>> List_2 = ['Dog', 'Turtle', 'Frog']
>>> Answer = (List_1 == List_2)
>>> print('Is List_1 equal to List_2? ' + str(Answer))
    Is List_1 equal to List_2? False
```

```
>>> List_1 = ['Dog', 'Turtle', 'Cat']
>>> List_3 = ['Cat', 'Dog', 'Turtle']
>>> Answer = (List_1 == List_3)
>>> print('Is List_1 equal to List_3 ' + str(Answer))
    Is List_1 equal to List_3? False
```

The == operator will check if two lists are equal and return a *Boolean* result, meaning a `True` or `False` value. To concatenate a Boolean type into the `print()` statement, you must convert it into a string with the `str()` function.

NOTE *Chapter 4 will discuss making comparisons using logical operators such as == in more depth.*

Notice that Python cares about order: Two lists with the same items but in a different order, such as Lists 1 and 3, will return `False` when compared.

Creating an Empty Dataframe with a List

Now that you've worked a bit with list objects, you probably recognize that you've been using them throughout the chapter when manually creating a dataframe using the `DataFrame()` method:

```
>>> import pandas
>>> dfPets = pandas.DataFrame({
...     'Type' : ['Dog', 'Dog', 'Cat'],
...     'Name' : ['Bounce', 'Holly', 'Ronnie']})
```

The values for the `Type` and `Name` columns are both contained in lists. This structure supports the notion that a column is a series object based on a list object. As a reminder, a dataframe object collects one or more series objects that share a common index.

Since lists are the building blocks of dataframes, you can use them in different ways when creating dataframes. One special case is creating an empty dataframe by defining only column labels without values. This technique is particularly valuable in workflow design where you need an empty dataframe to which you'll eventually post results. To do this, use the `columns` parameter, a feature within the `DataFrame()` method, as follows:

```
>>> import pandas as pd
>>> Column_List = ['X', 'Y', 'Z']
>>> dfNew = pd.DataFrame(columns = Column_List)
>>> print(dfNew)
    Empty DataFrame
    Columns: [X, Y, Z]
    Index: []
```

Begin by defining the list of column labels (here, it's named `Column_List`), then pass this list as an argument to the `DataFrame()` method's `columns` parameter.

Passing a list of three column labels without their associated values creates a dataframe with three empty series. When you print the new

dataframe, Python reports that this is an empty dataframe but displays the column labels.

Converting a Column into a List

When working with dataframes, you'll often need to export a column's values to a list. Use the `tolist()` method as follows:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Name' : ['Bounce', 'Holly', 'Ronnie'],
...     'Type' : ['Dog', 'Dog', 'Cat']})
>>> Name_List = dfPets['Name'].tolist()
>>> print(Name_List)
    ['Bounce', 'Holly', 'Ronnie']
```

Notice how the list of names looks the same when you use it to populate the new dataframe and then view the new dataframe as an exported list; the same square brackets contain the same data.

Next, I'll show you how to create one of its primary components, which we've touched on several times: the series.

Another Fragment of Dataframe DNA: The Series Object

If a list is the fundamental building block of the dataframe DNA, a series object (or just series) is a more complex link in that chain. Like dataframes, series are not native to Python but are the creatures of the pandas module. While dataframes are the easiest way for Python novices to meet and work with series, you should be familiar with what the `Series()` method looks like because you'll likely encounter it (even if only by mistake) during your first forays into coding.

The `Series()` method converts a list of elements into a series:

```
>>> import pandas as pd
>>> Pet_Series = pd.Series(['Dog', 'Dog', 'Cat'])
>>> print(Pet_Series)
    0    Dog
    1    Dog
    2    Cat
    dtype: object
>>> print(type(Pet_Series))
    <class 'pandas.core.series.Series'>
```

Note that `Series()` is among the rare methods with a capitalized first letter.

By default, the `Series()` method returns a numeric index starting with a 0 (since Python uses zero-based indexing). You can customize index labels by assigning the `index` parameter a list of values:

```
>>> import pandas as pd
>>> Pet_Series = pd.Series(['Dog', 'Dog', 'Cat'],
...                        index = ['D-1', 'D-1', 'C-1'])
```

```
>>> print(Pet_Series)
    D-1    Dog
    D-1    Dog
    C-1    Cat
    dtype: object
```

Like indexes in a dataframe, the index of a series object may contain duplicate, non-unique values. If the number of items in the data and index lists doesn't match, Python returns an error message.

Finally, note that a single dataframe column is a series object:

```
>>> import pandas as pd
>>> dfPets = pd.DataFrame({
...     'Name' : ['Bounce', 'Holly', 'Ronnie'],
...     'Type' : ['Dog', 'Dog', 'Cat']})
>>> print(dfPets['Name'])
    0    Bounce
    1     Holly
    2    Ronnie
    Name: Name, dtype: object
>>> print(type(dfPets['Name']))
    <class 'pandas.core.series.Series'>
```

This example reinforces how dataframes are constructed of different object types. Printing the variable's type is never required for your code to work, but it's a helpful practice when you're learning. It can also be a useful debugging technique if a method looks perfectly fine, yet still does not work as you expect.

Summary

In this chapter, you learned about common operations on dataframes and lists, such as creating them, concatenating them, and adding or removing items from them. This chapter has answered the following questions:

What is the difference between Python and pandas?

Python is the primary language, whereas pandas is a module added on to Python.

What is the technical definition of a dataframe?

A dataframe is a class of object comprising rows and columns, akin to a virtual Excel spreadsheet.

Are the various kinds of brackets interchangeable?

No. Python will interpret parentheses, `()`, differently than square brackets, `[]`.

Do I have to import a module like pandas for every dataframe script?

Yes. Dataframes are not native to Python.

How do I create a dataframe?

There are multiple ways to create a dataframe, including using the `DataFrame()` method, copying or subsetting a dataframe, and importing an Excel file.

What is the difference between a method, a function, and an attribute?

A method is specially designed for a particular object type, while a function operates with multiple data types. Methods and functions can change an object or produce a new one, while an attribute simply describes the object.

What is an easy way to count the number of rows and columns in a dataframe?

The `shape` attribute is a helpful way to return the dimensions of a dataframe.

How do I concatenate dataframes together?

The `concat()` method concatenates, or combines, a list of two or more dataframes.

How do I stack dataframes on top of each other, using the rows "in bulk"?

Pass the `axis = 0` argument into the `concat()` method, but watch for non-matching column labels, and make adjustments as needed.

How do I concatenate the columns from two dataframes side by side?

Pass the `axis = 1` argument into the `concat()` method, but watch for non-matching index labels, and make adjustments as needed.

What is a list in Python?

A list is a native Python object that contains elements within square brackets. The element may be a string, number, or another object type.

What is the difference between a list and a dataframe?

In Python, a list is an object that contains one or more values with no index. A dataframe is a collection of lists organized in columns that share an index.

How do I convert a column of values to a list?

The `tolist()` function exports column contents into a list object.

In the next chapter, you'll start digging in further by learning how to work with dataframe columns and the data they contain.