# Introducing Data as Data: The Series

Whenever we are talking about data herein, we are talking about regular or systematic measurements of phenomena. Anything can serve as data, if you are keen to observe it and can measure it in some reliable way. Social scientists have been coming up with all kinds of creative ways to measure phenomena. For example, to measure bias in hiring, Bertand and Mullainathain used call backs to resumes where the resumes were pretty much similar except in the use of racial or ethnically distinctive names (xx). To measure political engagement during a protest, Gonzalez-Bailon et al., measured frequency of tweets mentioning keywords (xx). Whatever it is we measure we will want to consider what is the unit of analysis. And if we have a single unit, we often then have multiples of that unit. We compare tweets, edits to Wikipedia, clicks on a page, times a light was turned on, etc. We can think of each unit as an object. What we first want to do is create a collection of objects.

There are many different kinds of collections in Python. Different collections have different features and syntax. So one kind of collection, such as a list, might be *indexed*, which means that you can retrieve members of the collection with a sequential index. In Python the first index is zero. The list ll = ["alpha", "bravo", "charlie"] is indexed such that ll[0] returns "alpha". Other collections are *keyed* meaning that they use key-value pairs. The keys don’t need to be in any specific order. A dictionary dd = {"sun":"warm", "cloud":"cool"} will be keyed such that dd["cloud"] will return "cool".

As we progress through the book, you will encounter increasingly complex combinations of indexed and keyed collections. In fact, a very large part of data science programming is knowing how to effectively use the right kind of collection for a task.

A simple example of a collection could be a range of numbers. Say, from 0 to 10. We would literally use a range object, such as range(10) for the first ten integers starting with , or range(1,20,4) for numbers starting from up to , going at a time. A more complex example of a collection could be a comment tree for a *reddit* post. This comment tree will be ordered in a different way from a range of numbers. With a module called praw which we will discuss in depth later, you can download the top level comments for a reddit post, then capture all the comments underneath, and so forth, while preserving the comment structure.

When we want to learn about a phenomenon we usually need to transform data from one structure where we collect or measure something into another structure that can allow us to gather insights and make claims. For example, we might take a set of survey responses and transform them into a table so we can get a sense of how strongly people feel about an issue, or learn which subsets of people feel more or less strongly about the issue. We might have a set of speeches as text files, but we will want to make claims about freuqencies of words or features of the speeches, like how many different words were used. That means calculating some measures, such as lexical diversity and putting them in a table for comparison.

The process of transforming data is called **“data wrangling”** and it is the most pervasive part of data science. Some researchers will use very sophisticated machine learning on texts, others will use data visualisations to discover and communicate insights, while others will just merge and filter data to make comparisons in tables. But everybody will wrangle data.

When wrangling, lists and dictionaries are alright, but they lack some useful features. For example, wouldn’t it be nice to have a collection that both has an index, so that you can count to the element and is keyed so that you can just ask for element by name? In Python, such a collection is a part of the pandas library. In fact, there are two such collections that can be indexed and keyed, and we will be seeing a lot of them. The first is the Series. The Series has an index that you can set and it has an order fron to . Importantly, a Series only has one one dimension. It is like a single list. Typically, we want at least two dimensions, like we would have in a spreadsheet. Not just a list of case IDs, but for each case ID we would want to know the age, location, number of followers, frequency of edits, date of last login, etc. So with a DataFrame, we treat each case as a row and each feature we want to measure as a column.

Below we will first introduce the features of the Series and then we will introduce the DataFrame. Then in the next chapter we will introduce some common data structures found on the web and show how to transform them into DataFrames. Then in every chapter that follows we will use DataFrames in some fashion to pose and answer questions about data.

Before we get started you might be asking, can *everything* be done in these dataframes? No, not everything that we wish to do with programming is best done in DataFrames. Later on throughout the book, we will be using different kinds of objects when they are fit for purpose. Despite this, when we want to make a scientific claim, we will want to extract data from these different kinds of objects and usually wrangle it into a DataFrame to make some sort of comparison. DataFrames enable us to use statistical measures to pose questions like “do we see more or less of something than we would expect”, “does something change over time”, or “what different things seem to consistently go together”? So although not everything can be done in DataFrames, most of what you will want to do will involve them at some point.

Before we get to the DataFrame, however, it makes sense to start with the Series. Then we will see how DataFrames are really just collections of Series objects. Consequently, to effectively get data into a DataFrame, to get data out a DataFrame, to filter that data, or to merge it with another table of data, you will often need to use a Series.

# The Series

The Series is like a list but the index can be labeled and you can give the Series a name. A Series is a class of object in Python within the pandas library. Therefore you can import and then create an empty Series in two ways:

from pandas import Series   
ser1 = Series()

or ~~~ python import pandas as pd ser1 = pd.Series() ~~~

The former is best when the Series is the only thing you want to import from pandas. However, in most cases we want to import the series, the DataFrame and maybe some helper methods, so in my code I tend to use the second approach. In case you didn’t see this before, as is a way to give a library a different, typically shorter name. So in this book you will see it used here, import pandas as pd and later, for example, I use it in import beautifulsoup4 as bs4.

The empty series with a default index is not very useful on its own. So we will instead create a Series with some data. Let’s start with the days of the week. A list of these days would be:

lweekdays = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

To turn this into a series we could write:

sweekdays = pd.Series(lweekdays,name="Weekdays")  
  
display(sweekdays)  
  
> 0 Monday  
> 1 Tuesday  
> 2 Wednesday  
> 3 Thursday  
> 4 Friday  
> 5 Saturday  
> 6 Sunday

This will transform the list into a Series with the default name (None) and the default index (in this case 0 through 6, since we have 7 elements).

However, now imagine that instead of simply listing the days of the week, we want a list where we have some *measurements*, based on days. We could count something per day, like the “hours of sleep that night”. So we could use a dictionary with the keys as days and the number of hours slept that night as the values.

dsleephours = {"Sunday":8,  
 "Monday":7,  
 "Tuesday":5,  
 "Wednesday":6,  
 "Thursday":8,  
 "Friday":9,  
 "Saturday":8}

Now we can create a Series from that dictionary, much in the way we did with a list. Except this time, the keys will be the indices rather than the numbers 0 through 6. Observe:

sleephours = pd.Series(dsleephours)  
  
display(sleephours)   
  
> Sunday 8  
> Monday 7  
> Tuesday 5  
> Wednesday 6  
> Thursday 8  
> Friday 9  
> Saturday 8

With this series we can now start to explore data. There are three ways in which we tend to extract data from a Series: by index, by value, as a distribution. See a description of each below.

## Working from index

Working from index means that we will start with an index and get a value. For example, we might want to know if we got 8 hours of sleep on Tuesday. To work from index means we get data by querying the series based on the index. Observe:

display(sleephours["Tuesday"])  
> 5  
  
# OR  
  
display(sleephours[2])  
> 5

It seems that Tuesday night was a rough night, as the data shows only 5 hours of sleep.

What happens if the index itself is comprised of integers? In that case, it will function like a label. See the notebook for examples of this gotcha. But as a rule, try to avoid using integer numbers as indices unless they are sequential and start from 0. If you absolutely must use a number, try using a string version. “0” is a string, whereas is a number.

## Working from values (and Slicing)

Working from values means that we will start with a value or set of values and discover the related indices. This is typically how we slice and filter data. For example, we might want to filter a series of Twitter accounts down to those who have been reported as bots. In this case, we would have a series with the Twitter account name as the index and the value of True or False for is\_bot as the values of the Series.

In our case, we have a series with hours of sleep as the values. We could then ask what night entailed greater than 7 hours of sleep. By using a boolean operator (which evaluates to True or False) we then get a new Series with the result of that query for each night. It has the same indices but the values are now True or False for whether the value greater than 7.

display(sleephours > 7)  
  
> Sunday True  
> Monday False  
> Tuesday False  
> Wednesday False  
> Thursday True  
> Friday True  
> Saturday True

This series shows that on Monday, Tuesday, and Wednesday we observed 7 or less hours of sleep.

Where this sort of Boolean logic is most useful is in slicing a Series or a DataFrame (it’s the same principle in both, but we will go over this again with DataFrames).

So we discovered that we can query by index with Series[index], and we discovered that we can create a new True/False series with a Boolean operator. What if we put the Boolean operator inside the query? Then we get a *slice*. So now instead of just asking for whether each day had 8 hours of sleep or more, we can query for *which* days had 8 or more hours. What we are doing here is SERIES[ [SERIES == TRUTH\_CONDITION ] ] to filter down the original series. Observe:

display(sleephours[sleephours >= 8])  
   
> Sunday True  
> Thursday True  
> Friday True  
> Saturday True  
> Name: days\_with\_s, dtype: bool

Building up this chain further, we can ask how many days are in this new slice with len() (it’s short for length). So if we want to know what proportion of days we observed the subject having 8 or more hours of sleep it could look like this:

days\_sleep = len(sleephours[sleephours >= 8])  
total\_days = len(sleephours)  
display(days\_sleep / total\_days)   
> 0.5714285714285714

* NOTE: I really don’t recommend printing the full number as displayed in research. Instead, represent the number to a meaningful scale. In this case, perhaps would be useful. In Chapter xx we will look at how to render numbers for display as part of presenting data.

## Working from distributions

Working from distributions means that we will try to summarize the values in some way. A key distinction here is in the type of data, and particular, whether the data is numerical or not. If the data in the series is numerical, we can produce numerous statistical summaries of the data, such as the mean, median, mode, skewness. If the data is non-numerical, most of what we can do with a distribution is get the max, min, mode and use a command to create a table of values.

With the original data on hours of sleep per night we might want to get a sense of how many days we had 5,6,7,8 or more hours of sleep. Alternatively, we may want to summarise the number of days with > 7 hours. To summarise a series by counting the number of unique entries we would use the value\_counts() method.

display(sleephours.value\_counts())  
  
> 8 3  
> 7 1  
> 6 1  
> 5 1  
> 9 1  
   
(sleephours > 7).value\_counts()  
  
> True 4  
> False 3

Notice that in the second case, we used the table of boolean values for sleephours > 7 and then summarised this in a value\_counts() table. Beyond value counts are a huge number of possible statistical routines. One obvious one would be mean() (often also called the average).

display(sleephours.mean())  
> 7.285714285714286  
  
display(sleephours.max())  
> 9

There are many statistical routines available for the series. We will explore these in more depth in Chapter xx, where we look at exploratory data. We will also look at these in depth again in Chapter xx where we visualise data.

Tip. In Python, you can use the directory method to display all of the methods that an object can use. Some of these methods will be internal, system commands. They are prefixed by \_\_ and should not be refernced directly. The rest are meant to be used. By using directory we can see the difference in what we can do with a series versus a list.

In the interest of saving space (and paper) we will not list off the methods here, but instead we will count them. Observe:

ex\_list = [] # Just an empty list  
ex\_series = pd.Series(ex\_list) # Now an empty series  
  
display( len( dir(ex\_list))) # Number of   
> 46  
  
display( len( dir(ex\_series)))  
> 458

It appears there are almost ten times as many methods for a Series as for a list! Many of these will be useful for describing, shaping and analyzing data.

## Adding data to a Series

In many ways a Series works like a list, but one key difference between them is that a list automatically indexs values by position. Thus, if you have a 5 item list, ldemo, then ldemo.append("TEXT") will then automatically append “TEXT” to the list and treat it as the sixth item in order.

Trying to append a value to a Series on the other hand will throw a TypeError error. Only a Series (or DataFrame) can be appended to a Series. This is because a Series expects index & data, not just data. We can append a Series or a DataFrame since these have explicit indices, whereas lists do not. In a list, by contrast, the index is implicit and pased solely on positon.

This leads to two different strategies for adding data to a Series. The first is to create the entire Series as a primitive data type (such as a list or dictionary), then convert it to a Series and append to the original. The second uses the index to add new values one at a time. In this latter case, you have to stipulate the index of the new value. Be careful, you can also over write values of a series this way. Observe both of these strategies:

# Convert a list to a Series and append it to an existing Series.  
# Step 1. Create Series 1.   
ldemo1 = ["Kermit","Piggy","Fozzie"]  
sdemo1 = pd.Series(ldemo1)   
  
# Step 2. Create series 2.   
ldemo2 = ["Animal","Janice", "Dr. Teeth"]  
sdemo2 = pd.Series(ldemo2)   
  
# Step 3. Append series 2.   
# Notice the 'ignore\_index' argument.   
# Try running this without that argument (you will notice the index will be messed up)  
sdemo1 = sdemo1.append(sdemo2,ignore\_index=True)  
display(sdemo1)

In the above, we first created a second series and then appended it to the first. Note that we also said sdemo1 = sdemo1.append(.... This is because, by default, the append commend does not add the data to the original series. Instead, it creates a new series that combines the two earlier Series. If you don’t assign that new Series to a variable it will disappear once it has been created.

In the example below, we will first assume that the index is sequential. Then we will add the elements one at a time, with their index being one number higher than the highest current index value. How do we know what’s the highest value? Since len() gives the length of the series, and the series starts at zero, then whatever length is will be the next number in the sequence.

ldemo1 = ["Kermit","Piggy","Fozzie"]  
sdemo1 = pd.Series(ldemo1)   
  
# The second way, let's append the data one new index at a time.  
ldemo2 = ["Animal","Janice", "Dr. Teeth"]  
  
for i in ldemo2:   
 sdemo1[len(sdemo1)] = i  
display(sdemo1)

This code might seem a little more straightforward, but it is not recommended for large tasks. The way that Series are stored means that you are actually creating a new series with a new index every time you append a single value. With four elements this makes virtually no difference but with hundreds of thousands of data points continually creating new Series with every loop will slow down code unnecessarily.

**NOTE**: In both cases, Python does not enforce unique indices, which can lead to surprises. For example, let’s see what happens when we first create a series with a duplicate index (with the values 4, 5, and 4 rather than the defalt 0,1,2). Observe what happens when we assign a new value:

sdemo = pd.Series(["Kermit","Piggy","Fozzie"],index=[4,5,4])  
sdemo[4] = "Gonzo"  
display(sdemo)  
  
> 4 Gonzo  
> 5 Piggy  
> 4 Gonzo

Notice that in this case, since the index for both Kermit and Fozzie was ‘4’, they were both replaced.

## Deleting Data from a Series

To delete a data from a series, you can either delete the data by index or you can create a new Series without the unwanted data. Or you can delete the data by index. We will first delete by index. Remember here that indices are assigned to the value, not automatically assigned by position. If you have a list ldemo = ["Mon","Tues","Weds"] and drop Tues, then “Weds” is now in the second position. This is not the case for an index unless you deliberately re-index the new list.

sdemo = pd.Series(["Kermit","Piggy","Fozzie"])  
del sdemo[1]  
display(sdemo)  
  
> 0 Kermit  
> 2 Fozzie  
   
sdemo.index = range(len(sdemo))  
display(sdemo)  
  
> 0 Kermit  
> 1 Fozzie

## Working with missing data in a Series

A series can have missing data. Typically this data is signified by the NaN (numeric Python’s “Not a Number” character, a.k.a. np.nan). For example, if we create a series with an index going 0,1,2,3,4 and no data, then each of the columns will have a NaN value. ~~~ python sdemo = pd.Series(index=[0,1,2,3,4]) sdemo[0] = “Kermit” sdemo[3] = “Fozzie” display(sdemo)

0 Kermit 1 NaN 2 NaN 3 Fozzie 4 NaN dtype: object ~~~

Three things we tend to want to do when dealing with missing data. 1. **Get rid of missing values**: use Series.dropna(). Notice that this takes the argument inplace=True. if you want to get rid of missing data in your Series use this argument. If you want to create a copy withough missing data and preserve the original, just omit that argument. 2. **Replace missing values**: use Series.fillna(). This is for instances where we might have missing data and simply want to insert some value here. For example, if we have a count of number of laughs a specific muppet recieved in an episode, we might end up with missing data if the muppet did not get any laughts or did not appear. In which case, smuppet.fillna(0) will fill all the missing values with 0. 3. **Filter in or out by missing values**. Rather than drop the missing values we often want to slice based on them. Here we can use Series.isna() inside a slice.

Observe all three of these below: ~~~ python sdemo = pd.Series(index=[0,1,2,3]) # Create a Series with index and no values. sdemo[0] = “Kermit” sdemo[3] = “Fozzie” display(sdemo)

0 Kermit 1 NaN 2 NaN 3 Fozzie

# Filling the N/A values

display(sdemo.fillna(“extra”))

0 Kermit 1 extra 2 extra 3 Fozzie

# Dropping the N/A values

display(sdemo.dropna())

0 Kermit 3 Fozzie

# Slicing by the NA values

display(sdemo[sdemo.isna()])

1 NaN 2 NaN ~~~

## Getting unique values in a Series

Depending on the data, you might want to know whether or how many values are unique. Some examples: 1. Reading log traffic data: how many IP addresses are unique? 2. Getting a stream of tweets: how many accounts are unique? 3. Checking that an index has entirely unique values.

The Series.unique() command will return a new series with only one entry for each unique value. This will be returned as an “array”, which is very similar to a list. To transform the array back into a series you will have to do that explicitly.

ser1 = pd.Series(["Kermit","Fozzie","Kermit","Piggy","Fozzie"])  
display(ser1.unique())  
  
> array(['Kermit', 'Fozzie', 'Piggy'], dtype=object)  
  
ser2 = pd.Series(ser1.unique()) # To transform back to a Series

# Sorting a Series

A series can be sorted by the values (Series.sort\_values()) or by the index (Series.sort\_index()). The sort will be ascending by default, but you can change it with the argument ascending=False. This is another method that requires the inplace=True argument. Otherwise, it will return a new, sorted, Series and leave the old one in place.

ser1 = pd.Series( {"Kermit":"Frog",  
 "Piggy":"Pig",  
 "Fozzie":"Bear",  
 "Robin":"Frog"} )  
  
ser1.sort\_values(ascending=True,inplace=True)  
display(ser1)  
  
> Fozzie Bear  
> Kermit Frog  
> Robin Frog  
> Piggy Pig  
  
ser2 = ser1.sort\_index(ascending=False)  
display(ser2)  
  
> Robin Frog  
> Piggy Pig  
> Kermit Frog  
> Fozzie Bear

## Changing Series Values I: Adding, Multiplying, etc…

With a series you can change the values using the standard arithmetic operators. These treat the series literally like a series of values and does something to each one. So for example, if you say Series + 1 it will add one to each value in the series. If the series is not just numbers (and valid) it will throw an error. Series + "A" will append A to each value in the series if they are characters and throw an error otherwise.

import numpy as np   
ser1 = pd.Series([1,np.NaN,7])  
  
ser1 = ser1\*2  
display(ser1)  
> 0 2.0  
> 1 NaN  
> 2 14.0  
  
ser1 = ser1-4  
display(ser1)  
> 0 -2.0  
> 1 NaN  
> 2 10.0  
  
ser1 = ser1 + "A" #Note that the Series is full of numbers so it throws an error  
> "TypeError ..."  
  
ser2 = pd.Series(["Kermit","Piggy","Fozzie"])  
ser2 = ser2 + " the Muppet"  
display(ser2)  
  
> 0 Kermit the Muppet  
> 1 Piggy the Muppet  
> 2 Fozzie the Muppet

## Changing Series Values II: Recoding values using Map

A really common task in social statistics is to recode values. For example, you might have a list of text values (such as “Strongly Agree”, “Agree”, “Disagree”, etc…) that you want to turn into numbers. You might have a text entry form that you want to recode (such as “How do you identify your gender” with answers like “Man”, “Female”, “Cis male”, “Transgendered male”, “agender”) which you might recode into more manageable categories. To recode these you can create a dictionary of values and then map those values on to your series.

A scenario that I encountered in a data cleaning exercise had to do just this. We asked people to label the gender of persons behind Twitter accounts. They were all politicians, so there was no need to create a “not a person” flag. All the MPs were cisgendered, meaning they presented as the gender they were assigned at birth. But still, the coders gave six or seven different ways of writing what was essentially “Male”, “Female”, “Unknown”.

display(gender\_series.unique())  
> array(['Male', 'Man', 'Male (sex)', "Woman", "Female", "Female "], dtype=object)

To recode the data, we first did a unique(), and then typed by hand the dictionary using those unique values. One thing that tripped us up was that someone had used "Female " with that trailing space. So here’s what the resulting dictionary looks like:

gender\_recode\_dict = {"Male":"M",   
 "Man":"M",  
 "Male (sex)": "M",  
 "Woman":"F",  
 "Female":"F",  
 "Female ":"F"}

Then to map it on to the series the following was done:

gender\_recode = gender\_series.map(gender\_recode\_dict)  
  
gender\_recode.value\_counts()  
> M 1046  
> F 942

## Changing Series Values III: Defining your own recode using Lambda

In the above example, map() took in a dictionary and then mapped the keys found in the series to the values found in the dictionary. But map is a whole lot more powerful when you can define your own function for what to do with the elements in a Series. As a trivial example, we might want to take every element in a Series of characters and transform it into lower case. At this point, if you had to do this, you might think to use a for loop. That is, for each value in the series, get the index, get the value, transform the value to lower case and then reinsert it. Not only is this very fragile (what about two values with the same index?) it doesn’t take advantage of some of the speed improvements that can happen in the back end.

With lambda we do this transformation inside of a map command. Lambda is fed a value and then returns some transformation of this value. Observe the example of a lambda command that squares every value in a Series:

ser1 = pd.Series([1,3,5],index=["one","three","five"])   
ser1 = ser1.map(lambda val: val\*\*2)  
display(ser1)  
> one 1  
> three 9  
> five 25

One of the most useful things about lambda is that you can embed your own functions in it. This is a fast and tidy way to do complex operations on values in a Series. For example, let’s say we want to check if a blob of text has an email address in it. We first define a function to detect email addresses. We won’t go into details here of how to do that (See chapter xx for an example of how to detect emails). What’s important is that we know the has\_email(TEXT) function works. It takes in some TEXT and then returns either True or False if it has an email.

smessages = pd.Series(["Hey, catch me at bernie.hogan@oii.ox.ac.uk",   
 "I once emailed steve@apple.com and got a reply",   
 "I don't really use email"])  
  
result = smessages.map(lambda : has\_email(x))   
  
display(result)   
> 0 True  
> 1 True  
> 2 False

## Summary: The Series

The series is a very powerful tool for manipulating data. Like a list it has ordered values. Like a dictionary the values can be accessed by key (or in this case, by ‘index’). We first showed how to create a series using a list and a dictionary (x.todo). We then showed how to add values to that new series. We can filter the series, delete specific values and transform all the values.

These operations are extremely foundational to the act of using Python for data science tasks. In the exercise sheet, you will be given a series and asked to clean this data. Since we are just dealing with one dimension of data, we are not yet at the point where we can ask some really interesting questions, but at least we will have some of the basics down. One thing that is worth attending to in the exercises is how to chain a bunch of operations together. For example, you will not only get value\_counts() on a Series, but then use that value\_counts() to investigate how to transform, summarise, and explain the data.

In the next section, we will look at DataFrames. These are tables of data with rows and columns. Each column is treated like a Series. Thus, you will discover that many of the operations that we have learned for the Series directly apply to the DataFrame. With the dataframe and some libraries we can do exceedingly powerful things with data. But first we have to learn how to get data in, get data out, and filter that data. Then in the next chapter, we will look at some file formats that can be usefully converted into DataFrames.

# Introducing The Table for Scientific Python: The DataFrame

A series is a flexible tool for managing a single distribution with an index. The index can be either the default sequence of numbers starting from zero, or it can be a list of labels. If we have two Series, each with the same index, we can combine them together. This will create a DataFrame. Depending on the nature of the research we will either be building a DataFrame by merging together series one by one or we will import a DataFrame from another context, be it a text file, a webpage or some other file format. In this section, we will be building small DataFrames from scratch in order to demonstrate their features and highlight the similarities and differences to these operations when done on a Series. In the next chapter we will show how to create a DataFrame from existing file formats.

## From a Series to a DataFrame

A DataFrame with one column of data looks very similar to a Series. However, the DataFrame comes with some extra features. Not the least of which is that when you display() a DataFrame, you get a nice looking HTML-formatted table. Observe the difference below:

import pandas as pd   
  
ser1 = pd.Series({"Kermit":"Frog", "Fozzie":"Bear", "Janice":"Hippy"})  
  
display(ser1)  
  
df1 = pd.DataFrame(ser1)  
  
display(df1)

Kermit Frog  
Fozzie Bear  
Janice Hippy  
dtype: object

0

Kermit

Frog

Fozzie

Bear

Janice

Hippy

Notice that the output from the second display command had the words ‘Kermit’, ‘Fozzie’, and ‘Janice’ in bold, with some text shading. This is a feature of Jupyter that means we can view DataFrames as rich tables. This will be very useful when we have large and complex tables. Viewing them in monospace text like what we do with the Series makes it harder to read.

You might also notice that above ‘frog’ in the table is the number . This is just the index of that column since we did not name the column. In our case we have two options. We could have named the Series, which will propagate the name to the DataFrame, or we can just name the columns in the DataFrame. Observe the same code, but this time we will name the Series first. Then below that we will change the name of the DataFrame column to something else:

ser1 = pd.Series({"Kermit":"Frog", "Fozzie":"Bear", "Janice":"Hippy"}, name="MuppetType")  
  
df1 = pd.DataFrame(ser1)  
  
display(df1)  
  
df1.columns = ["NewColumnName"]  
  
display(df1)

MuppetType

Kermit

Frog

Fozzie

Bear

Janice

Hippy

NewColumnName

Kermit

Frog

Fozzie

Bear

Janice

Hippy

# Getting Data in to a DataFrames

Up to this point we have only seen DataFrames that look like nicely formatted Series. However, it is important for us to be able to compare multiple columns of data. To get these multiple columns of data, here are a few approaches:

1. From a list of lists (or equivalently, an Array),
2. From a dictionary where the keys are indices and the values are lists of data,
3. By adding a new series to an existing DataFrame .

These different ways of building DataFrames will form the basis of a great deal of your work in data science. As you will see later, data comes in a variety of formats, but we need to transform the data into a consistent and workable format for analysis. Thus, getting this data into a DataFrame will be of central importance. Although there are many possible ways to create a DataFrame, they essentially are variants on these three: from a list of lists, from an existing DataFrame/Series, from a Dictionary.

For the uninitiated, you might wonder what is the advantage of doing this over just typing in data in Excel or a similar spreadsheet program? Returning to the discussion in Chapter 1 about fixed versus marginal costs, it will become evident through working with data that while spreadsheets have a very low fixed cost (since you just load them up and start typing in data), they have a very high marginal cost, since every operation and new data point can involve lots of clicking, saving, and typing. We want to avoid marginal costs (where each new row or line of data takes up our time) so that we can more effectively scale from three or four rows up to three or four thousand (or million) rows.

To create these multi-column DataFrames we will continue to use information about some of the Muppets we have already mentioned. This time, in addition to the ‘type of muppet’, we will add a column about their first apperance as well as well as their gender.

## From a List of Lists

The way to create this data depends largely on how the data was initially formatted. For example, in a list of lists, it might look something like this:

muppetList = [["Kermit","Frog",1955,"Male"],   
 ["Miss Piggy", "Pig", 1974, "Female"],   
 ["Gonzo", "Unknown", 1970, "Male"]]  
  
muppetFrame1 = pd.DataFrame(muppetList)  
display(muppetFrame1)

0

1

2

3

0

Kermit

Frog

1955

Male

1

Miss Piggy

Pig

1974

Female

2

Gonzo

Unknown

1970

Male

As a barebones DataFrame this is okay, but we are missing the columns and labelled row indices. The column names were not in the original list so we should just add those like we did above. But what about setting the first row (the Muppet name) as the index? There are a number of ways to do this. Here I use DataFrame.set\_index() to make one of the existing columns the index. Observe:

muppetFrame1.set\_index(0, inplace=True)  
display(muppetFrame1)

1

2

3

0

Kermit

Frog

1955

Male

Miss Piggy

Pig

1974

Female

Gonzo

Unknown

1970

Male

Notice the above ‘Kermit’? That’s because the index itself can have a name. In this case the column was named so that’s now our index name. Index names can be useful if you have complex nested data. For example, in Chapter xx on time series, we will see how to create an index by both the month and the year, so you can query and slice data by either of these in the same DataFrame.

So the MuppetType data was in a dictionary, then a Series, then a DataFrame. Can we skip a step and just add data to the DataFrame directly? Sometimes. There are some collection types that can be created as a DataFrame. A dictionary is one of them. However, it does not work the way a Series does. Instead you need to create a DataFrame using the from\_dict() method. Observe:

df1 = pd.DataFrame.from\_dict({"Kermit":"Frog", "Fozzie":"Bear", "Janice":"Hippy"},orient="index",columns=["MuppetType"])  
  
display(df1)

MuppetType

Kermit

Frog

Fozzie

Bear

Janice

Hippy

df1 = pd.DataFrame.from\_dict({"Kermit":["Frog"], "Fozzie":["Bear"], "Janice":["Hippy"]},orient="columns")#,columns=["MuppetType"])  
  
display(df1)

Kermit

Fozzie

Janice

0

Frog

Bear

Hippy

## From a dictionary

As noted above, to make a DataFrame from a dictionary requires you to have it structured like {KEY: [VALUES,..]}. If you have just {KEY:SINGLE\_VALUE} as your dictionary structure, you’re better off making it a Series. Observe below how we create a Dictionay and then a DataFrame from that Dictionary:

muppetDict = {"Kermit": ["Frog",1955,"Male"],   
 "Miss Piggy":["Pig", 1974, "Female"],   
 "Gonzo": ["Unknown", 1970, "Male"]}  
  
muppetFrame2 = pd.DataFrame.from\_dict(muppetDict,orient="index")  
display(muppetFrame2)

0

1

2

Kermit

Frog

1955

Male

Miss Piggy

Pig

1974

Female

Gonzo

Unknown

1970

Male

Notice that we used the from\_dict() method instead of simply creating an instance using pd.DataFrame().

One new argument here is orient="index". This means that the keys of the dictionary are going to be the indices for the rows. Otherwise, the dictionary would be treated in the other direction. You can try yourself to do it by using muppetFrame2 = pd.DataFrame(muppetDict). You’ll notice that it makes Kermit, Piggy, and Gonzo as the columns. Sometimes this is the behaviour we want, but not here. Remember: rows in cases and variables in columns.

## By Adding a New Series to an existing DataFrame

It is very common to attach new data to a DataFrame. You might be recoding a variable, getting some new calculation or just parsing the text that is already there. For example, since we have the year of the muppet’s first appearance, we could create a new column for ‘decade of first appearance’ by doing some calculation on that year value.

To note, if we want to link two DataFrames together, this is slightly more tricky and will be covered later in the section on merging in Chapter 6xx. For now we are just adding a single column or a single row to an existing DataFrame. To illustrate this, we will start with the simple DataFrame that has the Muppet name as the index and the type of character as a single column. Then we will add ‘year of first appearance’ first and calcuate ‘decade of first appearance’ second.

muppetFrame3 = pd.DataFrame.from\_dict({"Kermit":"Frog", "Miss Piggy":"Pig", "Gonzo":"Unknown"},orient="index",columns=["MuppetType"])  
  
muppet\_year = pd.Series({"Gonzo":1970,"Kermit":1955,"Miss Piggy":1974})  
  
muppetFrame3["MuppetYear"] = muppet\_year  
  
display(muppetFrame3)

MuppetType

MuppetYear

Kermit

Frog

1955

Miss Piggy

Pig

1974

Gonzo

Unknown

1970

One of the nice things about ensuring that cases are in the rows and variables in the columns is that it makes it easy to add new variables. Also, Pandas can be pretty clever about linking the data. Notice above that we had a dictionary with Gonzo first, but the DataFrame had Kermit first? Since Gonzo was the index for both the existing DataFrame and the new Series, when they were merged the program was able to link the data together. In this case, we can think of Gonzo’s name as the *key* that links the data. Series and dictionaries have keys, but lists do not. So if you add a list to a DataFrame then it will just get added in the order that’s in the list. Observe:

muppet\_gender = ["male","female","male"]  
  
muppetFrame3["MuppetGender"] = muppet\_gender  
  
display(muppetFrame3)

MuppetType

MuppetYear

MuppetGender

Kermit

Frog

1955

male

Miss Piggy

Pig

1974

female

Gonzo

Unknown

1970

male

Now we can start to see a pattern here of DataFrame[COLNAME] = column\_of\_data. Let’s use this to create a column from the data that is already in the DataFrame. We will take the year value, round it down to the nearest 10 years and call it muppet\_decade. To do this, we will use the map and lambda features that we had introduced with a series. This is because we will first query MuppetYear *as a series*, transform the series, and reinsert it into the data. Follow the code carefullly since this will all be done in one line. Observe:

muppetFrame3["MuppetDecade"] = muppetFrame3["MuppetYear"].map(lambda x: (x // 10)\*10)  
  
display(muppetFrame3)  
  
muppetdf = muppetFrame3

MuppetType

MuppetYear

MuppetGender

MuppetDecade

Kermit

Frog

1955

male

1950

Miss Piggy

Pig

1974

female

1970

Gonzo

Unknown

1970

male

1970

There are a number of ways to get an integer to round down to the nearest 10. In the way above I used ‘integer division’ which is division that gives a remainder rather than a precise decimal format. Then I just multiplied that by ten. So in effect, I simply removed the remainder of dividing by 10 from the year to get the decade.

# Returning Data from a DataFrame: Querying and Slicing

## Returning a single row or column

To return an entire DataFrame you simply invoke its name. In our examples, muppetFrame3 will return the entire DataFrame. To get a column in the data, it is like querying a list or a dictionary: you use the square [ and ] brackets. So if your DataFrame has a column “MuppetType”, then the syntax muppetFrame3["MuppetType"] will return the respective column as a series.

It turns out the reason this works is because querying by column is the default *indexer* for a DataFrame. When you use DataFrame[\*] you are using an indexer, as opposed to using DataFrame(\*), which is a method.

There are different kidns of indexers for DataFrames to accomplish different goals. Two in particular are worth considering here. These are the indexers that will return a row instead of a column. Recall that each row will have both a label and a position in terms of being first, second, etc… Accordingly, one of the indexers will index by row label and the other by row position.

* .loc[] returns a row based on the label of the row in the index. By default, the index is simply a list of sequential numbers, but that is merely the default. It could be anything. In our example it is the name of the Muppet. Thus, muppetFrame3.loc["Gonzo"] should return the row labelled Gonzo in the index.
* .iloc[] returns a row based on the position of the row in the sequence of rows in the DataFrame. Since Gonzo is in position 2 (as Python indexs from 0), muppetFrame3.iloc[2] should return the Gonzo row.
* *Tip*: The indexer starts with l for label and starts with i for index.

print(muppetdf.loc["Gonzo"])

MuppetType Unknown MuppetYear 1970 MuppetGender male MuppetDecade 1970 Name: Gonzo, dtype: object

print(muppetdf.iloc[2])

MuppetType Unknown MuppetYear 1970 MuppetGender male MuppetDecade 1970 Name: Gonzo, dtype: object

## Returning multiple rows

You can return multiple rows at once. This is handy when you want to filter and merge data. As a motivating example, imagine that you download data from *reddit* and insert it into a table. It will have a ton of extraneous columns. The data you want to study might be a small subset of what is available. Asking for the columns you want and building your dataset from there is a prudent way to keep focused on a research question.

To ask for multiple columns, you must ask for them as a collection *inside* the indexer. This means it typically looks like square brackets inside square brackets. For example, asking for MuppetYear and MuppetType in the same query would be as muppetdf[ ["MuppetYear","MuppetType"]]. Whatever order you ask for them is the order they will be in the resulting DataFrame.

### Returning a single element

Getting a single element of a DataFrame is an extension of what we just did. Now that we have a Series (the row), we can then query for one element of that row.

Notice that when we queried for .loc["Gonzo"], it returned a Series corresponding to Gonzo’s row in the DataFrame. Since this Series has labels (corresponding to Gonzo’s values in the table) we can then use the labels to get one element from Gonzo’s row. We can also use position in the Series, which I will show afterwards.

To get the year of Gonzo’s first appearance, we can chain together muppetdf.loc["Gonzo"] with ["MuppetYear"]. This will then look like muppetdf.loc["Gonzo"]["MuppetYear"]. Lukcily, Pandas provides a little “syntactic sugar”, so that you can put row then column in the same indexer like so: muppetdf.loc["Gonzo","MuppetYear"]. Since Gonzo was in position 2 and MuppetYear is in position 1, we could also write muppetdf.iloc[2,1]. Note the latter one is *iloc* since we are using position.

You might be wondering now about whether you can use muppetdf["MuppetYear"] first to get the entire column and then find Gonzo in that column. You sure can, however, it is worth pointing out that this is considered bad form. Generally speaking, go rows first. It is for this reason that while you can indeed query muppetdf["MuppetYear"]["Gonzo"], you can neither use the syntactic sugar of muppetdf["MuppetYear","Gozno"] nor muppetdf[1,2]. It is here that we are reminded that a DataFrame is not considered a completely symmetric data structure. Rather, rows are for cases and columns are for varaibles.

Of the three ways to query: ["row\_label"]["column\_label"], ["row\_label","column\_label"] or ["row\_position","column\_position"], which one would be considered the most Pythonic? Recall **FREE**. They are all functional. However, one approach is typically more robust: using labels. The labels for rows and columns should not change even if the data varies. Consequently, using labels (.loc) is often more robust than position (.iloc) since position can change with sort order.

So, of ["row"]["column"] or ["row","column"] which one would be more efficient? The second one is more efficient, because Pandas treats it as a single query. In the first instance (i.e. ["row"]["column"], you get the row as a new series, then you look inside it. In the second case, you get the row and the column in a single indexer request, no intermediary series necessary. This will not be noticable in these toy DataFrames, but when you are handling large amounts of data, these speed gains can really affect the time it takes to complete tasks. It has a second benefit that we will cover at the end of the chapter in the section on deep versus shallow copying. First lets get through all the ways to get data out, we have one left.

## Returning a slice of data

The last way of returning data that we should cover is a slice. Slices are incredibly useful for answering questions about data. They allow us powerful ways to filter the data in a DataFrame. There are multiple ways to slice up data in a DataFrame.

### Slicing by position

You can use the colon (:) to indicate a range of elements. For example, muppetdf.iloc[2:] will get all the rows from position two onwards. Putting a number after the column would be the position up to, but not including. So muppetdf.iloc[:2] will get all the rows up to but not including Gonzo in position 2.

### Boolean slicing

Recall above that we could filter a series using a boolean indexer? DataFrames work the same way. You can evaluate against a column of data and it will return the rows that fit the criteria. So, in our table we could ask for Muppets that are male or Muppets that first appeared after 1967. This means we first focus on a column and ask whether that column meets some criteria. This becomes a series of True, False statements. If the row corresponds to a True statement, it is kept. Observe:

muppetdf["MuppetYear"] > 1967

Kermit False Miss Piggy True Gonzo True Name: MuppetYear, dtype: bool

muppetdf[muppetdf["MuppetYear"] > 1967]

MuppetType

MuppetYear

MuppetGender

MuppetDecade

Miss Piggy

Pig

1974

female

1970

Gonzo

booger

1970

male

1970

The boolean query itself just returned a Series of True/ False elements with an index that corresponds to the DataFrame. We then pipe this into an indexer and out comes only the rows that were true: Miss Piggy, who first appeared in 1974 and Gonzo who first appeared in 1970.

## Deep Copies versus Shallow Copies

When we ask for data from python, sometimes it will give us a ‘view’ of the original data, this is a “shallow copy”. Sometimes it will copy the data to a new location and then return that newly copied data, this is a “deep copy”. Understanding when this happens can help for two issues we will encounter when we have to manage larger data sets: 1. Deep copies need their own memory space in the computer since they are now completely different DataFrames. 2. Altering or deleting data can lead to mishaps when changing data on a subset or copy of the data actually changes the original data. 3. Altering can also lead to accidents or warnings when you try to change the original data but actually end up working on a copy.

See in this code here what happens when we try to change the value of Gonzo’s type from “unknown” to “weirdo”.

# Attempt 1 (which will fail)  
muppetdf.loc["Gonzo"]["MuppetType"] = "weirdo"

/Users/crafty/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:1: SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy """Entry point for launching an IPython kernel.

# Attempt 2 (which will succeed)  
muppetdf.loc["Gonzo","MuppetType"] = "Weirdo"

NameError Traceback (most recent call last)

in 1 # Attempt 2 (which will succeed) —-> 2 muppetdf.loc[“Gonzo”,“MuppetType”] = “Weirdo”

NameError: name ‘muppetdf’ is not defined

In the first instance, what happened was that mupetdf.loc["Gonzo"] is one query that first created a slice. Then [“MuppetType”] was a subquery. So then when we assign a new value, the program was about to change the value of this newly created slice, not the value in the origianl muppetdf. So it threw an error. In the second one, we queried that cell directly and changed the object in the cell from "unknown" to "Weirdo".

The original query failed because we created a deep copy by accident. We can see this error work the other way as well, that is, when we give the DataFrame a new label but we are actually changing the original DataFrame. That can be especially dangerous because it won’t throw an error, it will just change the data. Observe:

newmuppetdf = muppetdf  
  
newmuppetdf.loc["Kermit","MuppetType"] = "Lizard" #change in newmuppetdf  
  
display(muppetdf.loc["Kermit","MuppetType"]) #it appears in original muppetdf

‘Lizard’

Above what happened is we just renamed the original DataFrame. One way to ensure that a newly assigned DataFrame is a copy is to use the copy() method directly, as in newmuppetdf = muppetdf.copy(). There are other instances where you should be careful about deep versus shallow copies. For now, we can only introduce the topic. Instead of trying to master this right now, I recommend simply being really careful with your data, and checking it at each step. It is easy to have a sneaky error propagate its way through your entire code. Practising *data skepticism* is critical here. We will say more about this in the chapter on exploring data.

# Summary: DataFrames

The DataFrame is a staple of data science and thus social data science. A surprising amount of what is done in data science can be described as getting the right data into a DataFrame from another shape, and aligning that data with other data that we might want to use. The DataFrame allows us to filter, make comparisons, and produce visualisations, both static charts and interactive diagrams.

The exercise sheet for this chapter focuses on a lot of little exercises to practice building, altering, and querying DataFrames. In the next chapter I will show you how to transform a number of common data structures into a DataFrame. These structures are pervasive on the web. For example, if you want data from Wikipedia, Facebook, Twitter, Reddit, Tumblr, or Google, you’ll be reciving data in formats like XML and JSON. They do not look like a DataFrame, but with a little care we can import them into DataFrames and then start asking questions about the data.