2. Pandas Data Structures

2.1 Introduction

Chapter 1 introduced the Pandas `DataFrame` and `Series` objects. These data structures resemble the primitive Python data containers (lists and dictionaries) for indexing and labeling, but have additional features that make working with data easier.

Concept Map

1. Prior knowledge

a. containers

b. using functions

c. subsetting and indexing

2. Loading in manual data

3. Series

a. creating a series

- dict

- ndarray

- scalar

- lists

b. slicing

4. DataFrame

Objectives

This chapter will cover:

1. Loading in manual data
2. The `Series` object
3. Basic operations on `Series` objects
4. The `DataFrame` object
5. Conditional subsetting and fancy slicing and indexing
6. Saving out data

2.2 Creating Your Own Data

Whether you are manually inputting data or creating a small test example, knowing how to create dataframes without loading data from a file is a useful skill. It is especially helpful when you are asking a question about a StackOverflow error.

2.2.1 Creating a Series

The Pandas `Series` is a one-dimensional container, similar to the built-in Python `list`. It is the data type that represents each column of the `DataFrame`. Table 1.1 lists the possible `dtype`s for Pandas `DataFrame` columns. Each column in a dataframe must be of the same `dtype`. Since a dataframe can be thought of a dictionary of `Series` objects, where each `key` is the column name and the `value` is the `Series`, we can conclude that a `Series` is very similar to a Python `list`, except each element must be the same `dtype`. Those who have used the `numpy` library will realize this is the same behavior as demonstrated by the `ndarray`.

The easiest way to create a `Series` is to pass in a Python `list`. If we pass in a list of mixed types, the most common representation of both will be used. Typically the `dtype` will be `object`.

```
import pandas as pd


s = pd.Series(['banana', 42])
print(s)
```

```
0    banana
1        42
dtype: object
```

Notice on the left that the “row number” is shown. This is actually the `index` for the series. It is similar to the row name and row index we saw in Section 1.3.2 for dataframes. It implies that we can actually assign a “name” to values in our series.

Click here to view code image

```
# manually assign index values to a series
# by passing a Python list
s = pd.Series(['Wes McKinney', 'Creator of Pandas'],
              index=['Person', 'Who'])
print(s)
```

```
Person         Wes McKinney
Who       Creator of Pandas
dtype: object
```

QUESTIONS

1. What happens if you use other Python containers such as `list`, `tuple`, `dict`, or even the `ndarray` from the `numpy` library?

2. What happens if you pass an `index` along with the containers?

3. Does passing in an `index` when you use a `dict` overwrite the index? Or does it sort the values?

2.2.2 Creating a DataFrame

As mentioned in Section 1.1, a `DataFrame` can be thought of as a dictionary of `Series` objects. This is why dictionaries are the the most common way of creating a `DataFrame`. The `key` represents the column name, and the `values` are the contents of the column.

Click here to view code image

```
scientists = pd.DataFrame({
    'Name': ['Rosaline Franklin', 'William Gosset'],
    'Occupation': ['Chemist', 'Statistician'],
    'Born': ['1920-07-25', '1876-06-13'],
    'Died': ['1958-04-16', '1937-10-16'],
    'Age': [37, 61]})
print(scientists)
```

```
   Age        Born        Died               Name    Occupation
0   37  1920-07-25  1958-04-16  Rosaline Franklin       Chemist
```

```
1   61  1876-06-13  1937-10-16     William Gosset  Statistician
```

Notice that order is not guaranteed.

If we look at the documentation for `DataFrame`,[1] we seethatwecan usethe `columns` parameter or specify the column order. If we wanted to use the `name` column for the row index, we can use the `index` parameter.

1. `DataFrame` documentation: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

Click here to view code image

```
scientists = pd.DataFrame(
    data={'Occupation': ['Chemist', 'Statistician'],
          'Born': ['1920-07-25', '1876-06-13'],
          'Died': ['1958-04-16', '1937-10-16'],
    'Age': [37, 61]},
    index=['Rosaline Franklin', 'William Gosset'],
    columns=['Occupation', 'Born', 'Died', 'Age'])
print(scientists)
```

```
                    Occupation        Born        Died  Age
Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
William Gosset     Statistician  1876-06-13  1937-10-16   61
```

The order is not guaranteed because Python dictionaries are not ordered. If we want an ordered dictionary, we need to use the `OrderedDict` from the `collections` module.[2] Doing so is not as simple as wrapping the `OrderedDict` function around our dictionary, however, because the dictionary would have already lost its order by the time it was created and passed into our `OrderedDict` function.

---

2. Collections module: https://docs.python.org/3.6/library/collections.html

Click here to view code image

```
from collections import OrderedDict



# note the round brackets after OrderedDict

# then we pass a list of 2-tuples

scientists = pd.DataFrame(OrderedDict([

    ('Name', ['Rosaline Franklin', 'William Gosset']),

    ('Occupation', ['Chemist', 'Statistician']),

    ('Born', ['1920-07-25', '1876-06-13']),

    ('Died', ['1958-04-16', '1937-10-16']),

    ('Age', [37, 61])

    ])

)

print(scientists)
```

```
                Name    Occupation        Born        Died  Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
1     William Gosset  Statistician  1876-06-13  1937-10-16   61
```

2.3 The Series

In Section 1.3.2.1, we saw how the slicing method affects the `type` of the result. If we use the `loc` attribute to subset the first row of our `scientists` dataframe, we will get a `Series` object back.

First, let's re-create our example dataframe.

Click here to view code image

```
# create our example dataframe
# with a row index label
scientists = pd.DataFrame(
    data={'Occupation': ['Chemist', 'Statistician'],
          'Born': ['1920-07-25', '1876-06-13'],
          'Died': ['1958-04-16', '1937-10-16'],
    'Age': [37, 61]},
    index=['Rosaline Franklin', 'William Gosset'],
    columns=['Occupation', 'Born', 'Died', 'Age'])
print(scientists)
```

```
                   Occupation        Born        Died  Age
Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
```

```
William Gosset     Statistician  1876-06-13  1937-10-16   61
```

Now we select a scientist by the row index label.

Click here to view code image

```
# select by row index label
first_row = scientists.loc['William Gosset']
print(type(first_row))
```

```
<class 'pandas.core.series.Series'>
```

```
print(first_row)
```

```
Occupation    Statistician
Born            1876-06-13
Died            1937-10-16
Age                     61
Name: William Gosset, dtype: object
```

When a series is printed (i.e., the string representation), the index is printed as the first "column," and the values are printed as the second "column." There are many attributes and methods associated with a `Series` object.[3] Two examples of attributes are `index` and `values`.

3. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

Click here to view code image

```
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

```
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' 61]
```

An example of a `Series` method is `keys`, which is an alias for the `index` attribute.

Click here to view code image

```
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

By now, you might have questions about the syntax for `index`, `values`, and `keys`. More information about attributes and methods is found in Appendix S on classes. Attributes can be thought of as properties of an object (in this example, our object is a `Series`). Methods can be thought of as some calculation or operation that is performed. The subsetting syntax for `loc`, `iloc`, and `ix` (from Section 1.3.2) consists of all attributes. This is why the syntax does not rely on a set of round parentheses, `()`, but rather a set of square brackets, `[ ]`, for subsetting. Since `keys` isamethod, if we wanted to get the first key (which is also the first index), we would use the square brackets *after* the method call. Some attributes for the series are listed in Table 2.1.

Click here to view code image

```
# get the first index using an attribute
print(first_row.index[0])
```

```
Occupation
```

```
# get the first index using a method
print(first_row.keys()[0])
```

```
Occupation
```

Table 2.1 Some of the Attributes Within a Series

`Series`	**Attributes Description**
`loc`	Subset using index value
`iloc`	Subset using index position
`ix`	Subset using index value and/or position
`dtype` or `dtypes`	The type of the `Series` contents
`T`	Transpose of the series
`shape`	Dimensions of the data
`size`	Number of elements in the `Series`
`values`	`ndarray` or `ndarray`-like of the `Series`

2.3.1 The Series Is ndarray-like

The Pandas data structure known as `Series` is very similar to the `numpy.ndarray` (Appendix R). In turn, many methods and functions that operate on a `ndarray` will also operate on a `Series`. A `Series` may sometimes be referred to as a "vector."

2.3.1.1 Series Methods

Let's first get a series of the "Age" column from our `scientists` dataframe.

```
# get the 'Age' column

ages = scientists['Age']

print(ages)
```

```
Rosaline Franklin    37

William Gosset       61

Name: Age, dtype: int64
```

`Numpy` is a scientific computing library that typically deals with numeric vectors. Since a `Series` can be thought of as an extension to the `numpy.ndarray`, there is an overlap of attributes and methods. When we have a vector of numbers, there are common calculations we can perform.[4]

4. Descriptive statistics: http://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics

```
print(ages.mean())
```

```
49.0
```

```
print(ages.min())
```

```
37
```

```
print(ages.max())
```

```
61
```

```
print(ages.std())
```

```
16.9705627485
```

The `mean`, `min`, `max`, and `std` are also methods in the `numpy.ndarray`.[5] Some `Series` methods are listed in Table 2.2.

5. `numpy ndarray` documentation: http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

Table 2.2 Some of the Methods That Can Be Performed on a Series

`Series` Methods	Description
`append`	Concatenates two or more `Series`
`corr`	Calculate a correlation with another `Series` *
`cov`	Calculate a covariance with another `Series` *
`describe`	Calculate summary statistics*
`drop_duplicates`	Returns a `Series` without duplicates
`equals`	Determines whether a `Series` has the same elements
`get_values`	Get values of the `Series` ; same as the `values` attribute
`hist`	Draw a histogram
`isin`	Checks whether values are contained in a `Series`
`min`	Returns the minimum value
`max`	Returns the maximum value
`mean`	Returns the arithmetic mean
`median`	Returns the median
`mode`	Returns the mode(s)
`quantile`	Returns the value at a given quantile

`replace`	Replaces values in the `Series` with a specified value
`sample`	Returns a random sample of values from the `Series`
`sort_values`	Sorts values
`to_frame`	Converts a `Series` to a `DataFrame`
`transpose`	Returns the transpose
`unique`	Returns a `numpy.ndarray` of unique values

* Indicates missing values will be automatically dropped.

2.3.2 Boolean Subsetting: Series

Chapter 1 showed how we can use specific indices to subset our data. Only rarely, however, will we know the exact row or column index to subset the data. Typically you are looking for values that meet (or don't meet) a particular calculation or observation.

To explore this process, let's use a larger data set.

Click here to view code image

```
scientists = pd.read_csv('../data/scientists.csv')
```

We just saw how we can calculate basic descriptive metrics of vectors. The `describe` method will calculate multiple descriptive statistics in a single method call.

```
ages = scientists['Age']
```

```
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```
# get basic stats
print(ages.describe())
```

```
count     8.000000
mean     59.125000
std      18.325918
min      37.000000
25%      44.000000
50%      58.500000
75%      68.750000
max      90.000000
```

```
Name: Age, dtype: float64
```

```
# mean of all ages
print(ages.mean())
```

```
59.125
```

What if we wanted to subset our ages by identifying those above the mean?

Click here to view code image

```
print(ages[ages > ages.mean()])
```

```
1    61
2    90
3    66
7    77
Name: Age, dtype: int64
```

Let's we tease out this statement and look at what `ages > ages.mean()` returns.

Click here to view code image

```
print(ages > ages.mean())
```

```
0    False
1     True
```

```
2     True
3     True
4    False
5    False
6    False
7     True
Name: Age, dtype: bool
```

```
print(type(ages > ages.mean()))
```

```
<class 'pandas.core.series.Series'>
```

This statement returns a `Series` with a `dtype` of `bool`. In other words, we can not only subset values using labels and indices, but also supply a vector of boolean values. Python has many functions and methods. Depending on how it is implemented, it may return labels, indices, or booleans. Keep this point in mind as you learn new methods and seek to piece together various parts for your work.

If we liked, we could manually supply a vector of `bool`s to subset our data.

Click here to view code image

```
# get index 0, 1, 4, and 5
manual_bool_values = [True, True, False, False, True, True, False, True]
print(ages[manual_bool_values])
```

```
0    37
1    61
4    56
5    45
7    77
Name: Age, dtype: int64
```

2.3.3 Operations Are Automatically Aligned and Vectorized (Broadcasting)

If you're familiar with programming, you would find it strange that `ages > ages.mean()` returns a vector without any `for` loops (Appendix M). Many of the methods that work on `series` (and also `DataFrames`) are vectorized, meaning that they work on the entire vector simultaneously. This approach makes the code easier to read, and typically optimizations are available to make calculations faster.

2.3.3.1 Vectors of the Same Length

If you perform an operation between two vectors of the same length, the resulting vector will be an element-by-element calculation of the vectors.

```
print(ages + ages)
```

```
0     74
1    122
2    180
3    132
4    112
```

```
5     90

6     82

7    154

Name: Age, dtype: int64
```

```
print(ages * ages)
```

```
0    1369

1    3721

2    8100

3    4356

4    3136

5    2025

6    1681

7    5929

Name: Age, dtype: int64
```

2.3.3.2 Vectors With Integers (Scalars)

When you perform an operation on a vector using a scalar, the scalar will be recycled across all the elements in the vector.

```
print(ages + 100)
```

```
0    137
```

```
1    161
2    190
3    166
4    156
5    145
6    141
7    177
Name: Age, dtype: int64
```

```
print(ages * 2)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

2.3.3.3 Vectors With Different Lengths

When you are working with vectors of different lengths, the behavior will depend on the `type` of the vectors. With a `Series`, the vectors will perform an operation matched by the index. The rest of the resulting vector will be filled with a "missing" value, denoted with `NaN`, signifying "not a number."

This type of behavior, which is called broadcasting, differs between languages. Broadcasting in Pandas refers to how operations are calculated between arrays with different shapes.

Click here to view code image

```
print(ages + pd.Series([1, 100]))
```

```
0     38.0
1    161.0
2      NaN
3      NaN
4      NaN
5      NaN
6      NaN
7      NaN
dtype: float64
```

With other `types`, the shapes must match.

Click here to view code image

```
import numpy as np



# this will cause an error

print(ages + np.array([1, 100]))
```

```
Traceback (most recent call last):

  File "<ipython-input-1-daaf3fc48315>", line 2, in <module>

    print(ages + np.array([1, 100]))

ValueError: operands could not be broadcast together with shapes (8,)

(2,)
```

2.3.3.4 Vectors With Common Index Labels (Automatic Alignment)

What's cool about Pandas is how data alignment is almost always automatic. If possible, things will always align themselves with the index label when actions are performed.

Click here to view code image

```
# ages as they appear in the data

print (ages)
```

```
0    37

1    61

2    90

3    66
```

```
4    56

5    45

6    41

7    77

Name: Age, dtype: int64
```

```
rev_ages = ages.sort_index(ascending=False)

print (rev_ages)
```

```
7    77

6    41

5    45

4    56

3    66

2    90

1    61

0    37

Name: Age, dtype: int64
```

If we perform an operation using `ages` and `rev_ages`, it will still be conducted on an element-by-element basis, but the vectors will be aligned first before the operation is carried out.

Click here to view code image

```
# reference output to show index label alignment
print(ages * 2)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```
# note how we get the same values
# even though the vector is reversed
print(ages + rev_ages)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
```

```
6     82

7    154

Name: Age, dtype: int64
```

2.4 The DataFrame

The `DataFrame` is the most common Pandas object. It can be thought of as Python's way of storing spreadsheet-like data. Many of the features of the `Series` data structure carry over into the `DataFrame`.

2.4.1 Boolean Subsetting: DataFrames

Just as we were able to subset a `Series` with a boolean vector, so we can subset a `DataFrame` with a `bool`.

Click here to view code image

```
# boolean vectors will subset rows

print(scientists[scientists['Age'] > scientists['Age'].mean()])
```

```
                   Name        Born        Died  Age     Occupation

1        William Gosset  1876-06-13  1937-10-16   61   Statistician

2  Florence Nightingale  1820-05-12  1910-08-13   90          Nurse

3           Marie Curie  1867-11-07  1934-07-04   66        Chemist

7          Johann Gauss  1777-04-30  1855-02-23   77  Mathematician
```

Because of how broadcasting works, if we supply a `bool` vector that is not the same as the number of rows in the dataframe, the maximum number of rows returned would be the length of the `bool` vector.

Click here to view code image

```
# 4 values passed as a bool vector

# 3 rows returned

print(scientists.loc[[True, True, False, True]])
```

```
                Name        Born        Died  Age    Occupation
0  Rosaline Franklin  1920-07-25  1958-04-16   37       Chemist
1     William Gosset  1876-06-13  1937-10-16   61  Statistician
3        Marie Curie  1867-11-07  1934-07-04   66       Chemist
```

Table 2.3 summarizes the various subsetting methods.

Table 2.3 Table of DataFrame Subsetting Methods

Syntax	Selection Result
`df[column_name]`	Single column
`df[[column1, column2, ...]]`	Multiple columns
`df.loc[row_label]`	Row by row index label (row name)
`df.loc[[label1, label2, ...]]`	Multiple rows by index label
`df.iloc[row_number]`	Row by row number
`df.iloc[[row1, row2, ...]]`	Multiple rows by row number
`df.ix[label_or_number]`	Row by index label or number
`df.ix[[lab_num1, lab_num2, ...]]`	Multiple rows by index label or number
`df[bool]`	Row based on
`bool df[[bool1, bool2, ...]]`	Multiple rows based on
`bool df[start:stop:step]`	Rows based on slicing notation

Note that `ix` no longer works after `Pandas v0.20`.

2.4.2 Operations Are Automatically Aligned and Vectorized (Broadcasting)

Pandas supports *broadcasting*, which comes from the `numpy` library.[6] In essence, it describes what happens when performing operations between array-like objects, which the `Series` and `DataFrame` are. These behaviors depend on the type of object, its length, and any labels associated with the object.

6. `numpy` library: http://www.numpy.org/

First let's create a subset of our dataframes.

Click here to view code image

```
first_half = scientists[:4]

second_half = scientists[4:]



print (first_half)
```

```
                    Name        Born        Died  Age    Occupation

0     Rosaline Franklin  1920-07-25  1958-04-16   37       Chemist

1        William Gosset  1876-06-13  1937-10-16   61  Statistician

2  Florence Nightingale  1820-05-12  1910-08-13   90         Nurse

3           Marie Curie  1867-11-07  1934-07-04   66       Chemist
```

```
print (second_half)
```

```
            Name        Born        Died  Age           Occupation
4  Rachel Carson  1907-05-27  1964-04-14   56            Biologist
5      John Snow  1813-03-15  1858-06-16   45            Physician
6    Alan Turing  1912-06-23  1954-06-07   41  Computer Scientist
7   Johann Gauss  1777-04-30  1855-02-23   77       Mathematician
```

When we perform an action on a dataframe with a scalar, it will try to apply the operation on each cell of the dataframe. In this example, numbers will be multiplied by 2, and strings will be doubled (this is Python's normal behavior with strings).

Click here to view code image

```
# multiply by a scalar
print(scientists * 2)
```

```
                                      Name                 Born  \
0        Rosaline FranklinRosaline Franklin  1920-07-251920-07-25
1             William GossetWilliam Gosset  1876-06-131876-06-13
2  Florence NightingaleFlorence Nightingale  1820-05-121820-05-12
3                   Marie CurieMarie Curie  1867-11-071867-11-07
4               Rachel CarsonRachel Carson  1907-05-271907-05-27
5                     John SnowJohn Snow  1813-03-151813-03-15
6                   Alan TuringAlan Turing  1912-06-231912-06-23
7                 Johann GaussJohann Gauss  1777-04-301777-04-30
```

```
                    Died  Age                            Occupation
0  1958-04-161958-04-16   74                        ChemistChemist
1  1937-10-161937-10-16  122              StatisticianStatistician
2  1910-08-131910-08-13  180                           NurseNurse
3  1934-07-041934-07-04  132                        ChemistChemist
4  1964-04-141964-04-14  112                     BiologistBiologist
5  1858-06-161858-06-16   90                     PhysicianPhysician
6  1954-06-071954-06-07   82  Computer ScientistComputer Scientist
7  1855-02-231855-02-23  154            MathematicianMathematician
```

If your dataframes are all numeric values and you want to "add" the values on a cell-by-cell basis, you can use the `add` method. The automatic alignment can be better seen in Chapter 4, when we concatenate dataframes together.

2.5 Making Changes to Series and DataFrames

Now that we know various ways of subsetting and slicing our data (see Table 2.3), we should be able to alter our data objects.

2.5.1 Add Additional Columns

The `type` of the `Born` and `Died` columns is `object`, meaning they are strings.

Click here to view code image

```
print(scientists['Born'].dtype)
```

```
object
```

```
print(scientists['Died'].dtype)
```

```
object
```

We can convert the strings to a proper `datetime` type so we can perform common date and time operations (e.g., take differences between dates or calculate a person's age). You can provide your own `format` if you have a date that has a specific format. A list of `format` variables can be found in the Python `datetime` module documentation.[7] More examples with datetimes can be found in Chapter 11. The format of our date looks like "YYYY-MM-DD," so we can use the '`%Y-%m-%d`' format.

7. `datetime` module documentation: https://docs.python.org/3.5/library/datetime.html#strftime-and-strptime-behavior

Click here to view code image

```
# format the 'Born' column as a datetime
born_datetime = pd.to_datetime(scientists['Born'], format='%Y-%m-%d')
print (born_datetime)
```

```
0   1920-07-25
1   1876-06-13
2   1820-05-12
3   1867-11-07
4   1907-05-27
```

```
5   1813-03-15
6   1912-06-23
7   1777-04-30
Name: Born, dtype: datetime64[ns]
```

```
# format the 'Died' column as a datetime
died_datetime = pd.to_datetime(scientists['Died'], format='%Y-%m-%d')
```

If we wanted, we could create a new set of columns that contain the `datetime` representations of the `object` (string) dates. The below example uses python's multiple assignment syntax (Appendix Q).

Click here to view code image

```
scientists['born_dt'], scientists['died_dt'] = (born_datetime,
                                                died_datetime)
print(scientists.head())
```

```
                  Name        Born        Died  Age    Occupation  \
0     Rosaline Franklin  1920-07-25  1958-04-16   37       Chemist
1        William Gosset  1876-06-13  1937-10-16   61  Statistician
2  Florence Nightingale  1820-05-12  1910-08-13   90         Nurse
3           Marie Curie  1867-11-07  1934-07-04   66       Chemist
4         Rachel Carson  1907-05-27  1964-04-14   56     Biologist
```

```
     born_dt    died_dt
0 1920-07-25 1958-04-16
1 1876-06-13 1937-10-16
2 1820-05-12 1910-08-13
3 1867-11-07 1934-07-04
4 1907-05-27 1964-04-14
```

```
print(scientists.shape)
```

```
(8, 7)
```

2.5.2 Directly Change a Column

We can also assign a new value directly to the existing column. The example in this section shows how to randomize the contents of a column. More complex calculations that involve multiple columns can be seen in Chapter 9, in the discussion of the `apply` method.

First, let's look at the original `Age` values.

```
print(scientists['Age'])
```

```
0    37
1    61
2    90
3    66
4    56
```

```
5    45

6    41

7    77

Name: Age, dtype: int64
```

Now let's shuffle the values.

Click here to view code image

```
import random


# set a seed so the randomness is always the same

random.seed(42)

random.shuffle(scientists['Age'])
```

```
/home/dchen/anaconda3/envs/book36/lib/python3.6/random.py:274:

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

  x[i], x[j] = x[j], x[i]
```

```
print(scientists['Age'])
```

```
0    66
1    56
2    41
3    77
4    90
5    45
6    37
7    61
Name: Age, dtype: int64
```

The `SettingWithCopyWarning` message[8] in the previous code tells us that the proper way of handling the statement would be to write it using `loc`, or we can use the built-in sample method to randomly sample the length of the column.

8. Indexing view versus copy: https://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy

In this example, you need to `reset_index` since `sample` picks out only the row index. Thus, if you try to reassign it or use it again, the "scrambled" values will automatically align to the index and order themselves back to the pre-`sample` order. The `drop=True` parameter in `reset_index` tells Pandas not to insert the index into the dataframe columns, so that only the values are kept.

Click here to view code image

```
# the random_state is used to keep the 'randomization' less random
```

```
scientists['Age'] = scientists['Age'].\

    sample(len(scientists['Age']), random_state=24).\

    reset_index(drop=True) # values stay randomized



# we shuffled this column twice

print(scientists['Age'])
```

```
0    61

1    45

2    37

3    90

4    56

5    66

6    77

7    41

Name: Age, dtype: int64
```

Notice that the `random.shuffle` method seems to work directly on the column. The documentation for `random.shuffle` [9] mentions that the sequence will be shuffled "in place," meaning that it will work directly on the sequence. Contrast this with the previous method, in which we assigned the newly calculated values to a separate variable before we could assign them to the column.

---

9. Random shuffle: https://docs.python.org/3.6/library/random.html#random.shuffle

We can recalculate the “real” age using `datetime` arithmetic. More information about `datetime` can be found in Chapter 11.

Click here to view code image

```
# subtracting dates gives the number of days
scientists['age_days_dt'] = (scientists['died_dt'] - \
                             scientists['born_dt'])
print(scientists)
```

```
                  Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   90
4         Rachel Carson  1907-05-27  1964-04-14   56
5             John Snow  1813-03-15  1858-06-16   66
6           Alan Turing  1912-06-23  1954-06-07   77
7          Johann Gauss  1777-04-30  1855-02-23   41

            Occupation    born_dt    died_dt age_days_dt
0              Chemist 1920-07-25 1958-04-16  13779 days
1        Statistician 1876-06-13 1937-10-16  22404 days
2               Nurse 1820-05-12 1910-08-13  32964 days
```

```
3             Chemist 1867-11-07 1934-07-04  24345 days
4           Biologist 1907-05-27 1964-04-14  20777 days
5           Physician 1813-03-15 1858-06-16  16529 days
6  Computer Scientist 1912-06-23 1954-06-07  15324 days
7       Mathematician 1777-04-30 1855-02-23  28422 days
```

```
# we can convert the value to just the year
# using the astype method
scientists['age_years_dt'] = scientists['age_days_dt'].\
    astype('timedelta64[Y]')
print (scientists)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   90
4         Rachel Carson  1907-05-27  1964-04-14   56
5             John Snow  1813-03-15  1858-06-16   66
6           Alan Turing  1912-06-23  1954-06-07   77
7          Johann Gauss  1777-04-30  1855-02-23   41
```

```
            Occupation    born_dt    died_dt age_days_dt  \
0              Chemist 1920-07-25 1958-04-16  13779 days
1         Statistician 1876-06-13 1937-10-16  22404 days
2                Nurse 1820-05-12 1910-08-13  32964 days
3              Chemist 1867-11-07 1934-07-04  24345 days
4            Biologist 1907-05-27 1964-04-14  20777 days
5            Physician 1813-03-15 1858-06-16  16529 days
6  Computer Scientist 1912-06-23 1954-06-07  15324 days
7       Mathematician 1777-04-30 1855-02-23  28422 days

   age_years_dt
0          37.0
1          61.0
2          90.0
3          66.0
4          56.0
5          45.0
6          41.0
7          77.0
```

Many functions and methods in pandas will have an `inplace` parameter that you can set to `True`, if you want to perform the action "in place."

This will directly change the given column without returning anything.

NOTE

We could have directly assigned the column to the `datetime` that was converted, but the point is that an assignment still needed to be performed. The `random.shuffle` example performs its method "in place," so there is nothing that is explicitly returned from the function. The value passed into the function is directly manipulated.

2.5.3 Dropping Values

To drop a column, we can either select all the columns we want to by using the column subsetting techniques, or select columns to drop with the drop method on our dataframe.[10]

---

10. Drop method: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html

Click here to view code image

```
# all the current columns in our data

print(scientists.columns)
```

```
Index(['Name', 'Born', 'Died', 'Age', 'Occupation', 'born_dt',

       'died_dt', 'age_days_dt', 'age_years_dt'],

      dtype='object')
```

```
# drop the shuffled age column

# you provide the axis=1 argument to drop column-wise
```

```
scientists_dropped = scientists.drop(['Age'], axis=1)


# columns after dropping our column

print(scientists_dropped.columns)
```

```
Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt', 'died_dt',

       'age_days_dt', 'age_years_dt'],

      dtype='object')
```

2.6 Exporting and Importing Data

In our examples so far, we have been importing data. It is also common practice to export or save out data sets while processing them. Data sets are either saved out as final cleaned versions of data or in intermediate steps. Both of these outputs can be used for analysis or as input to another part of the data processing pipeline.

2.6.1 pickle

Python has a way to `pickle` data. This is Python's way of serializing and saving data in a binary format reading pickle data is also backwards compatible.

2.6.1.1 Series

Many of the export methods for a `Series` are also available for a `DataFrame`. Those readers who have experience with `numpy` will know that a `save` method is available for `ndarray`s. This method has been deprecated, and the replacement is to use the `to_pickle` method.

Click here to view code image

```
names = scientists['Name']

print(names)
```

```
0      Rosaline Franklin
1         William Gosset
2   Florence Nightingale
3            Marie Curie
4          Rachel Carson
5              John Snow
6            Alan Turing
7           Johann Gauss
Name: Name, dtype: object
```

```
# pass in a string to the path you want to save

names.to_pickle('../output/scientists_names_series.pickle')
```

The `pickle` output is in a binary format. Thus, if you try to open it in a text editor, you will see a bunch of garbled characters.

If the object you are saving is an intermediate step in a set of calculations that you want to save, or if you know that your data will stay in the Python world, saving objects to a `pickle` will be optimized for Python as well as in terms of disk storage space. However, this approach means that people who do not use Python will not be able to read the data.

2.6.1.2 DataFrame

The same method can be used on `DataFrame` objects.

Click here to view code image

```
scientists.to_pickle('../output/scientists_df.pickle')
```

2.6.1.3 Reading pickle Data

To read in `pickle` data, we can use the `pd.read_pickle` function.

Click here to view code image

```
# for a Series
scientist_names_from_pickle = pd.read_pickle(
    '../output/scientists_names_series.pickle')
print(scientist_names_from_pickle)
```

```
0       Rosaline Franklin
1          William Gosset
2    Florence Nightingale
3             Marie Curie
4           Rachel Carson
5              John Snow
6             Alan Turing
7            Johann Gauss
Name: Name, dtype: object
```

```
# for a DataFrame
scientists_from_pickle = pd.read_pickle(
```

```
    '../output/scientists_df.pickle')
print(scientists_from_pickle)
```

```
                  Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   90
4         Rachel Carson  1907-05-27  1964-04-14   56
5             John Snow  1813-03-15  1858-06-16   66
6           Alan Turing  1912-06-23  1954-06-07   77
7          Johann Gauss  1777-04-30  1855-02-23   41

            Occupation    born_dt    died_dt age_days_dt  \
0              Chemist 1920-07-25 1958-04-16  13779 days
1        Statistician 1876-06-13 1937-10-16  22404 days
2               Nurse 1820-05-12 1910-08-13  32964 days
3              Chemist 1867-11-07 1934-07-04  24345 days
4           Biologist 1907-05-27 1964-04-14  20777 days
5           Physician 1813-03-15 1858-06-16  16529 days
6  Computer Scientist 1912-06-23 1954-06-07  15324 days
```

```
7       Mathematician 1777-04-30 1855-02-23  28422 days



   age_years_dt

0          37.0

1          61.0

2          90.0

3          66.0

4          56.0

5          45.0

6          41.0

7          77.0
```

The `pickle` files are saved with an extension of `.p`, `.pkl`, or `.pickle`.

2.6.2 CSV

Comma-separated values (CSV) are the most flexible data storage type. For each row, the column information is separated with a comma. The comma is not the only type of delimiter, however. Some files are delimited by a tab (TSV) or even a semicolon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open this kind of data structure. It can even be opened in a text editor.

The `Series` and `DataFrame` have a `to_csv` method to write a CSV file. The documentation for `Series` [11] and `DataFrame` [12] identifies many different ways you can modify the resulting CSV file. For example, if you

wanted to save a TSV file because there are commas in your data, you can change the `sep` parameter (Appendix O).

11. Saving a `Series` to a CSV file: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.to_csv.html

12. Saving a `DataFrame` to a CSV file: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

Click here to view code image

```
# save a series into a CSV

names.to_csv('../output/scientist_names_series.csv')

# save a dataframe into a TSV,

# a tab-separated value

scientists.to_csv('../output/scientists_df.tsv', sep='\t')
```

2.6.2.1 Removing Row Numbers From Output

If you open the CSV or TSV file created, you will notice that the first "column" looks like the row number of the dataframe. Many times this is not needed, especially when you are collaborating with other people. Keep in mind that this "column" is really saving the "row label," which *may* be important. The documentation will show that there is an `index` parameter with which to write row names (`index`).

Click here to view code image

```
# do not write the row names in the CSV output

scientists.to_csv('../output/scientists_df_no_index.csv', index=False)
```

2.6.2.2 Importing CSV Data

Importing CSV files was illustrated in Section 1.2. This operation uses the `pd.read_csv` function. In the documentation,[13] you can see there are various ways to read in a `CSV`. Look at Appendix O if you need more information on using function parameters.

---

13. `read_csv` documentation: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

2.6.3 Excel

Excel, which is probably the most commonly used data type (or the second most commonly used, after CSVs), has a bad reputation within the data science community, mainly because colors and other superfluous information can easily find its way into the data set, not to mention one-off calculations that ruin the rectangular structure of a data set. Some other reasons of why are listed in Section 1.1. The goal of this book isn't to bash Excel, but rather to teach you about a reasonable alternative tool for data analytics. In short, the more of your work you can do in a scripting language, the easier it will be to scale up to larger projects, catch and fix mistakes, and collaborate. However, Excel's popularity and market share is unrivaled. Excel has its own scripting language if you absolutely have to work in it. This will allow you to work with data in a more predictable and reproducible manner.

2.6.3.1 Series

The `Series` data structure does not have an explicit `to_excel` method. If you have a `Series` that needs to be exported to an Excel file, one option is to convert the `Series` into a one-column `DataFrame`.

Click here to view code image

```
# convert the Series into a DataFrame

# before saving it to an Excel file
```

```
names_df = names.to_frame()


import xlwt # this needs to be installed

# xls file

names_df.to_excel('../output/scientists_names_series_df.xls')


import openpyxl # this needs to be installed

# newer xlsx file

names_df.to_excel('../output/scientists_names_series_df.xlsx')
```

2.6.3.2 DataFrame

From the preceding example, you can see how to export a `DataFrame` to an Excel file. The documentation[14] shows several ways to further fine-tune the output. For example, you can output data to a specific "sheet" using the `sheet_name` parameter.

14. `DataFrame` to Excel documentation: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_excel.html

Click here to view code image

```
# saving a DataFrame into Excel format

scientists.to_excel('../output/scientists_df.xlsx',

                    sheet_name='scientists',

                    index=False)
```

2.6.4 Feather Format to Interface With R

The format called “feather” is used to save a binary object that can also be loaded into the R language. The main benefit of this approach is that it is faster than writing and reading a CSV file between the languages. The general rule of thumb for using this data format is to use it only as an intermediate data format, and to not use the `feather` format for long-term storage. That is, use it in your code only to pass in data into R; do not use it to save a final version of your data.

The `feather` formatter is installed via `conda install -c conda-forge feather-format` or `pip install feather-format`. You can use the `to_feather` method on a dataframe to save the `feather` object. Not every dataframe can be converted into a `feather` object. For example, our current data set contains a column of `date` values, which at the time of this writing is not supported by `feather`.[15]

---

15. Feather dates, `ArrowNotImplementedError`: https://github.com/wesm/feather/issues/121

2.6.5 Other Data Output Types

There are many ways Pandas can export and import data. Indeed, `to_pickle`, `to_csv`, and `to_excel`, and `to_feather` are only some of the data formats that can make their way into Pandas `DataFrame`s. Table 2.4 lists some of these other output formats.

Table 2.4 `DataFrame` Export Methods

Export Method	Description
`to_clipboard`	Save data into the system clipboard for pasting
`to_dense`	Convert data into a regular "dense" `DataFrame`
`to_dict`	Convert data into a Python
`dict to_gbq`	Convert data into a Google BigQuery table
`to_hdf`	Save data into a hierarchal data format (HDF)
`to_msgpack`	Save data into a portable JSON-like binary
`to_html`	Convert data into a HTML table
`to_json`	Convert data into a JSON string
`to_latex`	Convert data into a L_AT_EX tabular environment
`to_records`	Convert data into a record array
`to_string`	Show `DataFrame` as a string for `stdout`
`to_sparse`	Convert data into a `SparceDataFrame`
`to_sql`	Save data into a SQL database
`to_stata`	Convert data into a Stata `dta` file

For more complicated and general data conversions (not necessarily just exporting data), the `odo` library[16] has a consistent way to convert between data formats (Appendix T).

16. `odo` library http://odo.readthedocs.org/en/latest/

2.7 Conclusion

This chapter went in a little more detail about how the Pandas `Series` and `DataFrame` objects work in Python. There were some simpler examples of data cleaning shown, along with a few common ways to export data to share with others. Chapters 1 and 2 should give you a good basis on how Pandas works as a library.

The next chapter covers the basics of plotting in Python and Pandas. Data visualization is not only used in the end of an analysis to plot results, but also is heavily utilized throughout the entire data pipeline.