MODULE 4

DATA CLEANING AND PREPARATION

Data cleaning and preparation

- Most statistical theory focuses on data modeling, prediction and statistical inference while it is usually assumed that data are in the correct state for data analysis.
- In practice, a data analyst spends most of her/his time on preparing the data before doing any statistical operation.
- It is rarely seen that the raw data are in the correct format, without errors, complete and having all the correct labels and codes that are needed for analysis.
- Data Cleaning is the process of transforming raw data into consistent data that can be analyzed. It is aimed at improving the content of statistical statements based on the data as well as their reliability.

Data cleaning and preparation

> "Data analysis is a process of inspecting, cleansing, transforming and modeling data with the goal of discovering useful information, informing conclusion and supporting decision-making." *Wikipedia, February 2020.*

> Incorrect or inconsistent data leads to false conclusions. And so, how well you clean and understand the data has a high impact on the quality of the results.

> In fact, a simple method can outweigh a complex one just because it was given enough and high-quality data.

> Quality data beats fancy models/algorithms.

Data cleaning and preparation

> Today, companies are bombarded with an exploding volume of data in corporate applications and through social media.

> Not surprisingly, data inaccuracy is at an all time high.

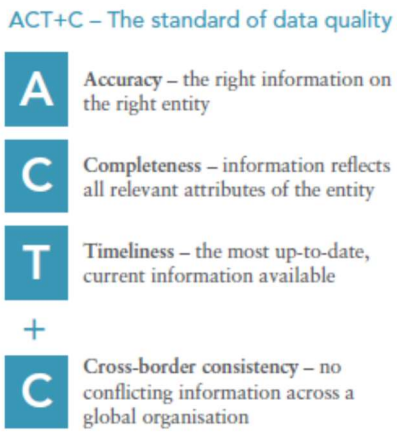
> At a time when companies are analyzing the **Return on Investment** (ROI) of every expenditure, why invest in data quality?

> Accurate and complete data throughout the organization provides a gold-mine of intelligence that can be used to strengthen competitive position.

Data cleaning and preparation

- According to The Gartner Group, poor data quality drains a company on average $8.2 million annually in squandered resources and expenses for operational inefficiencies, missed sales and unrealized new opportunities.
- In addition, a Forrester research study reveals that only 12% percent of companies use data-driven intelligence to guide key business functions and corporate strategy.
- That means that 88% are putting up with the waste, inefficiencies and lost opportunities that dirty data creates.
- When data is dirty, there is typically an underlying business process issue to address. To build a clear understanding of the current state, leading organizations perform a company-wide audit of data assets, requirements and processes that support business delivery.

Data Quality



Data Quality

- Validity
- Accuracy
- Completeness
- Consistency
- Uniformity

Validity

- The degree to which the data conform to defined business rules or constraints.
- Data-Type Constraints: values in a particular column must be of a particular datatype, e.g., boolean, numeric, date, etc.
- Range Constraints: typically, numbers or dates should fall within a certain range.
- Mandatory Constraints: certain columns cannot be empty.
- Unique Constraints: a field, or a combination of fields, must be unique across a dataset.
- Set-Membership constraints: values of a column come from a set of discrete values. For example, a person's gender may be male or female.

Validity

- Foreign-key constraints: as in relational databases, a foreign key column can't have a value that does not exist in the referenced primary key.
- Regular expression patterns: text fields that have to be in a certain pattern. For example, phone numbers may be required to have the pattern (999) 999–9999.
- Cross-field validation: certain conditions that span across multiple fields must hold. For example, a patient's date of discharge from the hospital cannot be earlier than the date of admission.

Accuracy

- The degree to which the data is close to the true values.
- While defining all possible valid values allows invalid values to be easily spotted, it does not mean that they are accurate.
- A valid street address might not actually exist. A valid person's eye color, say blue, might be valid, but not true (doesn't represent the reality).
- Another thing to note is the difference between accuracy and precision. Saying that you live on the earth is, actually true. But, not precise. Where on the earth?. Saying that you live at a particular street address is more precise.

Completeness

> The degree to which all required data is known.

> Missing data is going to happen for various reasons. One can mitigate this problem by questioning the original source, if possible, say re-interviewing the subject.

> Chances are, the subject is either going to give a different answer or will be hard to reach again.

Consistency

> The degree to which the data is consistent, within the same data set or across multiple data sets.

> Inconsistency occurs when two values in the data set contradict each other.

> A valid age, say 10, might not match with the marital status, say divorced. A customer is recorded in two different tables with two different addresses.

> Which one is true?.

Uniformity

- The degree to which the data is specified using the same unit of measure.
- The weight may be recorded either in pounds or kilos. The date might follow the USA format or European format. The currency is sometimes in USD and sometimes in TL.
- And so data must be converted to a single measure unit.

The workflow

- The workflow is a sequence of three steps aiming at producing high-quality data and taking into account all the criteria we've talked about.
- Inspection: Detect unexpected, incorrect, and inconsistent data.
- Cleaning: Fix or remove the anomalies discovered.
- Verifying: After cleaning, the results are inspected to verify correctness.
- Reporting: A report about the changes made and the quality of the currently stored data is recorded.
- What you see as a sequential process is, in fact, an iterative, endless process. One can go from verifying to inspection when new flaws are detected.

Inspection

- Inspecting the data is time-consuming and requires using many methods for exploring the underlying data for error detection.
- **Data profiling**
  - A **summary statistics** about the data, called data profiling, is really helpful to give a general idea about the quality of the data.
  - For example, check whether a particular column conforms to particular standards or pattern. Is the data column recorded as a string or number?.
  - How many values are missing?. How many unique values in a column, and their distribution?. Is this data set is linked to or have a relationship with another?.

Inspection

- **Visualizations**
  - By analyzing and visualizing the data using statistical methods such as mean, standard deviation, range, or quantiles, one can find values that are unexpected and thus erroneous.
  - For example, by visualizing the average income across the countries, one might see there are some **outliers.** Some countries have people who earn much more than anyone else. Those outliers are worth investigating and are not necessarily incorrect data.

Inspection

- **Software packages**
  - Several software packages or libraries available at your language will let you specify constraints and check the data for violation of these constraints.
  - Moreover, they can not only generate a report of which rules were violated and how many times but also create a graph of which columns are associated with which rules.
  - The age, for example, can't be negative, and so the height. Other rules may involve multiple columns in the same row, or across datasets.

Cleaning

- Data cleaning involve different techniques based on the problem and the data type. Different methods can be applied with each has its own trade-offs.
- Overall, incorrect data is either removed, corrected, or imputed.

Irrelevant data

- Irrelevant data are those that are not actually needed, and don't fit under the context of the problem we are trying to solve.
- For example, if we were analyzing data about the general health of the population, the phone number wouldn't be necessary — column-wise.
- Similarly, if you were interested in only one particular country, you don't want to include all other countries. Or, study only those patients who went to the surgery, we don't include everyone — row-wise.
- **Only if** you are sure that a piece of data is unimportant, you may drop it. Otherwise, explore the correlation matrix between feature variables.
- And even though you noticed no correlation, you should ask someone who is domain expert. You never know, a feature that seems irrelevant, could be very relevant from a domain perspective such as a clinical perspective.

Duplicates

- Duplicates are data points that are repeated in your dataset.
- It often happens when for example data are combined from different sources.
- The user may hit submit button twice thinking the form wasn't actually submitted.
- A request to online booking was submitted twice correcting wrong information that was entered accidentally in the first time.
- A common symptom is when two users have the same identity number. Or, the same article was scrapped twice.
- And therefore, they simply should be removed.

Type conversion

> Make sure numbers are stored as numerical data types. A date should be stored as a date object, or a Unix timestamp (number of seconds), and so on.

> Categorical values can be converted into and from numbers if needed.

> This can be spotted quickly by taking a peek over the data types of each column in the summary.

> A word of caution is that the values that can't be converted to the specified type should be converted to NA value (or any), with a warning being displayed. This indicates the value is incorrect and must be fixed.

Syntax errors

> Remove white spaces: Extra white spaces at the beginning or the end of a string should be removed.

" hello world " => "hello world

> Pad strings: Strings can be padded with spaces or other characters to a certain width. For example, some numerical codes are often represented with prepending zeros to ensure they always have the same number of digits.

313 => 000313 (6 digits)

Syntax errors

- Fix typos: Strings can be entered in many ways, and no wonder, can have mistakes.

  Gender
  m
  Male
  fem.
  FemalE
  Femle

- This categorical variable is considered to have 5 different classes, and not 2 as expected: male and female since each value is different.
- A bar plot is useful to visualize all the unique values. One can notice some values are different but do mean the same thing i.e. “information_technology” and “IT”. Or, perhaps, the difference is just in the capitalization i.e. “other” and “Other”.
- Therefore, our duty is to recognize from the above data whether each value is male or female. How can we do that?.

Syntax errors

- The first solution is to manually map each value to either “male” or “female”.

```
dataframe['gender'].map({'m': 'male', fem.': 'female', ...})
```

- The second solution is to use pattern match. For example, we can look for the occurrence of **m** or **M** in the gender at the beginning of the string.

```
re.sub(r"\^m\$", 'Male', 'male', flags=re.IGNORECASE)
```

> The third solution is to use fuzzy matching: An algorithm that identifies the distance between the expected string(s) and each of the given one. Its basic implementation counts how many operations are needed to turn one string into another.

Gender	male	female
m	3	5
Male	1	3
fem.	5	3
FemalE	3	2
Femle	3	1

> Furthermore, if you have a variable like a city name, where you suspect typos or similar strings should be treated the same. For example, “lisbon” can be entered as “lisboa”, “lisbona”, “Lisbon”, etc.

City	Distance from "lisbon"
lisbon	0
lisboa	1
Lisbon	1
lisbona	2
london	3
...	

> If so, then we should replace all values that mean the same thing to one unique value. In this case, replace the first 4 strings with “lisbon”.

Standardize

> Our duty is to not only recognize the typos but also put each value in the same standardized format.

> For strings, make sure all values are either in lower or upper case.

> For numerical values, make sure all values have a certain measurement unit.

> The height, for example, can be in meters and centimeters. The difference of 1 meter is considered the same as the difference of 1 centimeter. So, the task here is to convert the heights to one single unit.

> For dates, the USA version is not the same as the European version. Recording the date as a timestamp (a number of milliseconds) is not the same as recording the date as a date object.

Scaling / Transformation

> Scaling means to transform your data so that it fits within a specific scale, such as 0–100 or 0–1.
> For example, exam scores of a student can be re-scaled to be percentages (0–100) instead of GPA (0–5).
> It can also help in making certain types of data easier to plot. For example, we might want to reduce skewness to assist in plotting (when having so many outliers). The most commonly used functions are log, square root, and inverse.
> Scaling can also take place on data that has different measurement units.
> Student scores on different exams say, SAT and ACT, can't be compared since these two exams are on a different scale. The difference of 1 SAT score is considered the same as the difference of 1 ACT score. In this case, we need re-scale SAT and ACT scores to take numbers, say, between 0–1.
> By scaling, we can plot and compare different scores.

Other Issues

> **Missing values**
> **Outliers**

Reporting

- Reporting how healthy the data is, is equally important to cleaning.
- As mentioned before, software packages or libraries can generate reports of the changes made, which rules were violated, and how many times.
- In addition to logging the violations, the causes of these errors should be considered. Why did they happen in the first place?.
- Try to keep the original form of the data in a separate file!

DATA CLEANING AND PREPARATION

HANDLING MISSING DATA

Data Cleaning and Preparation

> During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging

> Such tasks are often reported to take up 80% or more of an analyst's time.

> Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk.

> Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

Handling Missing Data

> Missing data occurs commonly in many data analysis applications.

> One of the goals of pandas is to make working with missing data as painless as possible.

> For example, all of the descriptive statistics on pandas objects exclude missing data by default.

> For numeric data, pandas uses the floating-point value NaN (Not a Number) to represent missing data.

Handling Missing Data

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

Handling Missing Data

> The built-in Python None value is also treated as NA in object arrays:

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
1    False
2     True
3    False
dtype: bool
```

NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate
`fillna`	Fill in missing data with some value or using an interpolation method such as **`'ffill'`** or **`'bfill'`**
`isnull`	Return boolean values indicating which values are missing/NA
`notnull`	Negation of **`isnull`**

Filtering Out Missing Data

> There are a few ways to filter out missing data.

> While you always have the option to do it by hand using **`pandas.isnull`** and boolean indexing, the **`dropna`** can be helpful.

Filtering Out Missing Data

> On a Series, it returns the Series with only the non-null data and index values:

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

Filtering Out Missing Data

> With DataFrame objects, things are a bit more complex

> You may want to drop rows or columns that are all NA or only those containing any NAs.

Filtering Out Missing Data

> **dropna** by default drops any row containing a missing value:

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

Filtering Out Missing Data

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

Filtering Out Missing Data

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Filtering Out Missing Data

```
In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Filling In Missing Data

> Rather than filtering out missing data, you may want to fill in the “holes” in any number of ways.

> For most purposes, the **`fillna`** method is the workhorse function to use.

> Calling **`fillna`** with a constant replaces missing values with that value:

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Filling In Missing Data

> Calling **`fillna`** with a dict, you can use a different fill value for each column:

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Filling In Missing Data

> **`fillna`** returns a new object, but you can modify the existing object in-place:

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Filling In Missing Data

> Interpolation methods

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232
```

Filling In Missing Data

> Interpolation methods

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

Filling In Missing Data

> Interpolate with the mean or median value of a Series

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

`fillna` function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default **`'ffill'`** if function called with no other arguments
`axis`	Axis to fill on; default **`axis=0`**
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

Removing Duplicates

> Duplicate rows may be found in a DataFrame for any number of reasons

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})

In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

Removing Duplicates

> The DataFrame method **`duplicated`** returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

```
In [47]: data.duplicated()
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Removing Duplicates

> **`drop_duplicates`** returns a DataFrame where the duplicated array is False:

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

Removing Duplicates

> Alternatively, you can specify any subset of them to detect duplicates

```
In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

Removing Duplicates

> **`duplicated`** and **`drop_duplicates`** by default keep the first observed value combination.

> Passing keep='last' will return the last one:

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

Transforming Data Using a Function or Mapping

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
   ....:                               'Pastrami', 'corned beef', 'Bacon',
   ....:                               'pastrami', 'honey ham', 'nova lox'],
   ....:                      'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

Transforming Data Using a Function or Mapping

> Suppose you wanted to add a column indicating the type of animal that each food came from.

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

Transforming Data Using a Function or Mapping

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_animal)
```

Transforming Data Using a Function or Mapping

```
In [58]: data
Out[58]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

Transforming Data Using a Function or Mapping

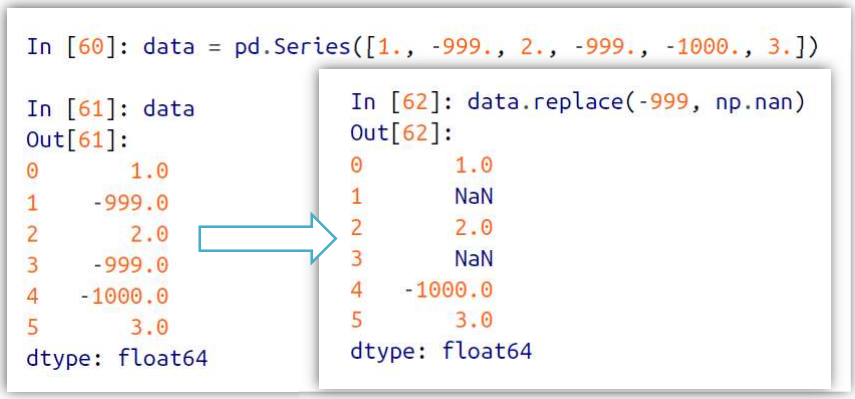
> We could also have passed a function that does all the work:

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

Replacing Values

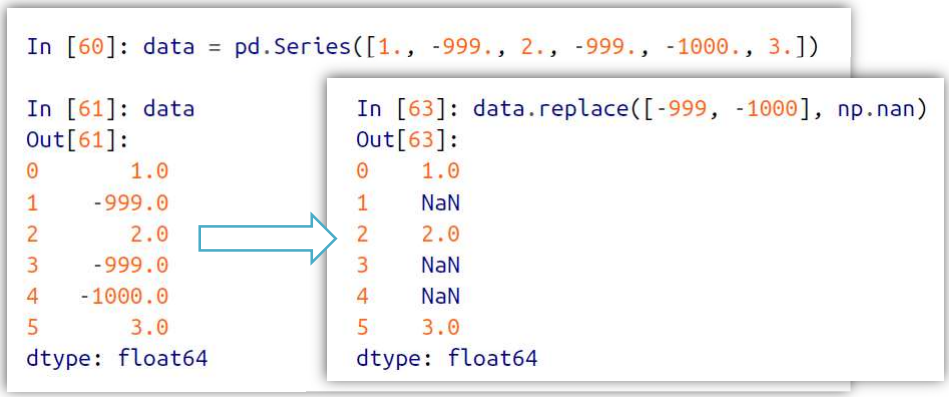
> Filling in missing data with the **`fillna`** method is a special case of more general value replacement

> **`map`** can be used to modify a subset of values in an object but replace provides a simpler and more flexible way to do so.



Replacing Values

> If you want to replace multiple values at once, you instead pass a list and then the substitute value:



Replacing Values

> To use a different replacement for each value, pass a list of substitutes:

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Replacing Values

> The argument passed can also be a dict:

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Renaming Axis Indexes

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=['Ohio', 'Colorado', 'New York'],
   ....:                     columns=['one', 'two', 'three', 'four'])
```

```
In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

Renaming Axis Indexes

> If you want to create a transformed version of a dataset without modifying the original, a useful method is **`rename`**:

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

Renaming Axis Indexes

> Notably, **`rename`** can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

Renaming Axis Indexes

> **`rename`** saves you from the chore of copying the DataFrame manually and assigning to its **`index`** and **`columns`** attributes.

> If you wish to modify a dataset in-place, pass **`inplace=True`**:

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [74]: data
Out[74]:
         one  two  three  four
INDIANA    0    1      2     3
COLO       4    5      6     7
NEW        8    9     10    11
```

Discretization and Binning

> Continuous data is often discretized or otherwise separated into “bins” for analysis.

> Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Discretization and Binning

> Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

> Let’s divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older.

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

Discretization and Binning

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Detecting and Filtering Outliers

> Filtering or transforming outliers is largely a matter of applying array operations

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.049091     0.026112    -0.002544    -0.051827
std       0.996947     1.007458     0.995232     0.998311
min      -3.645860    -3.184377    -3.745356    -3.428254
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.525865     2.735527     3.366626
```

Detecting and Filtering Outliers

> Suppose you wanted to find values in one of the columns exceeding **3** in ***absolute value***:

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
41    -3.399312
136   -3.745356
Name: 2, dtype: float64
```

Detecting and Filtering Outliers

> To select all rows having a value exceeding 3 or –3, you can use the **any** method on a boolean DataFrame

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

Detecting and Filtering Outliers

> Values can be set based on these criteria. Here is code to cap values outside the interval –3 to 3:

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.000000     2.735527     3.000000
```

Detecting and Filtering Outliers

> The statement **`np.sign(data)`** produces **`1`** and **`-1`** values based on whether the values in data are positive or negative:

```
In [99]: np.sign(data).head()
Out[99]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

Permutation and Random Sampling

> Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the **`numpy.random.permutation`** function.

> Calling **`permutation`** with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

Permutation and Random Sampling

> That array can then be used in iloc-based indexing or the equivalent **`take`** function

```
In [103]: df
Out[103]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

Permutation and Random Sampling

> To select a random subset without replacement, you can use the **`sample`** method on Series and DataFrame:

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

Permutation and Random Sampling

> To generate a sample *with* replacement (to allow repeat choices), pass **`replace=True`** to sample:

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)

In [108]: draws
Out[108]:
4    4
1    7
4    4
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
dtype: int64
```

Computing Indicator/Dummy Variables

> Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a “dummy” or “indicator” matrix.

> If a column in a DataFrame has k distinct values, you would derive a matrix or DataFrame with k columns containing all 1s and 0s.

> pandas has a **`get_dummies`** function for doing this

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],'data1': range(6)})
In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Computing Indicator/Dummy Variables

> In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other **`data.get_dummies`** has a prefix argument for doing this:

```
In [111]: dummies = pd.get_dummies(df['key'], prefix='key')

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

Computing Indicator/Dummy Variables

```
In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
   .....:                        header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy
5         6                         Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8                 Tom and Huck (1995)          Adventure|Children's
8         9                 Sudden Death (1995)                        Action
9        10                    GoldenEye (1995)     Action|Adventure|Thriller
```

Computing Indicator/Dummy Variables

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
```

```
In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

Computing Indicator/Dummy Variables

```
In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

```
for i, gen in enumerate(movies.genres):
    indices = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indices] = 1
```

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))
In [128]: movies_windic.iloc[0]
movie_id                                    1
title                         Toy Story (1995)
genres              Animation|Children's|Comedy
Genre_Animation                             1
Genre_Children's                            1
Genre_Comedy                                1
Genre_Adventure                             0
Genre_Fantasy                               0
Genre_Romance                               0
Genre_Drama                                 0
                                          ...
Genre_Crime                                 0
Genre_Thriller                              0
Genre_Horror                                0
Genre_Sci-Fi                                0
Genre_Documentary                           0
Genre_War                                   0
Genre_Musical                               0
Genre_Mystery                               0
Genre_Film-Noir                             0
Genre_Western                               0
Name: 0, Length: 21, dtype: object
```

DATA CLEANING AND PREPARATION

STRING MANIPULATION

String Object Methods

- Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing.
- Most text operations are made simple with the string object's built-in methods.
- For more complex pattern matching and text manipulations, regular expressions may be needed.
- pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

String Object Methods

> In many string munging and scripting applications, built-in string methods are sufficient

```
In [134]: val = 'a,b,  guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', '  guido']
```

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
Out[139]: 'a::b::guido'
```

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

A faster and more Pythonic way

String Object Methods

```
val = 'a,b,  guido'
```

> Locating substrings

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1

In [143]: val.find(':')
Out[143]: -1
```

```
In [145]: val.count(',')
Out[145]: 2
```

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b::  guido'

In [147]: val.replace(',', '')
Out[147]: 'ab  guido'
```

```
In [144]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

String Object Methods

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string
`endswith`	Returns **`True`** if string ends with suffix
`startswith`	Returns **`True`** if string starts with prefix
`join`	Use string as delimiter for concatenating a sequence of other strings
`index`	Return position of first character in substring if found in the string; raises **`ValueError`** if not found
`find`	Return position of first character of *first* occurrence of substring in the string; like index, but returns –1 if not found
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found
`replace`	Replace occurrences of string with another string

String Object Methods

Argument	Description
`strip`, **`rstrip`**, **`lstrip`**	Trim whitespace, including newlines; equivalent to **`x.strip()`** (and **`rstrip`**, **`lstrip`**, respectively) for each element
`split`	Break string into list of substrings using passed delimiter
`lower`	Convert alphabet characters to lowercase
`upper`	Convert alphabet characters to uppercase
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form
`ljust`, **`rjust`**	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width

Regular Expressions

> *Regular expressions* provide a flexible way to search or match (often more complex) string patterns in text.

> A single expression, commonly called a *regex*, is a string formed according to the regular expression language.

> Python's built-in **`re`** module is responsible for applying regular expressions to strings

> The **`re`** module functions fall into three categories: pattern matching, substitution, and splitting

> Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes

Regular Expressions

```
In [148]: import re

In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

```
In [153]: regex.findall(text)
Out[153]: ['    ', '\t ', '  \t']
```

Regular Expressions

```
import re
```

```
text = """Jack jack@google.com
James james@gmail.com
Kate kate@gmail.com
Jin jin@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

```
regex.findall(text)
```

```
['jack@google.com', 'james@gmail.com', 'kate@gmail.com', 'jin@yahoo.com']
```

Regular Expressions

```
regex.findall(text)
```

```
['jack@google.com', 'james@gmail.com', 'kate@gmail.com', 'jin@yahoo.com']
```

```
m = regex.search(text)
```

```
m
```

```
<re.Match object; span=(5, 20), match='jack@google.com'>
```

```
text[m.start():m.end()]
```

```
'jack@google.com'
```

Regular expression methods

Argument	Description
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like **`findall`**, but returns an iterator
`match`	Match **`pattern`** at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise **None**
`search`	Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub`, **`subn`**	Replace all (`sub`) or first n occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string

Vectorized String Functions in pandas

> Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [168]: data = pd.Series(data)

In [169]: data
Out[169]:
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object
```

```
In [170]: data.isnull()
Out[170]:
Dave     False
Rob      False
Steve    False
Wes       True
dtype: bool
```

Vectorized String Functions in pandas

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave     [(dave, google, com)]
Rob        [(rob, gmail, com)]
Steve    [(steve, gmail, com)]
Wes                        NaN
dtype: object
```

Vectorized String Functions in pandas

```
In [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave     True
Rob      True
Steve    True
Wes       NaN
dtype: object
```

Partial listing of vectorized string methods

Argument	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to **`x.endswith(pattern)`** for each element
`startswith`	Equivalent to **`x.startswith(pattern)`** for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`

Partial listing of vectorized string methods

Argument	Description
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower`, **`upper`**	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element

DATA CLEANING AND PREPARATION

DATA WRANGLING: JOIN, COMBINE, AND RESHAPE

Data Wrangling: Join, Combine, and Reshape

> In many applications, data may be spread across a number of files or databases or be arranged in a form that is not easy to analyze.

> This module focuses on tools to help combine, join, and rearrange data.

HIERARCHICAL INDEXING

Hierarchical Indexing

- *Hierarchical indexing* is an important feature of pandas that enables you to have multiple (two or more) index *levels* on an axis.
- It provides a way for you to work with higher dimensional data in a lower dimensional form.
- Create a Series with a list of lists (or arrays) as the index:

Hierarchical Indexing

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

Hierarchical Indexing

> With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64
```

```
In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

Hierarchical Indexing

> Selection is even possible from an “inner” level

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

Hierarchical Indexing

> Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table.

> For example, you could rearrange the data into a DataFrame using its **`unstack`** method:

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

Hierarchical Indexing

> The inverse operation of **`unstack`** is **`stack`**:

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

Hierarchical Indexing

> With a DataFrame, either axis can have a hierarchical index:

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....:                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   ....:                      columns=[['Ohio', 'Ohio', 'Colorado'],
   ....:                               ['Green', 'Red', 'Green']])

In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

Hierarchical Indexing

> The hierarchical levels can have names

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

Hierarchical Indexing

> With partial column indexing you can similarly select groups of columns

```
In [23]: frame['Ohio']
Out[23]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

Reordering and Sorting Levels

> At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level.

> The **`swaplevel`** takes two level numbers or names and returns a new object with the levels interchanged

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

Reordering and Sorting Levels

> **`sort_index`**, on the other hand, sorts the data using only the values in a single level.

> When swapping levels, it's not uncommon to also use **`sort_index`** so that the result is lexicographically sorted by the indicated level:

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
b    1        6   7        8
a    2        3   4        5
b    2        9  10       11
```

```
In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
     b        6   7        8
2    a        3   4        5
     b        9  10       11
```

Summary Statistics by Level

> Many descriptive and summary statistics on DataFrame and Series have a level option in which you can specify the level you want to aggregate by on a particular axis.

```
In [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1          6   8        10
2         12  14        16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

Indexing with a DataFrame's columns

> You may want to use one or more columns from a DataFrame as the row index

> You may also wish to move the row index into the DataFrame's columns.

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
   ....:                       'c': ['one', 'one', 'one', 'two', 'two',
   ....:                             'two', 'two'],
   ....:                       'd': [0, 1, 2, 0, 1, 2, 3]})

In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

Indexing with a DataFrame's columns

> DataFrame's **`set_index`** function will create a new DataFrame using one or more of its columns as the index:

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

Indexing with a DataFrame's columns

> By default the columns are removed from the DataFrame, though you can leave them in:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

Indexing with a DataFrame's columns

> **`reset_index`**, on the other hand, does the opposite of **`set_index`**

> The hierarchical index levels are moved into the columns:

```
In [34]: frame2.reset_index()
Out[34]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

COMBINING AND MERGING DATASETS

Combining and Merging Datasets

> Data contained in pandas objects can be combined together in a number of ways:
  - **`pandas.merge`** connects rows in DataFrames based on one or more keys.
    - This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
  - **`pandas.concat`** concatenates or "stacks" together objects along an axis.
  - The **`combine_first`** instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

Database-Style DataFrame Joins

> *Merge* or *join* operations combine datasets by linking rows using one or more *keys*.

> These operations are central to relational databases (e.g., SQL-based).

> The merge function in pandas is the main entry point for using these algorithms on your data.

Database-Style DataFrame Joins

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
    ....:                    'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
    ....:                    'data2': range(3)})
```

```
In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b
```

```
In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

Database-Style DataFrame Joins

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

> This is an example of a ***many-to-one*** join; the data in **`df1`** has multiple rows labeled **a** and **b**, whereas **`df2`** has only one row for each value in the key column.

Database-Style DataFrame Joins

> Note that we didn't specify which column to join on.

> If that information is not specified, merge uses the overlapping column names as the keys.

> It's a good practice to specify explicitly:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

Database-Style DataFrame Joins

> If the column names are different in each object, you can specify them separately:

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

Database-Style DataFrame Joins

> You may notice that the `'c'` and `'d'` values and associated data are missing from the result.

> By default `merge` does an `'inner'` join

  – the keys in the result are the intersection, or the common set found in both tables.

> Other possible options are `'left'`, `'right'`, and `'outer'`.

Database-Style DataFrame Joins

> The outer join takes the union of the keys, combining the effect of applying both left and right joins

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
7    NaN   d    2.0
```

Database-Style DataFrame Joins

> *Many-to-many* merges have well-defined, though not necessarily intuitive, behavior.

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                     'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                     'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b
```

```
In [48]: df2
Out[48]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
4      4   d
```

Database-Style DataFrame Joins

> Many-to-many joins form the Cartesian product of the rows.

> Since there were three `'b'` rows in the left DataFrame and two in the right one, there are six `'b'` rows in the result.

```
In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key  data2
0       0   b    1.0
1       0   b    3.0
2       1   b    1.0
3       1   b    3.0
4       2   a    0.0
5       2   a    2.0
6       3   c    NaN
7       4   a    0.0
8       4   a    2.0
9       5   b    1.0
10      5   b    3.0
```

Database-Style DataFrame Joins

> The join method only affects the distinct key values appearing in the result

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1 key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

Database-Style DataFrame Joins

> To merge with multiple keys, pass a list of column names:

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                      'key2': ['one', 'two', 'one'],
   ....:                      'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                       'key2': ['one', 'one', 'one', 'two'],
   ....:                       'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[53]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

Database-Style DataFrame Joins

> A last issue to consider in merge operations is the treatment of overlapping column names.

> merge has a **`suffixes`** option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar    one     3    one     6
5  bar    one     3    two     7
```

Database-Style DataFrame Joins

> A last issue to consider in merge operations is the treatment of overlapping column names.

> merge has a **`suffixes`** option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[55]:
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
1  foo       one     1        one     5
2  foo       two     2        one     4
3  foo       two     2        one     5
4  bar       one     3        one     6
5  bar       one     3        two     7
```

Different join types with how argument

Option	Behavior
`'inner'`	Use only the key combinations observed in both tables
`'left'`	Use all key combinations found in the left table
`'right'`	Use all key combinations found in the right table
`'outer'`	Use all key combinations observed in both tables together

`merge` function arguments

Argument	Description
`left`	DataFrame to be merged on the left side.
`right`	DataFrame to be merged on the right side
`how`	One of `'inner'`, `'outer'`, `'left'`, or `'right'`; defaults to `'inner'`
`on`	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys.
`left_on`	Columns in `left` DataFrame to use as join keys.
`right_on`	Analogous to `left_on` for `left` DataFrame.
`left_index`	Use row index in `left` as its join key (or keys, if a MultiIndex).
`right_index`	Analogous to `left_index`
`sort`	Sort merged data lexicographically by join keys; True by default (disable to get better performance in some cases on large datasets).

`merge` function arguments

Argument	Description
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to (**`'_x'`**, **`'_y'`**) (e.g., if **`'data'`** in both DataFrame objects, would appear as **`'data_x'`** and **`'data_y'`** in result).
`copy`	If **`False`**, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
`indicator`	Adds a special column **`_merge`** that indicates the source of each row; values will be **`'left_only'`**, **`'right_only'`**, or **`'both'`** based on the origin of the joined data in each row.

Merging on Index

> In some cases, the merge key(s) in a DataFrame will be found in its index.

> In this case, you can pass **`left_index=True`** or **`right_index=True`** (or both) to indicate that the index should be used as the merge key:

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                       'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [58]: left1
Out[58]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5
```

```
In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0
```

Merging on Index

```
In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

Merging on Index

> Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

Merging on Index

> With hierarchically indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge:

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....:                                'Nevada', 'Nevada'],
   ....:                       'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                       'data': np.arange(5.)})

In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....:                       index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
   ....:                               'Ohio', 'Ohio'],
   ....:                              [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                       columns=['event1', 'event2'])
```

Merging on Index

```
In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002
```

```
In [65]: righth
Out[65]:
                  event1  event2
Nevada 2001            0       1
       2000            2       3
Ohio   2000            4       5
       2000            6       7
       2001            8       9
       2002           10      11
```

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000       4       5
0   0.0    Ohio  2000       6       7
1   1.0    Ohio  2001       8       9
2   2.0    Ohio  2002      10      11
3   3.0  Nevada  2001       0       1
```

```
In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002
```

```
In [65]: righth
Out[65]:
                event1  event2
Nevada 2001          0       1
       2000          2       3
Ohio   2000          4       5
       2000          6       7
       2001          8       9
       2002         10      11
```

```
In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:          right_index=True, how='outer')
Out[67]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000     4.0     5.0
0   0.0    Ohio  2000     6.0     7.0
1   1.0    Ohio  2001     8.0     9.0
2   2.0    Ohio  2002    10.0    11.0
3   3.0  Nevada  2001     0.0     1.0
4   4.0  Nevada  2002     NaN     NaN
4   NaN  Nevada  2000     2.0     3.0
```

Merging on Index

> Using the indexes of both sides of the merge is also possible

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=['a', 'c', 'e'],
   ....:                      columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                       index=['b', 'c', 'd', 'e'],
   ....:                       columns=['Missouri', 'Alabama'])
```

```
In [70]: left2
Out[70]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0
```

```
In [71]: right2
Out[71]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0
```

Merging on Index

> Using the indexes of both sides of the merge is also possible

```
In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[72]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

Merging on Index

> DataFrame has a convenient **`join`** instance for merging by index.

> It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

Merging on Index

> In part for legacy reasons (i.e., much earlier versions of pandas), DataFrame's **`join`** method performs a left join on the join keys, exactly preserving the left frame's row index.

> It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [74]: left1.join(right1, on='key')
Out[74]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

Merging on Index

> For simple index-on-index merges, you can pass a list of DataFrames to **`join`** as an alternative to using the more general **`concat`** function

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:                        index=['a', 'c', 'e', 'f'],
   ....:                        columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0
```

Merging on Index

```
In [77]: left2.join([right2, another])
Out[77]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
b   NaN     NaN       7.0      8.0       NaN     NaN
c   3.0     4.0       9.0     10.0       9.0    10.0
d   NaN     NaN      11.0     12.0       NaN     NaN
e   5.0     6.0      13.0     14.0      11.0    12.0
f   NaN     NaN       NaN      NaN      16.0    17.0
```

Concatenating Along an Axis

> Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking.

> NumPy's **`concatenate`** function can do this with NumPy arrays:

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

Concatenating Along an Axis

> In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation.

> In particular, you have a number of additional things to think about:

– If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?

– Do the concatenated chunks of data need to be identifiable in the resulting object?

– Does the "concatenation axis" contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation.

Concatenating Along an Axis

> The **`concat`** function in pandas provides a consistent way to address these concerns.

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

Concatenating Along an Axis

> By default **`concat`** works along **`axis=0`**, producing another Series. If you pass **`axis=1`**, the result will instead be a DataFrame (**`axis=1`** is the columns):

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

Concatenating Along an Axis

> In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes.

> You can instead intersect them by passing **`join='inner'`**

```
In [87]: s4 = pd.concat([s1, s3])

In [88]: s4
Out[88]:
a    0
b    1
f    5
g    6
dtype: int64
```

```
In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6
```

```
In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
   0  1
a  0  0
b  1  1
```

Concatenating Along an Axis

> You can even specify the axes to be used on the other axes with **`join_axes`**:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[91]:
     0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

Concatenating Along an Axis

> The same logic extends to DataFrame objects:

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....:                    columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....:                    columns=['three', 'four'])

In [98]: df1
Out[98]:
   one  two
a    0    1
b    2    3
c    4    5

In [99]: df2
Out[99]:
   three  four
a      5     6
c      7     8
```

```
In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

Concatenating Along an Axis

> A last consideration concerns DataFrames in which the row index does not contain any relevant data:

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

Concatenating Along an Axis

Argument	Description
`levels`	Specific indexes to use as hierarchical index level or levels if keys passed
`names`	Names for created hierarchical levels if **`keys`** and/or **`levels`** passed
`verify_integrity`	Check new axis in concatenated object for duplicates and raise exception if so; by default (**`False`**) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation **axis**, instead producing a new **`range(total_length)`** index

RESHAPING WITH HIERARCHICAL INDEXING

Reshaping with Hierarchical Indexing

> There are a number of basic operations for rearranging tabular data.

> These are alternatingly referred to as *reshape* or *pivot* operations.

Reshaping with Hierarchical Indexing

> Hierarchical indexing provides a consistent way to rearrange data in a DataFrame.

> There are two primary actions:

- **`stack`**
  - This "rotates" or pivots from the columns in the data to the rows
- **`unstack`**
  - This pivots from the rows into the columns

Reshaping with Hierarchical Indexing

> Example:

```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                     index=pd.Index(['Ohio', 'Colorado'], name='state'),
   .....:                     columns=pd.Index(['one', 'two', 'three'],
   .....:                     name='number'))

In [121]: data
Out[121]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Reshaping with Hierarchical Indexing

> Using the **`stack`** method on this data pivots the columns into the rows, producing a Series:

```
In [122]: result = data.stack()

In [123]: result
Out[123]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

Reshaping with Hierarchical Indexing

> From a hierarchically indexed Series, you can rearrange the data back into a Data-Frame with **`unstack`**:

```
In [124]: result.unstack()
Out[124]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Reshaping with Hierarchical Indexing

> By default the innermost level is unstacked (same with stack).

> You can unstack a different level by passing a level number or name:

```
In [125]: result.unstack(0)
Out[125]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [126]: result.unstack('state')
Out[126]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

Reshaping with Hierarchical Indexing

> Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [130]: data2
Out[130]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

```
In [131]: data2.unstack()
Out[131]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

Reshaping with Hierarchical Indexing

> Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [132]: data2.unstack()
Out[132]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

```
In [133]: data2.unstack().stack()
Out[133]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64
```

Reshaping with Hierarchical Indexing

> Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [134]: data2.unstack().stack(dropna=False)
Out[134]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

Reshaping with Hierarchical Indexing

> When you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},
   .....:                   columns=pd.Index(['left', 'right'], name='side'))

In [136]: df
Out[136]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10
```

```
In [137]: df.unstack('state')
Out[137]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

Reshaping with Hierarchical Indexing

> When calling stack, we can indicate the name of the axis to stack:

```
In [138]: df.unstack('state').stack('side')
Out[138]:
state         Colorado  Ohio
number side
one    left          3     0
       right         8     5
two    left          4     1
       right         9     6
three  left          5     2
       right        10     7
```

Pivoting “Long” to “Wide” Format

> A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format:

```
In [139]: data = pd.read_csv('examples/macrodata.csv')

In [140]: data.head()
Out[140]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi  \
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5  29.54
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19
```

Pivoting “Long” to “Wide” Format

```
In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....:                          name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end')

In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

Pivoting “Long” to “Wide” Format

```
In [146]: ldata[:10]
Out[146]:
        date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

Pivoting “Wide” to “Long” Format

> An inverse **operation** to pivot for DataFrames is **pandas.melt**.

> Rather than transforming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input.

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

Pivoting “Wide” to “Long” Format

> `'key'` column may be a group indicator, and other columns are data values

> When using `pandas.melt`, we must indicate which columns (if any) are group indicators.

> Let’s use `'key'` as the only group indicator here:

```
In [159]: melted = pd.melt(df, ['key'])
```

```
In [160]: melted
Out[160]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

Pivoting “Wide” to “Long” Format

> Using pivot, we can reshape back to the original layout:

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

Pivoting “Wide” to “Long” Format

> Since the result of **`pivot`** creates an index from the column used as the row labels, we may want to use **`reset_index`** to move the data back into a column:

```
In [163]: reshaped.reset_index()
Out[163]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

Pivoting “Wide” to “Long” Format

> You can also specify a subset of columns to use as value columns:

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

Pivoting “Wide” to “Long” Format

> **`pandas.melt`** can be used without any group identifiers, too:

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
  variable  value
0        A      1
1        A      2
2        A      3
3        B      4
4        B      5
5        B      6
6        C      7
7        C      8
8        C      9
```

Pivoting “Wide” to “Long” Format

```
In [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[166]:
  variable value
0      key   foo
1      key   bar
2      key   baz
3        A     1
4        A     2
5        A     3
6        B     4
7        B     5
8        B     6
```