DATA 200

Database Systems and Data Management for Data Analytics

Professor Dick Forrester

Do Not Distribute

Spring 2022

Notice

- These course notes are primarily based upon the textbook "Introduction to Data Systems," by Bressoud and White. As such, the material is based on copyrighted material.

- The notes are intended to be used in conjunction with the course textbook, which you are required to purchase/rent as a student in this course.

- You are not allowed to distribute these notes to anyone. If you intentionally or unintentionally give access to these notes through post, social media, e-mail, or any other form of distribution, you will be in violation of copyright and the terms of acceptable use by which you are gaining access to these notes.

Chapter 1

- A Broad View of Data Systems
- The Sources of Data
- The Forms of Data

Broad View of Data Systems

- **Def:** A ***system*** is defined as
  - A regularly interacting or interdependent group of items forming a unified whole
  - An organization forming a network for serving a common purpose
- A system incorporates multiple interdependent pieces that are organized and work together to serve a common purpose.

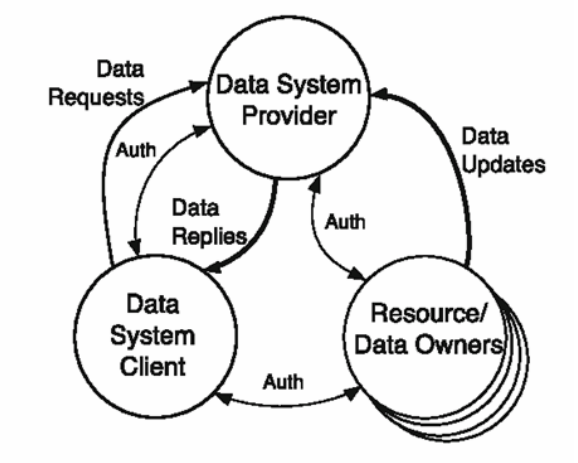
Broad View of Data Systems

- **Def:** ***Data systems*** are those systems whose focus is on the organization and structure of data, and the means by which such data can be obtained from the *provider* of the data by the *user* or *client* of the data.
- Most common is the *relational database system.*
- Data may be accessible in local data repositories or over the internet (cloud).
- Data can be static, can be dynamic, and in real-time as a stream of ever-changing data.

Data Systems Principals

- **Data System Provider:** Entity, consisting of hardware, software, and possible network infrastructure, that hosts the data, provides organization, and allows access through a well-defined interface.
- **Data System Client:** Local software and hardware whose goal is to acquire data from one or more providers and to manage and transform the data to make it useful for analysis.
- **Resource Owner:** Entity that owns the resource (data, organizational resources, etc).

Data Systems Principals



Example

- Suppose a social scientist wrote software to gather demographic information from The Movie Database (TMDB) of users and their top-rated movies.
- Software is the Data System Client (referred to as “client”).
- The users are the Resource Owners.
- TMBD is the Data System Provider (referred to as “provider”).

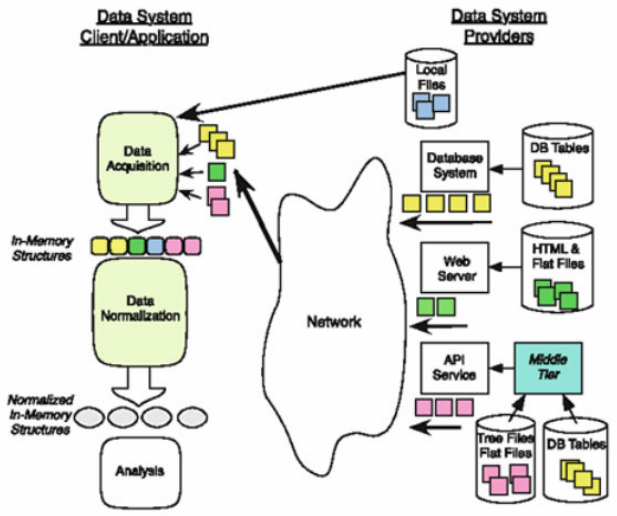
Focus of this Course

- We focus primarily on two complimentary subject areas of

1. The structure, operations, and constraints that govern how data is organized and represented, called *data models*.

2. The aspects of writing software to interact with providers of data systems, which requires understanding of *data sources* and how to interface with each.

Data System Architecture



Sources of Data

- **Local files**: files downloaded or present in the file system.
  - Individual files have a format that specifies the structure of the data
  - Common formats include `.txt`, `.docx`, `.csv`, and `.html`.
- **Relational databases**: provides data in the form of relational tables in response to *queries*.
  - Example systems include servers running Oracle, MySQL, and PostgreSQL.

Sources of Data Continued

- **Web servers**: includes any location on the Internet accessible from a web browser
  - Data system clients, using protocols, access data contained in flat files, such as comma-separated value (CSV), HyperText Markup Language (HTML), etc.

Sources of Data Continued

- **API Service Providers**: They provide a network-based Application Programming Interface (API) to allow clients to access their data over the network.
  - When requests arrive from a client to an API service, the request is processed and the data result is returned to the client, without the client needed to know the specifics of the implementation.
  - API service-based providers are the fastest growing segment of data systems and include everything from social media to government organizations.
  - Some providers charge for their services, others are free.
  - The data may be open or may be protected.

API Service Provider Examples

Table 1.1 API Service Provider Examples

Category	Example API Service Providers
Social Media	Twitter, LinkedIn, Facebook, Reddit, Slack, Tumblr
Sports/News	NBA Stats, New York Times, NPR One
Music/Video	Spotify, YouTube, The Movie Database
Health	Fitbit, RunKeeper
Support Data	Oxford Dictionary, GeoNames geolocation services, Yelp, Wikipedia, NASA
Government	Data.gov, Census.gov, EPA, Federal Register

Forms of Data

- The form of data acquired from an API can vary greatly.
  - Data could be organized in rows and columns in a tabular form.
  - Could be organized in a hierarchical tree structure, like eXtensible Markup Language (XML) or JavaScript Object Notation (JSON).
- Database queries always return data in organized in a relational table of rows and columns.

Data Form

- Our data models define the following for each form of data.

1. The ***structure*** used for data employing the particular model.
2. The ***operations*** that may be executed against data structed as given by the model.
3. The ***constraints*** that govern valid values for data in the model, as well as properties to be held invariant in and among the data represented in the model.

The Three Main Data Models

- In this course we will cover the three main data models:
  - Tabular data model
  - Relational data model
  - Hierarchical data model

Tabular Data

- *Structured* in rows and columns.
- *Constraints* ensure, for example, that there are no repeated rows and that columns represent individual variables.
- *Tabular operations* allow us to access subsets of data, to merge multiple tables together, and to mutate tables to meet logical constraints.

Relational Data Model

- Consists of database tables with additional structure, facilitating the use of database queries.
- The structure facilitates relationships between tables, so that an organization can represent complex data using multiple tables, one for each type of data the organization wishes to keep
- The constraints ensure sound database design, that all entries in a column have the same type, and that tables are appropriately linked to facilitate queries involving data spread across multiple tables (for example, data on both students and courses they are registered for).

Hierarchical Data Model

- Structured into the shape of a tree, so that some data entries are stored as children of other entries.
  - An example is the file system on your computer, where files are like children, stored in folders.
- The constraints ensure that the representation of the data really forms a tree, can constrain the shape of the tree, and can constrain the data types allowed at various places in the tree.
- The operations allow us to traverse that tree to locate and extract data stored at different places, or to create new tress and insert data.

End Goal

- The end goal of a data system client is not just to *acquire* the data, but to be able to *use* the data for some downstream objective, such as exploration, visualization, or analysis.

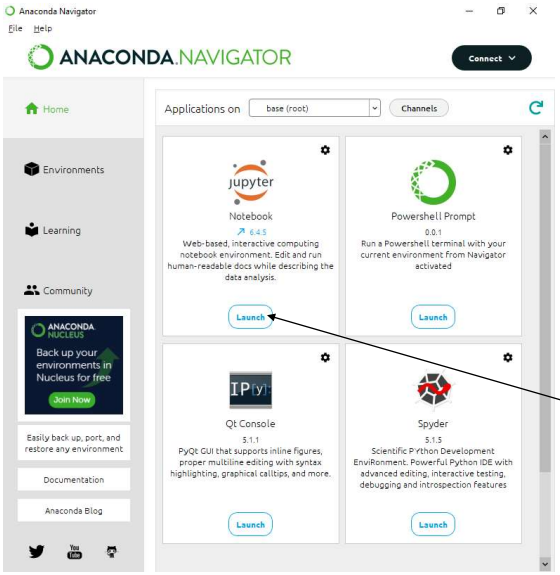
- If acquired data is not in the proper form, and does not follow the constraints of that form, then such downstream uses of data become difficult, requiring custom programming and case-by-case software adaption.

Jupyter Notebook: An Introduction

- The Jupyter Notebook is an open-source web application that you allows you to author engaging documents that combine live-code with narrative text, equations, images, video, and visualizations.

- By encoding a complete and reproducible record of computation, the documents can be shared with others.

Starting the Jupyter Notebook Server



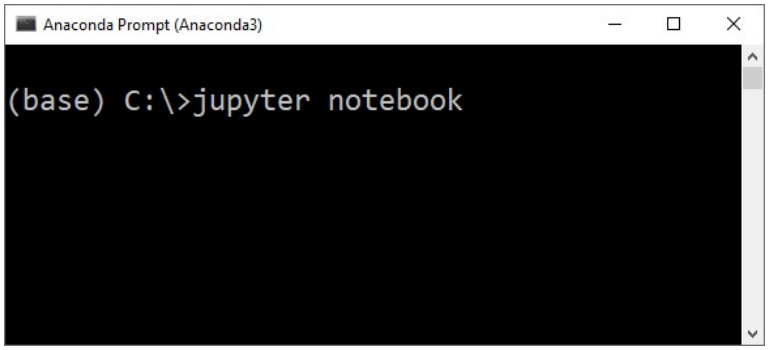
- The easiest way to start up Jupyter is by launching it through Anaconda Navigator

Click here

Starting the Jupyter Notebook Server

- Alternately, you can run the following command in a terminal window:

```
jupyter notebook
```

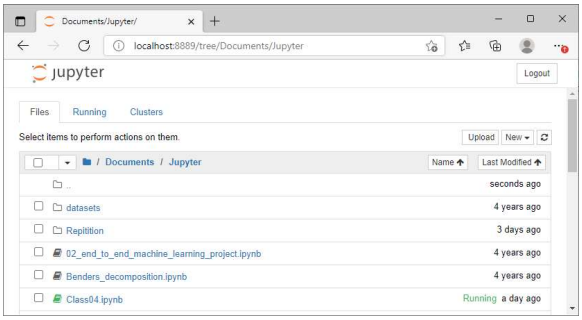


Jupyter Notebook Server

- When you start Jupyter, it will open your default browser to the following URL:

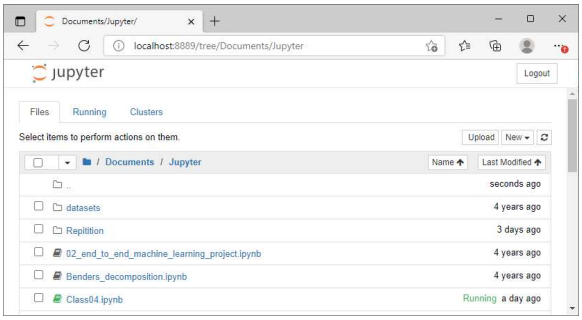
```
http://localhost:8888/tree
```

- Your browser should look something like this:



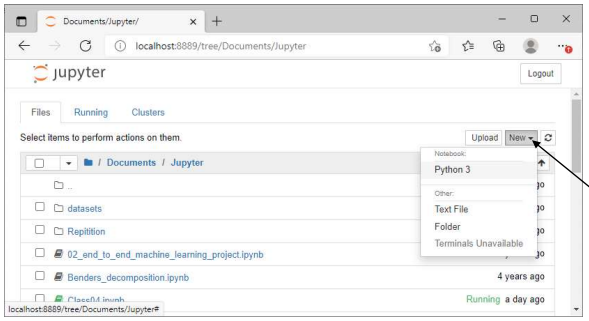
Notebook Server

- Note that right now you are not actually running a Notebook, but instead you are just running a Notebook server.



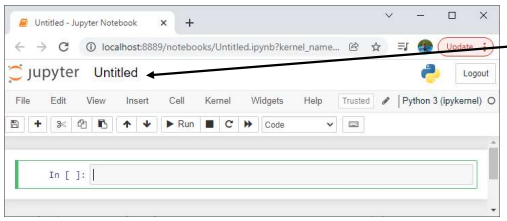
Creating a Notebook

- To create a new notebook, click the `New` button (upper right) and it will open a list of choices (depending on your installation). Let's choose `Python 3`.

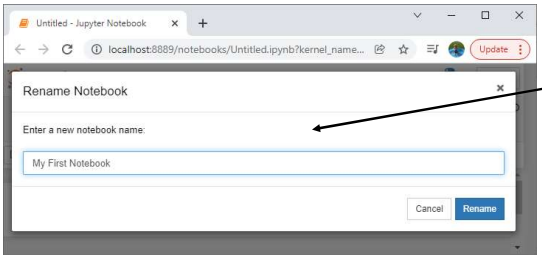


New button

Naming a Notebook



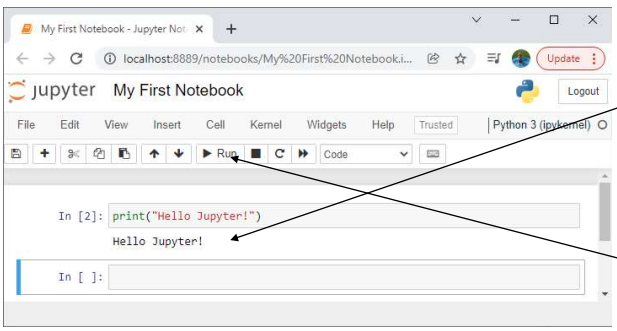
The top of the page is the word *Untitled*. Move your mouse over the word *Untitled* and click on it to rename the notebook to *My First Notebook*.



Note that this will also change the name of the file.

Running Cells

- A notebook's cell defaults to using code whenever you first create one, and that cell uses the *kernel* that you chose when you started the notebook (in our case Python 3).
- Add the following code to the cell: `print("Hello Jupyter!")`
- To execute a cell, you can select the cell and press the *Run* button.



Output produced from execution

Run button

Code in Jupyter Notebooks

- If you have multiple cells in your notebook, and you run the cells in order, you can share your variables and imports across cells.
- This makes it easy to separate your code into logical chunks without the need to reimport libraries or recreate variables or functions in every cell.
- When you run a cell, you will notice that there are some square braces next to the word *In* to the left of the cell. The square braces will auto fill with a number that indicates the order that you ran the cells.

The Menus

- The Jupyter Notebook has several menus that you can use to interact with your Notebook:
  - *File*
  - *Edit*
  - *View*
  - *Insert*
  - *Cell*
  - *Kernel*
  - *Widgets*
  - *Help*

The Menus

- The first menu is the *File* menu. Here you can create a new Notebook or open a preexisting one. This is also where you would go to rename a Notebook.
  - The *Save and Checkpoint* option allows you to create checkpoints that you can roll back to if you need to.
- The *Edit* menu can be used cut, copy, and paste cells. This is also where you would go if you wanted to delete, split, or merge a cell. You can reorder cells here too.
- The *View* menu is useful for toggling the visibility of the header and toolbar. You can also toggle Line Numbers within cells on or off.

The Menus

- The *Insert* menu is just for inserting cells above or below the currently selected cell.
- The *Cell* menu allows you to run one cell, a group of cells, or all the cells. You can also go here to change a cell's type, although the toolbar is more intuitive for that.
  - The other handy feature in this menu is the ability to clear a cell's output. If you are planning to share your Notebook with others, you will probably want to clear the output first so that the next person can run the cells themselves.

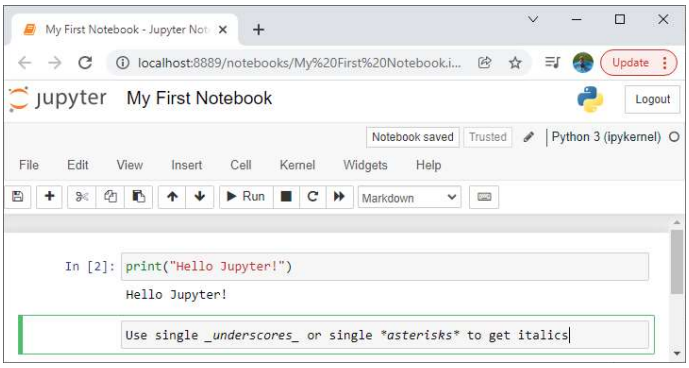
Adding Rich Content

- Jupyter Notebook supports adding rich content to its cells.
- There are technically four cell types: `Code`, `Markdown`, `Raw NBConvert`, and `Heading`.
- The `Heading` cell type is no longer supported. Instead, you are supposed to use `Markdown` for your Headings.
- The `Raw NBConvert` cell type is only intended for special use cases when using the `nbconvert` command line tool. Basically, it allows you to control the formatting in a very specific way when converting from a Notebook to another format.
- The primary cell types that you will use are the `Code` and `Markdown` cell types.

Styling Your Text

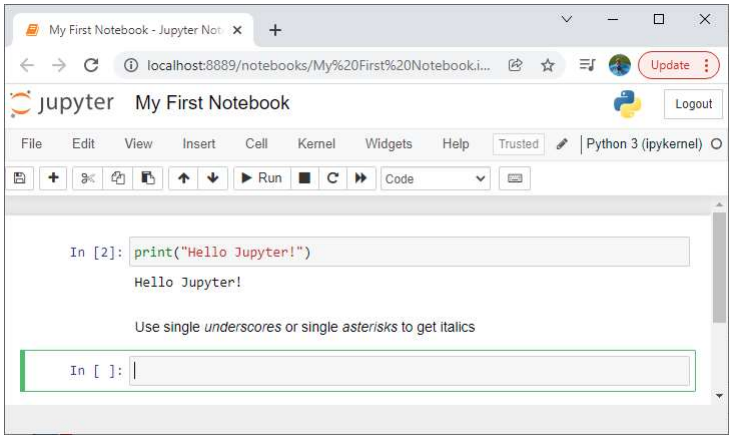
- Jupyter Notebook supports Markdown, which is a markup language that is a superset of HTML.
- Set a new cell to `Markdown` and add the following text to the cell:

```
Use single _underscores_ or single *asterisks* to get italics
```



Styling Your Text

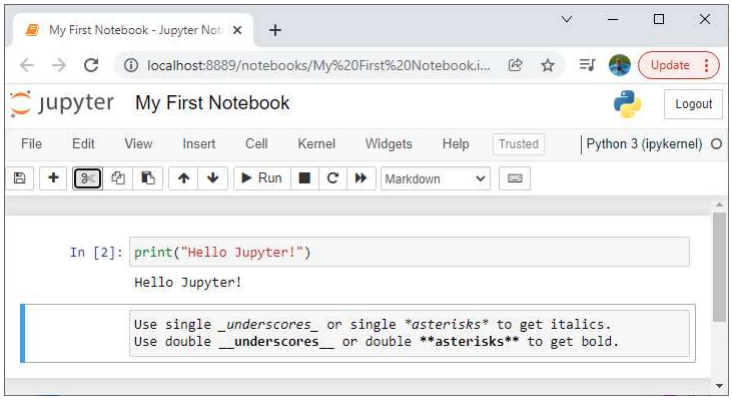
- When you run the cell, the output should look like this:



Styling Your Text

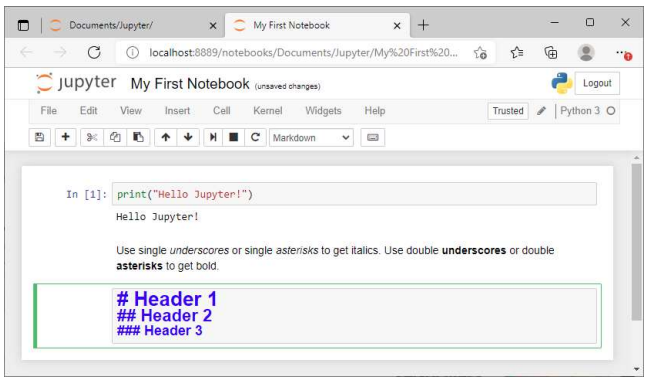
- To edit a `Markdown` cell after it has been run, just double click the cell.

- To make the text bold, use a double underscore or double asterisk.



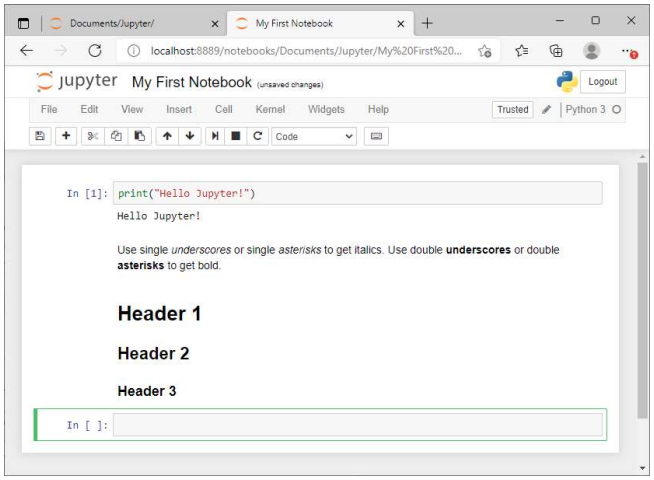
Headers

- Headers are created in `Markdown` using the pound sign.
- The more pound signs you use, the smaller the header. Jupyter Notebook preview it for you:



Headers

- When you run the cell, it will look like this:

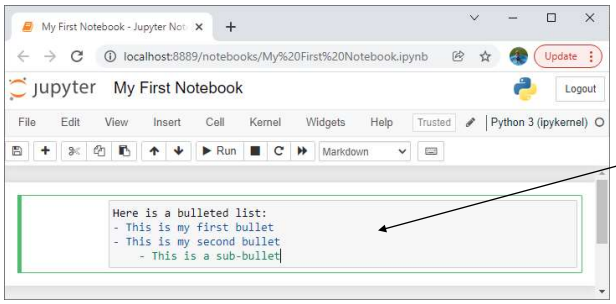


Additional Options

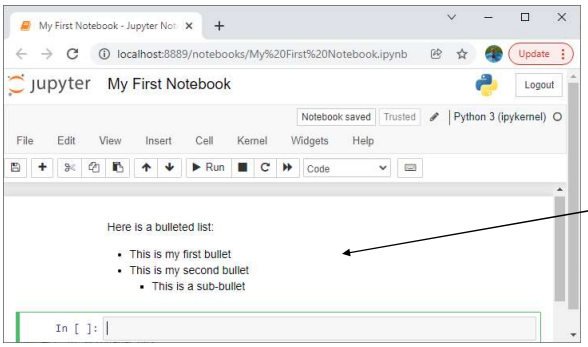
- You can switch to *monospace* font with a back single quotation mark (`): `` `string` ``
  - This is useful if you want to include commands or Python output in the Markdown cell.
- Sometimes markdown doesn't make line breaks when you want them. To force a line break, use the following: `<br>`
- Use the greater than sign (>) followed by a space to achieve an indent until the next carriage return.

Bulleted List

- To create bullet points, use one of the following methods. Note that each bullet point must be on its own line.
  - An asterisks (*) followed by one or two spaces
  - A hyphen (−) followed by one or two spaces
- To create a sub-bullet, press Tab before entering the bullet point before using one of the methods described above.



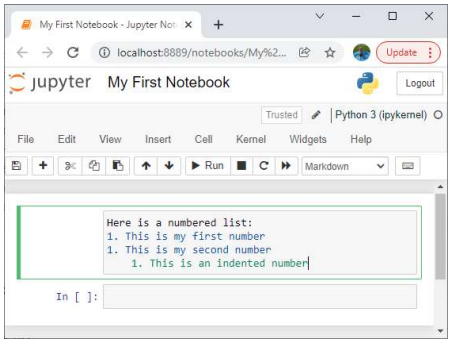
Before running the Markdown cell

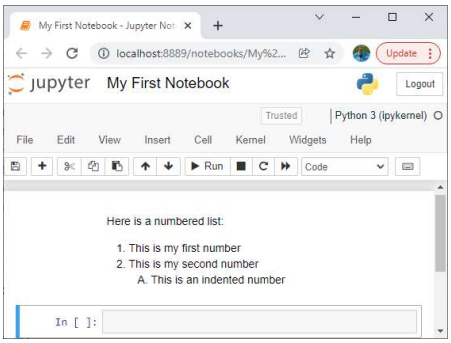


After running the Markdown cell

Numbered Lists

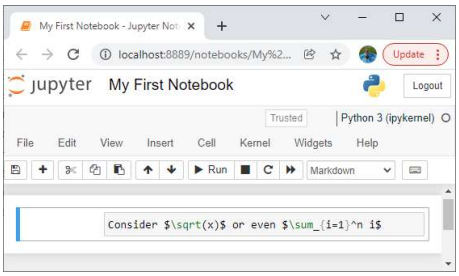
- To create a numbered list, enter `1.` followed by a space. For simplicity, you use `1.` before each entry and the list will be numbered correctly when you run the cell.

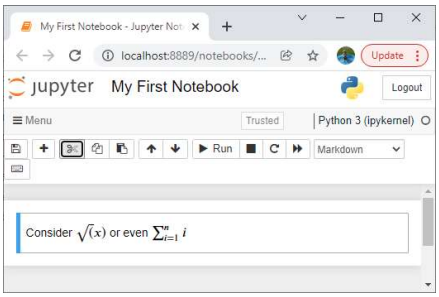




Mathematical Symbols

- You can create mathematical expressions using LaTeX syntax by using ($) symbol:





Additional Features

- Other available features include:
  - Inserting graphics
  - Using HTML code
  - Inserting web links
  - Inserting tables
  - …and more

Introduction to NumPy

- NumPy is an acronym for "Numerical Python" and is the fundamental Python package for high performance computing and data analysis.

- At the core of the NumPy package is the *ndarray* object, which stands for "*n*-dimensional array." There are several important differences between Numpy arrays and standard Python lists:

  - Numpy arrays utilize contiguous memory allocation and have a fixed size at creation. This is in contrast to Python lists, which can grow dynamically. Note that changing the size of an *ndarray* will create a new array and delete the original.

  - The elements in a Numpy array are all required to be of the same data type, and thus will be the same size in memory. This requirement forms the foundation for speedy computations.

  - Numpy arrays facilitate advanced mathematical and other type of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in lists. In particular, Numpy can perform calculations over entire arrays.

  - In this chapter we only provide an overview and suggest that you check out the Numpy documentation, which can be found at: https://numpy.org/

Importing Numpy

- The Numpy package is not part of the standard Python release and may need to be installed separately. If you use pip, you can install Numpy with:

```
pip install numpy
```

- Once Numpy is installed, you need to import the Numpy library as follows:

```
import numpy as np
```

- It is common practice to import Numpy with the alias '`np`'. This simply makes it easier to access functions as `np.function` as opposed to `numpy.function`.

1D Arrays and Vectorization

- Before we learn about how to create Numpy arrays, let's take a moment to understand a limitation of Python lists. Consider the following Python code that converts a list of weights from lbs. to kg and a list of heights from inches to meters:

```
# Create Python lists of height (inches)
height = [72, 68, 69, 68, 64, 72, 72]

# Convert height to meters
height_m = [0 for _ in range(len(height))]

for i in range(len(height)):
    height_m[i] = height[i]*0.0254

print(height_m)
```

- This produces the following output.

```
[1.8288, 1.7271, 1.7526, 1.7271, 1.6256, 1.8288, 1.8288]
```

- Numpy arrays give us a more elegant way of accomplishing this using *vectorization*, which could be defined as follows:

Def: ***Vectorization*** is the practice of replacing explicit loops with array expressions. In general, vectorized array operations will often be one to two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact seen in any kind of numerical computations.

- Before we learn about to use vectorization, we first need to learn how to create Numpy arrays.

- Numpy arrays can be created in several ways, but the easiest method is to use the `array` function to turn a Python list into an array:

```
np.array(list, type)
```

- Now consider the following Python code that creates a Numpy version of the weight list and converts the list to kg.

```
import numpy as np

# Create Python list of height (inch)
height = [72, 68, 69, 68, 64, 72, 72]

# Create Numpy version of height
np_height = np.array(height)

# Convert height to meters
np_height_m = np_height*0.0254

print(np_height_m)
```

- This produces the following output, which is identical to the previous version:

```
[1.8288, 1.7271, 1.7526, 1.7271, 1.6256, 1.8288, 1.8288]
```

- Note that we created Numpy arrays from the Python lists using the `array` command. Furthermore, we converted the units using *vectorization*, that is, Numpy performed the computations elementwise!

- Not only is the use of vectorization easier than using a for-loop, it is also much faster because all of the elements of a Numpy array are of a single type.

- It is also important to realize that Numpy array arrays behave differently than Python lists. For example, consider this:

```
import numpy as np

python_list1 = [2,4,6]
python_list2 = [1,3,5]

numpy_list1 = np.array(python_list1)
numpy_list2 = np.array(python_list2)

print("Adding the Python lists gives " + str(python_list1 + python_list2))
print("Adding the Numpy lists gives " + str(numpy_list1 + numpy_list2))
```

```
Adding the Python lists gives [2, 4, 6, 1, 3, 5]
Adding the Numpy lists gives [ 3  7 11]
```

- You can see that when adding the Python lists the result was to concatenate the lists, while the result when adding the Numpy arrays is to perform the additions elementwise.

- It is very important that you pay attention to the Python type you are dealing with because the behavior and outcome can be different.

Changing the Datatype

- `np.array()` has an additional parameter of `dtype` that can be used to define whether the elements are integers, floating point numbers, or complex numbers.

- Consider the following example that makes use of the `dtype` attribute of the Numpy array.

```
numbers = [10, 15, 20, 35]
numbers2 = [10.0, 15.1, 19.5,]

x = np.array(numbers)
y = np.array(numbers, dtype = "float")
z = np.array(numbers2)

print("The data type of x is " + str(x.dtype))
print("The data type of y is " + str(y.dtype))
print("The data type of z is " + str(z.dtype))
```

```
The data type of x is int32
The data type of y is float64
The data type of z is float64
```

- This implies that `x` is a 32-bit integer, while `y` is a 64-bit floating-point number (because we included the additional parameter `dtype = "float"`). Note that `z` is of type `float64` without having to include the parameter `dtype = "float"` because Python automatically recognized that these should be floats because of the included decimal point.

Indexing and Slicing in 1D Arrays

- It is important to remember that Python indexing starts from 0.

- Consider the indexing syntax below:

Syntax	Result
`x[start:end]`	Elements in `x` indexed from `start` to `end` (but `end` is not included).
`x[start:end:step]`	Elements in `x` indexed from `start` to `end` using the step size `step`. Note that `end` is not included, and the default step value is 1.
`x[start:]`	Elements in x indexed from `start` through the end of the array.
`x[:end]`	Elements in x indexed from the beginning through `end` (but `end` is not included).

- We illustrate these below.

```
age = np.array([18,23,89,11,35,37,26,52,76])
print("The array age is " + str(age))
print("The result of age[2] is " + str(age[2]))
print("The result of age[2:5] is " + str(age[2:5]))
print("The result of age[:3] is " + str(age[:3]))
print("The result of age[4:] is " + str(age[4:]))
print("The result of age[2:7:2] is " + str(age[2:5]))
print("The result of age[::-1] is " + str(age[::-1]))
```

```
The array age is [18 23 89 11 35 37 26 52 76]
The result of age[2] is 89
The result of age[2:5] is [89 11 35]
The result of age[:3] is [18 23 89]
The result of age[4:] is [35 37 26 52 76]
The result of age[2:7:2] is [89 11 35]
The result of age[::-1] is [76 52 26 37 35 11 89 23 18]
```

- Note that the last command, `age[::-1]`, reversed the array. However, it is important to note that this command does not actually modify the array `age`.

Creating Sequences of Numbers

- You can create sequences of numbers using the `arange` command, which creates an instance of `ndarray` with evenly spaced values and returns the *reference* to it:

```
numpy.arange(start, stop, step)
```

- Below we illustrate the `arange` command.

```
print("The result of np.arange(1,8) is " + str(np.arange(1,8)))
print("The result of np.arange(2,10,2) is " + str(np.arange(2,10,2)))
print("The result of np.arange(2,10.1,2) is " + str(np.arange(2,10.1,2)))
print("The result of np.arange(5) is " + str(np.arange(5)))
print("The result of np.arange(-6,3) is " + str(np.arange(-6,3)))
print("The result of np.arange(5,1,-1) is " + str(np.arange(5,1,-1)))
```

```
The result of np.arange(1,8) is [1 2 3 4 5 6 7]
The result of np.arange(2,10,2) is [2 4 6 8]
The result of np.arange(2,10.1,2) is [  2.   4.   6.   8.  10.]
The result of np.arange(5) is [0 1 2 3 4]
The result of np.arange(-6,3) is [-6 -5 -4 -3 -2 -1  0  1  2]
The result of np.arange(5,1,-1) is [5 4 3 2]
```

- Note that the default step size is 1.

- Can you explain why `np.arange(2,10.1,2)` includes 10, while `np.arange(2,10,2)` does not?

- Note that the single argument command `np.arange(5)` creates a sequence that starts at zero and stops at 4 with an increment of 1, while `np.arange(5,1,-1)` has a step size of –1, and thus counts backwards from 5 to 1.

Indexing with Boolean Arrays

- In Numpy, we also have the ability to do list *subsetting* using Boolean arrays.

- Suppose we wanted to get all the ages above 35. The following syntax will generate a Numpy array containing Booleans, where it is True if the corresponding value is above 35, and False otherwise.

```
age = np.array([18,23,89,11,35,37,26,52,76])
print(age > 35)
```

```
[False False  True False False  True False  True  True]
```

- If we wanted to generate a new Numpy array only containing those values in age that are greater than 35, we can use the Boolean array inside square brackets to do *subsetting*:

```
old = age[age > 35]
print(old)
```

```
[89 37 52 76]
```

- Recall that Python doesn't allow "ternary" comparisons like `35 < age < 55`. Instead, we need to combine comparisons with logical operators **and**, **not**, and **or**, as shown in the table below.

Logical Operator	Interpretation	Interpretation Elaborated
&	"bitwise" and	returns true (1) if both elements are true, and false (0) otherwise
~	"bitwise" not	returns false (0) if the element is true and true (1) if the element is false
|	"bitwise" or	returns true (1) if at least one of the elements are true, and false (0) otherwise

- We illustrate this below.

```
age = np.array([18,23,89,11,35,37,26,52,76])
middle_age=age[(age>35) & (age<55)]
print(middle_age)
```

```
[37 52]
```

2D Arrays

- We can create 2-dimensional Numpy arrays by passing in a list of lists into the Numpy `array` function:

```
height = [72, 68, 69, 68, 64, 72, 72]
weight = [200, 165, 160, 135, 120, 162, 190]
metrics = np.array([height,weight])
print(metrics)
```

```
[[ 72  68  69  68  64  72  72]
 [200 165 160 135 120 162 190]]
```

- We can access the dimension, the size (total number of elements), and the shape using the *attributes* `ndim`, `size`, and `shape`, respectively:

```
print("metrics.ndim = " + str(metrics.ndim))
print("metrics.size = " + str(metrics.size))
print("metrics.shape = " + str(metrics.shape))
```

```
metrics.ndim = 2
metrics.size = 14
metrics.shape = (2, 7)
```

Indexing in 2D Arrays

- You can access the element in the *i*th row and *j*th column in two different ways:

```
print(metrics[0][2])
print(metrics[0,2])
```

```
69
69
```

- Both give the same result, but the second method, `metrics[0,2]`, is more compact.

- We can take slices of two-dimensional arrays as in the following example.

```
A = np.array([[0,1,2],[3,4,5],[6,7,8]])
print(A)
print(A[:2,:])
print(A[:,0:2])
print(A[:,0])
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

The two-dimensional array `A`

```
[[0 1 2]
 [3 4 5]]
```

`A[:2,:]` gives rows 0 up to 2 and all columns

```
[[0 1]
 [3 4]
 [6 7]]
```

`A[:,0:2]` gives all rows and columns 0 up to 2

```
[0 3 6]
```

`A[:,0]` gives all rows and the first column, but the dimension has dropped by one!

- Essentially, you simply specify a one-dimensional slice for each *axis*. One can also supply a number for an axis rather than a slice but doing so will drop the dimension by one.

Reshaping Arrays

- We can reshape a Numpy array without changing its data using the `reshape(.)` method of the `ndarray`.

```
x = np.arange(1,10)
A = x.reshape(3,3)
print(x)
print(A)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]

[[1 2 3]
 [4 5 6]
```

```
 [7 8 9]]

[1 2 3 4 5 6 7 8 9]
```

- Note that the single-dimensional array `x` is reshaped into a 3×3 matrix `A`. Interestingly though, the `reshape(.)` method does not alter the shape of the original array `x`.

- However, the `resize(.)` method of the `ndarray` does change the shape and size of the array. For example:

```
x = np.arange(1,10)
A = x.reshape(3,3)
print(x)
x.resize(3,3)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- If the dimension is given as −1 in a reshaping, the other dimensions are automatically calculated provided that the given dimension is a multiple of the total number of elements in the array.

```
x = np.arange(1,16)
print(x)
B = x.reshape(5,-1)
print(B)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]

[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
```

- In the above code we only indicated that we wanted 5 rows and the −1 instructed Python to automatically calculate the number of elements in the other dimension (so that each row has the same number of elements).

- As we will see later in the course, reshaping data is quite common as many of the Python algorithms for performing data analytics require the data to be in a particular shape.

Vectorization in Two-Dimensions

- We can also do vectorization (elementwise calculations) with two-dimensional Numpy arrays. Consider the following dataset, where each row provides the height and weight for a different individual.

Height (in)	Weight (lbs)
72	200
68	165
69	160
68	135
64	120
72	162

```
data = np.array([[72,200], [68,165], [69,160], [68,135], [64,120], [72,162]])
print(data)
```

```
[[ 72 200]
 [ 68 165]
 [ 69 160]
 [ 68 135]
 [ 64 120]
 [ 72 162]]
```

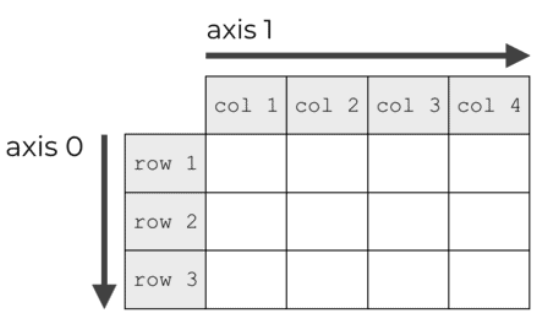
- Suppose we wanted to convert height from inches to meters and weight from lbs. to kg. This can be accomplished by multiplying each height by 0.0254 and each weight by 0.453592.

```
conversion = np.array([0.0254, 0.453592])
newData = data*conversion
print(newData)
```

```
[[ 1.8288   90.7184  ]
 [ 1.7272   74.84268 ]
 [ 1.7526   72.57472 ]
 [ 1.7272   61.23492 ]
 [ 1.6256   54.43104 ]
 [ 1.8288   73.481904]]
```

Understanding Axes Notation

- In Numpy, an *axis* refers to a single dimension of a multidimensional array. In a two-dimensional Numpy array, the axes are the directions along the rows and columns.



- Axis 0 is the axis that runs downward along the rows, while axis 1 is the axis that runs horizontally across the columns.

- The following array has 2 axes. The first axis has length 2 and the second axis has a length of 3.

```
[[0., 0., 0.],
 [1., 1., 1.]]
```

- As an example for how Numpy uses axes, consider the Numpy `sum` function, which returns the sum of array elements over the specified axis:

```
numpy.sum(arr, axis,…)
```

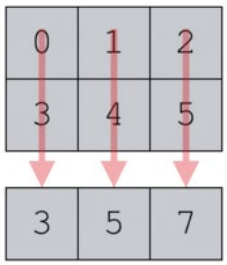
- Here, the first two parameters are `arr` (the input array) and `axis` (the axis along which we want to calculate the sum).

- Consider the following example:

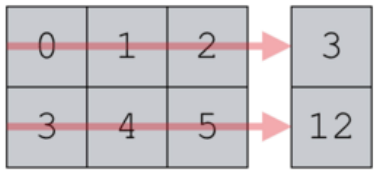
```
arr = [[14, 17, 12, 33, 44],
       [15, 6, 27, 8, 19],
       [23, 2, 54, 1, 4,]]

print('The sum of all values is ' + str(np.sum(arr)))
print()
print('The column sums are ' + str(np.sum(arr, axis=0)))
print()
print('The row sums are ' + str(np.sum(arr, axis=1)))
print()
```

```
The sum of all values is 279

The column sums are [52 25 93 42 67]

The row sums are [120  75  84]
```

- Note that when we set `axis=0`, the function sums down the columns. The axis parameter controls which axis will be *aggregated* or *collapsed*. So, when we set `axis=0` we are collapsing the rows:



- When we set `axis=1`, we collapse the columns:



Basic Statistics in Numpy

- Numpy is equipped with a number of statistical functions, including the following (see the Numpy documentation for the full suite of statistical functions).

Function	Result
`numpy.amin(.)`	Determines the minimum value of the element along a specified axis.
`numpy.amax(.)`	Determines the maximum value of the element along a specified axis.
`numpy.mean(.)`	Determines the mean value
`numpy.median(.)`	Determines the median value
`numpy.std(.)`	Determines the standard deviation
`numpy.var(.)`	Determines the variance
`numpy.average(.)`	Determines a weighted average
`numpy.percentile(.)`	Determines the *n*th percentile along an axis

- Consider the following example.

```
A = np.array([[31,64,70],[79,94,10],[49,91,60]])

print('Our array is:')
print(A)
print()
print('The median of all values is: ' + str(np.median(A)))
print()
print('The median of the first column is: ' + str(np.median(A[:,0])))
print()
print('The median of the second row is: ' + str(np.median(A[1,:])))
print()
print('The median along the first axis is: ' + str(np.median(A,axis=0)))
print()
print('The median along the second axis is: ' + str(np.median(A,axis=1)))
```

```
Our array is:
[[31 64 70]
 [79 94 10]
 [49 91 60]]

The median of all values is: 64.0

The median of the first column is: 49.0

The median of the second row is: 79.0

The median along the first axis is: [ 49.  91.  60.]

The median along the second axis is: [ 64.  79.  60.]
```

- Interestingly, there are `ndarray` *methods* that yield the same functionality as the Numpy functions. We demonstrate below:

```
list = np.array([10,-3,5,6,11,52,9,23,41,22])

print(list)
print('list.mean() is ', list.mean())
print('list.std()', list.std())
print('list.max()', list.max())
print('list.sum()', list.sum())

print()
array = np.array([[14, 17, 12, 33, 44],
       [15, 6, 27, 8, 19],
       [23, 2, 54, 1, 4,]])

Print(array)
print('The column means are', array.mean(axis=0))
print('The row means are', array.mean(axis=1))
```

```
[10 -3  5  6 11 52  9 23 41 22]
list.mean() is  17.6
list.std() 16.3474768695
list.max() 52
list.sum() 176

[[14 17 12 33 44]
 [15  6 27  8 19]
 [23  2 54  1  4]]
The column means are [ 17.33333333   8.33333333  31.          14.
22.33333333]
The row means are [ 24.   15.   16.8]
```

- You may be wondering which technique should you use, the `ndarray` methods or the Numpy functions. Unfortunately, there are no firm rules as to which method to employ. This is ultimately up to you, but you should use the method that is more readable in your code.

- We encourage you to explore the methods available to Numpy `ndarray`.

Missing Data

- In the real world, data is rarely clean and typically has some amount of missing data. To make matters even more complicated, different data sources may indicate missing data in different ways. Here we will discuss how Numpy chooses to represent missing numerical data.

- Numpy uses the data representation `NaN`, which is an acronym for Not a Number). This is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation. Consider the following:

```
data = np.array([2, np.nan, -1, 3])
print(data.dtype)
```

```
float64
```

- Note that Numpy chose a floating-point type for this array. You should be aware that `NaN` is a bit like a data-virus that infects any other object it touches. That is, regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`.

```
print('2 + np.nan yields ' + str(2 + np.nan))
print('4*np.nan yields ' + str(4*np.nan))
print('The sum of the data array is ' + str(np.sum(data)))
```

```
2 + np.nan yields nan
4*np.nan yields nan
The sum of the data array is nan
```

- Note that we can use Numpy's `nansum(.)` function to return the sum of array elements over a given axis treating `NaN` as zero:

```
print('The sum of data array, ignoring NaN is ' + str(np.nansum(data)))
```

```
The sum of data array, ignoring NaN is 4.0
```

- To check whether an array contains missing values, we can use the `isnan(.)` function, which returns a Boolean array.

```
print('The result of np.isnan(data) is ' + str(np.isnan(data)))
```

```
The result of np.isnan(data) is [False  True False False]
```

Higher-Dimensional Arrays?

- There are a number of situations where you might see data with greater than two dimensions.

- *Panel data* is multi-dimensional data involving measurements over time. For example, data that tracks a cohort (group) of individuals over time could be structured as (`subjects, dates, attributes`).

- Another example is color-image data for multiple images, which is typically stored in four dimensions. Each image is a three-dimensional array of (height, width, channels), where channels are usually red, green, and blue (RGB) values. A collection of images is then just (`image_number, height, width, channels`).

- We can manually create a 3d array by passing in a list of lists of lists as follows:

```
threeD = np.array([[[10,20,30], [40,50,60]], [[15,25,35], [45,55,65]],
[[70,80,90],[75,85,95]]])
print(threeD)
```

```
[[[10 20 30]
  [40 50 60]]

 [[15 25 35]
  [45 55 65]]

 [[70 80 90]
  [75 85 95]]]
```

- We can obtain the dimension, total number of elements, and shape using the `ndim`, `size`, and `shape` attributes:

```
print("The dimension is " + str(threeD.ndim))
print("The total number of elements is " + str(threeD.size))
print("The shape is " + str(threeD.shape))
```

```
The dimension is 3
The total number of elements is 18
The shape is (3, 2, 3)
```

- We further illustrate below—can you follow along?

```
matrix3D = np.arange(36).reshape(3, 4, 3)
print(matrix3D)
print("matrix3D[1,3,2] = " + str(matrix3D[1,3,2]))
print("matrix3D[0,2] = " + str(matrix3D[0,2]))
print("matrix3D[0] = " + str(matrix3D[0]))
print("matrix3D[0,:,1] = " + str(matrix3D[0,:,1]))
print("matrix3D[0,:,1].reshape(4,1) = " + str(matrix3D[0,:,1].reshape(4,1)))
```

A single command created the 3d array of size 3×4×3 containing the numbers 0 through 35

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]]

 [[12 13 14]
  [15 16 17]
  [18 19 20]
  [21 22 23]]

 [[24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
```

This matrix is `A[0,.,.]`

This matrix is `A[1,.,.]`

This matrix is `A[2,.,.]`

This accesses the single element at the location indexed by [1,3,2]

```
matrix3D[1,3,2] = 23
```

This extracts the row indexed by 2 in $A_{0\cdot\cdot}$

```
matrix3D[0,2] = [6 7 8]
```

This extracts the matrix $A_{0\cdot\cdot}$

```
matrix3D[0] = [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

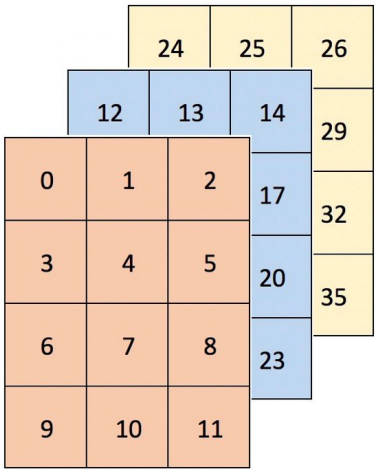
This extracts the column indexed by 1 in matrix $A_{0\cdot\cdot}$ Note that Python returns it as a "row"

```
matrix3D[0,:,1] = [ 1  4  7 10]
```

This reshapes the previous answer as a column

```
matrix3D[0,:,1].reshape(4,1) = [[ 1]
 [ 4]
 [ 7]
 [10]]
```

- Visual, the array matrix3D can be thought of as a container of three 4×3 grids, or a rectangular prism, and would look like this:



- Higher dimensional arrays can be tougher to picture, but they will still follow this "arrays within an array" pattern.

Broadcasting

- The following discussion is adapted from a blog post of Ayoosh Kathuria and the Python Data Science Handbook by Jake VanderPlas.

- Broadcasting is one of the best features of `ndarrays` as it allows you to perform arithmetic between `ndarrays` of different sizes.

- Numpy's operations are usually done element-by-element, which requires two arrays to have the exact same shape. For example, suppose we wanted to multiple every element of an array by 2. It appears that we would need to do something like what is illustrated in the example below.

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])

print('a = ', a)
print('b = ', b)
print('The result of a*b is', a*b)
```

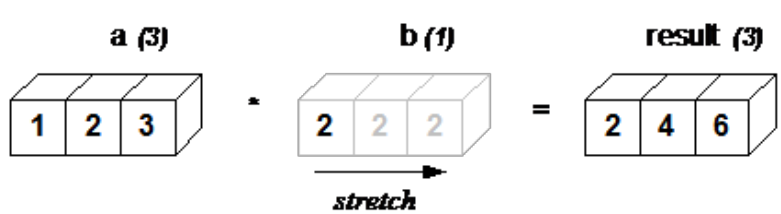
```
a =  [ 1.  2.  3.]
b =  [ 2.  2.  2.]
The result of a*b is [ 2.  4.  6.]
```

- Numpy's broadcasting rule relaxes the constraint of requiring the same shape when the array shapes meet certain criteria. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation. For example, consider the following code that accomplishes what we did in the last example.

```
a = np.array([1.0, 2.0, 3.0])
b = 2

print('a = ', a)
print('b = ', b)
print('The result of a*b is', a*b)
```

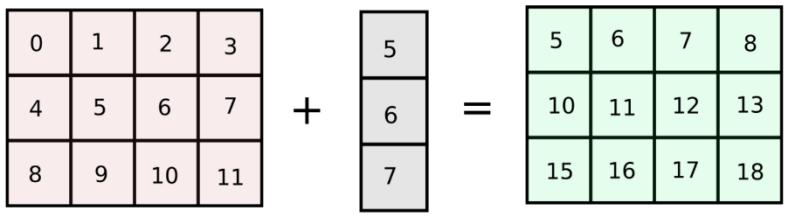
```
a =  [ 1.  2.  3.]
b =  2
The result of a*b is [ 2.  4.  6.]
```

- Note that the result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being stretched or duplicated during the arithmetic operation into an array with the same shape as `a`, as illustrated in the figure below.



- The stretching analogy is only conceptual as Numpy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

- As another example, suppose we have a two-dimensional array of shape `(3, 4)` containing 3 rows and 4 columns. Further suppose that we want to add a column to each of the columns in our array. We graphically illustrate what we are trying to achieve below:



- As with most tasks in Python, there are a number of different ways we could accomplish this. For example, we could loop over the columns of the matrix (two-dimensional array) and add the column.

```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7])

print(matrix)
print(vector)

num_cols = matrix.shape[1]

for col in range(num_cols):
      matrix[:, col] += vector

print()
print('The updated matrix is')
print(matrix)
```

The number of columns

Note that `matrix[:, col]` returns the column as a row!

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[5 6 7]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- While this accomplishes our goal, if the number of columns in our original array matrix are increased to a very large number, the code described will run slow as we are looping over the number of columns.

- Another way to accomplishing this would be to make a matrix of equal size as the original matrix with identical columns (which we will refer to as the column-stacking approach). This is illustrated and coded below.

```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7])

add_matrix = np.tile(vector,(4,1)).T

print(matrix)
print(add_matrix)

matrix += add_matrix

print()
print('The updated matrix is')
print(matrix)
```

Here we are using Numpy's `tile(.)` function and the transpose operator—take a moment to learn about this on the web.

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- This approach is much faster. However, while it worked well in the two-dimensional case, applying the same approach with higher dimensional arrays can be tricky.

- The good news is that we can use broadcasting to perform arithmetic operations on arrays of unequal sizes. According to the Numpy documentation

  Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python.

- Under the hood, Numpy does something similar to our column-stacking approach. However, we do not have to worry about stacking arrays in multiple directions explicitly.

- According to the Numpy documentation, the rule governing whether two arrays have compatible shapes for broadcasting can be expressed in a single sentence:

The Broadcasting Rule:

In order to broadcast, the size of the *trailing axes* for both arrays in an operation must either be the same size of one of them must be one.

- However, the Python Data Science Handbook offers a more comprehensive set of rules:

Rules of Broadcasting:

Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

- The size of the result array created by broadcast operations is the maximum size along each dimension from the input arrays. In the case the arrays are not compatible, you will get a `ValueError`.

- Let's revisit our previous example and see broadcasting in action.

```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7]).reshape(3,1)


print(matrix)
print(vector)

matrix += vector

print()
print('The updated matrix is')
print(matrix)
```

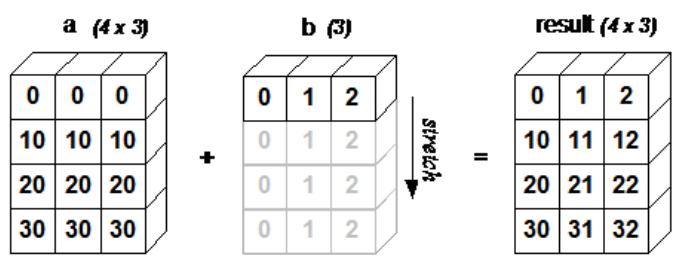
```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[5]
 [6]
 [7]]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- Consider the following example adapted from the Numpy documentation.

```
a = np.array([[ 0.0,  0.0,  0.0],
             [10.0, 10.0, 10.0],
             [20.0, 20.0, 20.0],
             [30.0, 30.0, 30.0]])

b = np.array([0, 1, 2])

print('a =')
print(a)
print(‘b = ‘, b)
print('a+b =')
print(a+b)
```

```
a =
[[  0.   0.   0.]
 [ 10.  10.  10.]
 [ 20.  20.  20.]
 [ 30.  30.  30.]]

b =  [0 1 2]

a+b =
[[  0.   1.   2.]
 [ 10.  11.  12.]
 [ 20.  21.  22.]
 [ 30.  31.  32.]]
```

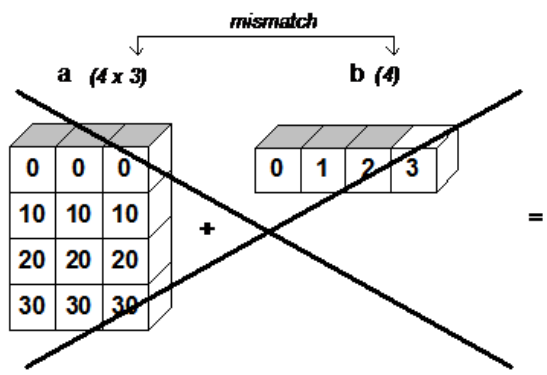
- You can see that `b` is added to each row of `a`, as illustrated below.



- If `b` is longer than the rows of `a`, as illustrated in the code and figure below, an exception is raised because of the incompatible shapes.

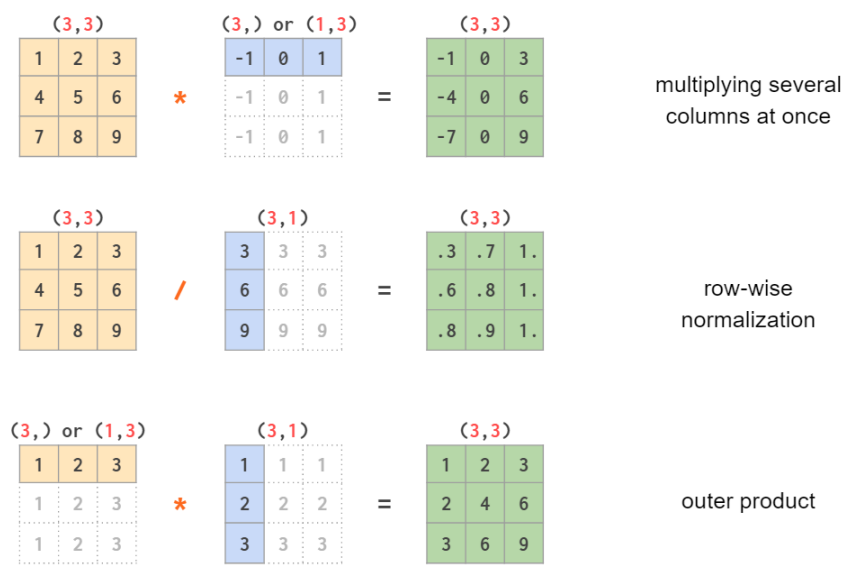
```
a = np.array([[ 0.0,  0.0,  0.0],
             [10.0, 10.0, 10.0],
             [20.0, 20.0, 20.0],
             [30.0, 30.0, 30.0]])

b = np.array([0, 1, 2,3])

print('a =')
print(a)
print('b = ', b)
print('a+b =')
print(a+b)
```

```
a =
[[  0.    0.    0.]
 [ 10.   10.   10.]
 [ 20.   20.   20.]
 [ 30.   30.   30.]]

b =  [0 1 2 3]

a+b =
Traceback (most recent call last):
  File "OneDExamples.py", line 279, in <module>
    print(a+b)
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```



- When the trailing dimensions of the arrays are unequal, broadcasting fails because it is impossible to align the values in the rows of the 1st array with the elements of the 2nd arrays for element-by-element addition.

- The website TowardDataScience has a nice graphical illustration of some different arithmetic operations in 2d:



- In 3d and above the broadcasting is even less intuitive and using it requires knowledge of the broadcasting rules in their generic forms.

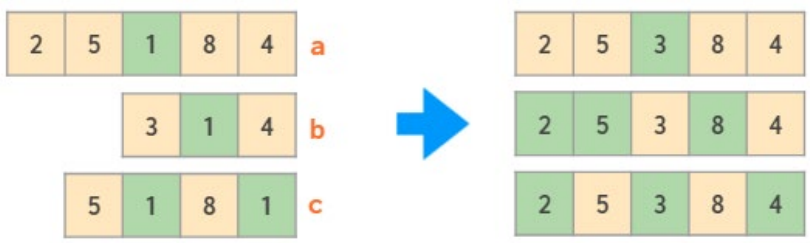
- As a final example, consider the following Python code.

```
a = np.ones((2,5,1,8,4))
b = np.ones((3,1,4))
c = np.ones((5,1,8,1))

d = a+b+c
print('The shape of d is ', d.shape)
```

Note that the `np.ones(.)` function returns an `ndarray` of given shape where all the elements are equal to 1.

- Can you guess the shape of `d`?

- Applying the broadcasting rules, the arrays `b` and `c` are stretched as illustrated below.



- Thus, the final size/shape is `(2, 5, 3, 8, 4)` as indicated by the Python output:

```
The shape of d is  (2, 5, 3, 8, 4)
```

- As we will see throughout this course, broadcasting is going to be a core technique needed to efficiently use Python for data science.

Mathematical Functions in Numpy

Matlab has a number of built-in math functions:

Function	**Result**
`abs(x)`	Absolute value of `x`
`sqrt(x)`	Square root of `x`
`log(x)`	Natural log of `x`
`log10(x)`	Computes $\log_{10}(x)$
`log2(x)`	Computes $\log_2(x)$
`exp(x)`	Exponential: e^x
`round(x)`	Rounds `x` to the nearest integer
`floor(x)`	Rounds `x` to the nearest integer toward $-\infty$
`ceil(x)`	Rounds `x` to the nearest integer toward ∞
`sign(x)`	Returns a value of -1 if $x < 0$, returns a value of 0 if $x = 0$, and returns a value of $+1$ if $x > 0$.
`mod(a,m)`	Modulo operator: remainder after division of `a` by `m`.

Below is a Python session illustrating some of these functions in action.

```
>>> np.sqrt(85)
9.2195444572928871

>>> x = np.array([4,9,25])
>>> np.sqrt(x)
array([ 2.,  3.,  5.])

>>> np.round(8.4)
8.0

>>> np.floor(-8.6)
-9.0

>>> np.floor(8.6)
8.0

>>> np.sign(-23)
-1

>>> np.mod(25,4)
1
```

Notice that Numpy used vectorization here to compute the square root of each entry of `x`.

- The trigonometric functions `sin`, `cos`, and `tan` all assume that angles are represented in radians.

- The trigonometric functions `sin`, `cos`, and `tan` all assume that angles are represented in radians. The value of π is a built-in constant, `np.pi`, in Python. Keep in mind that π is not a machine number (it cannot be represented exactly as a floating-point number) and therefore the constant `pi` in Python is only an approximation. Usually, this is not important; however, you may notice some surprising results—for example:

```
>>> np.sin(np.pi)

1.2246467991473532e-16
```

Trigonometric Functions (radians)

Function	Result
`sin(x)`	Sine of `x` in radians
`sinh(x)`	Hyperbolic sine of `x` in radians
`arcsin(x)`	Inverse sine of `x` in radians
`cos(x)`	Cosine of `x` in radians
`cosh(x)`	Hyperbolic cosine of `x` in radians
`arccos(x)`	Inverse cosine of `x` in radians
`tan(x)`	Tangent of `x` in radians
`tanh(x)`	Hyperbolic tangent of `x` in radians
`arctan(x)`	Inverse tangent of `x` in radians

- Numpy has built-in functions, `rad2deg` and `deg2rad`, to convert an angle from radians to degrees and degrees to radians, respectively.

- In the Python session below, we illustrate some of these functions in action.

```
>>> np.rad2deg(np.pi)
180.0

>>> np.deg2rad(90)
1.5707963267948966

>>> x = np.array([90, 180, 360])
>>> np.deg2rad(x)
array([ 1.57079633,  3.14159265,  6.28318531])
```

Wrapping Up

- There is a lot more to learn about Numpy, but this is a good solid start.

- We will learn more about Numpy as we need it. However, we do suggest that you check out the official Numpy documentation and this great guide available from BetterProgramming.

File Systems

- A ***file system*** defines how *persistent* data is stored and retrieved from a storage device such as a hard drive.
- We call storage ***persistent*** if the data is maintained when the computer is turned-off, rebooted, etc. Otherwise, the data is called ***transient***.
- Without a file system, the storage device would contain a big chunk of data stored back-to-back, and the *operating system* wouldn't be able to tell them apart.
- The ***operating system*** is responsible for organizing persistent data, and for allowing sharing and mediating access to the storage as required by *users*, the owners of the data.

Operating Systems

- Specific details of organization differ somewhat between operating systems, such as Windows, MacOS, or Linux.
- Logically coherent sets of data are organized into *files* comprised of sequences of bits (0's and 1's) forming bytes (8-bits).
- Files whose bytes represent sequences of characters are called ***text files*** (typically have a `.txt` extension).
- Files whose bytes represent underlying data types, like integers and floats, are called ***binary files***.

Files

- Files can store source programs, executable problems, libraries, media, data, etc.
- One program can create a file and another program can read and process the file.
- In this course, files on the *local* file system will be our first and simplest in our spectrum of data providers.

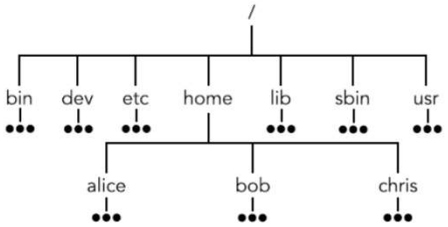
Three Main Parts of Most Files

1. **Header:** metadata about the contents of the file (file name, size, type, and so on).
2. **Data:** contents of the file as written by the creator or editor.
3. **End of file (EOF):** special character that indicates the end of the file.

Hierarchical Organization

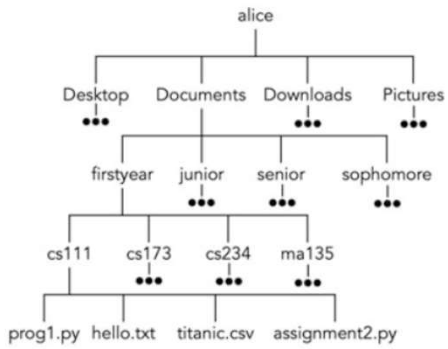
- Operating systems utilize a *hierarchical organization* to organize files.
- This hierarchy defines a *tree*, where the root and interior nodes of the tree provide the abstraction of a *directory* or *folder*.
- Each folder can have contents, which are the *children* of the folder node in the tree.
- The child contents of a folder can consist of zero or more files as well as zero or more (sub)folders.
- A *leaf* is a node with no children. As *internal node* is a node that is not a leaf. A file must be a leaf of the tree, as it cannot have any subordinate folders or files.

Fig. 2.1 Top levels of a file system hierarchy



- This is abstracted from a Linux system.
- "…" represents a subtree not pictured.
- The '/' denotes the root of the file system
- The *top-level directories*, direct children of the root, like `bin` and `dev`, are used for operating system resources.

Fig. 2.2 Alice hierarchy



- Here we focus on the user `alice` and the subtree in the file system that represents her *home directory*, which is the portion of the file system she sees when see logs in, and when she starts a *File Explorer*.

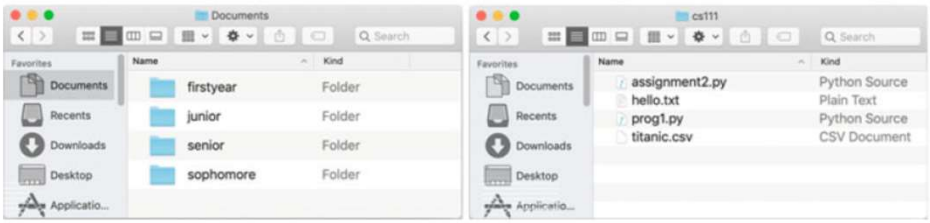


Fig. 2.3 File explorer view of alice hierarchy

- Above is what we might see using a File Explorer on a Mac after navigating to Alice's the `Documents` folder.
- It's important to realize that the File Explorer is not the only interface to the underlying file system structure.
- For example, files and folders can also be accessed directly using applications such as Terminal on a MacOS or Command Prompt on Windows.

Paths

- A ***path*** is a traversal of a resource tree starting at one node, called the ***source node*** and ending at another node, called the ***destination node***.
- The destination node could be a file or a directory node.
- A path whose source node is the root of the tree is called ***absolute***.

File Paths

- A ***file path*** is a string that represents the location of a file. It's broken up into three major parts:

1. **Folder Path:** the file folder location on the file system where subsequent folders are separated by a forward slash / (Unix or Mac) or backslash \ (Windows).
2. **File Name:** the actual name of the file.
3. **Extension:** the end of the file path pre-pended with a period (.) used to indicate the file type.

Table 2.1 Example absolute and relative paths

Path	Description
`/→home→alice`	An absolute path specifying the home directory of the user `alice`
`/→home→alice→Documents→firstyear→cs111→hello.txt`	An absolute path specifying the specific `hello.txt` file of `alice` in the `cs111` class[4]
`Documents → firstyear → cs111 → hello.txt`	A relative path that specifies the location of the specific `hello.txt` file of `alice` in the `cs111` class
`hello.txt`	A relative path that, relative to the `cs111` directory of `alice`, specifies the `hello.txt` file

In the following list, we give the string translation for each of the example paths in Table 2.1.

- `/→home→alice` is string `"/home/alice"`.
- `/→home→alice→Documents→firstyear→cs111→hello.txt` is string `"/home/alice/Documents/firstyear/cs111/hello.txt"`
- `Documents → firstyear → cs111 → hello.txt` is string `"Documents/firstyear/cs111/hello.txt"`
- `hello.txt` is string `"hello.txt"`

Path Separator Convention

- On the previous slide, we used the MacOS and Linux convention of forward slash (/) as the path separator and / for the root.
- Windows uses `<disk letter>:\` for specifying a file system tree root, and backslash (\) as the separator in paths.
- If the string representation of a path begins with the character(s) for the root, we know the path is absolute.

Working Directory

- Whenever we execute our programs, the operating system maintains a concept of a ***working directory***.
- Often, the working directory is initialized as the folder in which the file representing the program or executable resides.
- The working directory allows unambiguous resolution of a relative path to a location in the file system.

Table 2.2 Special character sequences for paths

Special	Meaning
.	A single period means the "current directory"; so leading a string representation of a path allows us to refer to the current working directory
. .	Two periods refer to the parent directory relative to the current location in the traversal
~	A tilde refers to the home directory of the current user[5]

- Suppose the current working directory is:
  `/home/alice/Documents`
- The path "`./firstyear/ma135`" defines the traversal
  `/home/alice/Documents/firstyear/ma123`
- The path "`../Pictures/profile.jpg`" defines the traversal
  `/home/alice/Pictures/profile.jpg`

Opening a File in Python

- Before you can work with a file, you must open it.
- In Python we can use the built-in `open(.)` function. It takes, as arguments, a string specifying a path to a file, and a second string that specifies in what way the program will use the file, known as its *mode*.
- The `open(.)` function creates and returns a *file object*, which can be used to call other file support methods.

```
file = open('hello.txt', 'r')
```

hello.txt

```
Hello World!
```

Access Modes

Table 2.5 File open modes

Mode	Mode argument	Description
read-only	`'r'`	Open an existing file for reading only. An error occurs if the file does not exist, if the user does not have read permission for the file, or if a write operation is attempted
write	`'w'`	Open a file for writing. If the file exists, its contents are deleted so that it begins anew as an empty file. An error occurs if the user does not have permission to create a file in the current/specified directory, or permission to write to an existing file
append	`'a'`	Open a file for appending. Creates a new file if the path does not exist, but prepares for writes to happen at the end of an existing file. An error occurs if the user does not have read and write permission for an existing file

Access Modes

- In Python modes, we use 't' for text files, and 'b' for binary files. A mode can include both the form of access as well as the file type, so a mode of 'rt' would be used for read access to a text file, or a mode of 'wb' would be used for write access to a binary file.

- The default mode for open(.) is 'rt' and can be omitted if this is the desired mode.

```
open('hello.txt')
```

OS Module in Python

- The OS module in Python provides functions for interacting with the operating system. For example:

```
import os

os.getcwd()

os.chdir("DataFolder")

os.mkdir("C:\MyProject")
```

Get the current working directory

Change the current working directory

Make a directory

File Path as a String

- Suppose we have a directory, referred to by the string `datadir`, that contains the file `hello.txt`:

```
datadir = "/home/alice"
```

- We can use the `os.path.join(.)` method to concatenate various path components with exactly one directory separator ('/')

```
filepath = os.path.join(datadir, "hello.txt")
```

`os.path.join` Example

```
import os
path = "/home"
print(os.path.join(path,"User/Desktop","file.txt"))

path = "User/Documents"
print(os.path.join(path, "/home", "file.txt"))
```

Output:

```
/home/User/Desktop/file.txt
/home/file.txt
```

Error Handling

- Below we demonstrate a simple `try-except` block that can handle the case where the file does not exist, or the user does not have sufficient permissions for the specified access mode for the file being opened.

```
filepath = os.path.join(datadir, "hello.txt")
try:
    inputFile = open(filepath, 'rt')
    print("Success opening file")
except OSError:
    print("Error opening file")
```

Closing a File

- When a program is finished with a file, you must call the `close(.)` method to allow the operating systems to clean up and release resources associated with the file connection:

  `file.close()`

- **Closing files is essential** and forgetting to close files can have major negative consequences.

File Position

- Once you've opened a file, you'll want to read or write to it.
- We must understand the concept of the current *position* in the file, which is an integer that operates much like a cursor in a word processor. It indicates where the next operation will occur.
- The file position starts at 0, which is the position at the very beginning of the file. When we perform any of a set of read-type operations, the operation reads in some number of characters and the file position is advanced.
- In this way, a subsequent read-type operation will continue from the point where the last read finished.

File Object Methods

Table 2.6 File object methods

Method	Description
`read(n)`	Read from the file object starting from the current file position, returning a string if the file was opened as a text file. If `n` is specified, the read is limited to at most `n` characters. If not specified, the read should read until the end of the file. If the current position is at the end of the file (beyond the last character), this operation returns an empty string (`""`), i.e., a string of length zero
`tell()`	Return an integer giving the current file position
`readline()`	Read from the file object starting from the current file position, up to and including the next newline. Note that if a file ends with an unterminated line, the result of this function may not necessarily have a newline as its last character. Like the `read()` method, if the current position is already at the end of the file, this operation returns an empty string
`readlines()`	Read from the file object starting from the current file position, up through the end of the file. Accumulate the lines (including the line termination) read from the file into a list of strings. If the current position is already at the end of the file, this operation returns an empty list

Example

```
filepath = os.path.join(datadir, "hello.txt")
inputFile = open(filepath, 'rt')

filecontents = inputFile.read()

inputFile.close()

print("Character sequence from file:", repr(filecontents))
```

```
| Character sequence from file: 'Hello World!\n'
```

- The function `repr(.)` prints the representation of all the content of a string, including showing a `\n` for the newline.

Another Example

twolines.txt

```
First line
A second line
```

```
inputFile = open("twolines.txt")
filecontents = inputFile.read()
Print(filecontents)
print(repr(filecontents))
```

Output:

```
This is the first line.
This is the second line.
'This is the first line.\nThis is the second line.\n'
```

`readline(.)` Method

```
filepath = os.path.join(datadir, "twolines.txt")
inputFile = open(filepath, 'rt')

line1 = inputFile.readline()
line2 = inputFile.readline()
line3 = inputFile.readline()

inputFile.close()

print("1:", repr(line1))
print("2:", repr(line2))
print("3:", repr(line3))
```

Output:

```
| 1: 'First line\n'
| 2: 'A second line\n'
| 3: ''
```

`strip(.)` and `rstrip(.)`

- If we are processing a file by lines and want to remove whitespace and newline characters, we can use
  - `strip(.)` to eliminate both leading and trailing whitespace in a string
  - `rstrip(.)` to eliminate training whitespace in a string

```
line1 = inputFile.readline()
line2 = inputFile.readline()
```

could be replaced with

```
line1 = inputFile.readline().rstrip()
line2 = inputFile.readline().rstrip()
```

and the output becomes

```
| 1: 'First line'
| 2: 'A second line'
```

Another `strip(.)` Example

```
txt = "     banana    "
x = txt.strip()
print("of all fruits", x, "is my favorite")
```

Output:

```
of all fruits banana is my favorite
```

`split(.)` String Method

- The `split(.)` method breaks a string at the specified *separator* and returns a list of strings.
  - The default separator is any whitespace.

```
text = "Python is a fun programming language"
print(text.split())
```

Output:

```
['Python', 'is', 'a', 'fun', 'programming',
'language']
```

Another `split(.)` Example

```
grocery = 'Milk, Chicken, Bread'
```

Command	Output
`print(grocery.split())`	`['Milk,', 'Chicken,', 'Bread']`
`print(grocery.split(', '))`	`['Milk', 'Chicken', 'Bread']`
`print(grocery.split(','))`	`['Milk', ' Chicken', ' Bread']`
`print(grocery.split(':'))`	`['Milk, Chicken, Bread']`

Note the comma is included in the string

Note the space

File Processing with `split(.)`

twolines.txt

```
First line
A second line
```

```
filepath = os.path.join(datadir, "twolines.txt")
inputFile = open(filepath, 'rt')
```

```
filedata = inputFile.read()

filelines = filedata.split('\n')

print("Total length of the file contents:", len(filedata),
      "\nNumber of splits on newline:", len(filelines))
```

```
| Total length of the file contents: 25
| Number of splits on newline: 3
```

Methods for Ensuring a File is Closed

- A very common error in file processing is to fail to include the `close(.)` for an open file.
- There are two ways that you can use to ensure that a file is properly closed, even when encountering an error: `try-finally` block and a `with` statement.

`try-finally` block

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

This is executed, even if we experience an error

- If you're unfamiliar with the `try-finally` block, check out the webpage [Python Exceptions: An Introduction].

Using `with` Statement

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

- The `with` statement automatically takes care of closing the file once it leaves the `with` block, even in cases of error.
- **It is highly recommended that you use the `with` statement** as much as possible, as it allows for cleaner code and makes handling any unexpected errors easier for you.

`with` Continued

- Most likely, you'll also want to use the second positional argument, `mode`.
  - This argument is a string that contains multiple characters to represent how you want to open the file. The default is read-only mode as a text file `'rt'`:

```
with open('dog_breeds.txt', 'r') as reader:
    # Further file processing goes here
```

Character	Meaning
`'r'`	Open for reading (default)
`'w'`	Open for writing, truncating (overwriting) the file first
`'rb'` or `'wb'`	Open in binary mode (read/write using byte data)

`readlines(.)` Method

- The `readlines(.)` method reads, in a single invocation, a list containing all the separate lines in a file.

dog_breeds.txt

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

`readlines(.)` Example

```
with open('dog_breeds.txt', 'r') as file_obj:
    lines = file_obj.readlines()

for line in lines:
    print(line, end='')
```

Output:

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Recall that the `print(.)` function ends with a newline. By default, the value of this parameter is '`\n`', but you can end a print statement with any character. Here, we are ending it with no characters (this prevents extra carriage returns in the output)

Processing Files for Data

- We now consider iterating over a file in order to process data contained therein as one or more lists.
- We begin with the assumption that our file has one datum per line, and that the set of lines form a coherent collection.

Baby Names

- The file below contains the most popular female (baby) names, in rank order, registered in 2010 with the US Social Security Administration (SSA).

Baby_2010_female_name.txt

```
Isabella
Sophia
Emma
Olivia
Ava
```

- Our first objective is to create a function that processes the file and returns a single Python list with a name per entry in the file.

readNames(.) Example

Path to the input file

Number of lines to read

```
def readNames(path, numlines):
    namelist = []

    if not os.path.isfile(path):
        print("File not found")
        return namelist

    with open(path, 'r') as inputFile:

        for lineindex in range(numlines):

            line = inputFile.readline()
            name = line.strip()
            namelist.append(name)

    return namelist
```

Error checking to ensure file exists at supplied path

for-loop over number of lines requested

readNames(.) Example

```
path = os.path.join(datadir, "baby_2010_female_name.txt")
lines = 5
names = readNames(path, lines)
for name in names:
  print(name)
```

```
| Isabella
| Sophia
| Emma
| Olivia
| Ava
```

Read in All Lines

- What if we wanted to process all the lines (names), rather than a set number.
- The pattern of processing a file by lines in an iterative manners, for as many lines as exist in the file, is a *very* common practice.
- Python file objects accommodate this by allowing the file object itself to be an *iterator*.

Iterating Example

We recall that `line` terminates with `'\n'`, so we use `strip()` when processing it.

```
namelist = []

for line in inputFile:
  name = line.strip() # removes \n and other whitespace
  namelist.append(name)

for name in namelist:
  print(name)
```

The for-loop continues as long as there are lines, i.e., until it has reached the end-of-file (EOF)

```
| Jacob
| Ethan
| Michael
| Jayden
| William
```

- **This method is typically quicker and more memory efficient. Therefore, it is the most suggested way to process a file.**

Multiple Data Items per Line

- File often store data with each line representing a single case or observation, and then within the line, there are multiple values of some variables for the observation

Baby_2010_female_name.txt

```
Isabella	22913
Sophia	20643
Emma	17345
Olivia	17028
Ava	15433
```

- Each line contains the name and then the number of babies registered with that name.
- The name and the number are separated on each line with a single tab (`'\t'`)

Example

```
def readNamesCounts(filepath):
  namelist = []
  countlist = []

  with open(filepath, 'r') as inputFile:
    for line in inputFile:
      multivalue = line.strip()
      values = multivalue.split('\t')
      namelist.append(values[0])
      countlist.append(int(values[1]))

  return namelist, countlist
```

Example Continued

```
filename = "babynames_2010_female_namecount.txt"
filepath = os.path.join(datadir, filename)
names, counts = readNamesCounts(filepath)
print("Names:", names)
```

```
| Names: ['Isabella', 'Sophia', 'Emma', 'Olivia', 'Ava']
```

```
print("Counts:", counts)
```

```
| Counts: [22913, 20643, 17345, 17028, 15433]
```

JavaScript Object Notation

- The JavaScript Object Notation (JSON) is a text format used for storing and transporting data.
- JSON is often used when data is sent from a server to a webpage.
- It allows us to directly write data structures to a test file (or to a JSON-formatted string) and to subsequently read the data into a data structure.
- **JSON formatted data is extremely common in data systems is the de facto standard for information exchange.**

Benefits of JSON

- **Programming language independent:** There are libraries in Python, Java, C++, etc, for working with JSON strings and files.
- **Programmer familiarity:** The notation/syntax is a close match for how programmers define data.
- **Readability:** The format is text-based so the data is readable and editable directly in a text editor.
- **Lightweight but powerful:** The notation is simply but can handle complex situations.
- **Straightforward interface:** Writing a JSON file only requires a data structure and an appropriate library.

JSON Data Structure Example

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobbies": ["running", "sky diving", "singing"],
    "age": 35,
    "children": [
        {
            "firstName": "Alice",
            "age": 6
        },
        {
            "firstName": "Bob",
            "age": 8
        }
    ]
}
```

- This looks almost identical to a Python dictionary!

Python Supports JSON Natively

- Python comes with a built-in package called `json` for encoding and decoding JSON data.

```
import json
```

- Encoding is essentially writing data to disk while decoding is essentially reading data into memory.

Serializing JSON

- Python objects to JSON:

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

Serialization Example

```
import json
data = {
    "president": {
        "name": "Zaphod Beeblebrox",
        "species": "Betelgeusian"
    }
}

with open("data_file.json", "w") as write_file:
    json.dump(data, write_file)
```

Deserializing JSON

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

- The result of the `load(.)` call could return any of the allowed data types from the conversion table.
- We will explore JSON more later in the course.

Tabular Model: Structure and Formats

Def: A ***data model*** is a notation for describing data or information, generally consisting of three parts—the *structure* of the data, the *operations* to obtain and update data within the model, and the *constraints* that, within the model, limit the data in various ways.

- **Tabular data** is data that is structured into rows, each of which contains information about some thing. Each row contains the same number of cells (although some of these cells may be empty), which provide values of the properties of the thing described by the row.
- In tabular data, cells within the same column provide values for the same property of the things described by each row. This is what differentiates tabular data from other line-oriented formats.

Top Baby Names

Year	Sex	Name	Count
2018	Male	Liam	19837
2018	Female	Emma	18688
2017	Male	Liam	18798
2017	Female	Emma	19800
2016	Male	Noah	19117
2016	Female	Emma	19496
2015	Male	Noah	19635
2015	Female	Emma	20455
2014	Male	Noah	19305
2014	Female	Emma	20936

- There are alternate ways that this data set could be represented in a tabular format.

Alternate Formats

Table 6.2 Top female baby names

Year	Name	Count
2018	Emma	18688
2017	Emma	19800
2016	Emma	19496
2015	Emma	20455
2014	Emma	20936

Table 6.3 Top male baby names

Year	Name	Count
2018	Liam	19837
2017	Liam	18798
2016	Noah	19117
2015	Noah	19635
2014	Noah	19305

Table 6.4 Top baby names by year

	2014	2015	2016	2017	2018
Female	Emma	Emma	Emma	Emma	Emma
Male	Noah	Noah	Noah	Liam	Liam

Table 6.5 Top baby name application counts by year

	2014	2015	2016	2017	2018
Female	20936	20455	19496	19800	18688
Male	19305	19635	19117	18798	19837

Alternate Formats?

- We can see that there are many ways of representing the same data values in two-dimensional tabular format.
  - Depending on which aspects of the data you want to highlight in a presentation, one format might be better than another.
- However, from a *processing-not-presentational* perspective, it should be clear that we need to have a common standard for representing data.
- It has been found that keeping exactly one variable in each column allows us to effectively use column-vector operations and allows us to “slice and dice” the rows, partitioning based on values of other values.

Tidy Data

Def: A data set in the tabular data model is said to be ***tidy data*** if it conforms to the following.

1. Each column represents exactly one variable of the data set. (*TidyData*1)

2. Each row represents exactly one unique (relational) mapping, that maps from a set of givens (the values of the independent variables) to the values of the dependent variables. (*TidyData2*)

3. Exactly one table is used for each set of mappings involving the same independent variables. (*TidyData3*)

Variable

- In a data set, a ***variable*** is a symbolic name with an associated value that can be a number, an integer, a real-valued number, a string, or it can be ***categorical*** which means that the value comes from a discrete set of valid values.

- Categorical variables often yield a partitioning of the data.

Tidy Data?

Table 6.2 Top female baby names

Year	Name	Count
2018	Emma	18688
2017	Emma	19800
2016	Emma	19496
2015	Emma	20455
2014	Emma	20936

Table 6.3 Top male baby names

Year	Name	Count
2018	Liam	19837
2017	Liam	18798
2016	Noah	19117
2015	Noah	19635
2014	Noah	19305

- The data is split into *two* tables.
- Both tables are defined by the value of a variable (Female and Male), which violates TidyData3.

Tidy Data?

Table 6.4 Top baby names by year

	2014	2015	2016	2017	2018
Female	Emma	Emma	Emma	Emma	Emma
Male	Noah	Noah	Noah	Liam	Liam

Table 6.5 Top baby name application counts by year

	2014	2015	2016	2017	2018
Female	20936	20455	19496	19800	18688
Male	19305	19635	19117	18798	19837

- We can see columns headed up by year *values*, so that we are violating TidyData1—the columns here are not a variable.
- We also violate TidyData3 by separating our data set into two different tables based on the variables of name and count.

Tidy Data?

Table 6.7 Top baby names with female/male columns

	FemaleName	FemaleCount	MaleName	MaleCount
2018	Emma	18688	Liam	19837
2017	Emma	19800	Liam	18798
2016	Emma	19496	Noah	19117
2015	Emma	20455	Noah	19635
2014	Emma	20936	Noah	19305

- Note that the columns like `FemaleName` do not represent exactly one variable.
- The name of the column itself shows two parts, one of which is a value of a categorical variable, and the other part is a variable that is then repeated in `MaleName`.
- This is a violation of TidyData1.

Tidy Data?

Table 6.1 Top baby names

Year	Sex	Name	Count
2018	Male	Liam	19837
2018	Female	Emma	18688
2017	Male	Liam	18798
2017	Female	Emma	19800
2016	Male	Noah	19117
2016	Female	Emma	19496
2015	Male	Noah	19635
2015	Female	Emma	20455
2014	Male	Noah	19305
2014	Female	Emma	20936

- Only the original table, Table 6.1, satisfies all three requirements of tidy data.

Format

Def: A ***format*** (*file* format or document *format*) is a specification for the encoding of information for a data structure into a sequence of characters or bytes, in a file or across a stream, to be communicated between a creator/writer/provider and a corresponding consumer/reader/client.

Examples: `jpg`, `mp3`, `docx`, `mov`

- The most common format for tabular data is **the *comma-separated value* (CSV)** format (and its many variations).

CSV Format

- CSV strikes a delicate balance between being readable by humans and readable by computers.
- CSV is a tabular format consisting of rows of data, each row containing multiple cells. Rows are (usually) separated by line terminators, so each row corresponds to one line.
- The CSV format assumes exactly one tabular structure per separate file.
  - This is more constrained than a spreadsheet model, in which a single file might contain multiple "worksheets" that could each hold its own tabular structure.
- CSV is a "text-based" format, i.e., a CSV file is a text file.

Typical Structure of CSV

- One “header row,” which gives the names of the columns.
- May “rows” of data.
  - One row of data is encoded in one line of the file.
  - Each row is called a *record*.
  - Values are separated by commas.
- Sometimes a ‘#’ is used for comment lines.

CSV Example

- Header row:

```
LastName, FirstName, Initial, Course, Time
```

- Data (subsequent rows):

```
Lauer, Hugh, C, CS-1004, 0800
Wills, Craig, E, CS-3013, 1200
Hamel, Glynis, M, CS-1101, 1000
```

Tabular Data in Python

- We could represent tabular data in Python with native built-in data structures using a *dictionary of lists* (DoL) or a *lists of lists* (LoL).
- As we will see, these approaches have distinct disadvantages.

Dictionary of Lists (DoL)

- We could represent tabular data in Python using a *dictionary of lists* (DoL):

Year	Sex	Name	Count
2018	Male	Liam	19837
2018	Female	Emma	18688
2017	Male	Liam	18798
2017	Female	Emma	19800
2016	Male	Noah	19117
2016	Female	Emma	19496

- This is effective for accessing and manipulating entire columns but working with rows requires additional code.

```
topnames = {'year': [2018, 2018, 2017, 2017, 2016, 2016],
            'sex': ['Male', 'Female', 'Male',
                    'Female', 'Male', 'Female'],
            'name': ['Liam', 'Emma', 'Liam', 'Emma',
                     'Noah', 'Emma'],
            'count': [19837, 18688, 18798, 19800, 19117, 19496]}
```

Dictionary of Lists (DoL)

```
topnames = {'year': [2018, 2018, 2017, 2017, 2016, 2016],
            'sex': ['Male', 'Female', 'Male',
                    'Female', 'Male', 'Female'],
            'name': ['Liam', 'Emma', 'Liam', 'Emma',
                     'Noah', 'Emma'],
            'count': [19837, 18688, 18798, 19800, 19117, 19496]}
```

With this representation, it is easy to get data for an entire column:

```
print(topnames['year'])
```

```
| [2018, 2018, 2017, 2017, 2016, 2016]
```

But getting data for a row is more involved. Consider the row with index 3:

```
row_index = 3
print(topnames['year'][row_index], topnames['sex'][row_index],
    topnames['name'][row_index], topnames['count'][row_index])
```

```
| 2017 Female Emma 19800
```

List of Lists (LoL)

- We could also represent tabular data in Python using a *list of lists* (LoL):

Year	Sex	Name	Count
2018	Male	Liam	19837
2018	Female	Emma	18688
2017	Male	Liam	18798
2017	Female	Emma	19800
2016	Male	Noah	19117
2016	Female	Emma	19496

- While effective for dealing by rows, there is more work for manipulating columns.

```
topnames = [[2018, 'Male', 'Liam', 19837],
            [2018, 'Female', 'Emma', 18688],
            [2017, 'Male', 'Liam', 18798],
            [2017, 'Female', 'Emma', 19800],
            [2016, 'Male', 'Noah', 19117],
            [2016, 'Female', 'Emma', 19496]]
columns = ['year', 'sex', 'name', 'count']
```

List of Lists (LoL)

```
topnames = [[2018, 'Male', 'Liam', 19837],
            [2018, 'Female', 'Emma', 18688],
            [2017, 'Male', 'Liam', 18798],
            [2017, 'Female', 'Emma', 19800],
            [2016, 'Male', 'Noah', 19117],
            [2016, 'Female', 'Emma', 19496]]
columns = ['year', 'sex', 'name', 'count']
```

In this representation, we can easily access and refer to a particular row, as long as we know its index:

```
row_index = 5
print(topnames[row_index])
```

```
| [2016, 'Female', 'Emma', 19496]
```

List of Lists (LoL)

```
topnames = [[2018, 'Male', 'Liam', 19837],
            [2018, 'Female', 'Emma', 18688],
            [2017, 'Male', 'Liam', 18798],
            [2017, 'Female', 'Emma', 19800],
            [2016, 'Male', 'Noah', 19117],
            [2016, 'Female', 'Emma', 19496]]
columns = ['year', 'sex', 'name', 'count']
```

To access a column, we need to know its index. In this example, we may need to know that the count values are at index 3 within the list in each row, and to access all the values in a particular column, we must refer to all of the individual rows:

```
col_index = 3
print(topnames[0][col_index], topnames[1][col_index],
      topnames[2][col_index], topnames[3][col_index],
      topnames[4][col_index], topnames[5][col_index])
```

```
| 19837 18688 18798 19800 19117 19496
```

Pandas

- The need for Python data structures to handle tabular data has led to the development of the `pandas` module.

Chapters 6 and 7
Introduction to Pandas and Access Operations

- Pandas is an open-source Python library for data analysis that was originally developed by Wes McKinney in 2008. The name Pandas stands for "Python Data Analysis Library" and is derived from the term "panel data," which is an econometrics terms for multidimensional structured data sets.

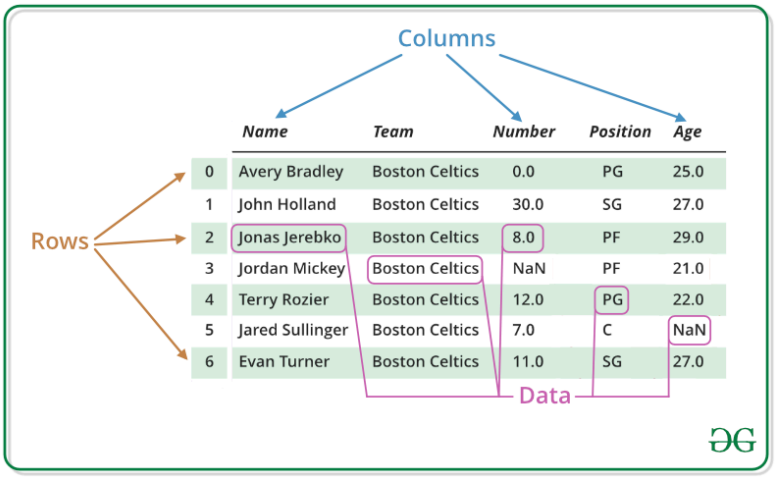
- Pandas gives Python the ability to work with spreadsheet-like data and is built on top of NumPy.

- Pandas introduces two new data types to Python: `Series` and `DataFrame`.

- The `DataFrame` is a two-dimensional data structure where data is aligned in a tabular fashion in rows and columns. Essentially, a Pandas `DataFrame` is an in-memory representation of an Excel spreadsheet in Python. A Pandas `Series` is a single column of a `DataFrame`.

- Each column (`Series`) of a `DataFrame` has to be the same type (just like a NumPy array), whereas each row can contain mixed types.

- A simple `DataFrame` is illustrated below (source):



- At a basic level, Pandas `DataFrame` objects can be thought of as enhanced versions of two-dimensional NumPy arrays in which the rows and columns are identified with labels rather than simple integer indices.

Installing and Importing Pandas

- The Pandas package is not part of the standard Python release and may need to be installed separately. If you use pip, you can install Pandas with:

```
pip3 install pandas
```

- Once Pandas is installed, you need to import the Pandas library as follows:

```
import pandas as pd
```

- It is common practice to import Pandas with the alias '`pd`'.

The Pandas `Series` Object

- The `Series` object is a one-dimensional array of indexed data. It can be created from a list or array as follows.

```
import pandas as pd
data = pd.Series([10,30,-6,9])
print(data)
```

```
0    10
1    30
2    -6
3     9
```

- As you can see, the `Series` object contains an array of data (of any NumPy data type) with associated indices (which can be numbers, strings, or any other data types). We can access these with the `values` and `index` attributes.

```
print(data.values)
print(data.index)
```

```
[10 30 -6  9]

RangeIndex(start=0, stop=4, step=1)
```

- It appears that a `Series` is interchangeable with a one-dimensional NumPy array. However, it comes with some additional functionality and can be thought of as a column of a spreadsheet. The `Series` can have a `name`, and most importantly, it has an explicitly defined `index` associated with the `values`.

- By default, the `Series` will be indexed from `0` to `n` where `n = size-1`.

```
age = pd.Series([10,13,9,16],index=['Bill','Peixin','Carlo','Thahn'],name='kids_ages')
print(age)
```

```
Bill      10
Peixin    13
Carlo      9
Thahn     16
Name: ages, dtype: int64
```

- We can access the elements of a `Series` by either the index *position* or *label*:

```
>>> age[1]
13

>>> age['Peixin']
13
```

- You can also construct a `Series` from a dictionary as illustrated in the example below (source):

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
print(population)
```

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

- Here are some of the common attributes of a `Series` object:

Attribute	Returns
`name`	The name of the `Series` object
`dtype`	The data type of the `Series` object
`shape`	Dimensions of the `Series` object in a tuple of the form (# of rows,)
`index`	The `Index` object that is part of the `Series` object
`columns`	The name of the columns (as an `Index` object)
`values`	The data in the `Series` object

- Note that the `Index` class makes the `Series` class more powerful than a NumPy array because it gives us row labels.

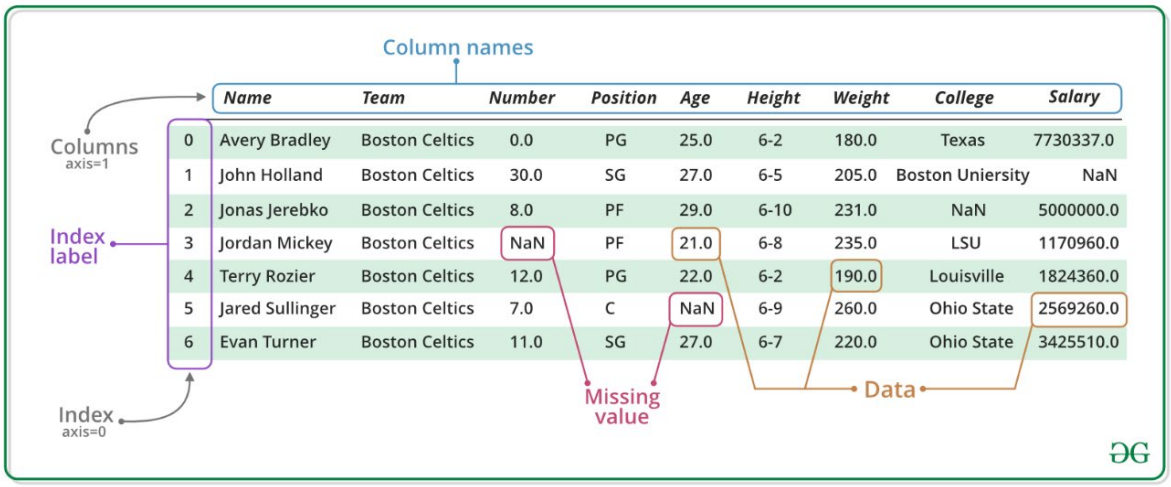
- Many of the methods and functions that operate on a NumPy array will also operate on a `Series`. For example:

```
>>> age.mean()
12.0
>>> age.max()
16
>>> age.std()
3.1622776601683795
```

- The `Pandas.Series` documentation contains more information on how to create a `Series` object and the full list of attributes and methods that are available.

The Pandas `DataFrame` Object

- The `DataFrame` object is a two-dimensional array with both flexible row indices and flexible column names. This is illustrated graphically below (source).



- Most of the time, a Pandas `DataFrame` will be created by importing data from existing storage, such as a CSV file, an SQL database, or an Excel spreadsheet.

Creating a `DataFrame`

- A Pandas `DataFrame` can be constructed in a number of ways, but we will demonstrate constructing one from a dictionary of lists (DoL).

```
dataDoL={'name':['Bella','Charlie','Lucy','Coooper','Max','Stella','Bernie'],
        'breed':['Labrador','Poodle','ChowChow','Schnauzer','Labrador',
                 'Chihuahua','St.Bernard'],
        'color': ['Brown','Black','Brown','Gray','Black','Tan','White'],
        'height_cm':[56,43,46,49,59,18,77],
        'weight_kg':[24,24,24,17,29,2,74],
        'date_of_birth':['2013-07-01','2016-09-16','2014-08-25',
                         '2011-12-11','2017-01-28','2015-04-20',
                         '2018-02-27']}

dogs = pd.DataFrame(dataDoL)
print(dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Pandas does allow for row labels, but since we have not specified the labels, Pandas has provided a set of integers (0–6) as the row labels.

- Notice that the order of the columns is not guaranteed because Python dictionaries are unordered. We could force the order of the columns by using an *ordered dictionary*. Specifically, we can use the `OrderedDict` class from the `collections` module.

```
from collections import OrderedDict

data=OrderedDict(
       {'name':['Bella','Charlie','Lucy','Coooper','Max','Stella','Bernie'],
        'breed':['Labrador','Poodle','ChowChow','Schnauzer','Labrador',
                 'Chihuahua','St.Bernard'],
        'color': ['Brown','Black','Brown','Gray','Black','Tan','White'],
        'height_cm':[56,43,46,49,59,18,77],
        'weight_kg':[24,24,24,17,29,2,74],
        'date_of_birth':['2013-07-01','2016-09-16','2014-08-25',
                         '2011-12-11','2017-01-28','2015-04-20',
                         '2018-02-27']})

dogs = pd.DataFrame(data)

print(dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Now the columns are in the order we originally specified.

- The follow are some commonly used `DataFrame` attributes.

Attribute	Returns
`dtypes`	The data type of each column
`shape`	Dimensions of the `DataFrame` object in a type of the form (# of rows, # of columns)
`index`	The `Index` object along the rows of the `DataFrame`.
`columns`	The name of the columns (as an `Index` object)
`values`	The data in the `DataFrame` object
`empty`	Check if the `DataFrame` object is empty.

- We can specify row labels by assigning values to the `index` attribute.

- As an example, consider the following data set of country-based indicators. We will assign the country codes as the row labels.

```
indicatorDoL = {
  'country': ['Canada', 'China', 'India',
              'Russia', 'United States', 'Vietnam'],
  'pop': [36.26, 1378.66, 1324.17, 144.34, 323.13, 94.59],
  'gdp': [1535.77, 11199.15, 2263.79, 1283.16, 18624.47,
          205.28],
  'life': [82.30, 76.25, 68.56, 71.59, 78.69, 76.25],
  'cell': [30.75, 1364.93, 1127.81, 229.13, 395.88, 120.60]}

codes = pd.Index(['CAN', 'CHN', 'IND', 'RUS', 'USA', 'VNM'],
                 name='code')

indicators = pd.DataFrame(indicatorDoL, index=codes)

print(indicators)
```

```
             country      pop       gdp   life     cell
code
CAN           Canada    36.26   1535.77  82.30    30.75
CHN            China  1378.66  11199.15  76.25  1364.93
IND            India  1324.17   2263.79  68.56  1127.81
RUS           Russia   144.34   1283.16  71.59   229.13
USA    United States   323.13  18624.47  78.69   395.88
VNM          Vietnam    94.59    205.28  76.25   120.60
```

- Below we demonstrate accessing the `values`, `columns`, and `index` of a `DataFrame`.

```
print(dogs.values)
print(dogs.columns)
print(dogs.index)
```

```
[['Bella' 'Labrador' 'Brown' 56 24 '2013-07-01']
 ['Charlie' 'Poodle' 'Black' 43 24 '2016-09-16']
 ['Lucy' 'ChowChow' 'Brown' 46 24 '2014-08-25']
 ['Coooper' 'Schnauzer' 'Gray' 49 17 '2011-12-11']
 ['Max' 'Labrador' 'Black' 59 29 '2017-01-28']
 ['Stella' 'Chihuahua' 'Tan' 18 2 '2015-04-20']
 ['Bernie' 'St.Bernard' 'White' 77 74 '2018-02-27']]

Index(['name', 'breed', 'color', 'height_cm', 'weight_kg', 'date_of_birth'],
dtype='object')

RangeIndex(start=0, stop=7, step=1)
```

Importing a CSV File and Exploring a DataFrame

- Recall that a **CSV (comma separated values)** files are one of the most popular file formats for importing and exporting tabular data such as spreadsheets and databases. For each row, the column information is separated with a comma. However, the comma is not the only delimiter. Some files are delimited by a tab (TSV file) 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, including a text editor.

- To import a CSV file, we can use the Pandas `read_csv(.)` function. We will explore the full functionality of this function later, but we will use it in its most basic form here.

- Download the `gapminder.csv` file from Teams and put it in your working directory. This is a data set prepared by Jennifer Bryan and is a subset of the gapminder teaching package for R. We can then import the file as follows.

```
df = pd.read_csv("gapminder.csv")
print("The data type of df is " + str(type(df)))

print('The shape of the dataframe is ' + str(df.shape))
print(df.head())
```

```
The data type of df is <class 'pandas.core.frame.DataFrame'>

The shape of the dataframe is (1704, 6)

       country continent  year  lifeExp       pop   gdpPercap
0  Afghanistan      Asia  1952   28.801   8425333  779.445314
1  Afghanistan      Asia  1957   30.332   9240934  820.853030
2  Afghanistan      Asia  1962   31.997  10267083  853.100710
3  Afghanistan      Asia  1967   34.020  11537966  836.197138
4  Afghanistan      Asia  1972   36.088  13079460  739.981106
```

- We used the Python `type` function to verify that `df` is a Pandas `DataFrame` and we used the `shape` *attribute* of the `DataFrame` object to obtain the number of rows (1704) and the number of columns (6).

- Finally, to get a sense of the `DataFrame` contents we used the Pandas `head(.)` function that returns the first few rows of the `DataFrame`.

- We can use the `info(.)` method to display the names of the columns, the data types they contain, and whether they have any missing values.

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
lifeExp      1704 non-null float64
pop          1704 non-null int64
gdpPercap    1704 non-null float64
dtypes: float64(2), int64(2), object(2)
```

- Note that each column has a single data type, but they do not all share the same type.

- The `describe(.)` method computes some summary statistics for numerical columns, including the mean, standard deviation, and the five-number summary. Note that `count` is the number of non-missing values in each column.

```
print(df.describe())
```

```
             year      lifeExp           pop      gdpPercap
count  1704.00000  1704.000000  1.704000e+03    1704.000000
mean   1979.50000    59.474439  2.960121e+07    7215.327081
std      17.26533    12.917107  1.061579e+08    9857.454543
min    1952.00000    23.599000  6.001100e+04     241.165877
25%    1965.75000    48.198000  2.793664e+06    1202.060309
50%    1979.50000    60.712500  7.023596e+06    3531.846989
75%    1993.25000    70.845500  1.958522e+07    9325.462346
max    2007.00000    82.603000  1.318683e+09  113523.132900
```

Access Operations

- Access operations are those that *read* or *query* data values out of a table. The most common access operations are:

  1. **Single-element access:** We may want to access a single data value (at the intersection of a row and column).

  2. **Column access:** We seek a subset of a data frame that is based on one or more columns, which we call a *projection* of the desired columns. For example, we may want a subtable obtained by projecting the `year` and `lifeExp` columns (with all rows included).

  3. **Single row:** We may wish to select a single row, containing the values of all columns in that row.

  4. **Multiple rows:** We want a subset of the data consisting of all the columns, but a limited set of the rows.

  5. **Subset of rows and columns:** The most general form of access operation would allow us to both project a subset of the columns *and* filter for a particular subset of the rows.

- We can **select a single column** of a `DataFrame` by using the name of the `DataFrame`, followed by square brackets with a column name inside. When we project a single column in this way, we obtain a `Series` as the projected column.

```
Name = dogs['name']
print(Name)
print(type(Name))
```

```
0      Bella
1    Charlie
2       Lucy
3    Coooper
4        Max
5     Stella
6     Bernie
Name: name, dtype: object

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

- We can also project a single column by making the *column name* an *attribute* of the `DataFrame` object as demonstrated below.

```
print(dogs.name)
```

```
0      Bella
1    Charlie
2       Lucy
3    Coooper
4        Max
5     Stella
6     Bernie
Name: name, dtype: object
```

- To **select (project) multiple columns**, you need two pairs of square brackets. The outer brackets are responsible for subsetting the `DataFrame`, and the inner square brackets are creating a list of column names to subset.

```
subtable = dogs[['height_cm', 'breed']]
print(subtable)
type(subtable)
```

```
   height_cm       breed
0         56    Labrador
1         43      Poodle
2         46    ChowChow
3         49   Schnauzer
4         59    Labrador
5         18   Chihuahua
6         77  St.Bernard

pandas.core.frame.DataFrame
```

- First, note that we requested the columns in an order different from their order in the `DataFrame` and this requested order was respected in the result. Second, note that the type returned is a `DataFrame`.

- There are a number of ways to **access rows**. One of the easiest methods is to use slicing notation. Keep in mind that row (and column) numbers start at 0 and up to the number of rows (or columns) minus one. The syntax for a slice is:

```
start : end [:stride]
```

- Recall our `indicators` `DataFrame`:

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

```
subrows = indicators[3:5]
print(subrows)
```

```
            country     pop       gdp   life    cell
code
RUS          Russia  144.34   1283.16  71.59  229.13
USA   United States  323.13  18624.47  78.69  395.88
```

- The output above corresponds to the rows indexed by 3 and 4 (since 5 is not included). We can select rows starting at the beginning and proceeding up to, but not including index 2, as follows.

```
print(indicators[:2])
```

```
     country      pop       gdp   life     cell
code
CAN   Canada    36.26   1535.77  82.30    30.75
CHN    China  1378.66  11199.15  76.25  1364.93
```

- We can also access rows using the labels (`index`) attribute within the slice.

```
print(indicators['IND':'USA'])
```

```
            country      pop       gdp   life     cell
code
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
```

A stride of –1 reverses the order

```
print(indicators['VNM':'RUS':-1])
```

```
            country     pop       gdp   life    cell
code
VNM         Vietnam   94.59    205.28  76.25  120.60
USA   United States  323.13  18624.47  78.69  395.88
RUS          Russia  144.34   1283.16  71.59  229.13
```

Row Selection by Condition

- **Another common method to select rows is to use a logical condition** inside of square brackets to filter against. For example, we can find all the dogs whose height is greater than 50 centimeters as follows.

```
print(dogs[dogs['height_cm'] > 50])
```

```
     name       breed  color  height_cm  weight_kg date_of_birth
0   Bella    Labrador  Brown         56         24    2013-07-01
4     Max    Labrador  Black         59         29    2017-01-28
6  Bernie  St.Bernard  White         77         74    2018-02-27
```

- Let's tease out this statement and look at what `dogs['height_cm'] > 50` returns.

```
print(dogs['height_cm'] > 50)
```

```
0     True
1    False
2    False
3    False
4     True
5    False
6     True
Name: height_cm, dtype: bool
```

- This statement returns a `Series` with a `dtype` of `bool`. Therefore, we can subset values using labels and indices, but also by supplying a vector of Boolean values.

- We can also subset rows based on text data. In the example below, we use the double equal sign in the logical condition to filter the dogs that are Labradors.

```
print(dogs[dogs['breed']=='Labrador'])
```

```
    name     breed  color  height_cm  weight_kg date_of_birth
0  Bella  Labrador  Brown         56         24    2013-07-01
4    Max  Labrador  Black         59         29    2017-01-28
```

- We can also subset based on dates. Below we filter all the dogs born before 2015. Notice that the dates are in quotes and are written in the *international standard date* format.

```
print(dogs[dogs["date_of_birth"] > "2015-01-01"])
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
1  Charlie      Poodle  Black         43         24    2016-09-16
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- To **subset rows that meet multiple conditions**, you can combine conditions using logical operations, such as the "`&`" operator as seen below.

```
is_lab = dogs['breed'] == 'Labrador'
is_brown = dogs['color'] == 'Brown'
print(dogs[is_lab & is_brown])
```

```
    name     breed  color  height_cm  weight_kg date_of_birth
0  Bella  Labrador  Brown         56         24    2013-07-01
```

- Note that we could have also done this in one line of code, but you would need to add parentheses around each condition.

- If you want to filter on multiple values of a categorical variable, the easiest way is to use the `isin(.)` method. This takes in a list of values to filter for. Below, we check if the color of a dog is black or brown.

```
is_black_or_brown = dogs[dogs['color'].isin(['Black','Brown'])]
print(dogs[is_black_or_brown])
```

```
      name     breed  color  height_cm  weight_kg date_of_birth
0    Bella  Labrador  Brown         56         24    2013-07-01
1  Charlie    Poodle  Black         43         24    2016-09-16
2     Lucy  ChowChow  Brown         46         24    2014-08-25
4      Max  Labrador  Black         59         29    2017-01-28
```

Selection using `loc` and `iloc`

- For any Pandas `DataFrame`, we can use `.loc[]` and `.iloc[]` to perform practically any data selection. **Note that `loc` is label-based, while `iloc` is integer index based**. The syntax is as follows.

  dataframe`.loc[`rowspec, colspec`]`

  dataframe`.iloc[`rowspec, colspec`]`

  Inside the access operator, *rowspec* and *colspec* are row and column *specifiers*.

- We provide some examples of using `loc` and `iloc` below.

```
print(indicators.loc['VNM', 'gdp'])
```

```
205.28
```

Here we access a single element, and the result is a scalar with the data type of the entry.

```
print(indicators.loc[:, 'pop'])
```

```
code
CAN      36.26
CHN    1378.66
IND    1324.17
RUS     144.34
USA     323.13
VNM      94.59
Name: pop, dtype: float64
```

Here we are accessing a single column, `pop`, and all the associated rows (`:`).

```
print(indicators.iloc[1:3, :])
```

```
     country      pop       gdp   life     cell
code
CHN    China  1378.66  11199.15  76.25  1364.93
IND    India  1324.17   2263.79  68.56  1127.81
```

Here we use `iloc` to access the rows indexed from 1 to 3 (but not including 3) and all the associated columns.

```
print(indicators.loc['USA', ['country', 'life', 'cell']])
```

```
country    United States
life               78.69
cell              395.88
Name: USA, dtype: object
```

Here we access a subset of the columns (`country`, `life`, and `cell`) for a single row (`USA`). Note that this returned a `Series` (and thus is now a column!).

```
print(indicators.iloc[4, [0, 3, 4]])
```

```
country    United States
life               78.69
cell              395.88
Name: USA, dtype: object
```

Here we access the same information as the last example, but we do so using the row and column indices (with `iloc`) as opposed to the labels.

```
print(indicators.loc[ indicators.gdp < 2000, ['country', 'pop']])
```

```
      country     pop
code
CAN    Canada   36.26
RUS    Russia  144.34
VNM   Vietnam   94.59
```

Here we access the two columns `country` and `pop` for the rows for which the `gdp` is less 2000.

Creating New Columns

- It is common to create new columns derived from existing columns. Creating and adding new columns can go by many names, including *mutating* a `DataFrame`, *transforming* a `DataFrame`, and *feature engineering*.

- Suppose we wanted to create a new column that has each dog's height in meters instead of centimeters. We can accomplish this with the command

  ```
  dogs["height_m"] = dogs["height_cm"]/100.
  ```

  On the left-hand side of the equals, we use square brackets with the name of the new column we want to create. On the right-hand side, we have the calculation. Note that we are utilizing vectorization! Below we show this in action.

```
dogs["height_m"] = dogs["height_cm"]/100
print(dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth  height_m
0    Bella    Labrador  Brown         56         24    2013-07-01      0.56
1  Charlie      Poodle  Black         43         24    2016-09-16      0.43
2     Lucy    ChowChow  Brown         46         24    2014-08-25      0.46
3  Coooper   Schnauzer   Gray         49         17    2011-12-11      0.49
4      Max    Labrador  Black         59         29    2017-01-28      0.59
5   Stella   Chihuahua    Tan         18          2    2015-04-20      0.18
6   Bernie  St.Bernard  White         77         74    2018-02-27      0.77
```

- Notice that both the existing column and the new column we just created are in the `DataFrame`.

- Now let's add a column that has the body mass index (BMI) of the dogs. Recall that BMI is calculated using the formula

  BMI = (weight in kg) / (height in m)^2.

  We can once again use vectorization to create the column as follows.

```
dogs['bmi'] = dogs['weight_kg'] / dogs['height_m']**2
print(dogs[['name','breed','color','date_of_birth','bmi']])
```

```
      name       breed  color date_of_birth         bmi
0    Bella    Labrador  Brown    2013-07-01   76.530612
1  Charlie      Poodle  Black    2016-09-16  129.799892
2     Lucy    ChowChow  Brown    2014-08-25  113.421550
3  Coooper   Schnauzer   Gray    2011-12-11   70.803832
4      Max    Labrador  Black    2017-01-28   83.309394
5   Stella   Chihuahua    Tan    2015-04-20   61.728395
6   Bernie  St.Bernard  White    2018-02-27  124.810255
```

- Notice that in the code above we only choose to print some of the columns.

Activity 02 - Intro to Descriptive Statistics in Python

DATA 200 - Database Systems and Data Management for Data Analytics
Dickinson College

In this notebook, we will learn how to perform some basic descriptive statistics in Python, including both graphical and numerical summaries of data. In particular, we will learn the basics of `Matplotlib`, which is Python's fundamental plotting library. It is extremely flexible and gives the user full control over all elements of the plot. Interestingly, `Matplotlib` emulates Matlab like graphs and visualizations.

As we learned in Introduction to Data Science, data visualization is vital step in any analysis. Simply put, visualizations are the easiest way to analyze and absorb information. They help in identifying patterns, relationships, and outliers in data. Furthermore, visualizations are a precursor to many high-level data analysis for Exploratory Data Analysis (EDA) and Machine Learning(ML).

Importing Matplotlib

Rather than importing the whole `matplotlib` package, it is common to only import the `pyplot` module using the dot (.) notation; this reduces the amount of typing we need to do in order to access what we need, and we don't take upmore space in memory with code we won't use. Note that `pyplot` is traditionally aliased as `plt`.Please run the cell below.

In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

The Basics of Matplotlib

`Matplotlib` creates visualizations with the *plot* commands; however, we won't see the visualization until we request to see it. It is done in this fashion so that we can continually tweak the visualization with additional code, until we are ready to finalize it.

`Matplotlib` uses the `plt.show()` function to display a visualization. It must be called after each visualization we create.

Below, we load in the `national_parks.csv` data file into the `DataFrame` called `parks`. This data set contains the number visitors to the Badlands, Grand Canyon, and Bryce Canyon national parks, from 1961-2017. Please run the code cell below.

In [2]:
```
parks = pd.read_csv('data/national_parks.csv')
parks.head()
```

Out[2]:

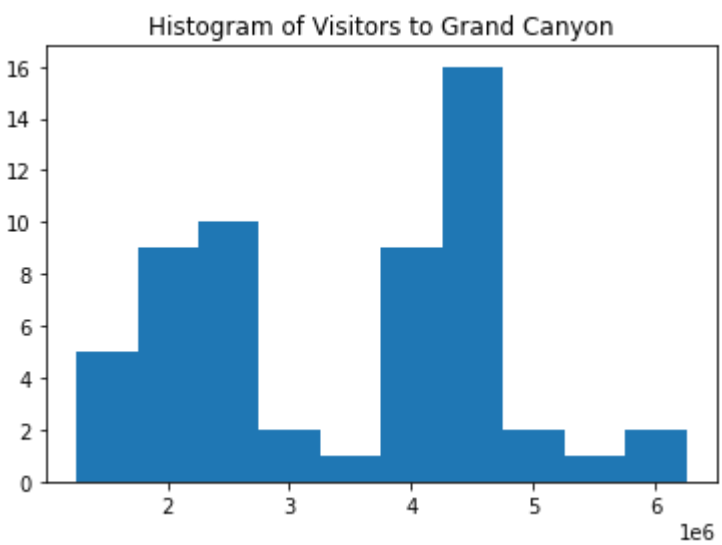
	Year	Badlands	GrandCanyon	BryceCanyon
0	1961	833300	1253000	264800
1	1962	1044800	1447400	251000
2	1963	1074000	1539500	289500
3	1964	1079800	1576600	300300
4	1965	1091300	1689200	366800

Histogram

Recall that a histogram is a graphical representation of the distribution of a data set. You can review histograms here. To create a frequency histogram with `matplotlib`, we use the `hist()` function. Study and run the code cell below.

In [3]:

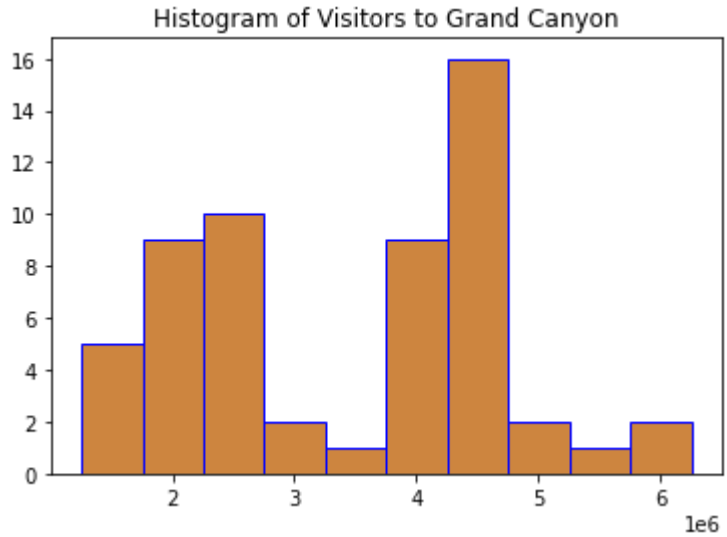
```
plt.hist(parks['GrandCanyon'],
         bins=10)
plt.title('Histogram of Visitors to Grand Canyon')
plt.show()
```



There are many ways to customize your histogram. Study and run the code cell below.

In [4]:

```
plt.hist(parks['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10)
plt.title('Histogram of Visitors to Grand Canyon')
plt.show()
```

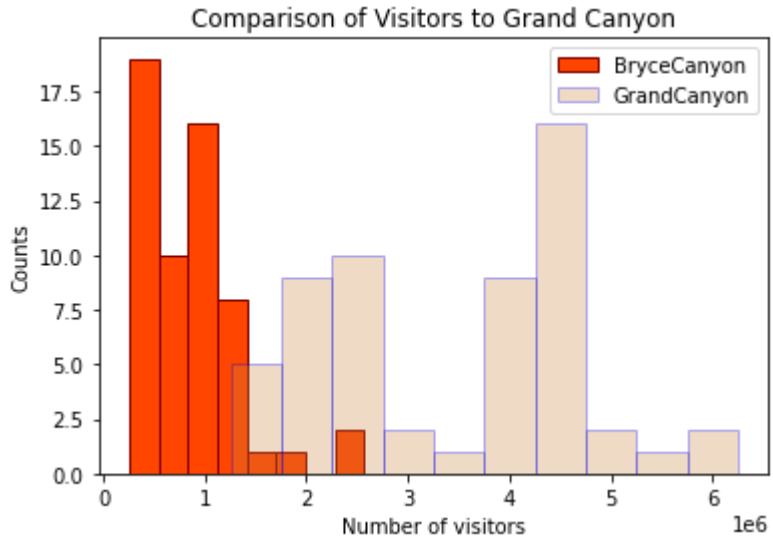


Multiple Histograms

Multiple (frequency) histograms are useful in understanding the relationship between two variables. In the code cell below, we simply call the `hist()` function two times before calling `show()`. We also add in x- and y-labels using the `xlabel(.)` and `ylabel(.)` functions. Finally, we add a legend using the `legend(.)` function. Study and run the code cell below.

In [5]:

```
plt.hist(parks['BryceCanyon'],
         facecolor='orangered',
         edgecolor='maroon',
         bins=8, label="BryceCanyon")

plt.hist(parks['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10,
         alpha = 0.3,
         label="GrandCanyon")

plt.title('Comparison of Visitors to Grand Canyon')
plt.ylabel("Counts")
plt.xlabel("Number of visitors")
plt.legend(loc='upper right')

plt.show()
```



We can see from the graph above that the Grand Canyon has comparably more visitors than Bryce Canyon.

Scatterplot

Scatter plots are useful for determining if there is a relationship for bivariate data. You can review scatterplots here.

Run the code cell below where we read in the `countries.csv` data set, which contains information on life expectancy, population, and GDP per capita for all countries from 1952-2007.

In [6]:

```
countries = pd.read_csv('data/countries.csv')
countries.head()
```

Out[6]:

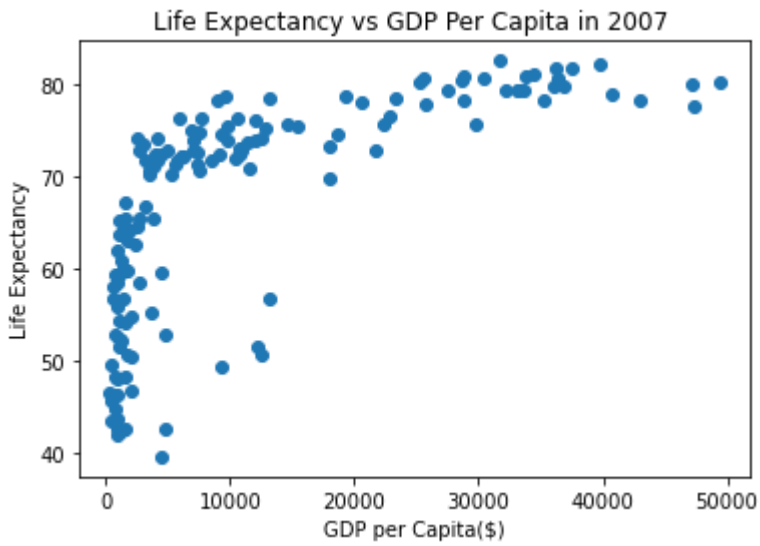
	country	continent	year	lifeExpectancy	population	gdpPerCapita
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

Let's create a scatterplot of the life expectancy versus the GDP Per Capita in 2007. To create a scatterplot, we use the `plt.scatter()` function, which requires two arrays of the same length. The first array is for the values of the x-axis and the second if for the y-axis. Study and run the code cell below (note that we first extract the data for 2007).

In [7]:

```
data_2007 = countries[countries['year'] == 2007]

plt.scatter(data_2007['gdpPerCapita'],data_2007['lifeExpectancy'])
plt.title('Life Expectancy vs GDP Per Capita in 2007 ')
plt.xlabel('GDP per Capita($)')
```

```
plt.ylabel('Life Expectancy')
plt.show()
```



This scatterplot clearly shows that as GDP of a country increases, life expectancy too increases.

Numerically Summarizing Data

Now let's see how we can use Python to numerically summarize data. Run the code cell below, which loads in the `MotorTrend.csv` file. This data set was extracted from a 1974 *Motor Trend* magazine and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-1974 models).

In [8]:

```
cars = pd.read_csv('data\MotorTrend.csv')
cars.head()
```

Out[8]:

	Name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	manual	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	manual	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	manual	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	auto	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	auto	3	2

Recall that we can use the `describe()` function to get the summary statistics of all the columns--run the code cell below.

In [9]:

```
cars.describe()
```

Out[9]:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	g
count	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000

	mpg	cyl	disp	hp	drat	wt	qsec	vs	g
mean	20.090625	6.187500	230.721875	146.687500	3.596563	3.217250	17.848750	0.437500	3.687
std	6.026948	1.785922	123.938694	68.562868	0.534679	0.978457	1.786943	0.504016	0.737
min	10.400000	4.000000	71.100000	52.000000	2.760000	1.513000	14.500000	0.000000	3.000
25%	15.425000	4.000000	120.825000	96.500000	3.080000	2.581250	16.892500	0.000000	3.000
50%	19.200000	6.000000	196.300000	123.000000	3.695000	3.325000	17.710000	0.000000	4.000
75%	22.800000	8.000000	326.000000	180.000000	3.920000	3.610000	18.900000	1.000000	4.000
max	33.900000	8.000000	472.000000	335.000000	4.930000	5.424000	22.900000	1.000000	5.000

Thus, for example, the average miles-per-gallon (`mpg`) of the cars in the data set is 20.09, while the average horsepower (`hp`) is 146.68.

We could compute the summary statistics for just the mileage (`mpg`) data of our cars as follows--run the code cell below.

In [10]:
```
cars.mpg.describe()
```

Out[10]:
```
count    32.000000
mean     20.090625
std       6.026948
min      10.400000
25%      15.425000
50%      19.200000
75%      22.800000
max      33.900000
Name: mpg, dtype: float64
```

We could also select individual statistics from say, horsepower, as demonstrated below--study and run the code cell below.

In [11]:
```
print('The mean horsepower is ' + str(cars.hp.mean()))
print('The standard deviation of the horsepower is ' + str(cars.hp.std()))
print('The 25th percentile of horsepower is ' + str(cars.hp.quantile(.25)))
```

```
The mean horsepower is 146.6875
The standard deviation of the horsepower is 68.56286848932059
The 25th percentile of horsepower is 96.5
```

Computing Summary Statistics by Categorical Variables

Suppose we wanted to compute summary statistics of `mpg` based upon transmission type `am` (automatic or manual). For example, if you wanted to compute the summary statistics for `mpg` specifically for cars with automatic transmission you could do the following--run the code cell below (do you understand how it works?).

In [12]:
```
cars.mpg[cars.am == 'auto'].describe()
```

Out[12]:
```
count    19.000000
mean     17.147368
std       3.833966
min      10.400000
25%      14.950000
50%      17.300000
75%      19.200000
max      24.400000
Name: mpg, dtype: float64
```

Using the `groupby()` function, we can compute summary information for a numeric variable based on all possible values of a categorical variable (factor). You can learn more about the `groupby()` function here.

In the code cell below, we first extract a subtable from cars that only contains the columns for `mpg` and `am`. Then we *stack* the `groupby('am')` call (using a .) and finally *stack* the `describe()` function. Essentially, this splits the data by the categorical variable `am`, that is, it will separate the data for `auto` and `manual`. Study and run the code cell below.

In [13]:
```
cars[['mpg', 'am']].groupby('am').describe()
```

Out[13]:

	mpg							
	count	mean	std	min	25%	50%	75%	max
am								
auto	19.0	17.147368	3.833966	10.4	14.95	17.3	19.2	24.4
manual	13.0	24.392308	6.166504	15.0	21.00	22.8	30.4	33.9

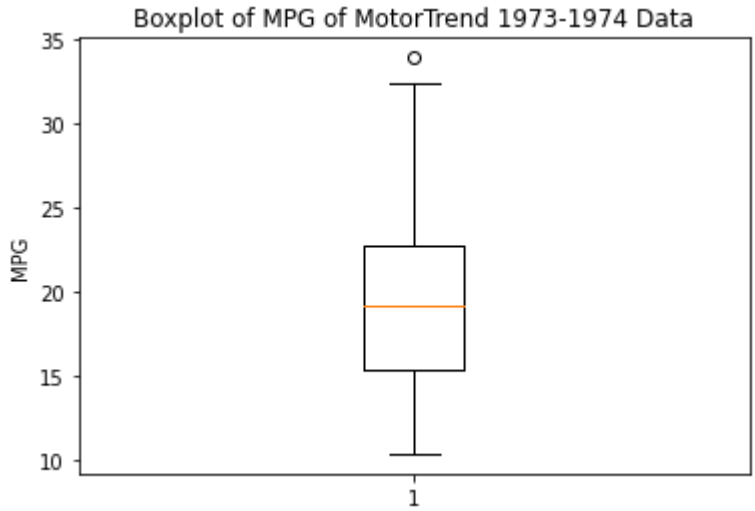
The table generated above allows us to easily see how cars with automatic transmissions compare with cars with manual transmitions. For example, we can see that cars with manual transmisions provide a larger `mpg` on average (24.39 compared to 17.14).

Boxplots

Recall that boxplots provide a graphical display of the five number summary: mininum value, lower-quartile, median, upper-quartile, and maximum value. In addition, a boxplot identifies potential outliers. You can review boxplots here.

To create a boxplot, we can use the `boxplot()` function. In the code cell below, we create a boxplot of the `mpg` of the cars. Please run the code cell now.

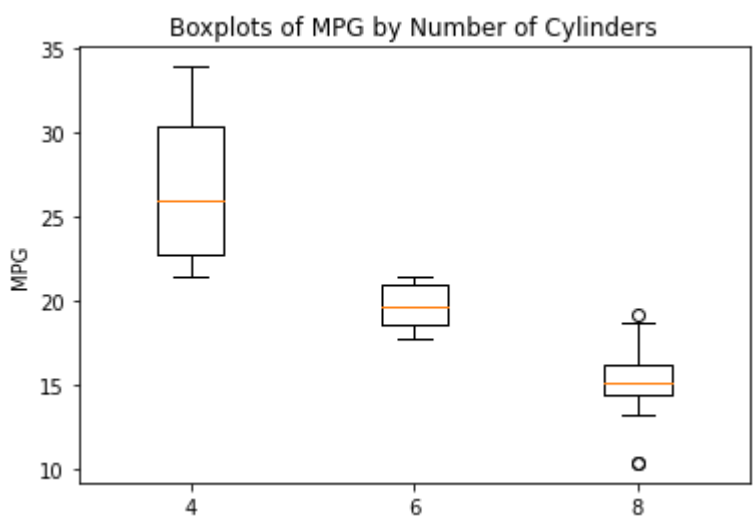
In [14]:
```
plt.boxplot(cars.mpg)
plt.title('Boxplot of MPG of MotorTrend 1973-1974 Data')
plt.ylabel('MPG')
plt.show()
```



Now suppose we wanted to compare the mileage (`mpg`) between cars with different cylinders using *side-by-side boxplots*. We can accomplish this by creating a list of columns. For example, `cars[cars.cyl == 4].mpg` is the column of `mpg` values for those cars with `4` cylinders. Study and run the code cell below. Note that if you pass in a list of columns to `boxplot()` , it will create a boxplot for each column (using the same scale). Study and run the code cell below

In [15]:

```
plt.boxplot([cars[cars.cyl == 4].mpg,
             cars[cars.cyl == 6].mpg,
             cars[cars.cyl == 8].mpg],
            labels=['4','6','8'])
plt.title('Boxplots of MPG by Number of Cylinders')
plt.ylabel('MPG')
plt.show()
```



We can clearly see from the graph above that the gas mileage of the cars decreases significantly as the number of cylinders increases.

Numerically Summarizing Categorical Data

Now let us focus on describing categorical data (recall that categorical variables are often referred to as *factors*). Specifically, we will focus on the `am` variable of our `cars` data set.

Discovering relationships between variables is the fundamental goal of data analysis. *Frequency tables* are a basic tool you can use to explore data and get an idea of the relationships between categorical variables. A frequency table is just a data table that shows the counts of one or more categorical variables.

One-Way Tables

We create frequency tables in Pandas using the `pd.crosstab()` function. You can read more about the `crosstab()` function here. The function takes one or more array-like objects as indexes or columns and then constructs a new `DataFrame` of variable `counts` based on the supplied arrays. Let's make a one-way table of the transmision variable `am` --study and run the code cell below.

In [16]:
```
pd.crosstab(index = cars.am, columns='Count')
```

Out[16]:

col_0	Count
am	
auto	19
manual	13

Thus, our data set contains 19 cars with automatic transmissions and 13 cars with manual transmissions.

Now suppose we wanted to extract the proportion of the data that belongs to each catergory. With a one-way table, you can do this by dividing each table value by the total number of records in the table. Study and run the code cell below.

In [17]:
```
oneway_am = pd.crosstab(index = cars.am, columns='Count')
oneway_am/oneway_am.sum()
```

Out[17]:

col_0	Count
am	
auto	0.59375
manual	0.40625

Thus, 40.6% of the cards have manual transmissions.

Creating a Two-Way Table

A *two-way table* is a table that describes two categorical variables together. It contains the number of cases for each combination of the categories in both variables. The analysis of categorical variables always starts with tables, and Pandas give you a whole toolset to work with them. But first, you have to create the tables.

Suppose we wanted to analyze the number of gears (3, 4, or 5) with type of transmission (automatic or manual). You can do this using the `crosstab()` function with **two arguments**. The `index` argument referes to the rows and the `columns` argument refers to the columns. Study and run the code cell below.

In [18]:
```
pd.crosstab(index = cars.am, columns = cars.gear)
```

Out[18]:

gear	3	4	5
am			
auto	15	4	0
manual	0	8	5

We can improve the readability of this table by adding some labels as follows. Study and run the code cell below.

In [19]:
```
twoway = pd.crosstab(index = cars.am, columns = cars.gear)
twoway.columns = ['3','4','5']
twoway.index = ['auto', 'manual']

twoway
```

Out[19]:

	3	4	5
auto	15	4	0
manual	0	8	5

This tells us, for example, that among cars with automatic transmission, 15 have 3 gears, 4 have 4 gears, and 0 have 5 gears.

Run the code cell below to load in the `smokers.csv` file. This data set contains 356 people that have been polled on their smoking status (`Smoke`) and their socioeconomic status (`SES`). For each person is was determined whether or not they are current smokers, former smokers, or have never smoked. Also, for each person, their socioeconomic status was determined (`low`, `middle`, or `high`).

In [20]:
```
smoke_data = pd.read_csv('data\smoker.csv')
smoke_data.head()
```

Out[20]:

	Smoke	SES
0	former	High

	Smoke	SES
1	former	High
2	former	High
3	former	High
4	former	High

Let's create a two-way table called `smoketable` that summarizes the data. Study and run the code cell below.

In [21]:

```
smoketable = pd.crosstab(index = smoke_data.Smoke, columns = smoke_data.SES)
smoketable.columns = ['High', 'Low', 'Middle']
smoketable.index = ['current', 'former', 'never']

smoketable
```

Out[21]:

	High	Low	Middle
current	51	43	22
former	92	28	21
never	68	22	9

Thus, there are 51 people who are current smokers and are in the high SES.

Conditional Distributions

Suppose we are interested in computing the *conditional distribution of SES given Smoking Status*. That is, we want the entires in our two-way table as percentages of the column totals (thus, the rows will sum to 1). We can accomplish this by using the `normalize` parameter of `crosstab()`, which normalizes by dividing all values by the sum of the values. In the code below, we set `normalize = 'index'`, which means we want to normalize along the rows. Study and run the code cell below.

In [22]:

```
smokeCond = pd.crosstab(index = smoke_data.Smoke,
                        columns = smoke_data.SES,
                        normalize = 'index')

smokeCond.columns = ['High', 'Low', 'Middle']
smokeCond.index = ['current', 'former', 'never']

smokeCond
```

Out[22]:

	High	Low	Middle
current	0.439655	0.370690	0.189655
former	0.652482	0.198582	0.148936
never	0.686869	0.222222	0.090909

Thus, for example, the table tells us that among current smokers, 43.9% have a high SES, 37.1% have a Low SES, and 18.9% have a Middle SES.

We could compute the *conditional distribution of Smoking Status given SES* by setting `normalize = 'columns'`. Note that this will ensure that the *columns* sum to 1. Study and run the code cell below.

In [23]:

```
smokeCond = pd.crosstab(index = smoke_data.Smoke,
                        columns = smoke_data.SES,
                        normalize = 'columns')

smokeCond.columns = ['High', 'Low', 'Middle']
smokeCond.index = ['current', 'former', 'never']

smokeCond
```

Out[23]:

	High	Low	Middle
current	0.241706	0.462366	0.423077
former	0.436019	0.301075	0.403846
never	0.322275	0.236559	0.173077

Thus, for example, the table tells us that among those individuals who are in a Low SES, 46.2% are current smokers, 30.1% are former smokers, and 23.6% have never smoked.

Creating a Categorical Classification Based on Data

Suppose we wanted to classify our vehicles in the `MotorTrend` data set as having Poor, Good, or Great gas mileage. Specifically, all vehicles with mpg less 18 should be classified as Poor, all vehicles with mpg between 18 and 22 will be classified as Good, and all vehicles with mpg greater than 22 will be classified as Great. We can accomplish this using Python's `loc()` function, which helps to retrieve data values from a dataset. You can read more about the `loc()` function here.

In the code below, we first create a new column in the `cars` `DataFrame` called `mpgClass` and initialize all the values to be `temp`. Then we use the `loc()` function to find the cars that have `mpg` in different ranges and assign the appropriate category to the `mpgClass` column.

Study and run the code cell below.

In [24]:

```
# Create a new column called mpgClass and initialize all values to be 'temp'
cars['mpgClass'] = 'temp'

# Use the loc() function to assign 'Poor', 'Good', or 'Great' to then
# entries in the new column 'mpgClass' depending on the value of mpg
cars.loc[cars.mpg <= 18, 'mpgClass'] = 'Poor'
cars.loc[(cars.mpg > 18) & (cars.mpg <= 22), 'mpgClass'] = 'Good'
cars.loc[cars.mpg > 22, 'mpgClass'] = 'Great'
```

```
cars
```

Out[24]:

	Name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpgClass
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	manual	4	4	Good
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	manual	4	4	Good
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	manual	4	1	Great
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	auto	3	1	Good
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	auto	3	2	Good
5	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	auto	3	1	Good
6	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	auto	3	4	Poor
7	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	auto	4	2	Great
8	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	auto	4	2	Great
9	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	auto	4	4	Good
10	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	auto	4	4	Poor
11	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	auto	3	3	Poor
12	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	auto	3	3	Poor
13	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	auto	3	3	Poor
14	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	auto	3	4	Poor
15	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	auto	3	4	Poor
16	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	auto	3	4	Poor
17	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	manual	4	1	Great
18	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	manual	4	2	Great
19	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	manual	4	1	Great
20	Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	auto	3	1	Good
21	Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	auto	3	2	Poor
22	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	auto	3	2	Poor
23	Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	auto	3	4	Poor
24	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	auto	3	2	Good
25	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	manual	4	1	Great
26	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	manual	5	2	Great
27	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	manual	5	2	Great
28	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	manual	5	4	Poor
29	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	manual	5	6	Good
30	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	manual	5	8	Poor

	Name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpgClass
31	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	manual	4	2	Good

You can see that we have our new column and it as correctly assigned the categories based on our criteria. We will use this technique in the next homework assignment.

Chapters 8
Advanced Operations in Pandas

Aggregating a Single Pandas `Series`

- Data analysis typically requires us to summarize data from a subset of a variable, called *aggregating*.

- The simplest form of aggregation occurs when we have a single `Series` (often a single column projection of a `DataFrame`). The most common aggregation functions/methods are provided in the table below.

Table 8.1 Aggregation function/methods

Method	Description
`mean()`	Arithmetic mean, not including missing values
`median()`	Value occurring halfway through the population, omitting missing values
`sum()`	Arithmetic sum of the non-missing values
`min()`	Smallest value in the set
`max()`	Largest value in the set
`nunique()`	Number of unique values in the set
`size()`	Size of the set
`count()`	Number of non-missing values in the set

- Two other handy methods are in the table below.

Table 8.2 Aggregation to `Series` methods

Method	Description
`unique()`	Construct a subset `Series` consisting of the unique values from the source `Series`
`value_count()`	Construct a `Series` of integers capturing the number of times each unique value is found in a source `Series`; the `Index` of the new `Series` consists of the unique values from the source `Series`

- We demonstrate some of these methods below on the `dogs` `DataFrame` from the previous chapter:

```
      name       breed  color  height_cm  weight_kg date_of_birth
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

```
avg_weight = dogs['weight_kg'].mean()
max_height = dogs['height_cm'].max()
min_weight = dogs['weight_kg'].min()
numColors = dogs['color'].nunique()

print('Mean weight = ', avg_weight)
print('Max height = ', max_height)
print('Min weight = ', min_weight)
print('Number of different colors = ', numColors)
```

```
Mean weight =  27.714285714285715
Max height =  77
Min weight =  2
Number of different colors =  5
```

Aggregating a `DataFrame`

- We are now interested in grouping rows into discrete partitions, aggregating each partition into a row of computed values, and combining the result.

- In Pandas, we use the `agg()` method of a `DataFrame` to perform this general form of the aggregation operation. The argument to `agg()` must convey, for each column of the original `DataFrame`, what aggregation function or functions to perform.

- Recall our `indicators` `DataFrame`:

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

- We demonstrate on the `indicators` `DataFrame` below, performing two aggregations on the `life` column (`min` and `median`), and just the `median` aggregation on the `cell` column. The result is a `DataFrame`, and the row labels on the result are the names of the aggregation function.

```
table = indicators.agg({'life':['min', 'median'], 'cell':'median'})
print(table)
```

```
         life     cell
min     68.56      NaN
median  76.25  312.505
```

There is a `NaN` value here because we didn't request the minimum from the `cell` column

- If a data frame aggregation only has a single aggregation for any given column, the result can be a one-dimensional objective (`Series`) with an entry for each of the columns being aggregated. In the following example, we compute the `min` aggregation for the `life` and `cell` columns.

```
avg = indicators.agg({'life': 'min', 'cell': 'min'})
print(avg)
```

```
life    68.56
cell    30.75
dtype: float64
```

Aggregating Selected Rows

- Often, we wish to characterize, through aggregation, a subset of a data frame. This "subset and aggregation" is also a critical piece in understanding the more general group partitioning and aggregation that we consider later.

- Consider the `topnames.csv` file (available on Teams). This dataset consists of the top male and female names, and the number (`count`) of babies with that name, from 1880 to 2018. The actual CVS file looks as follows:

```
year,sex,name,count
1880,Female,Mary,7065
1880,Male,John,9655
1881,Female,Mary,6919
1881,Male,John,8769
...
2017,Female,Emma,19800
2017,Male,Liam,18798
2018,Female,Emma,18688
2018,Male,Liam,19837
```

- Let's read this data into a `DataFrame` and take a look at the first and last 10 rows (to compare which names were popular from different eras).

```
topnames = pd.read_csv("topnames.csv")

print(topnames.head(10))
print()
print(topnames.tail(10))
```

```
   year     sex  name  count
0  1880  Female  Mary   7065
1  1880    Male  John   9655
2  1881  Female  Mary   6919
3  1881    Male  John   8769
4  1882  Female  Mary   8148
5  1882    Male  John   9557
6  1883  Female  Mary   8012
7  1883    Male  John   8894
8  1884  Female  Mary   9217
```

John was the most popular male name in 1880 with 9655 babies given that name

```
9  1884    Male  John   9388
     year     sex  name  count
268  2014  Female  Emma  20936
269  2014    Male  Noah  19305
270  2015  Female  Emma  20455
271  2015    Male  Noah  19635
272  2016  Female  Emma  19496
273  2016    Male  Noah  19117
274  2017  Female  Emma  19800
275  2017    Male  Liam  18798
276  2018  Female  Emma  18688
277  2018    Male  Liam  19837
```

- Suppose we are interested in finding the minimum, maximum, and median of the `count` column, but only for rows where Mary was the top name. In the code below, we first create a subtable `mary_rows` that only contains the rows from `topnames` that correspond to `Mary`. When then use the `agg()` function to find the `min`, `max`, and `median` of the `count` column.

```
mary_rows = topnames[topnames['name'] == 'Mary']
table = mary_rows.agg({'count': ['min', 'max', 'median']})
print(table)
```

```
          count
min      6919.0
max     73985.0
median  54423.0
```

General Partitioning and `groupby`

- In mathematics, a ***partition*** of a set is a grouping of its elements into subsets in such a way that every element is included in exactly one subset.

- In data analytics, we may want to form a partition of a `DataFrame` using a *groupby* operation (pronounced "group by"). Each of these partitions then could have an aggregation performed upon it, and so, from each partition, we obtain a single *row* of values. In general, for the aggregation, we need to yield a result that is a single row per partition.

- The Pandas method for the partitioning/groupby operation is called `groupby()`. The first parameter of `groupby()` is the name of the column (or columns) to use for the partitioning. The result is a `DataFrameGroupBy` object, which is different from a `DataFrame` because it, in essence, is a set of data frames. We demonstrate below.

```
groupby_sex = topnames.groupby('sex')
groupby_name = topnames.groupby('name')

print("The number of groups by sex is: ", len(groupby_sex))
print("The number of groups by name is: ", len(groupby_name))
type(groupby_name)
```

```
The number of groups by sex is:  2
The number of groups by name is:  18
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

- Thus, we can see that the partitioning by `sex` yields 2 groups, while the partitioning by `name` yields 18 groups (which is the same as the unique number of names from the data set). Finally, note that the type of object returned by `groupby()` is a `DataFrameGroupBy`.

- Suppose for the `groupby_name` partition, we wanted to compute the median and sum for `count`. This could be accomplished as follows.

```
namegroup = groupby_name.agg({'count': ['median', 'sum']})
print(namegroup)
```

```
            count
           median      sum
name
Ashley    40965.5    81931
David     85929.0    85929
Emily     25104.0   294508
Emma      19648.0   118188
Isabella  22609.5    45219
Jacob     25339.0   370779
James     86224.0  1056228
Jennifer  57117.0   859209
Jessica   47884.0   397962
John       9032.5   861403
Liam      19317.5    38635
Linda     85723.5   508407
Lisa      53355.0   420572
Mary      54423.0  3098428
Michael   68117.5  3084824
Noah      19211.0    76314
Robert    60699.0  1041984
Sophia    21842.0    65378
```

Sorting

- We can change the order of the rows by sorting using the `sort_values(.)` method. To utilize this method, simply pass in the column name that you want to sort by.

- To see this in action, we will once again consider our `dogs DataFrame`.

```
sorted_dogs = dogs.sort_values("weight_kg")
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Note that the `DataFrame` is now sorted by `weight_kg` in *ascending* ordered. Further note that the `sort_value(.)` returns a sorted `DataFrame` but does not actually modify the `DataFrame` from which it was called.

- We can set the `ascending` argument to `False` so that it will sort the data in descending order.

```
sorted_dogs = dogs.sort_values("weight_kg", ascending=False)
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
6   Bernie  St.Bernard  White         77         74    2018-02-27
4      Max    Labrador  Black         59         29    2017-01-28
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
5   Stella   Chihuahua    Tan         18          2    2015-04-20
```

- We can sort by multiple variables by passing a list of columns to `sort_values(.)`. For example, we first sort by weight, then by height below.

```
sorted_dogs = dogs.sort_values(["weight_kg", "height_cm"])
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
0    Bella    Labrador  Brown         56         24    2013-07-01
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Note that now Charlie, Lucy, and Bella are ordered from shortest to tallest, even though they weigh the same. To change the direction the values are sorted in, pass a list to the `ascending` argument indicating which direction sorting should be done for each variable:

```
sorted_dogs = dogs.sort_values(["weight_kg", "height_cm"],
                               ascending=[True, False])
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
0    Bella    Labrador  Brown         56         24    2013-07-01
2     Lucy    ChowChow  Brown         46         24    2014-08-25
1  Charlie      Poodle  Black         43         24    2016-09-16
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Now Charlie, Lucy, and Bella are ordered from tallest to shortest.

Operations to Delete Columns and Rows

- The easiest way to delete a single column from a `DataFrame` is to use Python's `del` statement, using an argument specifying the single column.

- Before we start deleting columns, let's make a copy of our `indicators DataFrame` using the `copy()` method as follows.

```
ind2 = indicators.copy()
print(ind2)
```

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

- We can remove the `cell` column from the `ind2 DataFrame` using `del` as follows.

```
del ind2['cell']
print(ind2)
```

```
            country      pop       gdp   life
code
CAN          Canada    36.26   1535.77  82.30
CHN           China  1378.66  11199.15  76.25
IND           India  1324.17   2263.79  68.56
RUS          Russia   144.34   1283.16  71.59
USA   United States   323.13  18624.47  78.69
VNM         Vietnam    94.59    205.28  76.25
```

- Alternately, we can use the `pop()` method to delete a single column. This method deletes and modifies the data structure in place and returns the element that was deleted. We demonstrate below for the `gdp` column.

```
ind2 = indicators.copy()

gdp_series = ind2.pop('gdp')

print(ind2)
print(gdp_series)
```

```
            country      pop   life     cell
code
CAN          Canada    36.26  82.30    30.75
CHN           China  1378.66  76.25  1364.93
IND           India  1324.17  68.56  1127.81
RUS          Russia   144.34  71.59   229.13
USA   United States   323.13  78.69   395.88
VNM         Vietnam    94.59  76.25   120.60

code
CAN     1535.77
CHN    11199.15
IND     2263.79
RUS     1283.16
USA    18624.47
VNM      205.28
Name: gdp, dtype: float64
```

Note that the `gdp` column has been dropped from `ind2`, but it has been stored in the `Series` `gdp_series`, which we can use later if needed.

Deleting Multiple Columns

- We can delete multiple columns using the `drop()` method of a `DataFrame`, where the first argument is a single column label, or a list of column labels. We demonstrate below where we delete columns `cell` and `life` from the original table.

```
ind2 = indicators.copy()
ind2.drop(['cell','life'], axis=1, inplace=True)
print(ind2)
```

```
            country      pop       gdp
code
CAN          Canada    36.26   1535.77
CHN           China  1378.66  11199.15
IND           India  1324.17   2263.79
RUS          Russia   144.34   1283.16
USA   United States   323.13  18624.47
VNM         Vietnam    94.59    205.28
```

- The `axis=1` argument indicates the dimension for what is to be dropped, in this case, 1 indicates the column (and an axis of 0 would indicate the row dimension). The `inplace=True` indicates that the method should *not* create a new `DataFrame`, but rather, drop the columns of the calling `DataFrame`. Note that `inplace=False` will create a new `DataFrame` and return the result but will leave the calling `DataFrame` alone.

Row Deletion

- We can use the `drop()` method to remove rows by specifying `axis=0`. We demonstrate below by dropping the rows corresponding to the USA and Russia.

```
ind2 = indicators.copy()
ind2.drop(['USA','RUS'], axis=0, inplace=True)
print(ind2)
```

```
      country      pop       gdp   life     cell
code
CAN    Canada    36.26   1535.77  82.30    30.75
CHN     China  1378.66  11199.15  76.25  1364.93
IND     India  1324.17   2263.79  68.56  1127.81
VNM   Vietnam    94.59    205.28  76.25   120.60
```

Adding a Column

- One of the most common mutation operations is adding a new column to an existing `DataFrame`. We have already seen this earlier in the course, but we demonstrate it again below where we create a new column representing a 15% growth in GDP.

```
ind2 = indicators.copy()
ind2['growth'] = ind2.gdp*1.15
print(ind2)
```

```
            country      pop       gdp   life     cell      growth
code
CAN          Canada    36.26   1535.77  82.30    30.75   1766.1355
CHN           China  1378.66  11199.15  76.25  1364.93  12879.0225
IND           India  1324.17   2263.79  68.56  1127.81   2603.3585
RUS          Russia   144.34   1283.16  71.59   229.13   1475.6340
USA   United States   323.13  18624.47  78.69   395.88  21418.1405
VNM         Vietnam    94.59    205.28  76.25   120.60    236.0720
```

- Since the `growth` column doesn't exist, Pandas creates the column and uses vectorization to fill in the data by multiplying the corresponding entries of the `gdp` column by 1.15.

- Another commonly used technique to create a new column is to use the `apply()` function to perform an operation on the elements of a vector.

- Recall that a lambda function is a small anonymous function that is defined without a name. For example,

  `to_caps = lambda s: s.upper()`

  will define a lambda function called `to_caps()` that takes a string and converts it to all capitals using the `upper()` string method.

- In the example below, we start with the original `indicators DataFrame`, drop the `life`, `cell`, `pop`, and `gdp` columns (for readability) and then use the lambda function `top_caps()` to create a new column with the country names in all capitals.

```
to_caps = lambda s: s.upper()

ind2 = indicators.copy().drop(['life','cell','pop','gdp'],
                              axis=1, inplace=False)
ind2['countryCaps'] = ind2['country'].apply(to_caps)
print(ind2)
```

```
            country    countryCaps
code
CAN          Canada         CANADA
CHN           China          CHINA
IND           India          INDIA
RUS          Russia         RUSSIA
USA   United States  UNITED STATES
VNM         Vietnam        VIETNAM
```

Updating Columns

- Instead of creating an entirely new column in a data frame, we often want to change values in an existing column. To accomplish, we specify an existing column on the left-hand side of an assignment operator and a valid column-vector operation on the right-hand side. Below we demonstrate by updating the `life` column of the original `indicators` `DataFrame`.

```
ind2 = indicators.copy()
print('The original `life` column is')
print(ind2.life)
ind2.life = ind2.life + 0.5
print('\nThe updated `life` column is')
print(ind2.life)
```

```
The original `life` column is
code
CAN    82.30
CHN    76.25
IND    68.56
RUS    71.59
USA    78.69
VNM    76.25
Name: life, dtype: float64

The updated `life` column is
code
CAN    82.80
CHN    76.75
IND    69.06
RUS    72.09
USA    79.19
VNM    76.75
Name: life, dtype: float64
```

Combining Tables: Concatenating Along the Row Dimension

- Given two or more data frames, we may be interested in combining them into a single data frame that has some union or intersection of the rows, columns, and values of the source tables.

- Our examples below make use of the assumption that for the tables being combined, the source data frames represent different information. When combining along the row dimension, that implies that there is no intersection of value combination of the independent variable.

- In *normalized* tidy data, a ***row index is meaningful*** if the index is composed of the independent variable(s) of the data set. So, if a data set has one independent variable, the value of that variable is unique per row, and the remaining columns give the values of the dependent variables.

- Consider a subset of the `indicators` `DataFrame` consisting of the columns `country`, `pop`, `gdp`, and `life` for the rows `CHN`, `IND`, and `USA`:

```
ind1 = indicators.loc[['CHN','IND','USA'], :'life']
print(ind1)
```

```
            country      pop       gdp   life
code
CHN           China  1378.66  11199.15  76.25
IND           India  1324.17   2263.79  68.56
USA   United States   323.13  18624.47  78.69
```

- Now consider the new data frame below which contains values for the same columns as `ind1`, but for the countries `DEU` and `GBR`:

```
data2 = [[ 'Germany', 82.66, 3693.20, 80.99 ],
         ['United Kingdom', 66.06, 2637.87, 81.16]]

ind2 = pd.DataFrame(data2, columns=['country', 'pop', 'gdp', 'life'],
                    index=['DEU', 'GBR'])
print(ind2)
```

```
            country    pop      gdp   life
DEU         Germany  82.66  3693.20  80.99
GBR  United Kingdom  66.06  2637.87  81.16
```

- Thus, the **two `DataFrames` have the same columns, but different rows**. We can combine them using the `pd.concat()` function, where the first argument is a list of the data frames to be combined, followed by the `axis` to concatenate along (in our case, we are combining in the row dimension, so `axis=0`).

```
combined = pd.concat([ind1, ind2], axis=0)
print(combined)
```

```
            country      pop       gdp   life
CHN           China  1378.66  11199.15  76.25
IND           India  1324.17   2263.79  68.56
USA   United States   323.13  18624.47  78.69
DEU         Germany    82.66   3693.20  80.99
GBR  United Kingdom    66.06   2637.87  81.16
```

- Note that the order of the resulting data frame is based on the order of the data frames in the list.

- Consider the indicator data from 2015 for `CHN`, `IND`, and `USA`:

```
data2015 = [[ 'China', 1371.22, 11015.54,76.09],
            ['India', 1310.15, 2103.59, 68.30],
            ['United States', 320.74, 18219.30, 78.69]]

indicators2015 = pd.DataFrame(data2015,
                              columns=['country', 'pop', 'gdp', 'life'],
                              index=['CHN', 'IND', 'USA'])
print(indicators2015)
```

```
           country      pop       gdp   life
CHN          China  1371.22  11015.54  76.09
IND          India  1310.15   2103.59  68.30
USA  United States   320.74  18219.30  78.69
```

- Now consider the indicator data from 2017 for `CHN`, `IND`, and `USA`:

```
data2017 = [[ 'China', 1386.40, 12143.49, 76.41],
            ['India', 1338.66, 2652.55, 68.80],
            ['United States', 325.15, 19485.39, 78.54]]

indicators2017 = pd.DataFrame(data2017,
                              columns=['country', 'pop', 'gdp', 'life'],
                              index=['CHN', 'IND', 'USA'])
print(indicators2017)
```

```
           country      pop       gdp   life
CHN          China  1386.40  12143.49  76.41
IND          India  1338.66   2652.55  68.80
USA  United States   325.15  19485.39  78.54
```

- If we combine in the same way as the last example, the result is a valid data frame, but one where the row labels have *duplicates*. While such non-uniqueness is allowed by Pandas, we no longer have a tidy data set. We demonstrate below.

```
combined = pd.concat([indicators2015, indicators2017], axis=0)
print(combined)
```

```
           country      pop       gdp   life
CHN          China  1371.22  11015.54  76.09
IND          India  1310.15   2103.59  68.30
USA  United States   320.74  18219.30  78.69
CHN          China  1386.40  12143.49  76.41
IND          India  1338.66   2652.55  68.80
USA  United States   325.15  19485.39  78.54
```

There are now duplicate row labels, and thus, this data is not tidy!

- The `concat()` function supports a `keys=` named parameter that allows us to specify a new, additional level for the row labels. The value is a list where each element specifies an index level value, so `keys=[2015, 2017]` will use an outer index of 2015 for the first data frame's rows and 2017 for the second data frame's rows.

```
combined = pd.concat([indicators2015, indicators2017], axis=0,
                     keys=[2015, 2017])
print(combined)
```

```
                country      pop       gdp   life
2015 CHN          China  1371.22  11015.54  76.09
     IND          India  1310.15   2103.59  68.30
     USA  United States   320.74  18219.30  78.69
2017 CHN          China  1386.40  12143.49  76.41
     IND          India  1338.66   2652.55  68.80
     USA  United States   325.15  19485.39  78.54
```

- This gives us the tidy two-level index with `year` and `code` as the levels. If we want the outer level to have a symbolic name, we can construct an `Index` object and specify both the list of outer level values as well as the name of the outer level index.

```
combined = pd.concat([indicators2015, indicators2017], axis=0,
                     keys=pd.Index([2015, 2017], name='year'))
print(combined)
```

```
                country      pop       gdp   life
year
2015 CHN          China  1371.22  11015.54  76.09
     IND          India  1310.15   2103.59  68.30
     USA  United States   320.74  18219.30  78.69
2017 CHN          China  1386.40  12143.49  76.41
     IND          India  1338.66   2652.55  68.80
     USA  United States   325.15  19485.39  78.54
```

- Suppose we wanted to access all the data corresponding to 2017 from the `combined` `DataFrame`. We can accomplish this using `.loc[]` and Pandas slicing notation:

```
print(combined.loc[2017,:])
```

```
           country      pop       gdp   life
CHN          China  1386.40  12143.49  76.41
IND          India  1338.66   2652.55  68.80
USA  United States   325.15  19485.39  78.54
```

- To access a single value, say `pop` from `USA` in 2015, we can use a tuple within `.loc[]`:

```
print(combined.loc[(2015,'USA'),'pop'])
```

```
320.74
```

- Below we construct two data frames from the `topname` data: one for the years 2010 and 2011, and another from 2017 and 2018.

```
topnames1 = topnames[topnames['year'].isin([2010, 2011])]
print(topnames1)
print()
topnames2 = topnames[topnames['year'].isin([2017, 2018])]
print(topnames2)
```

```
     year     sex      name  count
260  2010  Female  Isabella  22913
261  2010    Male     Jacob  22127
262  2011  Female    Sophia  21842
263  2011    Male     Jacob  20371

     year     sex  name  count
274  2017  Female  Emma  19800
275  2017    Male  Liam  18798
276  2018  Female  Emma  18688
277  2018    Male  Liam  19837
```

- If we perform `concat()` we obtain the following:

```
table = pd.concat([topnames1, topnames2], axis=0)
print(table)
```

```
     year     sex      name  count
260  2010  Female  Isabella  22913
261  2010    Male     Jacob  22127
262  2011  Female    Sophia  21842
263  2011    Male     Jacob  20371
274  2017  Female      Emma  19800
275  2017    Male      Liam  18798
276  2018  Female      Emma  18688
277  2018    Male      Liam  19837
```

- Note that the row indices run from 260-263, and then from 274-277. This could potentially cause errors, especially if we are performing a `.loc[]` operation. We can reindex our data frame using the `reset_index()` method:

```
table.reset_index(inplace=True)
print(table)
```

By setting `inplace=True` we tell Python to update the calling `DataFrame`.

```
   index  year     sex      name  count
0    260  2010  Female  Isabella  22913
1    261  2010    Male     Jacob  22127
2    262  2011  Female    Sophia  21842
3    263  2011    Male     Jacob  20371
4    274  2017  Female      Emma  19800
5    275  2017    Male      Liam  18798
6    276  2018  Female      Emma  18688
7    277  2018    Male      Liam  19837
```

Combining Tables: Concatenating Along the Column Dimension

- Here, we assume that the two source data frames have the *same rows* and an entirely *different set of columns.* In tidy data, the stipulation of the same rows means that the values of the independent variables are the same, and here we assume that they are incorporated into a meaningful index, and the stipulation of different columns means the data frames have different dependent variables

- Below we define a subtable called `cols1` that includes the `country` and `pop` columns for `IND`, `CHN`, and `USA` from the `indicators DataFrame`. In addition, we define a `DataFrame cols2` that includes the columns `imports` and `exports` for the same countries (rows).

```
cols1 = indicators.loc[['IND', 'CHN', 'USA'], ['country', 'pop']]

imports_exportsDoL = {
  'imports': [2241.66, 392.23, 1601.76],
  'exports': [1504.57, 266.16, 2280.54]}

codes = pd.Index(['USA', 'IND', 'CHN'],
                 name='code')

cols2 = pd.DataFrame(imports_exportsDoL, index=codes)

print(cols1)
print()
print(cols2)
```

```
            country      pop
code
IND           India  1324.17
CHN           China  1378.66
USA   United States   323.13

      imports  exports
code
USA   2241.66  1504.57
IND    392.23   266.16
CHN   1601.76  2280.54
```

- Note that both `cols1` and `cols2` have an index defined by `code`. We can combine them along the column dimension by the argument `axis=1` as follows.

```
table = pd.concat([cols1, cols2], axis=1)
print(table)
```

```
            country      pop  imports  exports
code
IND           India  1324.17   392.23   266.16
CHN           China  1378.66  1601.76  2280.54
USA   United States   323.13  2241.66  1504.57
```

- Note that the order of the columns was dictated by the order of the first `DataFrame`.

Joining Data Frames

- Pandas provides more powerful tools when combining data frames in more complicated ways than just along the row or column dimensions.

- One variation, `join()`, is a `DataFrame` method and uses the row label index for matching rows between the source frames. The other variation, `merge()`, is a function of the Pandas package and can use any column (or index level) for matching rows between the source frames.

- The combined frame has columns from both the original source frames with values populated based on the matching rows. If the two frames have columns with the same column name, the join/merge will include both, with the column names modified to distinguish the source frame of the overlapping column.

- Consider the two data frames below.

```
imports_exportsDoL = {
  'imports': [457.46, 643.52, 2342.67],
  'exports': [418.86, 441.11, 1545.61]}

codes = pd.Index(['CAN', 'GBR', 'USA'], name='code')

join1 = pd.DataFrame(imports_exportsDoL, index=codes)

country_landDoL = {
  'country': ['Belgium', 'United Kingdom', 'United States', 'Vietnam'],
  'land': [30280.0, 241930.0, 9147420.0, 310070.0]}

codes = pd.Index(['BEL', 'GBR', 'USA', 'VNM'], name='code')

join2 = pd.DataFrame(country_landDoL, index=codes)

print(join1)
print()
print(join2)
```

```
      imports  exports
code
CAN    457.46   418.86
GBR    643.52   441.11
USA   2342.67  1545.61
```

join1 data frame

```
             country       land
code
BEL          Belgium    30280.0
GBR   United Kingdom   241930.0
USA    United States  9147420.0
VNM          Vietnam   310070.0
```

join2 data frame

- Suppose we want all rows of `join1` to be represented in the combined result that includes `country` and `land`, but if a match is not found in `join2`, we just fill with missing values for `country` and `land`. If `join1` is the first (or left-hand) frame written as we combine, this type of combination is called a ***left join***.

- We invoke the `join()` method on the first data frame and specify a second argument of the "right" data frame. The named parameter `how=` is passed a string "`left`" to specify a left join, as illustrated below.

```
table = join1.join(join2, how="left")

print(table)
```

```
      imports  exports         country       land
code
CAN    457.46   418.86             NaN        NaN
GBR    643.52   441.11  United Kingdom   241930.0
USA   2342.67  1545.61   United States  9147420.0
```

- Note the appropriate values for the matched rows, and missing values where the right-hand frame did not have a match to correspond to the left frame. In this situation, we think of the left frame as being dominant, and taking whatever values it can from the right frame, to enrich rows present in the left frame.

- If the desired analysis requires that the combined result only has rows where a row label index matches from both frames, this is called an ***inner join***. It is a form of intersection, where the intersection is defined by matching just the value of the common row label index and disregarding any/all other column values. This inner join simply changes the `how=` named parameter argument to "`inner`":

```
table = join1.join(join2, how="inner")

print(table)
```

```
      imports  exports         country       land
code
GBR    643.52   441.11  United Kingdom   241930.0
USA   2342.67  1545.61   United States  9147420.0
```

- The result includes just the two rows where `code` is in common between the two frames, namely `GBR` and `USA`. In this case, we think of the new table as sitting between the two old tables, and drawing from both sides, whenever the same code is present in both.

- If we were to join the two tables in a different order, making `join2` be the lefthand frame and `join1` be the right-hand frame, then, in a left join, the result would have all the rows from `join2` with the `join2` columns of `country` and `land` and would fill in with missing values for the columns in `join1` (`imports` and `exports`) where there was no corresponding row in `join1`. In this case, `join2` is the dominant table.

```
table = join2.join(join1, how="left")

print(table)
```

```
             country       land  imports  exports
code
BEL          Belgium    30280.0      NaN      NaN
GBR   United Kingdom   241930.0   643.52   441.11
USA    United States  9147420.0  2342.67  1545.61
VNM          Vietnam   310070.0      NaN      NaN
```

- Rows `BEL` and `VNM` are present in `join2` but not in `join1`, so these are the ones that are augmented with missing values for the `join1` columns.

Merging Data Frames

- Sometimes, we wish to combine two tables based on common values between two tables, but the values are in a regular column and not part of an index.

- Recall that the `topnames` data frame has, by year and by sex, the name and count of the top baby names. The original `indicators` data frame has, among other things, the population for each of the countries, including the population for the USA.

- Suppose we wanted to create a new data frame based primarily on the `topnames` data set, but where we augment each row with the US population for that year. That could be used in an analysis, to divide the count of applicants by the population for that year to be able to see what percentage of the population is represented. This could give a fairer comparison as we try to compare between years.

- Below, we load in the original `indicators` data frame, which contains economic indicator data for 207 countries from 1960-2018. Then we create a new data frame called `us_pop`, that contains the population (in millions) for the USA from 1960 to 2017 by extracting the data from `indicators`. Can you follow how the data is extracted?

```
table = indicators[(indicators['code'] == 'USA') &
                   (indicators['year'] >= 1960) &
                   (indicators['year'] <= 2017)].reset_index()

us_pop = table[['year', 'pop']]
print(us_pop.head())
```

```
   year     pop
0  1960  180.67
1  1961  183.69
2  1962  186.54
3  1963  189.24
4  1964  191.89
```

- Recall the `topnames` data frame:

```
print(topnames.head())
```

```
   year     sex  name  count
0  1880  Female  Mary   7065
1  1880    Male  John   9655
2  1881  Female  Mary   6919
3  1881    Male  John   8769
4  1882  Female  Mary   8148
```

- The `merge()` function has, as its first two arguments, the two `DataFrame` objects to be combined, where the first argument is considered the "left" and the second argument is considered the "right." The `on=` named argument specifies the name of a column expected to exist in both data frames and to be used for matching values. The `how=` named parameter allows us to specify the logical equivalent of an inner join, a left join, or a right join.

- Below, we illustrate a left join/merge. Since `topnames` is the "left," this will give us all the rows of `topnames` and will add the `pop` column from matching rows in `us_pop`. For those rows and years where `us_pop` does not have population data (i.e., those years before 1960), the new data frame `merge1` has `NaN` indicating missing data.

```
merge1 = pd.merge(topnames, us_pop, on='year', how='left')
print(merge1.head())
print()
print(merge1[(merge1['year'] >= 1960) & (merge1['year'] <= 2017)].head())
```

```
   year     sex  name  count  pop
0  1880  Female  Mary   7065  NaN
1  1880    Male  John   9655  NaN
2  1881  Female  Mary   6919  NaN
3  1881    Male  John   8769  NaN
4  1882  Female  Mary   8148  NaN

     year     sex     name  count     pop
160  1960  Female     Mary  51475  180.67
161  1960    Male    David  85929  180.67
162  1961  Female     Mary  47680  183.69
163  1961    Male  Michael  86917  183.69
164  1962  Female     Lisa  46078  186.54
```

`merge1` for the years 1880-1882—note the `NaN` values for `pop`.

`merge1` for the years 1960-1962—note the `pop` contains the merged information.

- Now we illustrate an "inner" join/merge. Here, the result will only have rows where the `year` has common values from both data frames. So, this result will effectively prune the `topnames` portion to the years after 1960.

```
merge2 = pd.merge(topnames, us_pop, on='year', how='inner')
print(merge2)
```

```
     year     sex     name  count     pop
0    1960  Female     Mary  51475  180.67
1    1960    Male    David  85929  180.67
2    1961  Female     Mary  47680  183.69
3    1961    Male  Michael  86917  183.69
4    1962  Female     Lisa  46078  186.54
..    ...     ...      ...    ...     ...
109  2014    Male     Noah  19305  318.56
110  2015  Female     Emma  20455  320.90
111  2015    Male     Noah  19635  320.90
112  2016  Female     Emma  19496  323.13
113  2016    Male     Noah  19117  323.13

[114 rows x 5 columns]
```

Missing Data Handling

- We end this section with a brief discussion of the issue with missing data. Missing data can occur when:
  - Individuals do not respond to all questions on a survey.
  - Countries fail to maintain or report all their data to the World Bank.
  - An organization keeping personnel records does not know where everyone lives.
  - Laws prevent healthcare providers from disclosing certain types of data.
  - Different users of social media select different privacy settings, resulting in some individuals having only some of their data publicly visible.
  - Many other situations analogous to these.

- In the real world, there is no true standard for how missing data is encoded. It may be coded as a string such as "`N/A`", "`Not Applicable`", "`NA`", etc. Thankfully, information on how missing data is coded is typically contained in the *metadata* associated with he data set, sometimes called a *codebook*.

- Once we know how missing data is denoted, we should replace that coding with a special type. In Pandas, this type is denoted `nan` and is displayed as `NaN`. Please refer back to our section on Numpy to review how `NaN` values are treated.

Chapter 9

Tabular Model: Transformations and Constraints

Tabular Model Constraints

- ***Constraints*** are limitations on the organizational structure, the relationships, and the values entailed in a data set.
- For the tabular model in general, recall that the structural constraints are articulated by the precepts of tidy data:

Tidy Data:

1. Each column represents exactly one variable of the data set (**TidyData1**).
2. Each row represents exactly one unique (relational) mapping that maps from a set of givens (the values of the independent variables) to the values of the dependent variables (**TidyData2**).
3. Exactly one table is used for each set of mappings involving the same independent variables (**TidyData3**).

Additional Specific Constraints

Examples Beyond Tidy Data:

- A column named `age` must consist of integer values.
- A column named `income` is a categorical variable and must be one of the strings, `"high"`, `"middle"`, or `"low"`.
- A column named `year` must not have any missing values.
- If an `income` column is `"low"`, a `salary` column must be a float with values less than or equal to 24000.0.

Additional Constraints

- The first three of these are ***intra-column value constraints***, and the last is an ***intercolumn constraint***, which, for this structure, says something about a relationship between two variables.
- When data is created by a *provider*, it is often created without adhering to the constraints of tidy data!

Verification?

- When you begin examining data, you must check to see if it conforms to these principles.
- Outside documentation describing the form and assumptions of the data is called a ***data dictionary*** and often conveys ***metadata***.
- ***Metadata*** is defined as the data providing information about one or more aspects of the data; it is used to summarize basic information about data which can make tracking and working with specific data easier.

Goal of Chapter 9

- **Just because data is organized in rows and columns does *not* mean that it conforms to the tabular model!**
- The goal of this chapter is to
  - better understand the three principles of tidy data.
  - recognize when data does not conform to the model, and to give some tools that can transform untidy data into its tidy data equivalent.

Tabular Transformations

- ***Transformations*** are the process of converting from one tabular form into another tabular form, while maintaining the information and values represented.

- In this chapter, our focus will be on using transformations to *normalize* a data set from an untidy form.

Transpose

- A transpose can be thought of as swapping the rows and columns of a data frame.

- This is typically done when data is provided incorrectly, with variables as the rows and individuals as the columns.

Transpose in Pandas

- In Pandas/numPy, the transpose is obtained using the `.T` attribute of the `DataFrame`/`ndarray`.
- In Pandas, the transpose operates so that the row labels become the column labels and vice versa.

```
example1
```

	I	1	2	3
A	A.I	A.1	A.2	A.3
B	B.I	B.1	B.2	B.3
C	C.I	C.1	C.2	C.3

```
result1 = example1.T
result1
```

	A	B	C
I	A.I	B.I	C.I
1	A.1	B.1	C.1
2	A.2	B.2	C.2
3	A.3	B.3	C.3

Melt

- The ***melt operation*** converts columns into rows, like the transpose, but in a very specialized way.
- The name is meant to evoke a candle lying across the top of the data frame, melting from column information into row information.
- We use this transformation when information stored in column names should actually be stored in the rows.

```
females
```

year	2013	2014	2015
name	Sophia	Emma	Emma
count	21223	20936	20455

Melt Operation

- The melt operation takes a subset of the columns, the *melt columns*, and uses the *column labels* of the melt columns as table entry *values* under a single new column, called the *new variable*.
- The operation uses the values from the original melt columns to become a single new *value column* in the transformation.
- This operation is particularly useful when the names of columns in a data set are, in fact, values in the data set, and we need to create a new column for these values.

Melt Example

- Consider the columns of `example1`. We wish to melt the columns labeled `1`, `2`, and `3` and we wish not to melt column `I`.

```
example1

|      I    1    2    3
| A  A.I  A.1  A.2  A.3
| B  B.I  B.1  B.2  B.3
| C  C.I  C.1  C.2  C.3
```

Melt Example Continued

- The result will have three columns:
  - The non-melt column `I`, which remains and will repeat its values in the rows of the melted result.
  - The new variable column, which we name `K`, whose values come from the column labels in the original
  - The new value column, which we name `V`, whose values come from the values in the melt columns of the original.

Melt Example Continued

- The rows of the result are obtained by expanding, for each row in the original, *n* rows in the result, where *n* is the number of melt columns.

- The *n* rows will repeat the `I` value, will use exactly one of the column labels for the `K` column, and will use a lookup of `I` and the particular melt column to get the value for `V`.

Melt Example Continued

```
example1
```

```
|      I    1    2    3
| A  A.I  A.1  A.2  A.3
| B  B.I  B.1  B.2  B.3
| C  C.I  C.1  C.2  C.3
```

```
meltresult1=example1.melt(id_vars=['I'],
                          value_vars=['1', '2','3'],
                          var_name='K', value_name='V')
meltresult1
```

```
|      I  K    V
| 0  A.I  1  A.1
| 1  B.I  1  B.1
| 2  C.I  1  C.1
| 3  A.I  2  A.2
| 4  B.I  2  B.2
| 5  C.I  2  C.2
| 6  A.I  3  A.3
| 7  B.I  3  B.3
| 8  C.I  3  C.3
```

Melt Parameters

- When we consider the general information needed by a melt operation to perform its function, we see that its arguments must
  - Allow the function to partition the columns into the melt columns and the nonmelt columns,
  - Provide a name for the *new variable column*, and
  - Provide a name for the *new value column*.

Melt in Pandas

- In the Pandas `melt()` method of a `DataFrame`,
  - the `id_vars=` named parameter allows us to specify the non-melt column or columns,
  - the `value_vars=` named parameter allows us to specify the melt columns,
  - the `var_name=` provides the name for the new variable column, and
  - the `value_name=` provides the name for the new value column.
- Since we are partitioning the columns between those to be melted and those not melted, we only need to specify one of `id_vars` and `value_vars`, and the operation can determine the other.

Melt Example 2

Sex	2015	2016	2017
Female	Emma	Emma	Emma
Male	Noah	Noah	Liam

$\Rightarrow$

Sex	Year	Name
Female	2015	Emma
Male	2015	Noah
Female	2016	Emma
Male	2016	Noah
Female	2017	Emma
Male	2017	Liam

- `Sex` would be defined as the non-melt column(s) and the columns `2014`, `2015`, and `2016` are the melt column.
- The melt operation introduces a new variable `year` composed from the original column names.
- Lastly, the entries in the original table are gather in the new column `name`.

Melt Example 2 Continued

```
names_by_year DataFrame
```

	sex	2015	2016	2017
0	Female	Emma	Emma	Emma
1	Male	Noah	Noah	Liam

```
melted_names = names_by_year.melt(id_vars=['sex'],
                                  var_name='year',value_name='name')
melted_names
```

	sex	year	name
0	Female	2015	Emma
1	Male	2015	Noah
2	Female	2016	Emma
3	Male	2016	Noah
4	Female	2017	Emma
5	Male	2017	Liam

Melt Example 3

- When there is more than one column that makes up the non-melt columns, all the values of the non-melt columns are repeated in each of the rows of the result generated from the same row or the source.

Melt Example 3 Continued

- We illustrate below, where we have two non-melt columns, `I` and `J`.

```
|      I    J    1    2    3
| A  A.I  A.J  A.1  A.2  A.3
| B  B.I  B.J  B.1  B.2  B.3
| C  C.I  C.J  C.1  C.2  C.3
```

← example2 DataFrame

```
meltresult2 = example2.melt(id_vars=['I', 'J'],
                            var_name='K', value_name='V')
meltresult2
```

```
|      I    J  K    V
| 0  A.I  A.J  1  A.1
| 1  B.I  B.J  1  B.1
| 2  C.I  C.J  1  C.1
| 3  A.I  A.J  2  A.2
| 4  B.I  B.J  2  B.2
| 5  C.I  C.J  2  C.2
| 6  A.I  A.J  3  A.3
| 7  B.I  B.J  3  B.3
| 8  C.I  C.J  3  C.3
```

Pivot

- The `pivot()` operation provides the dual operation to that of `melt()`.
- It is used to combine a *set of rows* into a *single row* by adding columns and moving the data from the multiple rows into the new columns of the transformed data set.

Pivot Continued

- For a pivot transformation, there must exist a column, called the "Pivot" column, where the set of unique values in that column *becomes the labels* for the newly created columns.
- The *values* to be populated in the newly created columns come from a single "Value" column from the original table, and the column where the value is placed in the new table is based on the value in the "Pivot" column.
- We most often use a pivot transformation when the values of the "Pivot" column are actually column names in our desired data presentation

Pivot Example 1

```
pivot_ex1

  |       I    K     V
  | A1  A.I  K.1  A1.V
  | A2  A.I  K.2  A2.V
  | B1  B.I  K.1  B1.V
  | B2  B.I  K.2  B2.V
  | C1  C.I  K.1  C1.V
  | C2  C.I  K.2  C2.V
```

- The upcoming pivot of this table has "Index" as column `I`, pivot column as `K`, and value column as `V`.
- There are six distinct values from column `V` that we want to represent in the final table, two per each of `A`, `B`, and `C`.
- For each set of rows to combine, the values in the pivot column (`K.1` and `K.2`) distinguish the rows from each other, and the same pivot values are repeated in each set.
- We can think of `I` and `K` as two independent variables that yield a dependent variable `V`.

Pivot Example 1 Continued

```
pivot_ex1
```

```
|        I    K      V
| A1   A.I   K.1   A1.V
| A2   A.I   K.2   A2.V
| B1   B.I   K.1   B1.V
| B2   B.I   K.2   B2.V
| C1   C.I   K.1   C1.V
| C2   C.I   K.2   C2.V
```

```
pivot_ex1.pivot(index='I', columns='K')
```

```
|          V
| K      K.1    K.2
| I
| A.I   A1.V   A2.V
| B.I   B1.V   B2.V
| C.I   C1.V   C2.V
```

- In the Pandas `pivot` method, the named parameter `index=` allows us to specify the "Index" column and this becomes the row label/index of the result.

- The `columns=` named parameter allows us to specify the "Pivot" column. In this example, only one other column (`V`) exists, and is used as the "Value" column for the operation.

Pivot Example 1 Continued

```
pivot_ex1
```

```
|        I    K      V
| A1   A.I   K.1   A1.V
| A2   A.I   K.2   A2.V
| B1   B.I   K.1   B1.V
| B2   B.I   K.2   B2.V
| C1   C.I   K.1   C1.V
| C2   C.I   K.2   C2.V
```

```
pivot_ex1.pivot(index='I', columns='K')
```

```
|          V
| K      K.1    K.2
| I
| A.I   A1.V   A2.V
| B.I   B1.V   B2.V
| C.I   C1.V   C2.V
```

- Note that the resulting column label/index has two levels, with the outside level labeled with the "Value" column label and the inside level giving the new column labels represented by the "Pivot" column.

Pivot Example 1 Continued

- If one wishes to avoid the situation of a two-level column index, then all three of `index`, `columns`, and `values` should be specified to the `pivot` method:

```
pivot_ex1
```

```
|       I    K     V
| A1  A.I  K.1  A1.V
| A2  A.I  K.2  A2.V
| B1  B.I  K.1  B1.V
| B2  B.I  K.2  B2.V
| C1  C.I  K.1  C1.V
| C2  C.I  K.2  C2.V
```

```
pivot_ex1.pivot(index='I', columns='K', values='V')
```

```
| K     K.1   K.2
| I
| A.I  A1.V  A2.V
| B.I  B1.V  B2.V
| C.I  C1.V  C2.V
```

Pivot Example 2

- Extending the abstract example, we can see what happens if there is more than one "Value" column with the same "Index" and "Pivot" columns.

```
pivot_ex2
```

```
|       I    K     V1     V2
| A1  A.I  K.1  A1.V1  A1.V2
| A2  A.I  K.2  A2.V1  A2.V2
| B1  B.I  K.1  B1.V1  B1.V2
| B2  B.I  K.2  B2.V1  B2.V2
| C1  C.I  K.1  C1.V1  C1.V2
| C2  C.I  K.2  C2.V1  C2.V2
```

Pivot Example 2 Continued

```
pivot_ex2
```

	I	K	V1	V2
A1	A.I	K.1	A1.V1	A1.V2
A2	A.I	K.2	A2.V1	A2.V2
B1	B.I	K.1	B1.V1	B1.V2
B2	B.I	K.2	B2.V1	B2.V2
C1	C.I	K.1	C1.V1	C1.V2
C2	C.I	K.2	C2.V1	C2.V2

- Here, the "Index" column is still `I`, and the pivot column is still `K`, but now there are two "Value" columns, `V1` and `V2`, each with unique values to be represented in a pivoted result.
- In this case, beyond the row label index, we expect four columns, with one for each pairing of the pivot column values `K.1` and `K.2` with the two value columns `V1` and `V2`.

Pivot Example 2 Continued

```
pivot_ex2
```

	I	K	V1	V2
A1	A.I	K.1	A1.V1	A1.V2
A2	A.I	K.2	A2.V1	A2.V2
B1	B.I	K.1	B1.V1	B1.V2
B2	B.I	K.2	B2.V1	B2.V2
C1	C.I	K.1	C1.V1	C1.V2
C2	C.I	K.2	C2.V1	C2.V2

```
pivot_ex2.pivot(index='I', columns='K', values=['V1', 'V2'])
```

	V1		V2	
K	K.1	K.2	K.1	K.2
I				
A.I	A1.V1	A2.V1	A1.V2	A2.V2
B.I	B1.V1	B2.V1	B1.V2	B2.V2
C.I	C1.V1	C2.V1	C1.V2	C2.V2

- The two-level index of the result of a pivot is helpful here, as it allows a naming for the combinations of value column and pivot column in the result.

Pivot Example 3

- Given `melted_names` below, suppose we wanted a transformation with the values of `year` as the columns, and rows for the two values of `sex` (since pivot is dual to melt).

```
melted_names
```

	sex	year	name
0	Female	2015	Emma
1	Male	2015	Noah
2	Female	2016	Emma
3	Male	2016	Noah
4	Female	2017	Emma
5	Male	2017	Liam

```
melted_names.pivot(index='sex', columns='year', values='name')
```

year	2015	2016	2017
sex			
Female	Emma	Emma	Emma
Male	Noah	Noah	Liam

Pivot Example 4

- Given `melted_names` below, now suppose we wanted a transformation with the two values of `sex` as the columns, and a row for each year, we just need to reverse the "Index" and "Pivot" columns:

```
melted_names
```

	sex	year	name
0	Female	2015	Emma
1	Male	2015	Noah
2	Female	2016	Emma
3	Male	2016	Noah
4	Female	2017	Emma
5	Male	2017	Liam

```
melted_names.pivot(index='year', columns='sex', values='name')
```

sex	Female	Male
year		
2015	Emma	Noah
2016	Emma	Noah
2017	Emma	Liam

Pivot Table

- There are times when a `pivot()` operation is unable to handle some more general situations.
- Consider the following table, evaluating it for performing a pivot operation, with `I` as the "Index," `P` as the "Pivot," and `V1` and `V2` as two "Value" columns:

```
table
```

	I	P	V1	V2
0	A	w	15	23.5
1	A	x	10	42.5
2	A	x	5	18.0
3	B	w	8	10.2
4	B	w	4	14.3
5	B	x	6	12.5

- We want the result after a pivot to have two rows, one for `A` and the other for `B`.
- But we have subsets of rows of the same index with the same value of the pivot variable.
  - For the `A` rows, there are two rows with a `P` value of `x`.

Pivot Table Continued

- A `pivot_table()` goes beyond the `pivot()` and combines rows with duplicate (`I`,`P`) pairs into a single destination row using *aggregation*.
- When we invoke `pivot_table()`, we must specify an aggregation function to use to combine the values with duplicate (`I`,`P`) pairs.
- The default aggregation function is the mean, so that we would combine the `V1=8` and `V1=4` that appear in (`B`,`w`) rows into an output value of `V1=6`.

Pivot Table Example

```
table
```

	I	P	V1	V2
0	A	w	15	23.5
1	A	x	10	42.5
2	A	x	5	18.0
3	B	w	8	10.2
4	B	w	4	14.3
5	B	x	6	12.5

```
pivoted = table.pivot_table(index='I', columns='P')
pivoted
```

	V1		V2	
P	w	x	w	x
I				
A	15.0	7.5	23.50	30.25
B	6.0	6.0	12.25	12.50

- We can see that the value for `A` and `x` under `V1` is the mean of 10 and 5, and the value for `B` and `w` under `V2` is the mean of 10.2 and 14.3.

Pivot Table Example 2

- Consider the following table, and assume we wish to pivot on the `indicator` column, making `pop` and `gdp` into column labels, with values based on the values in column `value`.

```
ind2
```

	code	year	indicator	value
0	IND	2005	pop	1147.61
1	IND	2010	pop	1234.28
2	IND	2015	pop	1310.15
3	USA	2005	pop	295.52
4	USA	2010	pop	309.33
5	USA	2015	pop	320.74
6	IND	2005	gdp	820.38
7	IND	2010	gdp	1675.62
8	IND	2015	gdp	2103.59
9	USA	2005	gdp	13036.64
10	USA	2010	gdp	14992.05
11	USA	2015	gdp	18219.30

- We want the values of the independent variables (`code` and `year`) to uniquely determine each row.
- This means we have two columns that, together, define the row-label index of the result.

Pivot Table Example 2 Continued

```
pivoted = ind2.pivot_table(index=['code', 'year'],
                           columns='indicator')
print(pivoted)
```

```
|                 value
| indicator         gdp      pop
| code year
| IND  2005     820.38  1147.61
|      2010    1675.62  1234.28
|      2015    2103.59  1310.15
| USA  2005   13036.64   295.52
|      2010   14992.05   309.33
|      2015   18219.30   320.74
```

- The result builds a multi-level Index in both the row and the column dimensions.

Chapter 10

Relational Model: Structure and Architecture

Background

Def: A ***database*** is an organized collection of persistent data, whose organization allows for rapid search and retrieval.

- Often, a database system is hosted by a data system provider and provides access to data system clients with appropriate permissions.
- Occasionally, a database may be hosted locally.

Persistence

- *Persistence* means that the data repository maintains its information and integrity across clients coming and going and even across stopping and starting of provider system software and hardware.
  - A repository can maintain its integrity across unexpected failures, like power outages of errors in client applications.

Sources of Error

- Allowing data to violate structural constraints (violations of tidy data).
- Allowing data in a column to not have an appropriate data type or be restricted to a set of acceptable values.
- **Allowing some data to be repeated** (i.e., redundant) in a way that allows changes to one copy not be reflected consistently across all copies.

Relational Database

- Relational databases are the most common form of database.
- A relational database is a type of database that stores and provides access to data points that are related to one another.
- Relational databases are based on the *relational model*, an intuitive, straightforward way of representing data in tables.
- In a relational database, each row in the table is a *record* with a unique ID called the *key*. The columns of the table hold *attributes* of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points.

Relational Database Continued

- On the provider side, many commercial (Oracle) and open source (PostgreSQL, MySQL, SQLite) database realizations exist.
- Database *modeling* and *design* also supports the requirement of eliminating redundant information. In the design process, the tables are determined through a process called *normalization* such that each one models separate information.
- The net result is a larger number of tables but designed so that redundancy is eliminated.
- Through the design and normalization process, the relationships that relate one table to another are also explicitly established.

Database Management System

Def: A ***database management system*** (DBMS) is software for managing databases and providing access to them.

- A DBMS responds to *imperatives* ("statements") given by *application programs*, custom-written or general-purpose, executing on behalf of *users*.
- Imperatives are written in the language of the DBMS (e.g., SQL).
- Responses include completion codes, messages, and results of queries.

What does a DBMS Do?

- In response to requests given by application programs:
  - Creates and destroys variables
  - Takes note of integrity rules (constraints)
  - Takes note of authorizations (who is allowed do what, to what)
  - Updates variables (honoring constraints and authorizations)
  - Provides results of queries
  - and more…

Misuse Causes Confusion

- People often use the term “database” to refer to the DBMS. This is incorrect and a source of much confusion.
- The database is the *container* created and manipulated via the DBMS.

Operational Database

- An ***Operational Database*** is one used in the operation of an enterprise, and is dynamic, being used for maintaining information about contacts, vendors, inventory, and so forth.
-
  - The operations are a mix of queries and are undergoing constant update, with additions of records, deletions of records, and changes to existing records.
  - These databases are used in the minute-by-minute processes and are sometimes called an *online transactional processing* (OLTP) system.

Analytical Database

- An ***Analytical Database*** is used to support analysis with regards to decision support.
- Some analytical databases might be open and accessible for analysis by outside users and be used for data science and machine learning on databases maintained for the public good.
- These analytical databases are often used to track historical and time dependent data. They are relatively static, populated initially with a collection of data and then, possibly added to as additional data becomes available.
- So, the operations are primarily queries and the addition of new data, but there are rarely deletions or changes to existing records.

Structure

- In the relational database model, the structure is a set of inter-dependent tables, where each table is comprised of rows and columns.
- An individual table is also, in its mathematical context, called a *relation*. This is the origin of the model as a *relational* database.
  - A common misconception is that the name has to do with the relationships that might exist between tables.

Structure Continued

- A relational database is the collection of tables that are designed together in a single *schema*.
  - In this context, schema refers to information about the database and table layout and properties.
- The database includes information governing the relationships between individual tables, as well as information about the users/clients permitted to access the database, and what types of access are allowed.
- A single provider system may host many relational databases, each with different schemas and with different user/client permissions.

Tables

- Tables are the main structures in a database.
- Each table represents a single, specific subject. The logical order of rows and columns within a table has no importance.
- Every table contains at least one column—known as the *primary key*—that uniquely identifies each of its rows.
- The subject of a table can be either an *object* or an *event*. When the subject is an object, the table represents something that is tangible, like a person, place, or thing.

Example

Customers

CustomerID	FirstName	LastName	StreetAddress	City	State	ZipCode
1010	Angel	Kennedy	667 Red River Road	Austin	TX	78710
1011	Alaina	Hallmark	Route 2, Box 203B	Woodinville	WA	98072
1012	Liz	Keyser	13920 S.E. 40th Street	Bellevue	WA	98006
1013	Rachel	Patterson	2114 Longview Lane	San Diego	CA	92199
1014	Sam	Abolrous	611 Alpine Drive	Palm Springs	CA	92263
1015	Darren	Gehring	2601 Seaview Lane	Chico	CA	95926

ROWS

COLUMNS

- The `Customers` table above is one of the most common examples of a table representing an object.
- `CustomerID` is the primary key of the `Customers` table.

Example

Patient Visit

PatientID	VisitDate	VisitTime	Physician	BloodPressure	Temperature
92001	2006-05-01	10:30	Ehrlich	120 / 80	98.8
96105	2006-05-02	11:00	Hallmark	160 / 90	99.1
96203	2006-05-02	14:00	Hallmark	110 / 75	99.3
97002	2006-05-01	13:00	Hallmark	112 / 74	97.5
98003	2006-05-02	9:30	Fournier	120 / 82	98.6
99014	2006-05-02	9:30	Fournier	120 / 80	98.8

- When the subject of a table is an event, the table represents something that occurs at a given point in time and has characteristics you wish to record.
- Above is a table representing an event—each row represents a doctor's appointment.

Columns

- A column is the smallest structure in the database, and it represents a characteristic of the subject of the table to which is belongs.
- In the relational model, columns of a table are also called ***fields***. We often use the term *field* synonymously with *field name*.
- The name of the column should identify its contents, such as `FirstName`, `LastName`, and `City`.
  - Unlike Pandas, no whitespace nor special characters are allowed to be part of field (column) names.

Breaking Up Data

- It is extremely important to break data into multiple columns correctly.
- For example, city, state, and ZIP code should always be separate columns.
- When you break these out, it becomes possible to sort or filter data by specific columns.

Rows

- A row represents a unique instance of the subject of the table.
- It is composed of the entire set of columns in a table, regardless of whether or not the columns contain any values.
- Rows are often called ***records***.
- Each row is identified throughout the database by a unique value in the primary key column(s) of that row.

Example

Customers

CustomerID	FirstName	LastName	StreetAddress	City	State	ZipCode
1010	Angel	Kennedy	667 Red River Road	Austin	TX	78710
1011	Alaina	Hallmark	Route 2, Box 203B	Woodinville	WA	98072
1012	Liz	Keyser	13920 S.E. 40th Street	Bellevue	WA	98006
1013	Rachel	Patterson	2114 Longview Lane	San Diego	CA	92199
1014	Sam	Abolrous	611 Alpine Drive	Palm Springs	CA	92263
1015	Darren	Gehring	2601 Seaview Lane	Chico	CA	95926

ROWS

COLUMNS

- In the `Customers` table, each row represents a unique customer within the table, and the `CustomerID` column identifies a given customer throughout the database.

Functional Dependencies

- Consider the example of a table maintaining records for each of the courses offered at a college.
- The fields might include a `courseid`, used internally by the registrar to uniquely identify the course, and also might have `coursesubject` like ECON or MATH, and a `coursenum`, like 201, or 350. Students would look in the catalog for ECON-350 for the description of the course.
- Beyond these, we assume fields/variables of `coursetitle`, the full title of the course, and `coursehours`, an integer for the number of credit hours of the course.

Functional Dependency Continued

- In this example we can write the functional dependency:

`courseid` ⇒ `coursesubject, coursenum, coursetitle, coursehours`

- If we can argue that the combination of `coursesubject` and `coursenum` is unique, we can, with equal validity, write:

`coursesubject, coursenum` ⇒ `courseid, coursetitle, coursehours`

Functional Dependency Continued

- If `studentid` is unique, it can serve as the independent variable side of the functional dependency of the `students` table:

`studentid` ⇒ `studentlast, studentfirst,` *other dependent vars*

- It may be tempting to try to identify students by their first and last name, as shown in the following potential dependency:

`studentlast, studentfirst` ⇒ `studentid,` *other dependent vars*

- However, if the possibility exists for the combination of last name and first name to *not* be unique, then this would be *invalid*, because it is not a functional dependency.

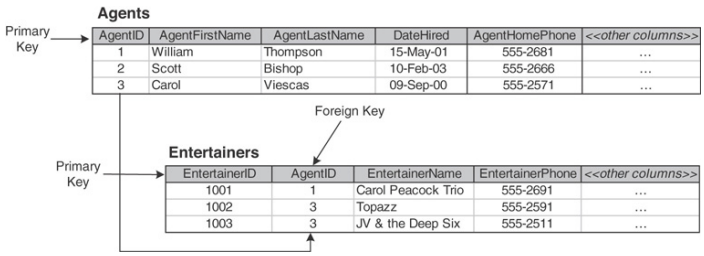
Keys

- A ***key*** is one or more fields (columns) that can potentially be used to *uniquely* identify an individual record (row).
- In the relational model, given a value for a key, there is only one record that corresponds to the given key.
- If a key is composed of more than one field, the key is called a ***composite key***.

Primary and Foreign Keys

- The design of a relational database must identify, for every table, a ***primary key***.
  - A primary key uniquely identifies each row within a table.
- A ***foreign key*** is a column (or group of columns) in a relational database that provide a link between data in two tables.
  - The term foreign key is derived from the fact that the second table already has a primary key.

Primary and Foreign Keys



AgentID	AgentFirstName	AgentLastName	DateHired	AgentHomePhone	<<other columns>>
1	William	Thompson	15-May-01	555-2681	…
2	Scott	Bishop	10-Feb-03	555-2666	…
3	Carol	Viescas	09-Sep-00	555-2571	…

EntertainerID	AgentID	EntertainerName	EntertainerPhone	<<other columns>>
1001	1	Carol Peacock Trio	555-2691	…
1002	3	Topazz	555-2591	…
1003	3	JV & the Deep Six	555-2511	…

- `AgentID` is the primary key of the `Agents` table, and it is the foreign key in the `Entertainer` table.
- The `Entertainer` table already has a primary key—`EntertainerID`.
- In this relationship, `AgentID` is the column that establishes the connection between the `Agents` and `Entertainers` tables.

Views

- A ***view*** is a virtual table composed of columns from one or more tables in the database.
- The tables that comprise the view are known as ***base tables***.
- The relational model refers to a view as *virtual* because it draws data from base tables rather than storing any data on its own.
- Views enable you to see information in a database from may perspectives.

A Sample View

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	*<<other columns>>*
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…
<< more rows here >>				

Engagements

EngagementNumber	CustomerID	StartDate	EndDate	StartTime	*<<other columns>>*
3	10001	2016-09-10	2016-09-15	13:00	…
13	10003	2016-09-17	2016-09-20	20:00	…
14	10001	2016-09-24	2016-09-29	16:00	…
17	10002	2016-09-29	2016-10-02	18:00	…
<< more rows here >>					

Customer_Engagements *(view)*

EngagementNumber	CustFirstName	CustLastName	StartDate	EndDate
3	Doris	Hartwig	2016-09-10	2016-09-15
13	Peter	Brehm	2016-09-17	2016-09-20
14	Doris	Hartwig	2016-09-24	2016-09-29
17	Deb	Waldal	2016-09-29	2016-10-02
<< more rows here >>				

Relationships

- If rows of one table are associated with rows of another table, the tables are said to have a *relationship*.
- Three types of relationships can exist between tables: *one-to-one*, *one-to-many*, and *many-to-many*.
- Understanding relationships is crucial to how multi-table queries are designed and work.

One-to-One Relationship

- A pair of tables have a ***one-to-one relationship*** when a single row in the first table is related to *only one* row in the second table, and a single row in the second table is related to *only one* row in the first table.
- In this relationship, one table is referred to as the ***primary table***, and other is referred to as the ***secondary table***.
  - The primary key of the primary table is inserted into the secondary table, where it becomes a foreign key.
- Thus, the foreign key also acts as the primary key of the secondary table.

One-to-One Example

Agents

AgentID	AgentFirstName	AgentLastName	DateOfHire	AgentHomePhone	*<<other columns>>*
1	William	Thompson	1997-05-15	555-2681	…
2	Scott	Bishop	1998-02-05	555-2666	…
3	Carol	Viescas	1997-11-19	555-2571	…

Compensation

AgentID	Salary	CommissionRate	*<<other columns>>*
1	$35,000.00	4.00%	…
2	$27,000.00	4.00%	…
3	$30,000.00	5.00%	…

One-to-Many Relationship

- In a ***one-to-many relationship***, a single row in the first table can be related to *many* rows in the second table, but a single row in the second table can be related to only *one* row in the first.
- This relationship is established by taking the primary key of the table on the “one” side and inserting it into the table on the “many” side, where it becomes a foreign key.

One-to-Many Example

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	*<<other columns>>*
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

Engagements

EngagementID	EntertainerID	CustomerID	StartDate	EndDate	*<<other columns>>*
5	1003	10006	2007-09-11	2007-09-14	…
7	1002	10004	2007-09-11	2007-09-18	…
10	1003	10005	2007-09-17	2007-09-26	…
12	1001	10014	2007-09-18	2007-09-26	…

Many-to-Many Relationship

- A pair of tables is in a ***many-to-many relationship*** when a single row in the first table can be related to *many* rows in the second table, and a single row in the second table can be related to *many* rows in the first table.
- To establish this relationship properly, you must create what is known as a ***linking table***.
- A linking table provides an easy way to associate rows from one table with those of the other and helps to ensure that you have no problems adding, deleting, or modifying any related data.

Linking Table

- A linking table is defined by taking a copy of the primary key of each table in the relationship and using them to form the structure of the new table.
- These columns serve two roles: together they form the composite primary key of the linking table, and separately they each serve as a foreign key.
- A many-to-many relationship that has not been properly established is said to be ***unresolved***.

Unresolved Example

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	*<<other columns>>*
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	*<<other columns>>*
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

- A single row in the `Customers` table can be related to many rows in the `Entertainers` tables, and vice-versa.
- How do you associate a single customer with several entertainers or a specific entertainer with several customers?

Linking Table Example

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	<<other columns>>
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…

Engagements (linking table)

EngagementID	CustomerID	EntertainerID	StartDate	<<other columns>>
43	10001	1001	2007-10-21	…
58	10001	1002	2007-12-01	…
62	10003	1005	2007-12-09	…
71	10002	1003	2007-12-22	…
125	10001	1003	2008-02-23	…

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	<<other columns>>
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

Design Phase of a Table

- Let's consider the design of a `courses` table.

Table 10.1 Fields of the `courses` table

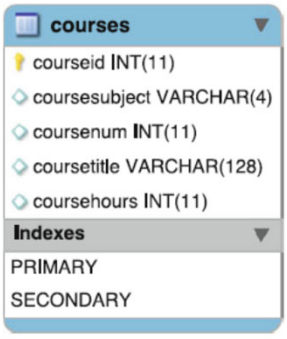
Field	Data type and size	Field description
`courseid`	Integer (4 bytes)	Unique number associated with a particular course
`coursesubject`	String of up to 4 chars	The identifier for the subject, be it department or program
`coursenum`	Integer (4 bytes)	The number of the course, as used in the catalog
`coursetitle`	String up to 128 chars	Full title of the course
`coursehours`	Integer (4 bytes)	Credit hours for course, from 1 to 4

- The dependencies are:

`courseid` → `coursesubject`, `coursenum`, `coursetitle`, `coursehours`

`coursesubject`, `coursenum` → `courseid`, `coursetitle`, `coursehours`

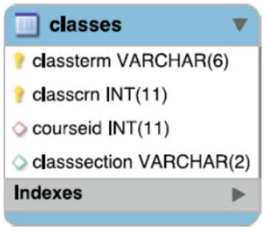
Table Schema Tool



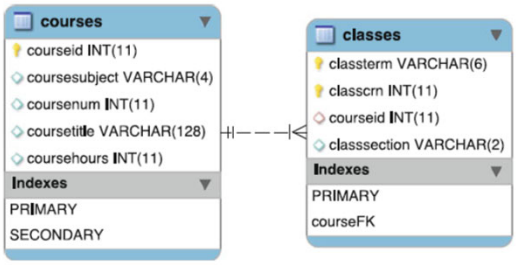
- A table is depicted in a box.
- The primary key is indicated with a key icon.
- The icon next to the other fields is either filled in, or not, indicated if a field is required to be non-NULL or not.
- The `Indexes` section refers to the auxiliary structures that allow access when given a key.
- A `PRIMARY` index will always exist. In this case, we are also showing a `SECONDARY` index that, while not shown in this picture, is the composite key of `coursesubject` and `coursenum`.

Multiple Tables Design

- The rows (records) of the `courses` table represent exactly one course, like an item in the list of courses of a catalog for a college.
- We could extend our database by adding a `classes` table, which would include a record (row) for each class instance taught.

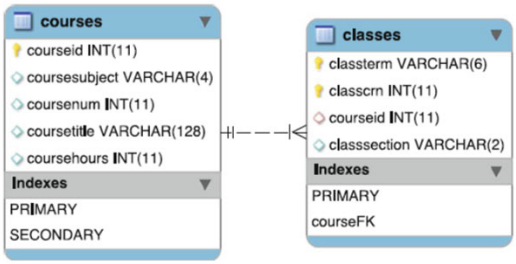


Schema for `courses` and `classes` Tables: One-to-Many Relationship



- The line denotes the relationship between the `classes` and `courses` tables.
- The "crows foot" on the `classes` side indicates that this is a *many* side of a table relationship.
- On the `courses` side, the relationship line is a hash, which indicates this is a *one* side of a table relationship.

Schema for `courses` and `classes` Tables: One-to-Many Relationship

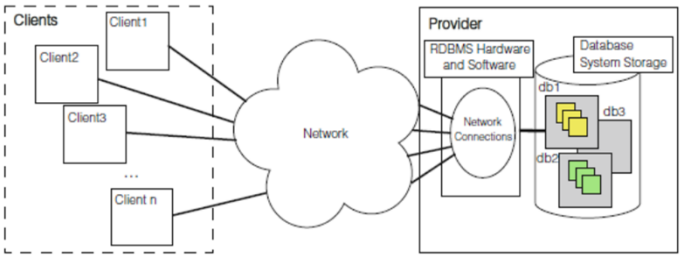


- Note the `courseFK` index for the `classes` table, which indicates that the `courseid` is a foreign key (for the `classes` table).
- Thus, we can use the `courseid` field in the `classes` table to reference the correct corresponding `courseid` in the `courses` table.

Discussion

- Note that a foreign key allows for efficient table combinations (such as a join operation).
- Relationships need to be part of the design of the database, before being populated with data.
- This can help to reduce common errors
  - If an "insert" to the `classes` table included a `courseid` that did not exist in the `courses` table, it would be rejected with an error. Such inter-table constraints are called ***referential integrity constraints***.

Database Architecture



- The sum total of the database, the DBMS, and the associated applications can be referred to as a "database system."
- Often the term "database" is also used to loosely refer to any of the DBMS, the database system or an application associated with the database

SQL on Your Personal Computer

- While learning the basics of SQL, we will make use of two tools on your personal computer: SQLiteStudio and IPython SQL Magic.

SQLite:

- SQLite is a small, fast, self-contained, high-reliability, full-featured, SQL database engine. It is the most used database engine in the world—it is built into all mobile phones and most computers. It comes bundled inside countless other applications that people use every day.

- Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database is contained in a single file.

- SQLite is not directly comparable to client/server SQL database engines such as MySQL, Oracle, or PostgreSQL. You can read more about SQLite here.

SQLiteStudio:

- We will make use of SQLiteStudio in this course, which is a desktop application for managing SQLite databases. It is free, cross-platform, and provides some of the most important features to work with SQLite databases, including importing and exporting data in various formats.

- SQLiteStudio is not the same as SQLite. SQLite is a database, while SQLiteStudio is an application to manage such databases.

- You can download SQLiteStudio by clicking on the **Download** button of the SQLiteStudio homepage:

  https://sqlitestudio.pl/

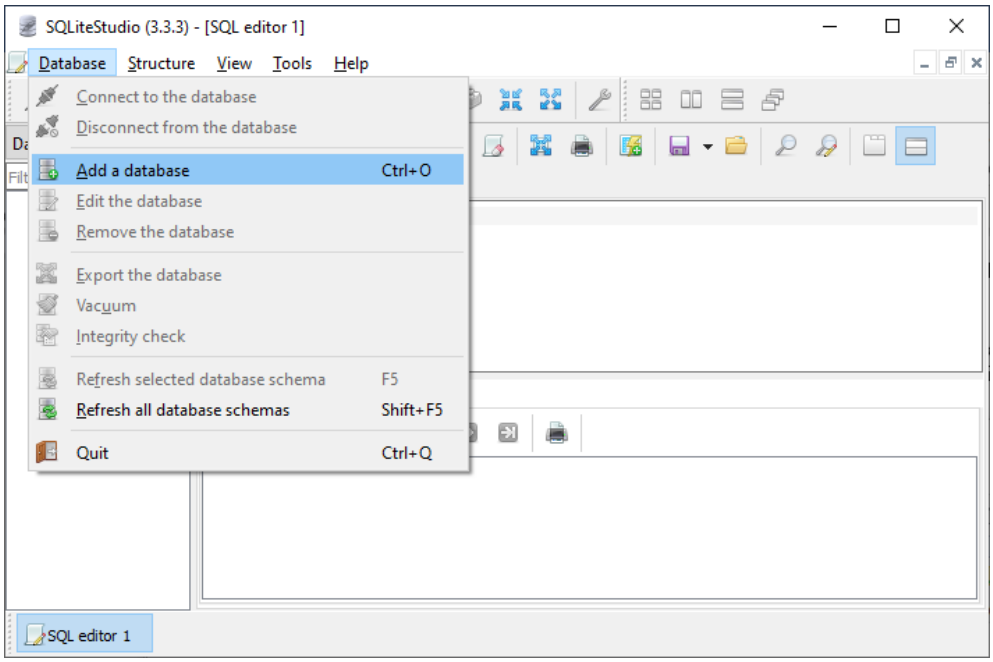
- **On MacOS**, the downloaded file should be **sqlitestudio-3.3.3.dmg**. This is a *container* for the app SQLiteStudio. There is no installation necessary—you just need to double-click the DMG file to open and mount it to your Mac. Note that if you get a "Cannot be opened because the developer cannot be verified" error, then please try the solutions listed on this web page:

  Lifewire Fix for Developer Cannot Be Verified

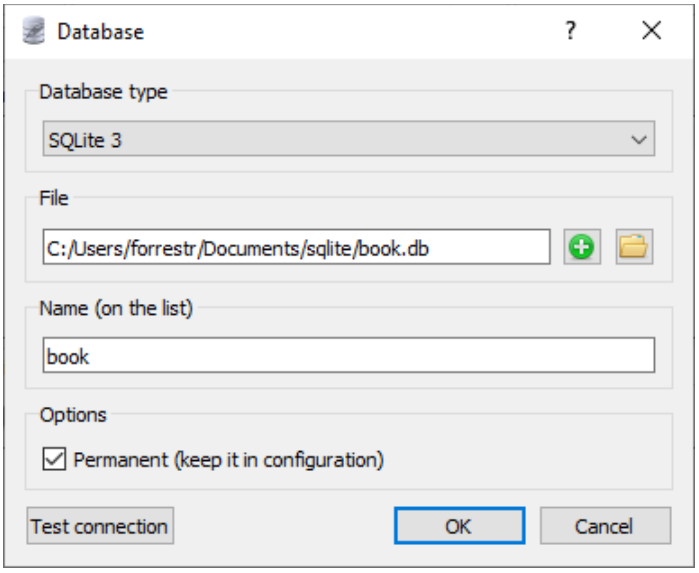
- **On Windows**, the downloaded file should be **sqlitestudio-3.3.3.zip**. Note that there is no installer for this application. Simply unzip the file to an appropriate folder and then double click on the **SQLiteStudio.exe** file to run the application.

- To get started, download the `book.db` SQLite database file from Teams and save it to an appropriate directory.

- Locate and run the SQLiteStudio application. From the **Database** menu, select **Add a database**.

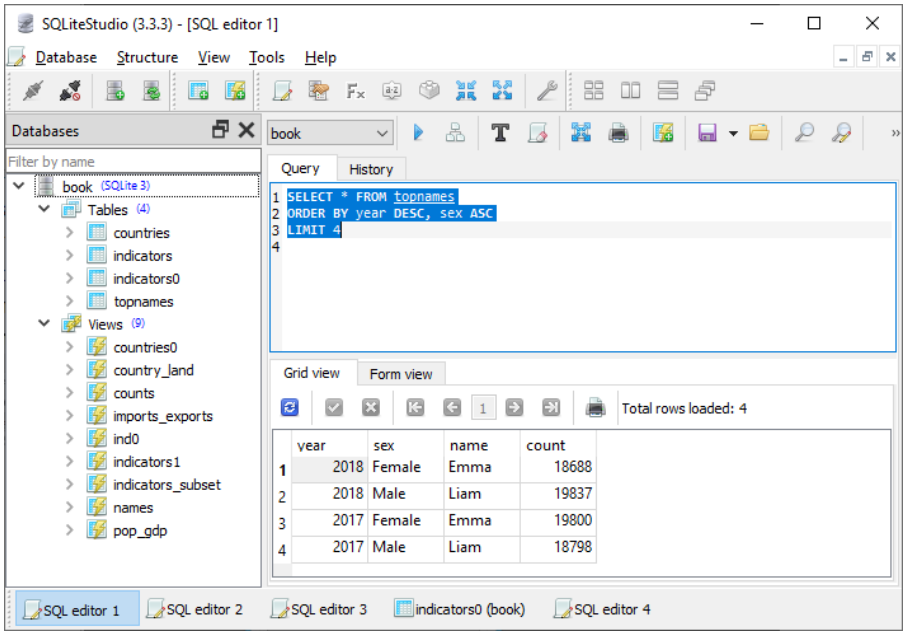


- This will open a dialog box. Click on the folder icon and navigate to the location of the `book.db` database file and select it. Then click **OK**.

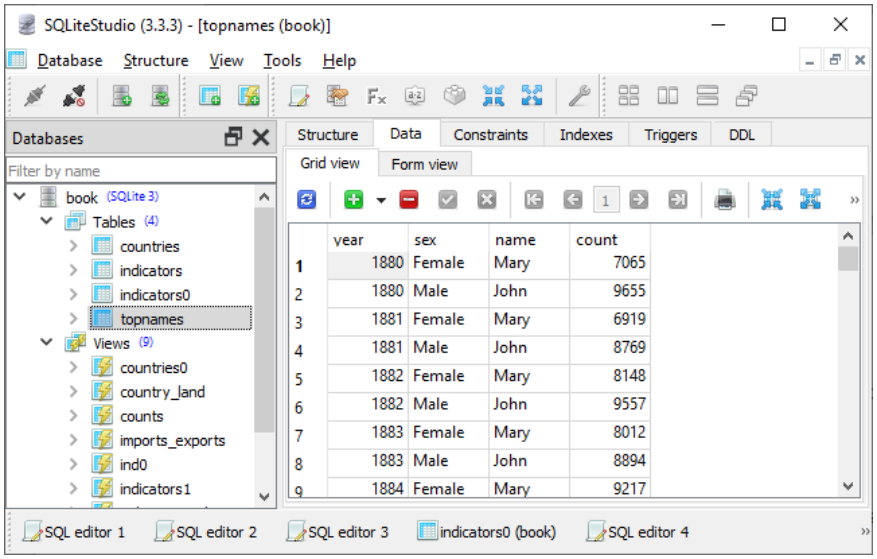


- If the SQL editor is not visible, then do the following: from the **Tools** menu, select **Open SQL editor**. From the editor, you can enter SQL commands, and when you are ready you can click the run button (standard "play" icon).

- For example, here is the output from the SQL code on page 324 of out textbook:



- To view the data contained in a particular table, click on the table name, such as **topnames** from the **book.db** database, and then click the **Data** tab:



- Below are a couple of nice tutorials on how to use SQLiteStudio:

  https://executecommands.com/how-touse-sqlite-studio-manage-sqlite-database/

  http://cswire.blogspot.com/2017/11/sqlitestudio-tutorial-basic-database.html

IPython SQL Magic:

- The IPython SQL magic extension makes it possible to write SQL queries directly into code cells of Jupyter notebooks, as well as read the results straight into a Pandas data frame.

- *Magic commands* are a set of convenient functions in Jupyter notebooks that are designed to solve some of the common problems in standard data analysis.

- To use Python magic with SQL, we need to install the `ipython-sql` library first. You can install it by running this code from **Anaconda Command Prompt** on Windows or **Terminal** on macOS:

```
conda install -c conda-forge ipython-sql
```

- Jupyter notebooks use "`%`" sign before the command to indicate that the function will be a "magic function." This "magic" is a group of pre-defined functions contained in the IDE's kernel that allows us to execute provided commands.

Loading the External SQL Module

- To load the SQL module using Python magic, run the following command in a code cell of a Jupyter notebook:

```
%load_ext sql
```

Connecting to an Existing Database

- To open an existing SQLite database, such as the `book.db` database, we could use the absolute path to the database like this (note that there are three backward slashes):

```
%sql sqlite:///C:\Users\forrestr\Documents\sqlite\book.db
```

- Alternately, we could give the relative path. For example, if the `book.db` file is in the same directory as the Jupyter notebook we could load it as follows:

```
%sql sqlite:///book.db
```

- Note that if SQL magic cannot find the database, then it will silently create an empty database (without throwing an error).

SQL Cell Magic

- *Cell magic* applies to the whole cell. It is utilized by double percentage signs (%). That is, all you have to type is `%%sql` and the rest of the cell will be treated as an SQL script.

- We demonstrate in a Jupyter notebook below with some code from page 322 of the textbook:

```
In [30]: %load_ext sql

In [35]: %sql sqlite:///book.db

In [36]: %%sql
         SELECT * FROM topnames
         ORDER BY year DESC, sex ASC
         LIMIT 4

          * sqlite:///book.db
         Done.
```

Out[36]:

year	sex	name	count
2018	Female	Emma	18688
2018	Male	Liam	19837
2017	Female	Emma	19800
2017	Male	Liam	18798

SQL Line Magic

- *Line magic* will turn only one line of your code cell to an SQL script. All you have to do is type `%sql` before your SQL query and the rest of the line will be interpreted as an SQL script. We demonstrate in a Jupyter notebook below:

```
In [38]: %sql SELECT name FROM topnames LIMIT 3

          * sqlite:///book.db
         Done.
```

Out[38]:

name
Mary
John
Mary

Assigning Results of an SQL Query to a Pandas Data Frame

- We can assign the results of an SQL query to a Python variable by using an assignment operator before the SQL magic line operation. For example, consider the following session:

```
In [47]: result = %sql SELECT year, sex, name FROM topnames LIMIT 6

         print(type(result))
         print()
         print(result)
         resultDF = pd.DataFrame(result)
         print()
         print(resultDF)

          * sqlite:///book.db
         Done.
         <class 'sql.run.ResultSet'>

         +------+--------+------+
         | year |  sex   | name |
         +------+--------+------+
         | 1880 | Female | Mary |
         | 1880 |  Male  | John |
         | 1881 | Female | Mary |
         | 1881 |  Male  | John |
         | 1882 | Female | Mary |
         | 1882 |  Male  | John |
         +------+--------+------+

               0       1     2
         0  1880  Female  Mary
         1  1880    Male  John
         2  1881  Female  Mary
         3  1881    Male  John
         4  1882  Female  Mary
         5  1882    Male  John
```

The result is a `sql.run.ResultSet`. You can easily convert it to a Pandas data frame by passing it into the `pd.DataFrame()` function.

- We can use the << operator to assign the results of a multi-line SQL query to a Python variable as in the example below.

```
In [48]: %%sql
         result << SELECT * FROM topnames
         ORDER BY sex, count DESC
         LIMIT 8

          * sqlite:///book.db
         Done.
         Returning data to local variable result
```

```
In [50]: resultDF = pd.DataFrame(result)
         print(resultDF)

               0       1      2      3
         0  1947  Female  Linda  99689
         1  1948  Female  Linda  96211
         2  1949  Female  Linda  91016
         3  1950  Female  Linda  80431
         4  1921  Female   Mary  73985
         5  1951  Female  Linda  73978
         6  1924  Female   Mary  73534
         7  1922  Female   Mary  72173
```

Chapter 11

Relational Model: Single Table Operations

Overview of Chapter

- This chapter focuses on operations involving a single database table.
- We will assume that the example databases are readily available and that a connection has been established.

Example Data Sets

The `indicators0` Table
This is a small subset of the full indicators data set. It contains the following variables, called *fields* or *attributes* in the relational model:

- `code`: the unique international three character code for the country.
- `pop`: the population of the country, in units of millions of persons.
- `gdp`: the gross domestic product of the country, measured in units of billions of US dollars,
- `life`: the life expectancy of the country, in years.
- `cell`: the number of cell phone subscribers in the country, in millions of subscribers.

For this data set, the functional dependency is:

$$\texttt{code} \rightarrow \texttt{pop}, \texttt{gdp}, \texttt{life}, \texttt{cell}$$

Example Data Sets Continued

The `indicators` and `countries` Tables
These are the two tables that comprise the full indicators data. The fields of the `indicators` table include:

- `year`: the year associated with the measurement of the indicators for the given country.
- `code`: the unique international three character code for the country.
- `pop`: the population of the country, in units of millions of persons.
- `gdp`: the gross domestic product of the country, measured in units of billions of US dollars.
- `life`: the average life expectancy of persons in the country, in years.
- `cell`: the number of cell phone subscribers in the country, in millions of subscribers.
- `imports`: the gross dollar value amount of the products imported by the country, measured in units of millions of US dollars.
- `exports`: the gross dollar value amount of the products exported by the country, measured in units of millions of US dollars.

Example Data Sets Continued

The fields of the `countries` table include:

- `code`: the unique international three character code for the country.
- `region`: the world region to which the given country belongs.
- `income`: the average income category for the persons in the given country.
- `land`: the land area of the given country, measured in square kilometers.

For the `indicators` table, the functional dependency is:

`year`, `code` → `pop`, `gdp`, `life`, `cell`, `imports`, `exports`

For the `countries` table, the functional dependency is:

`code` → `region`, `income`, `land`

Example Data Sets Continued

The `topnames` Table

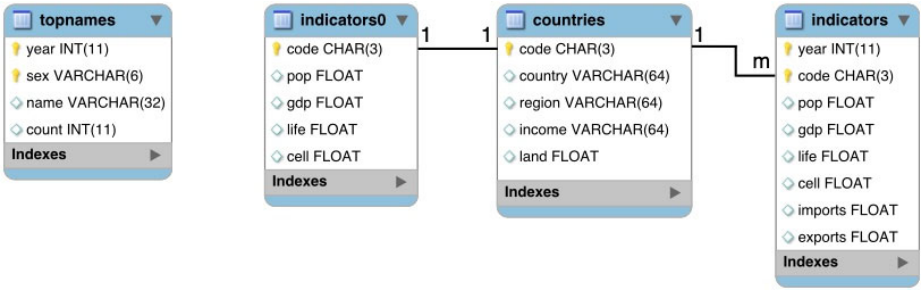
As described previously, the `topnames` table gives, over the years 1880 through 2018, the top female and top male names in applications to the US Social Security administration, along with the count of those applications. The fields of `topnames`, then, consist of:

- `year`: the year associated with the US Social Security top applications.
- `sex`: the sex (`'Female'` or `'Male'`) of the social security application for this record.
- `name`: the name with the most social security applications in the given year for the given sex.
- `count`: the count of applications for the associated name in the given year for the given sex.

And the functional dependency of the table is given by:

`year`, `sex` → `name`, `count`

Schema



Structured Query Language

- SQL stands for "Structured Query Language."
- SQL is a domain-specific language, NOT a general programming language.
- SQL is specialized to handle structured data that follows the relational data model.
- Used to interact with relational databases to manage data: create, populate, modify, or destroy data.

SQL vs Pandas

- The operations of SQL have many parallels with the Pandas operations we learned earlier.
- A significant difference between Pandas and SQL involves the data type of the result.
- In Pandas, the result of the operation might be a scalar (single number), a one-dimensional column (`Series`), or another data frame
- By contrast, **the result of any SQL query is always another two-dimensional table**.

Advantages of SQL

- SQL is a very small language designed to do one thing well—provide a way to efficiently read and write data from a database.
- SQL is not a propriety language. Almost every DBMS supports SQL, so learning SQL will enable you to interact with just about every database.
- Despite its simplicity, SQL is a very powerful language.

SQL Extensions

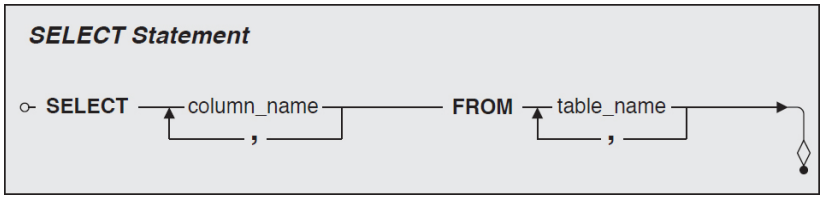
- Many SQL DBMS vendors have extended their support for SQL by adding statements to the language.
- These extensions tend to very DBMS specific, and they are rarely supported by more than one vendor.

SQL Syntax

- The syntax specifications of SQL follow a consistent notation.
- Each specification will consist of ***keywords***, which are reserved words that are part of the SQL language.
  - Never name a table or column using a keyword.
- SQL keywords and table/column names are NOT case sensitive. But **it is customary to present keywords in uppercase**.
- When composing SQL statements, the whitespace (outside of strings) does NOT matter, and we will often write more complex SQL statements using multiple lines and indentation to improve readability.

SELECT Statement

- The most frequently used SQL statement is SELECT.



- The result of this query, as will all SQL queries, is a table.
- Often, the result of a query can contain a very large number of records. We can restrict the results to a limited number of records (similar to the `head()` function) using the LIMIT keyword.

SELECT Statement Example

```
SELECT pop FROM indicators0
```

pop
1386.4
66.87
66.06
1338.66
325.15

```
SELECT name FROM topnames LIMIT 5
```

name
Mary
John
Mary
John
Mary

Projecting More Than One Column

- To project (select) multiple columns, we compose the set of field names in a comma-separated list:

```
SELECT year, sex, name FROM topnames LIMIT 4
```

year	sex	name
1880	Female	Mary
1880	Male	John
1881	Female	Mary
1881	Male	John

The resultant table uses the same column order as the query

AS Keyword

- SQL allows for an *alias* that allows us to rename columns of the result and will be convenient as our queries become more complex.

```
SELECT code, pop AS Population, gdp as GDP
FROM indicators0
```

code	Population	GDP
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
IND	1338.66	2652.55
USA	325.15	19485.4

Note that this query is written using two lines to help with readability!

Retrieving All Columns

- In addition to being able to specify desired columns, SELECT statements can also request all columns without having to list them individual—simply use an asterisk (*).

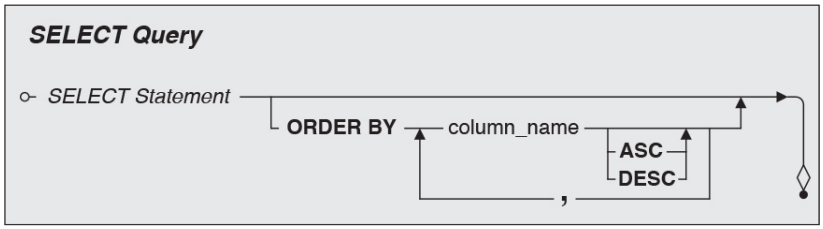
```
SELECT * FROM indicators0
```

code	pop	gdp	life	cell
CHN	1386.4	12143.5	76.4	1469.88
FRA	66.87	2586.29	82.5	69.02
GBR	66.06	2637.87	81.2	79.1
IND	1338.66	2652.55	68.8	1168.9
USA	325.15	19485.4	78.5	391.6

Ordering

- In the relational model, tables within a database do not have an ordering to their records (at least not one we should rely on).

- SQL provides the capability to order (sort) the results of a query with an **ORDER BY** clause that follows the rest of the query and comes before any LIMIT clause. The default ordering is ascending (ASC), but you can sort in descending order using DESC.



Ordering Examples

```
SELECT * FROM topnames
ORDER BY year DESC
LIMIT 4
```

year	sex	name	count
2018	Male	Liam	19837
2018	Female	Emma	18688
2017	Male	Liam	18798
2017	Female	Emma	19800

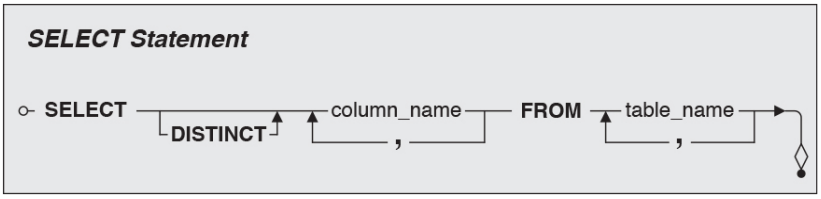
```
SELECT * FROM topnames
ORDER BY year DESC, sex ASC
LIMIT 4
```

Here we are sorting first by year, and then by sex.

year	sex	name	count
2018	Female	Emma	18688
2018	Male	Liam	19837
2017	Female	Emma	19800
2017	Male	Liam	18798

Uniqueness Filter

- Depending on a particular projection, the result of an SQL query could include records that have the same value for all fields of the result (i.e., duplicate records).
- The **DISTINCT** keyword instructs the database to only return distinct values.



DISTINCT Example

```
SELECT DISTINCT region, income FROM countries
LIMIT 6
```

region	income
Latin America & Caribbean	High income
South Asia	Low income
Sub-Saharan Africa	Lower middle income
Europe & Central Asia	Upper middle income
Europe & Central Asia	High income
Middle East & North Africa	High income

Using Comments

- There are two main ways of inserting comments into an SQL query (which are ignored by the database).
- Multi-line SQL comments start with `/*` and end with `*/`, and anything between them is ignored.
- A single line comment uses `--`.

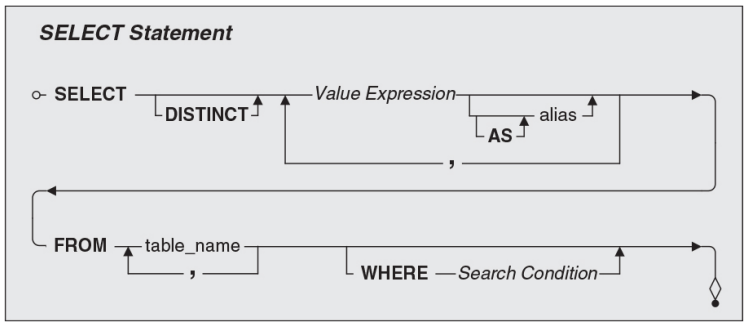
Comments Examples

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers
```

```
--SELECT * FROM Customers;
SELECT * FROM Products;
```

Filtering Data using WHERE

- Row selection involving a condition in SQL is accomplished using the WHERE clause inside of a SELECT statement.
- This type of row selection is called *filtering*.



WHERE Example

```
SELECT * FROM indicators1 WHERE pop > 1000
```

code	year	pop	gdp	life
CHN	2007	1317.88	3550.34	74.6
CHN	2017	1386.4	12143.5	76.4

- We can build more complex Boolean expressions by using logical operators. In SQL, these operators are realized using the keywords AND, OR, and NOT.

WHERE Example 2

```
SELECT * FROM indicators0 WHERE pop > 1000 OR life > 80
```

code	pop	gdp	life	cell
CHN	1386.4	12143.5	76.4	1469.88
FRA	66.87	2586.29	82.5	69.02
GBR	66.06	2637.87	81.2	79.1
IND	1338.66	2652.55	68.8	1168.9

Operators in the WHERE Clause

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

More WHERE Examples

```
SELECT code, country FROM countries
WHERE income = 'High income'
```

String constants are delimited by single quotes

```
SELECT name, sex FROM topnames
WHERE name <> 'John'
```

This only return records for which the name is NOT equal to John

Understanding Order of Evaluation

- WHERE clauses can contain any number of AND and OR operations.
- SQL (like most languages) processes AND operators before OR operators.

Order of Evaluation Example

- Suppose we wanted a list of the countries that have land area greater than 10,000 that are classified as either 'High income' or 'Upper middle income'.
- Consider the following SQL query:

```
SELECT code, income, land FROM countries
WHERE income = 'High income' OR
      income = 'Upper middle income' AND
      land > 10000
LIMIT 10
```

Example Continued

code	income	land
ABW	High income	180.0
ALB	Upper middle income	27400.0
AND	High income	470.0
ARE	High income	71020.0
ARG	Upper middle income	2736690.0
ARM	Upper middle income	28470.0
ATG	High income	440.0
AUS	High income	7692020.0
AUT	High income	82523.0
AZE	Upper middle income	82670.0

- This result is not what we wanted because it contains countries with land area less than 10,000.
- This occurred because AND has a higher order of precedence that OR.

Using Parentheses in WHERE Clause

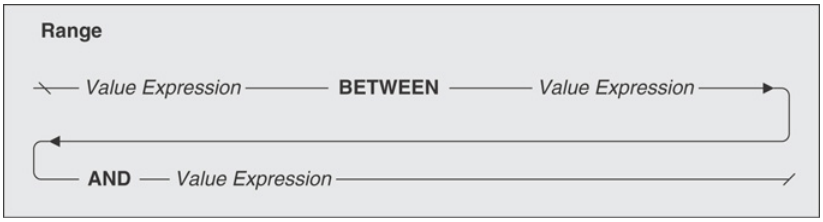
```
SELECT code, income, land FROM countries
WHERE (income = 'High income' OR
      income = 'Upper middle income') AND
      land > 10000
LIMIT 10
```

code	income	land
ALB	Upper middle income	27400.0
ARE	High income	71020.0
ARG	Upper middle income	2736690.0
ARM	Upper middle income	28470.0
AUS	High income	7692020.0
AUT	High income	82523.0
AZE	Upper middle income	82670.0
BEL	High income	30280.0
BGR	Upper middle income	108560.0
BHS	High income	10010.0

This is now correct because the parentheses ensure the correct order of operations.

Checking for a Range of Values

- To check for a range of values we can use the BETWEEN operator.



- Note that the BETWEEN operator is inclusive of the values.

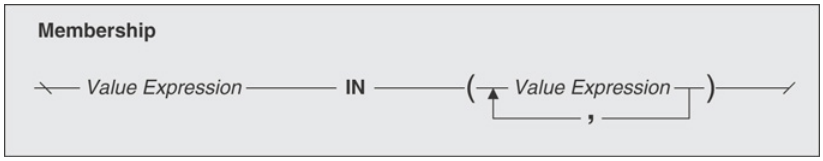
BETWEEN Example

```
SELECT * FROM indicators1
WHERE life BETWEEN 65 AND 75
```

code	year	pop	gdp	life
CHN	2007	1317.88	3550.34	74.6

Using the IN Operator

- The IN operator is used to specify a range of conditions, any of which can be matched.
- IN takes a comma-delimited list of valid values, all enclosed within parentheses.



IN Example

```
SELECT code, pop, gdp, life, cell FROM indicators
WHERE year = 2017 AND code IN ('CHN', 'IND', 'FRA',
'USA')
```

code	pop	gdp	life	cell
CHN	1386.4	12143.5	76.4	1469.88
FRA	66.87	2586.29	82.5	69.02
IND	1338.66	2652.55	68.8	1168.9
USA	325.15	19485.4	78.5	391.6

- The IN operator is cleaner and easier to read than repeatedly using OR operators.

String Pattern Matching with LIKE

- We can use the relational operator (=) to determine if a field matches a string (exactly), but sometimes we need more flexibility matching capabilities.
- We can use the LIKE operator to match string *patterns*.

Wildcards

- ***Wildcards*** are special characters used to match parts of a value.
- The most frequently used wildcard is the percent sign (%). Within a search string, % means *match any number of occurrences of any character*.

Wildcard Example

```
SELECT code, country, region FROM countries
WHERE country LIKE '%Republic'
```

code	country	region
CAF	Central African Republic	Sub-Saharan Africa
CZE	Czech Republic	Europe & Central Asia
DOM	Dominican Republic	Latin America & Caribbean
KGZ	Kyrgyz Republic	Europe & Central Asia
SVK	Slovak Republic	Europe & Central Asia
SYR	Syrian Arab Republic	Middle East & North Africa

This will match fields that **end** with the string 'Republic' as a suffix.

Wildcard Example 2

```
SELECT code, country, region FROM countries
WHERE country LIKE 'Republic%'
```

code	country	region

This will match fields that **begin** with the string 'Republic' as a suffix—there are no such countries!

Wildcard Example 3

```
SELECT code, country, region FROM countries
WHERE region LIKE '%Asia%'
LIMIT 6
```

code	country	region
AFG	Afghanistan	South Asia
ALB	Albania	Europe & Central Asia
AND	Andorra	Europe & Central Asia
ARM	Armenia	Europe & Central Asia
ASM	American Samoa	East Asia & Pacific
AUS	Australia	East Asia & Pacific

This will match fields such that 'Asia' appears anywhere within the field, regardless of any characters before or after.

The Underscore Wildcard

- Another useful wildcard is the underscore (_).
- The underscore is used just like a %, but instead of matching multiple characters, the underscore matches just a single character.
- We can use multiple underscores to match multiple characters.

Underscore Example

```
SELECT code, country, region FROM countries
WHERE code LIKE '_ZA'
```

code	country	region
DZA	Algeria	Middle East & North Africa
TZA	Tanzania	Sub-Saharan Africa

Since there is a single underscore, this will match strings where the first letter is anything, but the last two letters are 'ZA'.

```
SELECT code, country, region FROM countries
WHERE code LIKE '_A'
```

code	country	region

This is empty because there is no string that has a single character followed by an 'A'.

Underscore Example 2

```
SELECT code, country, region FROM countries
WHERE code LIKE '__A'
LIMIT 5
```

code	country	region
BFA	Burkina Faso	Sub-Saharan Africa
BRA	Brazil	Latin America & Caribbean
BWA	Botswana	Sub-Saharan Africa
DMA	Dominica	Latin America & Caribbean
DZA	Algeria	Middle East & North Africa

Note that there are now two underscores, so it will match strings where there are two characters followed by 'A'.

Another Wildcard Example

```
SELECT code, country, region FROM countries
WHERE country LIKE '%Republic' OR code LIKE '_ZA'
```

code	country	region
CAF	Central African Republic	Sub-Saharan Africa
CZE	Czech Republic	Europe & Central Asia
DOM	Dominican Republic	Latin America & Caribbean
DZA	Algeria	Middle East & North Africa
KGZ	Kyrgyz Republic	Europe & Central Asia
SVK	Slovak Republic	Europe & Central Asia
SYR	Syrian Arab Republic	Middle East & North Africa
TZA	Tanzania	Sub-Saharan Africa

Missing Values

- The special value NULL is used to represent missing data in a database.
- NULL should not be mistaken for the value 0, or a blank, or some other data type.
- We can test for NULL using the IS operator.

NULL Examples

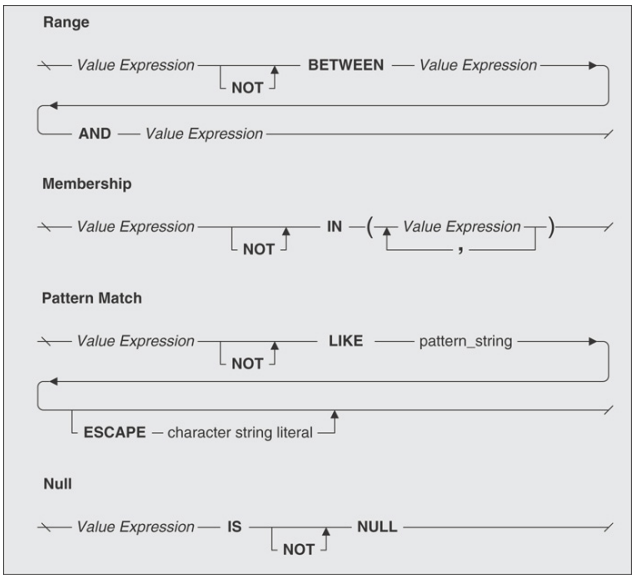
```
SELECT code, country, land FROM countries
WHERE land IS NOT NULL
```

This only return records for which land does **not** have a missing value (indicated by a NULL in the database)

```
SELECT code, country, land FROM countries
WHERE land IS NULL
```

This only return records for which land does have a missing value (indicated by a NULL in the database)

Summary



Calculated Fields

- Data stored within a database's tables is often not available in the exact format needed by your application.
- For example, City, state, and ZIP codes are stored in separate columns (as they should be), but your mailing label printing program needs them as a single field.
- We can use *calculated fields* to address this issue.
- Unlike all the columns that we have retrieved thus far, calculated fields do not actually exist in the database tables. Rather, a **calculated field is created on-the-fly within a SQL SELECT statement.**

Calculated Fields Continued

- When an *expression* follows the SELECT statement, the computed value generates a new field.
- We can name the new field using AS.

Calculated Field Example 1

- Suppose we wanted to select the `code` column of the `indicators0` table and wanted to create a new field (column) called `newpop` that computes a 15% increase from the `pop` field.

```
SELECT code, pop*1.15 AS newpop
FROM indicators0
```

code	newpop
CHN	1594.36
FRA	76.9005
GBR	75.969
IND	1539.459
USA	373.92249999999996

Calculated Field 2

- Consider the expression `life > 75`, which computes Boolean (TRUE/FALSE) values, where TRUE yields a 1 and FALSE yields a 0.

```
SELECT code, life > 75 AS longlife
FROM indicators0
```

code	longlife
CHN	1
FRA	1
GBR	1
IND	0
USA	1

Concatenating Fields

- In SQL SELECT statements, you can concatenate columns using a special operator.
- Depending on what DBMS you are using, this operator can be a plus sign (+) or two pipes (||).
  - SQL Server uses + for concatenation.
  - DB2, Oracle, PostgreSQL, and SQLite support ||.
  - You will need to refer to the documentation of the DBMS that you are using.

Concatenation Example

```
SELECT country || ' (' || code || ')' AS
     country2, region
FROM countries
WHERE code IN ('FRA', 'GBR', 'IND', 'USA')
ORDER BY country2
```

country2	region
France (FRA)	Europe & Central Asia
India (IND)	South Asia
United Kingdom (GBR)	Europe & Central Asia
United States (USA)	North America

Aggregation

- In the context of a single table, we want the ability to aggregate one or more columns, possibly in conjunction with filtering so that the aggregation is over a subset of rows.

The COUNT() Function

- Using COUNT(), we can determine the number of rows in a table or the number of rows that match a specific criteria.
- We can use COUNT() in two ways:
  - Use COUNT(*) to count the number of rows in a table (including rows that have NULLs).
  - Use COUNT(column) to count the number of rows that have values in a specific column, ignoring NULL values.

COUNT Examples

```
SELECT COUNT(*) FROM countries
```

COUNT(*)
217

Column rows with NULL values in them are ignored by COUNT() function if a column name is specified, but not if the asterisk (*) is used.

```
SELECT COUNT(*) AS total,
       COUNT(land) AS has_area
FROM countries
```

total	has_area
217	210

Note that the total number of rows is 217, but only 210 rows of the `land` column have non-NULL values

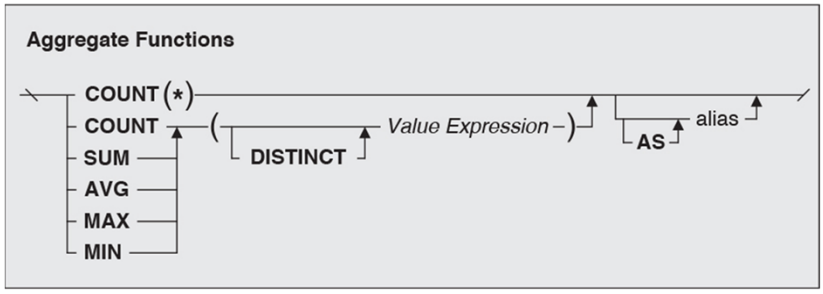
Common SQL Aggregation Methods

Table 11.34 Common SQL aggregation functions

Aggregation function	Description
`AVG` (*expr*)	Compute the average of the non-NULL values of *expr* from the rows of the result
`MAX` (*expr*)	Compute the maximum non-NULL value of *expr* from the rows of the result
`MIN` (*expr*)	Compute the minimum non-NULL value of *expr* from the rows of the result
`SUM` (*expr*)	Compute the sum of the non-NULL values of *expr* from the rows of the result

Uniqueness

- COUNT() can optionally include the DISTINCT keyword in the argument and then it will count the unique instances of a field.



Aggregation Examples

```
SELECT COUNT(DISTINCT name) AS numname
FROM topnames
```

numname
18

Aggregation Examples

```
SELECT COUNT(*) AS numrec, MAX(gdp) AS maxgdp,
       AVG(gdp) AS avggdp, MIN(life) AS minlife
FROM indicators0
```

numrec	maxgdp	avggdp	minlife
5	19485.4	7901.122	68.8

Subqueries

- All the SELECT statements we have seen thus far are simply queries—single statements retrieving data from individual database tables.
- SQL also enables you to create subqueries—queries that are embedded into other queries.

Subquery Example

- Suppose we wish to find the countries for which the GDP is above average from the `indicators0` table.
- We could compute the average GDP and use it in a WHERE clause to filter for the rows that exceed the average.

```
SELECT code, pop, gdp FROM indicators0
WHERE gdp > (SELECT AVG(gdp) FROM indicators0)
```

code	pop	gdp
CHN	1386.4	12143.5
USA	325.15	19485.4

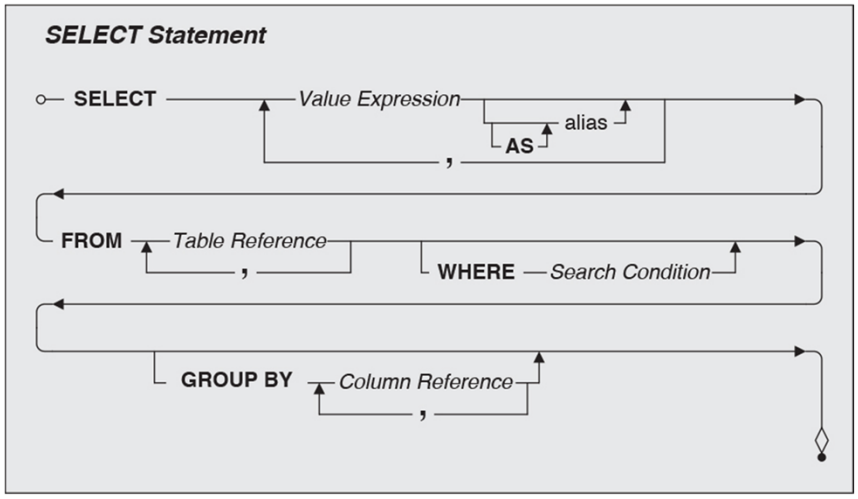
The subquery is contained inside of the paratheses.

In general, a subquery can be used in places where a table can be used.

Partitioning and Aggregating

- Aggregations that are based on the partitioning of a data table allow us to compare between different categories of the rows in our table.
- The SQL adds a GROUP BY clause that follows the WHERE clause. After the GROUP BY keywords, we specify one or more field-spec, which are used to determine, based on their unique set of values, the partitions.

GROUP BY



GROUP BY Example

```
SELECT region, SUM(land) AS region_area
FROM countries
GROUP BY region
```

region	region_area
East Asia & Pacific	24361338.4
Europe & Central Asia	27429254.6
Latin America & Caribbean	20038832.0
Middle East & North Africa	11223466.0
North America	18240984.0
South Asia	4771604.0
Sub-Saharan Africa	21242361.0

GROUP BY Example 2

```
SELECT sex, MIN(count) AS min_count,
       AVG(count) AS avg_count,
       MAX(count) AS max_count,
       COUNT(DISTINCT name) as num_name
FROM topnames
GROUP BY sex
```

sex	min_count	avg_count	max_count	num_name
Female	6919	42372.676258992804	99689	10
Male	6900	47597.81294964029	94757	8

GROUP BY Example 3

```
SELECT name, sex, COUNT(*) AS num_name,
AVG(count)
AS avg_count
FROM topnames
GROUP BY name, sex
ORDER BY num_name DESC, avg_count DESC
```

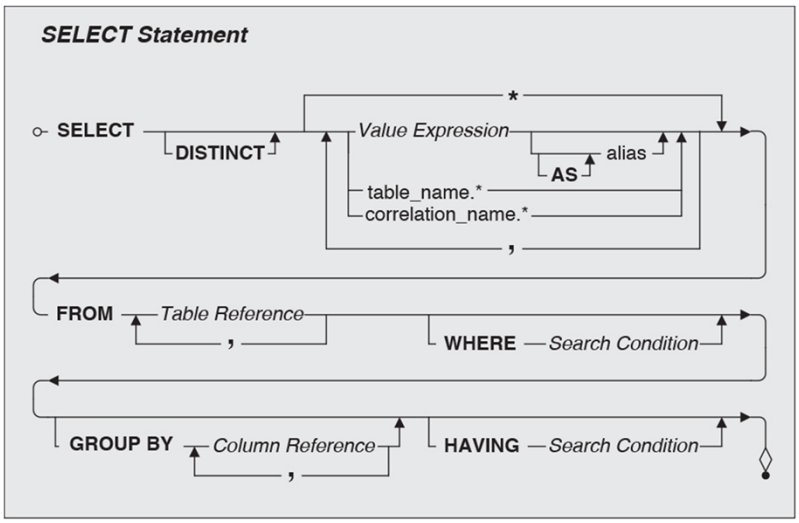
Note that we are grouping by name ***and*** sex

name	sex	num_name	avg_count
Mary	Female	76	40768.78947368421
Michael	Male	44	70109.63636363637
John	Male	44	19577.340909090908
Robert	Male	17	61293.17647058824
Jennifer	Female	15	57280.6
Jacob	Male	14	26484.214285714286

Filtering Groups?

- We have already learned about the WHERE clause to filter rows.
- Unfortunately, WHERE does not work with GROUP BY because it filters rows, not groups.
- SQL provides another clause, HAVING, that is very similar to WHERE.
- The only difference is that WHERE filters rows and HAVING filters groups.

Filtering Groups



Filtering By Groups Example

```
SELECT region, SUM(land) as region_area
FROM countries
GROUP BY region
HAVING region_area < 20000000
```

region	region_area
Middle East & North Africa	11223466.0
North America	18240984.0
South Asia	4771604.0

Chapter 12

Relational Model: Multiple Table Operations

Chapter Goals

- Much of the power in SQL is enabled by the operations that allow us to query tables in combination.
- Conceptually, we have already seen the combination of multiple tables with the `concat`/`merge`/`join` operations in Pandas (tabular model).
- In the tabular model, we have learned that many tables are not *tidy*. However, in the relational model, database design and normalization is much more formal and stricter, and violations are rare.

Overview of Join Operations

- One of SQL's most powerful features is the capability to join tables on-the-fly within data retrieval queries.
- Joins are one of the most important operations that you can perform using SQL SELECT.
- Joins in the relational model have the same operational meaning as when we encountered them in the tabular model.

Overview of Join Operations Continued

- In general, a join allows us to combine two tables based on a match condition that involves comparing the values of one or more field in one table to the values of corresponding fields in another table.
- When there is a match, we yield a row in the result, and the values of the fields in that row are the values from the fields of the pair of records that produced a match. That is, each row has columns from both tables.
- The variations of the join operation (left join, inner join, etc.) come about when the value of fields used in the match condition in one table has no matching value in the second table, or vice-versa.

Join Variations

- Consider the following tables:

```
SELECT * FROM pop_gdp
```

code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

```
SELECT * FROM country_land
```

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

Inner Join

- **Inner Join:** include rows when both tables have corresponding fields based on the match condition.
- In this example, this occurs for the three codes of FRA, GBR, and USA.

pop_gdp

code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

country_land

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

Left Join

- **Left Join:** include all rows of the `pop_gdp` table (the left table) and, for cases where there is no corresponding code value in the right table, fill in as missing the values for columns/rows in the right table that have no correspondence.
- In this example, the result would include rows with a code for CHN, FRA, GBR, and USA.

`pop_gdp`

code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

`country_land`

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

Right Join

- **Right Join:** include all rows of the `country_land` table (the right table) and, for cases where there is no corresponding code value in the left table, fill in as missing the values for columns/rows in the left table that have no correspondence.
- In this example, the result would include rows with a code for FRA, GBR, IND, USA, and VNM.

`pop_gdp`

code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

`country_land`

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

Outer Join

- **Outer Join:** merge all rows from both tables, and fill in as missing for rows/columns in either table that lack a correspondence.
- In this example, the result would include rows with a code for CHN, FRA, GBR, IND, USA, and VNM.

`pop_gdp`

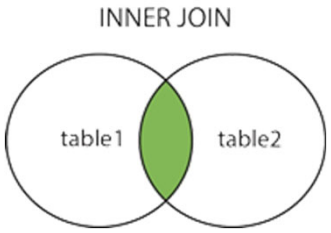
code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

`country_land`

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

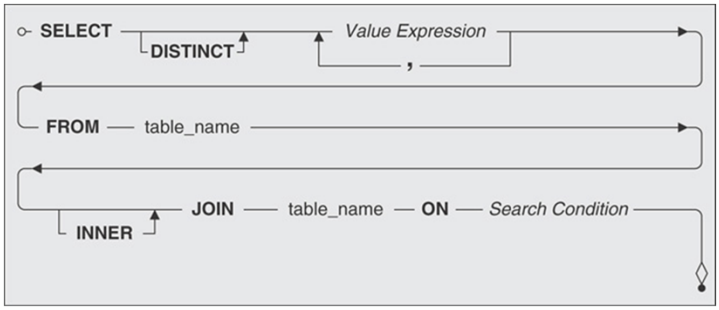
Inner Join

- An inner join selects records that have matching values in both tables.



Two Table SQL Inner Join

- We use an inner join either when we know that the matches will be complete or when the problem we are trying to solve requires that the resulting table has meaningful values for all the columns to be included in the result, even at the price of eliminating the rows that fail to have matches in one direction or the other.



Inner Join Example

```
SELECT pop_gdp.code, pop_gdp.pop, pop_gdp.gdp,
       country_land.code, country_land.country,
       country_land.land
FROM pop_gdp INNER JOIN country_land
       ON pop_gdp.code = country_land.code
```

- Note the columns specified as `pop_gdp.code` and `country_land.code`. These ***fully qualified column names*** (table and column separated by a period) is required here because if you just specified `code`, the DBMS cannot tell which `code` columns you are referring two (there are two of them, one in each table).

Inner Join Example

pop_gdp

code	pop	gdp
CHN	1386.4	12143.5
FRA	66.87	2586.29
GBR	66.06	2637.87
USA	325.15	19485.4

country_land

code	country	land
FRA	France	547557.0
GBR	United Kingdom	241930.0
IND	India	2973190.0
USA	United States	9147420.0
VNM	Vietnam	310070.0

Each DBMS must deal with the fact that some field names might be the same between tables. SQLite appended a '_1', but other systems may do something else.

code	pop	gdp	code_1	country	land
FRA	66.87	2586.29	FRA	France	547557.0
GBR	66.06	2637.87	GBR	United Kingdom	241930.0
USA	325.15	19485.4	USA	United States	9147420.0

Inner Join Example 2

- We can use aliases for the tables involved in the join, and also for fields specified in the SELECT to both shorten the query and to resolve any repeated column names.

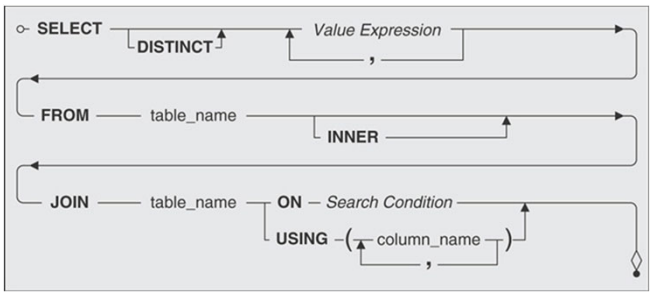
```
SELECT PG.code, PG.pop, PG.gdp,
       CL.code AS cl_code, CL.country, CL.land
FROM
       pop_gdp AS PG INNER JOIN country_land AS CL
       ON PG.code = CL.code
```

Inner Join Example 2

code	pop	gdp	cl_code	country	land
FRA	66.87	2586.29	FRA	France	547557.0
GBR	66.06	2637.87	GBR	United Kingdom	241930.0
USA	325.15	19485.4	USA	United States	9147420.0

USING Keyword

- In many cases, an inner join match-condition involves equality between two fields that are named the same between the left and right tables. In this case we can simplify the inner join syntax and instead of using the ON with a match condition, we use USING and put the one or more columns to be used for matching in a comma-separated list in parentheses following the USING keyword.



USING Example

```
SELECT *
FROM pop_gdp INNER JOIN country_land
USING (code)
```

code	pop	gdp	country	land
FRA	66.87	2586.29	France	547557.0
GBR	66.06	2637.87	United Kingdom	241930.0
USA	325.15	19485.4	United States	9147420.0

Match on Multiple Fields

- More generally, a match might require the equality of multiple fields. We illustrate with the two tables below.

names

year	sex	count
2016	Female	19496
2016	Male	19117
2017	Female	19800
2017	Male	18798

counts

year	sex	count	name
2016	Female	19496	Emma
2016	Male	19117	Noah
2017	Female	19800	Emma
2017	Male	18798	Liam

```
SELECT C.year, C.sex, C.count, N.name
FROM counts AS C INNER JOIN names AS N
   ON C.year = N.year AND C.sex = N.sex
```

Match on Multiple Fields

```
SELECT C.year, C.sex, C.count, N.name
FROM counts AS C INNER JOIN names AS N
   ON C.year = N.year AND C.sex = N.sex
```

year	sex	count	name
2016	Female	19496	Emma
2016	Male	19117	Noah
2017	Female	19800	Emma
2017	Male	18798	Liam

Match on Multiple Fields with USING

```
SELECT *
FROM counts INNER JOIN names
USING (year, sex)
```

year	sex	count	name
2016	Female	19496	Emma
2016	Male	19117	Noah
2017	Female	19800	Emma
2017	Male	18798	Liam

New Example Data Set

- To help illustrate some of the variations of multiple table operations, we will need a more sophisticated example database.
- The `school` database will be used for these more involved examples.

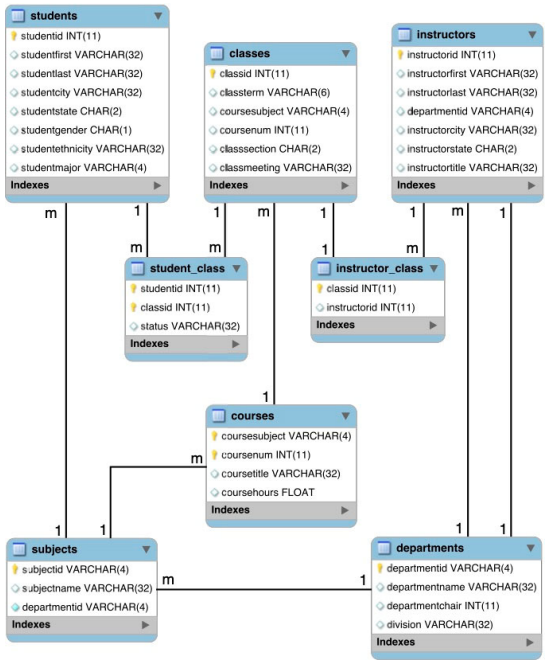
`school` Schema

Table 12.1 Tables in the `school` schema

Table name	Description
`students`	Maintains information about the full set of students in the data set. A generated id is used as the primary key, and the table includes fields for the student's name, home city/state, gender, ethnicity, and major
`instructors`	The full set of instructors at the school, with a generated id as primary key, name, id of the instructor's department, city/state part of their address, and a title
`departments`	The full set of departments at the school, with a character id for the department, the full name of the department, the id for the instructor serving as chair of the department, and a string giving the division to which the department belongs
`subjects`	This table lists the subjects that are taught at the school. A given department can teach multiple subjects. Each subject has a short character id, along with a longer textual name and a field giving the department, which is home to the subject

`school` Schema Continued

`courses`	The table represents the full catalog of courses taught at the school. Each course has an associated subject, along with an integer course number. The combination of a subject and number will be unique. A course also has a longer title and a number of credit hours
`classes`	The classes table represents an instance of a course, taught as one of the possibly multiple sections, and in a particular term. It has a unique integer id along with the course subject and number for the course it is an instance of. It also has a string representing its meeting days and times
`student_class`	This is a linking table, where entries consist of a student id and a class id. An entry in the table indicates that the given student is/was registered for the given class. A status entry gives information about, for a student/class pair whether the course was "normal" or "withdrawn"
`instructor_class`	This is another linking table, where entries consist of an instructor id and a class id. An entry in the table indicates that the given instructor was the teacher of record for the given class



Things to Notice in `school`

- Given a particular course, there can be multiple class instances of that course.
- Given a particular department, it can host multiple subjects.
- Multiple students (by their major) are associated with the same subject.
- Each class is present exactly once in the `instructor_class` table, but a particular instructor can be present in multiple records of the `instructor_class` table.
- Given a particular student, there can be multiple records in the `student_class` table.
- Given a particular class, there can be multiple records in the `student_class` table.

Inner Join to Fill Redundant Fields

- A common use for an inner join occurs when, by reducing redundancy in database design, we have split tables based on non-identical independent variables of the functional dependencies.
- Suppose a student has registered for four classes, whose `classid` fields are given by 21014, 21088, 21256, and 21444.
- If we want to find the instructors of those classes, we can use the *linking field* of `instructorid` to link to the instructors table, where `instructorid` is the primary field (intentionally named the same), and, by joining, can fill in a combined table starting with the `classid` fields and obtaining the instructor's last and first names.

Example

```
SELECT classid, instructorid, instructorlast,
       instructorfirst
FROM instructor_class INNER JOIN instructors
USING (instructorid)
WHERE classid IN (21014, 21088, 21256, 21444)
```

classid	instructorid	instructorlast	instructorfirst
21014	9167	Foster	Helen
21088	9029	Hawkins	Grace
21256	9146	Garrett	Jason
21444	9050	Price	Taylor

Joining Multiple Tables

- SQL imposes no limit to the number of tables that may be joined in a SELECT statement.
- The basic rules for creating the join remain the same. First, list all the tables, and then define the relationship between each.

Three-Table Join Example

- For example, in the `school` database, suppose that we would like to see the subject, number, section, and meeting time for each of a set of classes combined with the instructor teaching the class using the `instructorlast` and `instructorfirst` fields.

Example Continued

- To just see information about a particular set of classes, we would issue a single table query:

```
SELECT coursesubject || '-' || coursenum ||
       '-' || classsection
AS class, classmeeting
FROM classes
WHERE classid IN (21014, 21088, 21256, 21444)
```

class	classmeeting
PSYC-100-02	09:30-10:20 MWF
FYS-102-02	13:30-14:50 TR
ECON-240-01	11:30-13:20 MW
PHED-180-01	None

Example Continued

- The mapping of instructors to classes resides in a different table: `instructor_class`. So to retrieve the instructor for this class, we join on `classid` between the two tables:

```
SELECT coursesubject || '-' || coursenum ||
       '-' || classsection
AS class, classmeeting, instructorid
FROM classes INNER JOIN instructor_class USING (classid)
WHERE classid IN (21014, 21088, 21256, 21444)
```

class	classmeeting	instructorid
PSYC-100-02	09:30-10:20 MWF	9167
FYS-102-02	13:30-14:50 TR	9029
ECON-240-01	11:30-13:20 MW	9146
PHED-180-01	None	9050

Example Continued

- To get to the `instructorlast` and `instructorfirst`, we need to use the `instructorid` from this joined table result and join with the `instructors` table to then fill in the desired instructor fields:

Example Continued

```
SELECT coursesubject || '-' || coursenum ||
       '-' || classsection
AS class, classmeeting, instructorlast, instructorfirst
FROM classes INNER JOIN instructor_class USING (classid)
INNER JOIN instructors USING (instructorid)
WHERE classid IN (21014, 21088, 21256, 21444)
```

class	classmeeting	instructorlast	instructorfirst
PSYC-100-02	09:30-10:20 MWF	Foster	Helen
FYS-102-02	13:30-14:50 TR	Hawkins	Grace
ECON-240-01	11:30-13:20 MW	Garrett	Jason
PHED-180-01	None	Price	Taylor

Joining Multiple Table Considerations

- In the FROM clause, inner joins are processed in the left to right order they appear. If we want a different order, we can use parentheses around a join clause to enforce a different precedence.
- While it is true that SQL itself has no maximum number of tables per join restriction, many DBMSs do indeed have restrictions. Refer to you DBMS documentation to determine what restrictions there are, if any.

Performance Considerations

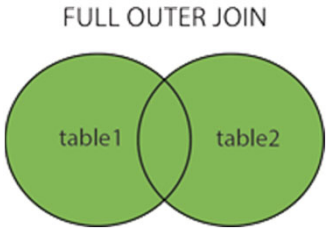
- DBMSs process joins at runtime relating each table as specified. This process can become very resource intensive, so be careful not to join tables unnecessarily.
- The more tables you join, the more performance will degrade.
- There is often more than one way to perform any given SQL operation. Performance can be affected by the type of operation, the DBMS being used, the amount of data in the tables, etc. It is often worth experimenting with different selection mechanisms to find the most efficient version.

Outer Joins

- An outer join allows us to control the join operation in the situation where records in one table do not have matching records in another table, beyond the inner join solution of throwing away any such cases.
- Let us assume that, in the binary operation of a join, the first table specified, which is positionally on the left side of the join operator, is called the ***left table***, and the second table specified, which is positionally on the right side of the join operator, is called the ***right table***.
- There are three cases to consider in the situation where a record in one table has no match in the other table:

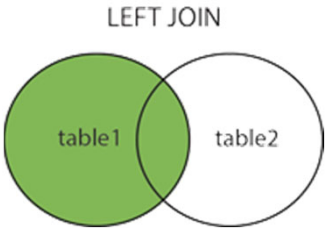
Full Outer Join

1. Return all the records that have values in either the left or right table. This is called a ***full outer join*** or just an ***outer join***.



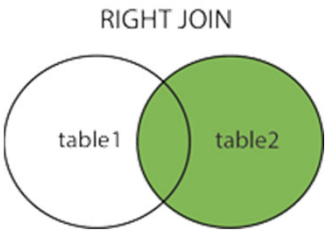
Left Outer Join

2. Return all records from the left table, and matching records from the right table. This is called a ***left outer join*** or just a ***left join***.



Right Outer Join

3. Return all records from the right table, and matching records from the left table. This is called a ***right outer join*** or just a ***right join***.



- Clearly, cases 2 and 3 are dual to each other, and we could transform a query from one to the other by swapping which is the left table and which is the right table.

Left Join Example

`ind0`

Code	Year	Pop	Gdp
FRA	2007	64.02	2657.21
GBR	2007	61.32	3084.12
USA	2007	301.23	14451.90
FRA	2017	66.87	2586.29
GBR	2017	66.06	2637.87
USA	2017	325.15	19485.40

`countries0`

Code	Country	Income
GBR	United Kingdom	High income
IND	India	Lower middle income
USA	United States	High income

```
SELECT I.*, C.code AS c_code, country, income
FROM ind0 AS I LEFT JOIN countries0 AS C
ON (I.code = C.code)
```

code	year	pop	gdp	c_code	country	income
FRA	2007	64.02	2657.21	None	None	None
GBR	2007	61.32	3084.12	GBR	United Kingdom	High income
USA	2007	301.23	14451.9	USA	United States	High income
FRA	2017	66.87	2586.29	None	None	None
GBR	2017	66.06	2637.87	GBR	United Kingdom	High income
USA	2017	325.15	19485.4	USA	United States	High income

Note the missing values corresponding to FRA, which was not present in the right table

Right Join Example

ind0

Code	Year	Pop	Gdp
FRA	2007	64.02	2657.21
GBR	2007	61.32	3084.12
USA	2007	301.23	14451.90
FRA	2017	66.87	2586.29
GBR	2017	66.06	2637.87
USA	2017	325.15	19485.40

countries0

Code	Country	Income
GBR	United Kingdom	High income
IND	India	Lower middle income
USA	United States	High income

```
SELECT I.*, C.code AS c_code, country, income
FROM ind0 AS I RIGHT JOIN countries0 AS C
ON (I.code = C.code)
```

Code	Year	Pop	Gdp	c_code	Country	Income
GBR	2007	61.32	3084.12	GBR	United Kingdom	High income
GBR	2017	66.06	2637.87	GBR	United Kingdom	High income
NA	NA	NA	NA	IND	India	Lower middle income
USA	2007	301.23	14451.90	USA	United States	High income
USA	2017	325.15	19485.40	USA	United States	High income

Need for Both Left and Right Joins?

- Because of their equivalent power, some DBMSs (such as SQLite and MySQL) do not implement a RIGHT JOIN and only provide a LEFT JOIN.
- However, we can easily achieve a right join using a left join by just swapping the left and right tables.

Another Left Join Example

- Let us again consider our `school` database.
- Using the departments table, we can find the set of divisions and the number of departments for each division:

```
SELECT division, COUNT(*) AS num_dept
FROM departments
GROUP BY division
```

division	num_dept
Fine Arts	5
Humanities	6
Interdisciplinary	13
Natural Sciences	6
Social Sciences	6

Note that the Natural Sciences division consists of six departments.

Another Left Join Example Continued

- Suppose our goal is to list all the departments in a particular division, including the name of the department chair.
- We recognize the problem as needing a join between the `departments` table and the `instructors` table with a match condition of the `departments.departmentchair` equaling the `instructors.instructorid`.

Another Left Join Example Continued

- A query to generate this table for all departments in the Natural Sciences division using an inner join might look like the following:

```
SELECT D.departmentid, D.departmentname,
  I.instructorlast || ', ' || I.instructorfirst AS chairname
FROM departments AS D INNER JOIN instructors AS I
  ON (D.departmentchair = I.instructorid)
WHERE division = 'Natural Sciences'
```

departmentid	departmentname	chairname
CHEM	Chemistry and Biochemistry	Burton, Judith
MATH	Mathematics & Computer Science	Bradley, Betty
PHYS	Physics	Lee, Jasmine
PSYC	Psychology	Nair, Hemant

Recall that there are six departments in the Natural Sciences division, but there are only four listed here?

Another Left Join Example Continued

- The (possibly unexpected) result has fewer departments listed than the full set of departments in the sciences division.
- The problem is that, in the database, there can be valid reasons why a `departmentchair` value is missing, for instance, in a transition between chairs.
- But in our goal, we want to list *all* departments, not just ones with a current chair. Since we want all the records in the left table and then the information, where possible, from the right table, the left join is the correct operation to select here.

Another Left Join Example Continued

```
SELECT D.departmentid, D.departmentname,
  I.instructorlast || ', ' || I.instructorfirst
AS chairname
FROM departments AS D LEFT JOIN instructors AS I
  ON (D.departmentchair = I.instructorid)
WHERE division = 'Natural Sciences'
```

departmentid	departmentname	chairname
BIOL	Biology	None
CHEM	Chemistry and Biochemistry	Burton, Judith
GEOS	Geosciences	None
MATH	Mathematics & Computer Science	Bradley, Betty
PHYS	Physics	Lee, Jasmine
PSYC	Psychology	Nair, Hemant

Now all six departments in the Natural Sciences division listed, even those that do not currently have chairs.

Full Outer Join Example

- The SQL syntax for a full outer join parallels the syntax for the INNER JOIN, LEFT JOIN, and RIGHT JOIN by simply replacing with FULL OUTER JOIN.

```
SELECT I.*, C.code AS c_code, country, income
FROM ind0 AS I FULL OUTER JOIN countries0 as C
ON (I.code = C.code)
```

- Many real systems do not provide a full outer join (such MySQL and SQLite), but this can be emulated by taking the UNION of two left joins.

Emulating an Outer Join

```
SELECT I.*, C.code AS c_code, country, income
FROM ind0 AS I LEFT JOIN countries0 AS C
  ON (I.code = C.code)
UNION
SELECT I.*, C.code AS c_code, country, income
FROM countries0 AS C LEFT JOIN ind0 AS I
  ON (I.code = C.code)
```

Outer Join Example

ind0

Code	Year	Pop	Gdp
FRA	2007	64.02	2657.21
GBR	2007	61.32	3084.12
USA	2007	301.23	14451.90
FRA	2017	66.87	2586.29
GBR	2017	66.06	2637.87
USA	2017	325.15	19485.40

countries0

Code	Country	Income
GBR	United Kingdom	High income
IND	India	Lower middle income
USA	United States	High income

code	year	pop	gdp	c_code	country	income
None	None	None	None	IND	India	Lower middle income
FRA	2007	64.02	2657.21	None	None	None
FRA	2017	66.87	2586.29	None	None	None
GBR	2007	61.32	3084.12	GBR	United Kingdom	High income
GBR	2017	66.06	2637.87	GBR	United Kingdom	High income
USA	2007	301.23	14451.9	USA	United States	High income
USA	2017	325.15	19485.4	USA	United States	High income

Key Takeaway

- An outer join (left, right, or full) lets you see both the matches and the records that failed to match.

Another Example of Outer Join

- Suppose we wanted to find all the instructors who were *not* teaching in the fall semester.
- We could find all the instructors who *were* teaching in the fall using an INNER JOIN between the `instructors` and `classes` tables passing through the `instructor_class` table.

```
SELECT I.instructorid, I.instructorfirst, I.instructorlast,
  C.coursesubject, C.coursenum,C.classterm
FROM (instructors AS I
      INNER JOIN instructor_class USING (instructorid)
     )
INNER JOIN classes AS C USING (classid)
WHERE classterm = 'FALL'
```

Example Continued

Table 12.32 All instructors who were teaching during the fall semester

instructorid	instructorfirst	instructorlast	coursesubject	coursenum	classterm
9204	Antoine	Arnaud	THTR	100	FALL
9114	Martha	Meyer	THTR	110	FALL
9122	Gary	Carlson	THTR	120	FALL
9267	Robert	Brooks	THTR	160	FALL
9122	Gary	Carlson	THTR	165	FALL
9267	Robert	Brooks	THTR	160	FALL
9114	Martha	Meyer	THTR	165	FALL
9204	Antoine	Arnaud	THTR	170	FALL
9204	Antoine	Arnaud	THTR	170	FALL
9242	Crystal	Boyd	THTR	230	FALL

- What we actually want is all rows in instructors that are not present in the query above.

Example Continued

- This can be accomplished by a LEFT JOIN, starting from `instructors` and joining to the table resulting from the query above.
- This will give us all rows in `instructors` that are not present in the previous subquery.

```
SELECT DISTINCT I2.instructorid, I2.instructorfirst,
  I2.instructorlast
FROM instructors as I2
LEFT JOIN
(SELECT I.instructorid, I.instructorfirst, I.instructorlast,
  C.coursesubject, C.coursenum,C.classterm
FROM (instructors AS I
      INNER JOIN instructor_class USING (instructorid)
     )
INNER JOIN classes AS C USING (classid)
WHERE classterm = 'FALL') USING (instructorid)
```

Example Continued

instructorid	instructorfirst	instructorlast
9000	Brandon	Santos
9001	Margaux	Gillet
9002	Tom	Werner
9003	Roger	Davidson
9004	Sharon	Jimenez
9005	Joyce	Bell
9006	Morgan	Johnson
9007	Dorothy	Perry
9008	Anthony	Duncan
9009	Ajay	Jayaraman

Set Difference

- This same pattern can be used whenever we wish to find all individuals in a ***set difference***, i.e., individuals satisfying some condition (being in a table A) but not some other condition (being in a table B).
- In this generality, we seek individuals whose B-values are NULL in A LEFT JOIN B. To find individuals in B but not in A, we seek individuals whose A-values are NULL in B LEFT JOIN A.

Partitioning and Grouping Information

- When we combine the ability of combining tables through a JOIN with the partitioning and aggregation of a GROUP BY, we gain considerable power in our SQL queries.
- For our first example, we consider the `indicators` and `countries` tables.
- Suppose we want to compare life expectancy in the year 2017 among the countries of the world organized by their region.

Partitioning and Grouping Example

```
SELECT region, ROUND(AVG(life),2) AS avg_life,
    COUNT(*) AS num_country, COUNT(life) AS num_life
FROM indicators LEFT JOIN countries
    USING (code)
WHERE year = 2017
GROUP BY region
ORDER BY avg_life DESC
```

region	avg_life	num_country	num_life
North America	80.8	3	3
Europe & Central Asia	77.83	58	52
Latin America & Caribbean	75.1	42	36
Middle East & North Africa	74.74	21	21
East Asia & Pacific	74.63	37	31
South Asia	70.81	8	8
Sub-Saharan Africa	62.04	48	48
None	None	1	0

Partitioning and Grouping Example 2

- In similar fashion, we can ask the same question across countries grouped by income categories.

```
SELECT income, ROUND(AVG(life),2) AS avg_life,
  COUNT(*) AS num_country, COUNT(life) AS num_life
FROM indicators LEFT JOIN countries
  USING (code)
WHERE year = 2017
GROUP BY income
ORDER BY avg_life DESC
```

income	avg_life	num_country	num_life
High income	79.73	79	66
Upper middle income	73.69	60	55
Lower middle income	67.64	47	47
Low income	62.14	31	31
None	None	1	0

Subqueries

- A subquery is a fully formed SQL statement that is subordinate to another SQL query.
- We now illustrate some of the most common forms of subquery.
- As one might expect, the subquery has precedence over the outer query and gets executed first. SQL also restricts the ability to perform a sort on subqueries, and so a subquery cannot include an ORDER BY clause.
- We can still, however, use an ORDER BY in the outer query.

Subquery with Singleton for Comparison in WHERE

```
SELECT code, gdp, life FROM indicators
WHERE year=2017 AND
  gdp > (SELECT AVG(gdp)*5
         FROM indicators
         WHERE year=2017)
```

code	gdp	life
BRA	2053.59	75.7
CHN	12143.5	76.4
DEU	3693.2	81.0
FRA	2586.29	82.5
GBR	2637.87	81.2
IND	2652.55	68.8
JPN	4859.95	84.1
USA	19485.4	78.5

Subquery with Vector of Values for IN

```
SELECT AVG(gdp) AS longlife_gdp
FROM indicators
WHERE year = 2017 AND
code IN (SELECT DISTINCT code
         FROM indicators
         WHERE life > 83)
```

longlife_gdp
1313.3771428571429

Subquery Providing Virtual Table in FROM

```
SELECT studentmajor, nummajor
FROM (SELECT studentmajor, COUNT(*) AS nummajor
      FROM students
      WHERE studentmajor IS NOT NULL
      GROUP BY studentmajor) AS majors
ORDER BY nummajor DESC
LIMIT 6
```

studentmajor	nummajor
ECON	251
BIOL	237
PSYC	196
COMM	160
HIST	131
POSC	123

Another Example

```
SELECT studentmajor, nummajor, departmentid
FROM (SELECT studentmajor,
      COUNT(*) AS nummajor
      FROM students
      WHERE studentmajor IS NOT NULL
      GROUP BY studentmajor) AS majors
    INNER JOIN
      subjects AS S
    ON S.subjectid = majors.studentmajor
LIMIT 8
```

Example Output

studentmajor	nummajor	departmentid
ARTH	17	ART
ARTS	42	ART
BCHM	44	CHEM
BIOL	237	BIOL
BLST	3	BLST
CHEM	39	CHEM
CINE	31	CINE
CLAS	5	CLAS

Chapter 13

Relational Model: Database Programming

Chapter Goals

- Now that we understand how to use SQL to build queries, we must learn how to use our knowledge within a client application written in a programming language.
- **That is, we need to graduate from using SQL Magic (which is not Python) to Python programming**:
  - to issue queries
  - incorporating Python variables
  - to obtain and process results
  - to incorporate into data frames
  - to create table and database schema
  - to populate database tables with rows of data

Chapter Goals

1. How to establish a connection to the provider and, if the provider supports multiple databases (schema), how to select a specific database.
2. How to build an SQL query in the programming language we are using. These queries will often have to incorporate the values of variables of the executing program.
3. Given a constructed query, how to send it to the provider.
4. How to retrieve the response from the provider.
5. Given the response, how to extract the table of data and incorporate it into data structures of the executing program.
6. How to close or terminate a connection with the provider.

Client/Provider Interaction

Client Application		Provider RDBMS
1. Establish Connection	→	1. Accept Connection
2. Select Database	→	2. Set Default Database
3. Repeat		3. Repeat
a. Transmit SQL Query	→	a. Receive SQL Query
		b. Optimize and Execute Query
b. Receive Table Response	←	c. Transmit Response
c. Process Response		
4. Close Connection	↔	4. Close Connection

Database Connection

- For many database systems, the connection may occur over a network, but for some, like SQLite, this connection is *virtual* and is realized through interaction with the local operating system.
- Over this connection, individual SQL queries are transmitted. Note that different database systems use different protocols.
- The interaction between a relational database client and a database management system proceeds in the same way, whether the client is written in Python, R, Java, or C++.
- In this chapter we will proceed using Python and will be using the `sqlalchemy` package.

The Connection String

- For `sqlalchemy` to make a connection to a database, it needs multiple pieces of information. It must determine:
  - What protocol scheme and/or kind of database system is on the other end of the connection.
  - Which lower-level database library (driver) will be required.
  - If the database system is over a network, the network specifics for the machine and software process executing the provider-side software.

The Connection String Continued

- If the database is local, like SQLite, the file system path of the database.
- If the database system supports multiple users in the same system, the credentials for the user associated with the currently executing client application connecting to this system.
- If the database system supports multiple databases, the specific database schema to select as the default database.

Connection Strings

- For illustration, we will show the construction of connection strings for both a network example, using a MySQL provider on a non-local machine, and for a local example, using SQLite.
- For information on connection strings for other specific database systems, the reader should refer to the database/engine connection documentation.
- In `sqlalchemy`, the connection string bears an intentional resemblance to a URL.

Dictionary

- Information about a database location is changeable and may contain password information, so it is advisable to put such information into an external file.

- A JSON file with a dictionary mapping data sources to the information needed in an appropriate connection string is shown as follows:

```
{
    "mysql": {"protocol": "mysql+mysqlconnector",
              "host": "server.college.edu",
              "user": "user",
              "password": "pass",
              "database": "book"},
    "sqlite": {"protocol": "sqlite+pysqlite",
               "path": "datadir/book.db"}
}
```

Python String Building

- Building a connection string is a matter of using Python string building and incorporating the information from the dictionary.

- In the upcoming examples, we use the Python string `format()` method with a pattern string to place the values into their corresponding places.

Python's `format()` Method

- The `format()` method was introduced with Python3 for handling complex string formatting more efficiently.
- Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces `{}` into a string and calling `str.format()`.

Python's `format()` Method Examples

```
print('We all are {}.'.format('equal'))
```

Output:

```
We all are equal.
```

```
print('{2} {1} {0}'.format('directions',
                           'the', 'Read'))
```

This uses *index-based* positioning.

Output:

```
Read the directions.
```

`format()` with Assigned Keywords

```
print('a: {a}, b: {b}, c: {c}'.format(a = 1,
                                      b = 'Two',
                                      c = 12.3))
```

Output:

```
a: 1, b: Two, c: 12.3
```

```
print('The first {p} was alright, but the {p}
{p} was tough.'.format(p = 'second'))
```

Output:

```
The first second was alright, but the second
second was tough.
```

MySQL String Connection Example

```
credspath = os.path.join(datadir, "creds.json")
with open(credspath, 'r') as file:
    creds = json.load(file)

mysqlD = creds["mysql"]
pattern = "{}://{}:{}@{}/{}"
cstring = pattern.format(mysqlD["protocol"], mysqlD["user"],
                         mysqlD["password"], mysqlD["host"],
                         mysqlD["database"])
print(cstring)
```

```
| "mysql+mysqlconnector://user:pass@server.college.edu/book"
```

SQLite String Connection Example

```
credspath = os.path.join(datadir, "creds.json")
with open(credspath, 'r') as file:
    creds = json.load(file)

sqliteD = creds["sqlite"]
pattern = "{}:///{}"
cstring = pattern.format(sqliteD["protocol"],
                         sqliteD["path"])
cstring
```

```
| "sqlite+pysqlite:///datadir/book.db"
```

Connecting and Closing the Connection

- We import the `sqlalchemy` package and use an `as` clause to define a mnemonic (`sa`) to reference the functions and objects of the package.

- Online tutorials and references for the `sqlalchemy` package often use a

  ```
  from sqlalchemy import xxx, yyy
  ```

  form of import to gain access to `sqlalchemy` functions and objects without having to use the mnemonic, but while we are learning, being explicit about what package a function belongs to is preferred.

Connecting

- With most of the work being accomplished by building the connection string, establishing the connection can be realized with:

```
import sqlalchemy as sa

engine = sa.create_engine(cstring)
connection = engine.connect()
```

Closing

- A connection must be closed when its use is complete, and the engine should be deleted.

- Because a connection to a provider is consuming local resources and is consuming remote resources, with the provider using memory and execution threads for each client interaction, we need to be especially careful to do our proper cleanup.

```
try:
    connection.close()
except:
    pass
del engine
```

Using `with`

- Similar to opening and using files, we can use a `with` construct that, at the end of the `with`, will automatically perform a `close()`:

```
engine = sa.create_engine(cstring)
with engine.connect() as connection:
    # Perform database requests and process replies
    pass
del engine
```

Executing Queries and Basic Retrieval of Results

- A connection object has a method called `execute()` that transmits an SQL query request.
- The SQL query itself is composed as a Python string, and the simplest form of `execute()` has that string as its sole parameter.
- Upon successful completion, the `execute()` method returns an object known as a ***result proxy***, which can then be used in subsequent processing to obtain the rows and columns of the result, as well as additional information about the result.
- On an error, `sqlachemy` will generate a Python exception.

Basic Query

- Making a request always requires the two-step process of building the query into a Python string, and then calling the `execute()` method on the connection with the built string.
- Any valid SQL query, as covered in the previous two chapters, could make up the value of the Python query string.

Result Proxy Object

- The result proxy object provides numerous ways of obtaining the table resulting from an executed query.
- Because the architecture is, in general, a client/provider system over a network, the motivation behind this variety is to enable reduction of load on the provider and managing the quantity of data transmitted over the network at any given point in time.
- This becomes especially important when the resultant data is large.

Fetching the Data

- At completion of the `execute()` method, the provider has completed its work in parsing and optimizing the query and executing the query at the provider machine, but **the provider holds the data on the remote side until it is *fetch'd* by the client**.

- If the size of the resultant table of data is manageable, the client can request the entirety of the data from the provider through the `fetchall()` method call of the result proxy.

Basic Query Example

```
query = "SELECT *  FROM indicators0"
```

```
result_proxy = connection.execute(query)
```

```
type(result_proxy)
```

```
sqlalchemy.engine.cursor.LegacyCursorResult
```

```
result_list = result_proxy.fetchall()
print(result_list)
```

```
[('CHN', 1386.4, 12143.5, 76.4, 1469.88), ('FRA', 66.87, 2586.29, 82.
5, 69.02), ('GBR', 66.06, 2637.87, 81.2, 79.1), ('IND', 1338.66, 265
2.55, 68.8, 1168.9), ('USA', 325.15, 19485.4, 78.5, 391.6)]
```

- The result is a list of the records of the resultant table.

Example Continued

- We can use `len()` to find the number of records resulting from the query.
- We can iterate over the list and print tuple records.

```
print('Number of records returned = ', len(result_list), '\n')

for record in result_list:
    print(record)
```

```
Number of records returned =  5

('CHN', 1386.4, 12143.5, 76.4, 1469.88)
('FRA', 66.87, 2586.29, 82.5, 69.02)
('GBR', 66.06, 2637.87, 81.2, 79.1)
('IND', 1338.66, 2652.55, 68.8, 1168.9)
('USA', 325.15, 19485.4, 78.5, 391.6)
```

Additional Functionality of Tuples

```
firstrecord = result_list[2]
print(firstrecord)
```

```
('GBR', 66.06, 2637.87, 81.2, 79.1)
```

```
firstrecord['code']
```

```
'GBR'
```

```
list(firstrecord.keys())
```

```
['code', 'pop', 'gdp', 'life', 'cell']
```

```
result_proxy.keys()
```

```
RMKeyView(['code', 'pop', 'gdp', 'life', 'cell'])
```

Native Data Structures to `pandas`

- We would like to built a `pandas` data frame from the results of an SQL query.
- We begin by simply using the `DataFrame` constructor passing the result:

```
ind0 = pd.DataFrame(result_list)
ind0
```

	0	1	2	3	4
0	CHN	1386.40	12143.50	76.4	1469.88
1	FRA	66.87	2586.29	82.5	69.02
2	GBR	66.06	2637.87	81.2	79.10
3	IND	1338.66	2652.55	68.8	1168.90
4	USA	325.15	19485.40	78.5	391.60

Column Names

- The data in the table is as desired, but we want to provide `pandas` with the column/field names.
- We can use the `keys()` method of the result proxy to retrieve these as a list of strings and then pass both the data and the column names:

```
fields = result_proxy.keys()
ind0 = pd.DataFrame(result_list, columns=fields)
ind0
```

	code	pop	gdp	life	cell
0	CHN	1386.40	12143.50	76.4	1469.88
1	FRA	66.87	2586.29	82.5	69.02
2	GBR	66.06	2637.87	81.2	79.10
3	IND	1338.66	2652.55	68.8	1168.90
4	USA	325.15	19485.40	78.5	391.60

Set Index

- We can set the `code` field as a `pandas` index of row names. This is accomplished by setting the index while dropping the integer index generated by default at `DataFrame` creation:

```
ind0.set_index('code', drop=True, inplace=True)
ind0
```

```
|            pop       gdp  life     cell
| code
| CHN   1386.40  12143.50  76.4  1469.88
| FRA     66.87   2586.29  82.5    69.02
| GBR     66.06   2637.87  81.2    79.10
| IND   1338.66   2652.55  68.8  1168.90
| USA    325.15  19485.40  78.5   391.60
```

Four Step Process

1. Compose the SQL query as a Python string.
2. Execute the query.
3. Fetch all records of the result.
4. Construct a `pandas DataFrame`.

Database requests Directly Through Pandas

- The `pandas` module provides two useful functions for streamlining the four-step process.

1. If we desire an entire table, we can use the `read_sql_table()` function and the result is a Pandas data frame, compressing all four steps into one.

2. If we are not requesting an entire table, we can submit an SQL query using the `read_sql_query()` function. Here we compose the query and then invoke the function and the result is a pandas data frame.

Read Entire Table

```
ind0 = pd.read_sql_table("indicators0", con=connection)
ind0
```

	code	pop	gdp	life	cell
0	CHN	1386.40	12143.50	76.4	1469.88
1	FRA	66.87	2586.29	82.5	69.02
2	GBR	66.06	2637.87	81.2	79.10
3	IND	1338.66	2652.55	68.8	1168.90
4	USA	325.15	19485.40	78.5	391.60

SQL Query

```
query = """
SELECT code, pop, gdp
FROM indicators0
WHERE life > 78
"""
ind1 = pd.read_sql_query(query, con=connection)
ind1
```

	code	pop	gdp
0	FRA	66.87	2586.29
1	GBR	66.06	2637.87
2	USA	325.15	19485.40

- It is advisable to used triple-double quoted strings when composing queries because you can compose the SQL query over multiple lines for readability (and since SQL itself uses single quote for string literals).

SQL Query with String Literal

```
query = """
SELECT DISTINCT sex, name
FROM topnames
WHERE name LIKE 'M%'
"""
M_names = pd.read_sql_query(query, con=connection)
M_names
```

	sex	name
0	Female	Mary
1	Male	Michael

Incorporating Variables

- We now want to learn how to incorporate variables into our SQL queries.
- That is, we wish to generalize a specific query to abstract some part of it so that we can dynamically create and execute the query based on parameters.
- Specifically, we can combine Python string manipulations, such as `format()`, string `+`, and `str()` with variables to create the queries.

Python String Composition

- Suppose we often request data from the `indicators` table for a specific year and based on a life expectancy threshold.
- We wish the value of the year and the value of the threshold to come from variables.
- The generic form of this query is:

```
template = """
SELECT code, pop, gdp, life
FROM indicators
WHERE year = {} AND life > {}
"""
```

Function Approach

- We can design a function to perform this query as follows:

```
def indicators_life(dbcon, year=2017, threshold=0):
    template = """SELECT I.code, country, pop, gdp, life
                  FROM indicators AS I INNER JOIN
                       countries AS C USING (code)
                  WHERE year = {} AND life > {}"""
    query = template.format(year, threshold)

    result_proxy = dbcon.execute(query)
    data = result_proxy.fetchall()
    df = pd.DataFrame(data, columns=result_proxy.keys())
    return df
```

Function Approach

```
indicators_life(connection, year=2000, threshold=80)
```

	code	country	pop	gdp	life
0	HKG	Hong Kong SAR, China	6.66	171.67	80.9
1	JPN	Japan	126.84	4887.52	81.1
2	MAC	Macao SAR, China	0.43	6.72	80.4

Function Approach Example 2

- When creating queries in this manner, we must be aware of the data types of both the Python variables and the SQL data types of the fields used in the integrated SQL query.
- Consider the example of writing a function to obtain a subset of the `students` table based on a string range that uses the `studentlast` field.
- We want to select those records where the student last name is between an inclusive low value and an exclusive high value.

Function Approach Example 2

```
template = """
SELECT studentid,
       studentlast || ', ' || studentfirst AS studentname,
       studentmajor
FROM students
WHERE studentlast >= {} AND studentlast < {}"""
```

- Consider the result of invoking `format()`, using an example low and high as shown:

```
low = "A"
high = "Am"
query = template.format(low, high)
print(query)
```

This query would not be interpreted correctly—why?

```
|
| SELECT studentid,
|        studentlast || ', ' || studentfirst AS studentname,
|        studentmajor
| FROM students
| WHERE studentlast >= A AND studentlast < Am
```

Function Approach Example 2

- SQL would interpret the previous query as asking if field `studentlast` is greater than or equal to *field* `A`, and if field studentlast is less than *field* `Am`.
- **Without the single quotes to inform SQL that these are string literals, an unadorned sequence of characters is interpreted as a field!**

Correct Solution

```
template = """
SELECT studentid,
       studentlast || ', ' || studentfirst AS studentname,
       studentmajor
FROM students
WHERE studentlast >= '{}' AND studentlast < '{}'
"""

low = "A"
high = "Am"
query = template.format(low, high)
print(query)
```

Now we include single quotes with the template string

```
|
| SELECT studentid,
|        studentlast || ', ' || studentfirst AS studentname,
|        studentmajor
| FROM students
| WHERE studentlast >= 'A' AND studentlast < 'Am'
```

Binding Variables

- `sqlalchemy` provide mechanisms to make incorporating variable values easier and more efficient that the previous string building examples. The general steps include:

1. Construct an object that has a combination of SQL syntax and elements designating the places where the value of a variable should be substituted. This is called the ***prepare step***.

2. When the variable values are available, a ***bind operation*** is performed, that associates a set of values with their corresponding locations in the prepared SQL text. This step constructs a new object to obtain a ***bound statement***.

3. The bound statement is then conveyed as a request to the database.

`sql` Class of `sqlalchemy`

- Within the `sql` class of `sqlalchemy` is a `text()` constructor that builds a prepared object, consistent with step 1 above.

```
import sqlalchemy as sa

pyquery = """SELECT I.code, country, pop, gdp, life
                  FROM indicators AS I INNER JOIN
                       countries AS C USING (code)
                  WHERE year = :yr AND life > :threshold"""
prepare_stmt = sa.sql.text(pyquery)
```

- The `pyquery` variable references a Python string that uses the syntax for binding, which are designated with a colon (`:`) immediately followed by a programmer selected name (`yr` and `threshold`) that should obey Python variable naming syntax.

- This SQL-specific template is then prepared by calling `text()` and assigning to the `prepare_stmt` variable for later reference.

Bind Step

- The bind step invokes the `bindparams()` method of a prepared statement yielding a *bound statement*.
- The `bindparams()` method associates values by simply using named parameters as its arguments.
- In this case, the named parameters are `yr` and `threshold`, and creation of the bound object is the single step:

```
bound_stmt = prepare_stmt.bindparams(yr = 2000, threshold = 80)
```

Execute Step

- The execute step is performed, as before, but using the bound statement instead of a string for a query.
- We can use the `execute()` method and `fetchall()` to obtain the results, or we can use the `read_sql_query()` function of `pandas` to combine those steps and return a data frame:

```
df = pd.read_sql_query(bound_stmt, connection)
df
```

	code	country	pop	gdp	life
0	HKG	Hong Kong SAR, China	6.66	171.67	80.9
1	JPN	Japan	126.84	4887.52	81.1
2	MAC	Macao SAR, China	0.43	6.72	80.4

Named Parameters

- The `execute()` method, if given named parameters, will combine the bind and execute steps on our behalf, allowing us to skip an explicit bind:

```
result_proxy = connection.execute(prepare_stmt, yr=1999,
                                  threshold=70)
data = result_proxy.fetchall()
print('Number of records returned:', len(data))
```

```
| Number of records returned: 96
```

Advantages?

- Two of the advantages of using SQL variable binding over string-based composition of queries are <u>reusability</u> and the <u>carrying of type information</u>.

- Variable binding in our SQL queries can also give us a clean way to build functional abstractions that nicely generalize operations we wish to perform against a database.

Example

```
def students_byname ( dbcon, name_min, name_max ):
    pyquery = """SELECT studentid, studentlast || ', '
                        || studentfirst AS studentname,
                  studentmajor
                  FROM students
                  WHERE studentlast >= :low AND
                        studentlast <  :high
                  ORDER BY studentname"""
    prepare_stmt = sa.sql.text(pyquery)

    bound_stmt = prepare_stmt.bindparams(low=name_min,
                                         high=name_max)
    df = pd.read_sql_query(bound_stmt, con=dbcon)
    return df
```

Example Continued

```
students_byname(connection, "E", "Eh")
```

	studentid	studentname	studentmajor
0	62787	Edwards, Alisha	BIOL
1	63723	Edwards, Billy	None
2	63019	Edwards, Carolyn	BCHM
3	62222	Edwards, Eloise	ECON
4	61659	Edwards, Eva	FREN
5	62940	Edwards, Gary	MATH
6	62094	Edwards, Nancy	HIST
7	62265	Edwards, Stephanie	ARTS

More Advanced Techniques

- As the number of rows retrieved in a result from a relational database system grows larger, we need greater control over how we retrieve the data itself.
- This control is one of the reasons that, on a query, the object returned is a result proxy.
- The result proxy is a relatively small object that can be transmitted from the provider to the client application, and then, using the proxy, we can fetch rows of the result.
- **In this way, the application can choose to retrieve subsets of the data, down to processing the result a record at a time**.

Example Query

- To illustrate the different techniques for obtaining records we will use the following query:

```
query="""
SELECT departmentid AS DeptID, departmentname AS Name,
       division AS Division
FROM departments
WHERE division = 'Fine Arts'
"""
```

Result Proxy as an Iterator

- One alternative for getting the data a record at a time is through direct iteration with the result proxy. A result proxy can act as an iterator, and so we can use it in a for loop and obtain each of the records in the result, one at a time:

```
result_proxy = connection.execute(query)

for record in result_proxy:
    print(record)
```

```
| ('ART', 'Art History and Visual Culture', 'Fine Arts')
| ('CINE', 'Cinema', 'Fine Arts')
| ('DANC', 'Dance', 'Fine Arts')
| ('MUS', 'Music', 'Fine Arts')
| ('THTR', 'Theatre', 'Fine Arts')
```

Result Proxy as an Iterator

- Alternatively:

```
for record in result_proxy:
    print(record['DeptID'], record['Division'])
```

```
| ART Fine Arts
| CINE Fine Arts
| DANC Fine Arts
| MUS Fine Arts
| THTR Fine Arts
```

Fetch One

- A result proxy also has a method named `fetchone()` that obtains the next record, keeping track of a current position in the set of result records.

```
row = result_proxy.fetchone()
while (row):

    print(row)
    # Process a single row

    row = result_proxy.fetchone()
```

- Notice that the result, `row`, evaluates to `False` when there are no more results to be processed, and allows termination of the while loop.

Chunks

- Processing records one at a time involves significantly more network processing than fetching the entire result.
- We may want a compromise between fetching all the records, which for some tables may be too large as a single result, and fetching one at a time, which add network latency and delay for each record.
- Logically, we want the control to specify a ***chunk***, which is an application-specified number of records to retrieve at a time.

Fetch Many

- The result proxy has a `fetchmany()` method that provides the capability of specifying the maximum number of records to retrieve in one operation.
- The method takes a single argument of an integer number of records to attempt to fetch.
- To use a *chunk size*, we need our processing to operate at an outer level—where at each iteration we obtain a chunk-sized collection of result records, and an inner level—where at each iteration, we process the records within the chunk.

Chunk Example

- Consider the following example, where we process with a chunk size of (at most) two records per fetch.
- The outer loop is an indefinite loop that terminates when a `fetchmany()` returns a `None` result (no more chunks).
- The inner loop uses the chunk returned by `fetchmany()` as an iterator that, on each iteration, yields a single record of the chunk result.

Chunk Example

```
chunk_size = 2
rowset = result_proxy.fetchmany(chunk_size)
while (rowset):
    print("Next chunk @ length", len(rowset))
    for row in rowset:
        print(row)
        # Process a single row

    rowset = result_proxy.fetchmany(chunk_size)
```

```
| Next chunk @ length 2
| ('ART', 'Art History and Visual Culture', 'Fine Arts')
| ('CINE', 'Cinema', 'Fine Arts')
| Next chunk @ length 2
| ('DANC', 'Dance', 'Fine Arts')
| ('MUS', 'Music', 'Fine Arts')
| Next chunk @ length 1
| ('THTR', 'Theatre', 'Fine Arts')
```

Using Pandas with Chunk Size

- The Pandas `read_sql_query()` method has a named parameter, `chunksize=`, that we can use to control the number of results fetched at a time.
- The method returns an iterator.
- This iterator yields a complete data frame on each next item iteration of the for loop.

Pandas Chunk Example

```
chunk_size = 3
iterator = pd.read_sql_query(query, con=connection,
                             chunksize=chunk_size)
for df in iterator:
    print(type(df))
    print(str(df))
```

```
| <class 'pandas.core.frame.DataFrame'>
|        name     sex
| 0      John    Male
| 1     James    Male
| 2  Jennifer  Female
| <class 'pandas.core.frame.DataFrame'>
|       name     sex
| 0  Jessica  Female
| 1    Jacob    Male
```

Unified Data Frame Example

- In the example below, we use the chunk ability of `read_sql_query()`, but combine the data frames into a single table.

```
chunk_size = 3
iterator = pd.read_sql_query(query, con=connection,
                             chunksize=chunk_size)
```

```
try:
    result_df = pd.DataFrame()
    while (True):
        df = next(iterator)
        print(type(df))
        print(str(df))
        print()
        result_df = pd.concat([result_df, df], axis=0,
                              ignore_index=True)
except Exception as e:
    pass
```

Working with Two Databases

- Sometimes the need arises for a client to work with multiple databases within a single application. There are two options:

Option 1: Explicitly create two engine objects and two connection objects, using different Python variable names. Then each connection `execute()` or the connection passed to Pandas would need to decide which connection to use for which query.

Option 2: Some DBMSs support multiple database schema served by the same provider (MySQL, for instance).

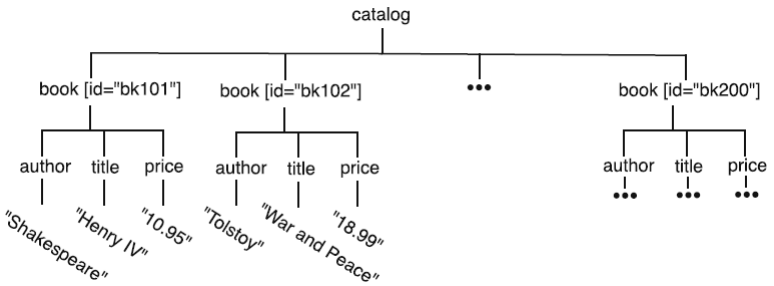
  – You can check the documentation of your DBMS for more details.

Chapter 15

Hierarchical Model: Structure and Formats

Hierarchical Database?

- A ***hierarchical database*** is a data model in which data is stored in the form of records and organized into a *tree-like structure*, or parent-child structure, in which one parent node can have many child nodes connected through links.



When is it Appropriate?

- The hierarchical database model is most appropriate when the information is based on a concrete hierarchy, such as several individual employees reporting to a single department at a company.

Hierarchical Database Schema

- The schema for hierarchical databases is defined by its tree-like organization.
  - There is typically a root "parent" directory of data stored as records that links to various other subdirectory branches, and each subdirectory branch, or child record, may link to various other subdirectory branches.
- The hierarchical database structure dictates that, while a parent record can have several child records, each child record can only have one parent record.
- Data within records is stored in the form of fields, and each field can only contain one value.
- Retrieving hierarchical data from a hierarchical database architecture requires traversing the entire tree, starting at the root node.

Difference?

- The difference between relational and hierarchical databases lies in the data structures. While the hierarchical database architecture is tree-like, data in a relational database is stored in tables with a unique identifier for each record.
- A relational database structure facilitates easy identification and access of data in relation to other data points in the database.
- Hierarchical database models are simply, but lack of flexibility.
- Unlike relational databases, hierarchical structures do not describe many-to-one relationships or many-to-many relationships due to the fact that child records can only have a single parent.

Advantages

- The key advantage of a hierarchical database is its ease of use.
- The one-to-many organization of data makes traversing the database simple and fast, which is ideal for use cases such as website drop-down menus or folder structures of an operating system.
- Information can easily be added or deleted without affecting the entirety of the database.
- Most major programming languages offer functionality for reading tree structure databases.

Disadvantages

- The major disadvantage of hierarchical databases is their inflexible nature.
- The one-to-many structure is not ideal for complex structures as it cannot describe relationships in which each child node has multiple parent nodes.
- Also, the tree-like organization of data requires top-to-bottom sequential searching, which is time consuming, and requires repetitive storage of data in multiple different entities, which can be redundant.

Motivation

- Data obtained from web scraping and from APIs is often naturally hierarchical.
- Many topics in statistics can be viewed more clearly through a hierarchical lens, including time series analysis, clustering, multiple linear regression, ANOVA, longitudinal data analysis, etc.
- We will limit our discussion to local files with JSON and XML formats, but you are encouraged to remember that these files can store data from a wide variety of application areas.

Comparison

- The relational model is the most structured (constraints are built-in), followed by the tabular model (constraints are "by convention" to achieve tidy data).
- We will see that **the hierarchical model is the least structured of the three**. This will create added complexity when we engage our discussion of hierarchical constraints.

Benefits

- The hierarchical model has tremendous expressive power.
- Anything that can be expressed in tabular form can also be expressed in hierarchical form (as we will see shortly).
- Hierarchical models are excellent for storing large and complex data sets in such a way as to make the data efficiently searchable.
  - We have already seen this with our example of the file-structure, where it is possible to get from the root directory to each file in a small number of steps.
- It is much easier to add data in the hierarchical model (anywhere in the tree) than it is to add data to the tabular or relational models.
  - This also makes it easier to including missing data in our model.
  - **Instead of requiring a special symbol like NA, we can represent missing data simply by the absence of a leaf in the tree**.

Graph Definition

- As mentioned earlier, hierarchical data refers to data where the underlying structure is that of a tree.
- A ***graph*** is a set V of ***nodes*** (***vertices***), and a set $E \subseteq V \times V$ of ***edges*** (***arcs***) between nodes.
- It is helpful to think of an edge $v_1 v_2$ as signifying a *relationship* between nodes v_1 and v_2, and when such an edge is present we say v_1 and v_2 are ***adjacent***.
- A ***path*** is a sequence of edges $v_1 v_2 \ldots v_k$ from some initial node v_1 (source) to some node v_k (destination).
- If $v_k = v_1$, the path is called a ***cycle***.

Example

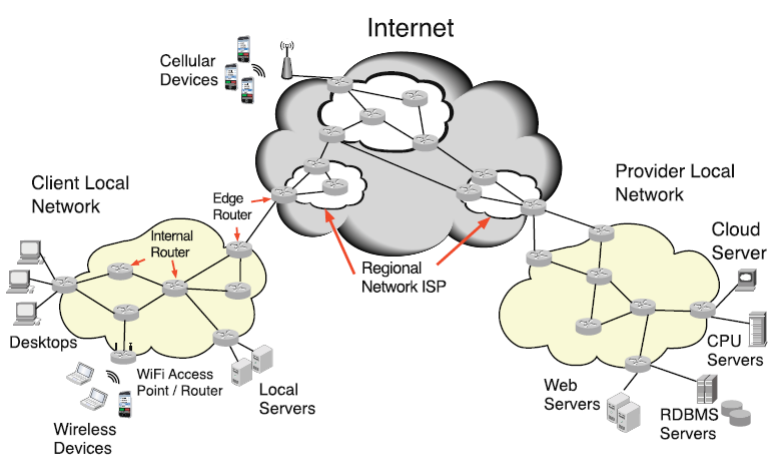


Fig. 15.1 Network architecture

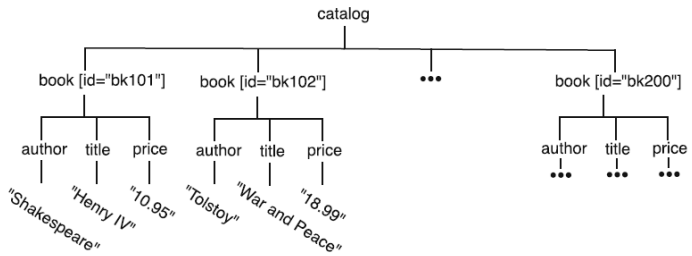
Graph Definitions Continued

- We say a graph *G* is ***connected*** if, for any two nodes in *G*, there exists at least one path between them.
- A graph that is connected and has no cycles is called a ***tree***.
- A graph with no cycles is called a ***forest***. In this way, a forest is made up of one or more disconnected trees.

Number of Edges

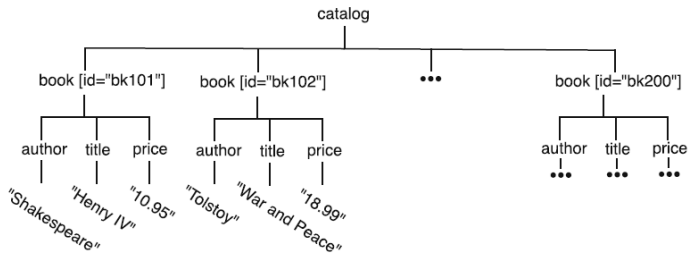
- It is not difficult to show that if a *tree* has n nodes, then it must have $n-1$ edges. With any fewer edges, the tree would not be connected.
- With any more edges, it would have a cycle.

More Terminology



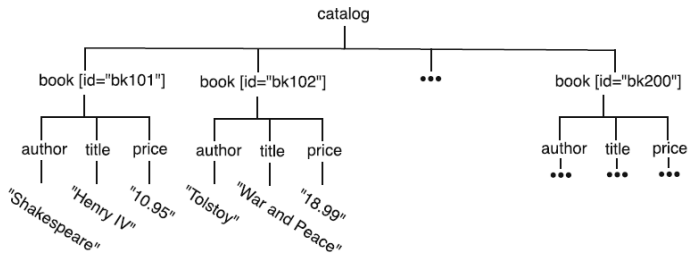
- The node at the top is called the ***root***.
- For a given node, the nodes directly below it are called its ***children***.
- For a given node, the node directly above it is called its ***parent***.
- For a given node *X*, the nodes along the path from the root to *X* are the ***ancestors*** of *X*. Nodes below *X*, whose paths from the root contain *X*, are the ***descendants*** of *X*.

More Terminology



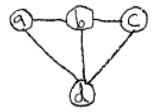
- If two nodes have the same parent, we call them ***siblings***.
- We allow nodes to have a symbolic name, sometimes called a *tag* or *label*.
  - We allow two different nodes to have the same label (e.g., there are 100 nodes labeled author), and we use their location in the tree to distinguish them.
- A node with no children is called a ***leaf*** node.

More Terminology



- We note that there is a unique path between any two nodes in a given tree.
- The length of this unique path (that is, the number of edges in the path) is called the ***distance*** between the two nodes.
- In figure above, all leaves are <u>*distance three*</u> from the root. In such a case, we say the tree has ***depth*** three.

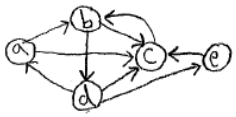
Simple Graph Examples



Undirected Graph G1

$V = \{a, b, c, d\}$

$E = \{(a,b), (a,d), (b,c), (b,d), (c,d)\}$



Directed Graph G2

$V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (b,c), (b,d), (c,b), (d,a), (d,c), (d,e), (e,c)\}$

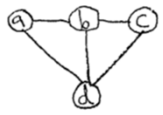
Graph Representation

- Graphs for computer algorithms are typically represented in one of two ways: the *adjacency matrix* and *adjacency lists*.

Def: Given a graph with *n* nodes, an ***adjacency matrix*** is an $n \times n$ Boolean matrix with one row and one column for each of the graph's nodes, in which the element in the *i*th row and *j*th column is equal to 1 if there is an edge from the *i*th node to the *j*th node, and equal to 0 if there is no such edge.

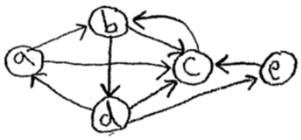
Def: The ***adjacency list*** of a graph is a collection of *linked lists*, one for each node, that contain all the nodes adjacent to the list's node.

Adjacency Matrix Examples



Undirected Graph G1

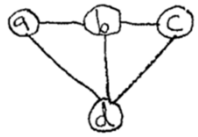
$$G1: \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{array}$$



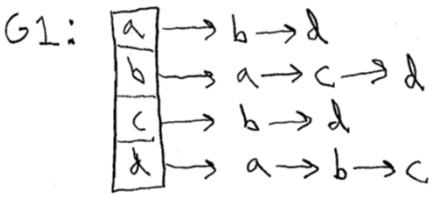
Directed Graph G2

$$G2: \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 1 & 0 \\ c & 0 & 1 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 & 1 \\ e & 0 & 0 & 1 & 0 & 0 \end{array}$$

Adjacency List Example



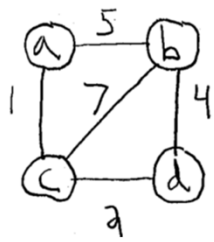
Undirected Graph G1



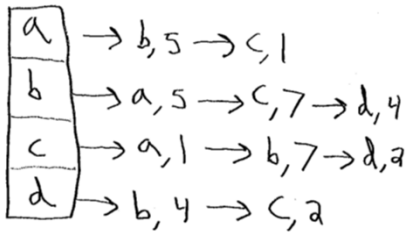
Weighted Graph

- A ***weighted graph*** is a graph with numbers assigned to its edges. These numbers are called *weights* or *costs*.
- We can easily modify our principal representation of a graph to handle weights.

Weighted Graph Example

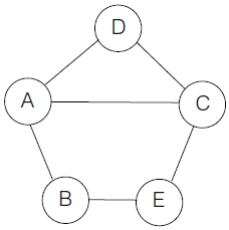


$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & \infty & 5 & 1 & \infty \\ b & 5 & \infty & 7 & 4 \\ c & 1 & 7 & \infty & 2 \\ d & \infty & 4 & 2 & \infty \end{array}$$



Representation in Python?

- You could represent an adjacency matrix using a 2D numPy array:

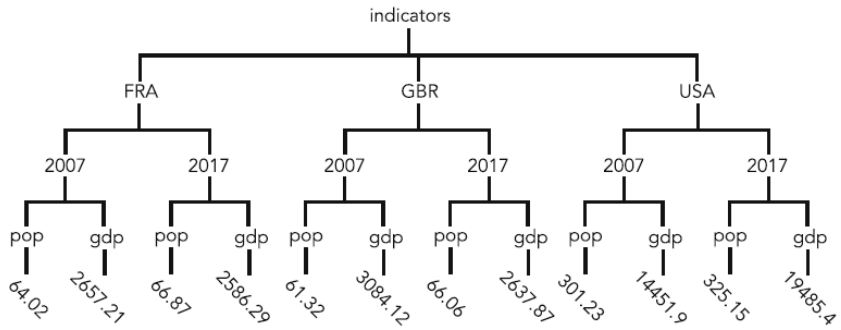



```
adjMatrix = [
  [0,1,1,1,0],
  [1,0,0,0,1],
  [1,0,0,1,1],
  [1,0,1,0,0],
  [0,1,1,0,0]
]
```

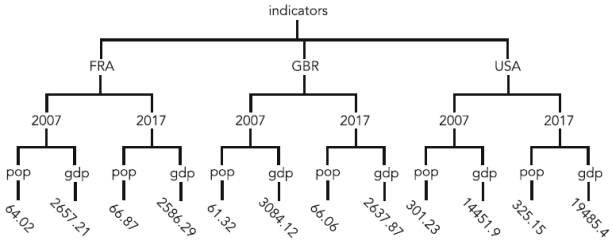
- Linked lists are typically taught in a data structures class (which we will not cover here).

Representing Trees

- Here is a tree representation of the data set `ind0`:




```
indicatorsDict = {
  "FRA": {
    "2007": {
      "pop": 64.02,
      "gdp": 2657.21},
    "2017": {
      "pop": 66.87,
      "gdp": 2586.29}
      },
    "GBR": {
      "2007": {
        "pop": 61.32,
        "gdp": 3084.12},
      "2017": {
        "pop": 66.06,
        "gdp": 2637.87}
      },
    "USA": {
      "2007": {
        "pop": 301.23,
        "gdp": 14451.9},
      "2017": {
        "pop": 325.15,
        "gdp": 19485.4}
      }
    }
```



- We can represent `ind0` as dictionary whose name corresponds to the root note (`indicators`) and whose keys correspond to the children of the root node (here, “FRA,” “GBR,” and “USA”).

- When we use dictionaries as the primary structure used in the nesting to achieve a tree, we use the ***Dictionary of Dictionaries*** (DoD) term introduced earlier in the course.

Trees as Lists

- We can also represent a tree as a Python list, again using the root as the name of the list, and children as the elements in the list.
- When we use lists as the primary structure used in the nesting to achieve a tree, we will use the same ***List of Lists*** (LoL) term introduced earlier in the course.
- One benefit of the list representation is that we do not need explicit names of nodes as we gave above, but instead use integer indices into the lists.

Example

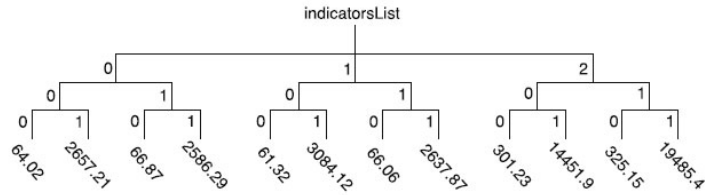
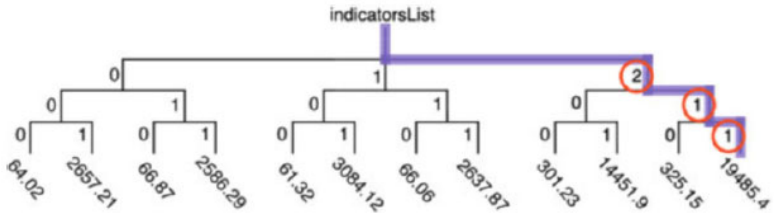


Fig. 15.5 Nested lists as a tree

```
indicatorsList = [[[64.02,2657.21],[66.87,2586.29]],
                  [[61.32,3084.12],[6606,2637.87]],
                  [[301.23,14451.9],[325.15,19485.4]]]
```

Traversals and Paths

- With either of these representations of trees through nested Python data structures, it is easy to build *paths* (*traversals*) in the tree.



JSON

- As discussed earlier in the course, the ***JavaScript Object Notation*** (JSON) format is a programming language independent way to move between data stored in a text file, and data stored in a data structure.
- This makes JSON a very convenient format for hierarchical data.
- It is important to note that the only scalars allowed for JSON are Booleans, numbers, strings, and a special data type called *null*.
- Thus, a JSON file cannot store a tree of file objects, but can store a tree of strings (including file names).
- Furthermore, strings must be represented by double quotes, instead of single quotes, e.g., “Chris” is an acceptable string, but ‘Lane’ is not.

JSON Specifications

- In JSON, when data is stored inside square brackets, in an ordered list, such as `[1,"one",True]`, the data type is known as an ***array***.

- When data is stored inside curly brackets, with `key:value` pairs, like `{"Noah":2, "Becky":1, "Evan":1}`, the data type is known as an ***object***.

JSON Dump

- If we have the data structures `indicatorsDict` and `indicatorsList` (defined on earlier slides) in a Python program, we can write them to a JSON file using the `dump()` function:

```
import json
pathD = os.path.join(datadir, "ind0Dict.json")
pathL = os.path.join(datadir, "ind0List.json")

f = open(pathD, 'w')
json.dump(indicatorsDict, f)
f.close()

g = open(pathL, 'w')
json.dump(indicatorsList, g)
g.close()
```

This creates two files, `indoDict.json` and `indoList.json` in the data directory.

JSON Formatted String

- Sometimes it is advantageous to have a JSON-formatted string. We demonstrate for `indicatorsDict`:

```
import json
mystring = json.dumps(indicatorsDict)
```

- The resulting string, `mystring`, could easily be passed as a parameter to a function.

```
{"FRA": {"2007": {"pop": 64.02, "gdp": 2657.21},
         "2017": {"pop": 66.87, "gdp": 2586.29}}, ...}
```

Load JSON File

- In addition to dumping Python data structures into JSON files, we can also load a data structure from a JSON file.

```
import json
pathD = os.path.join(datadir, "ind0Dict.json")

f = open(pathD, 'r')
indicatorsDict = json.load(f)
f.close()
```

XML

- More powerful and widespread than the JSON format, the ***eXtensible Markup Language*** (XML) is a general structure for modeling hierarchical data.
- XML is a markup language much like HTML.
- XML was designed to store and transport data.
- XML was designed to be self-descriptive.

XML Continued

- Despite the word “language” appearing in its name, XML does not represent a programming language.
- When given an XML file, there is no code to execute—an XML file doesn’t *do* anything.
- Instead, it is a language for communicating hierarchical data with great power and generality, with certain grammatical rules that we will discuss shortly.

BreakfastMenu.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
<food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>
   Two of our famous Belgian Waffles with plenty of real maple syrup
   </description>
    <calories>650</calories>
</food>
<food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>
    Light Belgian waffles covered with strawberries and whipped cream
    </description>
    <calories>900</calories>
</food>
<food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>
    Belgian waffles covered with assorted fresh berries and whipped cream
    </description>
    <calories>900</calories>
</food>
<food>
    <name>French Toast</name>
    <price>$4.50</price>
    <description>
    Thick slices made from our homemade sourdough bread
    </description>
    <calories>600</calories>
</food>
<food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>
    Two eggs, bacon or sausage, toast, and our ever-popular hash browns
    </description>
    <calories>950</calories>
</food>
</breakfast_menu>
```

- Here is a XSLT (Extensible Stylesheet Language Transformations) representation of this file:

Belgian Waffles - $5.95

Two of our famous Belgian Waffles with plenty of real maple syrup *(650 calories per serving)*

Strawberry Belgian Waffles - $7.95

Light Belgian waffles covered with strawberries and whipped cream *(900 calories per serving)*

Berry-Berry Belgian Waffles - $8.95

Light Belgian waffles covered with an assortment of fresh berries and whipped cream *(900 calories per serving)*

French Toast - $4.50

Thick slices made from our homemade sourdough bread *(600 calories per serving)*

Homestyle Breakfast - $6.95

Two eggs, bacon or sausage, toast, and our ever-popular hash browns *(950 calories per serving)*

XML Continued

- XML data is stored as plain text, so an XML file can be opened with any text editor (ideally, one that is “syntax aware”).
- Like JSON, XML is designed to be programming language agnostic, that is, to work with any application that seeks to use it.
- Because XML data is simple text data, it is easy to communicate between operating systems, data systems, applications, etc.
- In web scraping, one often comes across XML pages as well as HTML.
- XML files are also commonly used as a format for data hosted by the US Government.

Difference Between XML and HTML

- XML and HTML were designed with different goals:
  - XML was designed to carry data—with focus on what data is.
  - HTML was designed to display data—with focus on how data looks.
  - XML tags are not predefined like HTML tags are.

XML Does Not Use Predefined Tags

- HTML works with predefined tags like `<p>`, `<h1>`, `<table>`, etc.
- The XML language has no predefined tags.
- These tags are "invented" by the author of the XML document and are analogous to column headings in the tabular model.
- With XML, the author must define both the tags and the document structure.

XML Simplifies Things

- Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.
- XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.
- XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Separates Data from Presentation

- XML does not carry any information about how to be displayed.
- The same XML data can be used in many different presentation scenarios.
- In many HTML applications, XML is used to store or transport data, while HTML is used to format and display the same data.

Transaction Data

- Thousands of XML formats exist, in many different industries, to describe day-to-day data transactions:
  - Stocks and Shares
  - Financial transactions
  - Medical data
  - Mathematical data
  - Scientific measurements
  - News information
  - Weather services

- Below is an XML file from the NOAA's National Weather Service:

weather.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<current_observation>

<credit>NOAA's National Weather Service</credit>
<credit_URL>http://weather.gov/</credit_URL>

<image>
  <url>http://weather.gov/images/xml_logo.gif</url>
  <title>NOAA's National Weather Service</title>
  <link>http://weather.gov</link>
</image>

<location>New York/John F. Kennedy Intl Airport, NY</location>
<station_id>KJFK</station_id>
<latitude>40.66</latitude>
<longitude>-73.78</longitude>
<observation_time_rfc822>Mon, 11 Feb 2008 06:51:00 -0500 EST
</observation_time_rfc822>

<weather>A Few Clouds</weather>
<temp_f>11</temp_f>
<temp_c>-12</temp_c>
<relative_humidity>36</relative_humidity>
<wind_dir>West</wind_dir>
<wind_degrees>280</wind_degrees>
<wind_mph>18.4</wind_mph>
<wind_gust_mph>29</wind_gust_mph>
<pressure_mb>1023.6</pressure_mb>
<pressure_in>30.23</pressure_in>
<dewpoint_f>-11</dewpoint_f>
<dewpoint_c>-24</dewpoint_c>
<windchill_f>-7</windchill_f>
<windchill_c>-22</windchill_c>
<visibility_mi>10.00</visibility_mi>

<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base>
<icon_url_name>nfew.jpg</icon_url_name>
<disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
<copyright_url>http://weather.gov/disclaimer.html</copyright_url>

</current_observation>
```

XML Structure

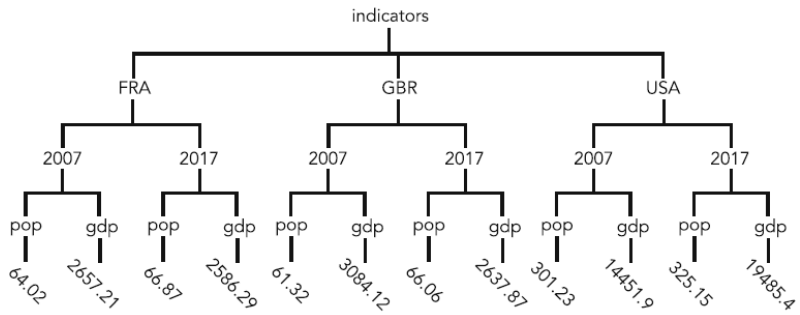
- An XML document represents a tree.
- As a tree, an XML document must have exactly one root.
- There are several types of nodes in an XML tree, including *text* nodes, *attribute* nodes, and *internal* nodes.
- Each internal node represents an XML ***element***, which starts with a ***start tag*** and ends with the corresponding ***end tag***.
- All data between the start tag and end tag represents data that is structurally subordinate to the node in question.

XML Structure Continued

- To validly represent a tree, an XML document must properly nest these start and end tags.
- Basic XML consists of three things:
  - Tagged elements, which may be nested within one another.
  - Attributes on elements, representing metadata associated with the element.
  - Text, often representing essential data.

ind0 table

Code	Year	Pop	Gdp
FRA	2007	64.02	2657.21
GBR	2007	61.32	3084.12
USA	2007	301.23	14451.90
FRA	2017	66.87	2586.29
GBR	2017	66.06	2637.87
USA	2017	325.15	19485.40



- Below is the XML representation of the `ind0` table. The tree structure is indicated both by start and end tags and via the horizontal white space on each line (to clearly display the nesting structure).
- Each country begins and ends with a tag.

ind0.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<indicators>
  <country code="FRA" name="France">
    <timedata year="2007">
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
    </timedata>
  </country>
  <country code="GBR" name="United Kingdom">
    <timedata year="2007">
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </timedata>
  </country>
  <country code="USA" name="United States">
    <timedata year="2007">
      <pop>301.23</pop>
      <gdp>14451.9</gdp>
    </timedata>
    <timedata year="2017">
      <pop>325.15</pop>
      <gdp>19485.4</gdp>
    </timedata>
  </country>
</indicators>
```

The first line is an optional XML prolog and not part of the XML tree.

`indicators` is the root of the tree.

These are *attributes* of the `country` tag (where attributes are separated by a space).

We can see that ***all*** the data associated with the United Kingdom is associated between these lines—this is an *element* of the tree.

We can see that the data associated with the year 2007 of the United States is associated between these lines.

- The person creating the XML data decides whether data appears as an *attribute* or as an *element*. For example, in `indicators`, instead of the element

```
<timedata year="2007">
  <pop>64.02</pop>
  <gdp>2657.21</gdp>
</timedata>
```

  we could provide the same data in the form

```
<timedata>
  <year>2007</year>
  <pop>64.02</pop>
  <gdp>2657.21</gdp>
</timedata>
```

- It is always possible to translate an XML that uses attributes into one that does not by making each attribute a child of the element in question.

- This decision is made based on whether year is thought of as metadata or not.

- XML files are required to satisfy a "grammar" with rules such as
  - there must be exactly one root element
  - every start-tag must have a corresponding end-tag
  - elements must nest properly" (e.g., in `indicators`, we could not close a `country` element before closing one of its `timedata` children).
  - Tag names cannot contain whitespace, colons, or the sequence `xml`, and attribute values cannot use the symbols `&`, <, or >, to avoid confusion with the symbols reserved for starting and ending tags.

- When an XML file satisfies the grammatical rules, we call it ***well-formed***.

- When an XML file is well-formed, it can be read in an automated way, the relationships between elements (i.e., the edge structure) can be mapped, and the hierarchical data represented can be drawn as a unique tree. This automated process is known as ***parsing*** (which we will explore later).

- You should be aware that there are XML files that omit end tags, by using a special syntax for start tags. Specifically, if a start tag ends with the symbol /, this means no end tag will appear.

- However, we will always include an end tag associated with every start tag in this course.

Chapter 16

Hierarchical Model: Operations and Programming

Goals of Chapter

- Extract hierarchical data from a JSON object into a Pandas data frame (when possible).
- Add data to a JSON-formatted hierarchical structure, and update data that is already present, by specifying a path within the tree.
- Extract hierarchical data from an XML file into a Pandas data frame (when possible).
- Create an XML file from a tree stored as a JSON object.
- Create and structure queries to get desired data out of a tree (i.e., XPath).

Operations Overview

- The basic operations when our data is stored in a hierarchical/tree structure are:
  - Access operations
  - Traversal operations
  - Update operations
- In data analytics, the work of understanding and transforming between different forms of data is known as ***data wrangling***.

Section 16.2: JSON Procedural Programming

- In this section, we illustrate how to extract (wrangle) data from a JSON-based data structure into a Pandas data frame.
- Note that not every tree can be put into tabular form in a meaningful way.
- **The design of each JSON-based tree of data can be different, therefore Pandas cannot automatically help us like it could with CSV-formatted files or in obtaining data from SQL queries.**

Processing Plan

- In the Jupyter notebook you will soon work through, we will utilize the following processing plan.

1. Wrangle the data from the tree into a list of dictionaries (LoD), where
   - The items in the list represent rows
   - The dictionary maps column names to field names

1. The LoD data structure can be passed to the `DataFrame()` constructor to obtain a Pandas data frame.

Typical Procedural Operations

- Given a node in the JSON-based DoD structure, traverse over its children.
  - For a Python dictionary `D`, `D.items()` yields an iterator of tuples `T` where `T[0]` is a key in `D` and `T[1]` is the associated value.
  - We iterate over `D.items()` and extract the data one row at a time.

- Restructure a DoD tree of depth=2 into a LoD or LoL to pass to Pandas.
  - When a leaf is a dictionary, use the dictionary access operator to get the value associated with a key.
  - To build the rows of a list of lists, we sometimes have to access individual row dictionary elements.

Problem Solving Strategy Takeaways

- We first need to understand the structure of the JSON-based data and to visualize the table or tables that are desired.
- We need to have a clear picture of the native Python data structure that we are processing into that will allow a data frame creation.
- On the positive side, with regards to this processing, it is easy to traverse over the lists and dictionaries from JSON-based data.

Activity 05 - Introduction to JSON and XML

DATA 200 - Database Systems and Data Management for Data Analytics
Dickinson College

In [1]:

```
## RUN THIS CELL TO GET THE RIGHT FORMATTING AND TO LOAD NumPy
import requests
import numpy as np
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/maste
HTML(styles)
```

Out[1]:

This notebook will cover Sections 15.4.2 (Extracting Data from an XML File) and 16.2 (JSON Procedural Programming).

Make sure that the `jsonXMLdata` folder is in the same working directory as this notebook.

Section 15.4.2: Extracting Data from an XML File

We can use the Python library `lxml` to extract data from an XML file. This abstracts the process of parsing the file to map out the tree structure and allows us to access individual elements in the tree.

Run the code cell below to import the needed libraries and to set the directory for our data sets.

In [2]:

```
import os.path
from lxml import etree
import json
import pandas as pd

datadir="jsonXMLdata"
```

The name `etree` stands for "Element tree", and the `etree` module is designed to parse a given XML file into the associated Element tree.

In the code below, we access the `ind0.xml` file, which contains the `ind0` table (from the `book.db` database) stored as an XML file. Specifically:

- `pathInd` = the path to the XML file

- `stripparser` is a custom parser that removes blank text nodes
- Using the `parse()` function of `etree`, we obtain the Element tree, which we assign to `indTree`
- Finally, we use the `getroot()` method of Element tree to obtain the root, which we assign to `indroot`.

Study and run the code cell below.

In [3]:
```
pathInd = os.path.join(datadir, "ind0.xml")

stripparser = etree.XMLParser(remove_blank_text=True)
indtree = etree.parse(pathInd, stripparser)
indroot = indtree.getroot()
```

Once we have extracted the root, we can look at summary information about it.

Study and run the code cell below.

In [4]:
```
print("Type:", type(indroot), "Length:", len(indroot))
```

```
Type: <class 'lxml.etree._Element'> Length: 3
```

Thus, the type of `indroot` is `Element`, and the length of an element is the number of children it has (in this case there are three children).

We can also extract the *tag* associated with an element, or its *attributes* (which are empty for the root of indicators).

Study and run the code cell below.

In [5]:
```
print("Root Tag:", indroot.tag)
print("Root Element Attributes", indroot.attrib)
```

```
Root Tag: indicators
Root Element Attributes {}
```

It is possible to print the entire tree, and doing so we recover the XML structure.

Note that in the code below, the option `pretty_print=True` guarantees that we display one start tag per line, and use horizontal whitespace to display the nesting structure.

Study and run the code cell below.

In [6]:
```
s = etree.tostring(indroot, pretty_print=True).decode('utf-8')
print(s)
```

```
<indicators>
  <country code="FRA" name="France">
    <timedata year="2007">
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </timedata>
    <timedata year="2017">
```

```
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
    </timedata>
  </country>
  <country code="GBR" name="United Kingdom">
    <timedata year="2007">
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </timedata>
  </country>
  <country code="USA" name="United States">
    <timedata year="2007">
      <pop>301.23</pop>
      <gdp>14451.9</gdp>
    </timedata>
    <timedata year="2017">
      <pop>325.15</pop>
      <gdp>19485.4</gdp>
    </timedata>
  </country>
</indicators>
```

The `find()` function will locate children with a given tag, one at a time.

This function returns the first Element found with the given tag, searching from top to bottom in the XML file (equivalently, searching the children from left to right in the associated tree).

Study and run the code cell below.

In [7]:
```
findCountry = indroot.find('country')
findCountry.attrib
```

Out[7]:
```
{'code': 'FRA', 'name': 'France'}
```

Note that France was found first as expected.

We can locate the other children with subsequent calls to `getnext()`.

Study and run the code cell below.

In [8]:
```
nextCountry = findCountry.getnext()
nextCountry.attrib
```

Out[8]:
```
{'code': 'GBR', 'name': 'United Kingdom'}
```

Note that United Kingdom was found second (as it is to the right of France in the tree).

We can locate the next child with another call to `getnext()`.

Study and run the code cell below.

In [9]:
```
next_nextCountry = nextCountry.getnext()
next_nextCountry.attrib
```

Out[9]:
```
{'code': 'USA', 'name': 'United States'}
```

The United States was the next country located as it is to the right of the United Kingdom note.

We can also use the `findall()` function to locate *all* children. This function returns a list of Elements.

Study and run the code cell below.

In [10]:
```
countrylist = indroot.findall('country')

for country in countrylist:
    print(country.attrib)
```

```
{'code': 'FRA', 'name': 'France'}
{'code': 'GBR', 'name': 'United Kingdom'}
{'code': 'USA', 'name': 'United States'}
```

Section 16.2: JSON Proceduarl Programming

We begin by demonstrating how to create a JSON file from a dictionary of dictionaries (DoD).

In the code cell below we define a DoD representation of some the `indicators` table related to the year 2016.

Study and run the code cell below.

In [11]:
```
ind2016DoD = {
    "CAN": {
        "country": "Canada",
        "pop": 36.26,
        "gdp": 1535.77,
        "life": 82.3,
        "cell": 30.75
    },
    "CHN": {
        "country": "China",
        "pop": 1378.66,
        "gdp": 11199.15,
        "life": 76.25,
        "cell": 1364.93
    },
    "IND": {
        "country": "India",
        "pop": 1324.17,
        "gdp": 2263.79,
        "life": 68.56,
        "cell": 1127.81
    },
```

```
    "RUS": {
        "country": "Russia",
        "pop": 144.34,
        "gdp": 1283.16,
        "life": 71.59,
        "cell": 229.13
    },
    "USA": {
        "country": "United States",
        "pop": 323.13,
        "gdp": 18624.47,
        "life": 78.69,
        "cell": 395.88
    },
    "VNM": {
        "country": "Vietnam",
        "pop": 94.57,
        "gdp": 205.28,
        "life": 76.25,
        "cell": 120.60
    }
}
```

Using this DoD, we can write the data to a JSON file using the `json.dump()` function we learned about earlier.

Study and run the code cell below.

In [12]:

```
jsonFile = os.path.join(datadir, "ind2016.json")

# Create the ind2016.json file and open it for "writing"
file = open(jsonFile, 'w')

# Export the DoD data into a JSON file and close it
json.dump(ind2016DoD, file)
file.close()
```

Example: Simple Table in JSON

Now that we have a JSON file, let's import it using the `json.load` function (this is just to give us practice doing so). We then make use of the `util.py` module that was created by the authors of our textbook (make sure this file is in the same directory as this notebook). This module is a set of functions to enable flexible exploration of (primarily) hierarchical data to ease discovery and understanding of structure.

In [13]:

```
# Import the util module (make sure it is in the same directory as this notebook)
import util

# Import the JSON file we just exported and assign it to ind2016
jsonFile = os.path.join(datadir, "ind2016.json")
with open(jsonFile) as file:
    ind2016 = json.load(file)
```

```
# Print the type of ind2016
print(type(ind2016))

# Use the print_data() function of the util module to print the first 15 lines of the i
util.print_data(ind2016, nlines=15)
```

```
<class 'dict'>
{
  "CAN": {
    "country": "Canada",
    "pop": 36.26,
    "gdp": 1535.77,
    "life": 82.3,
    "cell": 30.75
  },
  "CHN": {
    "country": "China",
    "pop": 1378.66,
    "gdp": 11199.15,
    "life": 76.25,
    "cell": 1364.93
  },
```

Note that the `json.load()` function properly loaded the JSON file into a dictionary structure.

Our goal is to load this data into a Pandas data frame. Looking at `ind2016`, it is clear that the internal dictionaries represent the rows we want in our resulting Pandas data frame. That is, we should have rows corresponding to country codes, and columns for `country`, `pop`, etc. Our plan will be to read this data into a list of dictionaries (one per country), where each of the dictionaries contains a `key:value` pair for `code`, that we can use as the index in the resulting Pandas data frame.

Using `ind2016.items()`, we can gain access to the `key:value` pairs in `ind2016`. Here each key is a code and the value is the dictionary of indicators. Because we want a column for the country codes, we add one more `key:value` pair to each interior dictionary, representing the code. Note that, because dictionaries are mutable, we make a `copy()` of each of the interior dictionaries so as not to mutate the originals in `ind2016`.

Study and run the code cell below.

In [14]:

```
# Create a empty list of dictionaries (LoD) which we will use to create a data frame
LoD = []

# Iteratate over the data in ind2016 using items() to create a LoD of the data
for code, rowD in ind2016.items():
    rowD = rowD.copy()
    rowD['code'] = code
    LoD.append(rowD)

# Create a data frame from the LoD representation
df = pd.DataFrame(LoD)
df.set_index('code', inplace=True)
df
```

Out[14]:

code	country	pop	gdp	life	cell
CAN	Canada	36.26	1535.77	82.30	30.75
CHN	China	1378.66	11199.15	76.25	1364.93
IND	India	1324.17	2263.79	68.56	1127.81
RUS	Russia	144.34	1283.16	71.59	229.13
USA	United States	323.13	18624.47	78.69	395.88
VNM	Vietnam	94.57	205.28	76.25	120.60

In the output above, in the first iteration we have a dictionary in `rowD` that is already close in structure and content to what we want as the first dictionary in LoD, which is `{'code':'CAN','country':'Canada',...,'cell':30.75}`. Making a copy of `rowD` and adding the code mapping to complete the row translation. From the perspective of traversals, this means it was enough for Python to only consider paths of length one in `ind2016`.

In general, we will have more work to do, looking at the subtree that represents each row. As the data is inherently two-dimensional, we should be ready to use nested loops to access the data. When we move to trees of depth three, we will need triply nested loops.

Single Table from JSON with Additional Level

We turn now to a tree of depth three, representing the indicators data from Chap. 15, which has, for each country, both multiple years and, for each year, multiple indicators.

We parse and print a prefix of the data to help understand its structure--**study and run the code cell below.**

In [15]:

```
jsonFile = os.path.join(datadir, "ind0.json")

# Import the ind0.json file into a Python dictionary
with open(jsonFile) as file:
    ind0 = json.load(file)

# Print the first 20 lines of the ind0 dictionary
util.print_data(ind0, nlines=20)
```

```
{
  "FRA": {
    "2007": {
      "pop": 64.02,
      "gdp": 2657.21
    },
    "2017": {
      "pop": 66.87,
      "gdp": 2586.29
    }
  },
```

```
  "GBR": {
    "2007": {
      "pop": 61.32,
      "gdp": 3084.12
    },
    "2017": {
      "pop": 66.06,
      "gdp": 2637.87
    }
```

Here, we see that we have children of the root representing countries, and then children of the country nodes representing years, and children of the years representing the individual indicators.

One solution strategy is to wrangle this data into a Pandas data frame with a two-level column index, with one level being year, and another level being the indicators within the year, like we saw originally in Chap. 6.

A row in our target data frame will contain all the information for one country, and there will be four data-carrying columns, one each for `pop` in 2007, `gdp` in 2007, `pop` in 2017, and `gdp` in 2017.

Our outer loop iterates over the children of the root (that is, country codes) and should generate exactly one row in our accumulation of rows per root. We have to "look inside" the internal dictionaries in `ind0`, and traverse each branch of the tree. Because each leaf is at depth three, this requires a triply nested loop.

Normally, a triply nested loop implies three-dimensional data. Of course, for a Pandas data frame, we want two-dimensional data. We can reduce from three dimensional data to two-dimensional data by inserting a composite value into our LoD, representing an indicator value and the year in which the data occurred. The solution below carries out this plan, using nested loops (by year, and then by indicator), and, within the inner loop, creating columns for each year/indicator combination and inserting data into these columns. We use string manipulation to combine the key of the year loop with the key of the indicator-loop to get our column names.

Study and run the code cell below.

In [16]:

```
# Create a empty list of dictionaries (LoD) which we will use to create a data frame
LoD = []

# Iteratate over the data in ind0 using items() to create a LoD of the data
for country, country_subtree in ind0.items():
    countryD = {'code': country}
    for year, indicators in country_subtree.items():
        for indicator, value in indicators.items():
            col_name = "{}-{}".format(year, indicator)
            countryD[col_name] = value
    LoD.append(countryD)

# Create a data frame from the LoD representation
df = pd.DataFrame(LoD)
df.set_index('code', inplace=True)
df
```

Out[16]:

	2007-pop	2007-gdp	2017-pop	2017-gdp

code	2007-pop	2007-gdp	2017-pop	2017-gdp
code				
FRA	64.02	2657.21	66.87	2586.29
GBR	61.32	3084.12	66.06	2637.87
USA	301.23	14451.90	325.15	19485.40

This sort of data frame should look familiar from Chap. 8. In order to make it conform to tidy data standards, we could convert the column names into a two-level index and then stack the year in Pandas.

Node Creation

Having discussed how to read a JSON-formatted tree into a Pandas data frame, by traversing each edge in the tree, we turn now to the problem of how to create new nodes in the tree.

This is a realm where hierarchical structure behaves very differently from tabular or relational structure. With tabular data, we would need to add an entire row to represent new data (and would need to include cells with value NA to represent missing data). **In the hierarchical model, we can add additional information anywhere in the tree whenever we decide we want to.** This means we do not have to add a value for every column/field, but rather only the ones that need it.

When the JSON data comes to us as a dictionary, we use the usual syntax for adding `key:value` pairs to a dictionary, namely `D[key] = value`. Here, the `key` is a country code, and the `value` is a tree, represented as a dictionary. For the example of `ind2016`, the equivalent of adding an entire row would be adding a subtree at depth one.

Study and run the code cell below.

In [17]:

```
# Add the information for China to the data set
ind2016["CHN"] = {"country": "China",
    "pop": 1378.66,
    "gdp": 11199.15,
    "life": 76.25,
    "cell": 1364.93
}
```

We could also add data for countries where we have incomplete information, by simply adding a dictionary with fewer than five entries.

Study and run the code cell below.

In [18]:

```
# Add partial information for Cuba
ind2016["CUB"] = {"country": "Cuba",
    "pop": 11.48,
    "life": 79.74,
    "cell": 3.99
}
```

We could even just add a single indicator.

Study and run the code cell below.

In [19]:
```
ind2016["IMN"] = {"country": "Isle of Man", "pop": 0.08}
```

Sometimes, data arrives at different times. In such situations, it can be added to the tree as it arrives. The first step is to create a subtree for the country in question.

Study and run the code cell below.

In [20]:
```
# First add the country (but no actual data)
ind2016["SXM"] = {"country": "Sint Maarten (Dutch part)"}

# Now add in the population
ind2016["SXM"]["pop"] = 0.4
```

Now let's print out the entire data set.

Run the code cell below.

In [21]:
```
util.print_data(ind2016)
```

```
{
  "CAN": {
    "country": "Canada",
    "pop": 36.26,
    "gdp": 1535.77,
    "life": 82.3,
    "cell": 30.75
  },
  "CHN": {
    "country": "China",
    "pop": 1378.66,
    "gdp": 11199.15,
    "life": 76.25,
    "cell": 1364.93
  },
  "IND": {
    "country": "India",
    "pop": 1324.17,
    "gdp": 2263.79,
    "life": 68.56,
    "cell": 1127.81
  },
  "RUS": {
    "country": "Russia",
    "pop": 144.34,
    "gdp": 1283.16,
    "life": 71.59,
    "cell": 229.13
  },
  "USA": {
    "country": "United States",
```

```
    "pop": 323.13,
    "gdp": 18624.47,
    "life": 78.69,
    "cell": 395.88
  },
  "VNM": {
    "country": "Vietnam",
    "pop": 94.57,
    "gdp": 205.28,
    "life": 76.25,
    "cell": 120.6
  },
  "CUB": {
    "country": "Cuba",
    "pop": 11.48,
    "life": 79.74,
    "cell": 3.99
  },
  "IMN": {
    "country": "Isle of Man",
    "pop": 0.08
  },
  "SXM": {
    "country": "Sint Maarten (Dutch part)",
    "pop": 0.4
  }
}
```

Note our new nodes in the output above, including the nodes with partial information.

To update the `SXM` node in `ind0` we could do the following--**study and run the code cell below**.

In [22]:

```
ind0["SXM"] = {"2016": {"pop":0.4}}
```

Node Attribute Updates

In the last examples above, we saw how to add a single leaf to an existing tree. The procedure for updating a leaf to take a different value is completely analogous, thanks to the simplicity of Python dictionaries.

For instance, if we learned that the data had an error, and in fact the population of Sint Maarten in 2016 was 40,000 instead of 400,000, then we could update the value by specifying the path and then specifying the new value. We illustrate for each of our two example trees.

Study and run the code cell below.

In [23]:

```
# Update in `ind2016`
ind2016["SXM"]["pop"] = 0.04

# Update in `ind0`
ind0["SXM"]["2016"]["pop"] = 0.04
```

If we wanted to change the units of all population leaves in the tree (to put each into units of

people, for instance), we could do so by specifying paths to each and updating them. We demonstrate for `ind2016` --**study and run the code cell below**.

In [24]:

```
for code in ind0:
    for year in ind0[code]:
        oldPop = ind0[code][year]["pop"]
        ind0[code][year]["pop"] = int(oldPop*1000000)

util.print_data(ind0, nlines=22)
```

```
{
  "FRA": {
    "2007": {
      "pop": 64019999,
      "gdp": 2657.21
    },
    "2017": {
      "pop": 66870000,
      "gdp": 2586.29
    }
  },
  "GBR": {
    "2007": {
      "pop": 61320000,
      "gdp": 3084.12
    },
    "2017": {
      "pop": 66060000,
      "gdp": 2637.87
    }
  },
  "USA": {
```

Congratulations on finishing this notebook--you are ready to start the homework associated with this material.

Section 16.3

XML Procedural Operations

Review of XML Format

- It is based on a Hierarchical/Tree Model.
- XML itself is a “meta-specification,” different grammars or vocabularies use the same framework to define domain or application-specific specifications, to which instances can conform.
- It does not require predefined, fixed schema.
- It has a flexible hierarchical structure (also arbitrary graphs).

XML XPath

- The XML XPath declarative query can be more complex to understand:
  - An alternative is to use procedural methods to extract the desired information.
  - Contrast this complexity with the relational model and its declarative language of SQL, which is easier to understand.

Reading and Traversing XML Data

- Given an XML file, we can use `lxml` and `etree` to represent the data in terms of its root `Element`.
- Such an `Element` has many important attributes:
- The **`.attrib`** attribute provides a dictionary of the XML attributes associated to the `Element`.
  - The usual Python methods and tools, `.keys()`, `.values()`, `.items()`, `[key]`, `in`, etc., work on the `.attrib` dictionary.
  - The `get(<attribute>)` method of an `Element` node; a `get()` for an attribute that does not exist yields `None`.
  - The `set(<attribute>)` method works similarly.

Reading and Traversing XML Data

- The `.text` attribute provides the text of the Element.
  - There is also `.tail` to get text that follows the `Element` and appears before the next `Element`.
- The `.tag` attribute gets the tag of the `Element`.

Useful Methods of `Element`

- `find(<tag-name>)` can be used to find the first matching subelement by tag name.
  - If there is no match as a child, the function reaches down further into the subtree.
  - This is a design decision, but in practice is better than counting on the order of data in the XML document to always give the right position for `find()`.
- `iterchildren(tag=<tagname>)` iterates over the children with the given tag.

More to Learn

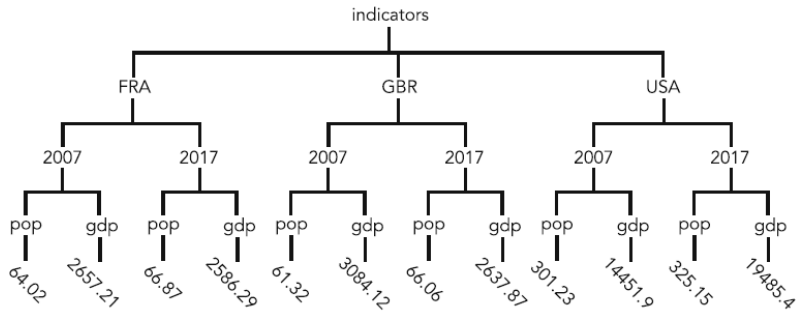
- We will demonstrate these operations presently.
- There are many, many, more attributes and methods associated with objects of type Element, and we encourage the reader to consult the online documentation

Wrangling XML Data

- Reconsider the `ind0` table and an XML version of the data:

ind0 table

Code	Year	Pop	Gdp
FRA	2007	64.02	2657.21
GBR	2007	61.32	3084.12
USA	2007	301.23	14451.90
FRA	2017	66.87	2586.29
GBR	2017	66.06	2637.87
USA	2017	325.15	19485.40



ind0.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<indicators>
  <country code="FRA" name="France">
    <timedata year="2007">
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
    </timedata>
  </country>
  <country code="GBR" name="United Kingdom">
    <timedata year="2007">
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </timedata>
  </country>
  <country code="USA" name="United States">
    <timedata year="2007">
      <pop>301.23</pop>
      <gdp>14451.9</gdp>
    </timedata>
    <timedata year="2017">
      <pop>325.15</pop>
      <gdp>19485.4</gdp>
    </timedata>
  </country>
</indicators>
```

- We begin by reading in the XML data and represent the tree in terms of its root, which we assign to `root0`:

```
xmlparser = etree.XMLParser(remove_blank_text=True)
ind0_path = os.path.join(datadir, "ind0.xml")

try:
    tree0 = etree.parse(ind0_path)
    root0 = tree0.getroot()
except:
  print("Exception in parsing XML")
```

- Because the data set `ind0` has a natural tabular structure (with a two-level index), it makes sense to attempt to wrangle the tree above into a Pandas data frame. To do this, we need to create a list of dictionaries (LoD) as in the previous section.

```
LoD = []

for country_node in root0:

    code = country_node.get('code')
    name = country_node.get('name')

   for timedata_node in country_node:

        year = int(timedata_node.get('year'))

        rowD = {'code': code, 'name': name, 'year': year}

        for indicator_node in timedata_node:

            indicator = indicator_node.tag
            value = float(indicator_node.text)
            rowD[indicator] = value

        LoD.append(rowD)

df = pd.DataFrame(LoD)
df.set_index(['code', 'year'], inplace=True)
df
```

Create an empty LoD

Loop over the nodes of the root, each of which is an `Element` representing a different country

Use the `get()` method of `Element` to retrieve the attributes `code` and `name`.

Loop over all the sub-Elements

Retrieve the `year` attribute

Create a dictionary containing `code`, `name`, and `year`

Loop over all the sub-sub-Elements which contain `pop` and `gdp`

Retrieve the tags (`pop` and `gdp`) and the text (the actual values). Then add them to `rowD`.

Append the dictionary to the list

Create the data frame from the LoD

- This produces the following output:

		name	pop	gdp
code	year			
FRA	2007	France	64.02	2657.21
	2017	France	66.87	2586.29
GBR	2007	United Kingdom	61.32	3084.12
	2017	United Kingdom	66.06	2637.87
USA	2007	United States	301.23	14451.90
	2017	United States	325.15	19485.40

- Now that we understand the structure of our solution above, we wish to apply it to the full `indicators` data set, stored as `indicators.xml`.

- Because the full data set has the same structure and constraints as `ind0`, it can be processed identically. Thus, we abstract our solution above into a function, and then apply it to the root `Element` obtained from parsing the full data set using `etree`.

```
def xml_indicators(root_node):
    LoD = []

    for country_node in root_node:
        code = country_node.get('code')
        name = country_node.get('name')

        for timedata_node in country_node:
            year = int(timedata_node.get('year'))
            rowD = {'code': code, 'name': name, 'year': year}

            for indicator_node in timedata_node:
                indicator = indicator_node.tag
                value = float(indicator_node.text)
                rowD[indicator] = value

            LoD.append(rowD)

    df = pd.DataFrame(LoD)
    df.set_index(['code', 'year'], inplace=True)

  return df
```

- We can now use this function to parse the `indicators.xml` data. First, we load in the data.

```
xmlparser = etree.XMLParser(remove_blank_text=True)
ind_path = os.path.join(datadir, "indicators.xml")

try:
    tree = etree.parse(ind_path)
    root = tree.getroot()
except:
  print("Exception in parsing XML")
```

- Now we pass the root into our function:

```
df = xml_indicators(root)

# Print the rows associated with indices 200-204
df.iloc[200:205,0:4]
```

- This produces the following output:

		name	pop	life	cell
code	year				
ALB	1983	Albania	2.84	70.9	0.0
	1984	Albania	2.90	71.1	0.0
	1985	Albania	2.96	71.4	0.0
	1986	Albania	3.02	71.6	0.0
	1987	Albania	3.08	71.8	0.0

- It is worth noting that our solution yields values of `NaN` in the Pandas data frame when there is missing data in the XML file. Our solution never attempts to access data that is missing from the tree (note that a list of headers is not provided along with the XML file).

- If, for a given country code and year, there is missing data, then the corresponding dictionary in the LoD has fewer entries than the other dictionaries in the LoD. When the LoD is passed to Pandas, those missing entries are automatically filled in with value `NaN`.

- As a consequence, this implies that if the XML file has a typo, e.g., if for the USA in 2015, "`gdp`" is entered erroneously as "`gpd`," then one of the dictionaries in the LoD will have a `key:value` pair where the key is "`gpd`," and Pandas will create a column called "`gpd`" which takes value `NaN` for all entries except the USA in 2015.

- One should always check the resulting Pandas data frames for errors of this sort.

Creating XML Data

- Suppose we have a JSON file containing a desired hierarchical structure that we wish to export as XML data.

- As our example, we will use the `ind0.json` data, which we import and print below.

```
# Import the JSON file and assign it to ind0
jsonFile = os.path.join(datadir, "ind0.json")
with open(jsonFile) as file:
    ind0 = json.load(file)

# Use the print_data() function of the util module to print the data
util.print_data(ind0)
```

- Below is the output:

```
{
  "FRA": {
    "2007": {
      "pop": 64.02,
      "gdp": 2657.21
    },
    "2017": {
      "pop": 66.87,
      "gdp": 2586.29
    }
  },
  "GBR": {
    "2007": {
      "pop": 61.32,
      "gdp": 3084.12
    },
    "2017": {
      "pop": 66.06,
      "gdp": 2637.87
    }
  },
  "USA": {
    "2007": {
      "pop": 301.23,
      "gdp": 14451.9
    },
    "2017": {
      "pop": 325.15,
      "gdp": 19485.4
    }
  }
}
```

- We assume:
  - The top level is a dictionary mapping country codes to a country dictionary,
  - The country dictionary maps from an integer year to an indicator dictionary, and
  - The indicator dictionary maps from indicator name to numeric value.

- To convert this data into XML, we need to iterate over the country dictionaries. For each, we need to build the corresponding XML subtree. Doing this will require us to iterate over the indicators dictionary and build the corresponding XML subtree for each. We then need to latch all the XML subtrees together. This will involve several new `Element` methods:

- For a given string key, the method `etree.Element(key)` creates an `Element` whose tag is `key`.

  – This function automatically creates both the start and end tags.

- For a given string `v`, and a given node, `node.text = v` sets the text of node.

- For text `tg`, `att`, the method `etree.Element(tg, <attribute>=att)` creates an `Element` whose tag is `tg` and with a given `<attribute>` set to `att`.

- For a node `p` and a node `c`, the method `p.append(c)` creates `c` as a child of `p`.

  – This automatically nests the start and end tags of `c` within the start and end tags of `p`.

- First, we consider the problem of converting from an indicator dictionary into an XML tree.

- Consider the function below. The first line creates an `Element`, corresponding to the year. Then the loop creates several more elements (one per indicator), which are then attached to the year `Element` via append. In the end, `timedate` represents the XML subtree.

```
import numpy as np

def buildTimedata(year, indvars):
    """
    Given a year (as a string or as an int), create an
    Element whose tag is timedata, with attribute of a
    year set to the given year, and with a set of
    children, one for each key-value pair in indvars.
    """
    timedata = etree.Element("timedata", year=str(year))

    for key, value in indvars.items():
        if not np.isnan(value):
            node = etree.Element(key)
            node.text = str(value)
            timedata.append(node)

  return timedata
```

- Below we test out our function:

```
node = buildTimedata("2007", {"pop": 64.02, "gdp": 2657.21})

util.print_xml(node)
```

```
<timedata year="2007">
  <pop>64.02</pop>
  <gdp>2657.21</gdp>
</timedata>
```

- With this function in hand, we can build the XML tree associated to a country dictionary reasonably easily. We again create an `Element` to serve as the root (labeled by the `code`), then attach the subtrees corresponding to each year, which we build using the function above.

```
def buildCountry(code, years):
    """
    Build a country xml subtree. Country node is tagged
    'country' and has an attribute of the country's code.
    """
    country_node = etree.Element("country", code=code)

    for year,indvars in years.items():
        timedata = buildTimedata(year, indvars)
        country_node.append(timedata)

  return country_node
```

- We illustrate this function with a sample invocation:

```
data = {"2007": {"pop": 64.02, "gdp": 2657.21},
        "2017": {"pop": 66.87, "gdp": 2586.29}}

node = buildCountry("FRA", data)

util.print_xml(node)
```

```
<country code="FRA">
  <timedata year="2007">
    <pop>64.02</pop>
    <gdp>2657.21</gdp>
  </timedata>
  <timedata year="2017">
    <pop>66.87</pop>
    <gdp>2586.29</gdp>
  </timedata>
</country>
```

- We now bring the entire solution together, again as a function, which we could use on the full indicators data set.

```
def buildTree(datadir, filename):
    # Load JSON into in-memory dictionary data structure
    path = os.path.join(datadir, filename)

    with open(path) as f:
        ind0Dict = json.load(f)

    # Create root Element and, using add'l function, build
    # each of the country subtrees, adding to created root.

    # Demonstrates constructor and append method of an Element node
    ind0_root = etree.Element('indicators')
    for code, yearD in ind0Dict.items():
        country_subtree = buildCountry(code, yearD)
        ind0_root.append(country_subtree)

    # Convert tree to a string
    xml_string = etree.tostring(ind0_root).decode('utf-8')

    return xml_string
```

- Below we test out this function on the `ind0.json` file and assign the XML string to `xml_string`, which we in turn write to the file `ind0_new.xml`. Finally, we load in the newly created XML file and print it to verify everything works as expected.

```
# Import ind0.json and create an XML tree from the data
xml_string = buildTree(datadir, "ind0.json")

# Write the string to the file ind0_new.xml
file = open("ind0_new.xml", 'w')
file.write(xml_string)
file.close()

# Import the file ind0_new.xml
pathInd = os.path.join(datadir, "ind0_new.xml")

stripparser = etree.XMLParser(remove_blank_text=True)
indtree = etree.parse(pathInd, stripparser)
indroot = indtree.getroot()

# Print the XML tree
s = etree.tostring(indroot, pretty_print=True).decode('utf-8')
print(s)
```

```
<indicators>
  <country code="FRA">
    <timedata year="2007">
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
    </timedata>
  </country>
  <country code="GBR">
    <timedata year="2007">
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </timedata>
    <timedata year="2017">
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </timedata>
  </country>
  <country code="USA">
    <timedata year="2007">
      <pop>301.23</pop>
      <gdp>14451.9</gdp>
    </timedata>
    <timedata year="2017">
      <pop>325.15</pop>
      <gdp>19485.4</gdp>
    </timedata>
  </country>
</indicators>
```

- As you can see, we have successfully converted the `ind0.json` file into an equivalent XML file!

Section 16.4

XPath

Introduction to XPath

- The ability to specify paths for locating and updating resources is fundamental to the hierarchical data model.
- XPath is a language that facilitates path-based operations in XML documents.
- The name “XPath” comes from “XML Path Language.” XPath has a number of important applications:
  - Extracting text or attribute data in XML documents.
  - Retrieving data stored within HTML files.
  - Iterating over web pages or documents in a file structure.

Introduction

- XPath uses a compact, string-based, rather than XML element-based syntax.
- Operates on the abstract, logical structure of an XML document (tree of nodes) rather than its surface syntax.
- Uses a path notation (like URLs) to navigate through this hierarchical tree structure.

Introduction Continued

- XPath defines a way to compute a string-value for each type of node: element, attribute, text.
- Evaluated to yield an object of 4 basic types.
  - node-set (unordered collection of nodes w/o duplicates).
  - boolean (true/false)
  - number (float)
  - string

Introduction Continued

- A location path describes a path from 1 point to another.
- Analogy: Set of street directions.

  "Second store on the left after the third light"
- The location path provides the mechanism for 'addressing' items in an XML document.

Expressions in XPath

- An XPath expression is a string, representing a sequence of steps in a tree, making up a path from some "initial" node to some "target" node.
- The initial node is often the root but does not have to be.
- XPath has the same notions of *absolute* and *relative* path to specify a traversal.

ind2.xml

```
<ind2>
  <FRA>
    <y2007>
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </y2007>
    <y2017>
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
     </y2017>
  </FRA>
  <GBR>
    <y2007>
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </y2007>
    <y2017>
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </y2017>
  </GBR>
</ind2>
```

Example XPath String

Path: `/ind2/FRA/y2017`

- This defines an individual internal node (Element):

Return:
```
Element='<y2017>
          <pop>66.87</pop>
          <gdp>2586.29</gdp>
          </y2017>'
```

Example XPath String

Path: `/ind2/FRA/y2017/pop`

Return: `Element='<pop>66.87</pop>'`

Path: `/ind2/FRA/y2017/pop/text()`

Return: `Text='66.87'`

`text()` extracts the text of the current node

Typical Goal

- The goal of Xpath is often like `findall()` to get a nodeset, the set of nodes (over multiple path-traversals) that satisfy a given path pattern.
- In Python, once we have an XPath expression, we give it as a parameter to the `xpath()` method of an `ElementTree` or an `Element`, which returns the nodeset.

Path Pattern

- A string argument for a *path pattern* defines location-steps with parts that have to "match" as specified, and other parts that are wildcards for one or more nodes or traversals that can match.
- The logic of wildcard characters in XPath works similar to regular expressions.

Path Pattern Example

Path: `/ind2/FRA/*/pop`

Return: `Element='<pop>64.02</pop>'`
`Element='<pop>66.87</pop>'`

Matches all grandchildren tagged `'pop'` under the `Element` tagged `'FRA'` in `ind2`, regardless of the parent of `'pop'`.

There are two elements in the returned nodeset.

Note: A wildcard matches any node at the given level, but not a path worth of nodes. The example above would not match nodes where `'pop'` is a greatgrandchild of `'FRA'`.

Expressions

- Certain characters/sequences of characters are *expressions* that have *meaning* that is not interpreted as tagged element names.

- **The following two paths are equivalent**, but the second one uses `[@code = 'FRA']` to specify a *predicate*, which will be satisfied by the country node whose code is `'FRA'`, but not the other country nodes.

Path: `/ind2/FRA/y2007/pop`

Path: `/indicators/country[@code='FRA']/timedata[@year ='2007']/pop`

This should be interpreted as a single string without a line break.

Return: `Element='<pop>64.02</pop>'`

indicators.xml

```
<indicators>
  <country code="USA" name="United States">
    <timedata year="1960">
      <pop>180.67</pop>
      <gdp>543.3</gdp>
      <life>69.8</life>      ⋮
      <cell>0.0</cell>
      <imports>16170.7</imports>
      <exports>20535.0</exports>
    </timedata>
    <timedata year="1961">
      <pop>183.69</pop>
      <gdp>563.3</gdp>
      <life>70.3</life>
       …
```

Logical Or

- If we wanted the population of France and the USA in *all* years, we would use a logical *or*, yielding the expression

Path: `/indicators/country[@code ='FRA' or @code ='USA']/*/pop`

Return:

```
Element= '<pop>46.62</pop>'
Element= '<pop>47.24</pop>'
Element= '<pop>47.9</pop>'
Element= '<pop>48.58</pop>'
…
```

Alternate Method

- An alternate method for finding the population of France and the USA in all years makes use of the | operator:

Path: `/indicators/country[@code='FRA']/*/pop | /indicators/country[@code = 'USA']/*/pop`

- The vertical bar (|) is used between XPaths in the same string to combine the nodeset results from the first XPath with the nodeset results from the second XPath.

school.xml

```
<school>
  <departments>
    <department id="ANSO">
      <name>Anthropology and Sociology</name>
      <division>Social Sciences</division>
    </department>
    …
  </departments>
  <courses>
    <course subject="ARAB" num="111">
      <title>Beginning Arabic I</title>
      <hours>4.0</hours>
      <class id="40184">
        <term>FALL</term>
        <section>01</section>
        <meeting>09:30-10:20 MWRF</meeting>
        <instructorid>9216</instructorid>
      </class>
    </course>
    …
```

Logical And

- We could use a logical *and* to find the title "Elementary Cinema Prod" with course number 219 in the `school.xml` file as follows:

Path: `/school/courses/course[@subject='CINE' and @num='219']/title`

Return: `Element= '<title>Elementary Cinema Prod</title>'`

Expressions

Expression	Meaning
/	When the first character, means the traversal starts at the root of the tree. If not the first character, is used to separate location-steps in the set of possible traversals
.	Refers to the current node of the traversal
..	Means the parent of the current node of the traversal. Every node has a parent, and the parent of the root is the root itself
@	Used to reference/match an *attribute* (instead of an element tag)
[]	Used, relative to the node of the current location-step, to specify a predicate (i.e., something that results in a Boolean true/false, often involving an attribute of the current node)

Expressions

Expression	Meaning
or, and	Used inside a [] expression to specify logical operators
vert bar	Used between XPaths in the same string to combine the nodeset results from the first XPath with the nodeset results from the second XPath
//	Matches *all* descendant traversals/paths from the node of the current location-step
*	Matches all the element siblings relative to the current location-step level
@*	Matches all the attributes relative to the current location-step level (or predicate, if used with [])
text()	Extracts the text of the current node

Match All Descendants

- The expression `//` is often a useful start of an XPath.
- We could use this to find all nodes with a specified text. For example, in `indicators` we could match all nodes with population larger than 300 million:

Path: `//pop[text() > 300]`

Backing Up a Level

- Suppose we wanted to find what department offers the course with the title "Elementary Cinema Prod“ in the `school.xml` data.
- We could locate that course then **back up one level** and extract the subject attribute:

Path:
```
//title[text() =
        "Elementary Cinema Prod"]/../@subject
```

Return: `Attribute='subject=CINE'`

breakfast.xml

```
<menu>
    <food price="5.95" calories="650">
        <name>Belgian Waffles</name>
        <description>Two of our famous Belgian Waffles with maple syrup</description>
    </food>
    <food price="7.95" calories="900">
        <name>Strawberry Belgian Waffles</name>
        <description>Light Belgian waffles covered with strawberries</description>
    </food>
    <food price="8.95" calories="900">
        <name>Berry-Berry Belgian Waffles</name>
        <description>Light Belgian waffles covered with fresh berries</description>
    </food>
    <food price="4.5" calories="600">
        <name>French Toast</name>
        <description>Thick slices made from our homemade sourdough bread</description>
    </food>
    <food price="6.95" calories="950">
        <name>Homestyle Breakfast</name>
        <description>Two eggs, bacon, toast, and hash browns</description>
    </food>
</menu>
```

Another Example

- The following (**equivalent**) paths extracts all the prices of the food in the `breakfast.xml` data set.

Input: `//@price`

Input: `/menu/food/@price`

Keep in mind that using `\\` can be very slow!

```
Attribute='price=5.95’
Attribute='price=7.95’
Attribute='price=8.95’
Attribute='price=4.5’
Attribute='price=6.95'
```

Searching Text Nodes

- XPath can also help us search text nodes, in a way similar to what we saw with regular expressions and in SQL `LIKE` queries.
- For instance, the `contains()` function allows us to match nodes where the text contains some specified string.
- For example, in the `topnames.xml` data, the expression

```
//name[contains(text(),'Jo')]
```

matches all name nodes where the text contains `'Jo'`. This includes `'John'` and would also contain `'Joanna'` or `'MoJo'` if those were ever top names.

Searching Text Nodes Continued

- A related function, `starts-with()`, determines whether the text starts with a given string.
- For example,

```
//name[starts-with(text(),'Jo')]/text()
```

matches `'John'` but would not match `'MoJo'`.

Example with Two Conditions

- The following path finds all the years that the USA has a GDP above 500 **and** a population less than 250.

Path: `/indicators/country[@code='USA']/timedata`
`/gdp[text()>5000]/../pop[text()<250]`
`/../@year`

Selects nodes with `gdp > 500`

Backs up and then tests if `pop < 250`

Backs up again to select the year

Return: `['1988', '1989', '1990']`

Python Programming with XPath

- We now show how to use XPath expressions in Python programming.

- We assume that we have already imported `etree` and parsed our set of input files (we will use the `getLocalXML(filename, datadir)` function we defined in Homework #12.

getLocalXML()

```
def getLocalXML(filename, datadir="."):
    # Set the data directory
    xmlPath = os.path.join(datadir, filename)

    try:
        tree = etree.parse(xmlPath)
        root = tree.getroot()
        return root
    except:
        print("Exception in parsing XML")
        return None
```

Obtain Root of breakfast.xml

```
breakfastRoot =
     getLocalXML("breakfast.xml", datadir)
```

Python Example 1

```python
foodList = breakfastRoot.xpath("/menu/food/@price")

print(foodList)
print("The total number of foods is " + str(len(foodList)))
```

```
['5.95', '7.95', '8.95', '4.5', '6.95']
The total number of foods is 5
```

- Note that `xpath()` takes as input an XPath expression as a Python string and **returns a list** of nodes that match.

Python Example 2

- Below we find the cost of French Toast:

```python
cost = breakfastRoot.xpath("/menu/food/name[text()='French Toast']/../@price")

print(cost)

print("The cost of French Toast is " + str(cost[0]))
```

```
['4.5']
The cost of French Toast is 4.5
```

Remember: `xpath()` always returns a list!

Python Example 3

- Below we use functional abstraction to find the cost of a provided food.
- Note the use of the triple double quotes (""").

```python
def findCost(root, food):
    xpath_string = """/menu/food/name[text()='{}']/../@price""".format(food)
    cost = root.xpath(xpath_string)

    return float(cost[0])
```

```python
findCost(breakfastRoot, "French Toast")
```

```
4.5
```

```python
findCost(breakfastRoot, "Belgian Waffles")
```

```
5.95
```

```python
findCost(breakfastRoot, "Egg McMuffin")
```

- This throws an error because the item is not in the data, so `xpath()` returns an empty list.
- The error occurs when we try to access `cost[0]`

More to Learn

- There is a lot more to learn about XPath and you are encouraged to find some additional tutorials and/or study Section 16.4 in our textbook.

Chapter 22

Introduction to Web Scraping

What is Web Scraping?

- ***Web scraping*** is a technique for gathering data or information on web pages.
- That is, it is a method to extract data from a website that does not have an API, or we want to extract a LOT of data which we can not do through an API due to rate limiting.
- **Through web scraping we can extract any data which we can see while browsing the web.**

But What is an API?

- The term API stands for “Application Programming Interface.”
- APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.
- For example, the weather bureau’s software system contains daily weather data. The weather app on your phone “talks” to this system via APIs and shows you daily weather updates on your phone.

API Continued

- API architecture is usually explained in terms of client and server.
- The application sending the request is called the ***client***, and the application sending the response is called the ***server***.
- So in the weather example, the bureau’s weather database is the server, and the mobile app is the client.

REST APIs

- REST stands for “Representational State Transfer.”
- REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP (which is a protocol for fetching resources such as HTML documents).
- REST APIs are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client.

REST APIs Continued

- REST APIs are covered in Chapter 23 of our textbook.
- Many important websites provide APIs, such as:
  - GitHub
  - IMDb (movie database)
  - Wikipedia
  - Twitter

Web Scraping in Real Life

- Extract product information
- Extract job postings and internships
- Extract offers and discounts from deal-of-the-day websites
- Crawl forums and social websites
- Extract data to make a search engine
- Gathering weather data

Web Scraping vs. Using an API

- Web Scraping is not rate limited
- Anonymously access the website and gather data
- Some websites do not have an API
- Some data is not accessible through an API
- and many more !

Essential Parts of Web Scraping

- Web Scraping follows this workflow:
  - Get the website - using HTTP library
  - Parse the HTML document - using any parsing library
  - Store the results - either a db, csv, text file, etc
- We will focus more on parsing.

Useful Libraries for Web Scraping

- Beautiful Soup
- **lxml**
- Selenium
- **Scrapy**

`lxml` Toolkit

- The `lxml` XML toolkit:
  - very fast
  - not purely in Python
  - If you have no "pure Python" requirement use `lxml`
  - `lxml` works with all python versions from 2.x to 3.x
- Because of the commonality in our study of XML, here we will focus on using `lxml`.

Scrapy

- Scrapy is very fast.
- Asynchronous (which allows you to process and collect data from a large number of web pages in parallel)
- Easy to use.
- Made in Python.
- You will learn about using Scrapy in the *Web Scaping in Python* course on DataCamp.

When to Use Web Scraping

- When you have to scrape millions of pages.
- When you want asynchronous support out of the box.
- When you don't want to reinvent the wheel

HTML Basics for Web Scraping

- HTML stands for HyperText Markup Language.
- It’s not a programming language like Python or Java, but it’s a *markup* language.
- It describes the elements of a page through tags characterized by angle brackets.

Simple HTML Example

```html
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<h3>Third Header</h2>
</body>
</head>
</html>
```

My Website

Second header

Third Header

- The document always begins and ends using `<html>` and `</html>`.
- `<body></body>` constitutes the visible part of HTML document.
- `<h1>` to `<h3>` tags are defined for the headings.

- We can also add a brief paragraph under the second header using the `<p>` tag:

```html
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." Albert Einstein</p>
<h3>Third Header</h2>
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." Albert Einstein

Third Header

- To be more informative, we can add a link on the word “Albert Einstein” to send visitors to the Wikipedia page directly. `<a>` is the tag specialized for HTML links. It has the `href` attribute to specify the link:

```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." <a href="https://it.wiki
<h3>Third Header</h2>
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." Albert Einstein

Third Header

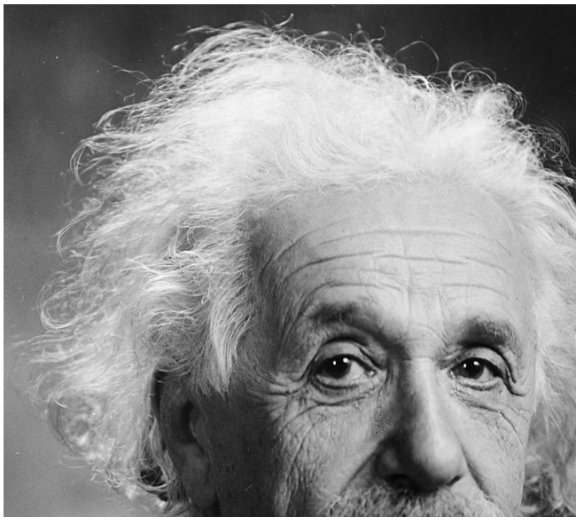
- We can insert an image of Albert Einstein. The tag for this task is `<img>`, which has the attribute `src` to specify the URL of the image.

```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." <a href="htt
<img src="https://upload.wikimedia.org/wikipedia/commons/thumb/d/d3/Albert_E
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." Albert Einstein



Classes and ID

- You are probably asking yourself why you need the classes and ID. The names alone seem so boring, but once you understand why you need them, you won't be able to do without them.
- The `id` is an attribute to specify a unique ID for an element. For example, you want a particular color and size for the title in the first header.
- The `class` is an attribute to define different elements with the same class name. Why do you need the same class in some elements? Because you would probably want to write some phrases with the same font, color, and size.

Classes and ID

- Both IDs and classes are defined in the `<style>` tags and the properties are defined in the curly braces (`{ }`).
- The syntax for the class needs the period (`.`) followed by the name of the class, while the ID needs the hashtag (`#`) followed by the name of the ID.
- Once you create the class and the ID in the `<style>` tags, you need to pass them in the elements you want.

```
<html>
<head>
<style>
#mytitle {
  font-size: 60;
  color: blue;
}
.cit {
  background-color: pink;
  color: white;
  font-family: Arial, Helvetica, sans-serif;
}
</style>
<body>
<h1 id="mytitle">My Website</h1>
<h2>Second header</h2>
<p class='cit'>"<i>Two things are infinite. The universe and human stupidity<i>.'
<p class='cit'>"<i>Immagination is more important than knowledge<i>." <b>Albert E
</body>
</head>
</html>
```

Tables

- Another important feature of HTML is the table, which is defined by the `<table>` tag.

- Within the `<table>` tag, there are three principal tags to remember:
  - The `<tr>` tag is used to build each row of the table.
  - The `<th>` tag is used to define the header.
  - The `<td>` tag is used to define the cell within the row.

```
<html>
<head>
<style>
table, th, td {
  border: 1px solid black;
  border-collapse: collapse;
}
</style>
```

```
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"<i>Two things are infinite. The universe and human stupidity<i>." <a href="https://it
<table>
<tr><th>Discoveries</th><tr>
<tr><td>Special Relativity</td></tr>
<tr><td>General Relativity</td></tr>
</table>
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." *Albert Einstein*

Discoveries
Special Relativity
General Relativity

Lists

- There are two types of lists that can be defined in HTML.
- The first one is an unordered list that starts with the `<ul>` tag, while the other type is an ordered list specified by the `<ol>` tag.
- Each item of both types of the list is specified by the `<li>` tag.

```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"<i>Two things are infinite. The universe and human stupidity<i>." <a href="htt
<p>Discoveries</p>
```

```html
<ul>
<li>Special Relativity</li>
<li>General Relativity</li>
</ul>
<p>Awards</p>
<ol>
<li>Max Planck Medal</li>
<li>Nobel Price in Physics</li>
</ol>
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." *Albert Einstein*

Discoveries

- *Special Relativity*
- *General Relativity*

Awards

1. *Max Planck Medal*
2. *Nobel Price in Physics*

Blocks

- The most common elements of a website are usually called Blocks or Containers.
- They are useful to group together different elements and apply the same properties. So, the elements we did until now, `<h1>` to `<h3>`, `<p>`, `<ul>`, `<ol>`, can form one block together.
- For example, we want to divide the page into two parts. To create these two different blocks we can specify the `<div>` tag.

```html
<html>
<head>
<style>
* {
  box-sizing: border-box;
}

/* Create two equal columns that floats next to each other */
.column {
  float: left;
  width: 50%;
  padding: 10px;
  height: 300px;
}

.row:after {
  content: "";
  display: table;
  clear: both;
}
</style>
```

```html
<body>

<h2>Glossary</h2>

<div class="row">
  <div class="column" style="background-color:#e6e6ff;">
    <h2>General Relativity</h2>
    <p>General relativity, also known as the general theory of relativity, is the geometric the
  </div>
  <div class="column" style="background-color:#9999ff;">
    <h2>Special relativity</h2>
    <p>In physics, the special theory of relativity, or special relativity for short, is a scie
  </div>
</div>

</body>
</head>
</html>
```

Glossary

General Relativity

General relativity, also known as the general theory of relativity, is the geometric theory of gravitation published by Albert Einstein in 1915 and is the current description of gravitation in modern physics.

Special relativity

In physics, the special theory of relativity, or special relativity for short, is a scientific theory regarding the relationship between space and time.

Final Thoughts

- There are many other tags, but this quick overview should provide a good starting point.
- Web scraping is a task that needs to be divided into subtasks.
- Once you know basic HTML, you can easily use Python libraries such as `lxml`, Beautiful Soup, Scrapy, and Selenium.

Activity 06 - Introduction to Web Scraping (Chapter 22)

DATA 200 - Database Systems and Data Management for Data Analytics
Dickinson College

In [1]:

```
## RUN THIS CELL TO GET THE RIGHT FORMATTING AND TO LOAD NumPy
import requests
import numpy as np
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/maste
HTML(styles)
```

Out[1]:

This notebook will cover Chapter 22 (Web Scrapping) in our textbook

Section 22.2.1: Formulating Requests for HTML

In this section we will consider two examples that both scrap data from an HTML table.

Run the code cell below to import the needed libraries(note that we are importing the `io` and `request` packages).

In [2]:

```
import os.path
from lxml import etree
import pandas as pd
import util

# New libraries we have not used before
import io
import requests
```

It important to note that not all HTML follow strict tree formating. Nonetheless, to accomplish web scraping, we need a tree that we can use for both discovery and for programmatic extraction of data. So, instead of using the default `ElementTree` parser, or specifying a custom XML parser, we define a parser that can take many forms of "broken" HTML and parse into a properly formed `ElementTree`.

In the code cell below we parse a string version of "broken" HTML (which lacks the closing tag corresponding to the `html`, `head`, `title`, and `body` start tags and closes a `h1` with an `h3`).

But using the `ElementTree` constructor for an `HTMLParser` and then using that to parse the string, we obtain an appropriate tree as the result.

Study and run the code cell below.

In [3]:

```
# Define string representing bad HTML
bad_html = "<html><head><title>test<body><h1>header title</h3>"

# Parse the "bad" HTML
from lxml import etree
htmlparser = etree.HTMLParser()
tree = etree.parse(io.StringIO(bad_html), htmlparser)

# Print the tree structure obtained from the parser
util.print_xml(tree.getroot())
```

```
<html>
  <head>
    <title>test</title>
  </head>
  <body>
    <h1>header title</h1>
  </body>
</html>
```

Note that all unclosed end tags have been closed, and the `<h1>` was closed with a `</h1>`. The parser uses known structure and common nesting to make a *best-effort* construction of a tree from the HTML input.

If, in the course of web scraping, there is HTML input that is even more malformed and unable to be parsed in the above manner, you may need to use a submodule of `lxml` called `html` (which you can read more about on your own).

There are a numbe of Python packages out there that specifically support web scraping, such as Beautiful Soup, Selenium, and Scapy.

Because of the commonality in what we are doing here with our earlier treatment of XML, we will continue to use lxml.

Section 22.2.2: Simple Tables

Our first example entails an HTML table at datasystems.denison.edu/ind2016.html , which is where the authors of our textbook are from. This is a very basic HTML page to help us get started. Our goal is to

- Acquire the web page
- Discover how the data set is represented,
- Extract the data into a data frame.

We submit a `GET` request, passing the url for the HTML resource, and use the *raw bytes* version of the body of the response to obtain a parse tree to use for subsequent operations. To display the results, we again use `util.print_xml()` and use its parameters to limit the depth of the recursion and the maximum number of children at a particular level. This way we can help our discovery and examine subtrees as we explore a given HTML.

Study and run the code cell below.

In [5]:

```
# Use requests.get() to obtain the URL and assign it to response
url = "https://datasystems.denison.edu/ind2016.html"
response = requests.get(url)

assert response.status_code == 200

# Parse response using raw bytes version
tree1 = etree.parse(io.BytesIO(response.content), htmlparser)
root1 = tree1.getroot()

# Display the results
util.print_xml(root1, nlines=15, depth=4, nchild=3, width=100)
```

```
<html>
  <head>
    <meta charset='utf-8'></meta>
    <meta name='viewport' content='width=device-width, initial-scale=1.0'></meta>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'></meta>
     ...
  </head>
  <body>
    <div class='wrapper'>
      <<cyfunction Comment at 0x0000013C603311E0>>Page Content</<cyfunction Comment at 0
x0000013C603
      <div id='content-no-side'>
        <<cyfunction Comment at 0x0000013C603311E0>>Top Navigation</<cyfunction Comment
at 0x0000013
        <nav id='top-nav' class='navbar fixed-top navbar-expand-sm offset-sm-0'>
        </nav>
        <<cyfunction Comment at 0x0000013C603311E0>>Division for the rest of the content
of the page
```

Note that the structure of the HTML has a `head` and a `body`, and in the body, we have significant nesting of `div` nodes.

To be able to process a table for its data content, we must find the HTML table element in question from within the overall tree of HTML. We can start by using XPath and querying for the set of `table` elements within the body of the document.

Study and run the code cell below.

In [8]:

```
nodeset = root1.xpath("/html/body/div//table")
print(len(nodeset))

table = nodeset[0]
util.print_xml(table, depth=3, nchild=4)
```

```
1
<table class='table table-dark' style='width: 600px'>
  <thead>
    <tr>
      <th title='Field #1'>code</th>
      <th title='Field #2'>country</th>
      <th title='Field #3' class='text-right'>pop</th>
      <th title='Field #4' class='text-right'>gdp</th>
       ...
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>CAN</td>
      <td>Canada</td>
      <td align='right'>36.26</td>
      <td align='right'>1535.77</td>
       ...
    </tr>
    <tr>
      <td>CHN</td>
      <td>China</td>
      <td align='right'>1378.66</td>
      <td align='right'>11199.15</td>
       ...
    </tr>
    <tr>
      <td>IND</td>
      <td>India</td>
      <td align='right'>1324.17</td>
      <td align='right'>2263.79</td>
       ...
    </tr>
    <tr>
      <td>RUS</td>
      <td>Russia</td>
      <td align='right'>144.34</td>
      <td align='right'>1283.16</td>
       ...
    </tr>
     ...
  </tbody>
</table>
```

We find that the `table` has children of `thead` and `tbody`, that there is a single row in `thead` that has `th` elements for each of the column names, and that the data is contained in the `tr` elements of `tbody`. We can obtain a vector of column names via XPath as follows.

Study and run the code cell below.

In [7]:

```
column_names = table.xpath("./thead/tr/th/text()")
column_names
```

Out[7]:

```
['code', 'country', 'pop', 'gdp', 'life', 'cell']
```

We have two options for acquiring the data itself. Given the simple nature of the table, we see that the field values are uniformly represented in the text of each of the `td` nodes. In the code cell

below we constuct a dictionary of column lists (DoL) representation. Specifically, we design an XPath expression that yields a list of data values for a particular column. This uses a predicate that involves the 1-relative position of the column (i.e., starting to count at 1 instead of 0). This strategy is then employed in a loop to create each of the dictionary columns based on the set of column names obtained above.

In the code below we use Python's `enumerate()` method, which you can read about here.

Study and run the code cell below.

In [13]:

```
DoL = {}

for index, column in enumerate(column_names):
    xpath = ".//table//tr/td[{}]/text()".format(index+1)
    DoL[column] = root1.xpath(xpath)
    #print(DoL[column])
util.print_data(DoL, nlines=20)
```

```
{
  "code": [
    "CAN",
    "CHN",
    "IND",
    "RUS",
    "USA",
    "VNM"
  ],
  "country": [
    "Canada",
    "China",
    "India",
    "Russia",
    "United States",
    "Vietnam"
  ],
  "pop": [
    "36.26",
    "1378.66",
```

It's straightforward to create a Pandas dataframe from `DoL`.

Study and run the code cell below.

In [15]:

```
df = pd.DataFrame(DoL)

df.set_index('code', inplace=True)

df
```

Out[15]:

	country	pop	gdp	life	cell
code					
CAN	Canada	36.26	1535.77	82.3	30.75
CHN	China	1378.66	11199.15	76.25	1364.93

code	country	pop	gdp	life	cell
IND	India	1324.17	2263.79	68.56	1127.81
RUS	Russia	144.34	1283.16	71.59	229.13
USA	United States	323.13	18624.47	78.69	395.88
VNM	Vietnam	94.57	205.28	76.25	120.6

If a web page were to have multiple tables contained within the body of the HTML, our discovery step would need to be more involved, possibly printing parts of the structure of multiple tables returned from the `nodeset = root1.xpath("/html/body/div//table")` and determining which one contained the desired data.

Many other variations beyond a simple table like this one are possible.

Section 22.2.3: Wikipedia Table

Consider the following common scenario: a client application developer is seeking data to complement other parts of their application. For instance, they may find they need data on the latest estimated population of each of the states in the United States.

While the data may be available from a number of open data sources, (`census.gov` , for instance), the developer finds the data they want on Wikipedia:https://en.m.wikipedia.org/wiki/List_of_states_and_territories_of_the_United_States_by_popula

Take a moment and check out the table of population of the states on the Wikipedia web page.

The subset is in terms of both dimensions of the columns in the table as well as rows in the table.

Goal

We see in the rendered page a table of state populations. Population data and ranks are relative to the 2010 and 2021 census. In our case, we are interested in the most recent data, and so we want to extract the current rank (as an integer), the string of the name of the state (we do not care about the state flag picture), and the population as of July 1, 2021. These are the first, third, and fourth columns in the table.

Before engaging in web scraping, a developer must look at the acceptable use policy of a provider and also look at their policy on automatic access to their pages. In this case, such an investigation revealed that programmatic access should go through a defined API and that requests can be limited in number and frequency. However, we will work directly from the webpage.

Discovery

Below we obtain and parse the tree for the page giving the population by state data set.

Study and run the code cell below.

In [19]:

```
url = "https://en.m.wikipedia.org/wiki/List_of_U.S._states_and_territories_by_populatio

response = requests.get(url)
assert response.status_code == 200

tree2 = etree.parse(io.BytesIO(response.content), htmlparser)
root2 = tree2.getroot()
```

In this case, we suspect there are multiple table HTML elements in the page, so we use XPath to obtain the collection and delve into the tables to determine the correct one.

Study and run the code cell below.

In [20]:

```
table_list = root2.xpath("/html/body//table")
print(len(table_list))
```

```
3
```

If there are multiple tables, we need to discover which table carries the data we desire. We happen to know that a characteristic of data-carrying Wikipedia tables is that they are sortable and carry an xml class attribute with a "wikitable sortable plainrowheaders" value.

Study and run the code cell below.

In [21]:

```
xpath = "/html/body//table[@class='wikitable sortable plainrowheaders']"
table_list = root2.xpath(xpath)

print(len(table_list))
```

```
1
```

Note that there is only 1 table (which is different from our textbook as the Wikipedia page has been updated since the textbook was published). Let's go head and examine this table a little more closely.

Study and run the code cell below.

In [22]:

```
table = table_list[0]
util.print_xml(table, depth=4, nchild=3)
```

```
<table class='wikitable sortable plainrowheaders' style='wi
  <tbody>
    <tr style='vertical-align:middle'>
      <th colspan='2'>Rank</th>
      <th rowspan='2'>State or territory</th>
```

```
      <th colspan='2'>
        <div style='display:inline-block; line-height:1.2em
        <sup ...>
        </sup>
        <sup ...>
        </sup>
      </th>
       ...
    </tr>
    <tr>
      <th>2021</th>
      <th>2010</th>
      <th>
        <div style='display:inline-block; line-height:1.2em
      </th>
       ...
    </tr>
    <tr>
      <td style='text-align:center'>
        <span ...>1</span>
      </td>
      <td style='text-align:center'>
        <span ...>1</span>
      </td>
      <th scope='row' style='text-align:left'>
        <span ...>
        </span>
        <a href='/wiki/California' title='California'>Calif
      </th>
       ...
    </tr>
     ...
  </tbody>
</table>
```

The print of part of the tree above shows us that there is a `tbody` child of `table` present, but, in this case, no `thead`. Looking more closely at the first two `tr` child nodes of `tbody`, we see that they are populated with `th` elements, and these correspond to the two rows of the header of the table. The data-carrying rows begin with the third `tr` child of the `tbody`. To continue our discovery, we use XPath relative to the table to obtain the third row entry and examine its structure.

Study and run the code cell below.

In [23]:

```
rowlist = table.xpath(".//tr[3]")
datarow = rowlist[0]

util.print_xml(datarow, depth=3, nchild=5, width=300)
```

```
<tr>
  <td style='text-align:center'>
    <span ...>1</span>
  </td>
  <td style='text-align:center'>
    <span ...>1</span>
  </td>
  <th scope='row' style='text-align:left'>
    <span ...>
      <noscript>
```

```
      </noscript>
      <span ...></span>
    </span>
    <a href='/wiki/California' title='California'>California</a>
  </th>
  <td>39,237,836</td>
  <td>37,253,956</td>
   ...
</tr>
```

The data for this row of the table is contained in the `td` elements, and this first row corresponds to California, whose first and second columns have value 1, and the third field has a picture and a link whose label is California.

We observe:

1. The current state rank is in first `td` in row and is the text of a `span` node under the `td` Element. It is not always the case that the first visible table entry corresponds to the first `td` node in a row. Some tables can use additional `td` elements in the set of rows, used for spacing, borders, and other rendering results.

2. The state rank in 2010 is in the second `td` in the row; we will disregard this field based on our goals of wanting the most recent estimate.

3. The third `td` contains the state information, and we will explore that further below.

4. The fourth `td` contains the population in 2021, and here the value is directly in the text of the `td` element.

So we will use *positioning* to get the columns (as the relative `td` within the row) that we are interested in. Focusing on the data cell where we find the state information--**study and run the code cell below.**

In [24]:

```
util.print_xml(datarow[2], depth=3, nchild=4, width=300)
```

```
<th scope='row' style='text-align:left'>
  <span ...>
    <noscript>
      <img alt='' src='//upload.wikimedia.org/wikipedia/commons/thumb/0/01/Flag_of_Calif
ornia.svg/23px-Flag_of_California.svg.png' decoding='async' width='23' height='15' class
='thumbborder' data-file-width='900' data-file-height='600'></img>
    </noscript>
    <span ...></span>
  </span>
  <a href='/wiki/California' title='California'>California</a>
</th>
```

Data Extraction

Understanding the tree structure of the table, we can now acquire the data for columns 1, 3, and 4 of the table using XPath.

We know that the data-carrying rows begin after row position 2, and because the table has a row for the District of Columbia, we want to extract data from 51 rows. We do not want to go further, as these rows contain information on territories and have aggregate data.

Extracting column 1 from rows 3 through 53, and finding the data in the text property of a `span` underneath the appropriate `td`, we show the first four entries of the 51 extracted.

Study and run the code cell below.

In [25]:
```
xpath = ".//tr[position() > 2 and position() < 54]/td[1]/span/text()"
rank_column = table.xpath(xpath)
rank_column = [int(s) for s in rank_column]
rank_column[:4]
```

Out[25]:
```
[1, 2, 3, 4]
```

Extracting the state strings from column 3 from rows 3 through 53, by traversing into the a node and extracting the text property, we again show the first four entries of the 51.

Study and run the code cell below.

In [26]:
```
xpath = ".//tr[position() > 2 and position() < 54]/th[1]//a/text()"
state_column = table.xpath(xpath)
state_column[:4]
```

Out[26]:
```
['California', 'Texas', 'Florida', 'New York']
```

Finally, for the population, we need to do a little work to convert a comma-separated digit string into an integer, so we define a lambda function to perform the conversion and then, after we use XPath to extract a vector of strings, use a list comprehension to apply the conversion.

Study and run the code cell below.

In [27]:
```
convert_pop = lambda p: int(p.replace(",", ""))
xpath = ".//tr[position() > 2 and position() < 54]/td[3]/text()"
pop_column = table.xpath(xpath)
pop_column = [convert_pop(p) for p in pop_column]

pop_column[:4]
```

Out[27]:
```
[39237836, 29527941, 21781128, 19835913]
```

Now that we have the three desired columns, and we can define a dictionary of column list representation and construct the data frame:

In [28]:
```
DoL = {'rank': rank_column, 'state': state_column, 'population': pop_column}

df = pd.DataFrame(DoL)
df.set_index('rank', inplace=True)

df.head(10)
```

Out[28]:

rank	state	population
1	California	39237836
2	Texas	29527941
3	Florida	21781128
4	New York	19835913
5	Pennsylvania	12964056
6	Illinois	12671469
7	Ohio	11780017
8	Georgia	10799566
9	North Carolina	10551162
10	Michigan	10050811

Exercise 1:

Suppose we wish to scrap the data from the following website:

https://datasystems.denison.edu/topnames.html

Go to the above web page using Chrome, navigate to `View->View Source` and also `View->Developer->Inspect Elements` to become fimilar with the layout.

Exercise 2:

Write code to scrape this webpage and create a Pandas data frame from the table. Set the index so that it is a multi-index with `year` and `sex`. Print the first 10 lines of the dataframe. In particular, you should follow the same basic steps as the first example in this notebook:

- Use `requests.get()` to obtain a `response` object
- Use the raw bytes version of the `response` to obtain a parse tree
- Set `topnamesRoot` to be the root of the tree
- Use an XPath to navigate to the `table` element and assign it to `table`
- Use an XPath to find the column names of the table and assign them to `column_names`
- Construct a dictionary of column lists (DoL) representation using a for-loop and an appropriate XPath expression
- Create the Pandas data frame from the DoL
- Set the index to be `sex`, `year`

The output from my solution is as follows.

```
                     name  count
year sex
2005 Female     Emily  23940
```

```
     Male       Jacob  25833
2006 Female     Emily  21404
     Male       Jacob  24845
2007 Female     Emily  19355
     Male       Jacob  24282
2008 Female      Emma  18813
     Male       Jacob  22594
2009 Female  Isabella  22306
     Male       Jacob  21175
```

In [30]:

```
url = "https://datasystems.denison.edu/topnames.html"

response = requests.get(url)

assert response.status_code == 200

ind2016Tree = etree.parse(io.BytesIO(response.content), htmlparser)
ind2016Root = ind2016Tree.getroot()
```

In [31]:

```
table_list = ind2016Root.xpath("/html/body//table")
print(len(table_list))
```

```
1
```

In [32]:

```
table = table_list[0]
util.print_xml(table, depth=5, nchild=4)
```

```
<table class='table table-dark' style='width: 600px'>
  <thead>
    <tr>
      <th title='Field #1'>year</th>
      <th title='Field #2'>sex</th>
      <th title='Field #3'>name</th>
      <th title='Field #4'>count</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td align='right'>2005</td>
      <td>Female</td>
      <td>Emily</td>
      <td align='right'>23940</td>
    </tr>
    <tr>
      <td align='right'>2005</td>
      <td>Male</td>
      <td>Jacob</td>
      <td align='right'>25833</td>
    </tr>
    <tr>
      <td align='right'>2006</td>
      <td>Female</td>
      <td>Emily</td>
      <td align='right'>21404</td>
```

```
    </tr>
    <tr>
      <td align='right'>2006</td>
      <td>Male</td>
      <td>Jacob</td>
      <td align='right'>24845</td>
    </tr>
     ...
  </tbody>
</table>
```

In [33]:

```
column_names = table.xpath("./thead/tr/th/text()")
column_names
```

Out[33]:

```
['year', 'sex', 'name', 'count']
```

In [34]:

```
DoL = {}

for index, column in enumerate(column_names):
    xpath = ".//tr/td[{}]/text()".format(index+1)
    DoL[column] = table.xpath(xpath)

util.print_data(DoL, nlines=20)
```

```
{
  "year": [
    "2005",
    "2005",
    "2006",
    "2006",
    "2007",
    "2007",
    "2008",
    "2008",
    "2009",
    "2009",
    "2010",
    "2010",
    "2011",
    "2011",
    "2012",
    "2012",
    "2013",
    "2013",
```

In [35]:

```
df = pd.DataFrame(DoL)

df.set_index(['year','sex'], inplace=True)
print(df.head(10))
```

```
                 name  count
year sex
2005 Female     Emily  23940
     Male       Jacob  25833
2006 Female     Emily  21404
     Male       Jacob  24845
2007 Female     Emily  19355
```

```
     Male       Jacob  24282
2008 Female      Emma  18813
     Male       Jacob  22594
2009 Female  Isabella  22306
     Male       Jacob  21175
```

In [ ]:

```

```

Chapter 23

RESTful Application Programming Interfaces

Motivation

- Most of the time, data analytics professionals work with data that comes from their employer's internal systems.
- However, it is common to utilize third party data in order to enrich an analysis.
- Many companies and organizations offer their data through an Application Programming Interface (API): weather data, stock prices, crime statistics, etc.
- APIs allow developers and end users to access data in a secure and standardized way.

Why Offer Data Through an API?

- **Open Data:** Often collected directly by local, regional, and national governments or organizations and are provided openly as a public service.
- **Service for Profit:** Some providers are purveyors of data, which they gather and curate and then, charge for access to the data.

Commonalities with Web Scraping

- **Intention:** In web scraping, web pages are designed to be viewed primarily in browsers, and data extraction is often a repurposing of the data contained therein. In API services, the provider is intending to provide the data itself.
- **Format:** In web scraping, the format is HTML, which was not intended for data exchange. In API services, the format is consistent with the various data models covered during the semester: CSV, JSON, and XML.

Web Scraping vs APIs

- When web scraping, the acquired HTML is often static, already consisting of a textual HTML file in the file system of the web server.
- In API service providers, the data is often dynamic, being generated on the fly based on a request.
- In this regard, API service providers are more akin to the relational database systems that, in response to an SQL request, dynamically access the back end and respond with the requested data.
- **Whenever there is a choice, an application should use an API rather than performing web scraping.**

REST APIs

- REST stands for "Representational State Transfer."
- REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP (which is a protocol for fetching resources such as HTML documents).
- One of the guiding principals is that the API is ***stateless***, which means that a data request is "packaged together" in a unit, so the functionality is provided independent of any prior interaction

HTTP

- Recall that HTTP stands for **Hyper Text Transfer Protocol** and forms the foundation of the World Wide Web.

HTTP Request / Response

- Communication between clients and servers is done by ***requests*** and ***responses***:

  1. A client (a browser) sends an **HTTP request** to the web
  2. A web server receives the request
  3. The server runs an application to process the request
  4. The server returns an **HTTP response** (output) to the browser
  5. The client (the browser) receives the response

HTTP for REST API Requests

- RESTful APIs use the existing application level protocol of HTTP for making their requests and use the HTTP response message for carrying the outcome/status of the request and, if successful, the data resulting from the request.

The Movie Database (TMDB)

- The Movie Database (TMDB) is a community-built movie and TV database. Every piece of data has been added by the community dating back to 2008.

  https://www.themoviedb.org/

- Millions of people use the service daily

API Documentation

- Before accessing an API, the first step is to read the API documentation.
- This step is crucial because in order to know what data to pull, how to pull it, and how to unpack it, you need some understanding of the API and how it is structured.
- TMDB API documentation can be found here:

https://developers.themoviedb.org/3/getting-started/introduction

API Key

- As with most APIs, the TMDB APIs are not available for anonymous access but require requests to identify themselves as belonging to a particular application that has been previously registered.
- In the TMDB database, all accesses require the request to provide an authenticating API key.
- The API key is obtained by an application developer registering as a TMDB user, complete with a username and password, on the TMDB site.
- The resultant API key is then integrated into the application, usually by storing the key in a file or database so that the application can securely obtain the key when needed.

Get Movie Detail

- In order to find out whether a movie belongs to a collection using the TMDB API, we need to use the "get movie detail" query to pull the movie detail for each movie.
- The documentation for this query can be found here:

https://developers.themoviedb.org/3/movies/get-movie-details

- The basic query looks like this:

```
https://api.themoviedb.org/3/movie/{movie_id
}?api_key=<<api_key>>&language=en-US
```

Query Movie Information by ID

- By substituting “API key” below with your API key (“1a2b3c4d5e” for demo purposes) and providing a movie id of interest (using “414906”, The Batman, as an example here), we can form an API query like this:

```
https://api.themoviedb.org/3/movie/
414906?api_key=1a2b3c4d5e&language=en-US
```

Understanding the Result


```
{"adult":false,"backdrop_path":"/tRS6jvPM9qPrrnx2KRp3ew96Yot.jpg","bel
ongs_to_collection":{"id":948485,"name":"The Batman
Collection","poster_path":"/tuzKA9K5Ae9IzaA0R9oAgbyhAI3.jpg","backdrop
_path":"/qS2ViQwlWUECiAdkIuEaJZoq0QW.jpg"},"budget":185000000,"genres"
:[{"id":80,"name":"Crime"},{"id":9648,"name":"Mystery"},{"id":53,"name
":"Thriller"}],"homepage":"https://www.thebatman.com","id":414906,"imd
b_id":"tt1877830","original_language":"en","original_title":"The
Batman","overview":"In his second year of fighting crime, Batman
uncovers corruption in Gotham City that connects to his own family
while facing a serial killer known as the
Riddler.","popularity":9937.303,"poster_path":"/74xTEgt7R36Fpooo50r9T2
5onhq.jpg","production_companies":[{"id":101405,"logo_path":null,"name
":"6th &
Idaho","origin_country":"US"},{"id":119245,"logo_path":null,"name":"Dy
lan Clark
Productions","origin_country":"US"},{"id":128064,"logo_path":"/13F3Jf7
EFAcREU0xzZqJnVnyGXu.png","name":"DC
Films","origin_country":"US"},{"id":174,"logo_path":"/IuAlhI9eVC9Z8UQW
OIDdWRKSEJ.png","name":"Warner Bros.
Pictures","origin_country":"US"}],"production_countries":[{"iso_3166_1
":"US","name":"United States of America"}],"release_date":"2022-03-
01","revenue":759793000,"runtime":176,"spoken_languages":[{"english_na
me":"English","iso_639_1":"en","name":"English"}],"status":"Released",
"tagline":"Unmask the truth.","title":"The
Batman","video":false,"vote_average":7.8,"vote_count":4129}
```

Display JSON with `util`

- Suppose we save this output to `batman.json`.
- We can display the information in a more readably format using `util.print_data`:

```
with open("batman.json") as f:
    out = json.load(f)

util.print_data(out, nlines=40)
```


```
{
 "adult": false,
 "backdrop_path": "/tRS6jvPM9qPrrnx2KRp3ew96Yot.jpg",
 "belongs_to_collection": {
  "id": 948485,
  "name": "The Batman Collection",
  "poster_path": "/tuzKA9K5Ae9IzaA0R9oAgbyhAI3.jpg",
  "backdrop_path": "/qS2ViQwlWUECiAdkIuEaJZoq0QW.jpg"
 },
 "budget": 185000000,
 "genres": [
  {
   "id": 80,
   "name": "Crime"
  },
  {
   "id": 9648,
   "name": "Mystery"
  },
   …
 ],
 "homepage": "https://www.thebatman.com",
 "id": 414906,
 "imdb_id": "tt1877830",
 "original_language": "en",
 "original_title": "The Batman",
 "overview": "In his second year of fighting crime, Batm...
 "popularity": 9937.303,
  …
```

Search for Movies

https://developers.themoviedb.org/3/search/search-movies

creds.json

```
{
  "tmdb":
    { "protocol": "https",
       "location": "api.themoviedb.org",
       "apikey": "your_key_here"
    }
}
```

TMDB API Examples (Chapter 23)

DATA 200 - Database Systems and Data Management for Data Analytics
Dickinson College

Run the code cell below to import the needed libraries (make sure that `util.py` is in the same directory as this notebook).

In [1]:
```
import pandas as pd
import util
import requests
```

Please be sure to obtain your own API key from TMDB and update the `creds.json` to include it before running the code cell below. Note that `creds.json` should be in the same directory as this notebook.

In [2]:
```
datadir="."

# Read in the credentials
tmdb_creds = util.read_creds("tmdb", datadir)
protocol = tmdb_creds['protocol']
location = tmdb_creds['location']
apikey = tmdb_creds['apikey']
```

Study and run the code cell below which defines a function that performs a query to find the current most popular movies.

In [3]:
```
def GetPopularMoviePage(apikey, page=1):
    """
    Use popular movies endpoint to request a page of the
    movies currently deemed popular by some metric.
    Parameters:
      apikey: previously obtained key for client from TMDB
      page: which page of data to request, default 1
    Return: data structure from JSON-formatted results
    """

    host = "api.themoviedb.org"
    resourcepath = "/3/movie/popular"
    url = util.buildURL(resourcepath, host, protocol='https')
    queryParams = {"api_key": apikey, "page": page}

    response = requests.get(url, params=queryParams)

    assert response.status_code == 200

    return response.json()
```

Run the code cell below which uses the `GetPopularMoviePage()` function and displays some of the results.

In [4]:
```
popmovies_page = GetPopularMoviePage(apikey)

util.print_data(popmovies_page, nchild=7, nlines=30)
```


```
{
  "page": 1
  "results":
  [
    {
      "adult": False
      "backdrop_path": "/tRS6jvPM9qPrrnx2KRp3ew96Yot.jpg"
      "genre_ids":
      [
        80
        9648
        53
      ]
      "id": 414906
      "original_language": "en"
      "original_title": "The Batman"
      "overview": "In his second year of fighting crime, ...
      ...
    }
    {
      "adult": False
      "backdrop_path": "/aEGiJJP91HsKVTEPy1HhmN0wRLm.jpg"
      "genre_ids":
      [
        28
        12
        10751
      ]
      "id": 335787
      "original_language": "en"
```


Study and run the code cell below that defines a function that produces a list of row dictionaries (LoD) representation from the JSON-parsed data structure of a request. We will later use this representation to construct a data frame.

In [5]:
```
def MovieResult2LoD(result, maxelements=None, include_adult=False):
    """
    Process the JSON-based result of a popular movie query
    and, for each element of the result, create a row
    in a LoD representation of the movie fields.
    Parameters:
      result: JSON-based data structure from popular movie query
      maxelements: limit on the number of results to process. If None, process all
      include_adult: boolean indicating whether or not results tagged as adult should b
    Return: LoD with rows of movies
    """

    assert isinstance(result, dict)
    assert 'results' in result

    LoD = []
    count = 0

    for movie in result['results']:
        if maxelements != None and count >= maxelements:
```

```
            break
        if movie['adult'] and not include_adult:
            break

        D = {}

        D['id'] = movie['id']
        D['title'] = movie['title']
        genre_list = movie['genre_ids']

        genres = ",".join([str(g) for g in genre_list])
        D['genres'] = genres
        D['popularity'] = movie['popularity']
        D['vote_count'] = movie['vote_count']
        D['vote_average'] = movie['vote_average']
        D['release_date'] = movie['release_date']

        LoD.append(D)

        count += 1

    return LoD
```

Run the code in which we test our the `MovieResult2LoD()` function.

In [6]:

```
LoD = MovieResult2LoD(popmovies_page)

util.print_data(LoD[0])
```

```
{
  "id": 414906,
  "title": "The Batman",
  "genres": "80,9648,53",
  "popularity": 9937.303,
  "vote_count": 4112,
  "vote_average": 7.8,
  "release_date": "2022-03-01"
}
```

Study and run the code cell below where we define a function that returns a data frame of information related to the current most popular movies.

In [7]:

```
def GetPopularMovies(apikey, num_movies=20):
    """
    Use the building block functions so that, based on
    num_movies, make a series of requests and process
    responses, aggregating into a composite LoD structure
    and finally converting to a data frame.
    Parameters:
      apikey: previously obtained key for client from TMDB
      num_movies: total number of movies to retrieve and process
    Return: DataFrame structure
    """

    Composite_LoD = []

    page = 1
    movies_left = num_movies
```

```
    more_pages = True

    while more_pages and movies_left > 0:
        movie_page = GetPopularMoviePage(apikey, page)

        if movie_page['page'] == movie_page['total_pages']:
            more_pages = False

        page_LoD = MovieResult2LoD(movie_page)
        Composite_LoD.extend(page_LoD)
        movies_left -= len(page_LoD)

        page += 1

    df = pd.DataFrame(Composite_LoD)

    return df
```

Run the code cell below to display the returned data frame.

In [8]:

```
movies = GetPopularMovies(apikey, num_movies=40)

movies
```

Out[8]:

	id	title	genres	popularity	vote_count	vote_average	release_date
0	414906	The Batman	80,9648,53	9937.303	4112	7.8	2022-03-01
1	335787	Uncharted	28,12,10751	10763.209	1513	7.2	2022-02-10
2	634649	Spider-Man: No Way Home	28,12,878	5465.394	12254	8.1	2021-12-15
3	508947	Turning Red	16,10751,35,14	4278.097	1963	7.5	2022-03-10
4	675353	Sonic the Hedgehog 2	28,878,35,10751	4429.903	772	7.6	2022-03-30
5	799876	The Outfit	18,53,80	3195.953	157	7.0	2022-02-25
6	829557	365 Days: This Day	10749,18	3624.453	229	6.1	2022-04-27
7	406759	Moonfall	28,12,878	2633.340	853	6.5	2022-02-03
8	606402	Yaksha: Ruthless Operations	28	2172.543	72	6.2	2022-04-08
9	568124	Encanto	16,35,10751,14	2019.249	6257	7.7	2021-11-24
10	294793	All the Old Knives	53,28	1988.087	183	6.0	2022-04-08
11	823625	Blacklight	28,53	1891.616	322	6.2	2022-02-10
12	919689	War of the Worlds: Annihilation	878,28	1774.735	37	5.6	2021-12-22
13	696806	The Adam Project	28,12,35,878	1607.314	2036	7.0	2022-03-11
14	676705	Pil's Adventures	12,16,35,10751,14	1491.474	73	6.7	2021-08-11
15	585083	Hotel Transylvania: Transformania	16,10751,14,35,12	1385.282	795	7.1	2022-02-25

	id	title	genres	popularity	vote_count	vote_average	release_date
16	661791	The Grandmother	27	1505.586	139	6.1	2022-01-28
17	760868	Black Crab	28,53	1266.217	387	6.3	2022-03-18
18	833425	No Exit	27,53	1368.762	408	6.7	2022-02-25
19	580489	Venom: Let There Be Carnage	878,28,12	1258.832	7135	7.0	2021-09-30
20	774825	The Ice Age Adventures of Buck Wild	16,35,12,10751	1192.965	1062	6.9	2022-01-28
21	454626	Sonic the Hedgehog	28,878,35,10751	1221.143	7884	7.4	2020-02-12
22	768744	My Hero Academia: World Heroes' Mission	16,28,14,12	1091.979	159	7.3	2021-08-06
23	760926	Gold	53,28	1168.732	220	6.4	2022-01-13
24	338953	Fantastic Beasts: The Secrets of Dumbledore	14,12	1180.999	820	6.8	2022-04-06
25	893370	Virus:32	27	1396.018	24	6.8	2022-04-21
26	438695	Sing 2	16,35,10751,10402	1151.318	2656	8.1	2021-12-01
27	526896	Morbius	28,878,14	903.198	558	5.9	2022-03-30
28	938596	Bull Shark	27	1140.657	14	4.9	2022-02-05
29	818750	The In Between	10749,878,18	942.559	231	7.0	2022-02-11
30	524434	Eternals	878,28	1095.699	5405	7.1	2021-11-03
31	453395	Doctor Strange in the Multiverse of Madness	14,28,12	880.919	17	6.2	2022-05-04
32	635302	Demon Slayer - Kimetsu no Yaiba- The Movie: Mug...	16,28,12,14	952.230	2383	8.4	2020-10-16
33	928381	Restless	28,53,80	989.369	237	5.9	2022-02-25
34	763285	Ambulance	28,80,53	1156.855	378	6.8	2022-03-16
35	939243	Sonic the Hedgehog 3	28,12,14	968.873	0	0.0	
36	803114	The Requin	53	916.972	104	4.8	2022-01-28
37	801071	The Jack in the Box: Awakening	27	895.580	38	6.1	2022-02-24
38	58	Pirates of the Caribbean: Dead Man's Chest	12,14,28	793.255	13339	7.3	2006-07-06
39	860623	Last Man Down	28,53	879.239	433	6.3	2021-10-19

In [ ]:

```

```