UNIT I

1.1 Benefits and uses of data science and big data

Data science and big data are used almost everywhere in both commercial and noncommercial settings. Commercial companies in almost every industry use data science and big data to gain insights into their customers, processes, staff, completion, and products. Many companies use data science to offer customers a better user experience, as well as to cross-sell, up-sell, and personalize their offerings. A good example of this is Google AdSense, which collects data from internet users so relevant commercial messages can be matched to the person browsing the internet. Human resource professionals use people analytics and text mining to screen candidates, monitor the mood of employees, and study informal networks among coworkers. People analytics is the central theme in the book *Moneyball: The Art of Winning an Unfair Game*. In the book (and movie) we saw that the traditional scouting process for American baseball was random, and replacing it with correlated signals changed everything. Relying on statistics allowed them to hire the right players and pit them against the opponents where they would have the biggest advantage. Financial institutions use data science to predict stock markets, determine the risk of lending money, and learn how to attract new clients for their services. Governmental organizations are also aware of data's value. Many governmental organizations not only rely on internal data scientists to discover valuable information, but also share their data with the public. You can use this data to gain insights or build data-driven applications. *Data.gov* is but one example; it's the home of the US Government's open data. A data scientist in a governmental organization gets to work on diverse projects such as detecting fraud and other criminal activity or optimizing project funding. well-known example was provided by Edward Snowden, who leaked internal documents of the American National Security Agency and the British Government Communications Headquarters that show clearly how they used data science and big data to monitor millions of individuals. Those organizations collected 5 billion data records from widespread applications such as Google Maps, Angry Birds, email, and text messages, among many other data sources. Nongovernmental organizations (NGOs) are also no strangers to using data. They use it to raise money and defend their causes. The World Wildlife Fund (WWF), for instance, employs data scientists to increase the effectiveness of their fundraising efforts. Universities use data science in their research but also to enhance the study experience of their students. The rise of massive open online courses (MOOC) produces a lot of data, which allows universities to study how this type of learning can complement traditional classes.

1.2 Facets of data

In data science and big data you'll come across many different types of data, and each of them tends to require different tools and techniques. The main categories of data are these:

- Structured
- Unstructured
- Natural language
- Machine-generated
- Graph-based
- Audio, video, and images
- Streaming

Structured data

Structured data is data that depends on a data model and resides in a fixed field within a record. As such, it's often easy to store structured data in tables within databases or Excel

AMSCE-IOT

files (figure 1.1). SQL, or Structured Query Language, is the preferred way to manage and query data that resides in databases. You may also come across structured data that might give you a hard time storing it in a traditional relational database. Hierarchical data such as a family tree is one such example.

1	Indicator ID	Dimension List	Timeframe	Numeric Value	Missing Value Flag	Confidence Inte
2	214390830	Total (Age-adjusted)	2008	74.6%		73.8%
3	214390833	Aged 18-44 years	2008	59.4%		58.0%
4	214390831	Aged 18-24 years	2008	37.4%		34.6%
5	214390832	Aged 25-44 years	2008	66.9%		65.5%
6	214390836	Aged 45-64 years	2008	88.6%		87.7%
7	214390834	Aged 45-54 years	2008	86.3%		85.1%
8	214390835	Aged 55-64 years	2008	91.5%		90.4%
9	214390840	Aged 65 years and over	2008	94.6%		93.8%
10	214390837	Aged 65-74 years	2008	93.6%		92.4%
11	214390838	Aged 75-84 years	2008	95.6%		94.4%
12	214390839	Aged 85 years and over	2008	96.0%		94.0%
13	214390841	Male (Age-adjusted)	2008	72.2%		71.1%
14	214390842	Female (Age-adjusted)	2008	76.8%		75.9%
15	214390843	White only (Age-adjusted)	2008	73.8%		72.9%
16	214390844	Black or African American only (Age-adjusted)	2008	77.0%		75.0%
17	214390845	American Indian or Alaska Native only (Age-adjusted)	2008	66.5%		57.1%
18	214390846	Asian only (Age-adjusted)	2008	80.5%		77.7%
19	214390847	Native Hawaiian or Other Pacific Islander only (Age-adjusted)	2008	DSU		
20	214390848	2 or more races (Age-adjusted)	2008	75.6%		69.6%

1.2.2 Unstructured data

Unstructured data is data that isn't easy to fit into a data model because the content is context-specific or varying. One example of unstructured data is your regular email (figure 1.2). Although email contains structured elements such as the sender, title, and body text, it's a challenge to find the number of people who have written an email complaint about a specific employee because so many ways exist to refer to a person, for example. The thousands of different languages and dialects out there further complicate this.

Delete Move Spam

New team of UI engineers

CDA@engineer.com
To xyz@program.com
Today 10:21

An investment banking client of mine has had the go ahead to build a new team of UI engineers to work on various areas of a cutting-edge single-dealer trading platform.

They will be recruiting at all levels and paying between 40k & 85k (+ all the usual benefits of the banking world). I understand you may not be looking. I also understand you may be a contractor. Of the last 3 hires they brought into the team, two were contractors of 10 years who I honestly thought would never turn to what they considered "the dark side."

This is a genuine opportunity to work in an environment that's built up for best in industry and allows you to gain commercial experience with all the latest tools, tech, and processes.

There is more information below. I appreciate the spec is rather loose – They are not looking for specialists in Angular / Node / Backbone or any of the other buzz words in particular, rather an "engineer" who can wear many hats and is in touch with current tech & tinkers in their own time.

For more information and a confidential chat, please drop me a reply email. Appreciate you may not have an updated CV, but if you do that would be handy to have a look through if you don't mind sending.

Reply Reply to All Forward

1.2.3 Natural language

Natural language is a special type of unstructured data; it's challenging to process because it requires knowledge of specific data science techniques and linguistics. The natural language processing community has had success in entity recognition, topic recognition, summarization, text completion, and sentiment analysis, but models trained in one domain don't generalize well to other domains. Even state-of-the-art techniques aren't able to decipher the meaning of every piece of text. This shouldn't be a surprise though: humans struggle with natural language as well. It's ambiguous by nature.

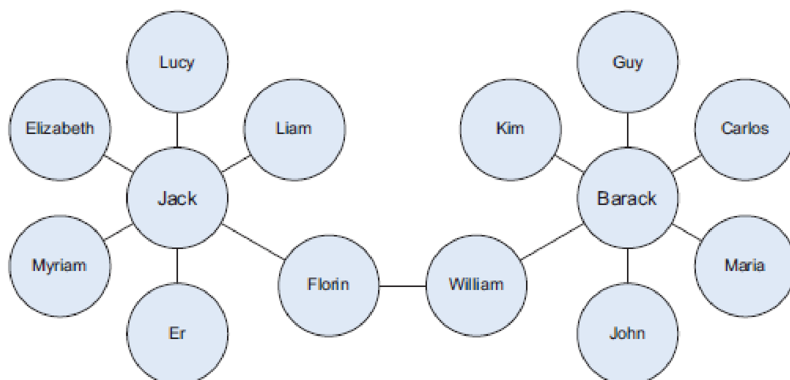
1.2.4 Machine-generated data

Machine-generated data is information that's automatically created by a computer, process, application, or other machine without human intervention. Machine-generated data is becoming a major data resource and will continue to do so. IDC (International Data Corporation) has estimated there will be 26 times more connected things than people in 2020. This network is commonly referred to as *the internet of things*. The analysis of machine data relies on highly scalable tools, due to its high volume and speed. The machine data shown in figure 1.3 would fit nicely in a classic table-structured database.

```
CSIPERF:TXCOMMIT;313236
2014-11-28 11:36:13, Info                  CSI    00000153 Creating NT transaction (seq
69), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000154 Created NT transaction (seq 69)
result 0x00000000, handle @0x4e54
2014-11-28 11:36:13, Info                  CSI    00000155@2014/11/28:10:36:13.471
Beginning NT transaction commit...
2014-11-28 11:36:13, Info                  CSI    00000156@2014/11/28:10:36:13.705 CSI perf
trace:
CSIPERF:TXCOMMIT;273983
2014-11-28 11:36:13, Info                  CSI    00000157 Creating NT transaction (seq
70), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000158 Created NT transaction (seq 70)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:13, Info                  CSI    00000159@2014/11/28:10:36:13.764
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015a@2014/11/28:10:36:14.094 CSI perf
trace:
CSIPERF:TXCOMMIT;386259
2014-11-28 11:36:14, Info                  CSI    0000015b Creating NT transaction (seq
71), objectname [6]"(null)"
2014-11-28 11:36:14, Info                  CSI    0000015c Created NT transaction (seq 71)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:14, Info                  CSI    0000015d@2014/11/28:10:36:14.106
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015e@2014/11/28:10:36:14.428 CSI perf
trace:
CSIPERF:TXCOMMIT;375581
```

1.2.5 Graph-based or network data

"Graph data" can be a confusing term because any data can be shown in a graph. "Graph" in this case points to mathematical *graph theory*. In graph theory, a graph is a mathematical structure to model pair-wise relationships between objects. Graph or network data is, in short, data that focuses on the relationship or adjacency of objects. The graph structures use nodes, edges, and properties to represent and store graphical data. Graph-based data is a natural way to represent social networks, and its structure allows you to calculate specific metrics such as the influence of a person and the shortest path between two people. Examples of graph-based data can be found on many social media websites.



Graph databases are used to store graph-based data and are queried with specialized query languages such as SPARQL.

1.2.6 Audio, image, and video

Audio, image, and video are data types that pose specific challenges to a data scientist. Tasks that are trivial for humans, such as recognizing objects in pictures, turn out to be challenging for computers. MLBAM (Major League Baseball Advanced Media) announced in 2014 that they'll increase video capture to approximately 7 TB per game for the purpose of live, in-game analytics. High-speed cameras at stadiums will capture ball and athlete movements to calculate in real time, for example, the path taken by a defender relative to two baselines. a company called DeepMind succeeded at creating an algorithm that's capable of learning how to play video games. This algorithm takes the video screen as input and learns to interpret everything via a complex process of deep learning. It's a remarkable feat that prompted Google to buy the company for their own Artificial Intelligence (AI) development plans.

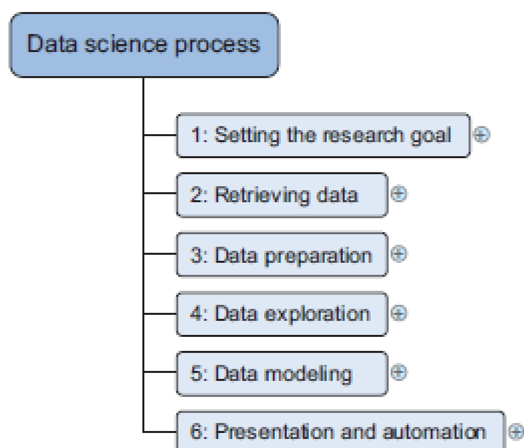
1.2.7 Streaming data

While streaming data can take almost any of the previous forms, it has an extra property. The data flows into the system when an event happens instead of being loaded into a data store in a batch. Although this isn't really a different type of data, Examples are the "What's trending" on Twitter, live sporting or music events, and the stock market.

MSCE-101

1.3 The data science process

The data science process typically consists of six steps.



1.3.1 Setting the research goal

Data science is mostly applied in the context of an organization. When the business asks you to perform a data science project, you'll first prepare a project charter. This charter contains information such as what

you're going to research, how the company benefits from that, what data and resources you need, a timetable, and deliverables.

1.3.2 Retrieving data

The second step is to collect data. You've stated in the project charter which data you need and where you can find it. In this step you ensure that you can use the data in your program, which means checking the existence of, quality, and access to the data. Data can also be delivered by third-party companies and takes many forms ranging from Excel spreadsheets to different types of databases.

1.3.3 Data preparation

Data collection is an error-prone process; in this phase you enhance the quality of the data and prepare it for use in subsequent steps. This phase consists of three subphases: *data cleansing* removes false values from a data source and inconsistencies across data sources, *data integration* enriches data sources by combining information from multiple data sources, and *data transformation* ensures that the data is in a suitable format for use in your models.

1.3.4 Data exploration

Data exploration is concerned with building a deeper understanding of your data. You try to understand how variables interact with each other, the distribution of the data, and whether there are outliers. To achieve this you mainly use descriptive statistics, visual techniques, and simple modelling. This step is also known as Exploratory Data Analysis.

1.3.5 Data modeling or model building

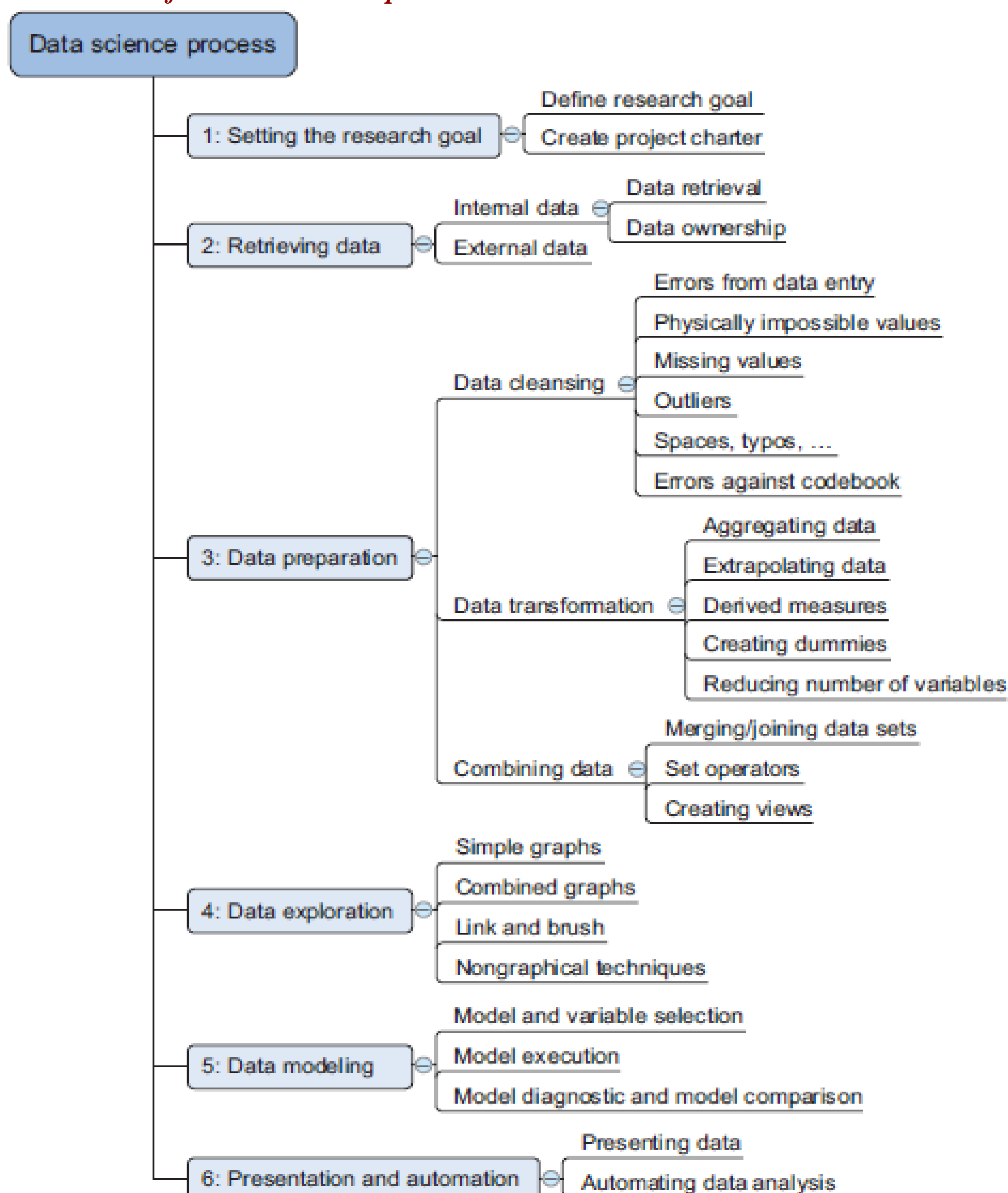
In this phase you use models, domain knowledge, and insights about the data you found in the previous steps to answer the research question. You select a technique from the fields of statistics, machine learning, operations research, and so on. Building a model is an iterative process that involves selecting the variables for the model, executing the model, and model diagnostics.

1.3.6 Presentation and automation

Finally, you present the results to your business. These results can take many forms, ranging from presentations to research reports. Sometimes you'll need to automate the execution of the process because the business will want to use the insights you gained in another project or enable an operational process to use the outcome from your model.

AMSCE-IT01

2.1 Overview of the data science process



1 The first step of this process is setting a *research goal*. The main purpose here is making sure all the stakeholders understand the *what*, *how*, and *why* of the project. In every serious project this will result in a project charter.
2 The second phase is *data retrieval*. You want to have data available for analysis, so this step includes finding suitable data and getting access to the data from the data owner. The result is data in its raw form, which probably needs polishing and transformation before it becomes usable.
3 Now that you have the raw data, it's time to *prepare* it. This includes transforming the data from a raw form into data that's directly usable in your models. To achieve this, you'll detect and correct different kinds of errors in the data, combine data from different data sources, and transform it. If you have successfully completed this step, you can progress to data visualization and modeling.

4 The fourth step is *data exploration*. The goal of this step is to gain a deep understanding of the data. You'll look for patterns, correlations, and deviations based on visual and descriptive techniques. The insights you gain from this phase will enable you to start modeling.
5 Finally, we get to the sexiest part: *model building* (often referred to as "data modeling" throughout this book). It is now that you attempt to gain the insights or make the predictions stated in your project charter. Now is the time to bring out the heavy guns, but remember research has taught us that often (but not always) a combination of simple models tends to outperform one complicated model. If you've done this phase right, you're almost done.
6 The last step of the data science model is *presenting your results and automating the analysis,* if needed. One goal of a project is to change a process and/or make better decisions. You may still need to convince the business that your findings will indeed change the business process as expected. This is where you can
shine in your influencer role. The importance of this step is more apparent in projects on a strategic and tactical level. Certain projects require you to perform the business process over and over again, so automating the project will save time.

In reality you won't progress in a linear way from step 1 to step 6. Often you'll regress and iterate between the different phases. This process ensures you have a well-defined research plan, a good understanding of the business question, and clear deliverables before you even start looking at data. The first steps of your process focus on getting high-quality data as input for your models. This way your models will perform better later on. In data science there's a well-known saying: *Garbage in equals garbage out.*

AMSCE-101

Step 1: Defining research goals and creating a project charter

A project starts by understanding the *what*, the *why*, and the *how* of your project (figure 2.2). What does the company expect you to do? And why does management place such a value on your research? Is it part of a bigger strategic picture or a "lone wolf" project originating from an opportunity someone detected? Answering these three questions (what, why, how) is the goal of the first phase, so that everybody knows what to do and can agree on the best course of action. The outcome should be a clear research goal, a good understanding of the context, well-defined deliverables, and a plan of action with a timetable. This information is then best placed in a project charter.

Spend time understanding the goals and context of your research

An essential outcome is the research goal that states the purpose of your assignment in a clear and focused manner. Understanding the business goals and context is critical for project success.

Create a project charter

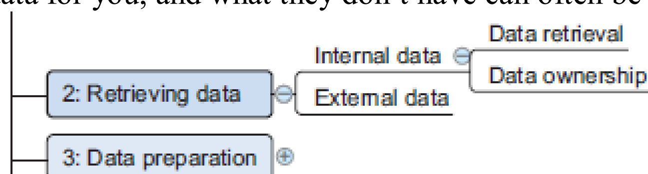
Clients like to know upfront what they're paying for, so after you have a good understanding of the business problem, try to get a formal agreement on the deliverables. All this information is best collected in a project charter

A project charter requires teamwork, and your input covers at least the following:
■ A clear research goal
■ The project mission and context
■ How you're going to perform your analysis
■ What resources you expect to use

■ Proof that it's an achievable project, or proof of concepts
■ Deliverables and a measure of success
■ A timeline

Step 2: Retrieving data

The next step in data science is to retrieve the required data (figure 2.3). Sometimes you need to go into the field and design a data collection process yourself, but most of the time you won't be involved in this step. Many companies will have already collected and stored the data for you, and what they don't have can often be bought from third parties



Data can be stored in many forms, ranging from simple text files to tables in a database. The objective now is acquiring all the data you need. This may be difficult, and even if you succeed, data is often like a diamond in the rough: it needs polishing to be of any use to you.

Start with data stored within the company

Your first act should be to assess the relevance and quality of the data that's readily available within your company. Most companies have a program for maintaining key data, so much of the cleaning work may already be done. This data can be stored in official data repositories such as databases, data marts, data warehouses, and data lakes maintained by a team of IT professionals. The primary goal of a database is data storage, while a data warehouse is designed for reading and analyzing that data.A data mart is a subset of the data warehouse and geared toward serving a specific business unit. While data warehouses and data marts are home to preprocessed data, data lakes contains data in its natural or raw format. But the possibility exists that your data still resides in Excel files on the desktop of a domain expert.

Don't be afraid to shop around

If data isn't available inside your organization, look outside your organization's walls. Many companies specialize in collecting valuable information. For instance, Nielsen and GFK are well known for this in the retail industry. Other companies provide data so that you, in turn, can enrich their services and ecosystem. Such is the case with Twitter, LinkedIn, and Facebook.

MSCE-1101

Table 2.1 A list of open-data providers that should get you started

Open data site	Description
Data.gov	The home of the US Government's open data
https://open-data.europa.eu/	The home of the European Commission's open data
Freebase.org	An open database that retrieves its information from sites like Wikipedia, MusicBrains, and the SEC archive
Data.worldbank.org	Open data initiative from the World Bank
Aiddata.org	Open data for international development
Open.fda.gov	Open data from the US Food and Drug Administration

Expect to spend a good portion of your project time doing data correction and cleansing, sometimes up to 80%. Most of the errors you'll encounter during the data gathering phase are easy to spot, but being too careless will make you spend many hours solving data issues that could have been prevented during data import. You'll investigate the data during the import, data preparation, and exploratory phases. During *data retrieval*, you check to see if the data is

equal to the data in the source document and look to see if you have the right data types. With *data preparation, The focus is on the content of the variables: you want to get rid of typos and other data entry errors and bring the data to a common standard among the data sets. For example, you might correct USQ to USA and United Kingdom to UK. During the exploratory phase your focus shifts to what you can learn from the data.*

Step 3: Cleansing, integrating, and transforming data

The data received from the data retrieval phase is likely to be "a diamond in the rough." Your task now is to prepare it for use in the modelling and reporting phase. Doing so is tremendously important because your models will perform better and you'll lose less time trying to fix strange output. Your model needs the data in a specific format, so data transformation will always come into play.

2.4.1 Cleansing data

Data cleansing is a subprocess of the data science process that focuses on removing errors in your data so your data becomes a true and consistent representation of the processes it originates from. By "true and consistent representation" we imply that at least two types of errors exist. The first type is the interpretation error, such as when you take the value in your data for granted, like saying that a person's age is greater than 300 years. The second type of error points to inconsistencies between data sources or against your company's standardized values. An example of this class of errors is putting "Female" in one table and "F" in another when they represent the same thing: that the person is female. Another example is that you use Pounds in one table and Dollars in another.

Table 2.2 An overview of common errors

General solution	
Try to fix the problem early in the data acquisition chain or else fix it in the program.	
Error description	**Possible solution**
Errors pointing to false values within one data set	
Mistakes during data entry	Manual overrules
Redundant white space	Use string functions
Impossible values	Manual overrules
Missing values	Remove observation or value
Outliers	Validate and, if erroneous, treat as missing value (remove or insert)
Errors pointing to inconsistencies between data sets	
Deviations from a code book	Match on keys or else use manual overrules
Different units of measurement	Recalculate
Different levels of aggregation	Bring to same level of measurement by aggregation or extrapolation

Sometimes you'll use more advanced methods, such as simple modeling, to find and identify data errors; We do a regression to get acquainted with the data and detect the influence of individual observations on the regression line. When a single observation has too much influence, this can point to an error in the data, but it can also be a valid point.

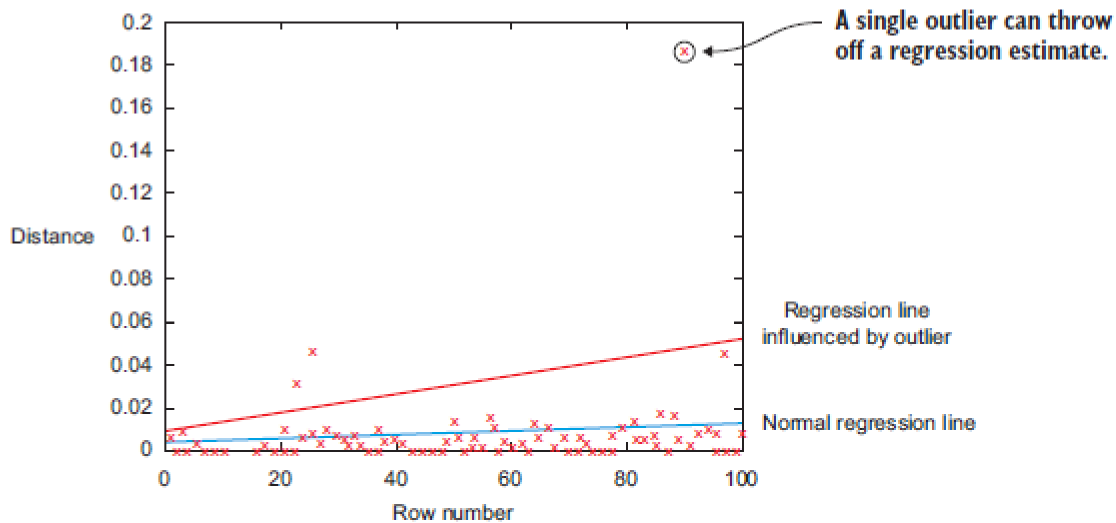


Figure 2.5 The encircled point influences the model heavily and is worth investigating because it can point to a region where you don't have enough data or might indicate an error in the data, but it also can be a valid data point.

DATA ENTRY ERRORS

Data collection and data entry are error-prone processes. They often require human intervention, and because humans are only human, they make typos or lose their concentration for a second and introduce an error into the chain. But data collected by machines or computers isn't free from errors either. For small data sets you can check every value by hand. Detecting data errors when the variables you study don't have many classes can be done by tabulating the data with counts - frequency table.

Table 2.3 Detecting outliers on simple variables with a frequency table

Value	Count
Good	1598647
Bad	1354468
Godo	15
Bade	1

Most errors of this type are easy to fix with simple assignment statements and if-then else rules:

```
if x == "Godo":
x = "Good"
if x == "Bade":
x = "Bad"
```

REDUNDANT WHITESPACE

Whitespaces tend to be hard to detect but cause errors like other redundant characters would. The cleaning during the ETL phase wasn't well executed, and keys in one table contained a whitespace at the end of a string. This caused a mismatch of keys such as "FR " – "FR",

dropping the observations that couldn't be matched. strip() function to remove leading and trailing spaces.

FIXING CAPITAL LETTER MISMATCHES

Capital letter mismatches are common. Most programming languages make a distinction between "Brazil" and "brazil".

you can solve the problem by applying a function that returns
both strings in lowercase, such as .lower() in Python. "Brazil".lower() == "brazil".lower() should result in true.

IMPOSSIBLE VALUES AND SANITY CHECKS

Sanity checks are another valuable type of data check. Here you check the value against physically or theoretically impossible values.
Sanity checks can be directly expressed with rules:
check = 0 <= age <= 120

OUTLIERS

An outlier is an observation that seems to be distant from other observations or, more specifically, one observation that follows a different logic or generative process than the other observations. The easiest way to find outliers is to use a plot or a table with the minimum and maximum values. An example is shown in figure 2.6. The plot on the top shows no outliers, whereas the plot on the bottom shows possible outliers on the upper side when a normal distribution is expected. The high values in the bottom graph can point to outliers when assuming a normal distribution.

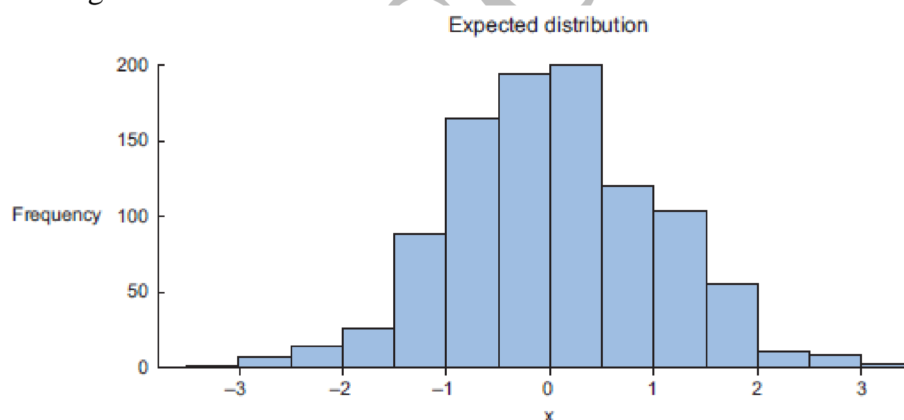


Figure 2.6 Distribution plots are helpful in detecting outliers and helping you understand the variable.

DEALING WITH MISSING VALUES

101

Missing values aren't necessarily wrong, but you still need to handle them separately; certain modeling techniques can't handle missing values.

Table 2.4 An overview of techniques to handle missing data

Technique	Advantage	Disadvantage
Omit the values	Easy to perform	You lose the information from an observation
Set value to `null`	Easy to perform	Not every modeling technique and/or implementation can handle `null` values
Impute a static value such as 0 or the mean	Easy to perform You don't lose information from the other variables in the observation	Can lead to false estimations from a model
Impute a value from an estimated or theoretical distribution	Does not disturb the model as much	Harder to execute You make data assumptions
Modeling the value (nondependent)	Does not disturb the model too much	Can lead to too much confidence in the model Can artificially raise dependence among the variables Harder to execute You make data assumptions

DEVIATIONS FROM A CODE BOOK

A code book is a description of your data, a form of metadata. It contains things such as the number of variables per observation, the number of observations, and what each encoding within a variable means. Example University exam Marksheet.

A code book also tells the type of data you're looking at: is it hierarchical, graph, something else?

DIFFERENT UNITS OF MEASUREMENT

When integrating two data sets, you have to pay attention to their respective units of measurement. sets can contain prices per gallon and others can contain prices per liter. A simple conversion will do the trick in this case.

DIFFERENT LEVELS OF AGGREGATION

Having different levels of aggregation is similar to having different types of measurement.

An example of this would be a data set containing data per week versus one containing data per work week.

2.4.2 Correct errors as early as possible

(i) Decision-makers may make costly mistakes on information based on incorrect data from applications that fail to correct for the faulty data.

(ii) If errors are not corrected early on in the process, the cleansing will have to be done for every project that uses that data.

(iii) Data errors may point to a business process that isn't working as designed.

(iv) Data errors may point to defective equipment, such as broken transmission lines and defective sensors.

(v) Data errors can point to bugs in software or in the integration of software that may be critical to the company.

2.4.3 Combining data from different data sources

Your data comes from several different places, and in this substep we focus on integrating these different sources.

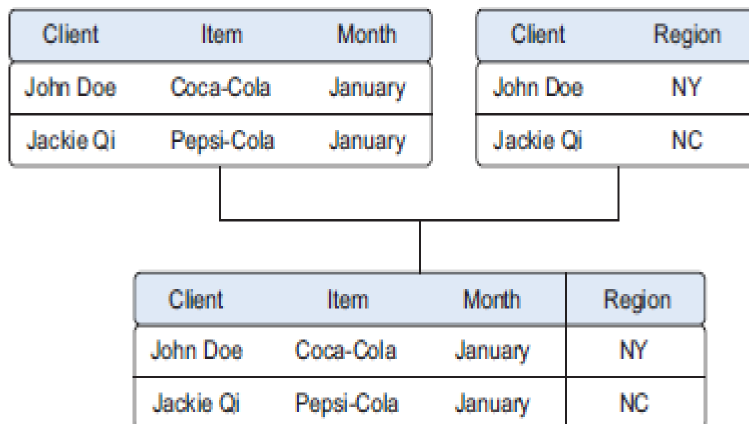
You can perform two operations to combine information from different data sets.

(i) The first operation is *joining*: enriching an observation from one table with information from another table.

(ii) The second operation is *appending* or *stacking*: adding the observations of one table to those of another table.

Joining Tables:

SCE-1101



Client	Item	Month
John Doe	Coca-Cola	January
Jackie Qi	Pepsi-Cola	January

Client	Region
John Doe	NY
Jackie Qi	NC

Client	Item	Month	Region
John Doe	Coca-Cola	January	NY
Jackie Qi	Pepsi-Cola	January	NC

Figure 2.7 Joining two tables on the Item and Region keys

APPENDING TABLES

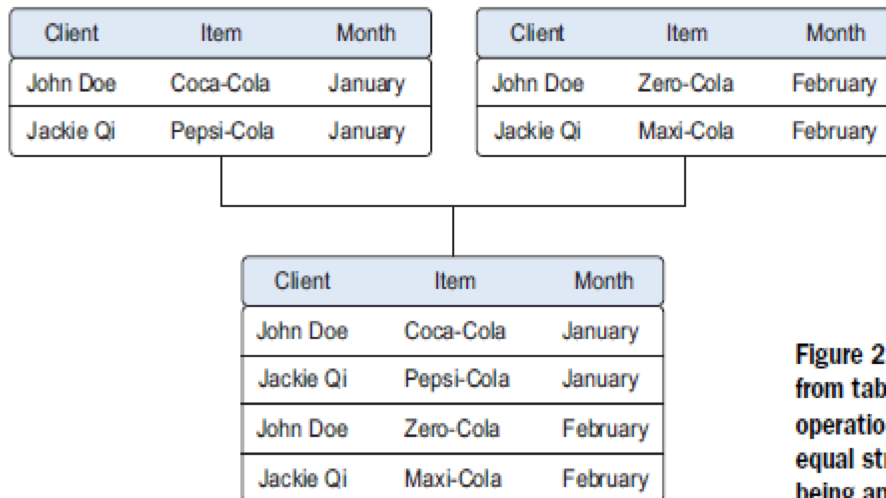


Figure 2.8 Appending data from tables is a common operation but requires an equal structure in the tables being appended.

USING VIEWS TO SIMULATE DATA JOINS AND APPENDS

To avoid duplication of data, you virtually combine data with views. In the previous example we took the monthly data and combined it in a new physical table.

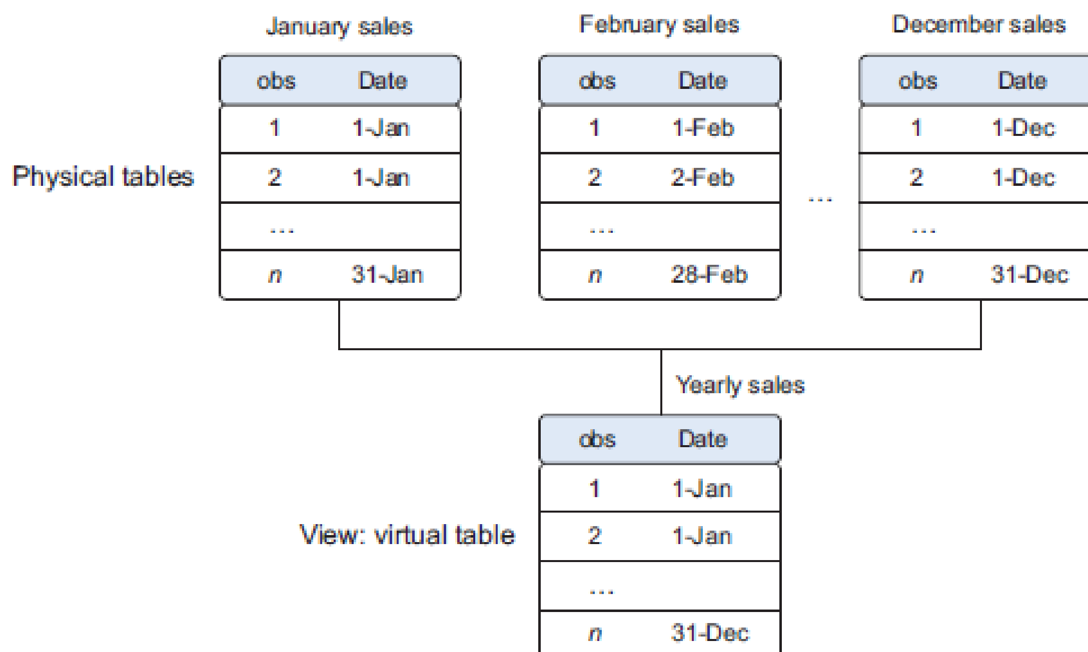


Figure 2.9 A view helps you combine data without replication.

2.4.4 Transforming data

Certain models require their data to be in a certain shape. Transforming your data so it takes a suitable form for data modelling.

TRANSFORMING DATA

Relationships between an input variable and an output variable aren't always linear. Take, for instance, a relationship of the form $y = ae^{bx}$. Taking the log of the independent variables simplifies the estimation problem dramatically.

x	1	2	3	4	5	6	7	8	9	10
log(x)	0.00	0.43	0.68	0.86	1.00	1.11	1.21	1.29	1.37	1.43
y	0.00	0.44	0.69	0.87	1.02	1.11	1.24	1.32	1.38	1.46

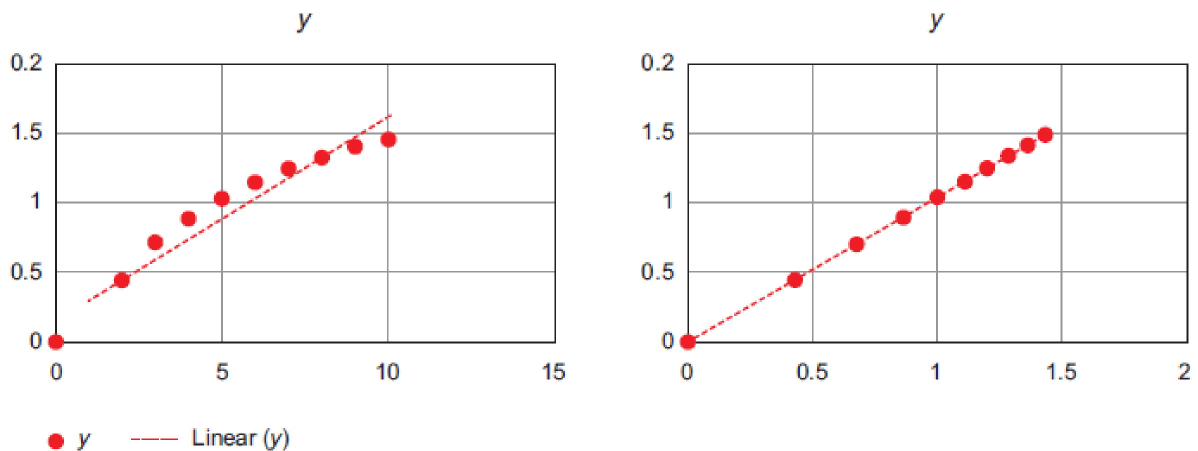


Figure 2.11 Transforming x to log x makes the relationship between x and y linear (right), compared with the non-log x (left).

REDUCING THE NUMBER OF VARIABLES

Sometimes you have too many variables and need to reduce the number because they don't add new information to the model. Having too many variables in your model makes the model difficult to handle, and certain techniques don't perform well when you overload them with too many input variables. For instance, all the techniques based on a Euclidean distance perform well only up to 10 variables.

TURNING VARIABLES INTO DUMMIES

Variables can be turned into dummy variables (figure 2.13). Dummy variables can only take two values: true(1) or false(0). They're used to indicate the absence of a categorical effect that may explain the observation.

AMSCE -

Customer	Year	Gender	Sales
1	2015	F	10
2	2015	M	8
1	2016	F	11
3	2016	M	12
4	2017	F	14
3	2017	M	13

M F

Customer	Year	Sales	Male	Female
1	2015	10	0	1
1	2016	11	0	1
2	2015	8	1	0
3	2016	12	1	0
3	2017	13	1	0
4	2017	14	0	1

Figure 2.13 Turning variables into dummies is a data transformation that breaks a variable that has multiple classes into multiple variables, each having only two possible values: 0 or 1.

2.5 Step 4: Exploratory data analysis

Information becomes much easier to grasp when shown in a picture, therefore you mainly use graphical techniques to gain an understanding of your data and the interactions between variables.

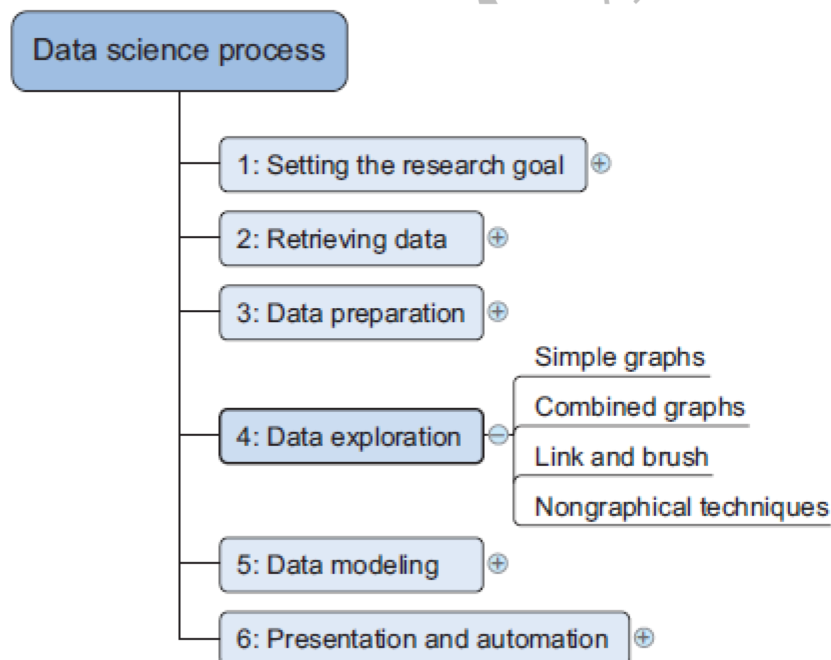


Figure 2.14 Step 4: Data exploration

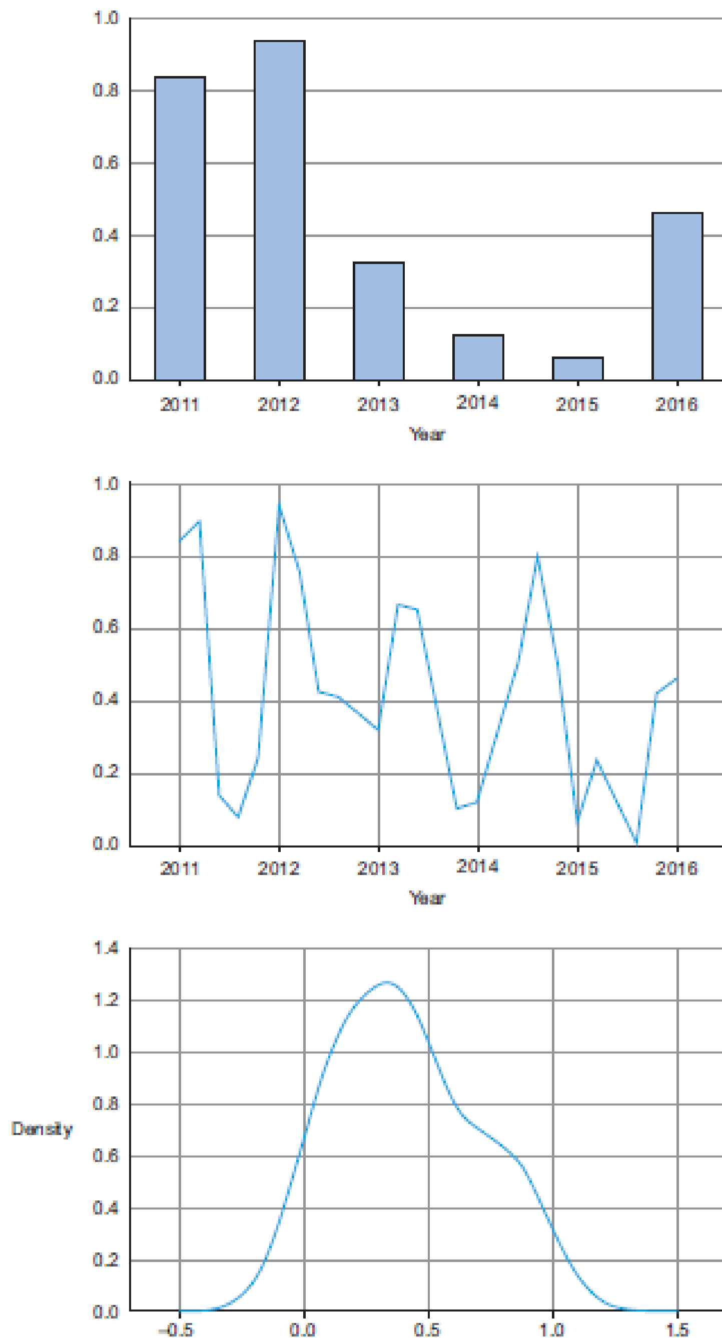


Figure 2.15 From top to bottom, a bar chart, a line plot, and a distribution are some of the graphs used in exploratory analysis.

The visualization techniques you use in this phase range from simple line graphs orhistograms, as shown in figure 2.15, to more complex diagrams such as Sankey andnetwork graphs.

These plots can be combined to provide even more insight, as shown in figure 2.16.Overlaying several plots is common practice. In figure 2.17 we combine simplegraphs into a Pareto diagram, or 80-20 diagram.Figure 2.18 shows another technique: *brushing and linking*. With brushing and linkingyou combine and link different graphs and tables (or views) so changes in onegraph are automatically transferred to the other graphs.

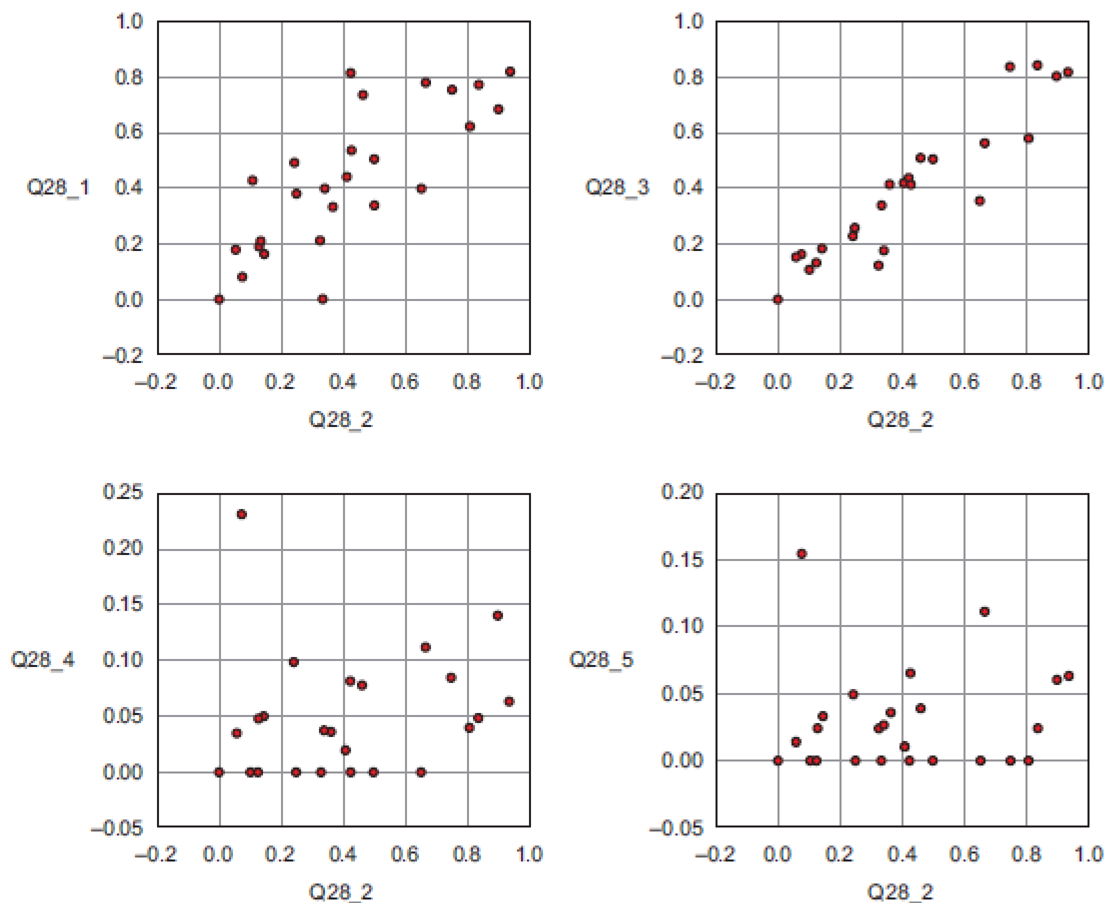


Figure 2.16 Drawing multiple plots together can help you understand the structure of your data over multiple variables.

Pareto diagram: Combination of values and cumulative distribution

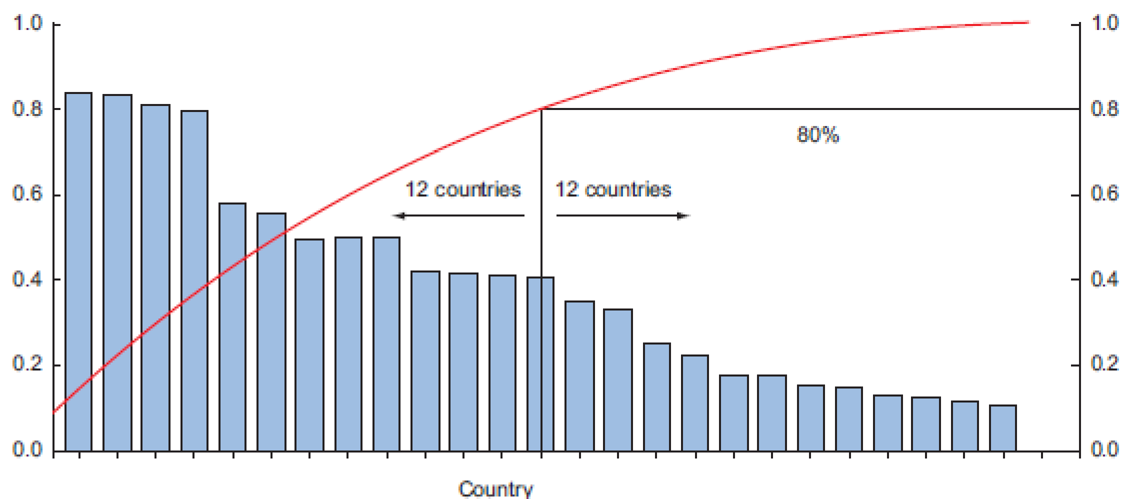


Figure 2.17 A Pareto diagram is a combination of the values and a cumulative distribution. It's easy to see from this diagram that the first 50% of the countries contain slightly less than 80% of the total amount. If this graph represented customer buying power and we sell expensive products, we probably don't need to spend our marketing budget in every country; we could start with the first 50%.

Link and Brush:

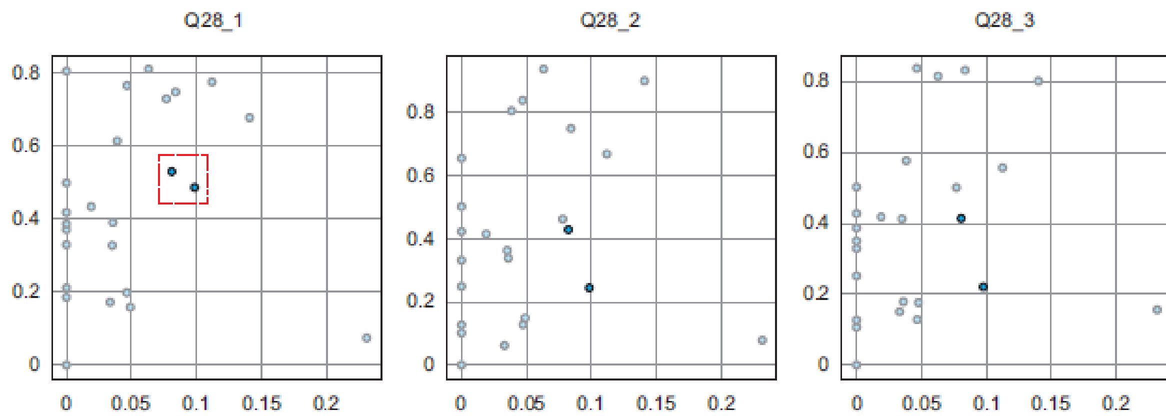


Figure 2.18 Link and brush allows you to select observations in one plot and highlight the same observations in the other plots.

Two other important graphs are the histogram shown in figure 2.19 and the boxplot shown in figure 2.20.

In a histogram a variable is cut into discrete categories and the number of occurrencesin each category are summed up and shown in the graph. The boxplot show how many observations are present but does offer animpression of the distribution within categories. It can show the maximum, minimum,median, and other characterizing measures at the same time.

Histogram:

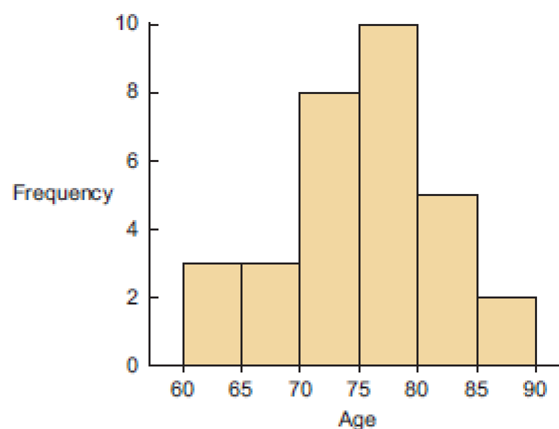


Figure 2.19 Example histogram: the number of people in the age-groups of 5-year intervals

Box plot: It can show the maximum, minimum, median, and other characterizing measures at the same time.

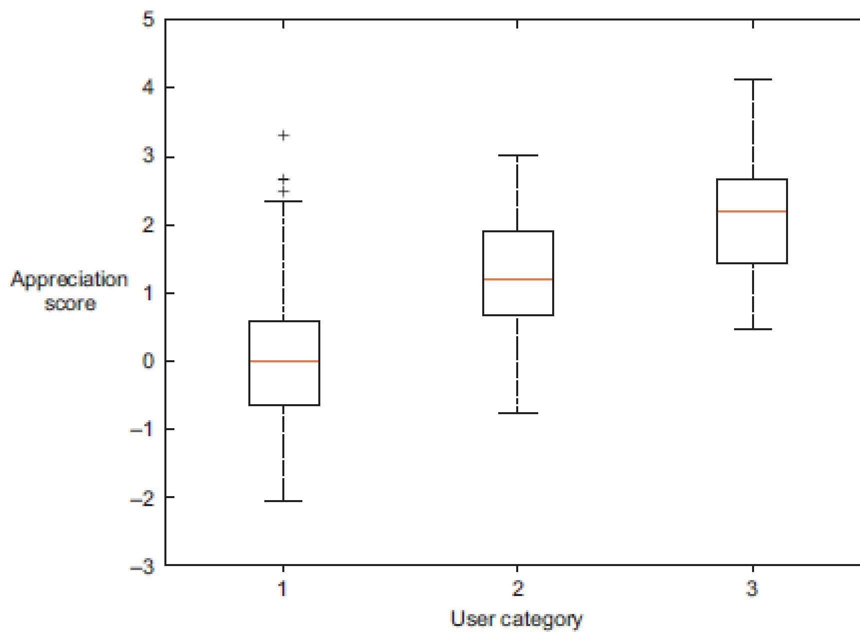


Figure 2.20 Example boxplot: each user category has a distribution of the appreciation each has for a certain picture on a photography website.

2.6 Step 5: Build the models

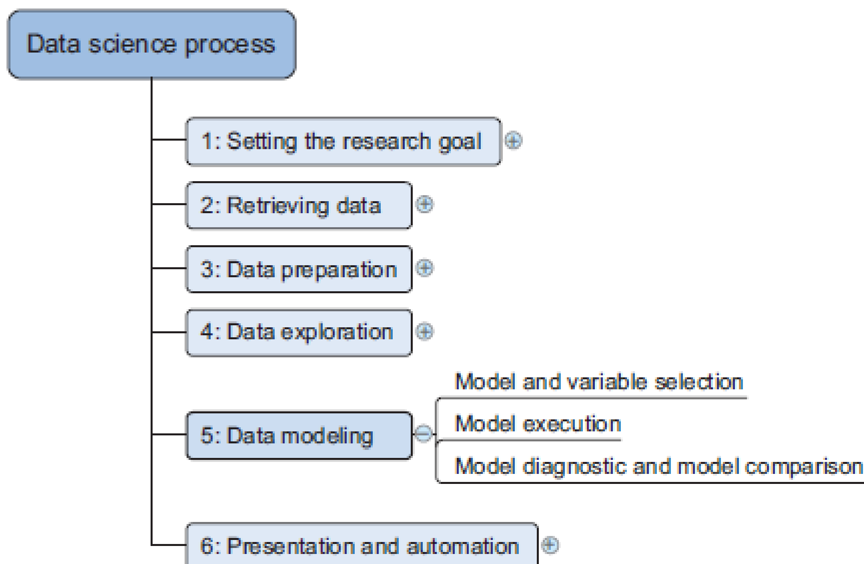


Figure 2.21 Step 5: Data modeling

The techniques you'll use now are borrowed from the field of machine learning, data mining, and/or statistics. most models consist of the following main steps:
1 Selection of a modelling technique and variables to enter in the model
2 Execution of the model
3 Diagnosis and model comparison

2.6.1 Model and variable selection

You'll need to select the variables you want to include in your model and a modelling technique. Your findings from the exploratory analysis should already give a fair idea of what variables will help you construct a good model.
You'll need to consider model performance and whether your project meets all the requirements to use your model, as well as other factors:

■ Must the model be moved to a production environment and, if so, would it be easy to implement?
■ How difficult is the maintenance on the model: how long will it remain relevant if left untouched?
■ Does the model need to be easy to explain?

2.6.2 Model execution

Once you've chosen a model you'll need to implement it in code. Luckily, most programming languages, such as Python, already have libraries such as StatsModels or Scikit-learn. These packages use several of the most popular techniques.

Listing 2.1 Executing a linear prediction model on semi-random data

```
import statsmodels.api as sm
import numpy as np
predictors = np.random.random(1000).reshape(500,2)
target = predictors.dot(np.array([0.4, 0.6])) + np.random.random(500)
lmRegModel = sm.OLS(target,predictors)
result = lmRegModel.fit()
result.summary()
```

Imports required Python modules.

Creates random data for predictors (x-values) and semi-random data for the target (y-values) of the model. We use predictors as input to create the target so we infer a correlation here.

Fits linear regression on data.

Shows model fit statistics.

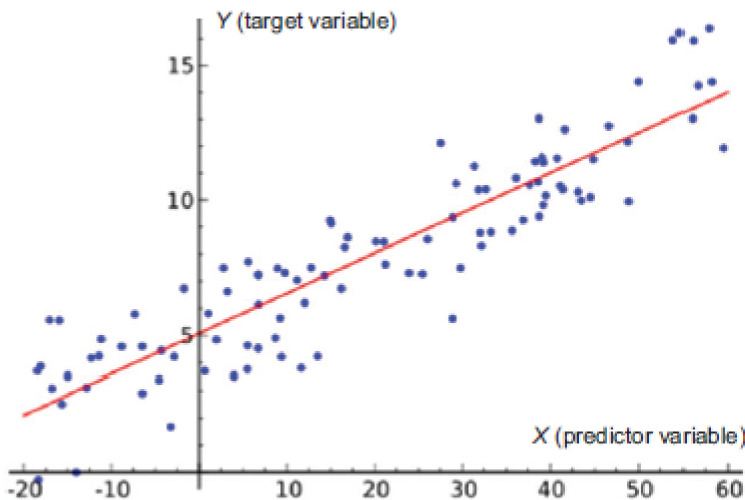


Figure 2.22 Linear regression tries to fit a line while minimizing the distance to each point

We created predictor values that are meant to predict how the target variables behave. For a linear regression, a "linear relation" between each x (predictor) and the y (target) variable is assumed, as shown in figure 2.22.

Dep. Variable:	y	R-squared:	0.893
Model:	OLS	Adj. R-squared:	0.893
Method:	Least Squares	F-statistic:	2088.
Date:	Fri, 30 Oct 2015	Prob (F-statistic):	7.13e-243
Time:	12:44:31	Log-Likelihood:	-176.74
No. Observations:	500	AIC:	357.5
Df Residuals:	498	BIC:	365.9
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[95.0% Conf. Int.]
x1	0.7658	0.040	19.130	0.000	0.687 0.844
x2	1.1252	0.039	28.603	0.000	1.048 1.202

Omnibus:	34.269	Durbin-Watson:	1.943
Prob(Omnibus):	0.000	Jarque-Bera (JB):	13.480
Skew:	-0.125	Prob(JB):	0.00118
Kurtosis:	2.235	Cond. No.	2.51

Model fit: higher is better but too high is suspicious.

p-value to show whether a predictor variable has a significant influence on the target. Lower is better and <0.05 is often considered "significant."

Linear equation coefficients.
y = 0.7658xl + l.l252x2.

Figure 2.23 Linear regression model information output

■ *Model fit*—For this the R-squared or adjusted R-squared is used. This measure is an indication of the amount of variation in the data that gets captured by the model. The difference between the adjusted R-squared and the R-squared is minimal here because the adjusted one is the normal one + a penalty for model complexity.
A model gets complex when many variables (or features) are introduced. You don't need a complex model if a simple model is available, so the adjusted R-squared punishes you for overcomplicating. At any rate, 0.893 is high, and it should be because we cheated.
■ *Predictor variables have a coefficient*—For a linear model this is easy to interpret. Detecting influences is more important in scientific studies than perfectly fitting models (not to mention more realistic).
■ *Predictor significance*—Coefficients are great, but sometimes not enough evidence exists to show that the influence is there. This is what the p-value is about. the p-value is lower than 0.05, the variable is considered significant for most people. It means there's a 5% chance the predictor doesn't have any influence.
Linear regression works if you want to predict a value, but what if you want to classify something? Then you go to classification models, the best known among them being k-nearest neighbors.

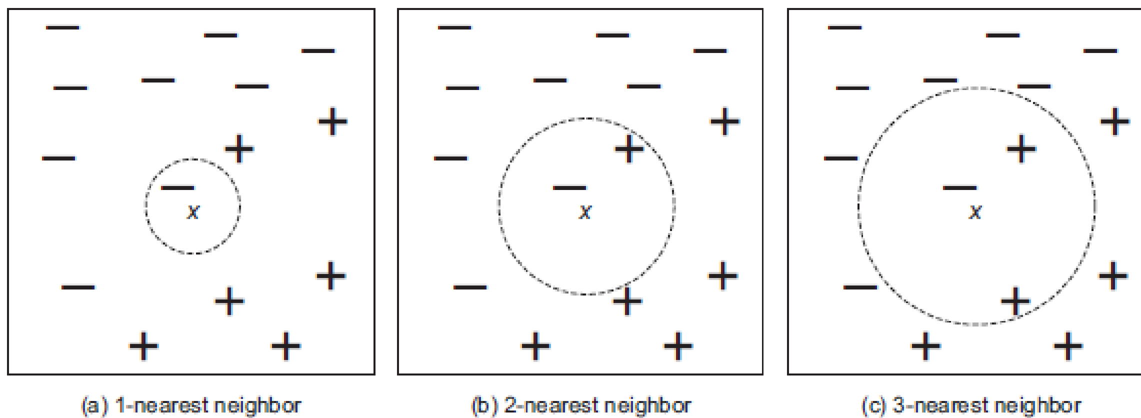


Figure 2.24 K-nearest neighbor techniques look at the k-nearest point to make a prediction.

Listing 2.2 Executing k-nearest neighbor classification on semi-random data

```
from sklearn import neighbors
predictors = np.random.random(1000).reshape(500,2)
target = np.around(predictors.dot(np.array([0.4, 0.6])) +
         np.random.random(500))
clf = neighbors.KNeighborsClassifier(n_neighbors=10)
knn = clf.fit(predictors,target)
knn.score(predictors, target)
```

Imports modules.

Creates random predictor data and semi-random target data based on predictor data.

Fits 10-nearest neighbors model.

Gets model fit score: what percent of the classification was correct?

Don't let knn.score() fool you; it returns the model accuracy, but by "scoring a model" we often mean applying it on data to make a prediction.

prediction = knn.predict(predictors)

Now we can use the prediction and compare it to the real thing using a confusion matrix.

metrics.confusion_matrix(target,prediction)

We get a 3-by-3 matrix as shown in figure 2.25.

AMSC

```
In [45]: metrics.confusion_matrix(target,prediction)
```

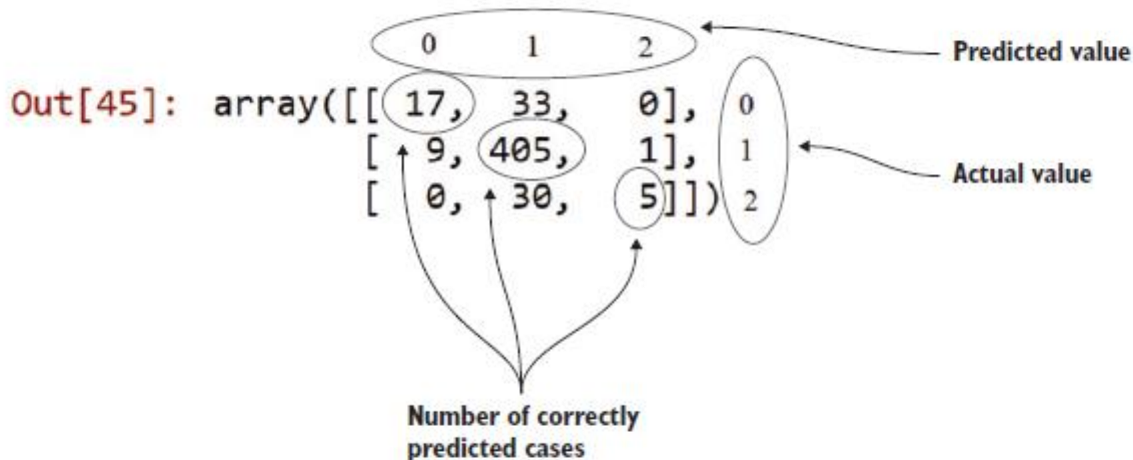


Figure 2.25 Confusion matrix: it shows how many cases were correctly classified and incorrectly classified by comparing the prediction with the real values. Remark: the classes (0,1,2) were added in the figure for clarification.

The confusion matrix shows we have correctly predicted 17+405+5 cases, so that's good.

2.6.3 Model diagnostics and model comparison

You'll be building multiple models from which you then choose the best one based onmultiple criteria. Working with a holdout sample helps you pick the best-performingmodel. A holdout sample is a part of the data you leave out of the model building so itcan be used to evaluate the model afterward.

The principle here is simple: the model should work on unseen data.The model is then unleashed on the unseen data and error measures are calculated to evaluate it. Multiple error measures are available, and in figure 2.26 we show the general idea on comparing models. The error measure used in the example is the mean square error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Figure 2.26 Formula for mean square error

Mean square error is a simple measure: check for every prediction how far it was from the truth, square this error, and add up the error of every prediction.

To estimate the models, we use 800 randomly chosen observations out of 1,000 (or 80%), without showing the other 20% of data to the model.

	n	Size	Price	Predicted model 1	Predicted model 2	Error model 1	Error model 2
	1	10	3				
	2	15	5				
	3	18	6				
	4	14	5				
	...	...					
80% train	800	9	3				
	801	12	4	12	10	0	2
	802	13	4	12	10	1	3
	...						
	999	21	7	21	10	0	11
20% test	1000	10	4	12	10	-2	0
				Total		5861	110225

Figure 2.27 A holdout sample helps you compare models and ensures that you can generalize results to data that the model has not yet seen.

Once the model is trained, we predict the values for the other 20% of the variables based on those for which we already know the true value, and calculate the model error with an error measure. Then we choose the model with the lowest error. In this example we chose model 1 because it has the lowest total error.

Many models make strong assumptions, such as independence of the inputs, and you have to verify that these assumptions are indeed met. This is called model diagnostics.

2.7 Step 6: Presenting findings and building applications on top of them

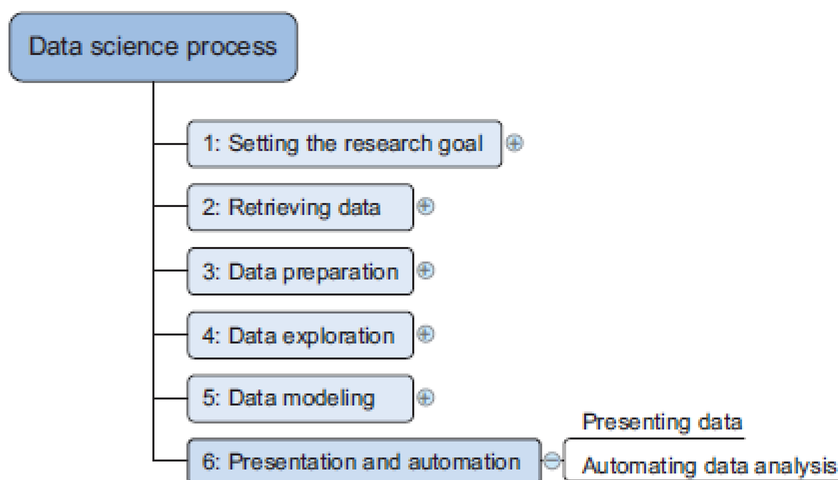


Figure 2.28 Step 6: Presentation and automation

After you've successfully analyzed the data and built a well-performing model, you're ready to present your findings to the world. Sometimes people get so excited about your work that you'll need to repeat it over and over again because they value the predictions of your models or the insights that you produced.

For this reason, you need to automate your models. This doesn't always mean that you have to redo all of your analysis all the time. Sometimes it's sufficient that you implement only the model scoring; other times you might build an application that automatically updates reports, Excel spreadsheets, or PowerPoint presentations. The last stage of the data science process is where your *soft skills* will be most useful, and yes, they're extremely important

Data Mining

Data mining should have been more appropriately named "knowledge mining from data," which is unfortunately somewhat long. However, the shorter term, *knowledge mining* may not reflect the emphasis on mining from large amounts of data. Nevertheless, mining is a vivid term characterizing the process that finds a small set of precious nuggets from a great deal of raw material (Figure 1.3).

In addition, many other terms have a similar meaning to data mining—for example, *knowledge mining from data*, *knowledge extraction*, *data/pattern analysis*, *data archaeology*, and *data dredging*

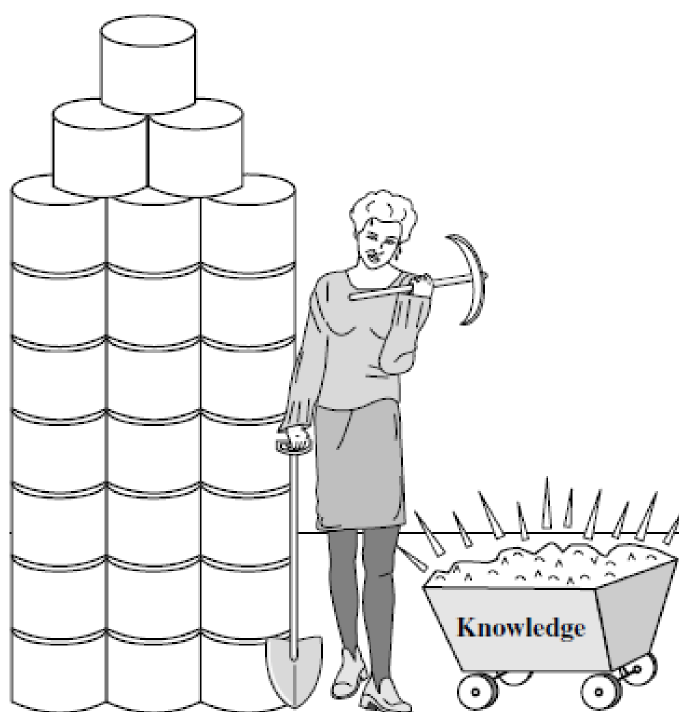


Figure 1.3 Data mining—searching for knowledge (interesting patterns) in data.

Many people treat data mining as a synonym for another popularly used term, **knowledge discovery from data**, or **KDD**, while others view data mining as merely an essential step in the process of knowledge discovery.

The knowledge discovery process is shown in Figure 1.4 as an iterative sequence of the following steps:

1. Data cleaning (to remove noise and inconsistent data)

2. Data integration (where multiple data sources may be combined)

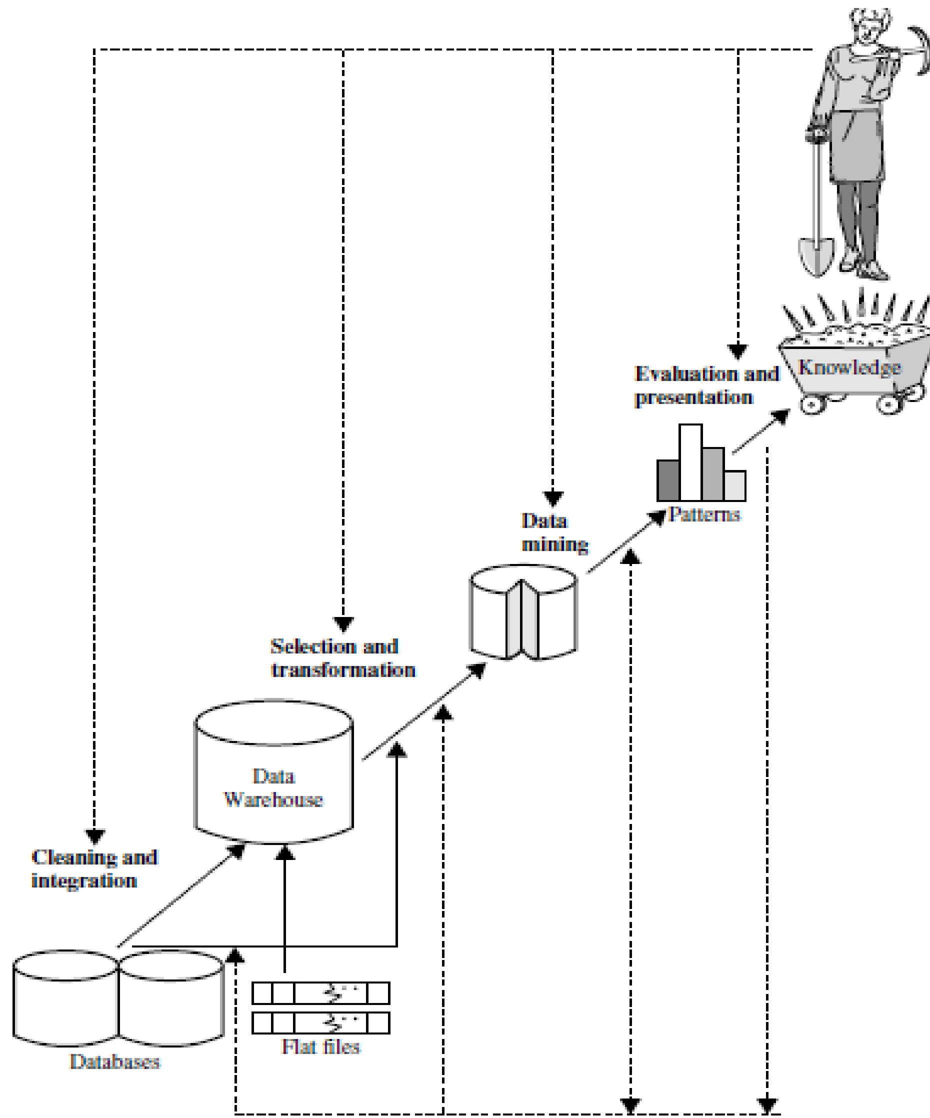


Figure 1.4 Data mining as a step in the process of knowledge discovery.

3. Data selection (where data relevant to the analysis task are retrieved from the database)
4. Data transformation (where data are transformed and consolidated into forms appropriate for mining by performing summary or aggregation operations)
5. Data mining (an essential process where intelligent methods are applied to extract data patterns)
6. Pattern evaluation (to identify the truly interesting patterns representing knowledge based on *interestingness measures*)
7. Knowledge presentation (where visualization and knowledge representation techniques are used to present mined knowledge to users)
Steps 1 through 4 are different forms of data preprocessing, where data are prepared for mining. The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user and may be stored as new knowledge in the knowledge base.
Data mining is the process of discovering interesting patterns and knowledge from large amounts of data. The data sources can include databases, data warehouses, theWeb, other information repositories, or data that are streamed into the system dynamically.

4.1.4 DataWarehousing: A Multitiered Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 4.1.

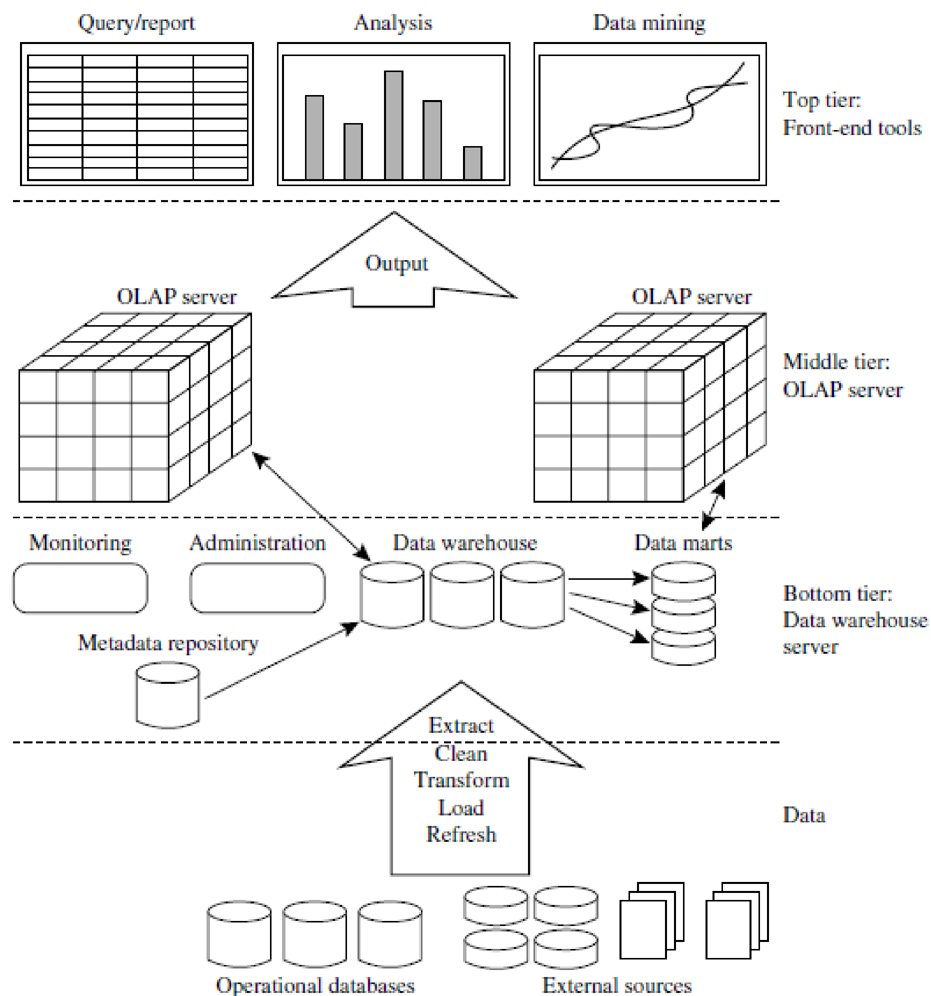


Figure 4.1 A three-tier data warehousing architecture.

1. **The bottom tier is a warehouse database server that is almost always a relational database system. Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources** (e.g., customer profile information provided by external consultants). **These tools and utilities performdata extraction, cleaning, and transformation** (e.g., to merge similar data from different sources into a unified format), as well as load and refresh functions to update the data warehouse (see Section 4.1.6). **The data are extracted using application program interfaces known as gateways. A gateway is supported by the underlying DBMS and allows client programs to generate SQL code to be executed at a server**. Examples of gateways include ODBC (Open Database Connection) and OLEDB (ObjectLinking and Embedding Database) by Microsoft and JDBC (Java Database Connection).

This tier also contains a metadata repository, which stores information about the data warehouse and its contents.

2. The middle tier is an **OLAP (Online analytical processing) server** that is typically implemented using either (1) a **relationalOLAP(ROLAP)** model (i.e., an extended relational DBMS that maps operations on multidimensional data to standard relational operations); or

(2) a **multidimensional OLAP (MOLAP) model** (i.e., a special-purpose server that directly implements multidimensional data and operations).
3. The top tier is a **front-end client layer**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).

- **Relational OLAP (ROLAP) servers:** These are the intermediate servers that stand in between a relational back-end server and client front-end tools. They use a *relational* or *extended-relational DBMS* to store and manage warehouse data, and OLAP middleware to support missing pieces. ROLAP servers include optimization for each DBMS back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.
- **Multidimensional OLAP (MOLAP) servers:** These servers support multidimensional data views through *array-based multidimensional storage engines*. They map multidimensional views directly to data cube array structures. The advantage of using a data cube is that it allows fast indexing to precomputed summarized data. Notice that with multidimensional data stores, the storage utilization may be low if the dataset is sparse. Many MOLAP servers adopt a two-level storage representation to handle dense and sparse data sets: Denser subcubes are identified and stored as array structures, whereas sparse subcubes employ compression technology for efficient storage utilization.
- **Hybrid OLAP (HOLAP) servers:** The hybrid OLAP approach combines ROLAP and MOLAP technology, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP. For example, a HOLAP server may allow large volumes of detailed data to be stored in a relational database, while aggregations are kept in a separate MOLAP store. The Microsoft SQL Server 2000 supports a hybrid OLAP server.
- **Specialized SQL servers:** To meet the growing demand of OLAP processing in relational databases, some database system vendors implement specialized SQL servers that provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

AMSCE-IT

Basic Statistical Descriptions of Data
Basic statistical descriptions can be used to identify properties of the data and highlight which data values should be treated as noise or outliers.
Measuring the Central Tendency: Mean, Median, and Mode
we look at various ways to measure the central tendency of data. The mean of this set of values is

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x_1 + x_2 + \cdots + x_N}{N}.$$

Example 2.6 **Mean.** Suppose we have the following values for *salary* (in thousands of dollars), shown in increasing order: 30, 36, 47, 50, 52, 52, 56, 60, 63, 70, 70, 110. Using Eq. (2.1), we have

$$\bar{x} = \frac{30+36+47+50+52+52+56+60+63+70+70+110}{12} = \frac{696}{12} = 58.$$

Thus, the mean salary is $58,000. ■

Example 2.7 Median. Let's find the median of the data from Example 2.6. The data are already sorted in increasing order. There is an even number of observations (i.e., 12); therefore, the median is not unique. It can be any value within the two middlemost values of 52 and 56 (that is, within the sixth and seventh values in the list). By convention, we assign the average of the two middlemost values as the median; that is, $\frac{52+56}{2} = \frac{108}{2} = 54.$

Thus, the median is $54,000.

Suppose that we had only the first 11 values in the list. Given an odd number of values, the median is the middlemost value. This is the sixth value in this list, which has a value of $52,000.

Example 2.8 Mode. The data from Example 2.6 are bimodal. The two modes are $52,000 and

$70,000. (They are repeated two times).

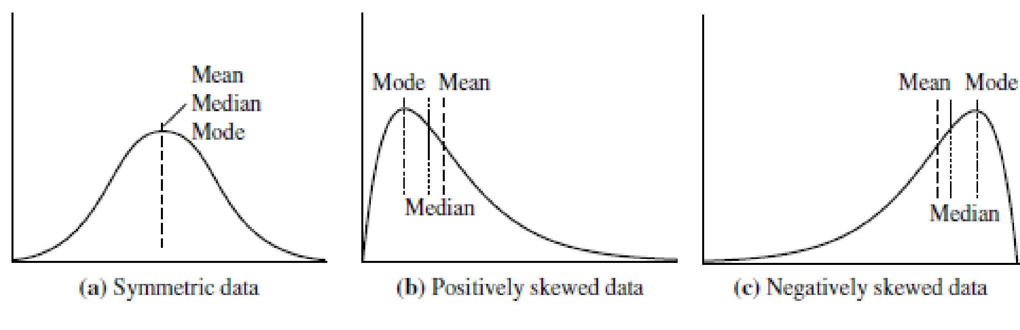
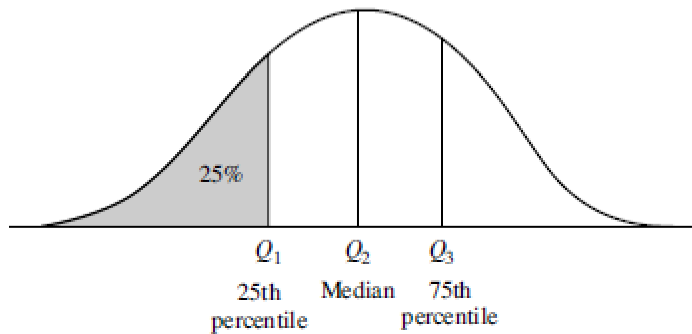


Figure 2.1 Mean, median, and mode of symmetric versus positively and negatively skewed data.

Measuring the Dispersion of Data: Range, Quartiles, Variance, Standard Deviation, and Interquartile Range

- Quantiles are points taken at regular intervals of a data distribution, dividing it into essentially equal size consecutive sets.
- The 2-quantile is the data point dividing the lower and upper halves of the data distribution. It corresponds to the median. The 4-quantiles are the three data points that split the data distribution into four equal parts; each part represents one-fourth of the data distribution. They are more commonly referred to as quartiles. The 100-quantiles are more commonly referred to as percentiles; they divide the data distribution into 100 equal-sized consecutive sets. The median, quartiles, and percentiles are the most widely used forms of quantiles.



The quartiles give an indication of a distribution's center, spread, and shape. The **first quartile**, denoted by Q1, is the 25th percentile. It cuts off the lowest 25% of the data. The **third quartile**, denoted by Q3, is the 75th percentile—it cuts off the lowest 75% (or highest 25%) of the data. The **second quartile** is the 50th percentile. As the median, it gives the center of the data distribution.
The distance between the first and third quartiles is a simple measure of spread that gives the range covered by the middle half of the data. This distance is called the **interquartile range** (IQR) and is defined as

IQR = Q3-Q1.

Interquartile range. The quartiles are the three values that split the sorted data set into four equal parts. 30, 36, 47, 50, 52, 52, 56, 60, 63, 70, 70, 110 Thus, the quartiles for this data are the third, sixth, and ninth values, respectively, in the sorted list. Therefore, Q1 = $47,000 and Q3 is $63,000. Thus, the interquartile range is IQR = 63-47 = $16,000.

For odd number of dataset with 9 values.
30,36,47,50,52,52,56,60,63
Median =52
Lower half = 36+47/2 59.5
Upper half = 56+60/2 86
IQR= 26.5

AMSCE-1101

Five-Number Summary, Boxplots, and Outliers

The five-number summary of a distribution consists of the median (Q2), the quartiles Q1 and Q3, and the smallest and largest individual observations, written in the order of Minimum, Q1, Median, Q3, Maximum.

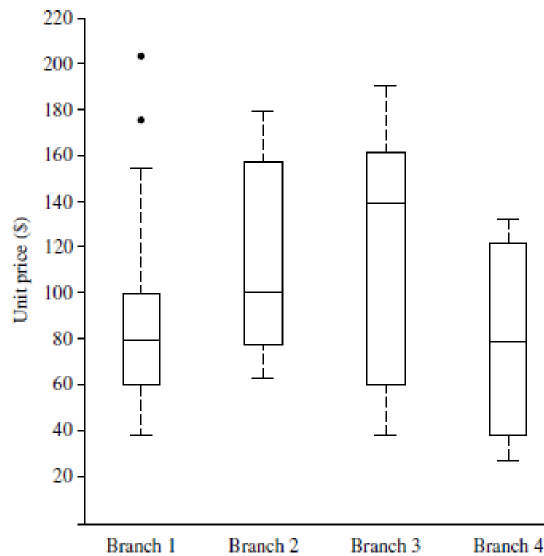


Figure 2.3 Boxplot for the unit price data for items sold at four branches of *AllElectronics* during a given time period.

Variance and Standard Deviation

Variance and standard deviation are measures of data dispersion. They indicate how spread out a data distribution is. A low standard deviation means that the data observations tend to be very close to the mean, while a high standard deviation indicates that the data are spread out over a large range of values.

The **variance** of N observations, $x1$,$x2$, : : : ,xN, for a numeric attribute X is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 = \left(\frac{1}{N}\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2,$$

where $\bar{x}$ is the mean value of the observations, as defined in Eq. (2.1). The **standard deviation**, σ, of the observations is the square root of the variance, σ 2.

Example 2.12 **Variance and standard deviation.** In Example 2.6, we found $\bar{x} = \$58{,}000$ using Eq. (2.1) for the mean. To determine the variance and standard deviation of the data from that example, we set $N = 12$ and use Eq. (2.6) to obtain

$$\sigma^2 = \frac{1}{12}(30^2 + 36^2 + 47^2 \ldots + 110^2) - 58^2$$
$$\approx 379.17$$
$$\sigma \approx \sqrt{379.17} \approx 19.47.$$

■

$\bar{x}$ represents mean

σ standard deviation

σ^2 Variance

UNIT II

Types of data

THREE TYPES OF DATA

Any statistical analysis is performed on data, a collection of actual observations or scores in a survey or an experiment. The precise form of a statistical analysis often depends on whether data are qualitative, ranked, or quantitative.

Qualitative data consist of words (Yes or No), letters (Y or N), or numerical codes (0 or 1) that represent a class or category. **Ranked data consist of numbers** (1st, 2nd, . . . 40th place) that represent relative standing within a group. **Quantitative data consist of numbers** (weights of 238, 170, . . . 185 lbs) that represent an amount or a count.

Table 1.2
QUALITATIVE DATA: "DO YOU HAVE A FACEBOOK PROFILE?" YES (Y) OR NO (N) REPLIES OF STATISTICS STUDENTS

Y	Y	Y	N	N	Y	Y	Y
Y	Y	Y	N	N	Y	Y	Y
N	Y	N	Y	Y	Y	Y	Y
Y	Y	N	Y	N	Y	N	Y
Y	N	Y	N	N	Y	Y	Y
Y	Y	N	Y	Y	Y	Y	Y
N	N	N	N	Y	N	N	Y
Y	Y	Y	Y	Y	N	Y	N
Y	Y	Y	Y	N	N	Y	Y
N	Y	N	N	Y	Y	Y	Y
	Y	Y	N				

TYPES OF VARIABLES

A **variable** is a characteristic or property that can take on different values.

Discrete and Continuous Variables

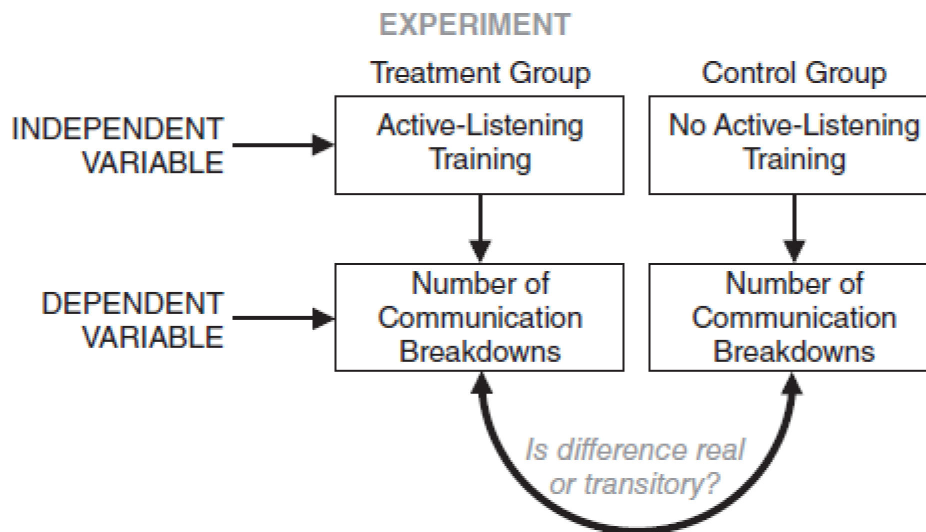
Quantitative variables can be further distinguished in terms of whether they are discrete or continuous. A **discrete variable** consists of isolated numbers separated by gaps. Examples include most counts, such as the number of children in a family (1, 2,3, etc., but never 1 1/2.

A **continuous variable** consists of numbers whose values, at least in theory, have no restrictions. Examples include amounts, such as weights of male statistics students; durations, such as the reaction times of grade school children to a fire alarm; and standardized test scores, such as those on the Scholastic Aptitude Test (SAT).

Independent and Dependent Variables

For example, a psychologist might wish to investigate whether couples who undergo special training in "active listening" tend to have fewer communication breakdowns than do couples who undergo no special training. To study this, the psychologist may expose couples to two different conditions by randomly assigning them either to a treatment group that receives special training in active listening or to a control group that receives no special

training. Such studies are referred to as experiments. *An* **experiment** *is a study in which the investigator decides who receives the special treatment.*



Independent Variable (*The treatment manipulated by the investigator in an experiment.***)**
Since training is assumed to influence communication, it is an independent variable. *In an experiment, an* **independent variable** *is the treatment manipulated by the investigator.*

Once the data have been collected, any difference between the groups can be interpreted as being *caused* by the independent variable.

If, for instance, a difference appears in favor of the active-listening group, the psychologist can conclude that training in active listening causes fewer communication breakdowns between couples. Having observed this relationship, the psychologist can expect that, if new couples were trained in active listening, fewer breakdowns in communication would occur.

Dependent Variable (*A variable that is believed to have been influenced by the independent variable.***)**

To test whether training influences communication, the psychologist counts the number of communication breakdowns between each couple, as revealed by inappropriate replies, aggressive comments, verbal interruptions, etc., while discussing a conflict-provoking topic, such as whether it is acceptable to be intimate with a third person.

In an experimental setting, the dependent variable is measured, counted, or recorded by the investigator.
Unlike the independent variable, the dependent variable isn't manipulated by the investigator. Instead, it represents an outcome: the data produced by the experiment

Confounding Variable

Couples willing to devote extra effort to special training might already possess a deeper commitment that co-varies with more active-listening skills. *An uncontrolled variable that compromises the interpretation of a study is known as a* **confounding variable.** You can avoid confounding variables, as in the present case, by assigning subjects randomly to the various groups in the experiment and also by standardizing all experimental conditions, other than the independent variable, for subjects in both groups.

Describing Data with Tables and Graphs

2.1 FREQUENCY DISTRIBUTIONS FOR QUANTITATIVE DATA

A *frequency distribution* is a collection of observations produced by sorting observations into classes and showing their frequency (*f*) of occurrence in each class.

ANSCE

Frequency distribution for ungrouped data

Table 2.1 FREQUENCY DISTRIBUTION (UNGROUPED DATA)	
WEIGHT	*f*
245	1
244	0
243	0
242	0
*	
*	
*	
161	0
160	4
159	1
158	2
157	3
*	
*	
*	
136	0
135	2
134	0
133	1
Total	53

AMSCE-1101

Not Always Appropriate
Frequency distributions for ungrouped data are much more informative when the number of possible values is less than about 20. Under these circumstances, they are a straightforward method for organizing data. Otherwise, if there are 20 or more possible values, consider using a frequency distribution for grouped data.

Grouped Data

Table 2.2 shows another way to organize the weights in Table 1.1 according to their frequency of occurrence. When observations are sorted into classes of *more than one value,* as in Table 2.2, the result is referred to as a **frequency distribution for grouped data.**

Data are grouped into class intervals with 10 possible values each. The bottom class includes the smallest observation (133), and the top class includes the largest observation (245). The distance between bottom and top is occupied by an orderly series of classes. The frequency (*f*) column shows the frequency of observations in each class and, at the bottom, the total number of observations in all classes.

Table 2.2
FREQUENCY DISTRIBUTION (GROUPED DATA)

WEIGHT	*f*
240–249	1
230–239	0
220–229	3
210–219	0
200–209	2
190–199	4
180–189	3
170–179	7
160–169	12
150–159	17
140–149	1
130–139	3
Total	53

2.2 GUIDELINES

The "Guidelines for Frequency Distributions" box lists seven rules for producing a well-constructed frequency distribution. The first three rules are essential and should not be violated. The last four rules are optional and can be modified or ignored as circumstances warrant.

AMSCE-1101

GUIDELINES FOR FREQUENCY DISTRIBUTIONS

Essential

1. ***Each observation should be included in one, and only one, class.***

   Example: 130–139, 140–149, 150–159, etc. It would be incorrect to use 130–140, 140–150, 150–160, etc., in which, because the boundaries of classes overlap, an observation of 140 (or 150) could be assigned to either of two classes.

2. ***List all classes, even those with zero frequencies.***

   Example: Listed in Table 2.2 is the class 210–219 and its frequency of zero. It would be incorrect to skip this class because of its zero frequency.

3. ***All classes should have equal intervals.***

   Example: 130–139, 140–149, 150–159, etc. It would be incorrect to use 130–139, 140–159, etc., in which the second class interval (140–159) is twice as wide as the first class interval (130–139).

Optional

4. ***All classes should have both an upper boundary and a lower boundary.***

   Example: 240–249. Less preferred would be 240–above, in which no maximum value can be assigned to observations in this class. (Nevertheless, this type of open-ended class is employed as a space-saving device when many different tables must be listed, as in the *Statistical Abstract of the United States*. An open-ended class appears in the table "Two Age Distributions" in Review Question 2.17 at the end of this chapter.)

5. ***Select the class interval from convenient numbers, such as 1, 2, 3, . . . 10, particularly 5 and 10 or multiples of 5 and 10.***

   Example: 130–139, 140–149, in which the class interval of 10 is a convenient number. Less preferred would be 130–142, 143–155, etc., in which the class interval of 13 is not a convenient number.

6. ***The lower boundary of each class interval should be a multiple of the class interval.***

   Example: 130–139, 140–149, in which the lower boundaries of 130, 140, are multiples of 10, the class interval. Less preferred would be 135–144, 145–154, etc., in which the lower boundaries of 135 and 145 are not multiples of 10, the class interval.

7. ***Aim for a total of approximately 10 classes.***

   Example: The distribution in Table 2.2 uses 12 classes. Less preferred would be the distributions in Tables 2.3 and 2.4. The distribution in Table 2.3 has too many classes (24), whereas the distribution in Table 2.4 has too few classes (3).

CONSTRUCTING FREQUENCY DISTRIBUTIONS

1. ***Find the range, that is,*** the difference between the largest and smallest observations. The range of weights in Table 1.1 is 245 − 133 = 112.
2. ***Find the class interval required to span the range*** by dividing the range by the desired number of classes (ordinarily 10). In the present example,

$$\textit{Class interval} = \frac{\textit{range}}{\textit{desired number of classes}} = \frac{112}{10} = 11.2$$

3. ***Round off to the nearest convenient interval*** (such as 1, 2, 3, . . . 10, particularly 5 or 10 or multiples of 5 or 10). In the present example, the nearest convenient interval is 10.
4. ***Determine where the lowest class should begin.*** (Ordinarily, this number should be a multiple of the class interval.) In the present example, the smallest score is 133, and therefore the lowest class should begin at 130, since 130 is a multiple of 10 (the class interval).
5. ***Determine where the lowest class should end*** by adding the class interval to the lower boundary and then subtracting one unit of measurement. In the present example, add 10 to 130 and then subtract 1, the unit of measurement, to obtain 139—the number at which the lowest class should end.
6. ***Working upward, list as many equivalent classes as are required to include the largest observation.*** In the present example, list 130–139, 140–149, . . . , 240–249, so that the last class includes 245, the largest score.
7. ***Indicate with a tally the class in which each observation falls.*** For example, the first score in Table 1.1, 160, produces a tally next to 160–169; the next score, 193, produces a tally next to 190–199; and so on.
8. ***Replace the tally count for each class with a number—the frequency (f)—and show the total of all frequencies.*** (Tally marks are not usually shown in the final frequency distribution.)
9. ***Supply headings for both columns and a title for the table.***

2.4 RELATIVE FREQUENCY DISTRIBUTIONS

Relative frequency distributions show the frequency of each class as a part or fraction of the total frequency for the entire distribution.

Constructing Relative Frequency Distributions

To convert a frequency distribution into a relative frequency distribution, divide the frequency for each class by the total frequency for the entire distribution.

Table 2.5 RELATIVE FREQUENCY DISTRIBUTION		
WEIGHT	*f*	RELATIVE *f*
240–249	1	.02
230–239	0	.00
220–229	3	.06
210–219	0	.00
200–209	2	.04
190–199	4	.08
180–189	3	.06
170–179	7	.13
160–169	12	.23
150–159	17	.32
140–149	1	.02
130–139	3	.06
Total	53	1.02*

** The sum does not equal 1.00 because of rounding-off errors.*

For instance, to obtain the proportion of .06 for the class 130–139, divide the frequency of 3 for that class by the total frequency of 53.

Percentages or Proportions?

To convert the relative frequencies in Table 2.5 from proportions to percentages, multiply each proportion by 100; that is, move the decimal point two places to the right. For example, multiply .06 (the proportion for the class 130–139) by 100 to obtain 6 percent.

2.5 CUMULATIVE FREQUENCY DISTRIBUTIONS

Cumulative frequency distributions show the total number of observations in each class and in all lower-ranked classes.

AMSCE

Table 2.6
CUMULATIVE FREQUENCY DISTRIBUTION

WEIGHT	*f*	CUMULATIVE *f*	CUMULATIVE PERCENT
240–249	1	53	100
230–239	0	52	98
220–229	3	52	98
210–219	0	49	92
200–209	2	49	92
190–199	4	47	89
180–189	3	43	81
170–179	7	40	75
160–169	12	33	62
150–159	17	21	40
140–149	1	4	8
130–139	3	3	6
Total	53		

For class 130-139 the cumulative frequency is 3 since, there are no lower classes.
For class 140-149 the cumulaive frequency is 1+3 = 4
For class 150-159 the cumulative frequency is 1+3+17= 21

The cumulative percent for class 130-139 is given by (cumulative frequency / Total no.of freq)*100.
Example (3/53)*100 = 5.66 = 6

Percentile Ranks

When used to describe the relative position of any score within its parent distribution, cumulative percentages are referred to as percentile ranks. *The* **percentile rank** *of a score indicates the percentage of scores in the entire distribution with similar or smaller values than that score.*

Approximate Percentile Ranks (from Grouped Data)

The assignment of *exact* percentile ranks requires that cumulative percentages be obtained from frequency distributions for ungrouped data. If we have access only to a frequency distribution for grouped data, as in Table 2.6, cumulative percentages can be used to assign *approximate* percentile ranks.In Table 2.6, for example, any weight in the class 170–179 could be assigned an approximate percentile rank of 75, since 75 is the cumulative percent for this class.

2.6 FREQUENCY DISTRIBUTIONS FOR QUALITATIVE (NOMINAL) DATA

Qualitative data consist of words (Yes or No), letters (Y or N), or numerical codes (0 or 1) that represent a class or category.

AMSCE-I

Table 2.7
FACEBOOK PROFILE SURVEY

Response	*f*
Yes	56
No	27
Total	83

Relative and Cumulative Distributions for Qualitative Data

Frequency distributions for qualitative variables can always be converted into relative frequency distributions, as illustrated in Table 2.8. Furthermore, if measurement is ordinal because observations can be ordered from least to most, cumulative frequencies (and cumulative percentages) can be used.

Table 2.8
RANKS OF OFFICERS IN THE U.S. ARMY (PROJECTED 2016)

RANK	*f*	PROPORTION	CUMULATIVE PERCENT
General	311	.004*	100.0
Colonel	13,156	.167	99.6
Major	16,108	.204	82.9
Captain	29,169	.370	62.5
Lieutenant	20,083	.255	25.5
Total	78,827		

PROPORTION – RELATIVE FREQUENCY

Example: 29169/78827 = 0.370

To find cumulative percent we have to find cumulative frequency:

	F	Cumulative Frequency	Cumulative percent
	311	78827	100
	13156	78516	99.6
	16108	65360	82.9
	29169	49252	62.5
	20083	20083	25.5
Total	78827		

Cumulative percent = 49252/78827 =0.624*100=62.48

AMSC

2.8 GRAPHS FOR QUANTITATIVE DATA

Histograms

Table 2.2
FREQUENCY DISTRIBUTION (GROUPED DATA)

WEIGHT	*f*
240–249	1
230–239	0
220–229	3
210–219	0
200–209	2
190–199	4
180–189	3
170–179	7
160–169	12
150–159	17
140–149	1
130–139	3
Total	53

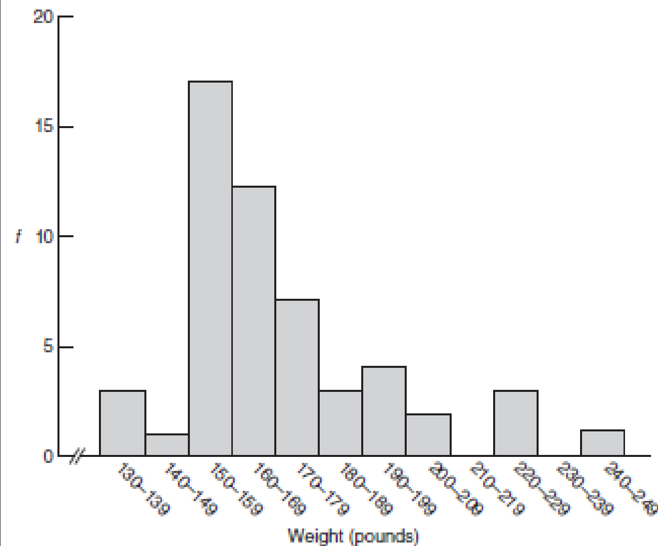


FIGURE 2.1
Histogram.

The weight distribution described in Table 2.2 appears as a **histogram** in **Figure 2.1.** A casual glance at this histogram confirms previous conclusions: a dense concentration of weights among the 150s, 160s, and 170s, with a spread in the direction of the heavier weights. Let's pinpoint some of the more important features of histograms.

1. Equal units along the horizontal axis (the *X* axis, or abscissa) reflect the various class intervals of the frequency distribution.
2. Equal units along the vertical axis (the Y axis, or ordinate) reflect increases in frequency.
3. The intersection of the two axes defines the origin at which both numerical scales equal 0.
4. Numerical scales always increase from left to right along the horizontal axis and from bottom to top along the vertical axis. It is considered good practice to use wiggly lines to highlight breaks in scale, such as those along the horizontal axis in Figure 2.1, between the origin of 0 and the smallest class of 130–139.
5. The body of the histogram consists of a series of bars whose heights reflect the frequencies for the various classes

Frequency Polygon

An important variation on a histogram is the **frequency polygon,** or line graph. *Frequency polygons may be constructed directly from frequency distributions.*

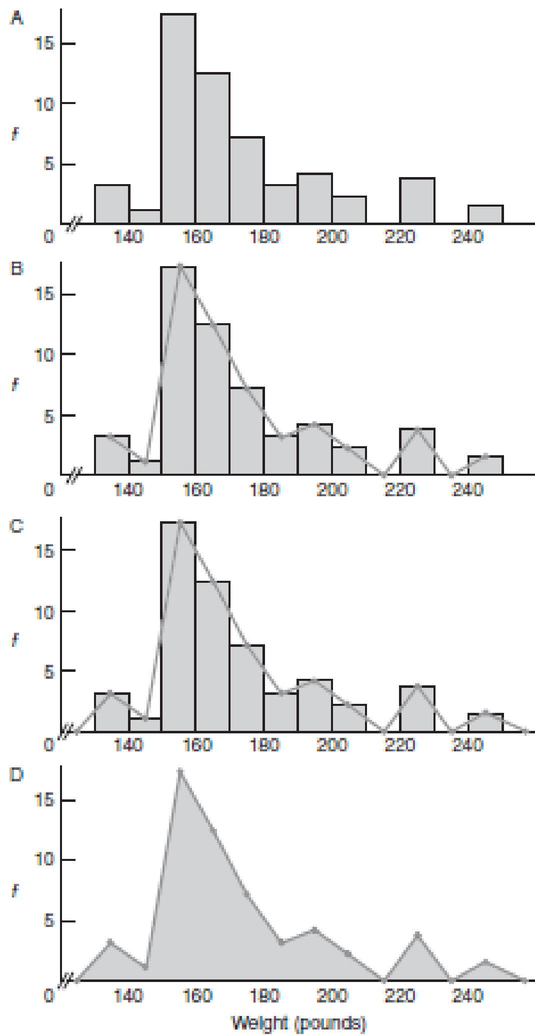


FIGURE 2.2
Transition from histogram to frequency polygon.

Stem and Leaf Displays

Still another technique for summarizing quantitative data is a **stem and leaf display.**

Constructing a Display

The leftmost panel of **Table 2.9** re-creates the weights of the 53 male statistics students listed in Table 1.1.

Table 1.1
QUANTITATIVE DATA: WEIGHTS (IN POUNDS) OF MALE STATISTICS STUDENTS

160	168	133	170	150	165	158	165
193	169	245	160	152	190	179	157
226	160	170	180	150	156	190	156
157	163	152	158	225	135	165	135
180	172	160	170	145	185	152	
205	151	220	166	152	159	156	
165	157	190	206	172	175	154	

Draw a vertical line to separate the stems, which represent multiples of 10, from the space to be occupied by the leaves, which represent multiples of 1.

Table 2.9
CONSTRUCTING STEM AND LEAF DISPLAY FROM WEIGHTS OF MALE STATISTICS STUDENTS

RAW SCORES

160	165	135	175
193	168	245	165
226	169	170	185
152	160	156	154
180	170	160	179
205	150	225	165
163	152	190	206
157	160	159	165
151	190	172	157
157	150	190	156
220	133	166	135
145	180	158	
158	152	152	
172	170	156	

STEM AND LEAF DISPLAY

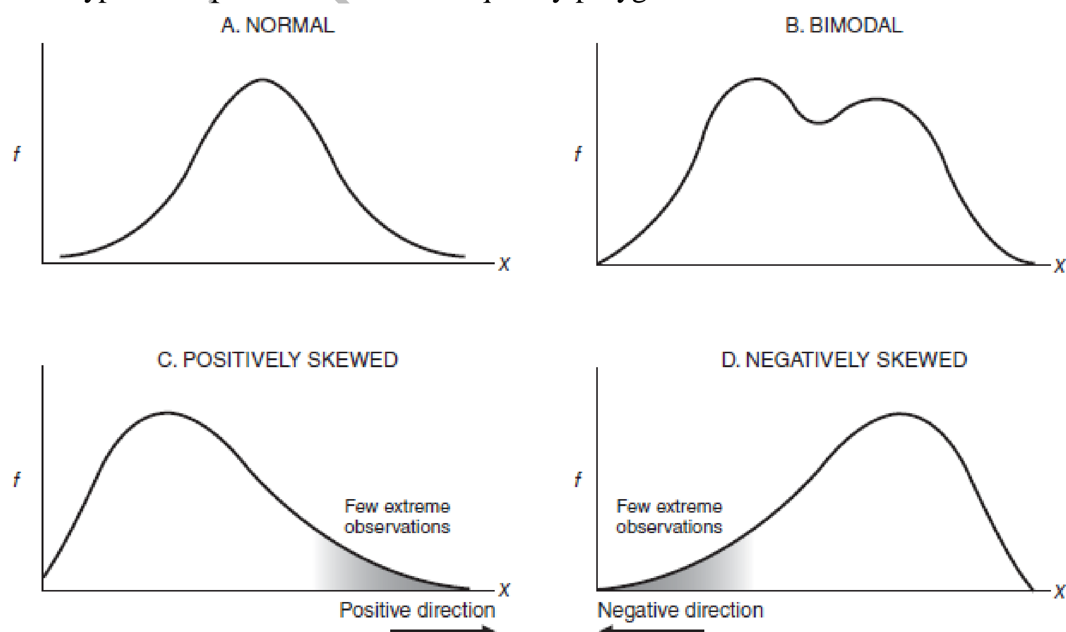
Stem	Leaves
13	3 5 5
14	5
15	2 7 1 7 8 0 2 0 2 6 9 8 2 6 4 7 6
16	0 3 5 8 9 0 0 0 6 5 5 5
17	2 0 0 0 2 5 9
18	0 0 5
19	3 0 0 0
20	5 6
21	
22	6 0 5
23	
24	5

Selection of Stems

Stem values are not limited to units of 10. Depending on the data value of 10, such as 1, 100, 1000, or even .1, .01, .001, and so on can be selected.
For instance, an annual income of $23,784 could be displayed as a stem of 23 (thousands) and a leaf of 784. (Leaves consisting of two or more digits, such as 784, are separated by commas.)

2.9 TYPICAL SHAPES

Whether expressed as a histogram, a frequency polygon, or a stem and leaf display, an important characteristic of a frequency distribution is its shape. **Figure 2.3** shows some of the more typical shapes for smoothed frequency polygons



Normal

Any distribution that approximates the normal shape. The familiar bell-shaped silhouette of the normal curve can be superimposed on many frequency distributions.

Bimodal

Any distribution that approximates the bimodal shape in panel B, might, as suggested previously, reflect the coexistence of two different types of observations in the same distribution. For instance, the distribution of the ages of residents in a neighborhood consisting largely of either new parents or their infants has a bimodal shape.

Positively Skewed

The two remaining shapes in Figure 2.3 are lopsided. A *lopsided distribution caused by a few extreme observations in the positive direction*

Negatively Skewed

A lopsided distribution caused by a few extreme observations in the negative direction (to the left of the majority of observations)

2.10 A GRAPH FOR QUALITATIVE (NOMINAL) DATA

"Do you have a Facebook profile?" appears as a **bar graph** in **Figure 2.4.** As with histograms, equal segments along the horizontal axis are allocated to the different words or classes that appear in the frequency distribution for qualitative data.

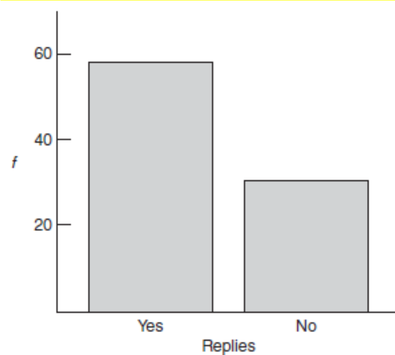


FIGURE 2.4
Bar graph.

2.11 MISLEADING GRAPHS

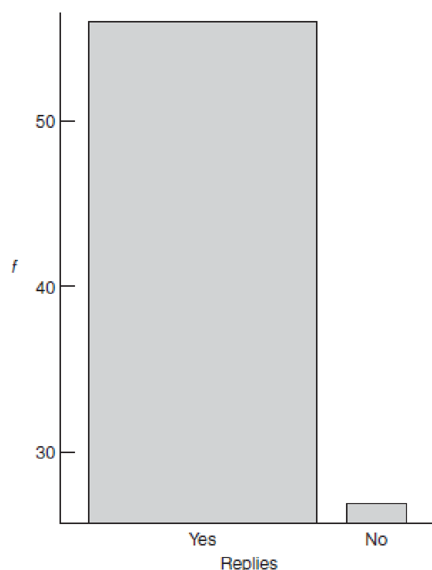


FIGURE 2.5
Distorted bar graph.

ASCE-101

CONSTRUCTING GRAPHS

1. ***Decide on the appropriate type of graph,*** recalling that histograms and frequency polygons are appropriate for quantitative data, while bar graphs are appropriate for qualitative data and also are sometimes used with discrete quantitative data.
2. ***Draw the horizontal axis, then the vertical axis,*** remembering that the vertical axis should be about as tall as the horizontal axis is wide.
3. ***Identify the string of class intervals that eventually will be superimposed on the horizontal axis.*** For qualitative data or ungrouped quantitative data, this is easy—just use the classes suggested by the data. For grouped quantitative data, proceed as if you were creating a set of class intervals for a frequency distribution. (See the box "Constructing Frequency Distributions" on page 27.)
4. ***Superimpose the string of class intervals (with gaps for bar graphs) along the entire length of the horizontal axis.*** For histograms and frequency polygons, be prepared for some trial and error—use a pencil! Do not use a string of empty class intervals to bridge a sizable gap between the origin of 0 and the smallest class interval. Instead, use wiggly lines to signal a break in scale, then begin with the smallest class interval. Also, do not clutter the horizontal scale with excessive numbers—use just a few convenient numbers.
5. ***Along the entire length of the vertical axis, superimpose a progression of convenient numbers,*** beginning at the bottom with 0 and ending at the top with a number as large as or slightly larger than the maximum observed frequency. If there is a considerable gap between the origin of 0 and the smallest observed frequency, use wiggly lines to signal a break in scale.
6. Using the scaled axes, ***construct bars (or dots and lines) to reflect the frequency of observations within each class interval.*** For frequency polygons, dots should be located above the midpoints of class intervals, and both tails of the graph should be anchored to the horizontal axis, as described under "Frequency Polygons" in Section 2.8.
7. ***Supply labels for both axes and a title (or even an explanatory sentence) for the graph.***

Describing Data with Averages

3.1 MODE

The mode reflects the value of the most frequently occurring score.

Table 3.1 TERMS IN YEARS OF 20 RECENT U.S. PRESIDENTS, LISTED CHRONOLOGICALLY	
4	(Harrison)
4	
4	
8	
4	
8	
2	
6	
4	
12	
8	
8	
2	
6	
5	
3	
4	
8	
4	
8	(Clinton)

Four years is the modal term, since the greatest number of presidents, 7, served this term. Note that the mode equals 4 years

More Than One Mode

Distributions can have more than one mode (or no mode at all).

Distributions with two obvious peaks, even though they are not exactly the same height, are referred to as **bimodal.** Distributions with more than two peaks are referred to as **multimodal.** The presence of more than one mode might reflect important differences among subsets of data.

AMSCE-1101

3.2 MEDIAN

The median reflects the middle value when observations are ordered from least to most.

Table 3.2
FINDING THE MEDIAN

A. INSTRUCTIONS

1. Order scores from least to most.
2. Find the middle position by adding one to the total number of scores and dividing by 2.
3. *If the middle position is a whole number*, as in the left-hand panel below, use this number to *count* into the set of ordered scores.
4. The value of the median equals the value of the score located at the middle position.
5. *If the middle position is not a whole number*, as in the right-hand panel below, use the two nearest whole numbers to *count* into the set of ordered scores.
6. The value of the median equals the value midway between those of the two middlemost scores; to find the midway value, add the two given values and divide by 2.

B. EXAMPLES

Set of five scores:	Set of six scores:
2, 8, 2, 7, 6	3, 8, 9, 3, 1, 8
1 2, 2, 6, 7, 8	1 1, 3, 3, 8, 8, 9
2 $\frac{5+1}{2}=3$	2 $\frac{6+1}{2}=3.5$
2, 2, 6, 7, 8 ↑	
3 1, 2, 3	
4 median = 6	1, 3, 3, 8, 8, 9 ↑ ↑
	5 1, 2, 3, 4
	6 median $=\frac{3+8}{2}=5.5$

3.3 MEAN

The mean is found by adding all scores and then dividing by the number of scores.

$$Mean = \frac{sum\ of\ all\ scores}{number\ of\ scores}$$

Sample or Population?

Statisticians distinguish between two types of means—the population mean and the sample mean—depending on whether the data are viewed as a **population** (*a complete set of scores*) or as a **sample** (*a subset of scores*).

Formula for Sample Mean

$\overline{X}$ designates the **sample mean,** and the formula becomes

The balance point for a sample, found by dividing the sum for the values of all scores in the sample by the number of scores in the sample.

SAMPLE MEAN

$$\overline{X} = \frac{\sum X}{n}$$

AMSC

Formula for Population Mean

The formula for the population mean differs from that for the sample mean only because of a change in some symbols. In statistics, Greek symbols usually describe population characteristics, such as the population mean, while English letters usually describe sample characteristics, such as the sample mean.

POPULATION MEAN

$$\mu = \frac{\sum X}{N}$$

Mean as Balance Point

The mean serves as the balance point for its frequency distribution.

3.4 WHICH AVERAGE?

If Distribution Is Not Skewed

When a distribution of scores is not too skewed, the values of the mode, median, and mean are similar, and any of them can be used to describe the central tendency of the distribution.

If Distribution Is Skewed

When extreme scores cause a distribution to be skewed, as for the infant death rates for selected countries listed in **Table 3.4,** the values of the three averages can differ appreciably.

Table 3.4
INFANT DEATH RATES FOR SELECTED COUNTRIES (2012)

COUNTRY	INFANT DEATH RATE*
Sierra Leone	182
Pakistan	86
Ghana	72
India	56
South Africa	45
Cambodia	40
Mexico	16
China	14
Brazil	14
United States	7
Cuba	6
United Kingdom	6
Netherlands	4
Israel	4
France	4
Denmark	4
Germany	4
Japan	3
Sweden	3

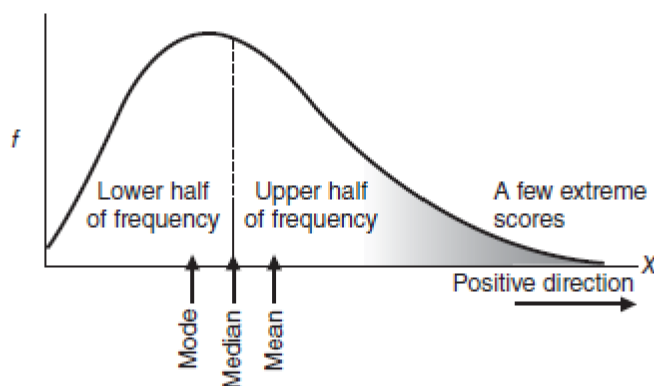
The modal infant death rate of 4 describes the most typical rate (since it occurs most frequently, five times, in Table 3.4).

MSCE-1101

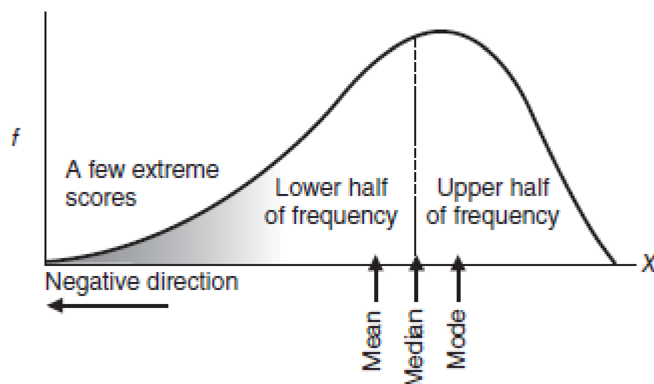
The median infant death rate of 7 describes the middle-ranked rate (since the United States, with a death rate of 7, occupies the middle-ranked, or 10th, position among the 19 ranked countries).
The mean infant death rate of 30.00 describes the balance point for all rates (since the sum of all rates, 570, divided by the number of countries, 19, equals 30.00).
Unlike the mode and median, the mean is very sensitive to extreme scores, or outliers.

Interpreting Differences between Mean and Median

- *Ideally, when a distribution is skewed, report both the mean and the median.*
- Appreciable differences between the values of the mean and median signal the presence of a skewed distribution.
- If the mean exceeds the median the underlying distribution is positively skewed.
- If the median exceeds the mean, the underlying distribution is negatively skewed.



A. Positively Skewed Distribution
(mean exceeds median)



B. Negatively Skewed Distribution
(median exceeds mean)

FIGURE 3.2
Mode, median, and mean in positively and negatively skewed distributions.

3.5 AVERAGES FOR QUALITATIVE AND RANKED DATA

Mode Always Appropriate for Qualitative Data

The mode always can be used with qualitative data.

Median Sometimes Appropriate

The median can be used whenever it is possible to order qualitative data from least to most because the level of measurement is ordinal.

Averages for Ranked Data

When the data consist of a series of ranks, with its ordinal level of measurement, the median rank always can be obtained.

Describing Variability

Variability is the measures of amount by which scores are dispersed or scattered in a distribution.

In **Figure 4.1,** each of the three frequency distributions consists of seven scores with the same mean (10) but with different variabilities. rank the three distributions from least to most variable. Your intuition was correct if you concluded that distribution A has the *least* variability, distribution B has *intermediate* variability, and distribution C has the *most* variability. For distribution A with the least (zero) variability, all seven scores have the same value (10). For distribution B with intermediate variability, the values of scores vary slightly (one 9 and one 11), and for distribution C with most variability, they vary even more (one 7, two 9s, two 11s, and one 13).

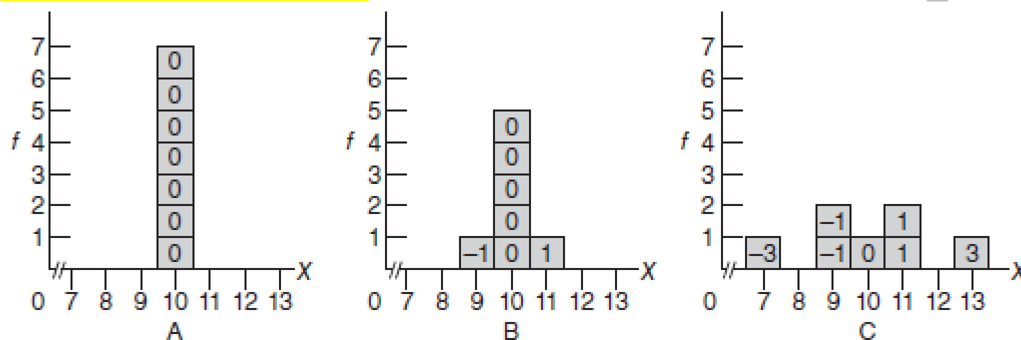


FIGURE 4.1

Three distributions with the same mean (10) but different amounts of variability. Numbers in the boxes indicate distances from the mean.

4.2 RANGE

The range is the difference between the largest and smallest scores.

In Figure 4.1, distribution A, the least variable, has the smallest range of 0 (from 10 to 10); distribution B, the moderately variable, has an intermediate range of 2 (from 11 to 9); and distribution C, the most variable, has the largest range of 6 (from 13 to 7).

Shortcomings of Range

1. The range has several shortcomings. First, since its value depends on only two scores—the largest and the smallest—it fails to use the information provided by the remaining scores.
2. The value of the range tends to increase with increases in the total number of scores.

4.3 VARIANCE

The mean of all squared deviation scores.

the variance also qualifies as a type of mean, that is, as the balance point for some distribution. In the case of the variance, each original score is re-expressed as a distance or deviation from the mean by subtracting the mean.

Reconstructing the Variance

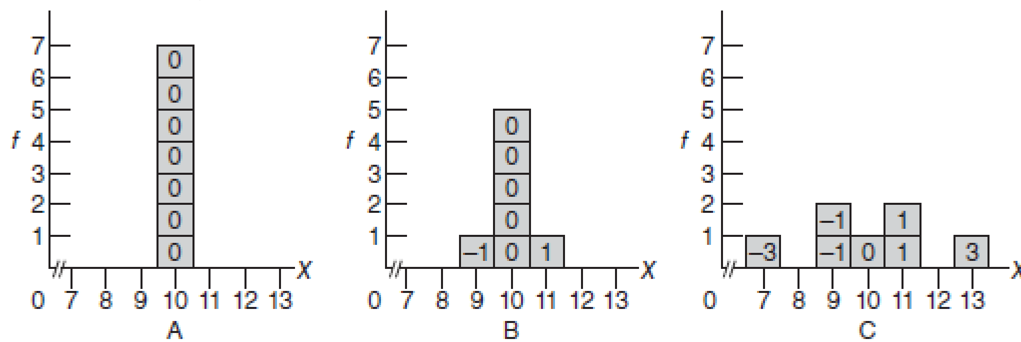


FIGURE 4.1

In distribution C, one score coincides with the mean of 10, four scores (two 9s and two 11s) deviate 1 unit from the mean, and two scores (one 7 and one 13) deviate 3 units from the mean, yielding a set of seven deviation scores: one 0, two –1s, two 1s, one –3, and one 3. (Deviation scores above the mean are assigned positive signs; those below the mean are assigned negative signs.)

Mean of the Squared Deviations

Multiplying each deviation by itself—generates a set of squared deviation scores, all of which are positive. add the consistently positive values of all squared deviation scores and then dividing by the total number of scores to produce the mean of all squared deviation scores, also known as the **variance**.

Example of Variance in Finance

Here's a hypothetical example to demonstrate how variance works. Let's say returns for stock in Company ABC are 10% in Year 1, 20% in Year 2, and −15% in Year 3. The average of these three returns is 5%. The differences between each return and the average are 5%, 15%, and −20% for each consecutive year.

Squaring these deviations yields 0.25%, 2.25%, and 4.00%, respectively. If we add these squared deviations, we get a total of 6.5%. When you divide the sum of 6.5% by one less the number of returns in the data set, as this is a sample (2 = 3-1), it gives us a variance of 3.25% (0.0325). Taking the square root of the variance yields a standard deviation of 18% ($\sqrt{0.0325} = 0.180$) for the returns.

4.4 STANDARD DEVIATION

The square root of the variance. This produces a new measure, known as the standard deviation, that describes variability in the original units of measurement the standard deviation, the square root of the mean of all squared deviations from the mean, that is,

$$standard\ deviation = \sqrt{variance}$$

Majority of Scores within One Standard Deviation

For instance, among the seven deviations in distribution C, a majority of five scores deviate less than one standard deviation (1.77) on either side of the mean. Essentially the same pattern describes a wide variety of frequency distributions including the two shown in Figure 4.3, where the lowercase letter 's' represents the standard deviation. As suggested in the top panel of Figure 4.3,

if the distribution of IQ scores for a class of fourth graders has a mean (X) of 105and a standard deviation (s) of 15, a majority of their IQ scores should be within one standard deviation on either side of the mean, that is, between 90 and 120.

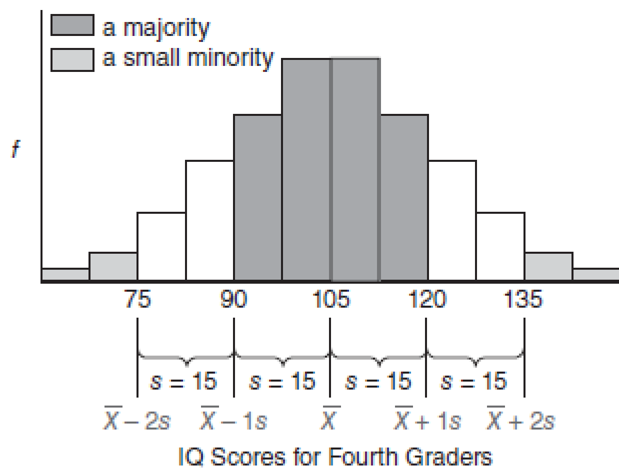


FIGURE 4.3
Some generalizations that apply to most frequency distributions

Standard Deviation: A Measure of Distance

There's an important difference between the standard deviation and its indispensable co-measure, the mean. The mean is a measure of position, but the standard deviation is a measure of distance (on either side of the mean of the distribution).

4.5 DETAILS: STANDARD DEVIATION

Sum of Squares (SS)

Calculating the standard deviation requires that we obtain first a value for the variance. However, calculating the variance requires, in turn, that we obtain the sum of the squared deviation scores. The sum of squared deviation scores, or more simply the sum of squares, symbolized by SS,

There are two formulas for the sum of squares

AMCE 1101

Sum of Squares Formulas for Population

The definition formula provides the most accessible version of the population sum of squares:

SUM OF SQUARES (*SS*) FOR POPULATION (DEFINITION FORMULA)

$$SS = \Sigma(X - \mu)^2$$

SS represents the sum of squares, Σ directs us to sum over the expression to its right, and $(X - \mu)^2$ denotes each of the squared deviation scores.

1. Subtract the population mean, μ, from each original score, X, to obtain a deviation score, $X - \mu$.

2. Square each deviation score, $(X - \mu)^2$, to eliminate negative signs.

3. Sum all squared deviation scores, $\Sigma (X - \mu)^2$.

SUM OF SQUARES (*SS*) FOR POPULATION (COMPUTATION FORMULA)

$$SS = \sum X^2 - \frac{(\sum X)^2}{N}$$

where $\sum X^2$, the sum of the squared X scores, is obtained by *first squaring each X score and then summing all squared X scores;* $\sum X^2$, the square of sum of all *X* scores, is obtained by *first adding all X scores and then squaring the sum of all X scores;* and *N* is the population size.

Table 4.1
CALCULATION OF POPULATION STANDARD DEVIATION σ (DEFINITION FORMULA)

A. COMPUTATION SEQUENCE

Assign a value to *N* **1** representing the number of *X* scores
Sum all *X* scores **2**
Obtain the mean of these scores **3**
Subtract the mean from each *X* score to obtain a deviation score **4**
Square each deviation score **5**
Sum all squared deviation scores to obtain the sum of squares **6**
Substitute numbers into the formula to obtain population variance, σ^2 **7**
Take the square root of σ^2 to obtain the population standard deviation, σ **8**

B. DATA AND COMPUTATIONS

X	**4** $X - \mu$	**5** $(X - \mu)^2$
13	3	9
10	0	0
11	1	1
7	–3	9
9	–1	1
11	1	1
9	–1	1

1 $N = 7$ **2** $\sum X = 70$ **6** $SS = \sum (X - \mu)^2 = 22$

3 $\mu = \frac{70}{7} = 10$

7 $\sigma^2 = \frac{SS}{N} = \frac{22}{7} = 3.14$ **8** $\sigma = \sqrt{\frac{SS}{N}} = \sqrt{\frac{22}{7}} = \sqrt{3.14} = 1.77$

Sum of Squares Formulas for Sample

SUM OF SQUARES (*SS*) FOR SAMPLE (DEFINITION FORMULA)

$$SS = \Sigma(X - \overline{X})^2$$

(COMPUTATION FORMULA)

$$SS = \Sigma X^2 - \frac{(\Sigma X)^2}{n}$$

Table 4.2
CALCULATION OF POPULATION STANDARD DEVIATION (σ) (COMPUTATION FORMULA)

A. COMPUTATIONAL SEQUENCE

Assign a value to *N* representing the number of *X* scores **1**
Sum all *X* scores **2**
Square the sum of all *X* scores **3**
Square each *X* score **4**
Sum all squared *X* scores **5**
Substitute numbers into the formula to obtain the sum of squares, *SS* **6**
Substitute numbers into the formula to obtain the population variance, σ^2 **7**
Take the square root of σ^2 to obtain the population standard deviation, σ **8**

B. DATA AND COMPUTATIONS

X	**4** X^2
13	169
10	100
11	121
7	49
9	81
11	121
9	81

1 $N = 7$ **2** $\sum X = 70$ **5** $\sum X^2 = 722$

3 $(\sum X)^2 = 4900$

6 $SS = \sum X^2 - \frac{(\sum X)^2}{N} = 722 - \frac{4900}{7} = 722 - 700 = 22$

7 $\sigma^2 = \frac{SS}{N} = \frac{22}{7} = 3.14$ **8** $\sigma = \sqrt{\frac{SS}{N}} = \sqrt{\frac{22}{7}} = \sqrt{3.14} = 1.77$

Standard Deviation for Population σ

$$variance = \frac{sum\ of\ all\ squared\ deviation\ scores}{number\ of\ scores}$$

VARIANCE FOR POPULATION

$$\sigma^2 = \frac{SS}{N}$$

STANDARD DEVIATION FOR POPULATION

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{SS}{N}}$$

Standard Deviation for Sample (s)

VARIANCE FOR SAMPLE

$$s^2 = \frac{SS}{n-1}$$

STANDARD DEVIATION FOR SAMPLE

$$s = \sqrt{s^2} = \sqrt{\frac{SS}{n-1}}$$

where s^2 and s represent the sample variance and **sample standard deviation,** SS is the sample sum of squares

Table 4.3
CALCULATION OF SAMPLE STANDARD DEVIATION (S) (DEFINITION FORMULA)

A. COMPUTATION SEQUENCE

Assign a value to n **1** representing the number of X scores
Sum all X scores **2**
Obtain the mean of these scores **3**
Subtract the mean from each X score to obtain a deviation score **4**
Square each deviation score **5**
Sum all squared deviation scores to obtain the sum of squares **6**
Substitute numbers into the formula to obtain the sample variance, s^2 **7**
Take the square root of s^2 to obtain the sample standard deviation, s **8**

B. DATA AND COMPUTATIONS

X	**4** $X - \bar{X}$	**5** $(X - \bar{X})^2$
7	4	16
3	0	0
1	–2	4
0	–3	9
4	1	1

1 $n = 5$ **2** $\Sigma X = 15$ **6** $SS = \Sigma(X - \bar{X})^2 = 30$

3 $\bar{X} = \frac{15}{5} = 3$

7 $s^2 = \frac{SS}{n-1} = \frac{30}{4} = 7.50$ **8** $s = \sqrt{\frac{SS}{n-1}} = \sqrt{\frac{30}{4}} = \sqrt{7.50} = 2.74$

Table 4.4
CALCULATION OF SAMPLE STANDARD DEVIATION (*S*)
(COMPUTATION FORMULA)

A. COMPUTATIONAL SEQUENCE

Assign a value to n representing the number of X scores **1**
Sum all X scores **2**
Square the sum of all X scores **3**
Square each X score **4**
Sum all squared X scores **5**
Substitute numbers into the formula to obtain the sum of squares, SS **6**
Substitute numbers into the formula to obtain the sample variance, s^2 **7**
Take the square root of s^2 to obtain the sample standard deviation, s **8**

B. DATA AND COMPUTATIONS

X	**4** X^2
7	49
3	9
1	1
0	0
4	16

1 $n = 5$ **2** $\Sigma X = 15$ **5** $\Sigma X^2 = 75$

3 $(\Sigma X)^2 = 225$

6 $$SS = \sum X^2 - \frac{(\sum X)^2}{n} = 75 - \frac{225}{5} = 75 - 45 = 30$$

7 $$s^2 = \frac{SS}{n-1} = \frac{30}{4} = 7.50$$ **8** $$s = \sqrt{\frac{SS}{n-1}} = \sqrt{\frac{30}{4}} = \sqrt{7.50} = 2.74$$

4.6 DEGREES OF FREEDOM (*df*)

***Degrees of freedom (df)* refers to the number of values that are free to vary, given one or more mathematical restrictions, in a sample being used to *estimate* a population characteristic.**

when n deviations about the sample mean are used to estimate variability in the population, only $n - 1$ are free to vary. As a result, there are only $n - 1$ degrees of freedom, that is, $df = n - 1$. One df is lost because of the zero-sum restriction.

VARIANCE FOR SAMPLE

$$s^2 = \frac{SS}{n-1} = \frac{SS}{df}$$

STANDARD DEVIATION FOR SAMPLE

$$s = \sqrt{\frac{SS}{n-1}} = \sqrt{\frac{SS}{df}}$$

where s^2 and s represent the sample variance and standard deviation, SS is the sum of squares, df is the degrees of freedom and equals $n - 1$.

4.7 INTERQUARTILE RANGE (IQR)

Table 4.6
CALCULATION OF THE IQR

A. INSTRUCTIONS

1. Order scores from least to most.
2. To determine how far to penetrate the set of ordered scores, begin at either end, then add 1 to the total number of scores and divide by 4. If necessary, round the result to the nearest whole number.
3. Beginning with the largest score, count the requisite number of steps (calculated in step 2) into the ordered scores to find the location of the third quartile.
4. The third quartile equals the value of the score at this location.
5. Beginning with the smallest score, again count the requisite number of steps into the ordered scores to find the location of the first quartile.
6. The first quartile equals the value of the score at this location.
7. The IQR equals the third quartile minus the first quartile.

B. EXAMPLE

1. 7, 9, 9, 10, 11, 11, 13
2. (7 + 1)/4 = 2
3. 7, 9, 9, 10, 11, 11, 13

4. third quartile = 11
5. 7, 9, 9, 10, 11, 11, 13

6. first quartile = 9
7. IQR = 11 − 9 = 2

AN

5. Normal Distributions and Standard (*z*) Scores

5.1 THE NORMAL CURVE

FBI agents are to be selected only from among applicants who are no taller than exactly 66 inches, what proportion of all of the original applicants will be eligible? This question can't be answered without additional information. One source of additional information is the relative frequency distribution of heights for the 3091 men shown in **Figure 5.1**.

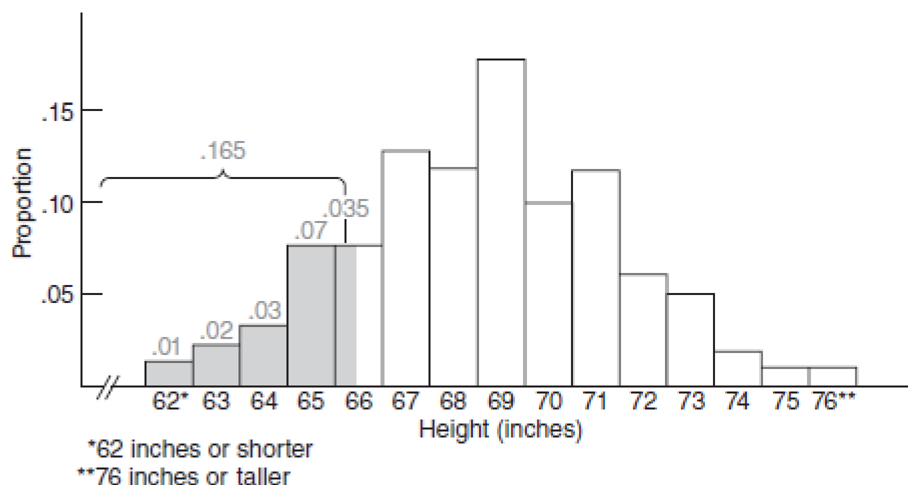


FIGURE 5.1

Relative frequency distribution for heights of 3091 men. Source: National Center for Health Statistics, 1960–62, Series 11, No.14. Mean updated by authors.

.10 of these men, that is, one-tenth of 3091, (3091/10) or about 309 men, are 70 inches tall. .10 of these men, that is, one-tenth of 3091, or about 309 men, are 70 inches tall. Only half of the bar at 66 inches is shaded to adjust for the fact that any height between 65.5 and 66.5 inches is reported as 66 inches, whereas eligible applicants must be shorter than *exactly* 66 inches, that is, 66.0 inches.

Properties of the Normal Curve

Let's note several important properties of the normal curve:

■Obtained from a mathematical equation, the **normal curve** is a theoretical curve defined for a continuous variable, and noted for its symmetrical bell-shaped form, as revealed in Figure 5.2.

■ Because the normal curve is symmetrical, its lower half is the mirror image of its upper half.

■ Being bell shaped, the normal curve peaks above a point midway along the horizontal spread and then tapers off gradually in either direction from the peak (without actually touching the horizontal axis, since, in theory, the tails of a normal curve extend infinitely far).

■ The values of the mean, median (or 50th percentile), and mode, located at a point midway along the horizontal spread, are the same for the normal curve.

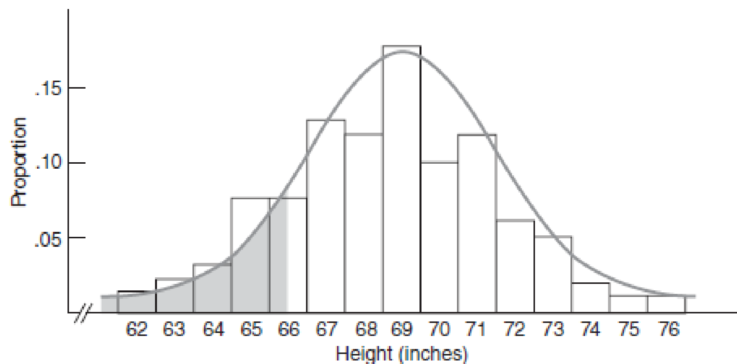


FIGURE 5.2

Normal curve superimposed on the distribution of heights.

Different Normal Curves

For example, changing the mean height from 69 to 79 inches produces a new normal curve that, as shown in panel A of **Figure 5.3**, is displaced 10 inches to the right of the original curve. Dramatically new normal curves are produced by changing the value of the standard deviation. As shown in panel B of Figure 5.3, changing the standard deviation from 3 to 1.5 inches produces a more peaked normal curve with smaller variability, whereas changing the standard deviation from 3 to 6 inches produces a shallower normal curve with greater variability.

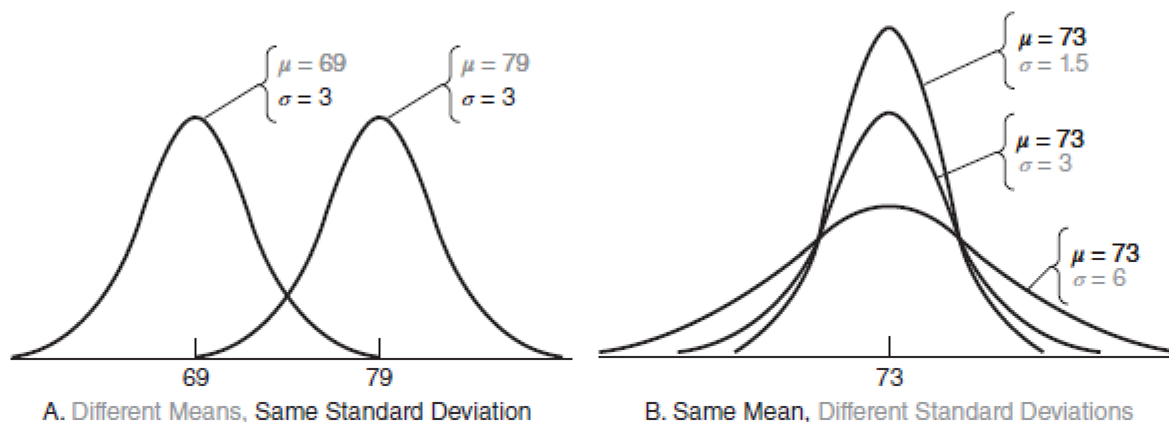


FIGURE 5.3

Different normal curves.

Because of their common mathematical origin, every normal curve can be interpreted in exactly the same way *once any distance from the mean is expressed in standard deviation units*.

5.2 *z* SCORES

A *z* score is a unit-free, standardized score that, regardless of the original units of measurement, indicates how many standard deviations a score is above or below the mean of its distribution.

To obtain a *z* score, express any original score, whether measured in inches, milliseconds, dollars, IQ points, etc., as a deviation from its mean (by subtracting its mean) and then split this deviation into standard deviation units (by dividing by its standard deviation), that is,

z SCORE

$$z = \frac{X - \mu}{\sigma}$$

where X is the original score and μ and σ are the mean and the standard deviation, respectively,

A z score consists of two parts:
1. *a positive or negative* sign indicating whether it's above or below the mean; and
2. *a number* indicating the size of its deviation from the mean in standard deviation units.

Converting to z Scores
To answer the question about eligible FBI applicants, replace X with 66 (the maximum permissible height), μ with 69 (the mean height), and σ with 3 (the standard deviation of heights) and solve for z as follows:

$$\frac{66 - 69}{3} = \frac{-3}{3} = -1$$

This informs us that the cutoff height is exactly one standard deviation below the mean. Knowing the value of z, we can use the table for the standard normal curve to find the proportion of eligible FBI applicants. First, however, we'll make a few comments about the standard normal curve.

5.3 STANDARD NORMAL CURVE
If the original distribution approximates a normal curve, then the shift to standard or z scores will always produce a new distribution that approximates the **standard normal curve**. The standard normal curve always has a mean of 0 and a standard deviation of 1.
However, to verify (rather than prove) that the mean of a standard normal distribution equals 0, replace X in the z score formula with μ, the mean of any (nonstandard) normal distribution, and then solve for z:

$$\text{Mean of } z = \frac{X - \mu}{\sigma} = \frac{\mu - \mu}{\sigma} = \frac{0}{\sigma} = 0$$

to verify that the standard deviation of the standard normal distribution equals 1, replace X in the z score formula with $\mu + 1\sigma$, the value corresponding to one standard deviation above the mean for any (nonstandard) normal distribution, and then solve for z:

$$\text{Standard deviation of } z = \frac{X - \mu}{\sigma} = \frac{\mu + 1\sigma - \mu}{\sigma} = \frac{1\sigma}{\sigma} = 1$$

Although there is an infinite number of different normal curves, each with its own mean and standard deviation, there is only one standard normal curve, with a mean of 0 and a standard deviation of 1.

AMSE-101

TABLE 5.1
PROPORTIONS (OF AREAS) UNDER THE STANDARD NORMAL CURVE FOR VALUES OF z (FROM TABLE A OF APPENDIX C)

A z	B	C	A z	B	C	A z	B	C
0.00	.0000	.5000	0.40	.1554	.3446	0.80	.2881	.2119
0.01	.0040	.4960	0.41	.1591	.3409	0.81	.2910	.2090
•	•	•	•	•	•	•	•	•
						•	•	•
						•	•	•
						•	•	•
•	•	•	•	•	•	•	•	•
						0.99	.3389	.1611
						1.00	.3413	.1587
•	•	•	•	•	•	1.01	.3438	.1562
						•	•	•
						•	•	•
•	•	•	•	•	•	•	•	•
0.38	.1480	.3520	0.78	.2823	.2711	1.18	.3810	.1190
0.39	.1517	.3483	0.79	.2852	.2148	1.19	.3830	.1170
−z A′	B′	C′	−z A′	B′	C′	−z A′	B′	C′

Z score

Page 458

AMS

5.4 SOLVING NORMAL CURVE PROBLEMS

using rough graphs of normal curves as an aid to visualizing the solution

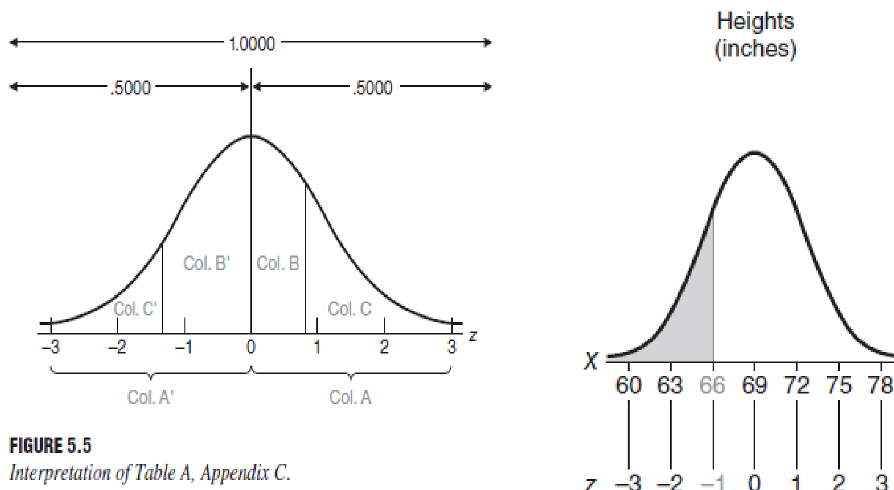


FIGURE 5.5
Interpretation of Table A, Appendix C.

Key Facts to Remember

for any z score, the corresponding proportions in columns B and C (or columns B′ and C′) always sum to .5000. Similarly, the total area under the normal curve always equals 1.0000, the sum of the proportions in the lower and upper halves, that is, .5000 + .5000. Finally, although a z score can be either positive or negative, the proportions of area under the curve are always positive or zero but *never* negative (because an area cannot be negative). **Figure 5.5** summarizes how to interpret the normal curve table in this book.

5.5 FINDING PROPORTIONS

1. **Sketch a normal curve and shade in the target area**
2. **Plan your solution according to the normal table**.

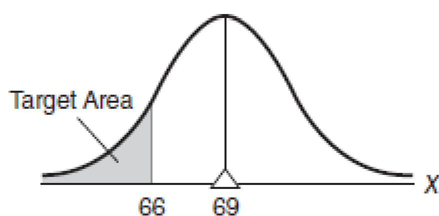
Decide precisely how you will find the value of the target area. In the present case, the answer will be obtained from column C′ of the standard normal table, since the target area coincides with the type of area identified with column C′, that is, the area in the lower tail beyond a negative z.

3. **Convert X to z.**

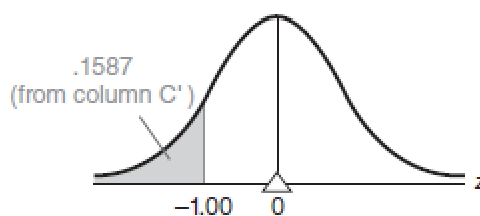
Convert X to z. Express 66 as a z score:

$$z = \frac{X - \mu}{\sigma} = \frac{66 - 69}{3} = \frac{-3}{3} = -1$$

Find: Proportion Below 66



Solution:



Answer: .1587

4. **Find the target area**. Refer to the standard normal table, using the bottom legend, as the z score is negative. The arrows in Table 5.1 show how to read the table. Look up column A’ to 1.00 (representing a z score of –1.00), and note the corresponding proportion of .1587 in column C’: This is the answer, as suggested in the right part of

Figure 5.6. It can be concluded that only .1587 (or .16) of all of the FBI applicants will be shorter than 66 inches.

Example: Finding Proportions *between* Two Scores

Find: Proportion Between 245 and 255

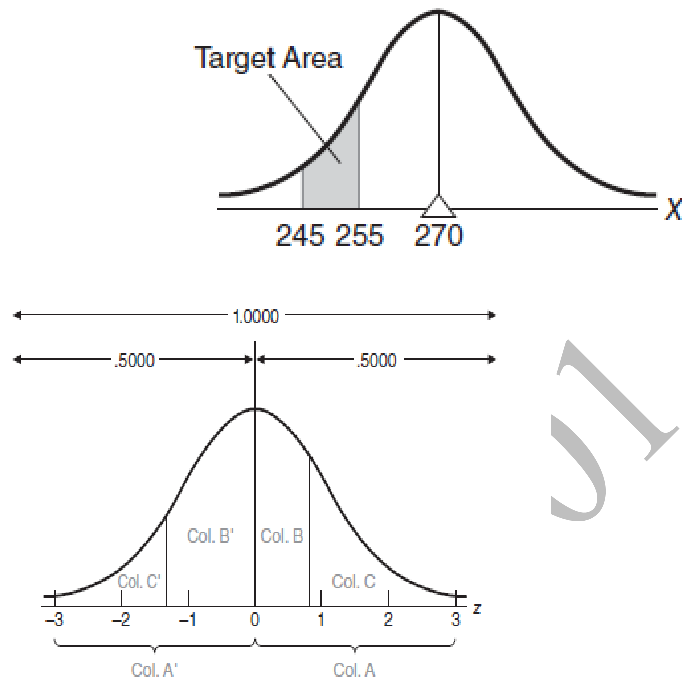


FIGURE 5.5
Interpretation of Table A, Appendix C.

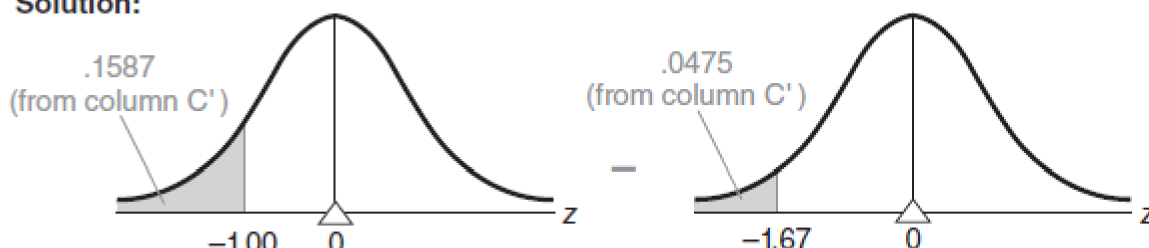
Convert X to z by expressing 255 as

$$z = \frac{255-270}{15} = \frac{-15}{15} = -1.00$$

and by expressing 245 as

$$z = \frac{245-270}{15} = \frac{-25}{15} = -1.67$$

Solution:



Answer: .1587
− .0475
.1112

Look up column A′ to a negative z score of –1.00 (remember, you must imagine the negative sign), and note the corresponding proportion of .1587 in column C′. Likewise, look up

column A′ to a z score of –1.67, and note the corresponding proportion of .0475 in column C′.
Subtract

AMSCE-1101

UNIT III

6.2 SCATTERPLOTS

A **scatterplot** *is a graph containing a cluster of dots that represents all pairs of scores.*

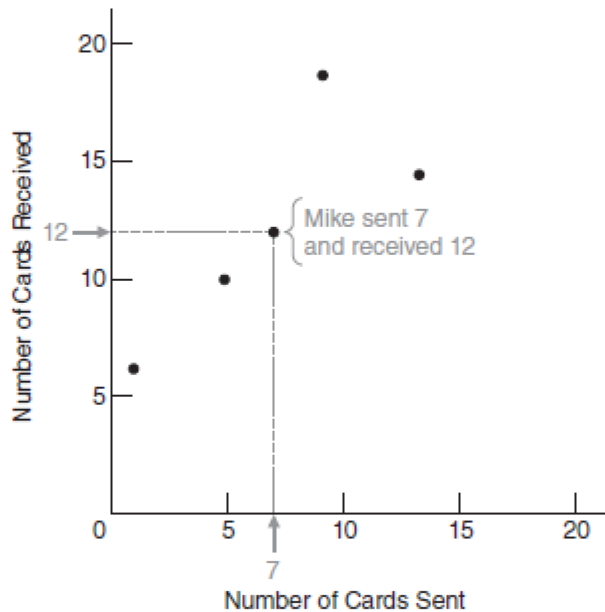


FIGURE 6.1
Scatterplot for greeting card exchange.

MSC-1101

Positive, Negative, or Little or No Relationship?

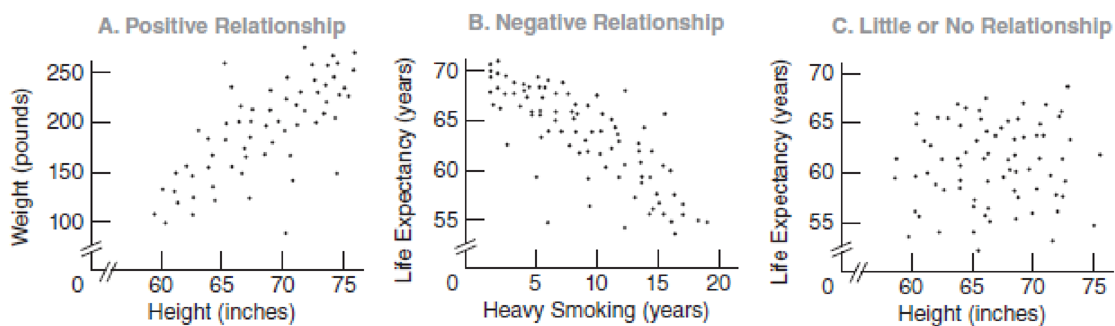


FIGURE 6.2
Three types of relationships.

A dot cluster that has a slope from the lower left to the upper right, as in panel A of **Figure 6.2,** *reflectsa positive relationship.* Small values of one variable are paired with small values of the other variable, and large values are paired with large values.

Strong or Weak Relationship?

The more closely the dot cluster approximates a straight line, the stronger (the more regular) the relationship will be.

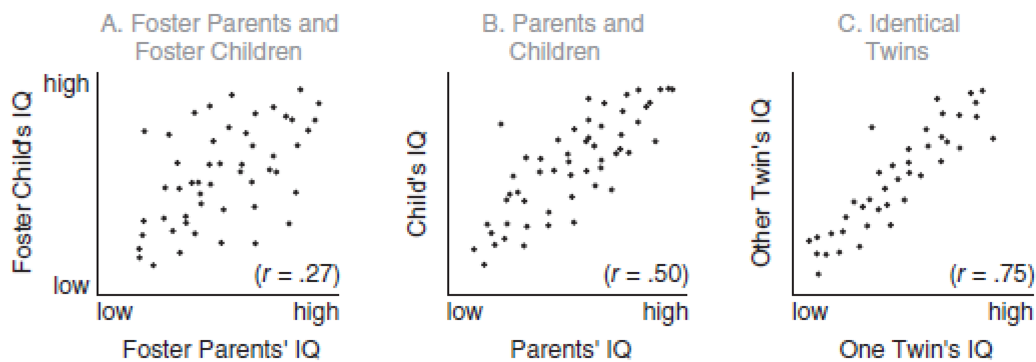


FIGURE 6.3
Three positive relationships. (Scatterplots simulated from a 50-year literature survey.) Source: Erlenmeyer-Kimling, L., & Jarvik, L. F. (1963). "Genetics and Intelligence: A Review." Science, 142, 1477–1479.

Perfect Relationship

A dot cluster that equals (rather than merely approximates) a straight line reflects a perfect relationship between two variables. In practice, perfect relationships are most unlikely.

Curvilinear Relationship

Sometimes adot cluster approximates a *bent* or *curved* line, as in **Figure 6.4,** and therefore reflectsa **curvilinear relationship.** Descriptions of these relationships are more complex thanthose of linear relationships.

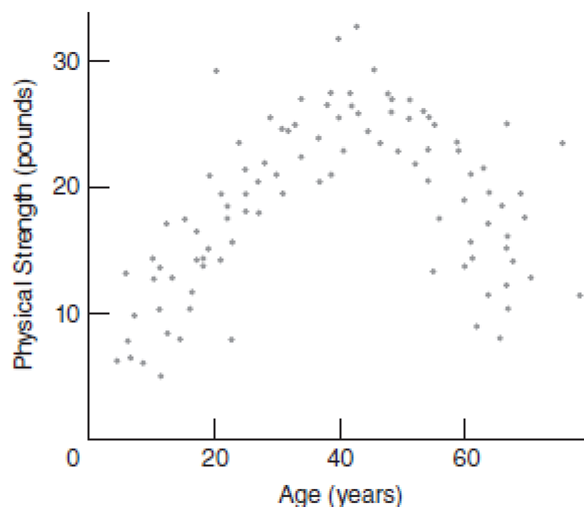


FIGURE 6.4
Curvilinear relationship.

6.3 A CORRELATION COEFFICIENT FOR QUANTITATIVE DATA : *r*

A **correlation coefficient** is a number between –1 and 1 that describes the relationshipbetween pairs of variables.

Key Properties of *r*

1. *The sign of **r** indicates the type of linear relationship, whether positive or negative.*
2. *The numerical value of **r**, without regard to sign, indicates the strength of the linear relationship.*

Sign of *r*

A number with a plus sign (or no sign) indicates a positive relationship, and a number with a minus sign indicates a negative relationship. For example, an *r* with a plus sign describes the positive relationship between height and weight shown in panel A of Figure 6.2, and an *r* with a minus sign describes the negative relationship between heavy smoking and life expectancy shown in panel B.

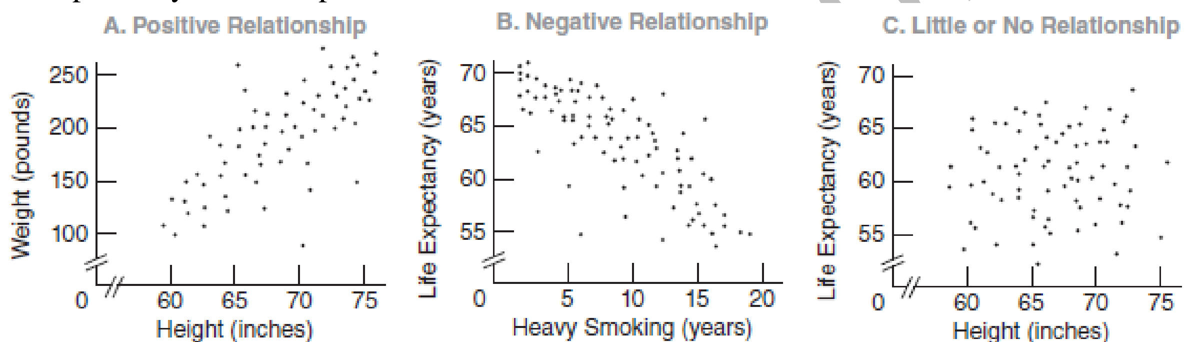


FIGURE 6.2
Three types of relationships.

Numerical Value of *r*

The more closely a value of r approaches either –1.00 or +1.00, the stronger (more regular) the relationship. Conversely, the more closely the value of r approaches 0, the weaker (less regular) the relationship. Figure 6.3, notice that the values of r shift from .75 to .27 as the analysis for pairs of IQ scores shifts from a relatively strong relationship for identical twins to a relatively weak relationship for foster parents and foster children.

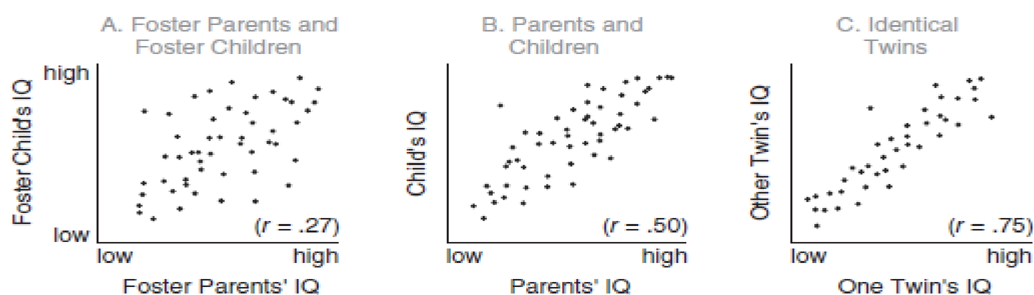


FIGURE 6.3
Three positive relationships. (Scatterplots simulated from a 50-year literature survey.) Source: Erlenmeyer-Kimling, L., & Jarvik, L. F. (1963). "Genetics and Intelligence: A Review." Science, 142, 1477–1479.

Interpretation of *r*

Located along a scale from –1.00 to +1.00, the value of *r* supplies information about the direction of a linear relationship—whether positive or negative—and, generally, information about the relative strength of a linear relationship—whether relatively weak (and a poor describer of the data) because *r* is in the vicinity of 0, or relatively strong (and a good describer of the data) because *r* deviates from 0 in the direction of either +1.00 or –1.00.

***r* Is Independent of Units of Measurement**

The value of *r* is independent of the original units of measurement.

Range Restrictions

The value of the correlation coefficient declines whenever the range of possible *X* or *Y* scores is restricted.

For example, **Figure 6.5** shows a dot cluster with an obvious slope, represented by an *r* of .70 for the positive relationship between height and weight for all college students. If, however, the range of heights along *Y* is restricted to students who stand over 6 feet 2 inches (or 74 inches) tall, the abbreviated dot cluster loses its obvious slope because of the more homogeneous weightsamong tall students. Therefore, as depicted in Figure 6.5, the value of *r* drops to .10.

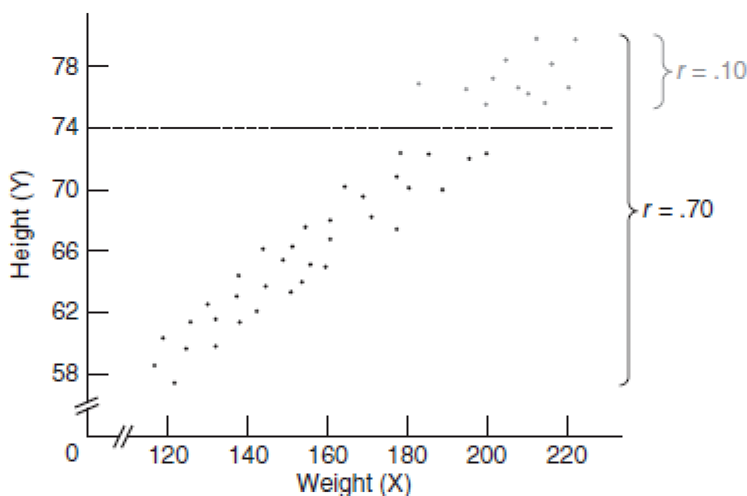


FIGURE 6.5
Effect of range restriction on the value of r.

Verbal Descriptions

An *r* of .70 for the height and weight of college students could be translated into “Taller students tend to weigh more”

6.4 DETAILS: COMPUTATION FORMULA FOR *r*

Calculate a value for *r* by using the following computation formula:

CORRELATION COEFFICIENT (COMPUTATION FORMULA)

$$r = \frac{SP_{xy}}{\sqrt{SS_x SS_y}}$$

where the two sum of squares terms in the denominator are defined as

$$SS_x = \Sigma\left(X - \bar{X}\right)^2 = \Sigma X^2 - \frac{(\Sigma X)^2}{n}$$

$$SS_y = \Sigma\left(Y - \bar{Y}\right)^2 = \Sigma Y^2 - \frac{(\Sigma Y)^2}{n}$$

and the sum of the products term in the numerator SP_{xy}:

SUM OF PRODUCTS (DEFINITION AND COMPUTATION FORMULAS)

$$SP_{xy} = \Sigma\left(X - \bar{X}\right)\left(Y - \bar{Y}\right) = \Sigma XY - \frac{(\Sigma X)(\Sigma Y)}{n}$$

AMSC 1101

Table 6.3
CALCULATION OF *r*: COMPUTATION FORMULA

A. COMPUTATIONAL SEQUENCE

Assign a value to n (1), representing the number of pairs of scores.
Sum all scores for X (2) and for Y (3).
Find the product of each pair of X and Y scores (4), one at a time, then add all of these products (5).
Square each X score (6), one at a time, then add all squared X scores (7).
Square each Y score (8), one at a time, then add all squared Y scores (9).
Substitute numbers into formulas (10) and solve for SP_{xy}, SS_x, and SS_y.
Substitute into formula (11) and solve for r.

B. DATA AND COMPUTATIONS

	CARDS		4	6	8
FRIEND	**SENT, *X***	**RECEIVED, *Y***	***XY***	***X*²**	***Y*²**
Doris	13	14	182	169	196
Steve	9	18	162	81	324
Mike	7	12	84	49	144
Andrea	5	10	50	25	100
John	1	6	6	1	36

1 $n = 5$ 2 $\Sigma X = 35$ 3 $\Sigma Y = 60$ 5 $\Sigma XY = 484$ 7 $\Sigma X^2 = 325$ 9 $\Sigma Y^2 = 800$

10 $$SP_{xy} = \Sigma XY - \frac{(\Sigma X)(\Sigma Y)}{n} = 484 - \frac{(35)(60)}{5} = 484 - 420 = 64$$

$$SS_x = \Sigma X^2 - \frac{(\Sigma X)^2}{n} = 325 - \frac{(35)^2}{5} = 325 - 245 = 80$$

$$SS_y = \Sigma Y^2 - \frac{(\Sigma Y)^2}{n} = 800 - \frac{(60)^2}{5} = 800 - 720 = 80$$

11 $$r = \frac{SP_{xy}}{\sqrt{SS_x SS_y}} = \frac{64}{\sqrt{(80)(80)}} = \frac{64}{80} = .80$$

7.2 A REGRESSION LINE

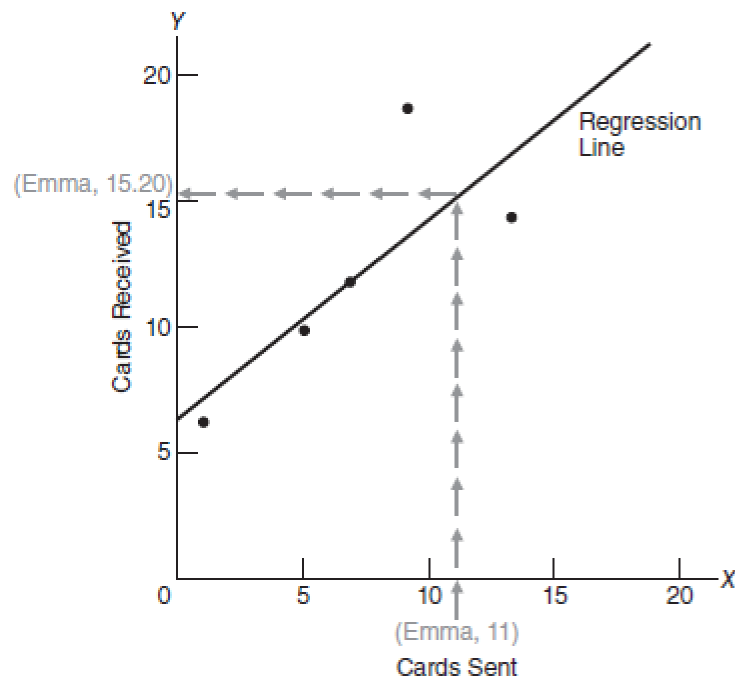


FIGURE 7.2
Prediction of 15.20 for Emma (using the regression line).

Predictive Errors
Figure 7.3 illustrates the predictive errors that would have occurred if the regression line had been used to predict the number of cards received by the five friends.

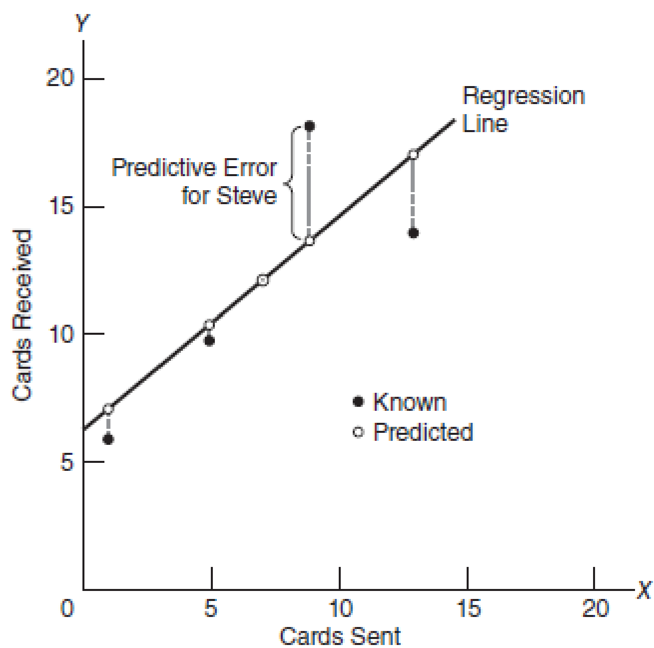


FIGURE 7.3
Predictive errors.

7.3 LEAST SQUARES REGRESSION LINE

The placement of the regression line minimizes **not the total predictiveerror but *the total squared predictive error*. that is, the total for all squared predictive errors. When located in this fashion, the *regression* line is often referred to as the *leastsquares regression line*.**

Least Squares Regression Equation

LEAST SQUARES REGRESSION EQUATION

$$Y' = bX + a$$

Y' represents the predicted value
X represents the known value
and b and a represent numbers calculated from the original correlation analysis,

Finding Values of *b* and *a*

SOLVING FOR *b*

$$b = r\sqrt{\frac{SS_y}{SS_x}}$$

where r represents the correlation between X and Y

SS_yrepresents the sum of squares for all Y scores

SS_xrepresents the sum of squares for all X scores

SOLVING FOR *a*

$$a = \overline{Y} - b\overline{X}$$

Where $\overline{Y}$ and $\overline{X}$ refers to mean values of X and Y scores.

AMSCE-1101

Table 7.1
DETERMINING THE LEAST SQUARES REGRESSION EQUATION

A. COMPUTATIONAL SEQUENCE

Determine values of SS_x, SS_y, and r (1) by referring to the original correlation analysis in Table 6.3.
Substitute numbers into the formula (2) and solve for b.
Assign values to $\overline{X}$ and $\overline{Y}$ (3) by referring to the original correlation analysis in Table 6.3.
Substitute numbers into the formula (4) and solve for a.
Substitute numbers for b and a in the least squares regression equation (5).

B. COMPUTATIONS

1 $SS_x = 80^*$
$SS_y = 80^*$
$r = .80$

2 $$b = r\sqrt{\frac{SS_Y}{SS_X}} = .80\sqrt{\frac{80}{80}} = .80$$

3 $\overline{X} = 7^{**}$
$\overline{Y} = 12^{**}$

4 $$a = \overline{Y} - (b)(\overline{X}) = 12 - (.80)(7) = 12 - 5.60 = 6.40$$

5 $$Y' = (b)(X) + a$$
$$= (.80)(X) + 6.40$$

=.80*13+6.40
=16.8

13	14
9	18
7	12
5	10
1	6

7.4 STANDARD ERROR OF ESTIMATE, $s_{y|x}$ (s sub y given x.”) ***[Error caused during prediction]***

$$s_{y|x} = \sqrt{\frac{SS_{y|x}}{n-2}} = \sqrt{\frac{\Sigma(Y-Y')^2}{n-2}}$$

$SSy|x$, represents the sum of the squares for predictive errors, $Y - Y'$

STANDARD ERROR OF ESTIMATE (COMPUTATION FORMULA)

$$s_{y|x} = \sqrt{\frac{SS_y\left(1-r^2\right)}{n-2}}$$

where SSy is the sum of the squares for Y scores (cards received by the five friends), that is,

$$SS_y = \Sigma(Y-\bar{Y}) = \Sigma Y^2 - \frac{(\Sigma Y)^2}{n}$$

Table 7.3
CALCULATION OF THE STANDARD ERROR OF ESTIMATE, $s_{y|x}$

A. COMPUTATIONAL SEQUENCE

Assign values to SS_y and r (1) by referring to previous work with the least squares regression equation in Table 7.1.

Substitute numbers into the formula (2) and solve for $s_{y|x}$.

B. COMPUTATIONS

1 $SS_y = 80$

$r = .80$

2 $s_{y|x} = \sqrt{\frac{SS_y(1-r^2)}{n-2}} = \sqrt{\frac{80\left(1-[.80]^2\right)}{5-2}} = \sqrt{\frac{80(.36)}{3}} = \sqrt{\frac{28.80}{3}} = \sqrt{9.60}$

$= 3.10$

7.5 ASSUMPTIONS

Linearity

Use of the regression equation requires that the underlying relationship be linear. You need to worry about violating this assumption only when the scatterplot for the original correlation analysis reveals an obviously bent or curvilinear dot cluster, such as illustrated in Figure 6.4. In the unlikely event that a dot cluster describes a pronounced curvilinear trend consult advanced statistics technique.

Homoscedasticity

Use of the standard error of estimate, sy|x, assumes that except for chance, *the dots in the original scatterplot will be dispersed equally about all segments of the regression line*. You need to worry about violating this assumption *homoscedasticity* only when the scatterplot reveals a dramatically different
type of dot cluster such as that shown in **Figure 7.4**

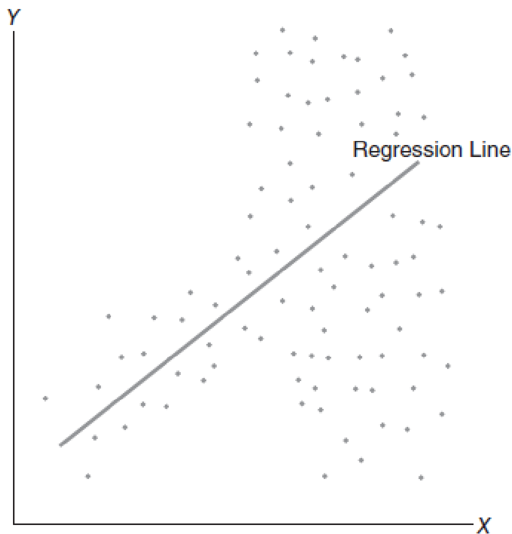


Figure 7.4

7.6 INTERPRETATION OF r^2

Squared correlation coefficient, r^2 A measure of predictive accuracy that supplements the*standard error of estimate*, *sy*|*x*. even though our ultimate goal is to show therelationship between *r*2 and predictive accuracy, we will initially concentrate on two kinds of predictive errors—those due to the repetitive prediction of the mean and those due to the regression equation.

AMSCE-1101

Repetitive Prediction of the Mean

- we know the *Y* scores (cards received), but not the corresponding *X* scores (cards sent).
- Lacking information about the relationship between *X* and *Y* scores, we could not construct a *regression* equation
- We could, however, mount a primitive predictive effort by *always* predicting the mean, *Y*, for each of the five friends' *Y* scores.
- using the repetitive prediction of *Y* for each of the *Y* scores of all five friends will supply us with a *frame of reference against which to evaluate our customary predictive effort* based on the correlation between cards sent (*X*) and cards received (*Y*).

Predictive Errors

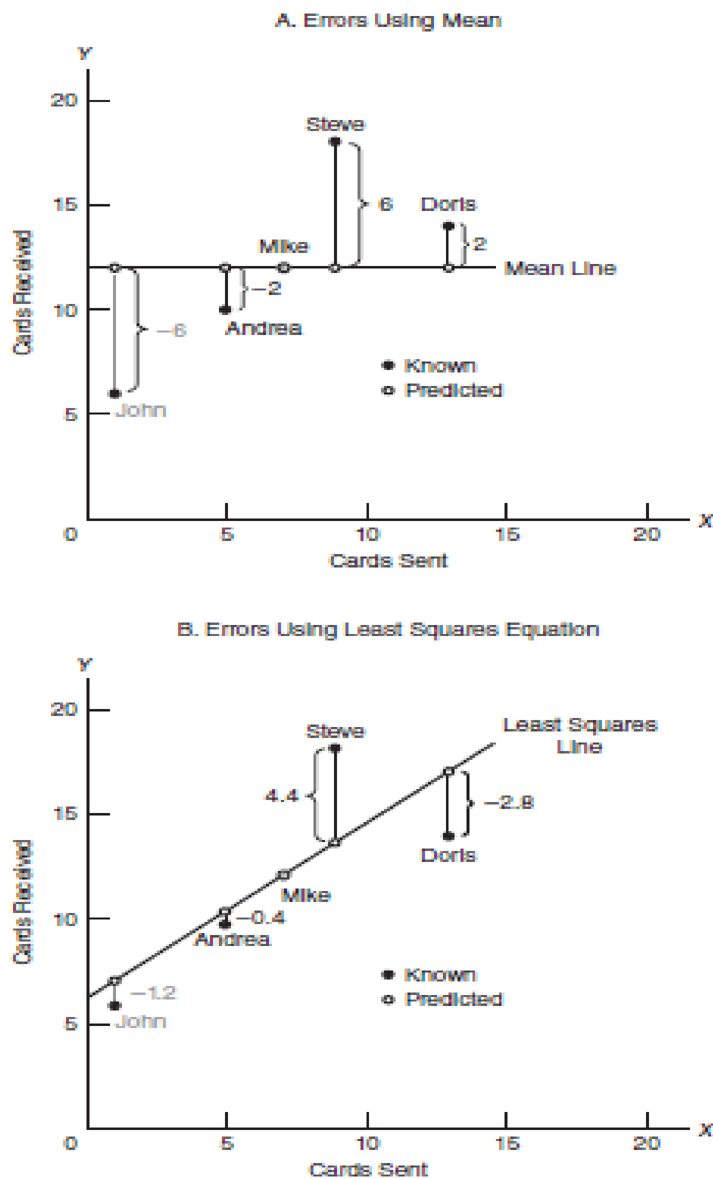


FIGURE 7.5
Predictive errors for five friends.

Panel A of **Figure 7.5** shows the predictive errors for all five friends when the mean for all five friends, *Y*, of 12 (shown as the mean line) is *always* used to predict each of their five *Y* scores. Panel B shows the corresponding predictive errors for all five friends when a series of different *Y′* values, obtained from the least squares equation (shown as the least squares line), is used to predict each of their five *Y* scores.

Positive and negative errors indicate that *Y* scores are either above or below theircorresponding predicted scores.
Overall, as expected, errors are smaller when customized predictions of *Y′* from the least squares equation can be used than when only the repetitive prediction of *Y* can be used.

Error Variability (Sum of Squares)
To more precisely evaluate the accuracy of our two predictive efforts, we need some measure. sum of squares is used to measure collective errors produced by these efforts. sum of squares calculate by first squaring each error (to eliminate negative signs), then summing all squared errors.

The error variability for the repetitive prediction of the mean can be designated as *SSy*. since each *Y* score is expressed as a squared deviation from $\overline{Y}$ and then summed.

$$SS_y = \sum(Y - \overline{Y})^2$$

Using the errors for the five friends shown in Panel A of Figure 7.5, this becomes

$$SS_y = [(-6)^2 + (-2)^2 + 0^2 + 6^2 + 2^2] = 80$$

The error variability for the customized predictions from the least squares equation
can be designated as *SSy*/*x*

$$SS_{y|x} = \sum(Y - Y')^2$$

Using the errors for the five friends shown in Panel B of Figure 7.5, we obtain:

$$SS_{y|x} = [(-1.2)^2 + (-0.4)^2 + 0^2 + (4.4)^2 + (-2.8)^2] = 28.8$$

Proportion of *Predicted* Variability

*SSy*measures the *total* variability
SSy/*x*measures the *residual* variability
The error variability of 28.8 for the least squares predictions is much smaller than the error variability of 80 for the repetitive prediction of *Y*, confirming the greater accuracy of the least squares predictions
apparent in Figure 7.5

To obtain an *SS* measure of the actual *gain in accuracy* due to the least squarespredictions, subtract the residual variability from the total variability, that is, subtract
SSy/*x*from *SSy*, to obtain

$$SS_y - SS_{y|x} = 80 - 28.8 = 51.2$$

To express this difference, 51.2, as a gain in accuracy *relative* to the original errorvariability for the repetitive prediction of *Y*,

$$\frac{SS_y - SS_{y|x}}{SS_y} = \frac{80 - 28.8}{80} = \frac{51.2}{80} = .64$$

This result, .64 or 64 percent, represents the proportion or percent gain in predictive accuracy.

AMSCE 1101

Square the value of 0.80 in previous problem yields .64.

the square of the correlation coefficient, *r*2, *always* indicates the proportion oftotal variability in one variable that is predictable from its relationship with theother variable.

$$r^2 = \frac{SS_{Y'}}{SS_Y} = \frac{SS_Y - SS_{Y|X}}{SS_Y}$$

SS_{y} *is* variability explained by or predictable from the regression equation

$$SS_{y'} = \sum(Y' - \overline{Y})^2$$

r^2 Does Not Apply to Individual Scores

Small Values of r^2

a value of r^2 in the vicinity of .01, .09, or .25 reflects a weak, moderate, or strongrelationship, respectively

r^2provides us with a straightforward measure of the worth of our least squares predictive effort

7.7 MULTIPLE REGRESSION EQUATIONS

General form : y = mx1 + mx2+ mx3+ b

$Y' = .410(X1) + .005(X2) + .001(X3) + 1.03$

Y′ represents predicted college GPA
*X*1, *X*2, and *X*3 refer to high school GPA, IQ score, and SAT score,
these **multiple regression equations** supply more accurate predictions for *Y′* (often referred to as the *criterion variable*)

AMSCE-1101

7.8 REGRESSION TOWARD THE MEAN

Regression toward the mean *refers to a tendency for scores, particularly extreme scores, to shrink toward the mean.*

Table 7.4 lists the top 10 hitters in the major leagues during 2014 and shows how they fared during 2015. Notice that 7 of the top 10 batting averages regressed downward, toward **260s**, the approximate mean for all hitters during 2015. Incidentally, it is not true that, viewed as a group, all major league hitters are headed toward mediocrity. Hitters among the top 10 in 2014, who were not among the top 10 in 2015, were replaced by other mostly above-average hitters, who also were very lucky during 2015. Observed regression toward the mean occurs for individuals or subsets of individuals, not for entire groups.

Table 7.4
REGRESSION TOWARD THE MEAN: BATTING AVERAGES OF TOP 10 HITTERS IN MAJOR LEAGUE BASEBALL DURING 2014 AND HOW THEY FARED DURING 2015

	BATTING AVERAGES*		
TOP 10 HITTERS (2014)	2014	2015	REGRESS TOWARD MEAN?
1. J. Alture	.341	.313	Yes
2. V. Martinez	.335	.282	Yes
3. M. Brantley	.327	.310	Yes
4. A. Beltre	.324	.287	Yes
5. J. Abreu	.317	.290	Yes
6. R. Cano	.314	.287	Yes
7. A. McCutchen	.314	.292	Yes
8. M. Cabrera	.313	.338	No
9. B. Posey	.311	.318	No
10. B. Revere	.306	.306	No

The Regression Fallacy

The **regression fallacy** *is committed whenever regression toward the mean is interpreted as a real, rather than a chance, effect.*

Example: Aeroplane landing effect

Some trainees were praised after very good landings, while others were reprimanded after very bad landings. On their next landings, praised trainees did more poorly and reprimanded trainees did better. **It was concluded, therefore, that praise hinders but a reprimand helps performance!**

A valid conclusion considers regression toward the mean. It's reasonable to assume that, **in addition to skill, chance plays a role in landings**. Some trainees who made very good landings were **lucky.** while some who made very bad landings were unlucky.

Avoiding the Regression Fallacy.

The regression fallacy can be avoided by **splitting the subset of extreme observations into two groups**. In the previous example, **one group of trainees would continue to be praised after very good** landings and reprimanded after very poor landings. A **second group of trainees would receive no feedback whatsoever after very good and very bad landings.** In effect, the **second group would serve as a control for regression toward the mean,** since **any shift toward the mean on their second landings would be due to chance.** Most important, any **observed difference between the two groups would be viewed as a real difference not attributable to the regression effect.**

AMSCE-1101

UNIT IV

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation. NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays

NumPy Array Attributes

a one-dimensional, two-dimensional, and three-dimensional array.

NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated

Syntax:

numpy.random.randint(low, high, size, dtype='l')

```
import numpyas np
np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # One-dimensional array
            #             low value, high value, size
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array

print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
// output
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60 #size int or tuple of ints, optional
      Output shape. If the given shape is, e.g., (m, n, k),
      then m * n * k samples are drawn.

print("dtype:", x3.dtype)        #data type of the array
```

AMSCE-1101

dtype: int64

itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

itemsize: 8 bytes
nbytes: 480 bytes *# 8 bytes * 60[m*n*k]*

Array Indexing: Accessing Single Elements

In a one-dimensional array, you can access the ith value (counting from zero) by specifying the desired index in square brackets.

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
# produced since randomly generated
In[6]: x1[0]
Out[6]: 5
In[7]: x1[4]
Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
Out[5]: array([5, 0, 3, 3, 7, 9]) # produced since randomly generated
In[8]: x1[-1]
Out[8]: 9
In[9]: x1[-2]
Out[9]: 7
```

In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[10]: x2
Out[10]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
```

AMSCE-1101

```
                   [1, 6, 7, 7]])
```

Row, column

```
In[11]: x2[0, 0]
Out[11]: 3


In[12]: x2[2, 0]
Out[12]: 1
In[13]: x2[2, -1]
Out[13]: 7
```

modify values using any of the above index notation

```
In[14]: x2[0, 0] = 12
        x2

Out[14]: array([[12, 5, 2, 4],
                [ 7, 6, 8, 8],
                [ 1, 6, 7, 7]])
```

if you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

```
array([5, 0, 3, 3, 7, 9])
In[15]: x1[0] = 3.14159 # this will be truncated!
        x1
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character.

Syntax

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1.

```
In[16]: x = np.arange(10)
        x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In[17]: x[:5] # first five elements
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

AMSCE-1101

Out[17]: array([0, 1, 2, 3, 4])

In[18]: x[5:] *# elements after index 5*
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Out[18]: array([5, 6, 7, 8, 9])

In[19]: x[4:7] *# middle subarray*
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Out[19]: array([4, 5, 6])

In[20]: x[::2] *# every other element*
Out[20]: array([0, 2, 4, 6, 8])
Prints every second element

In[21]: x[1::2] *# every other element, starting at index 1*
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Out[21]: array([1, 3, 5, 7, 9])

In[22]: x[::-1] *# all elements, reversed*
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In[23]: x[5::-2] *# reversed every other from index 5 from array [x]*
Out[23]: array([5, 3, 1])

```
x[6::-2]
array([6, 4, 2, 0])
```

AMS CE-1101

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas.
For example

In[24]: x2
Out[24]: array([[12, 5, 2, 4],
[7, 6, 8, 8],
[1, 6, 7, 7]])
In[25]: x2[:2, :3] *# two rows, three columns*

```
Out[25]: array([[12, 5, 2],
[ 7, 6, 8]])

In[26]: x2[:3, ::2] # all rows, every other column, prints every other element
Out[26]: array([[12, 2],
                [ 7, 8],
                [ 1, 7]])

In[27]: x2[::-1, ::-1]   #All rows and columns are reversed.
Out[27]: array([[ 7, 7, 6, 1],
                [ 8, 8, 6, 7],
                [ 4, 2, 5, 12]])
```

Example – STEP by STEP breakup.

```
In x2
Out array([[3, 5, 2, 4],
           [7, 6, 8, 8],
           [1, 6, 7, 7]])

In X2[::-1]
array([[1, 6, 7, 7],
       [7, 6, 8, 8],
       [3, 5, 2, 4]])
==============================================
In x2[::-1, ::-1]
array([[7, 7, 6, 1],
[8, 8, 6, 7],
[4, 2, 5, 3]])
X2
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
x2[::-2]
array([[1, 6, 7, 7],
       [3, 5, 2, 4]])

x2[::-2,::-2]
array([[7, 6],
       [4, 5]])


x2[::-3]
```

```
array([[1, 6, 7, 7]])

x2[::-3,::-3]
array([[7, 1]])
```

Accessing array rows and columns

Accessing single rows or columns of an array can be done by combining indexing and slicing, using an empty slice marked by a single colon (:)

```
array([[12, 5, 2, 4],
       [ 7, 6, 8, 8],
       [ 1, 6, 7, 7]])


In[28]: print(x2[:, 0]) # first column of x2
[12 7 1]

In[29]: print(x2[0, :]) # first row of x2
[12 5 2 4]
```

Or

```
In[30]: print(x2[0]) # equivalent to x2[0, :]
[12 5 2 4]
```

Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data.

```
In[31]: print(x2)
        [[12 5 2 4]
        [ 7 6 8 8]
        [ 1 6 7 7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
print(x2_sub)
        [[12 5]
        [ 7 6]]
```

AMSCE-1101

if we modify this subarray we'll see that the original array is changed

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)
        [[99 5]
        [ 7 6]]
In[34]: print(x2)
        [[99 5 2 4]
        [ 7 6 8 8]
        [ 1 6 7 7]]
```

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitlycopy the data within an array or a subarray.

copy()

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)

Out:    [[99 5]
        [ 7 6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
[[42 5]
[ 7 6]]

In[37]: print(x2)
        [[99 5 2 4]
        [ 7 6 8 8]
        [ 1 6 7 7]]
```

Reshaping of Arrays

AMSCE-1101

The most flexible way ofdoing this is with the reshape() method

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
print(grid)

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

the size of the initial array must match the size of the reshaped array

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. you can do this with the reshape method, or more easily by making use of the newaxis keyword within a slice operation:

```
In[39]: x = np.array([1, 2, 3])
# row vector via reshape
x.reshape((1, 3))

Out[39]: array([[1, 2, 3]])

In[40]: # row vector via newaxis
x[np.newaxis, :]# notice 2 [[  this add 1 row, 3 col as axis to array
Out[40]: array([[1, 2, 3]])

In[41]: x.reshape((3,1))
Out[41]: ([[1]
           [2]
           [3]])
```

Array Concatenation and Splitting

AMSCE-101

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines np.concatenate, np.vstack, and np.hstack.

```
In[43]: x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])# Merges two arrays
Out[43]: array([1, 2, 3, 3, 2, 1])


In[45]: grid = np.array([[1, 2, 3],
[4, 5, 6]])
In[46]: # concatenate along the first axis
np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
[4, 5, 6],
[1, 2, 3],
[4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the np.vstack(vertical stack) and np.hstack (horizontal stack) functions:

```
In[48]: x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
[6, 5, 4]])
# vertically stack the arrays
np.vstack([x, grid])
Out[48]: array([[1, 2, 3],
[9, 8, 7],
[6, 5, 4]])
In[49] y=np.array([[99],
                   [99]])
np.hstack([grid,y])
Out[49]: array([[9,8,7,99],
                   [6,5,4,99]])
```

Splitting of arrays

AMSCE-1101

The opposite of concatenation is splitting, which is implemented by the functions.
np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
# 3, 5 represents indices so elements are split at 3 and 5
print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]

In[51]: grid = np.arange(16).reshape((4, 4))
        grid
Out[51]: array([[ 0, 1, 2, 3],
               [ 4, 5, 6, 7],
               [ 8, 9, 10, 11],
               [12, 13, 14, 15]])
In[52]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)

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

        [[ 8 9 10 11]
        [12 13 14 15]]
In[53]: left, right = np.hsplit(grid, [2])
print(left)
print(right)
[[ 0 1]
[ 4 5]
[ 8 9]
[12 13]]

[[ 2 3]
[ 6 7]
[10 11]
[14 15]]
```

2. Aggregations: Min, Max, and Everything in Between

AMSCE-1101

Summing the Values in an Array

```
In[1]: import numpyas np
In[2]: L = np.random.random(100)
        sum(L)
Out[2]: 55.61209116604941
```

NumPy's version of theoperation is computed much more quickly:

```
In[4]: big_array= np.random.rand(1000000)#10 lakh
%timeitsum(big_array)#Measure execution time of small code snippets
%timeitnp.sum(big_array)

10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

the sum function and the np.sum function are not identical.

Minimum and Maximum

Similarly, Python has built-in min and max functions, used to find the minimum value and maximum value of any given array:

```
In[5]: min(big_array), max(big_array)
Out[5]: (1.1717128136634614e-06, 0.9999976784968716)
```

NumPy's corresponding functions have similar syntax, and again operate much morequickly:

```
In[6]: np.min(big_array), np.max(big_array)
Out[6]: (1.1717128136634614e-06, 0.9999976784968716)
```

Other syntax are as below:

For min, max, sum, and several other NumPy aggregates, a shorter syntax is to usemethods of the array object itself:

```
In[8]: print(big_array.min(), big_array.max(), big_array.sum())

1.17171281366e-06 0.999997678497 499911.628197
```

Multidimensional aggregates

One common type of aggregation operation is an aggregate along a row or column

AMSCE-1101

```
In[9]: M = np.random.random((3, 4))#3 rows and 4columns
print(M)

[[ 0.8967576 0.03783739 0.75952519 0.06682827]
[ 0.8354065 0.99196818 0.19544769 0.43447084]
[ 0.66859307 0.15038721 0.37911423  0.6687194 ]]
```

By default, each NumPy aggregation function will return the aggregate over the entirearray:

```
In[10]: M.sum()
Out[10]: 6.0850555667307118 # returns sum of allvalues
```

we can find the minimum value within each column by specifying axis=0

```
In[11]: M.min(axis=0) #axis 0 represents column
Out[11]: array([ 0.66859307, 0.03783739, 0.19544769, 0.06682827])
```

Similarly, we can find the maximum value within each row:

```
In[12]: M.max(axis=1)#axis 1 represents row
Out[12]: array([ 0.8967576 , 0.99196818, 0.6687194 ])
```

Other aggregation functions

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

Example: What Is the Average Height of US Presidents?

CE-1101

```
In[13]: !head -4 data/president_heights.csv
#The datas are in the formorder,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
#Pandas package is used
In[14]: import pandas as pd
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178
173174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193
182 183177 185 188 188 182 185]

In[15]: print("Mean height: ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height: ", heights.min())
print("Maximum height: ", heights.max())
Mean height: 179.738095238
Standard deviation: 6.93184344275
Minimum height: 163
Maximum height: 193

In[16]: print("25th percentile: ",np.percentile(heights, 25))
print("Median: ", np.median(heights))
print("75th percentile: ", np.percentile(heights,75))
25th percentile: 174.25 #First quartile
Median: 182.0
75th percentile: 183.0 #Third quartile
```

it's more useful to see a visual representation of this data

```
In[17]: %matplotlib inline
import matplotlib.pyplotas plt
import seaborn; seaborn.set() # set plot style
#Seaborn is a Python data visualization library based on matplotlib
In[18]: plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```

AMSCE-1101

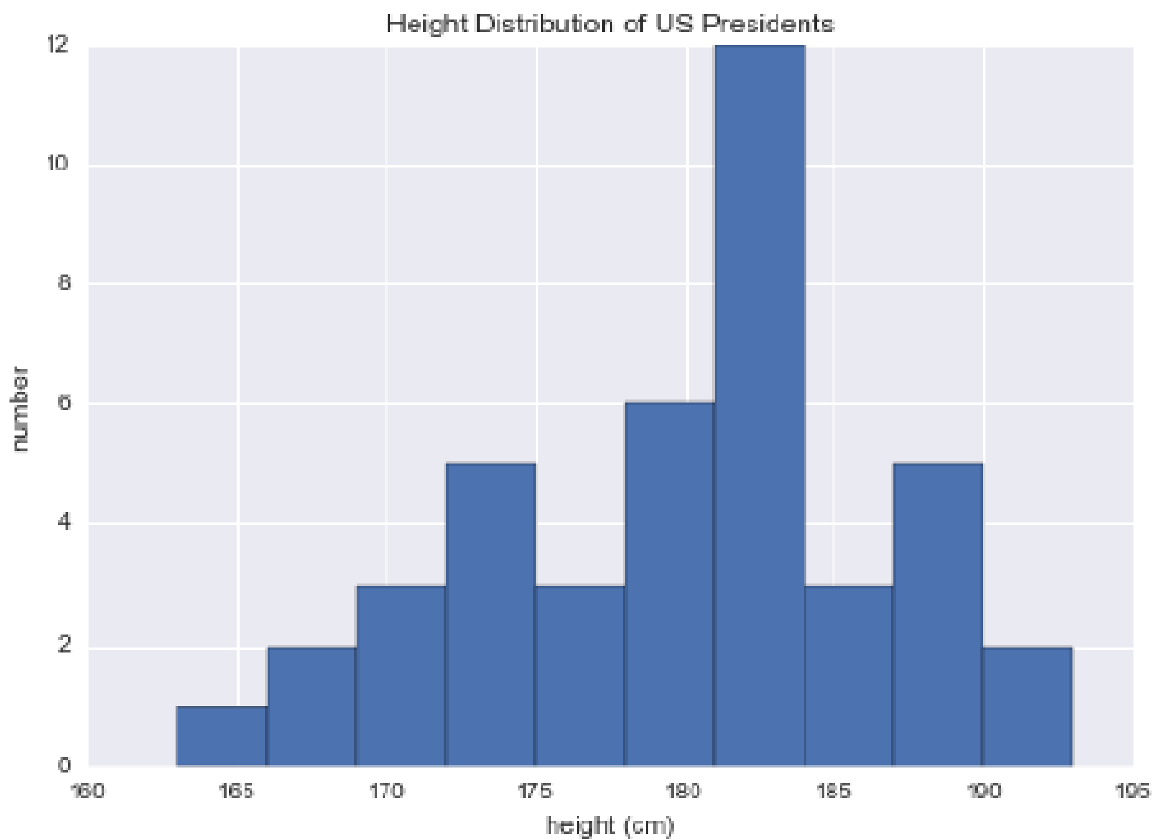


Figure 2-3. Histogram of presidential heights

AMSCL

3. Computation on Arrays: Broadcasting

Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Arrays of the same size, binary operations are performed on anelement-by-element basis

```
In[1]: import numpyas np
In[2]: a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
        a + b
Out[2]: array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes

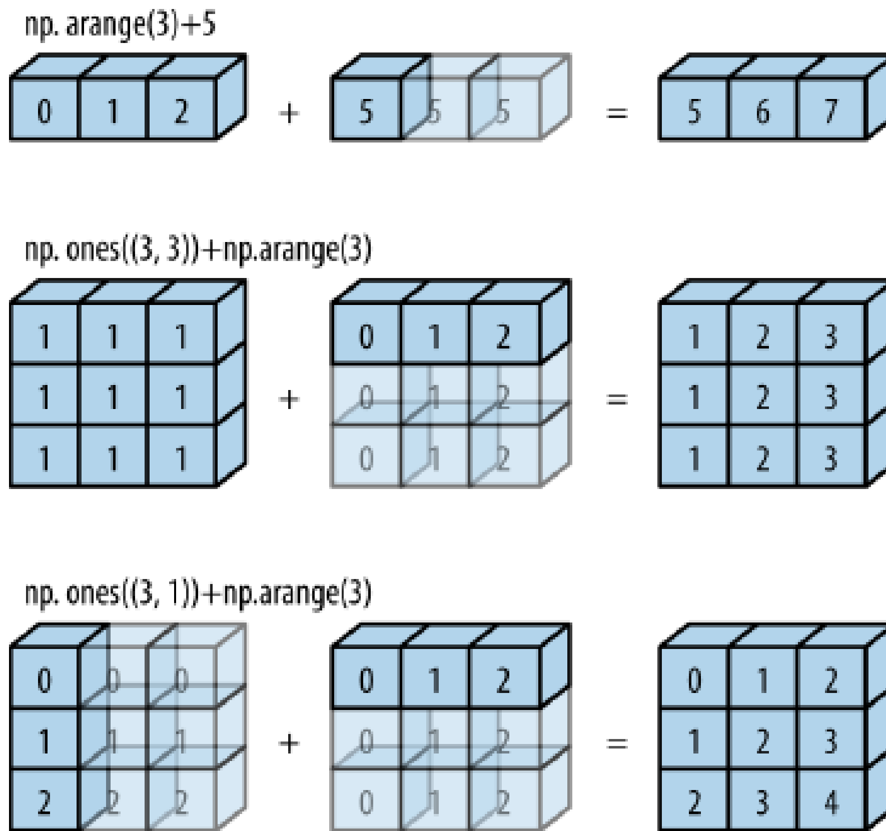


Figure 2.4

```
In[3]: a + 5
Out[3]: array([5, 6, 7])

In[4]: M = np.ones((3, 3))
                M
Out[4]: array([[ 1., 1., 1.],
[ 1., 1., 1.],
[ 1., 1., 1.]])
In[5]: M + a
Out[5]: array([[ 1., 2., 3.],
[ 1., 2., 3.],
[ 1., 2., 3.]])
```

Here the one-dimensional array a is stretched, or broadcast, across the seconddimension in order to match the shape of M.

Broadcasting of both arrays

```
In[6]: a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
```

```
print(a)
print(b)


[0 1 2]      #print a
[[0]   # print b
[1]
[2]]
In[7]: a + b
Out[7]: array([[0, 1, 2],
[1, 2, 3],
[2, 3, 4]])
```

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.

AMSCE-1101

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

```
In[8]: M = np.ones((2, 3))
a = np.arange(3)

out:  [[1. 1. 1.]
[1. 1. 1.]] #output of M
[0 1 2]      # output of a
```

Shapes of the arrays are

M.shape = (2, 3)
a.shape = (3,)

By rule 1 that the array a has fewer dimensions, so we pad it on the left with ones:

M.shape -> (2, 3)
a.shape -> (1, 3)

By rule 2, we now see that the first dimension disagrees, so we stretch this dimensionto match:

M.shape -> (2, 3)
a.shape -> (2, 3)

The shapes match, and we see that the final shape will be (2, 3):

```
In[9]: M + a
Out[9]: array([[ 1., 2., 3.],
               [ 1., 2., 3.]])
```

Broadcasting example 2

example where both arrays need to be broadcast

```
In[10]: a = np.arange(3).reshape((3, 1))
        b = np.arange(3)
```

a.shape = (3, 1)
b.shape = (3,)

```
out: [[0]
      [1]
      [2]]
     [0 1 2]
```

Rule 1 says we must pad the shape of b with ones:

a.shape -> (3, 1)
b.shape -> (1, 3)

rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array

a.shape -> (3, 3)
b.shape -> (3, 3)

the result matches, these shapes are compatibleIn[11]: a + b   # refer figure 2.4

Out[11]: array([[0, 1, 2],

AMSCE-1101

```
[1, 2, 3],
[2, 3, 4]])
```

Broadcasting example 3

an example in which the two arrays are not compatible

```
In[12]: M = np.ones((3, 2))
        a = np.arange(3)

Out     [[1. 1.]
         [1. 1.]
         [1. 1.]]  # M output

        [0 1 2] # a output
```

M.shape = (3, 2)
a.shape = (3,)
rule 1 tells us that we must pad the shape of a with ones:
M.shape -> (3, 2)
a.shape -> (1, 3)
rule 2, the first dimension of a is stretched to match that of M
M.shape -> (3, 2)# since its 2 here we cannotstrech
a.shape -> (3, 3)
rule 3—the final shapes do not match, so these two arrays are incompatible

```
In[13]: M + a
```

Error: ValueError: operands could not be broadcast together with shapes (3,2) (3,)

The Right-side padding is done explicitly by reshaping the array

A new keyword np.newaxis is used for this purpose.

```
In[14]: a[:, np.newaxis].shape
Out[14]: (3, 1)   # [0 1 2] is changed

In[15]: M + a[:, np.newaxis]
```

AMSCE-1101

```
[[1. 1.]          [[0]
 [1. 1.]    +      [1]
 [1. 1.]]          [2]]
```

```
Out[15]: array([[ 1., 1.],
[ 2., 2.],
[ 3., 3.]])
```

Broadcasting in Practice

Centring an arrayufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. example is centering an array of data

```
In[17]: X = np.random.random((10, 3))

Out
([[0.6231582 , 0.62830284, 0.48405648],
  [0.4893788 , 0.96598238, 0.99261057],
  [0.18596872, 0.26149718, 0.41570724],
  [0.74732252, 0.96122555, 0.03700708],
  [0.71465724, 0.92325637, 0.62472884],
  [0.53135009, 0.20956952, 0.78746706],
  [0.67569877, 0.45174937, 0.53474695],
  [0.91180302, 0.61523213, 0.18012776],
  [0.75023639, 0.46940932, 0.11044872],
  [0.86844985, 0.07136273, 0.00521037]])
```

```
In[18]: Xmean= X.mean(0)
        Xmean
Out[18]: array([0.64980236, 0.55575874, 0.41721111])
     # mean values of elements in first, second, third column.
```

we can center the X array by subtracting the mean value from each element in array.(Ex:)

```
In[19]: X_centered= X – Xmean
array([[-0.02664416,  0.0725441 ,  0.06684537],
  [-0.16042356,  0.41022364,  0.57539946],
  [-0.46383364, -0.29426156, -0.00150386],
  [ 0.09752016,  0.40546681, -0.38020403],
  [ 0.06485488,  0.36749763,  0.20751773],
  [-0.11845227, -0.34618922,  0.37025595],
  [ 0.02589641, -0.10400937,  0.11753584],
  [ 0.26200066,  0.05947339, -0.23708334],
```

```
[ 0.10043403, -0.08634941, -0.30676239],
[ 0.21864749, -0.484396  , -0.41200073]])
```

we can check that the centered array has near zero mean by

```
In[20]: X_centered.mean(0)
Out[20]: array([ 0.00000000e+00, -1.11022302e-16, -6.66133815e-17])
```

Plotting a two-dimensional function

Broadcasting is very useful in displaying images with 2 dimensional functions. to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid.

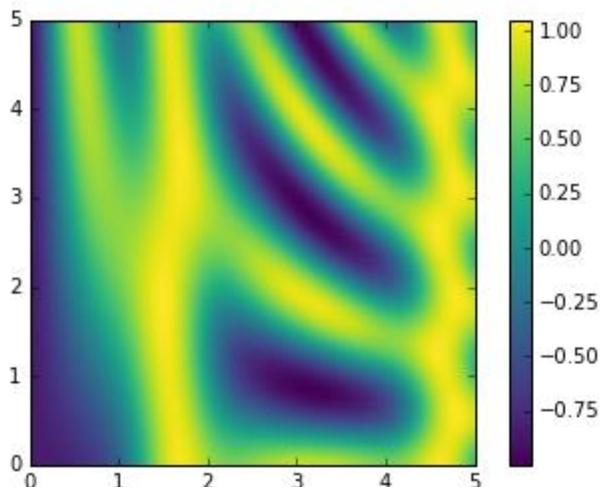
```
In[21]: # x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50) # Returns num evenly spaced samples, calculated over the interval [start, stop].
y = np.linspace(0, 5, 50)[:, np.newaxis]
z = np.sin(x)**10+np.cos(10+y*x) * np.cos(x)
```

```
Out [z]

[[-0.83907153 -0.83470697 -0.8216586  ...  0.8956708   0.68617261   0.41940746]
 [-0.83907153 -0.82902677 -0.8103873  ...  0.92522407  0.75321348   0.52508175]
 [-0.83907153 -0.82325668 -0.79876457 ...  0.96427357  0.84172689   0.66446403]
 ...
 [-0.83907153 -0.48233077 -0.01646558 ...  0.96449925  0.75196531   0.41982581]
 [-0.83907153 -0.47324558  0.00392612 ...  0.92542163  0.68540362   0.37440839]
 [-0.83907153 -0.46410908  0.02431613 ...  0.89579384  0.65690314   0.40107702]]
```

We'll use Matplotlib to plot this two-dimensional array

```
In[22]: %matplotlib inline
import matplotlib.pyplotas plt
In[23]: plt.imshow(z, origin='lower',extent=[0, 5, 0, 5],cmap='viridis')
plt.colorbar();
```

z -array,
origin - [0,0] index of z should be at the lower-left corner of the plot,
extent = left, right, bottom, and top boundaries of the image,
cmap - color map.



4.Comparisons, Masks, and Boolean Logic

The use of Boolean masks to examine and manipulate values within NumPy arrays.

Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion.

Example: Count all values greater than a certain value.

Remove all outliers that are above some threshold.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city.

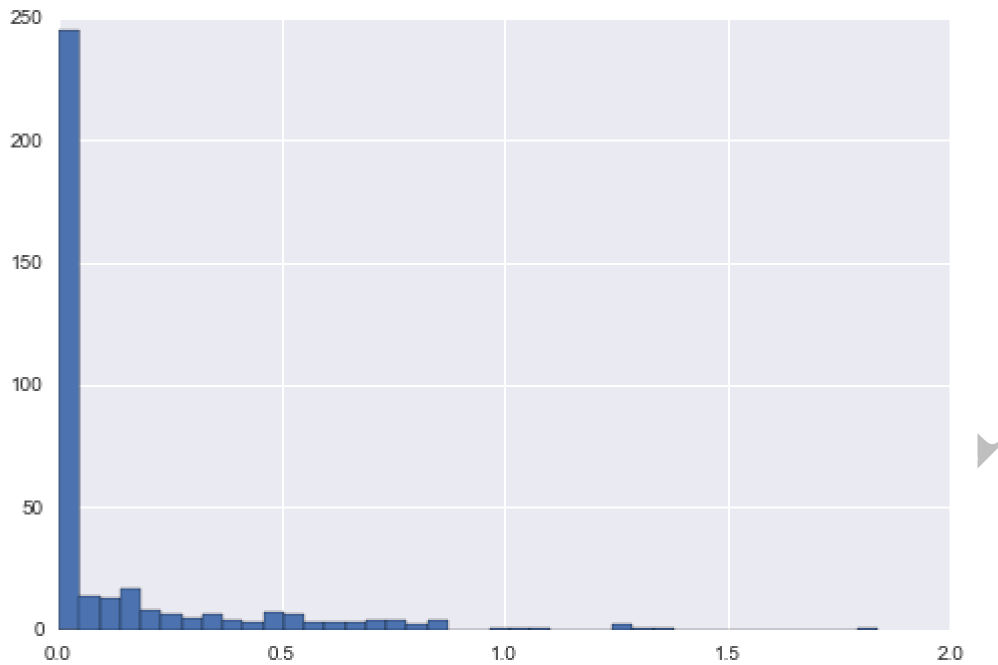
```
In[1]:
import numpyas np
import pandas as pd
# use Pandas to extract rainfall inches as a NumPy array
rainfall=pd.read_csv('data/Seattle2014.csv')['PRCP'].values
 # reads PRCP column values
inches = rainfall / 254 # 1/10mm -> inches
inches.shape
Out[1]: (365,)
```

Array contains 365 values

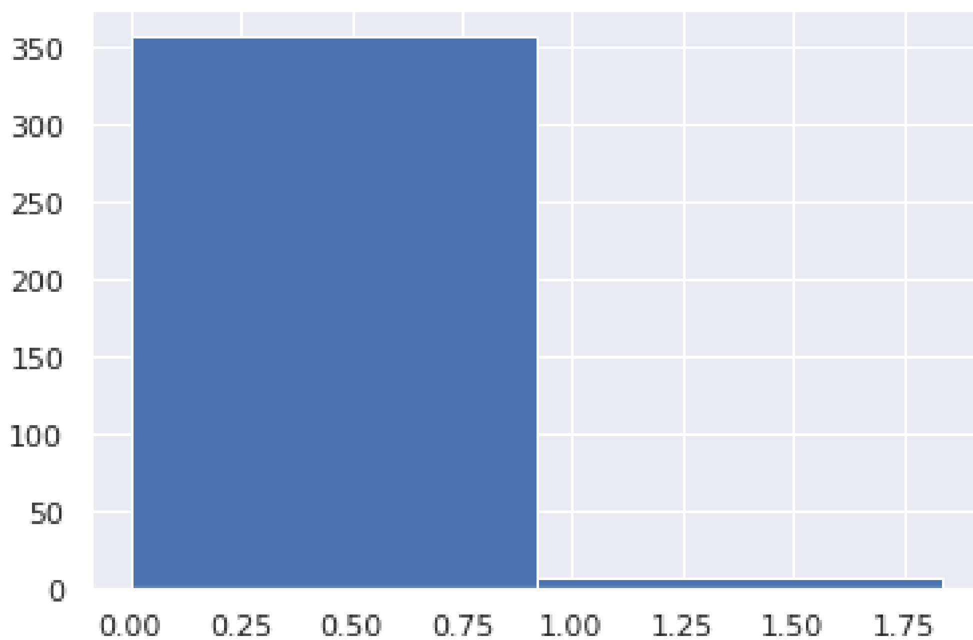
Histogram of rainy daysgenerated using Matplotlib

AMSCE-1101

```
import matplotlib.pyplotas plt
import seaborn; seaborn.set() # set plot styles
In[3]: plt.hist(inches, 40); # 40 represents number of bars
```




```
plt.hist(inches, 2);
```



Comparison Operators as ufuncs

```
In[4]: x = np.array([1, 2, 3, 4, 5])
In[5]: x <3 # less than
Out[5]: array([ True, True, False, False, False], dtype=bool)
```

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

```
In[12]: rng= np.random.RandomState(0) # pseudo-random number
generator
x = rng.randint(10, size=(3, 4)) # max number (row, column)
x
Out[12]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])
```

```
In[13]: x <6
Out[13]: array([[ True, True, True, True],
[False, False, True, True],
[True, True, False, False]], dtype=bool)
```

AMSCE-1101

Working with Boolean Arrays

```
In[14]: print(x)
[[5 0 3 3]
[7 9 3 5]
[2 4 7 6]]
```

Counting entries

```
In[15]: # how many values less than 6?
np.count_nonzero(x <6)# prints values less than 6
Out[15]: 8
```

(Or)

```
In[16]: np.sum(x <6)
Out[16]: 8
```

False is interpreted as 0, and True is interpreted as 1.

Benefit of sum(). This summation can be done along rows or columns as well.

```
In[17]: np.sum(x <6, axis=1)# how many values less than 6 in each row?
#axis 1 = row
Out[17]: array([4, 2, 2])
```

Counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any() or np.all():

```
In[18]: np.any(x >8)# are there any values greater than 8?
Out[18]: True
```

np.all() and np.any() can be used along particular axes as well. For example:

```
In[22]: # are all values in each row less than 8?
np.all(x <8, axis=1)
Out[22]: array([ True, False, True], dtype=bool)
```

Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.

BOOLEAN OPERATORS

We have already seen

- All days with rain less than four inches,
- All days with rain greater than two inches
- All days with rain less than four inches and greater than one inch?

Accomplished through Python's bitwise logic operators, &, |, ^, and ~.

AMSCE-1101

In[23]: np.sum((inches >0.5) &(inches <1))
Out[23]: 29 # days with rainfall between 0.5 and 1

Or

In[24]: np.sum(~((inches <= 0.5) | (inches >= 1)))
Out[24]: 29

<table>
<tr><th>Operator</th><th>Equivalent ufunc</th></tr>
<tr><td>&</td><td>np.bitwise_and</td></tr>
<tr><td>|</td><td>np.bitwise_or</td></tr>
<tr><td>^</td><td>np.bitwise_xor</td></tr>
<tr><td>~</td><td>np.bitwise_not</td></tr>
</table>

Boolean Arrays as Masks
In[26]: x
Out[26]: array([[5, 0, 3, 3],
[7, 9, 3, 5],
[2, 4, 7, 6]])

Boolean array for this condition

In[27]: x <5
Out[27]: array([[False, True, True, True],
[False, False, True, False],
[True, True, False, False]], dtype=bool)

Now to *select* these values from the array, we can simply index on this Boolean array;
this is known as a *masking* operation:
In[28]: x[x <5]
Out[28]: array([0, 3, 3, 3, 2, 4])

AMSCE-1101

Using the Keywords and/or Versus the Operators &/|

The difference is this: **and** and **or** gauge the truth or falsehood of *entire object*, while & and | refer to *bits within each object*. In Python, all nonzero integers will evaluate as True

```
In[30]: bool(42), bool(0)
Out[30]: (True, False)
In[31]: bool(42 and 0)
Out[31]: False
In[32]: bool(42 or 0)
Out[32]: True

In[33]: bin(42)
Out[33]: '0b101010'   #binary representation

In[34]: bin(59)
Out[34]: '0b111011'   #binary representation

In[36]: bin(42 | 59)
Out[36]: '0b111011'
```

1 = True and 0 = False

```
In[37]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
Out[37]: array([ True, True, True, False, True, True], dtype=bool)
```

Using or on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
In[38]: A or B

ValueError Traceback (most recent call last)
<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B
```

AMSCE-1101

ValueError: The truth value of an array with more than one element is...

5. Fancy Indexing

We'll look at another style of array indexing, known as *fancy indexing instead of (e.g., arr[0]), slices (e.g., arr[:5]),.*

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array.

```
In[1]:
import numpyas np
rand = np.random.RandomState(42) # 42- type of random number
generator
x = rand.randint(100, size=10)
print(x)
    [51 92 1471 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
In[2]: [x[3], x[7], x[2]]
Out[2]: [71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
    [51 92 14 7160 20 82 86 74 74]

In[3]: ind= [3, 7, 4]
x[ind]
Out[3]: array([71, 86, 60])

    [51 92 14 7160 20 82 86 74 74]

In[4]: ind= np.array([[3, 7],
```

AMSCE-1101

```
[4, 5]])
x[ind]
Out[4]: array([[71, 86],
               [60, 20]])

In[5]: X = np.arange(12).reshape((3, 4))
X
Out[5]: array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]])
```

The first index refers to the row, and the second to the column:

```
In[6]: row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
Out[6]: array([ 2, 5, 11])
```

The first value in the result is X[0, 2], the second is X[1, 1], and the third is X[2, 3]. The pairing of indices in fancy indexing follows all the broadcasting rules.

AMSCE-1101

Combined Indexing

Fancy indexing can be combined with the other indexing schemes we've seen:

```
In[9]: print(X)
      0 1 2 3
    0 [[ 0 1 2 3]
    1 [ 4 5 6 7]
    2 [ 8910 11]]
```

We can combine fancy and simple indices:

In[10]: X[2, [2, 0, 1]] #row, indices

Out[10]: array([10, 8, 9])

We can also combine fancy indexing with slicing:

```
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8910 11]]
```

In[11]: X[1:, [2, 0, 1]] #row, indices

Out[11]: array([[6, 4, 5],
[10, 8, 9]])

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

And we can combine fancy indexing with masking:

In[12]: mask = np.array([1, 0, 1, 0],dtype=bool))# masked with 1 is printed and rest are blocked

X[row[:, np.newaxis], mask]

Out[12]: array([[0, 2],
[4, 6],
[8, 10]])

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array

In[18]: x = np.arange(10)# returns evenly spaced values with in a given interval.

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
i = np.array([2, 1, 8, 4]) # represents indices
 x[i] = 99
 print(x)
```

[0 99 99 3 99 5 6 7 99 9] # respective indices are replaced by values 99

AMSCE-1101

```
x[i] -= 10 # x[i]=x[i]-10
print(x) #minuses value 10 and prints it
[ 0 89 89 3 89 5 6 7 89 9]
```

In[20]:
```
x = np.zeros(10) # print 10 zeros
x[[0, 0]] = [4, 6] # assign x[0]=4 , x[0]=6
print(x)
```

[6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

First assign x[0] = 4, followed by x[0] = 6. The result, of course, is that x[0] contains the value 6.

```
Array [ 6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

0 1 2 3 4 5 6 7 8 9

```
In[21]: i = [2, 3, 3, 4, 4, 4]
x[i] += 1  # x[2]=x[2]+1
x
```

Out[21]: array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.]) # the value is 1 since the values are overwritten.

But if you want to update then we have to use

```
In[22]: x = np.zeros(10)
i = [2, 3, 3, 4, 4, 4]
  np.add.at(x, i, 1) # array, indices, values
  print(x)
0 0 0 0 0 0 0 0 0 0
0 0 1 1
0 0 1 2
```

[0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]

AMSCE-1101

Structured Data: NumPy's Structured Arrays

```
In[2]: name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

Theres nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data.

```
In[3]: x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
In[4]: # Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
'formats':('U10', 'i4', 'f8')}) # 4 represents number of names
print(data.dtype)

OUT: [('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

'U10' translates to "Unicode string of maximum length 10,"
'i4' translates to "4-byte (i.e., 32 bit) integer," and
'f8' translates to "8-byte (i.e., 64 bit) float."
Now that we've created an empty container array, we can fill the array with our lists of values:

```
In[5]: data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
Out
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)('Doug', 19, 61.5)]
```

you can now refer to values either by index or by name:

```
In[6]: # Get all names
data['name']
Out[6]: array(['Alice', 'Bob', 'Cathy', 'Doug'],dtype='<U10')
```

AMSCE-1101

```
In[7]: # Get first row of data
data[0]
Out[7]: ('Alice', 25, 55.0)
In[8]: # Get the name from the last row
data[-1]['name']# prints last value
Out[8]: 'Doug'
In[9]: # Get names where age is under 30
data[data['age'] <30]['name']
Out[9]: array(['Alice', 'Doug'],dtype='<U10')
```

Creating Structured Arrays

Dictionary method

```
In[10]: np.dtype({'names':('name', 'age', 'weight'),
'formats':('U10', 'i4', 'f8')})
Out[10]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified with Python types or NumPy dtypes instead:

```
In[11]: np.dtype({'names':('name', 'age', 'weight'),
'formats':((np.str_, 10), int, np.float32)})
Out[11]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

```
In[12]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
Out[12]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter, then it can be written as

```
In[13]: np.dtype('S10,i4,f8')
Out[13]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

AMSCE-1101

Table 2-4. NumPy data types

Character	Description	Example
'b'	Byte	np.dtype('b')
'i'	Signed integer	np.dtype('i4') == np.int32
'u'	Unsigned integer	np.dtype('u1') == np.uint8
'f'	Floating point	np.dtype('f8') == np.int64
'c'	Complex floating point	np.dtype('c16') == np.complex128
'S', 'a'	string	np.dtype('S5')
'U'	Unicode string	np.dtype('U') == np.str_
'V'	Raw data (void)	np.dtype('V') == np.void

More Advanced Compound Types

```
In[14]: tp= np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))]) #element , dtype
X = np.zeros(1, dtype=tp)   #shape, datatype
print(X[0])
(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
print(X['mat'][0])
[[ 0. 0. 0.]
[ 0. 0. 0.]
[ 0. 0. 0.]]
```

RecordArrays: Structured Arrays with a Twist

NumPy also provides the np.recarrayclass, which is almost identical to the structured arrays.

```
In[15]: data['age']
Out[15]: array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
In[16]: data_rec= data.view(np.recarray)
data_rec.age
Out[16]: array([25, 45, 37, 19], dtype=int32)
```

AMSCE-IT

The downside is that for record arrays, there is some extra overhead involved in accessing the fields.

```
In[17]: %timeit data['age']
%timeitdata_rec['age']
%timeitdata_rec.age
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

Data Manipulation with Pandas

Data Indexing and Selection

- Accessing and modifying values in Pandas Series and DataFrameobjects

Data Selection in Series

- a Series object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary

Series as dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values:

```
In[1]: import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0], #values
index=['a', 'b', 'c', 'd'])          # keys
data

Out[1]: a 0.25
       b 0.50
       c 0.75
       d 1.00
dtype: float64
In[2]: data['b'] # returns the value of b in OUT[1]
Out[2]: 0.5

In[3]: 'a' in data # is a available in dataset
```

AMSCE-1101

Out[3]: True

In[4]: data.keys()# returns a view of all objects
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In[5]: list(data.items())# lists keys and values
Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]

The series can be extended by assigning values
In[6]: data['e'] = 1.25
Data

Out[6]: a 0.25
b 0.50
c 0.75
d 1.00
e 1.25
dtype: float64

Series as one-dimensional array
A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays—that is, slices, masking, and fancy indexing.

Out[6]: a 0.25
b 0.50
c 0.75
d 1.00
e 1.25
dtype: float64

In[7]: *# slicing by explicit index*
data['a':'c']
Out[7]: a 0.25
b 0.50
c 0.75
dtype: float64

AMSCE-1101

```
In[9]: # masking
data[(data >0.3) &(data <0.8)]
Out[9]: b 0.50
c 0.75
dtype: float64
```

```
In[10]: # fancy indexing
data[['a', 'e']]
Out[10]: a 0.25
e 1.25
dtype: float64
```

- slicing may be the source of the most confusion
- when you are slicing with an explicit index (i.e., data['a':'c']), the final index is *included* in the slice, while when you're slicing with an implicit index (i.e., data[0:2]), the final index is *excluded* from the slice.

```
data['a':'c']
Out[7]: a 0.25
b 0.50
c 0.75
```

```
data[0:2]
        a 0.25
        b 0.50
```

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion.

```
In[11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
Out[11]:
Implicit  Explicit  Values
0           1 a
1           3 b
```

ASCE-1001

2 5 c
dtype: object

In[12]: *# explicit index when indexing*
data[1]
Out[12]: 'a'

In[13]: *# implicit index when slicing*
data[1:3]
Out[13]: 3 b
5 c
dtype: object

Because of this potential confusion in the case of integer indexes, Pandas provides some special indexer attributes that explicitly expose certain indexing schemes
The loc attribute allows indexing and slicing that always references the explicit index:
Out[11]:
Implicit Explicit Values
0 1 a
1 3 b
2 5 c
dtype: object
In[14]: data.loc[1]
Out[14]: 'a'

In[15]: data.loc[1:3]
Out[15]: 1 a
3 b
dtype: object

The ilocattribute allows indexing and slicing that always references the implicit Python-style index:

AMSCE-IT01

```
Implicit  Explicit  Values
0            1 a
1            3 b
2            5 c
dtype: object

In[16]: data.iloc[1]
Out[16]: 'b'

In[17]: data.iloc[1:3]
Out[17]: 3 b
5 c
dtype: object
```

A third indexing attribute, ix, is a hybrid of the two, and for Series objects is equivalent to standard []-based indexing.

"explicit is better than implicit."

Data Selection in DataFrame

DataFrame acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of Series structures sharing the same index.

DataFrame as a dictionary

```
In[18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                          'New York': 141297, 'Florida': 170312,
                          'Illinois': 149995})#area variable
pop= pd.Series({'California': 38332521, 'Texas': 26448193,
                'New York': 19651127, 'Florida': 19552860,
                'Illinois': 12882135})#population variable
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

AMSCE-1101

```
Out[18]:            area  pop
California  423967      38332521
Florida             170312      19552860
Illinois            149995      12882135
New York            141297      19651127
Texas               695662      26448193
```

The individual Series that make up the columns of the DataFramecan be accessed via dictionary-style indexing of the column name:

```
In[19]: data['area']
Out[19]: California     423967
         Florida        170312
         Illinois       149995
         New York  141297
         Texas          695662
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
In[20]: data.area
Out[20]: California     423967
         Florida        170312
         Illinois       149995
         New York  141297
         Texas          695662
Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access

```
In[21]: data.areais data['area']
Out[21]: True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases!
Since, there is already a function called as POP(), used to remove a element in an array.

AMSCE-1101

```
In[22]: data.popis data['pop']
Out[22]: False
```

Like with the Series objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case to add a new column:

```
In[23]: data['density'] = data['pop'] / data['area']
    Data # show pop/area value
Out[23]:
              area     pop          density
California 423967     38332521     90.413926
Florida    170312     19552860     114.806121
Illinois   149995     12882135     85.883763
New York   141297     19651127     139.076746
Texas      695662     26448193     38.018740
```

DataFrame as two-dimensional array

DataFrame as an enhanced two dimensional array

```
In[24]: data.values      # from out 23 represent in float
Out[24]:
array([[ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [ 1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [ 1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
       [ 1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [ 6.95662000e+05, 2.64481930e+07, 3.80187404e+01]])
```

Matrix transpose

```
In[25]: data.T
Out[25]:
         California   Florida      Illinois     New York     Texas
area     4.239670e+05 1.703120e+05 1.499950e+05 1.412970e+05 6.956620e+05
pop      3.833252e+07 1.955286e+07 1.288214e+07 1.965113e+07 2.644819e+07
density  9.041393e+01 1.148061e+02 8.588376e+01 1.390767e+02 3.801874e+01
```

AMSCE-1101

In[26]: data.values[0] #access row 0

Out[26]: array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])

Passing a single "index" to a DataFrame accesses a column:
In[27]: data['area']
Out[27]: California 423967
Florida 170312
Illinois 149995
New York 141297
Texas 695662
Name: area, dtype: int64

Pandas again uses the loc, iloc, and ix indexers
Example dataframe
Out[23]:

area	pop	density	
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

In[28]: data.iloc[:3, :2] # row, column from out[23] it prints only 3 row and 2 columns

OUT [28]

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

In[29]: data.loc[:'Illinois', :'pop'] # cuts at illinois and pop
Out[29]: area pop

AMSCE-1101

California 423967 38332521
Florida 170312 19552860
Illinois 149995 12882135

In[30]: data.ix[:3, :'pop'] # Cuts at 3 row and pop
Out[30]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

In[31]: data.loc[data.density> 100, ['pop', 'density']]
prints values more than 100 in both pop and density.
Out[31]:

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

In[32]: data.iloc[0, 2] = 90 # assignment of value 90 at row 0 and column 2
data # prints the data
Out[32]:

	area	pop	density
California	423967	38332521	90.0000000 #value changed
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

AMSCE-1101

Additional indexing conventions

Example dataframe
Out[23]:

area	pop	density	
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121

```
Illinois    149995     12882135     85.883763
New York    141297     19651127     139.076746
Texas       695662     26448193     38.018740
```

```
In[33]: data['Florida':'Illinois']          # from out 23
Out[33]: area pop density
Florida 170312   19552860  114.806121
Illinois 149995   12882135  85.883763
```

```
In[34]: data[1:3]
Out[34]: area pop density
Florida 170312   19552860  114.806121
Illinois 149995   12882135  85.883763
```

```
In[35]: data[data.density>100]     # prints density value >100
Out[35]: area pop density
Florida      170312      19552860  114.806121
New York 141297      19651127  139.076746
```

Operating on Data in Pandas

Ufuncs: Index Preservation
Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects

Let's start by defining a simple Series andDataFrame

```
In[1]: import pandas as pd
import numpyas np
In[2]: rng= np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
#random.randint(low, high=None, size=None, dtype=int)
ser
Out[2]: 0    6  #6 3 7 4 are produced from pd.series
        1    3
          2  7
```

AMSCE 1101

```
        3       4
dtype: int64
In[3]: df= pd.DataFrame(rng.randint(0, 10, (3, 4)), # 0 to 10 values 3
rows and 4 columns with names A B C D
columns=['A', 'B', 'C', 'D'])
df

Out[3]: A B C D
        0 6 9 2 6
        1 7 4 3 7
        2 7 2 5 4
```

If we apply a NumPy ufunc (equivalent operators)on either of these objects, the result will be another Pandas object *with the indices preserved*

```
In[4]: np.exp(ser)       #exponential of ser
Out[4]: 0   403.428793       # exponential value of 6 and so on
        1   20.085537
        2   1096.633158
        3   54.598150
        dtype: float64
```

UFuncs: Index Alignment

Suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
In[6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
'California': 423967}, name='area')
population = pd.Series({'California': 38332521,'Texas': 26448193,
'New York': 19651127}, name='population')
In[7]: population / area
Out[7]: Alaska     NaN # Not a Number
        California  90.413926
        New York   NaN
        Texas      38.018740
        dtype: float64
```

AMSCE-1101

The resulting array contains the *union* of indices of the two input arrays which we could determine using standard Python set arithmetic on these indices:

```
In[8]: area.index| population.index
Out[8]: Index(['Alaska', 'California', 'New York',
'Texas'],dtype='object')
```

Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number," which is how Pandas marks missing data

Example:

```
In[9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])#values with index are added
       B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
Out[9]:     0 NaN
            1 5.0
            2 9.0
            3 NaN
dtype: float64
```

If using NaN values is not the desired behaviour, we can modify by calling A.add(B) is equivalent to calling A + B,

```
In[10]: A.add(B, fill_value=0)

Out[10]:    0      2.0   #  2 + Nan =2
            1      5.0   #  4+1 = 5
            2      9.0   #  6+3=9
            3      5.0  #  NaN+5 = 5
dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on DataFrames

```
In[11]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),      #0 to 20 val
columns=list('AB'))
A
Out[11]:        A  B
            0  111
```

AMSCE-1101

1 5 1

In[12]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B

Out[12]: B A C
0 4 0 9
1 5 8 0
2 9 2 6

In[13]: A + B
Out[13]: A B C
0 1.0 15.0 NaN
1 13.0 6.0 NaN
2 NaNNaNNaN

Here we'll fill with the mean of all values in A (which we compute by first stacking the rows of A):
In[14]: fill = A.stack().mean() # all values in A are stacked and added to find mean = 4.5 obtained from (1+5+11+1)/4

A.add(B, fill_value=fill)
A B C B A C
0 1114.5 + 0 4 09
1 5 14.5 1 5 8 0
2 4.5 4.5 4.5 2 9 2 6

A is added with B and mean value is added to missing values
Nan Values are filled with OUT[12] + 4.5
the values from out[11] are kept and 4.5 is added to
Out[14]: A B C
0 1.0 15.0 13.5
1 13.0 6.0 4.5
2 6.5 13.5 10.5

AMSCE-1101

Table 3-1. Mapping between Python operators and Pandas methods

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Ufuncs: Operations Between DataFrame and Series

When you are performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained. The operation is similar to operations between a two-dimensional and one-dimensionalNumPy array.

```
In[15]: A = rng.randint(10, size=(3, 4))
        A
Out[15]: array([[3, 8, 2, 4],
                [2, 6, 4, 8],
                [6, 1, 3, 8]])
In[16]: A - A[0] # Value of array is subtracted from row 0.

Out[16]: array([[ 0, 0, 0, 0],
                [-1, -2, 2, 4],
                [ 3, -7, 1, 4]])
```

In Pandas, the convention similarly operates row-wise bydefault:

```
In[17]: df= pd.DataFrame(A, columns=list('QRST'))
df- df.iloc[0] # subtract df from row 0 of df
Out[17]:        Q  R  S  T
           0    0  0  0  0
           1   -1 -2  2  4
           2    3 -7  1  4
```

AMSCE-T01

Handling Missing Data

Difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python.

we'll refer to missing data in general as null, NaN, or NA values.

Trade-Offs in Missing Data Conventions

A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value that indicatesa missing entry.

Missing Data in Pandas

1. **None: Pythonic missing data**

```
In[1]: import numpyas np
import pandas as pd
In[2]: vals1 = np.array([1, None, 3, 4])
vals1
Out[2]: array([1, None, 3, 4], dtype=object) # dtype is object due to None.

In[3]: for dtypein ['object', 'int']:
    print("dtype =", dtype)

Out: dtype = object

dtype = int

    %timeitnp.arange(1E6, dtype=dtype).sum()
    print()
```

AMSCE101

dtype = object
10 loops, best of 3: 78.2 ms per loop
dtype = int
100 loops, best of 3: 3.06 ms per loop

dtype=object means python objects. This is done at python level and has more overhead.

The use of Python objects in an array also means that if you perform aggregations like sum() or min() across an array with a None value, you will generally get an error:

```
In[4]: vals1.sum()
TypeError Traceback (most recent call last)
<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()
```

NaN: Missing numerical data

The other missing data representation, NaN (acronym for *Not a Number*), is different; it is a special floating-point valuerecognized by all systems that use the standard IEEE floating-point representation

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
Out[5]: dtype('float64')
```

the result of arithmetic with NaNwill be another NaN:

```
In[6]: 1 + np.nan
Out[6]: nan
In[7]: 0 * np.nan
Out[7]: nan
In[8]: vals2.sum(), vals2.min(), vals2.max()
Out[8]: (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values

```
In[9]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[9]: (8.0, 1.0, 4.0)
```

AMSCE-1101

NaN and None in Pandas

```
In[10]: pd.Series([1, np.nan, 2, None])
Out[10]:
0 1.0
    1 NaN
    2 2.0
    3 NaN
    dtype: float64
```

Pandas automatically type-castswhen NA values are present. For example, if we set a value in an integer array tonp.nan, it will automatically be upcast to a floating-point type to accommodate the NA:

```
In[11]: x = pd.Series(range(2), dtype=int)
    x
Out[11]: 0 0
    1 1
    dtype: int64
In[12]: x[0] = None
    x
Out[12]: 0 NaN
    1 1.0
    dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the None to a NaN value.

Table 3-2. Pandas handling of NAs by type

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

Operating on Null Values

Pandas treats None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures.

isnull()Generate a Boolean mask indicating missing values

MSCE-1101

notnull()Opposite of isnull()
dropna()Return a filtered version of the data
fillna()Return a copy of the data with missing values filled or imputed

Detecting null values

Pandas data structures have two useful methods for detecting null data: isnull() and notnull(). Either one will return a Boolean mask over the data. For example:

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])
In[14]: data.isnull()
Out[14]:    0 False
            1 True
            2 False
            3 True
            dtype: bool
In[15]: data[data.notnull()]  #Displays elements that are not null
Out[15]: 0 1
         2 hello
      dtype: object
```

Dropping null values

There are the convenience methods, dropna() (which removes NA values) and fillna() (which fills in NA values).

```
In[16]: data.dropna()
Out[16]: 0 1
          2 hello
dtype: object

In[17]: df= pd.DataFrame([[1, np.nan, 2],
[2, 3, 5],
[np.nan, 4, 6]])
df

Out[17]:        0    1     2
            0 1.0   NaN2
            1 2.0   3.0   5
            2 NaN 4.0   6
```

AMSCE-1101

By default, dropna() will drop all rows in which *any* null value is present:

```
In[18]: df.dropna()
Out[18]:      0    1    2
        1 2.0  3.0  5       #dispalys row with no missing values
```

you can drop NA values along a different axis; axis=1 drops all columns containing a null value:

```
In[19]: df.dropna(axis='columns')
Out[19]:     2    # Displays only column with no missing values.
        0 2
        1 5
        2 6
In[20]: df[3] = np.nan # add column 3 to df.
        df
Out[20]:     0      1     2     3
        0 1.0   NaN 2 NaN
        1 2.0   3.0   5 NaN
        2 NaN 4.0   6 NaN
In[21]: df.dropna(axis='columns', how='all')
Out[21]:      0     1    2      # drops all NaN column since (axis=col)
        0 1.0 NaN 2
        1 2.0 3.0   5
        2 NaN 4.0 6
```

the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)
Out[22]:      0   1    2  3
        1 2.0 3.0 5 NaN
```

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value.

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data

Out[23]:
```

AMSCE-1101

a 1.0
b NaN
c 2.0
d NaN
e 3.0
dtype: float64

We can fill NA entries with a single value, such as zero:
In[24]: data.fillna(0)

Out[24]:
a 1.0 # filled with 0 values
b 0.0
c 2.0
d 0.0
e 3.0
dtype: float64

We can specify a forward-fill to propagate the previous value forward:
In[25]: # forward-fill
data.fillna(method='ffill')

Out[25]:
a 1.0 # fills previous value 1 in NaN value
b 1.0
c 2.0
d 2.0
e 3.0
dtype: float64

we can specify a back-fill to propagate the next values backward
In[26]: # back-fill
data.fillna(method='bfill')

Out[26]:
a 1.0 # fills below value 2 in NaN value
b 2.0

AMSCE-1101

c 2.0
d 3.0
e 3.0
dtype: float64

For DataFrames, the options are similar, but we can also specify an axis along which the fills take place:
In[27]: df

```
Out[27]:
0      1    2 3
0 1.0  NaN  2 NaN
1 2.0  3.0  5 NaN
2 NaN  4.0  6 NaN
```

In[28]: df.fillna(method='ffill', axis=1) #column wise fill from prev

```
Out[28]:
0   1    2    3
0 1.0  1.0  2.0  2.0
1 2.0  3.0  5.0  5.0
2 NaN  4.0  6.0  6.0
```

Notice that if a previous value is not available during a forward fill, the NA value remains.

AMSCE-1101

Hierarchical Indexing

While Pandas does provide Panel and Panel4D objects that natively handle three-dimensional and four-dimensional data, a far more common pattern in practice is to make use of hierarchical indexing (also known as multi-indexing) to incorporate multiple index levels within a single index.

creation of MultiIndex objects

```
In[1]: import pandas as pd
import numpyas np

In[2]: index = [('California', 2000), ('California', 2010),
                ('New York', 2000), ('New York', 2010),
                ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
```

```
                     18976457, 19378102,
                     20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
Out[2]:      (California, 2000) 33871648
             (California, 2010) 37253956
             (New York, 2000) 18976457
             (New York, 2010) 19378102
             (Texas, 2000) 20851820
             (Texas, 2010) 25145561
dtype: int64
```

you can straightforwardly index or slice the series based on this multiple index:

```
In[3]: pop[('California', 2010):('Texas', 2000)] # remove item at a
                                                   given index
Out[3]:      (California, 2010) 37253956
             (New York, 2000) 18976457
             (New York, 2010) 19378102
             (Texas, 2000) 20851820
             dtype: int64
```

if you need to select all values from 2010, you'll need to do some messy.

```
In[4]: pop[[ifor iin pop.indexif i[1] == 2010]]
# pop.index returns index of dataframe.
Out[4]: (California, 2010) 37253956
             (New York, 2010) 19378102
             (Texas, 2010) 25145561
             dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets), So we go for multiindex

The better way: Pandas MultiIndex

We can create a multi-index from the tuples as follows

```
In[5]: index = pd.MultiIndex.from_tuples(index)
       Index
```

AMSCE-1101

Out[5]: MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])

- MultiIndex contains multiple levels of indexing
- In this case, the state names and the years, as well as multiple labels for each data point which encode these levels.
- reindex of series with MultiIndex shows the hierarchical representation of data

```
In[6]: pop = pop.reindex(index)
       pop
Out[6]: 0 California    0 2000      33871648
          0             1 2010      37253956
        1 New York      0 2000      18976457
          1         1 2010      19378102
        2 Texas         0 2000      20851820
          2   1 2010      25145561
              dtype: int64
```

Blank entry indicates the same value as the line above it.

```
In[7]: pop[:, 2010]      #access data of 2010
Out[7]: California 37253956
          New York 19378102
          Texas 25145561
dtype: int64
```

MultiIndex as extra dimension

we could easily have stored the same data using a simple DataFrame with index and column labels. The unstack() method will quickly convert a multiplyindexed Series into a conventionally indexed DataFrame

```
In[8]: pop_df= pop.unstack()
       pop_df
```

AMSCE-1101

Out[8]:

	2000	2010# Difference
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

stack() method provides the opposite operation:

```
In[9]: pop_df.stack()
Out[9]:
        California  2000        33871648
                    2010        37253956
        New York    2000        18976457
                    2010        19378102
        Texas       2000        20851820
                    2010        25145561
dtype: int64
```

so why do we need multiple indexing?

we were able to use multi-indexing to represent two-dimensional data within a one-dimensional Series, we can also use it to represent data of three or more dimensions in a Series or DataFrame.

Now we add another column with population under 18.

```
In[10]:pop_df= pd.DataFrame({'total': pop,
                             'under18': [9267089, 9284094,
                             4687374, 4318033,
                             5906301, 6879014]})
        pop_df

Out[10]:                  total     under18
        California 2000 33871648  9267089
                   2010 37253956  9284094
        New York 2000  18976457  4687374
                   2010 19378102  4318033
        Texas      2000 20851820  5906301
                   2010 25145561  6879014
```

AMSCE-1101

The fraction of people under 18 year calculate by

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
        f_u18.unstack()
Out[11]:    2000        2010
California  0.273594    0.249211
New York    0.247010    0.222831
Texas       0.283251    0.273568
```

Methods of MultiIndex Creation

you pass a dictionary with appropriate tuples as keys, Pandas will automaticallyrecognize this and use a MultiIndex by default:

```
In[13]: data = {('California', 2000): 33871648,     # (tuple) - key
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)

Out[13]:    California  2000    33871648
                        2010    37253956
            New York    2000    18976457
                        2010    19378102
            Texas       2000    20851820
                        2010    25145561
dtype: int64
```

Explicit MultiIndex constructors

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

MultiIndex level names

AMSCE-IOT

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

MultiIndex for columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well

```
In[19]:
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data

Out[19]: subject      Bob        Guido        Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

Indexing and Slicing a MultiIndex

Multiply indexed Series

In[21]: pop

Out[21]: state   year

```
California  2000      33871648
            2010      37253956
New York    2000      18976457
            2010      19378102
Texas       2000      20851820
            2010      25145561
dtype: int64
In[22]: pop['California', 2000]     # prints California yr 2000 popul.
Out[22]: 33871648
In[23]: pop['California']
Out[23]: year                       # prints available data on california
         2000 33871648
         2010 37253956
dtype: int64
```

Combining Datasets: Concat and Append

```
In[1]: import pandas as pd
import numpyas np
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for iin ind]
    for c in cols}
    return pd.DataFrame(data, ind)
    # example DataFrame
    make_df('ABC', range(3))
Out[2]: A B C
    0 A0 B0 C0
    1 A1 B1 C1
    2 A2 B2 C2
```

```
In[4]: x = [1, 2, 3]
       y = [4, 5, 6]
       z = [7, 8, 9]
       np.concatenate([x, y, z])

Out[4]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

AMSCE-1101

```
In[5]: x = [[1, 2],
            [3, 4]]
       np.concatenate([x, x], axis=1)

Out[5]: array([[1, 2, 1, 2],
               [3, 4, 3, 4]])
```

Simple Concatenation with pd.concat

pd.concat(), which has a similar syntax to np.concatenate.

```
pd.concat(objs, axis=0, join='outer', join_axes=None,
ignore_index=False,keys=None, levels=None, names=None,
verify_integrity=False,copy=True)
```

pd.concat() can be used for a simple concatenation of Series or DataFrameobjects,just as np.concatenate() can be used for simple concatenations of arrays:

```
In[6]:
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
Out[6]: 1 A
        2 B
        3 C
        4 D
        5 E
        6 F
dtype: object
```

```
In[7]: df1 = make_df('AB', [1, 2])
       df2 = make_df('AB', [3, 4])
       print(df1); print(df2); print(pd.concat([df1, df2]))

df1              df2              pd.concat([df1, df2])
     A    B           A    B           A    B
 1   A1   B1      3   A3   B3      1   A1   B1
 2   A2   B2      4   A4   B4      2   A2   B2
                                   3   A3   B3
                                   4   A4   B4
```

AMSCE-1101

By default, the concatenation takes place row-wise within the DataFrame (i.e., axis=0). Like np.concatenate, pd.concat allows specification of an axis along whichconcatenation will take place.
We could have equivalently specified axis=1; here we've used the more intuitive axis='col'.

```
In[8]: df3 = make_df('AB', [0, 1])
       df4 = make_df('CD', [0, 1])
       print(df3); print(df4); print(pd.concat([df3, df4], axis='col'))
```

```
df3            df4            pd.concat([df3, df4], axis='col')
    A   B          C   D          A   B   C   D
 0  A0  B0      0  C0  D0      0  A0  B0  C0  D0
 1  A1  B1      1  C1  D1      1  A1  B1  C1  D1
```

Duplicate indices

Difference between np.concatenate and pd.concat is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices

```
In[9]: x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index= x.index# make duplicate indices!
print(x); print(y); print(pd.concat([x, y]))
```

```
x              y              pd.concat([x, y])
    A   B          A   B          A   B
 0  A0  B0      0  A2  B2      0  A0  B0
 1  A1  B1      1  A3  B3      1  A1  B1
                               0  A2  B2
                               1  A3  B3
```

While this is valid within DataFrames, the outcome is often undesirable. pd.concat() gives us a few ways to handle it.

Catching the repeats as an error. If you'd like to simply verify that the indices in theresult of pd.concat() do not overlap, you can specify the verify_integrity flag.
With this set to True, the concatenation will raise an exception if there are duplicateindices.
The following code is a enclosed in "try except block"

```
In[10]: try:
pd.concat([x, y], verify_integrity=True)
except ValueErroras e:
```

```
print("ValueError:", e)
ValueError: Indexes have overlapping values: [0, 1]
```

Ignoring the index. Sometimes the index itself does not matter, and you would preferit to simply be ignored. You can specify this option using the ignore_indexflag. Withthis set to True, the concatenation will create a new integer index for the resultingSeries:

```
In[11]: print(x); print(y); print(pd.concat([x, y], ignore_index=True))

 x              y             pd.concat([x, y], ignore_index=True)
      A   B          A   B         A   B
  0  A0  B0      0  A2  B2     0  A0  B0
  1  A1  B1      1  A3  B3     1  A1  B1
                               2  A2  B2
                               3  A3  B3
```

Adding MultiIndex keys. Another alternative is to use the keys option to specify a labelfor the data sources; the result will be a hierarchically indexed series containing thedata:

```
In[12]: print(x); print(y); print(pd.concat([x, y], keys=['x', 'y']))

 x                y                pd.concat([x, y], keys=['x', 'y'])
      A   B            A   B                A   B
  0  A0  B0       0  A2  B2        x  0  A0  B0
  1  A1  B1       1  A3  B3           1  A1  B1
                                   y  0  A2  B2
                                      1  A3  B3
```

Concatenation with joins

Concatenating different column datasets.

In the simple examples we just looked at, we were mainly concatenating DataFrameswith shared column names. In practice, data from different sources might have different sets of column names, and pd.concat offers several options in this case. Considerthe concatenation of the following two DataFrames, which have some (but not all!) columns in common:

```
In[13]: df5 = make_df('ABC', [1, 2])
        df6 = make_df('BCD', [3, 4])
        print(df5); print(df6); print(pd.concat([df5, df6])

df5              df6              pd.concat([df5, df6])
    A   B   C        B   C   D         A   B   C    D
 1  A1  B1  C1    3  B3  C3  D3    1   A1  B1  C1  NaN
 2  A2  B2  C2    4  B4  C4  D4    2   A2  B2  C2  NaN
                                   3  NaN  B3  C3   D3
                                   4  NaN  B4  C4   D4
```

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the join and join_axesparametersof the concatenate function. By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns usingjoin='inner':

```
In[14]: print(df5); print(df6);
        print(pd.concat([df5, df6], join='inner'))

df5              df6              pd.concat([df5, df6], join='inner')
    A   B   C        B   C   D         B   C
 1  A1  B1  C1    3  B3  C3  D3    1   B1  C1
 2  A2  B2  C2    4  B4  C4  D4    2   B2  C2
                                   3   B3  C3
                                   4   B4  C4
```

#Common Columns are added

Another option is to directly specify the index of the remaining colums using thejoin_axes argument, which takes a list of index objects. Here we'll specify that thereturned columns should be the same as those of the first input:

```
df5              df6              pd.concat([df5, df6], join_axes=[df5.columns])
    A   B   C        B   C   D         A   B   C
 1  A1  B1  C1    3  B3  C3  D3    1   A1  B1  C1
 2  A2  B2  C2    4  B4  C4  D4    2   A2  B2  C2

 3  NaN  B3  C3
 4  NaN  B4  C4
```

Only columns common to df5 are added

The append() method

rather than calling pd.concat([df1, df2]), you can simply calldf1.append(df2):

```
In[16]: print(df1); print(df2); print(df1.append(df2))

df1              df2              df1.append(df2)
     A    B           A    B           A    B
 1   A1   B1      3   A3   B3      1   A1   B1
 2   A2   B2      4   A4   B4      2   A2   B2
                                   3   A3   B3
                                   4   A4   B4
```

unlike the append() and extend() methods of Python lists, the append() method in Pandas does not modify the original object—instead, it creates a new object with the combined data.

Combining Datasets: Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join andmerge operations. The main interface for this is the pd.merge function.

Relational Algebra

The behaviour implemented in pd.merge() is a subset of what is known as *relationalalgebra*, which is a formal set of rules for manipulating relational data, and forms theconceptual foundation of operations available in most databases

Categories of Joins

The pd.merge() function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins.

One-to-one joins

```
In[2]:
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1); print(df2)

df1                              df2
  employee        group            employee  hire_date
0      Bob   Accounting        0       Lisa       2004
1     Jake  Engineering        1        Bob       2008
2     Lisa  Engineering        2       Jake       2012
3      Sue           HR        3        Sue       2014
```

```
In[3]: df3 = pd.merge(df1, df2)
       df3

Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

CSE-101

The employee column is common hence the datas are merged.

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicateentries.

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                           'supervisor': ['Carly', 'Guido', 'Steve']})
       print(df3); print(df4); print(pd.merge(df3, df4))

df3                                      df4
  employee        group  hire_date              group supervisor
0      Bob   Accounting       2008       0   Accounting      Carly
1     Jake  Engineering       2012       1  Engineering      Guido
2     Lisa  Engineering       2004       2           HR      Steve
3      Sue           HR       2014

pd.merge(df3, df4)
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

Many-to-many joins

If the key column in both the left and right array contains duplicates, thenthe result is a many-to-many merge

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                     'Engineering', 'Engineering', 'HR', 'HR'],

                           'skills': ['math', 'spreadsheets', 'coding', 'linux',
                                      'spreadsheets', 'organization']})
print(df1); print(df5); print(pd.merge(df1, df5))

df1                            df5
  employee        group              group        skills
0      Bob   Accounting       0   Accounting          math
1     Jake  Engineering       1   Accounting  spreadsheets
2     Lisa  Engineering       2  Engineering        coding
3      Sue           HR       3  Engineering         linux
                              4           HR  spreadsheets
                              5           HR  organization

pd.merge(df1, df5)
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

Specification of the Merge Key

The on keywordexplicitly specify the name of the key column using the on keyword for joining

```
In[6]: print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))

df1                              df2
  employee        group              employee  hire_date
0      Bob   Accounting            0     Lisa       2004
1     Jake  Engineering            1      Bob       2008
2     Lisa  Engineering            2     Jake       2012
3      Sue           HR            3      Sue       2014


 pd.merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

The left_on and right_on keywords

At times you may wish to merge two datasets with different column names. we may have a dataset in which the employee name is labeled as "name" ratherthan "employee".

1101

```
In[7]:
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
print(df1); print(df3);
print(pd.merge(df1, df3, left_on="employee", right_on="name"))

df1                              df3
  employee        group              name  salary
0      Bob   Accounting          0    Bob   70000
1     Jake  Engineering          1   Jake   80000
2     Lisa  Engineering          2   Lisa  120000
3      Sue           HR          3    Sue   90000


pd.merge(df1, df3, left_on="employee", right_on="name")
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

You can drop a redundant column by drop()

```
In[8]:
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)

Out[8]:   employee        group  salary
        0      Bob   Accounting   70000
        1     Jake  Engineering   80000
        2     Lisa  Engineering  120000
        3      Sue           HR   90000
```

The left_index and right_index keywords

Sometimes, rather than merging on a column, you would instead like to merge on anindex.

```
In[9]: df1a = df1.set_index('employee')
       df2a = df2.set_index('employee')
       print(df1a); print(df2a)

df1a                         df2a
                group                   hire_date
employee                      employee
Bob        Accounting         Lisa          2004
Jake      Engineering         Bob           2008
Lisa      Engineering         Jake          2012
Sue                HR         Sue           2014
```

```
In[10]:
print(df1a); print(df2a);
print(pd.merge(df1a, df2a, left_index=True, right_index=True))

df1a                         df2a
                group                   hire_date
employee                      employee
Bob        Accounting         Lisa          2004
Jake      Engineering         Bob           2008
Lisa      Engineering         Jake          2012
Sue                HR         Sue           2014

pd.merge(df1a, df2a, left_index=True, right_index=True)
                group  hire_date
employee
Lisa      Engineering       2004
Bob        Accounting       2008
Jake      Engineering       2012
Sue                HR       2014
```

 # Index with common names are merged and associated together.

DataFrames implement the join() method, which performs a merge that defaults to joining on indices:

```
In[11]: print(df1a); print(df2a); print(df1a.join(df2a))

df1a.join(df2a)
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

Specifying Set Arithmetic for Joins

Mary is only merged since its in both df6 and 7

```
In[13]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                            'food': ['fish', 'beans', 'bread']},
                           columns=['name', 'food'])
        df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                            'drink': ['wine', 'beer']},
                           columns=['name', 'drink'])
        print(df6); print(df7); print(pd.merge(df6, df7))
```

```
df6                  df7                pd.merge(df6, df7)
    name   food           name drink        name   food  drink
0  Peter   fish      0    Mary  wine    0   Mary  bread   wine
1   Paul  beans      1  Joseph  beer
2   Mary  bread
```

Overlapping Column Names: The suffixes Keyword

you may end up in a case where your two input DataFrames have conflictingcolumn names.

```
In[17]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [1, 2, 3, 4]})

        df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [3, 1, 4, 2]})
        print(df8); print(df9); print(pd.merge(df8, df9, on="name"))
```

```
df8                df9                pd.merge(df8, df9, on="name")
   name  rank         name  rank         name  rank_x  rank_y
0   Bob     1      0   Bob     3      0   Bob       1       3
1  Jake     2      1  Jake     1      1  Jake       2       1
2  Lisa     3      2  Lisa     4      2  Lisa       3       4
3   Sue     4      3   Sue     2      3   Sue       4       2
```

Aggregation and Grouping

sum(), mean(), median(), min(), and max()

Planets.head() # show from index 0 to 4

Planets Data

ANU101

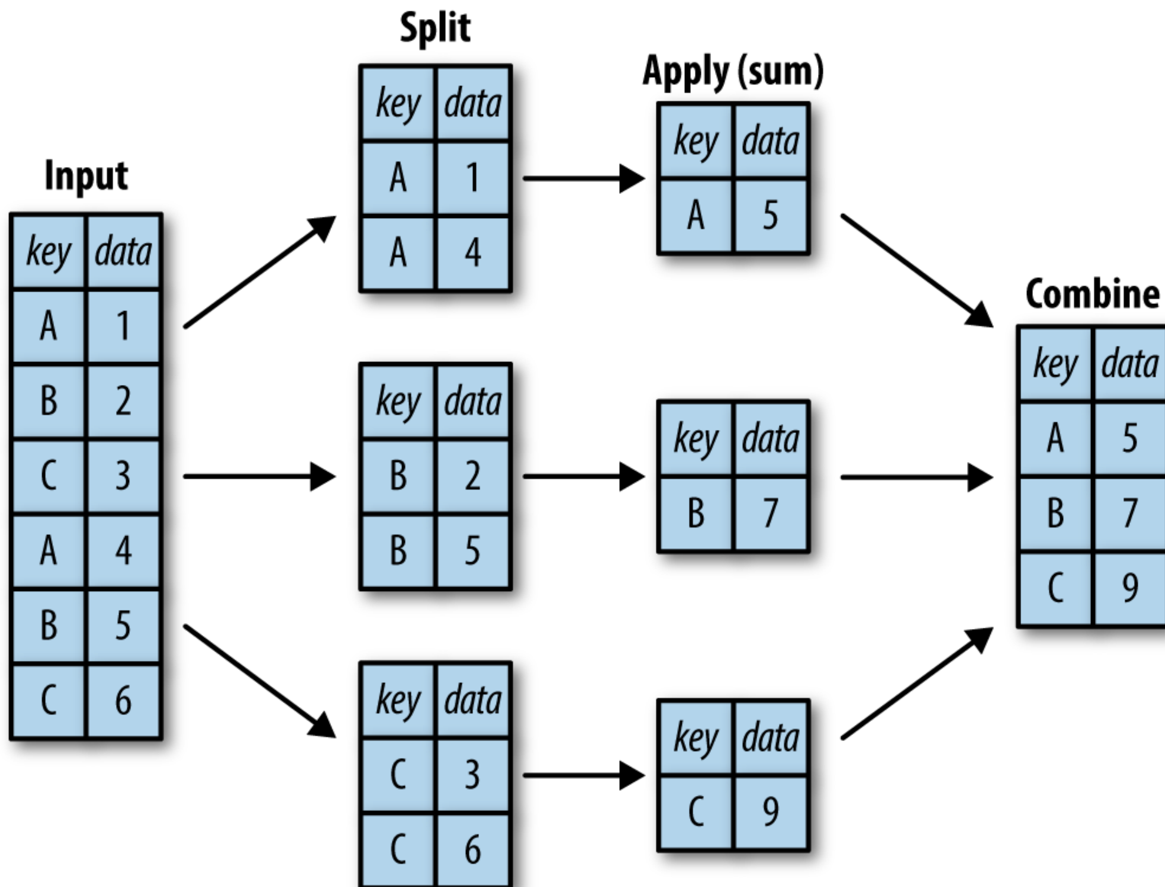
```
In[2]: import seaborn as sns
       planets = sns.load_dataset('planets')
       planets.shape

Out[2]: (1035, 6)

In[3]: planets.head()

Out[3]:     method           number  orbital_period  mass   distance  year
         0  Radial Velocity  1       269.300         7.10   77.40     2006
         1  Radial Velocity  1       874.774         2.21   56.95     2008
         2  Radial Velocity  1       763.000         2.60   19.84     2011
         3  Radial Velocity  1       326.030         19.40  110.62    2007
         4  Radial Velocity  1       516.220         10.50  119.47    2009
```

Simple Aggregation in Pandas

```
In[4]: rng= np.random.RandomState(42)
```

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
       df

Out[7]:           A         B
         0  0.155995  0.020584
         1  0.058084  0.969910
         2  0.866176  0.832443
         3  0.601115  0.212339
         4  0.708073  0.181825

In[8]: df.mean()

Out[8]: A    0.477888
        B    0.443420
        dtype: float64
```

GroupBy: Split, Apply, Combine




```
In[11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])
        df

Out[11]:    key  data
         0   A     0
         1   B     1
         2   C     2
         3   A     3
         4   B     4
         5   C     5
```

```
In[12]: df.groupby('key')

Out[12]: <pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

```
In[13]: df.groupby('key').sum()

Out[13]:       data
         key
         A        3
         B        5
         C        7
```

Pivot Tables

Titanic example

```
In[1]: import numpy as np
       import pandas as pd
       import seaborn as sns
       titanic = sns.load_dataset('titanic')

In[2]: titanic.head()
```

```
In[3]: titanic.groupby('sex')[['survived']].mean()

Out[3]:         survived
        sex
        female  0.742038
        male    0.188908
```

```
In[4]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()

Out[4]: class      First     Second    Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Pivot Table Syntax

```
In[5]: titanic.pivot_table('survived', index='sex', columns='class')

Out[5]: class      First     Second    Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Multilevel pivot tables

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
       titanic.pivot_table('survived', ['sex', age], 'class')

Out[6]:  class               First     Second    Third
         sex     age
         female  (0, 18]   0.909091  1.000000  0.511628
                 (18, 80]  0.972973  0.900000  0.423729
         male    (0, 18]   0.800000  0.600000  0.215686
                 (18, 80]  0.375000  0.071429  0.133663
```

Survival rate from 0 to 18 and 18 to 80

UNIT V

Visualization with Matplotlib

Color version available online at

https://jakevdp.github.io/PythonDataScienceHandbook/

https://matplotlib.org/

General Matplotlib Tips

```
In[1]: import matplotlib as mpl
import matplotlib.pyplotas plt
```

plt.styledirective to choose appropriate aesthetic styles for our figures

```
In[2]: plt.style.use('classic')
```

Plotting from a script

```
import matplotlib.pyplotas plt
import numpyas np
x = np.linspace(0, 10, 100)     #numpy.linspace(start, stop, num=50)
```

Return evenly spaced numbers over a specified interval.
Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

plt.show() command should be used *only once* per
Python session

Plotting from an IPython shell

IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the %matplotlib magic command after starting ipython:

```
In [1]: %matplotlib #enables the drawing of matplotlib figures in the IPython environment
Using matplotlib backend: TkAgg
In [2]: import matplotlib.pyplotas plt
```

Plotting from an IPython notebook

The IPython notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document

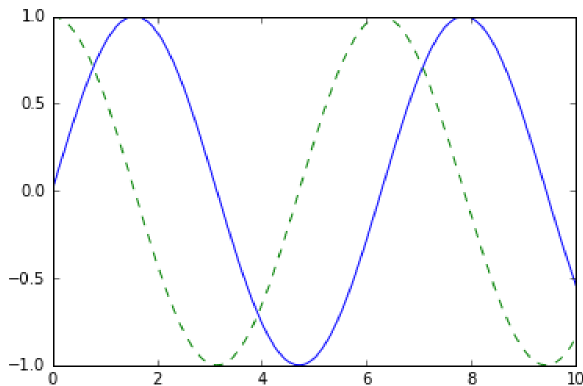
- %matplotlib notebook will lead to *interactive* plots embedded within the notebook
- %matplotlib inline will lead to *static* images of your plot embedded in the Notebook

```
In[3]: %matplotlib inline
```

```
In[4]: import numpyas np
```

AMSCE-101

```
x = np.linspace(0, 10, 100)
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```



Saving Figures to File

In[5]: fig.savefig('my_figure.png')

Save figure as png

IN [6] ls -lh my_figure.png # shows figure properties, ls list file, -lh – human readable format,

Out [6]: -rw-r--r-- 1 jakevdp staff 16K Aug 11 10:59 my_figure.png

read write,

read only for group,

read only for others,

no. of link to file,

name of file owner,

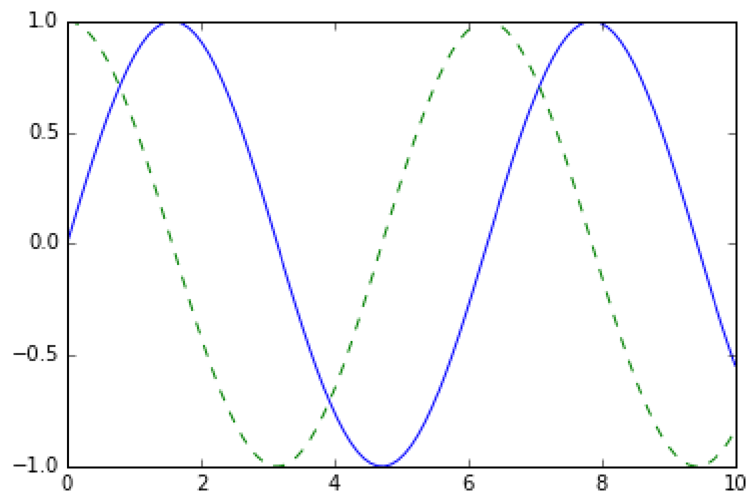
Group associate with file,

size,

last modified.

```
In[7]: from IPython.displayimport Image
Image('my_figure.png')
```

AMSCE-1101



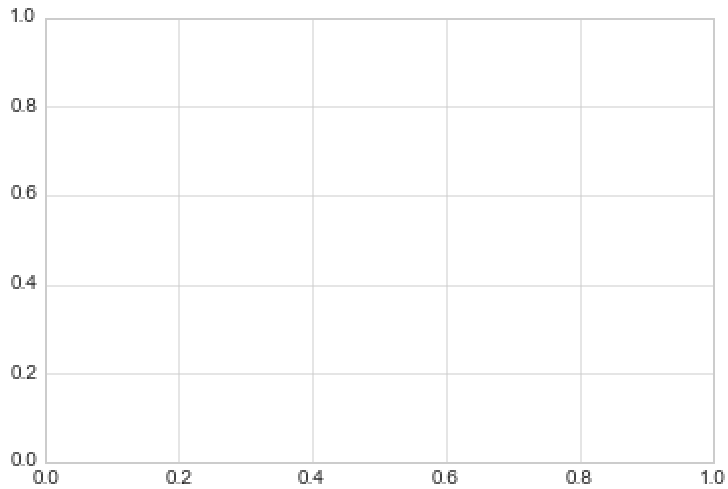

```
In[8]: fig.canvas.get_supported_filetypes()    # list supported file format to save figures
Out[8]: {'eps': 'Encapsulated Postscript',
'jpeg': 'Joint Photographic Experts Group',
'jpg': 'Joint Photographic Experts Group',
'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX',
'png': 'Portable Network Graphics',
'ps': 'Postscript',
'raw': 'Raw RGBA bitmap',
'rgba': 'Raw RGBA bitmap',
'svg': 'Scalable Vector Graphics',
'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format',
'tiff': 'Tagged Image File Format'}
```

AMSCE-1101

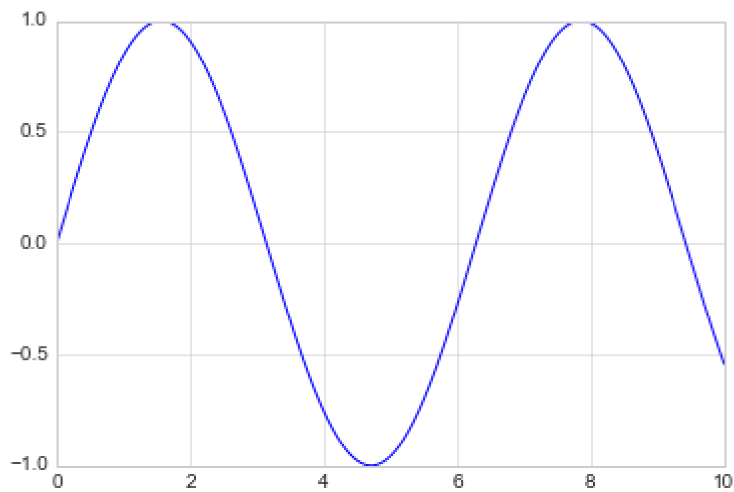
Simple Line Plots

```
In[1]: %matplotlib inline
import matplotlib.pyplotas plt
plt.style.use('seaborn-whitegrid')
import numpyas np
```

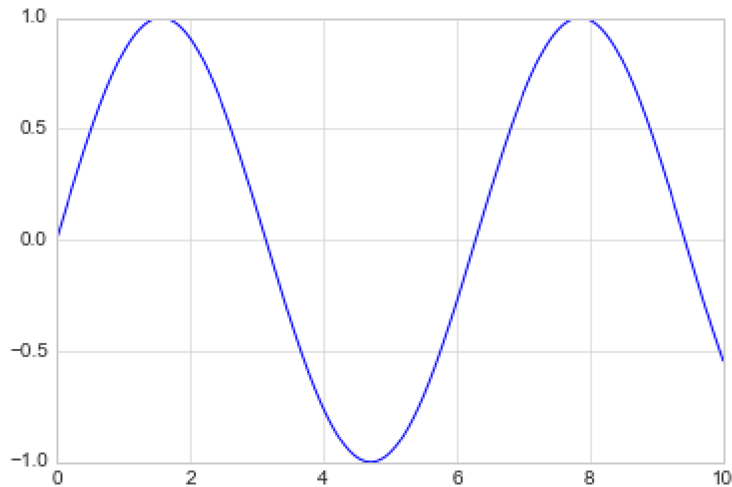
```
In[2]: fig = plt.figure()          #creates figure
ax= plt.axes()          #creates axes
```



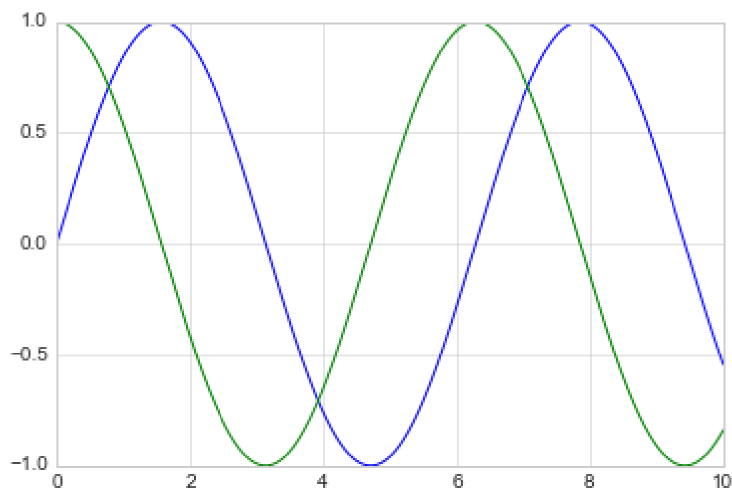

```
In[3]: fig = plt.figure()
ax= plt.axes()
x = np.linspace(0, 10, 1000)
# start, stop, no. of points.
# Return evenly spaced numbers over a specified interval. will also work with out 1000
ax.plot(x, np.sin(x));
```




```
In[4]: plt.plot(x, np.sin(x));
```



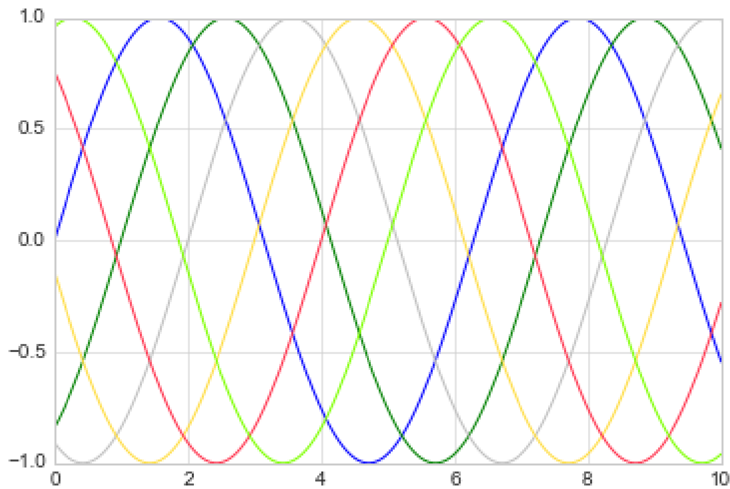

```
In[5]: plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```



Adjusting the Plot: Line Colors and Styles

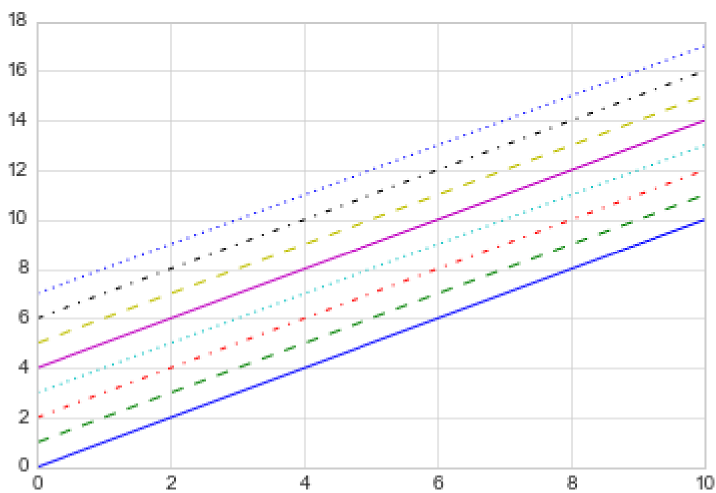
```
In[6]:
plt.plot(x, np.sin(x - 0), color='blue') # specify color by name
plt.plot(x, np.sin(x - 1), color='g') # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

AN...101

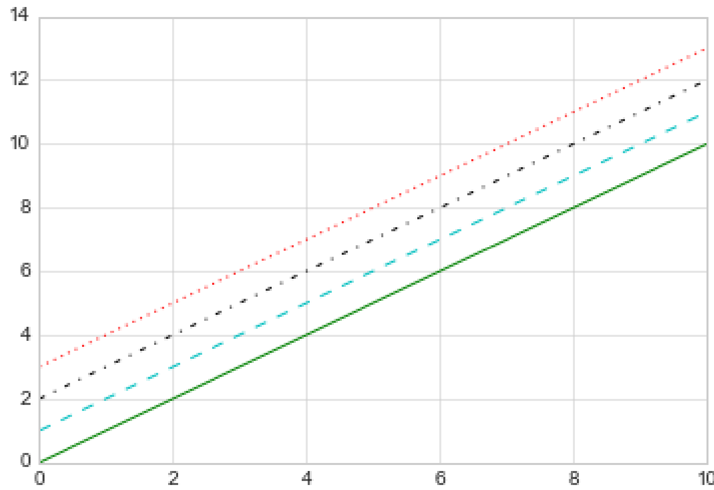



```
In[7]: plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-') # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```

E-1101



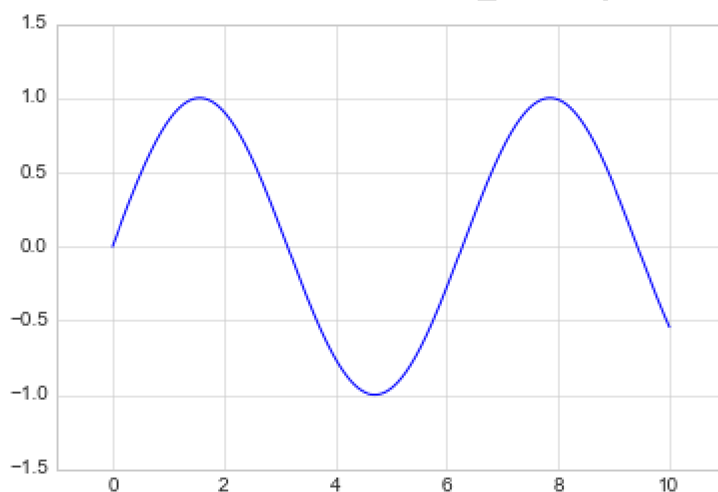
```
In[8]: plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
```



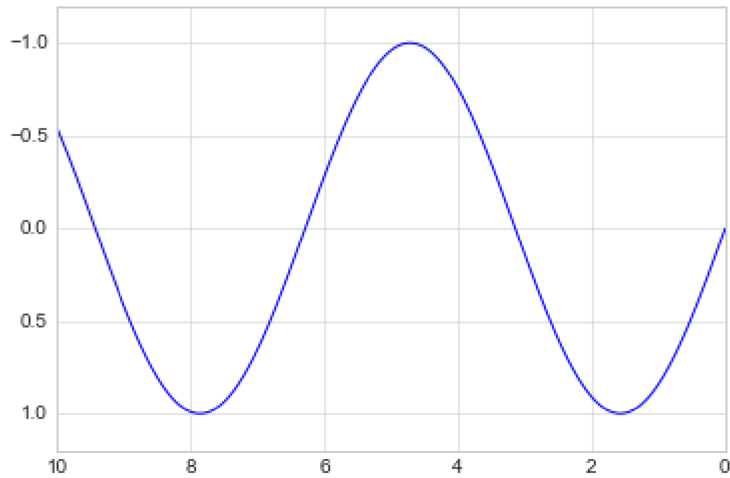
Adjusting the Plot: Axes Limits

```
In[9]: plt.plot(x, np.sin(x))
plt.xlim(-1, 11) # plotted line is between -1 to 11
plt.ylim(-1.5, 1.5);
```

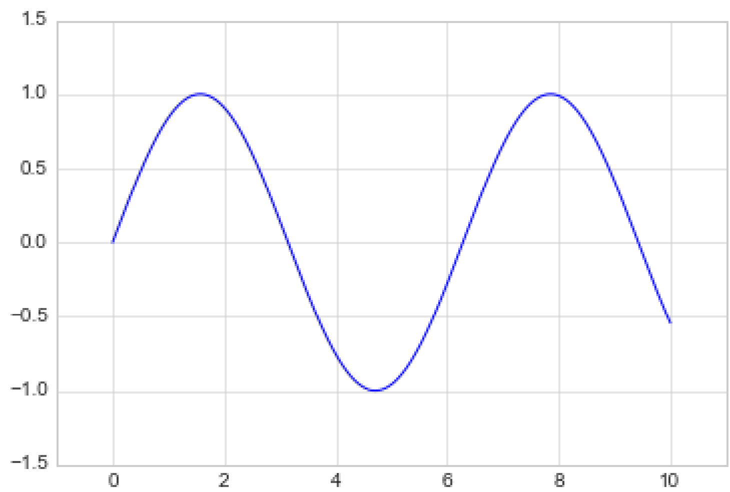
CE-1101



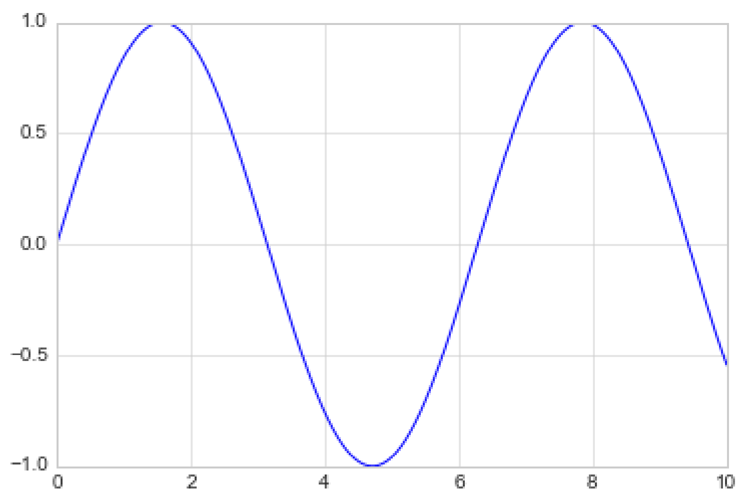

```
In[10]: plt.plot(x, np.sin(x))
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```




```
In[11]: plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]); # x axis limit -1 to 11
```

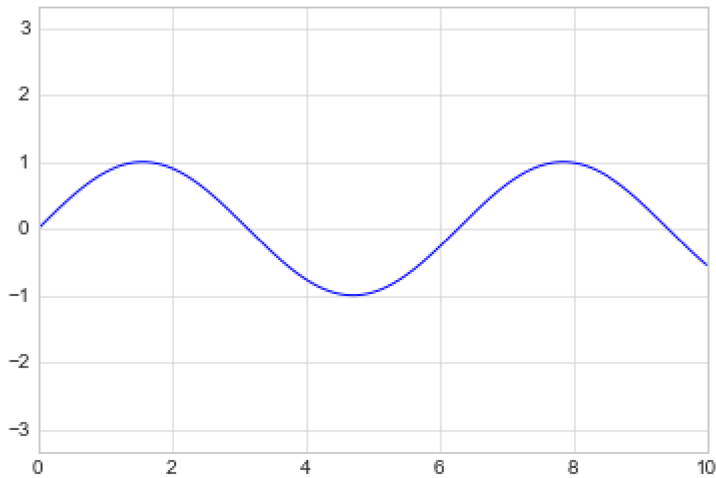



```
In[12]: plt.plot(x, np.sin(x))
plt.axis('tight'); # Frame is tight fitted
```



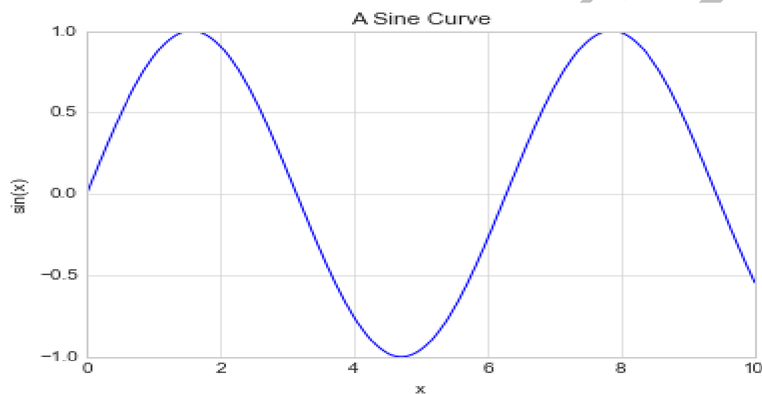

```
In[13]: plt.plot(x, np.sin(x))
```

```
plt.axis('equal');
```

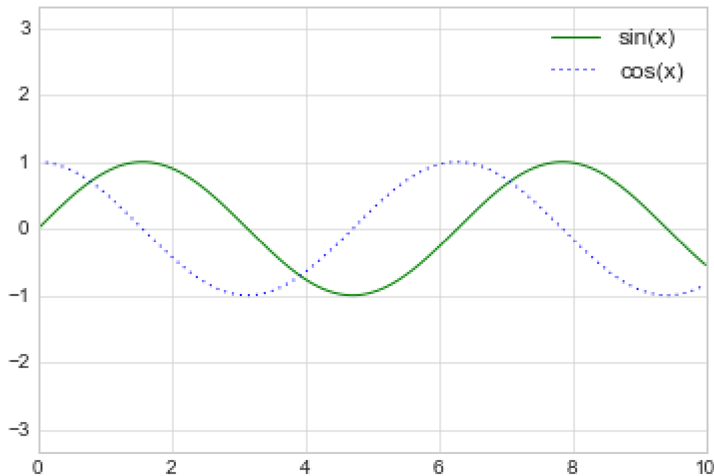


Labeling Plots

```
In[14]: plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```



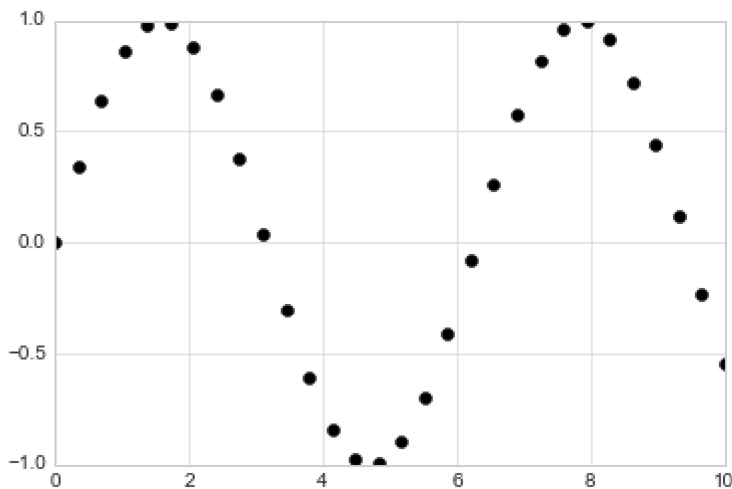

```
In[15]:
plt.plot(x, np.sin(x), '-g', label='sin(x)')        # green label sin
plt.plot(x, np.cos(x), ':b', label='cos(x)')        # blue label cos
plt.axis('equal')
plt.legend();
```



Simple Scatter Plots

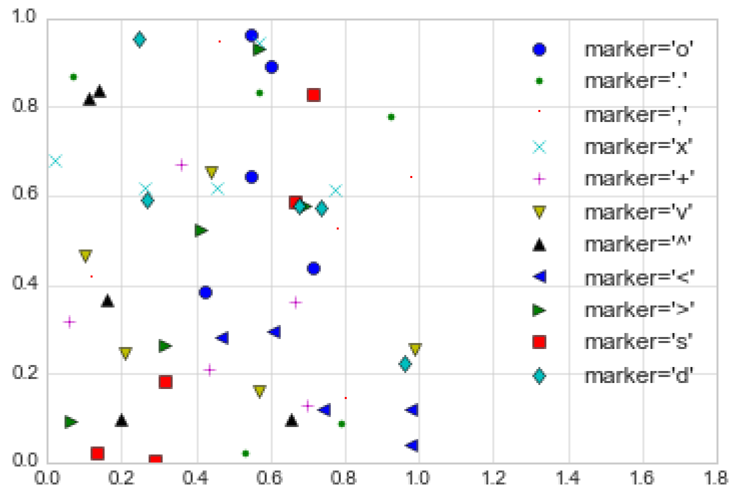
```
In[1]: %matplotlib inline
import matplotlib.pyplotas plt
plt.style.use('seaborn-whitegrid')
import numpyas np
```

```
In[2]: x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black');
```

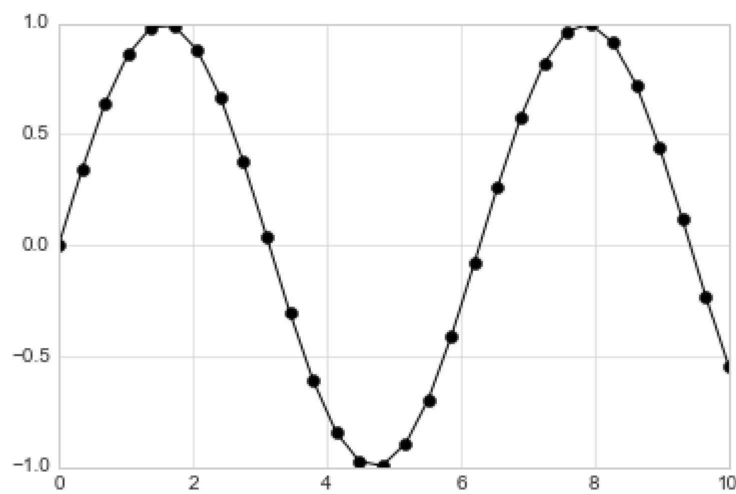


```
In[3]: rng= np.random.RandomState(0) # seed value, produces same random numbers again
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
plt.plot(rng.rand(5), rng.rand(5), marker,label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

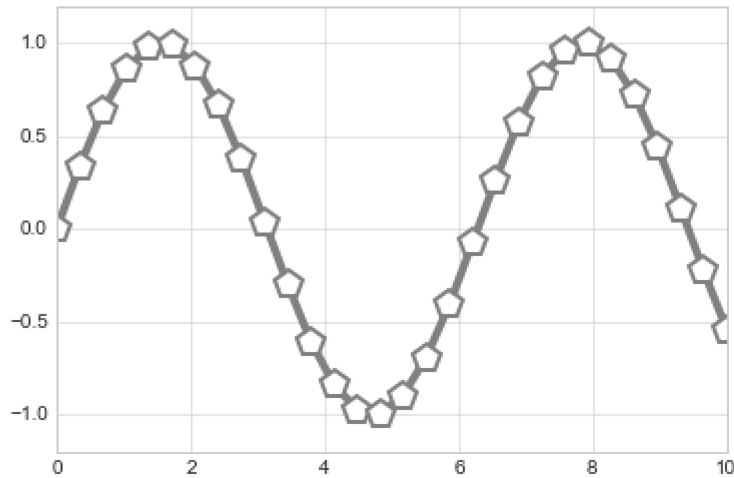
E-1101




```
In[4]: plt.plot(x, y, '-ok'); # line (-), circle marker (o), black (k)
```

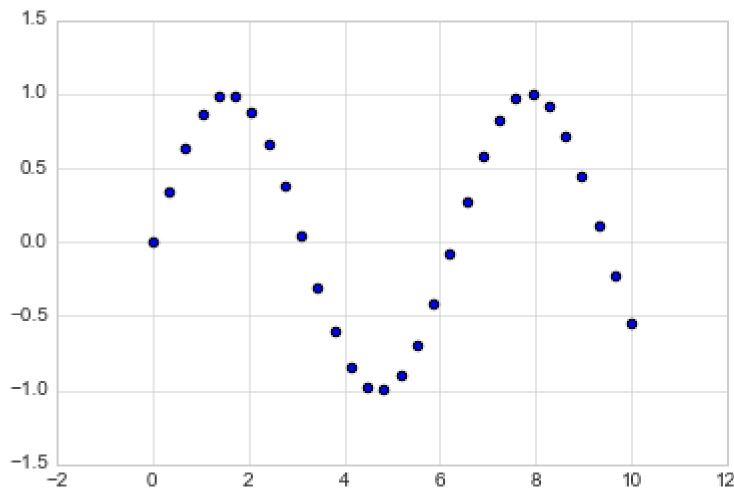



```
In[5]: plt.plot(x, y, '-p', color='gray',  # -p pentagon
markersize=15, linewidth=4,
markerfacecolor='white',
markeredgecolor='gray',
markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

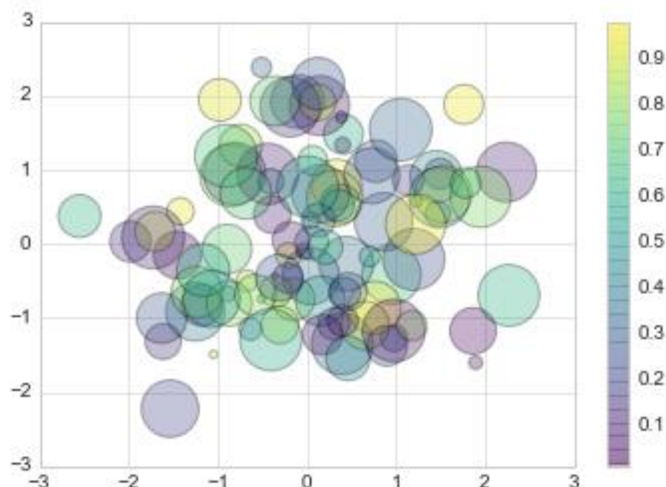


Scatter Plots with plt.scatter

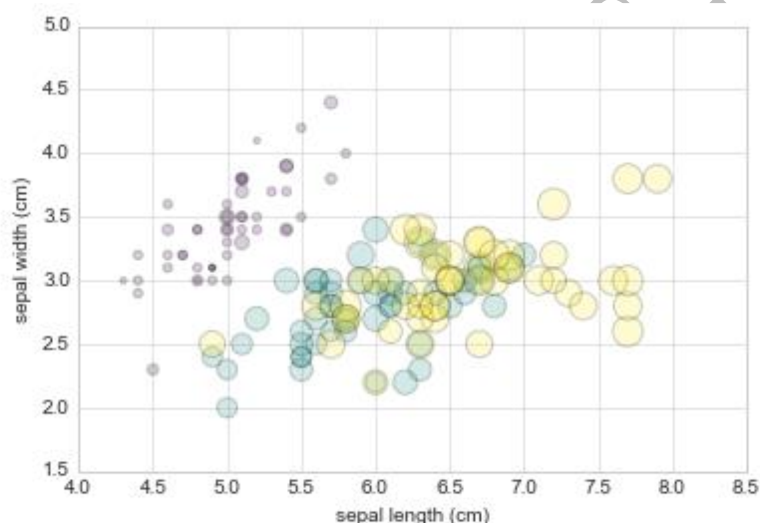
```
In[6]: plt.scatter(x, y, marker='o');
```




```
In[7]: rng= np.random.RandomState(0)
x = rng.randn(100) # Normal distribution, generates an array of 100 random numbers
y = rng.randn(100)
colors= rng.rand(100)
sizes = 1000 * rng.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis') # alpla - transparency and color
map style
plt.colorbar(); # show color scale
```




```
In[8]: from sklearn.datasetsimport load_iris
iris = load_iris()
features = iris.data.T   #Transpose
plt.scatter(features[0], features[1], alpha=0.2,s=100*features[3], c=iris.target, cmap='viridis')
# color of marker for each target variable – iris.target.
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
```



Visualizing Errors

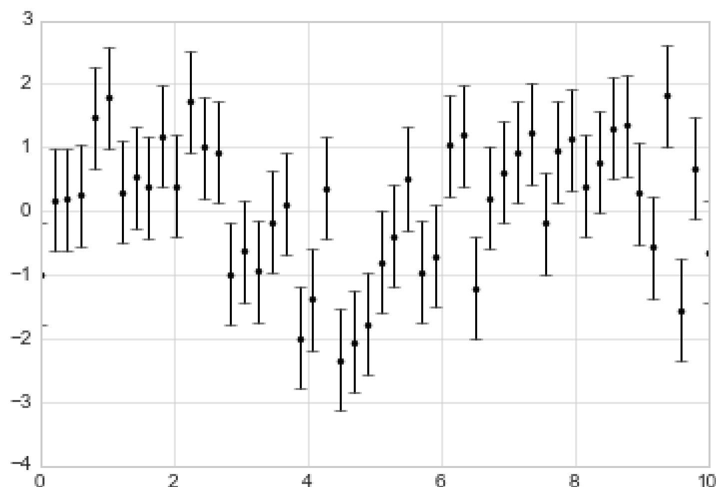
Basic Errorbars

```
In[1]: %matplotlib inline
import matplotlib.pyplotas plt
plt.style.use('seaborn-whitegrid')
import numpyas np
In[2]: x = np.linspace(0, 10, 50)
# start, stop, no. of points.
# Return evenly spaced numbers over a specified interval.
dy= 0.8
y = np.sin(x) + dy* np.random.randn(50)
```

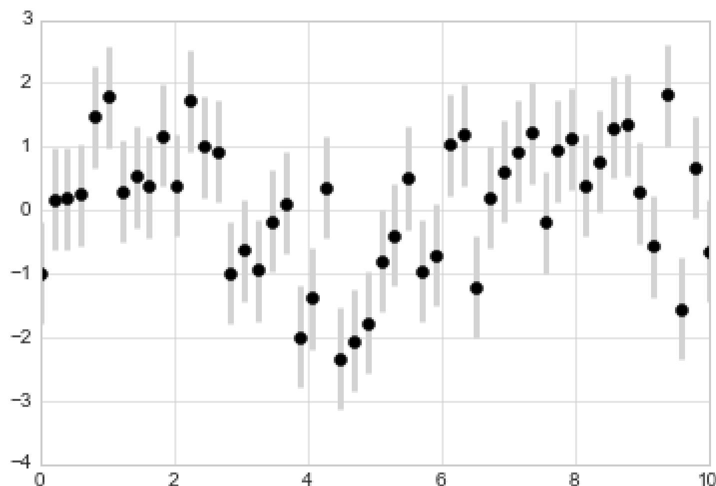
plt.errorbar(x, y, yerr=dy, fmt='.k'); *#**xerr, yerr:** These parameter contains an array.And the error array should have positive values.*

fmt: This parameter is an optional parameter and it contains the string value.

K means data points will be black in color.



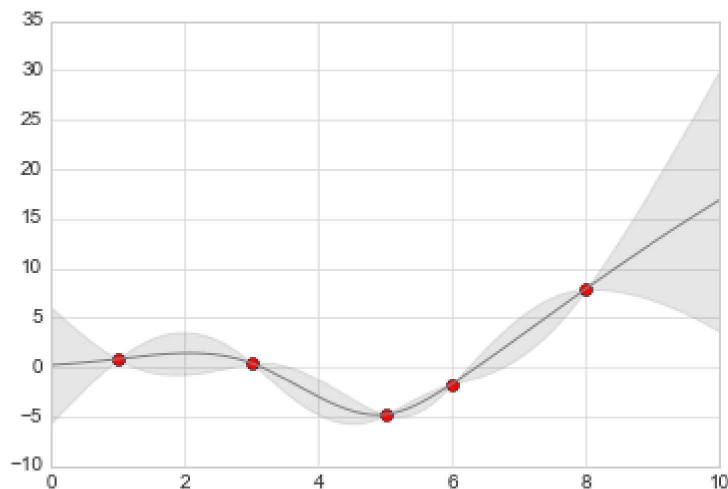

```
In[3]: plt.errorbar(x, y, yerr=dy, fmt='o', color='black', ecolor='lightgray', elinewidth=3,
capsize=0);
# edge color,
# Edge line width
# Capsize length of the caps at the end of the error bars
```



Continuous Errors

```
In[4]: from sklearn.gaussian_processimport GaussianProcess
# define the model and draw some data
model = lambda x: x * np.sin(x)
xdata= np.array([1, 3, 5, 6, 8])
ydata= model(xdata)
# Compute the Gaussian process fit
# cubic correlation function
gp= GaussianProcess(corr='cubic', theta0=1e-2, -thetaL=1e 4, thetaU=1E-
1,random_start=100)
```

```
gp.fit(xdata[:, np.newaxis], ydata)
xfit= np.linspace(0, 10, 1000)
yfit, MSE = gp.predict(xfit[:, np.newaxis], eval_MSE=True)
dyfit= 2 * np.sqrt(MSE) # 2*sigma ~ 95% confidence region
In[5]: # Visualize the result
        plt.plot(xdata, ydata, 'or')
        plt.plot(xfit, yfit, '-', color='gray')
        plt.fill_between(xfit, yfit- dyfit, yfit+ dyfit,
        color='gray', alpha=0.2)
        plt.xlim(0, 10);
```



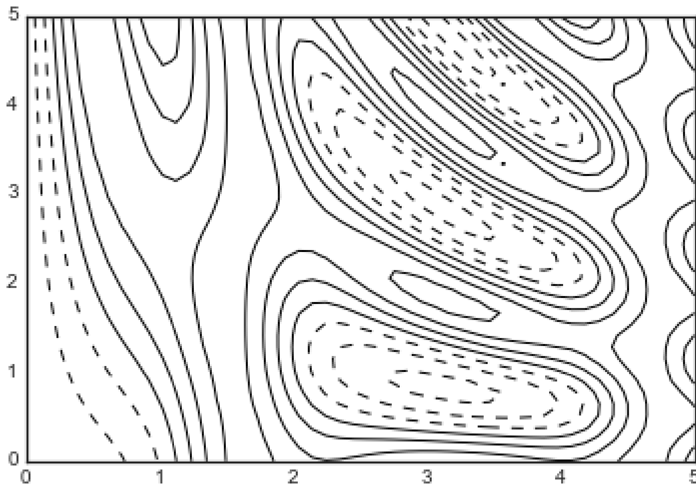
Density and Contour Plots

```
In[1]: %matplotlib inline
import matplotlib.pyplotas plt
plt.style.use('seaborn-white')
import numpyas np
```

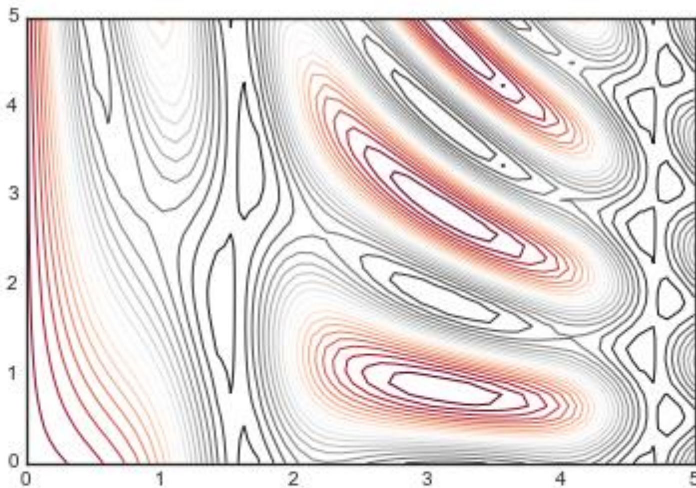
Visualizing a Three-Dimensional Function

```
In[2]: def f(x, y):
return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
In[3]: x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)      #Return coordinate matrices from coordinate vectors
Z = f(X, Y)
In[4]: plt.contour(X, Y, Z, colors='black');
```

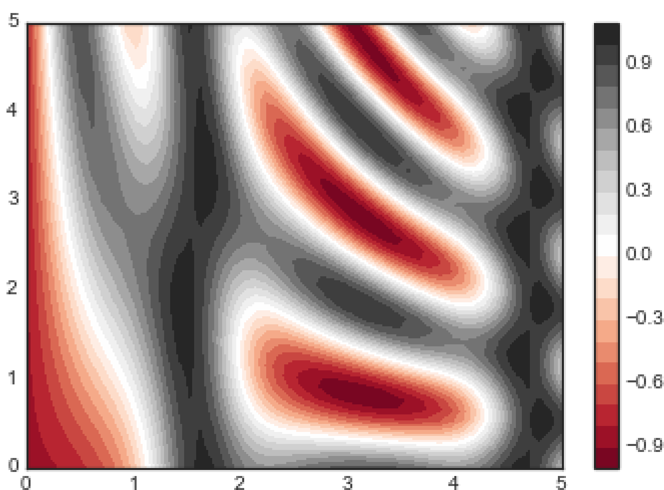
AMSCE 101



In[5]: plt.contour(X, Y, Z, 20, cmap='RdGy');# 20 - contour levels

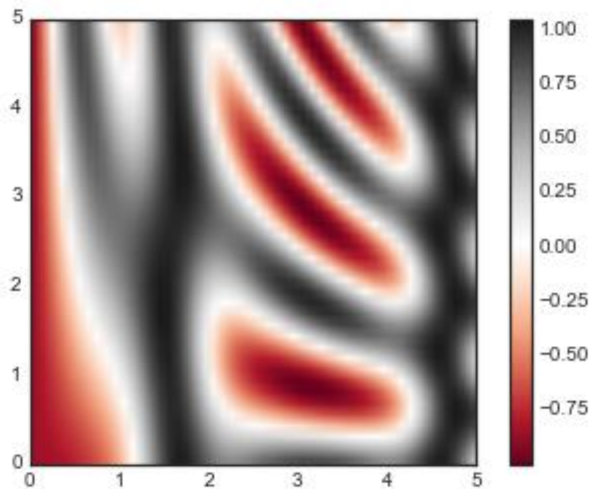


In[6]: plt.contourf(X, Y, Z, 20, cmap='RdGy') # 20 - contour levels
plt.colorbar();

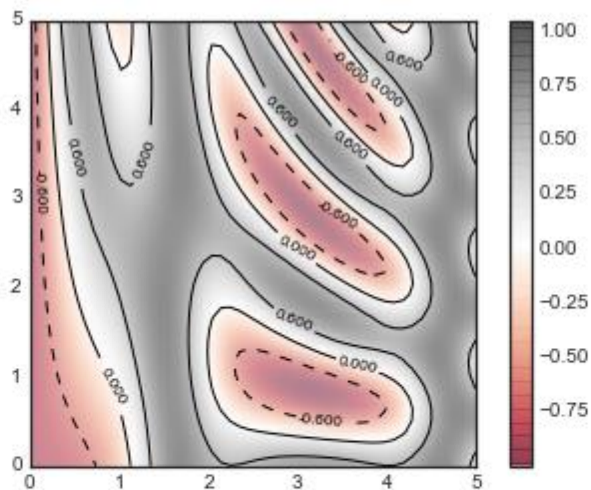


In[7]: plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',cmap='RdGy')
# converts color step to continuous. plt.imshow() doesn't accept an *x* and *y* grid, so you must manually specify the*extent*[*xmin*, *xmax*, *ymin*, *ymax*]
plt.colorbar()

```
plt.axis(aspect='image'); #aspect – aspect ratio
```

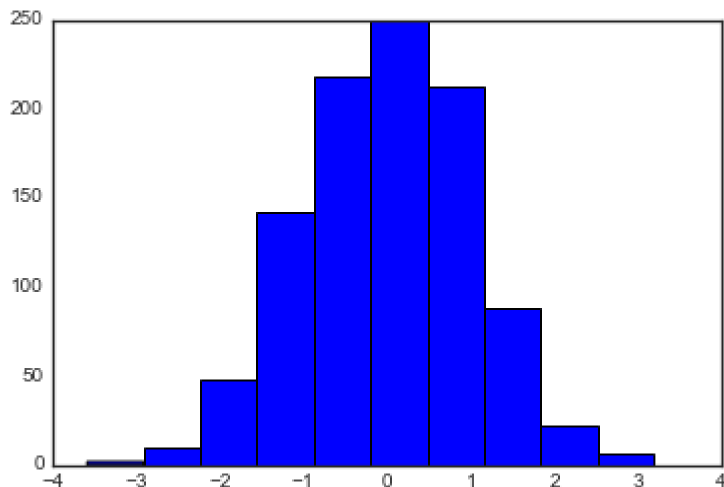



```
In[8]: contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8) #contour labels are placed inline next to contor line
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy', alpha=0.5) # extent x, y axis of displayed image
plt.colorbar();
```

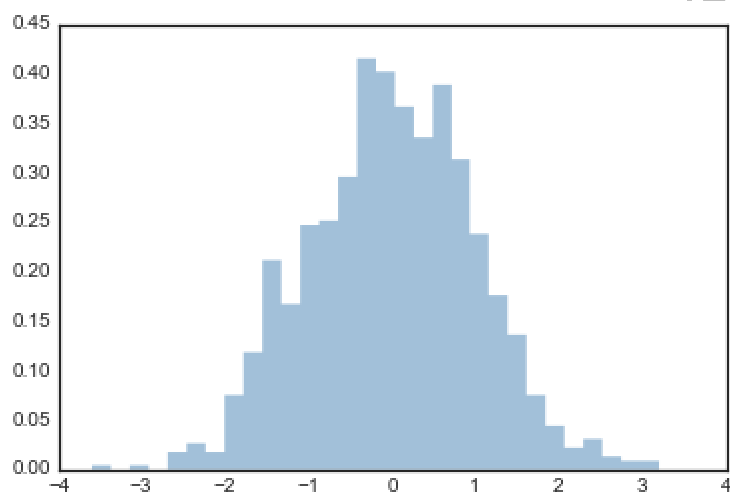


Histograms, Binnings, and Density

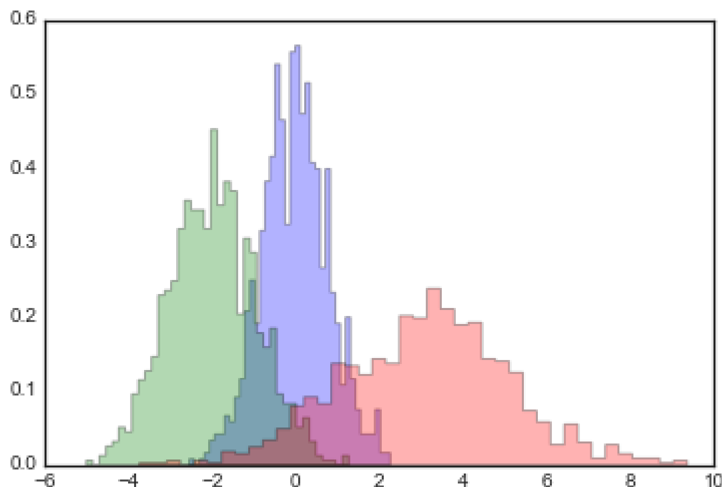
```
In[1]: %matplotlib inline
       import numpyas np
       import matplotlib.pyplotas plt
       plt.style.use('seaborn-white')
       data = np.random.randn(1000)
In[2]: plt.hist(data);
```




```
In[3]: plt.hist(data, bins=30, normed=True, alpha=0.5, histtype='stepfilled', color='steelblue',
       edgecolor='none'); # bins – bar, normed - histogram is normalized,
       histtype - generates a lineplot that is by default filled.
       histtype{'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'
```




```
In[4]: x1 = np.random.normal(0, 0.8, 1000)
       x2 = np.random.normal(-2, 1, 1000)
       x3 = np.random.normal(3, 2, 1000)
       kwargs= dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)
       plt.hist(x1, **kwargs) #**kwargs in function definitions in python is used to pass a
       keyworded, variable-length argument list
       plt.hist(x2, **kwargs)
       plt.hist(x3, **kwargs);
```




```
In[5]: counts, bin_edges= np.histogram(data, bins=5) # bin_edges – contain edges of bin
       print(counts)
       [ 12 190 468 301 29]
```

Two-Dimensional Histograms and Binnings

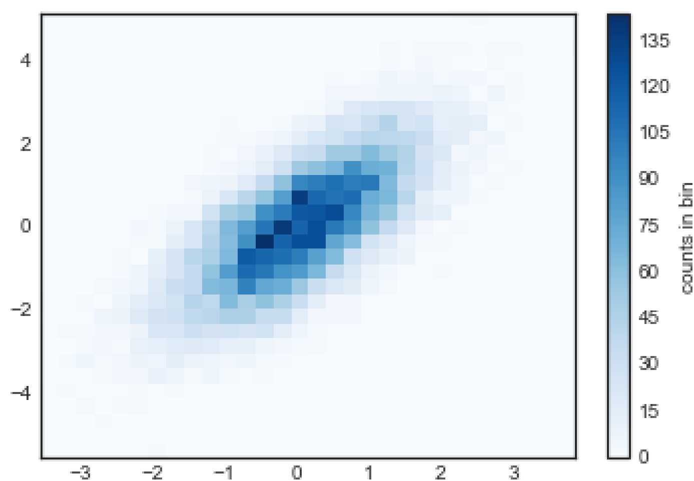
- Shows how much energy available in two dimensional bins
- Bins are represented in both x and y axis

```
In[6]: mean = [0, 0]
       cov= [[1, 1], [1, 2]]
       x, y = np.random.multivariate_normal(mean, cov, 10000).T#generate samples from
       multivariate normal distribution
```

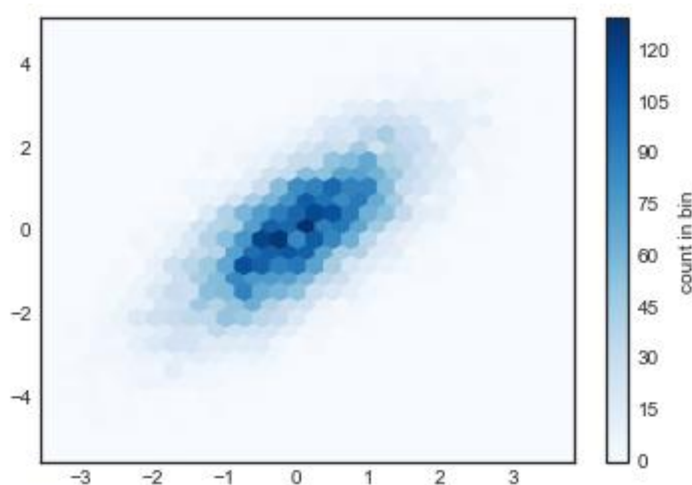
plt.hist2d: Two-dimensional histogram

```
In[12]: plt.hist2d(x, y, bins=30, cmap='Blues')
       cb= plt.colorbar()
       cb.set_label('counts in bin')
```

CCE-1101




```
In[8]: counts, xedges, yedges= np.histogram2d(x, y, bins=30)        # return values count,x,y
       plt.hexbin: Hexagonal binnings
In[9]: plt.hexbin(x, y, gridsize=30, cmap='Blues')
       cb= plt.colorbar(label='count in bin')
```

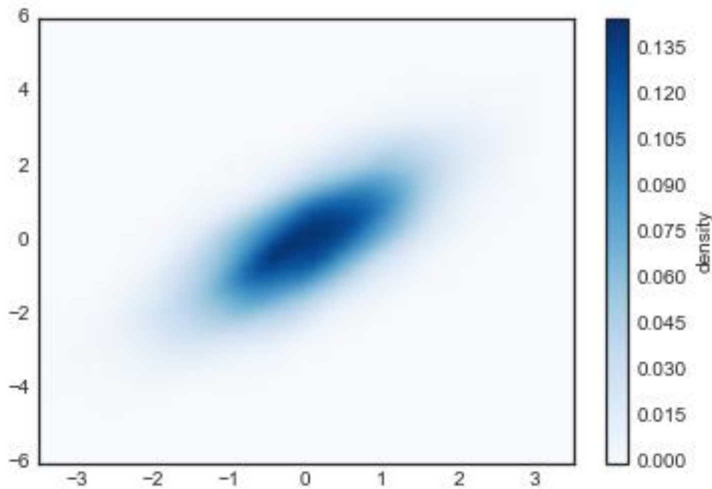


Kernel density estimation.

- Another common method of evaluating densities in multiple dimensions is kernel density estimation (KDE).
- KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function.

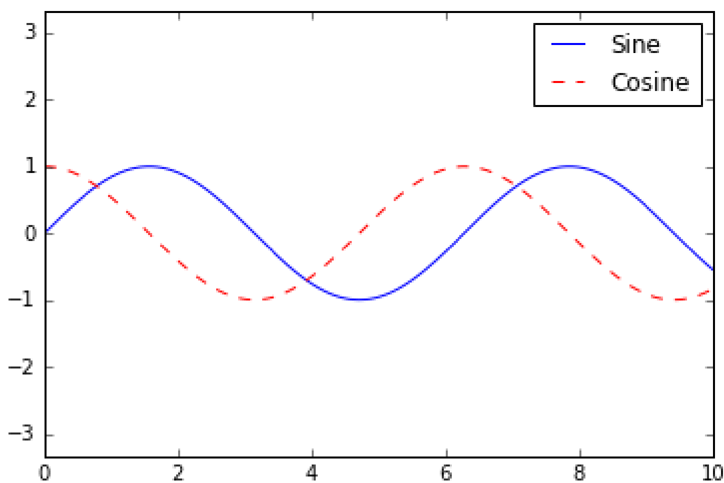
In[10]:
```
from scipy.statsimport gaussian_kde
# fit an array of size [Ndim, Nsamples]
data = np.vstack([x, y])
kde= gaussian_kde(data)

# evaluate on a regular grid
xgrid= np.linspace(-3.5, 3.5, 40)
ygrid= np.linspace(-6, 6, 40)
Xgrid, Ygrid= np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()])) # returns 1D array




# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),origin='lower', aspect='auto',extent=[-3.5, 3.5, -6,
6],cmap='Blues') # reshapes 1D Z to 2D #aspect ratio # x and y axiss        # color
map
cb= plt.colorbar()
cb.set_label("density")
```

AMSCE-1101



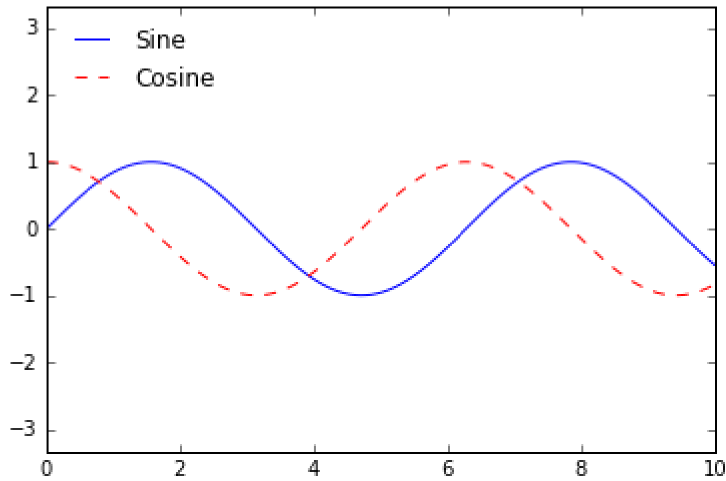
Customizing Plot Legends

```
In[1]: import matplotlib.pyplotas plt
       plt.style.use('classic')
In[2]: %matplotlib inline
       import numpyas np
In[3]: x = np.linspace(0, 10, 1000)
       fig, ax= plt.subplots()  # create a figure and a set of subplots.
       ax.plot(x, np.sin(x), '-b', label='Sine')
       ax.plot(x, np.cos(x), '--r', label='Cosine')
       ax.axis('equal')
       leg = ax.legend(); # prints legend
```

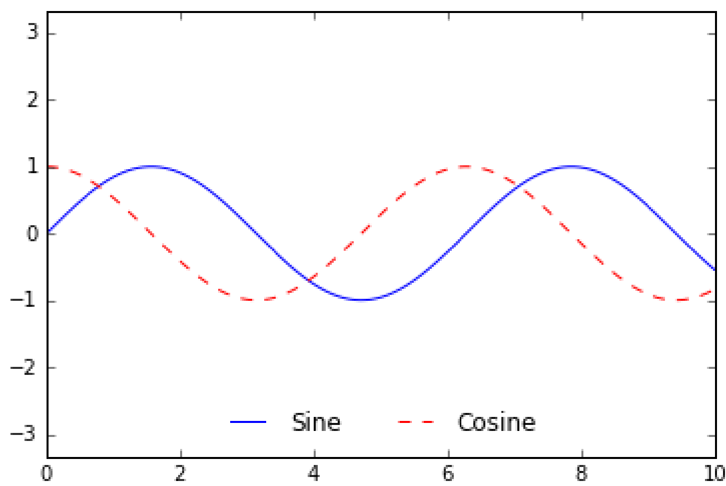



```
In[4]: ax.legend(loc='upper left', frameon=False)
       Fig
```

E-1101

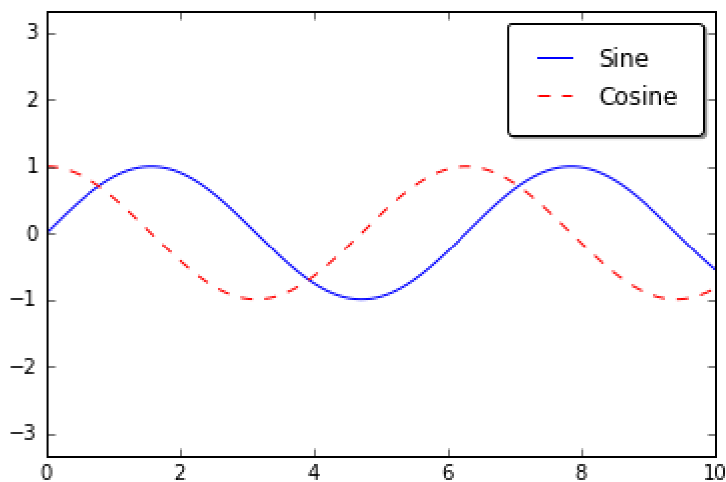


In[5]: ax.legend(frameon=False, loc='lower center', ncol=2)
Fig



M 101

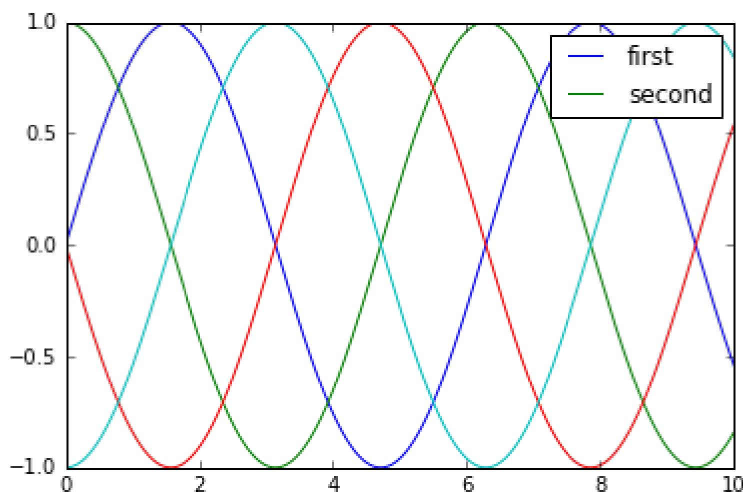
In[6]: ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
frame alpha – type of box frame
Fig



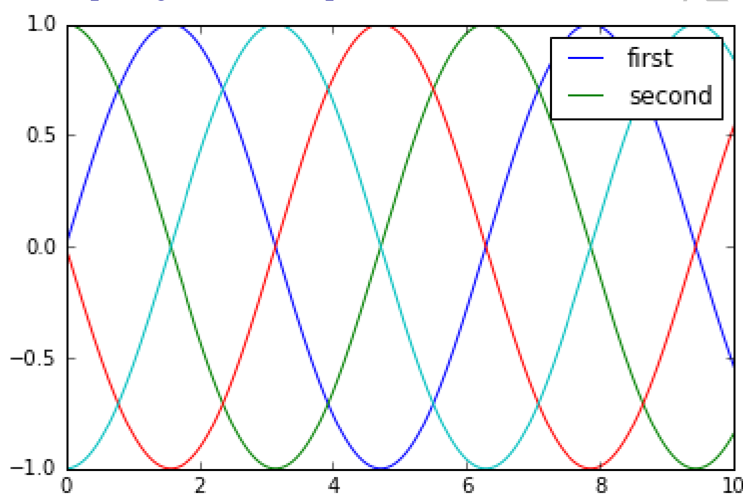
Choosing Elements for the Legend

In[7]: y = np.sin(x[:, np.newaxis] + np.pi* np.arange(0, 2, 0.5)) # arrange return evenly spaced values within a given interval
lines = plt.plot(x, y) # plots multiple lines at once

```
# lines is a list of plt.Line2D instances
plt.legend(lines[:2], ['first', 'second']);
```



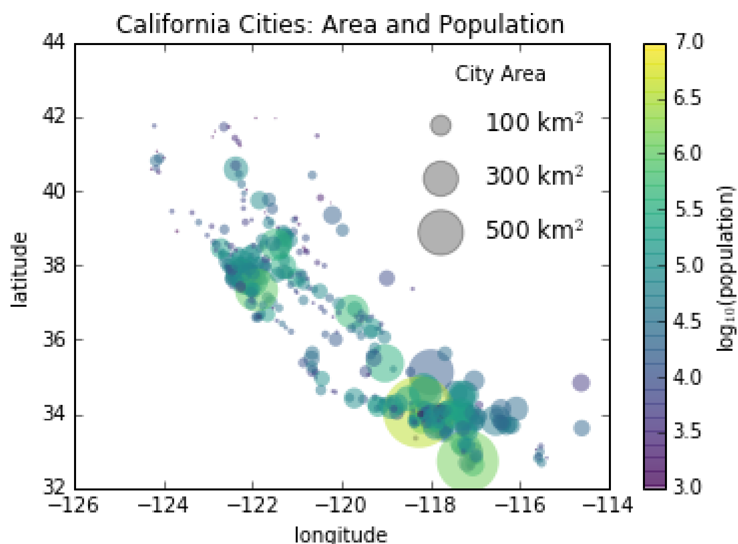

```
In[8]: plt.plot(x, y[:, 0], label='first')
       plt.plot(x, y[:, 1], label='second')
       plt.plot(x, y[:, 2:])  #x with y - all columns from the third column onward
       plt.legend(framealpha=1, frameon=True);
```



Legend for Size of Points

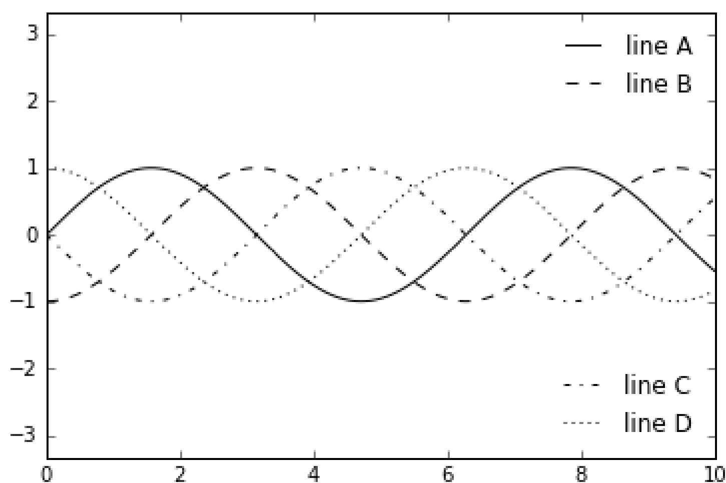
```
In[9]: import pandas as pd
       cities = pd.read_csv('data/california_cities.csv')
       # Extract the data we're interested in
       lat, lon= cities['latd'], cities['longd']
       population, area = cities['population_total'], cities['area_total_km2']
       # Scatter the points, using size and color but no label
       plt.scatter(lon, lat, label=None,c=np.log10(population), cmap='viridis',
       s=area, linewidth=0, alpha=0.5)
       plt.axis(aspect='equal')
       plt.xlabel('longitude')
       plt.ylabel('latitude')
       plt.colorbar(label='log$_{10}$(population)')
       plt.clim(3, 7)   #Set the color limits of the current image. 3- lower, 7 – upper
       # Here we create a legend:
       # we'll plot empty lists with the desired size and label
```

```
for area in [100, 300, 500]: #For area in values of 100, 300, 500
plt.scatter([], [], c='k', alpha=0.3, s=area, label=str(area) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')
plt.title('California Cities: Area and Population');
```



Multiple Legends

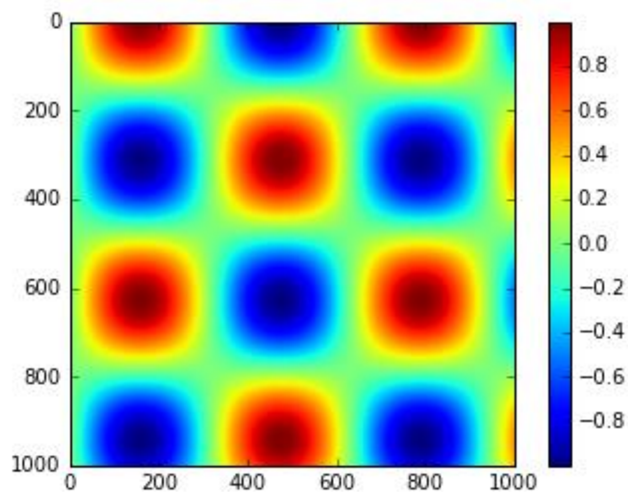
```
In[10]: fig, ax= plt.subplots()
        lines = []
        styles = ['-', '--', '-.', ':']
        x = np.linspace(0, 10, 1000)
        for iin range(4):
        lines += ax.plot(x, np.sin(x - i* np.pi/ 2),styles[i], color='black')
        ax.axis('equal')
        # specify the lines and labels of the first legend
        ax.legend(lines[:2], ['line A', 'line B'],
        loc='upper right', frameon=False)
        # Create the second legend and add the artist manually.
        from matplotlib.legendimport Legend
        leg = Legend(ax, lines[2:], ['line C', 'line D'],
        loc='lower right', frameon=False)
        ax.add_artist(leg);
```



AMSCE-1101

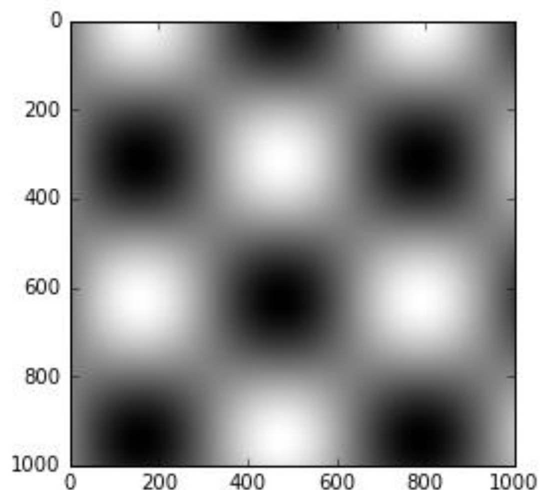
Customizing Colorbars

```
In[1]: import matplotlib.pyplotas plt
       plt.style.use('classic')
In[2]: %matplotlib inline
       import numpyas np
In[3]: x = np.linspace(0, 10, 1000)
       I = np.sin(x) * np.cos(x[:, np.newaxis])
       plt.imshow(I)
       plt.colorbar();
```



Customizing Colorbars

```
In[4]: plt.imshow(I, cmap='gray');
```



CE-1101

Choosing the colormap

- ***Sequential colormaps***

These consist of one continuous sequence of colors (e.g., binary or viridis).

- ***Divergent colormaps***

These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., RdBuor PuOr).

- ***Qualitative colormaps***

These mix colors with no particular sequence (e.g., rainbow or jet).

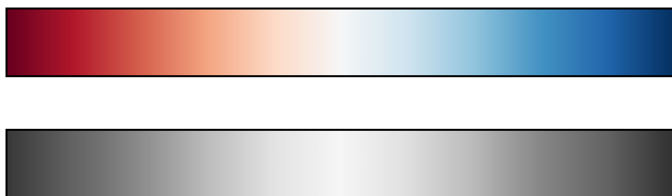
```
frommatplotlib.colorsimportLinearSegmentedColormap

defgrayscale_cmap(cmap):
"""Return a grayscale version of the given colormap"""
cmap=plt.cm.get_cmap(cmap)
colors=cmap(np.arange(cmap.N))

# convert RGBA to perceived grayscale luminance
# cf. http://alienryderflex.com/hsp.html
RGB_weight= [0.299, 0.587, 0.114]
    luminance =np.sqrt(np.dot(colors[:, :3] **2, RGB_weight))
colors[:, :3] = luminance[:, np.newaxis]            #color store rgb values,

returnLinearSegmentedColormap.from_list(cmap.name +"_gray", colors, cmap.N)


defview_colormap(cmap):
"""Plot a colormap with its grayscale equivalent"""
cmap=plt.cm.get_cmap(cmap)
colors=cmap(np.arange(cmap.N))

cmap=grayscale_cmap(cmap)
    grayscale =cmap(np.arange(cmap.N))

    fig, ax=plt.subplots(2, figsize=(6, 2), #size of figure
subplot_kw=dict(xticks=[], yticks=[]))# ticks are empty
ax[0].imshow([colors], extent=[0, 10, 0, 1]) # extent image boundary
ax[1].imshow([grayscale], extent=[0, 10, 0, 1]) # limits of x and y axis
```

In[6]: view_colormap('jet')

In[7]: view_colormap('viridis')

In[8]: view_colormap('cubehelix')

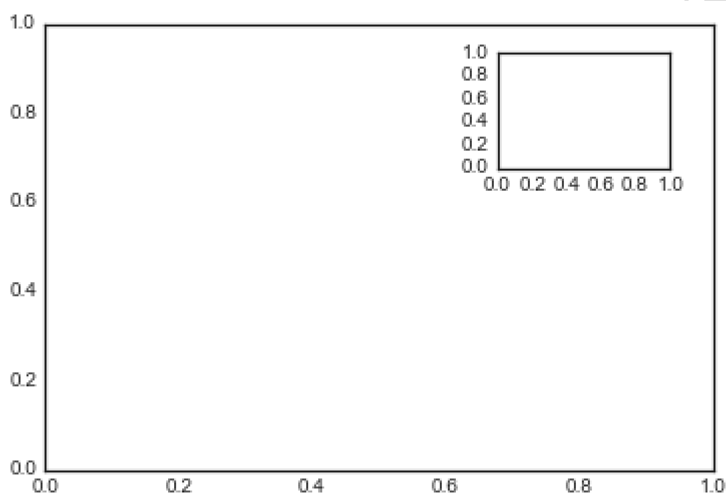
```
In[9]: view_colormap('RdBu')
```



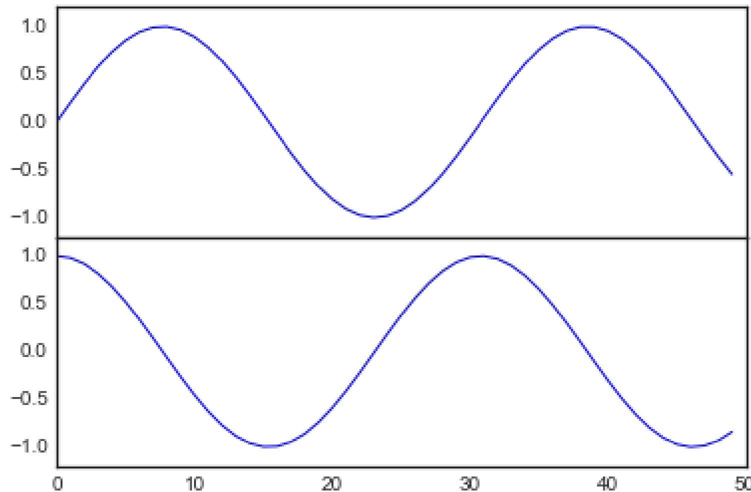
Multiple Subplots

```
In[1]: %matplotlib inline
       import matplotlib.pyplotas plt
       plt.style.use('seaborn-white')
       import numpyas np
In[2]: ax1 = plt.axes() # standard axes
       ax2 = plt.axes([0.65, 0.65, 0.2, 0.2]) # position, size of subplot
```

Create an inset axes at the top-right corner of another axes by setting the *x* and *y* position to 0.65 (that is, starting at 65% of the width and 65% of the height of the figure) and the *x* and *y* extents to 0.2 (that is, the size of the axes is 20% of the width and 20% of the height of the figure).

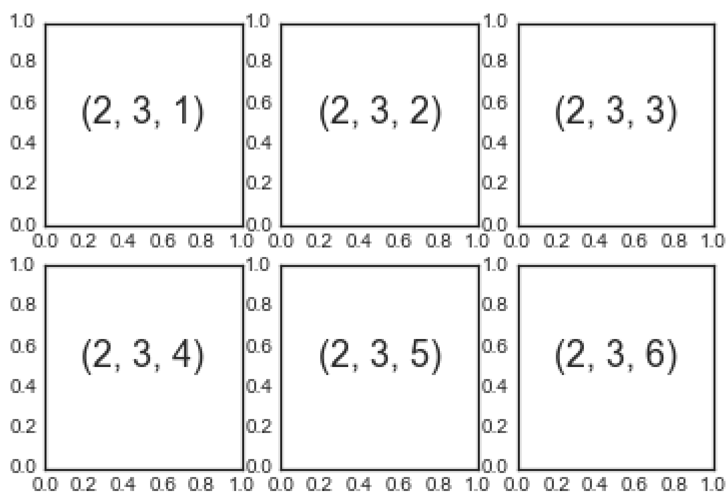



```
In[3]: fig = plt.figure()
       ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],  #position and size of the subplot within the
       figure.
       xticklabels=[], ylim=(-1.2, 1.2))
       ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
       ylim=(-1.2, 1.2))
       x = np.linspace(0, 10)
       ax1.plot(np.sin(x))
       ax2.plot(np.cos(x));
```

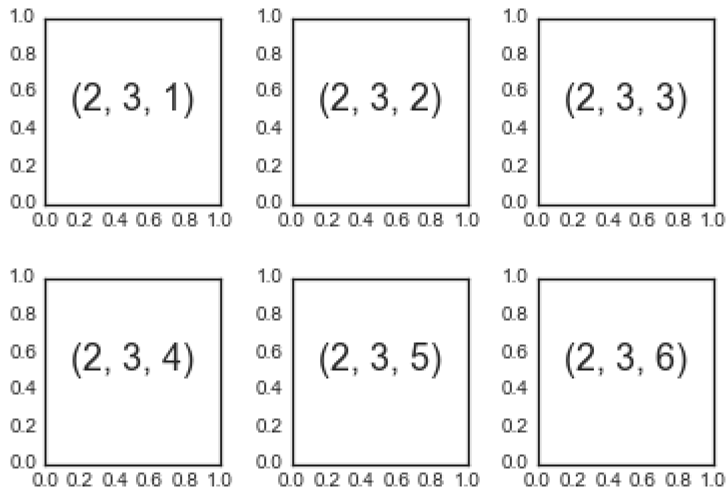


plt.subplot: Simple Grids of Subplots

```
In[4]: for iin range(1, 7):
        plt.subplot(2, 3, i)
        plt.text(0.5, 0.5, str((2, 3, i)),fontsize=18, ha='center')# Text coordinate
```

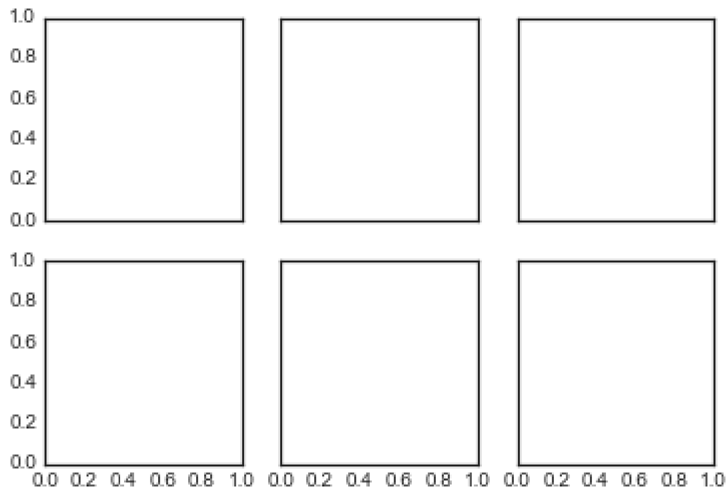



```
In[5]: fig = plt.figure()
        fig.subplots_adjust(hspace=0.4, wspace=0.4)#set spacing bw plots, height width
        spacing bw subplots
        for iin range(1, 7):
        ax= fig.add_subplot(2, 3, i)
        ax.text(0.5, 0.5, str((2, 3, i)),
        fontsize=18, ha='center')
```

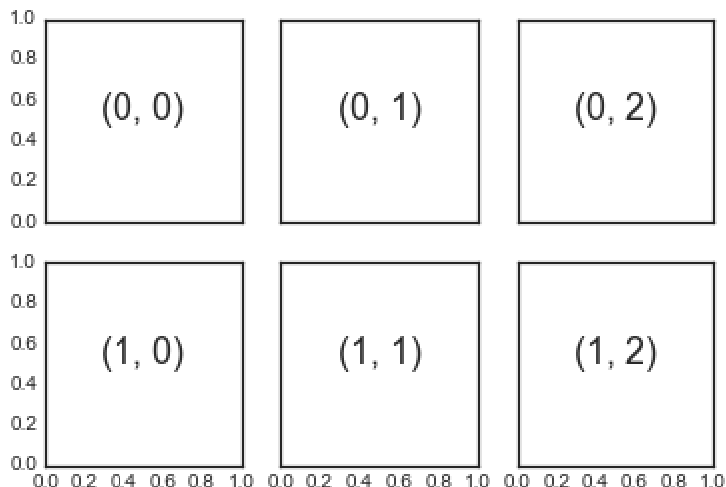


plt.subplots: The Whole Grid in One Go

In[6]: fig, ax= plt.subplots(2, 3, sharex='col', sharey='row') #row, column , share x, y axis scale

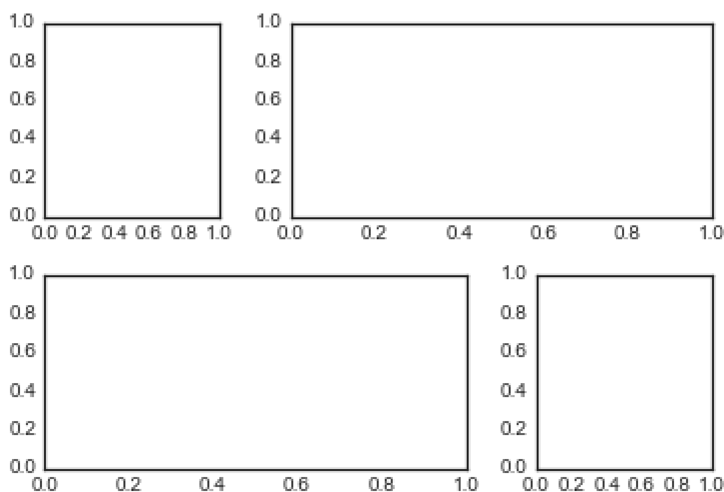



```
In[7]: # axes are in a two-dimensional array, indexed by [row, col]
        for iin range(2):
        for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)),
        fontsize=18, ha='center')
        fig
```



plt.GridSpec: More Complicated Arrangements

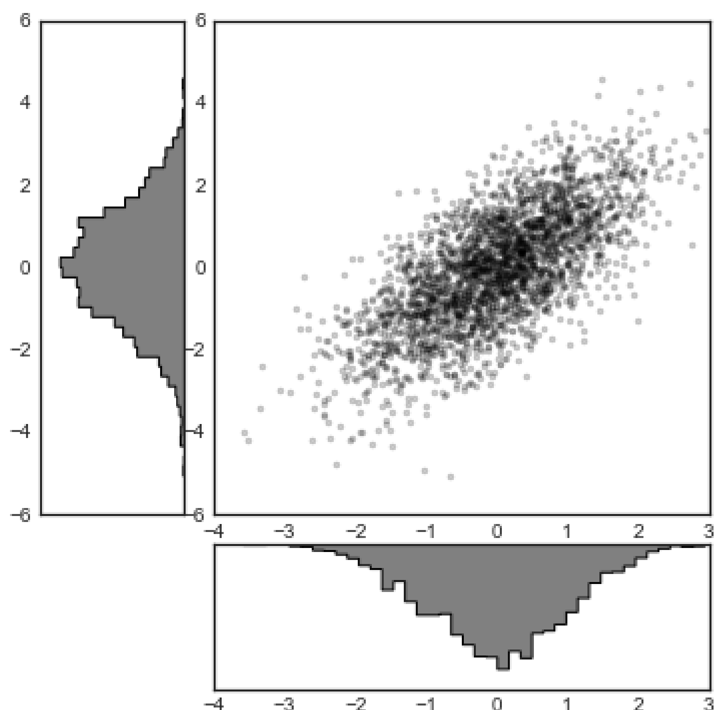
```
In[8]: grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3) # 2 rows and 3 columns, width , height
In[9]: plt.subplot(grid[0, 0])
       plt.subplot(grid[0, 1:])
       plt.subplot(grid[1, :2])
       plt.subplot(grid[1, 2]);
```



```
In[10]: # Create some normally distributed data
        mean = [0, 0]
        cov= [[1, 1], [1, 2]]
        x, y = np.random.multivariate_normal(mean, cov, 3000).T
        # Set up the axes with gridspec
        fig = plt.figure(figsize=(6, 6)) # width , height
        grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)#grid layout to place subplotswithin a figure
        main_ax= fig.add_subplot(grid[:-1, 1:])
        y_hist= fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax) #left fig
        x_hist= fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax) # right fig
        # scatter points on the main axes
        main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2) # center figure
        # histogram on the attached axes
```

1101

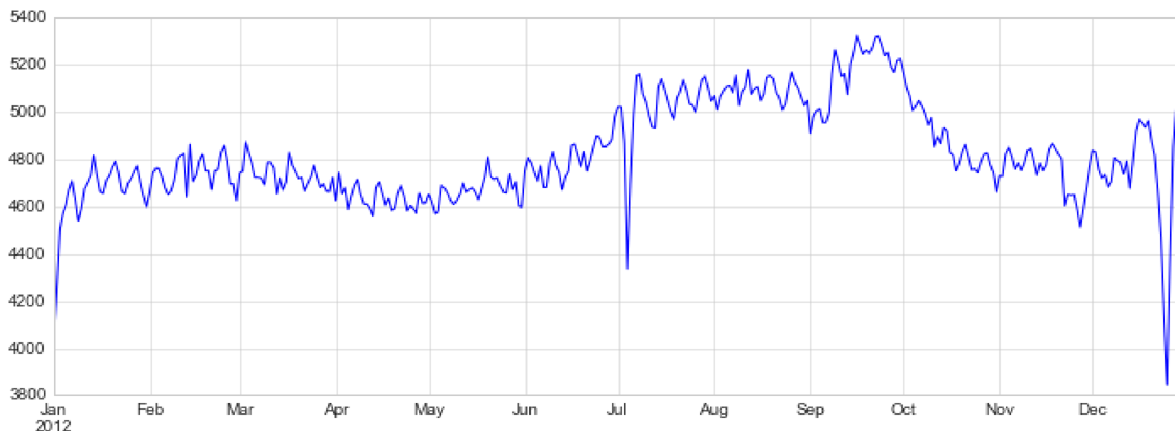
```
x_hist.hist(x, 40, histtype='stepfilled',orientation='vertical', color='gray')
        x_hist.invert_yaxis()
y_hist.hist(y, 40, histtype='stepfilled',orientation='horizontal', color='gray')
        y_hist.invert_xaxis()
```



Text and Annotation

```
In[1]: %matplotlib inline
    import matplotlib.pyplotas plt
    import matplotlib as mpl
    plt.style.use('seaborn-whitegrid')
    import numpyas np
    import pandas as pd
In[2]:
    births = pd.read_csv('births.csv')
    quartiles = np.percentile(births['births'], [25, 50, 75])
    mu, sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])
    births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
    #5 standard deviations from the mean
    births['day'] = births['day'].astype(int)
    births.index= pd.to_datetime(10000 * births.year+100 * births.month+births.day,
    format='%Y%m%d') # convert to date like object to date time objects
    births_by_date= births.pivot_table('births',[births.index.month, births.index.day])
    births_by_date.index= [pd.datetime(2012, month, day)
    for (month, day) in births_by_date.index]
    In[3]: fig, ax= plt.subplots(figsize=(12, 4))
    births_by_date.plot(ax=ax);
```

AMSCE 101




```
In[4]: fig, ax= plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax)
       # Add labels to the plot
       style = dict(size=10, color='gray')
       ax.text('2012-1-1', 3950, "New Year's Day", **style)
       ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
       ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
       ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
       ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
       ax.text('2012-12-25', 3850, "Christmas ", ha='right', **style)
       # Label the axes
       ax.set(title='USA births by day of year (1969-1988)',ylabel='average daily births')
       # Format the x axis with centered month labels
       ax.xaxis.set_major_locator(mpl.dates.MonthLocator())#locate major ticks at the
       beginning of each month
       ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
       ax.xaxis.set_major_formatter(plt.NullFormatter()) #set major tick labels of the x-axis
       to null
       ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```

MSCE-101

Transforms and Text Position

- ax.transData

Transform associated with data coordinates

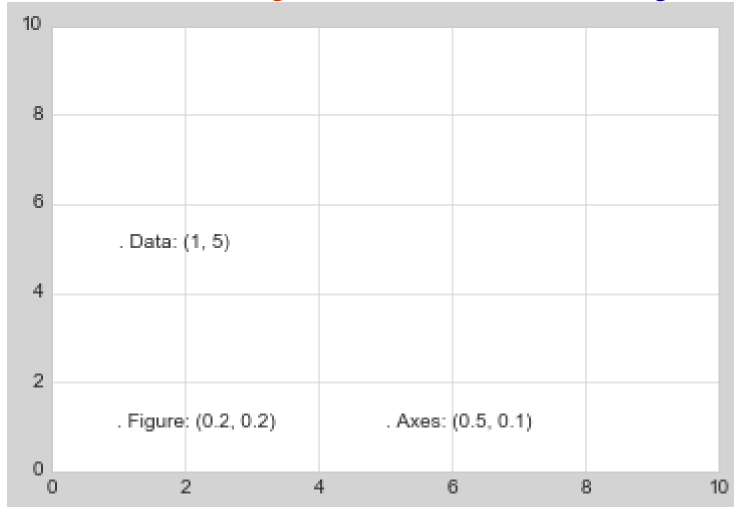
- ax.transAxes

Transform associated with the axes (in units of axes dimensions)

- fig.transFigure

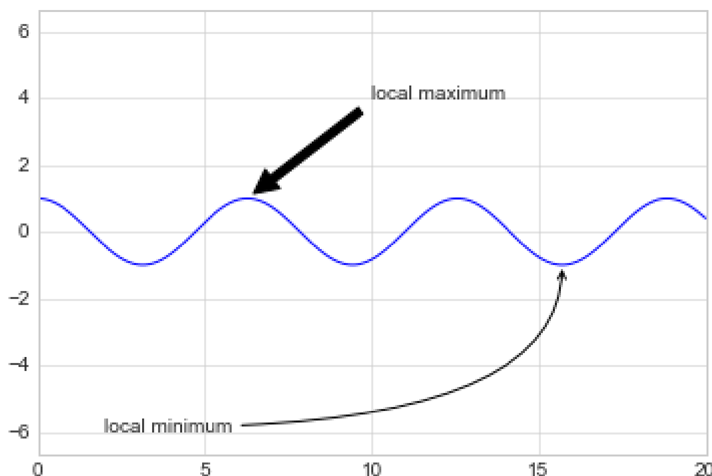
Transform associated with the figure (in units of figure dimensions)

```
In[5]: fig, ax= plt.subplots(facecolor='lightgray')
ax.axis([0, 10, 0, 10])
# transform=ax.transData is the default, but we'll specify it anyway
ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)        # look at axis 5 and 1
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure); # figure length, width
```



Arrows and Annotation

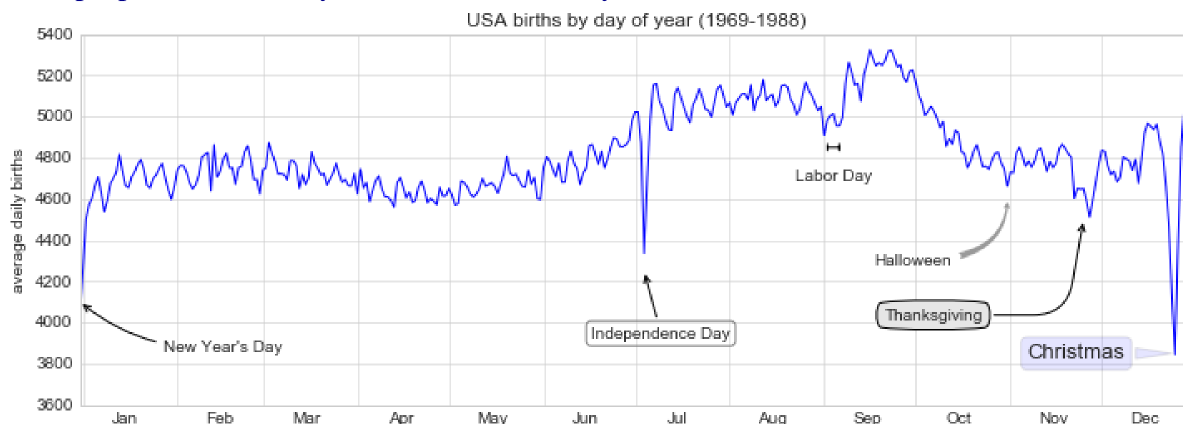
```
In[7]: %matplotlib inline
       fig, ax= plt.subplots()
       x = np.linspace(0, 20, 1000)
       ax.plot(x, np.cos(x))
       ax.axis('equal')
       ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),
       arrowprops=dict(facecolor='black', shrink=0.05))
       ax.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
       arrowprops=dict(arrowstyle="->",
       connectionstyle="angle3,angleA=0,angleB=-90"));
```




```
In[8]:
fig, ax= plt.subplots(figsize=(12, 4))
```

AMSCE-1101

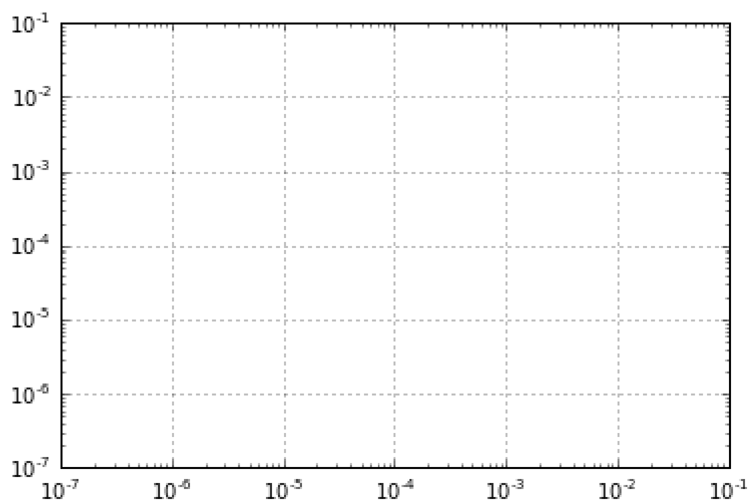
```
births_by_date.plot(ax=ax)
# Add labels to the plot
ax.annotate("New Year's Day", xy=('2012-1-1', 4100), xycoords='data',
xytext=(50, -30), textcoords='offset points',   #xytext position of text
arrowprops=dict(arrowstyle="->",connectionstyle="arc3,rad=-0.2"))
```



Customizing Ticks

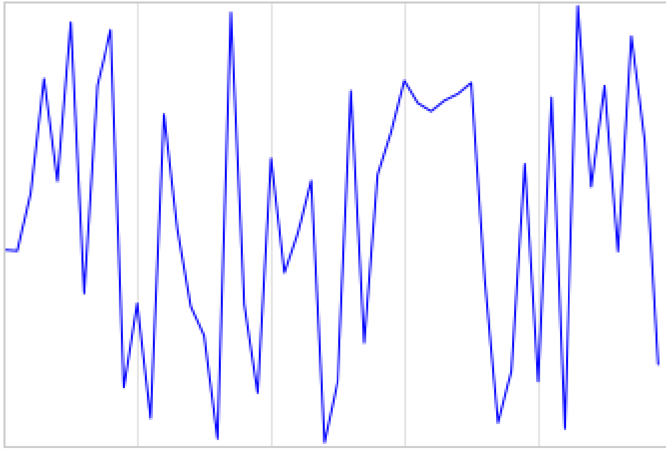
Major and Minor Ticks

```
In[1]: %matplotlib inline
       import matplotlib.pyplotas plt
       plt.style.use('seaborn-whitegrid')
       import numpyas np
In[2]: ax= plt.axes(xscale='log', yscale='log')
```



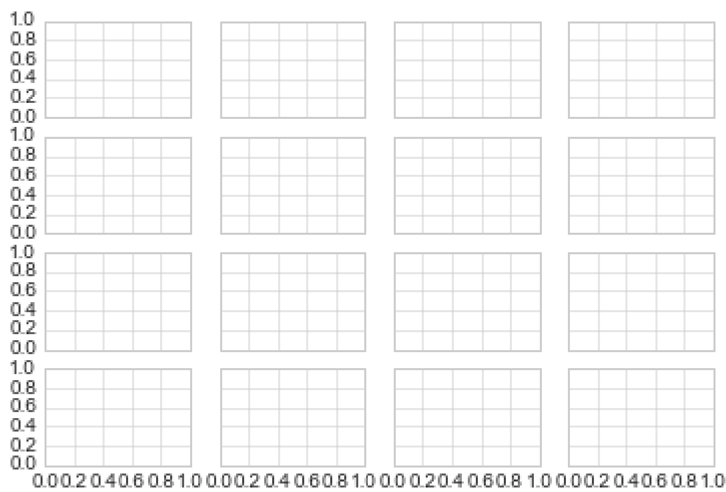
Hiding Ticks or Labels

```
In[5]: ax= plt.axes()
ax.plot(np.random.rand(50))
ax.yaxis.set_major_locator(plt.NullLocator()) #supresses y axis
ax.xaxis.set_major_formatter(plt.NullFormatter())
```



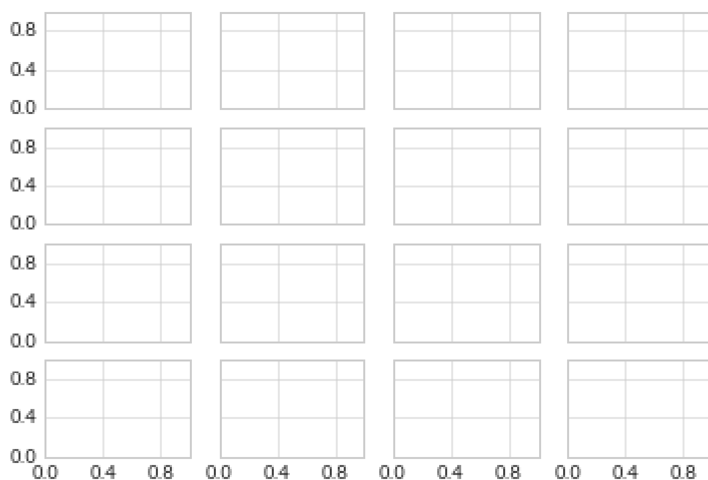
Reducing or Increasing the Number of Ticks

```
In[7]: fig, ax= plt.subplots(4, 4, sharex=True, sharey=True)
```



The above figure has crowded labels

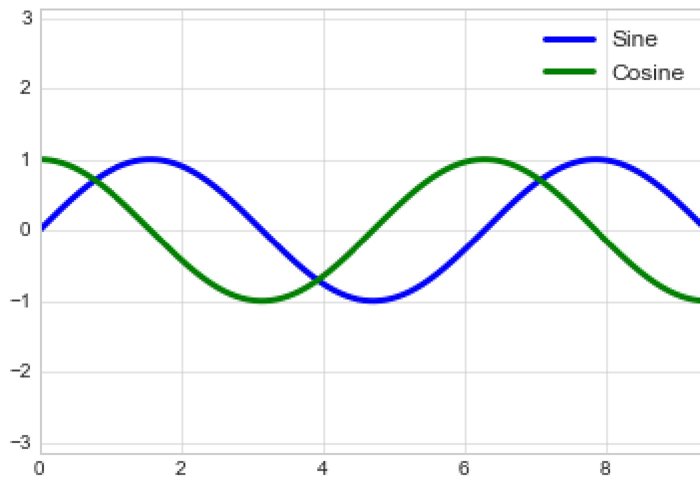
```
In[8]: # For every axis, set the x and y major locator
for axiin ax.flat:
axi.xaxis.set_major_locator(plt.MaxNLocator(3))     #set the number of tics
axi.yaxis.set_major_locator(plt.MaxNLocator(3))
fig
```



AMC 101

Fancy Tick Formats

```
In[9]: # Plot a sine and cosine curve
        fig, ax= plt.subplots()
        x = np.linspace(0, 3 * np.pi, 1000)
        ax.plot(x, np.sin(x), lw=3, label='Sine') #lw line width
        ax.plot(x, np.cos(x), lw=3, label='Cosine')
        # Set up grid, legend, and limits
        ax.grid(True)
        ax.legend(frameon=False)
        ax.axis('equal')
        ax.set_xlim(0, 3 * np.pi);
```



AMSCE 101

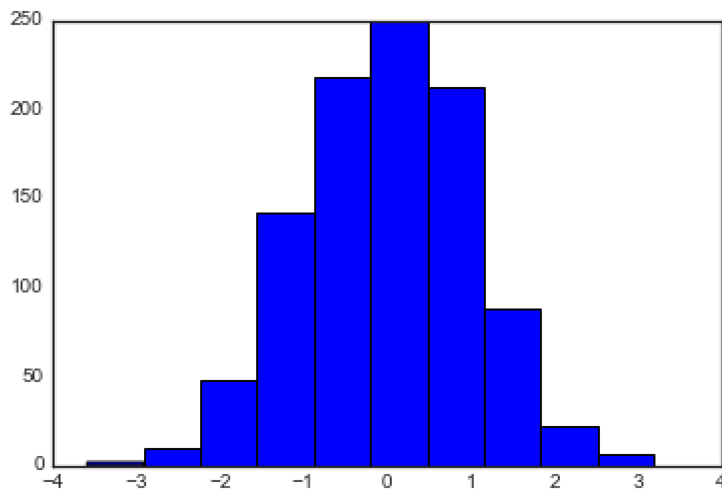
Customizing Matplotlib: Configurations and Stylesheets

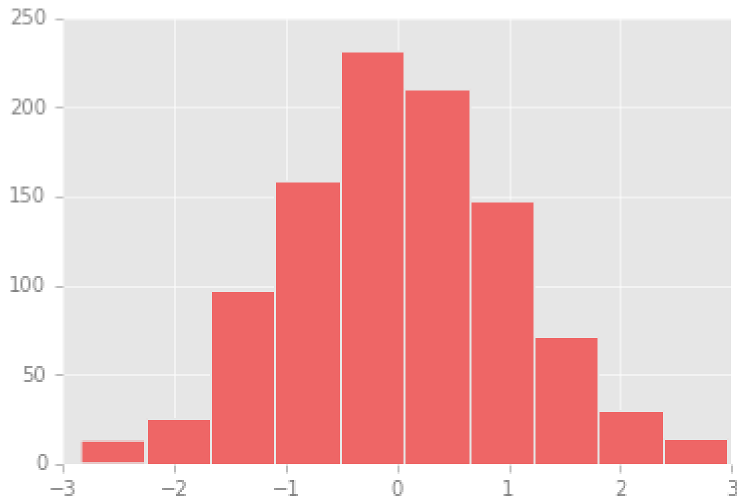
Plot Customization by Hand

First image shows a normal histogram to improve its quality

We do

```
In[3]: # use a gray background
ax= plt.axes(axisbg='#E6E6E6')       # light shade of grey hexadecimal color code
ax.set_axisbelow(True)         #Ticks and gridlines are below all Artists.
# draw solid white grid lines
plt.grid(color='w', linestyle='solid')
# hide axis spines
for spine in ax.spines.values():        #disable border lines
spine.set_visible(False)
# hide top and right ticks
ax.xaxis.tick_bottom()          # hide ticks on top and botom "-"
ax.yaxis.tick_left()
# lighten ticks and labels
ax.tick_params(colors='gray', direction='out')          # tick colour and direction
for tick in ax.get_xticklabels():
tick.set_color('gray')
for tick in ax.get_yticklabels():
tick.set_color('gray')
# control face and edge color of histogram
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');  #bar edge colour and bar colour
```





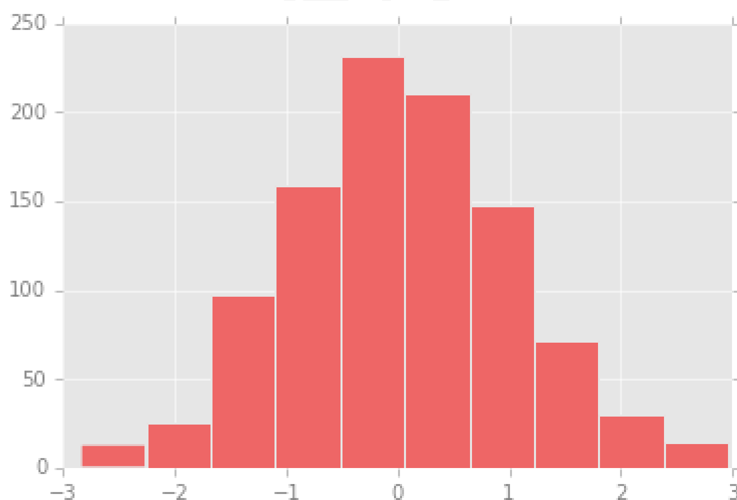
Since, this is hard to do all the modifications each time its best to change the defaults

Changing the Defaults: rcParams

Each time matplotlib loads it defines a runtime configuration (rc) containing default style for each plot. plt.rc.

```
In[4]: IPython_default= plt.rcParams.copy()
In[5]: from matplotlib import cycler
        colors= cycler('color',
        ['#EE6666', '#3388BB', '#9988DD',
        '#EECC55', '#88BB44', '#FFBBBB'])
        plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
        axisbelow=True, grid=True, prop_cycle=colors)
        plt.rc('grid', color='w', linestyle='solid')
        plt.rc('xtick', direction='out', color='gray')
        plt.rc('ytick', direction='out', color='gray')
        plt.rc('patch', edgecolor='#E6E6E6')
        plt.rc('lines', linewidth=2)
In[6]: plt.hist(x);
```

MSCE-1101




```
In[7]: for iin range(4):
plt.plot(np.random.rand(10))
```

Stylesheets

In[8]: plt.style.available[:5] #names of the first five available Matplotlib styles

```
Out[8]: ['fivethirtyeight',
'seaborn-pastel',
'seaborn-whitegrid',
'ggplot',
'grayscale']
```

The basic way to switch to a stylesheet is to call:

```
plt.style.use('stylename')
```

this will change the style for the rest of the session

```
with plt.style.context('stylename'):
make_a_plot()
```

Let's create a function that will make two basic types of plot:

```
In[9]: def hist_and_lines():
    np.random.seed(0)
    fig, ax= plt.subplots(1, 2, figsize=(11, 4))
    ax[0].hist(np.random.randn(1000))
    for iin range(3):
    ax[1].plot(np.random.rand(10))
    ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

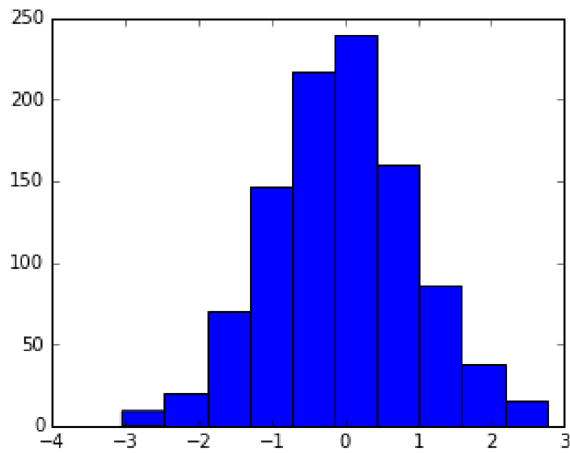
Default style

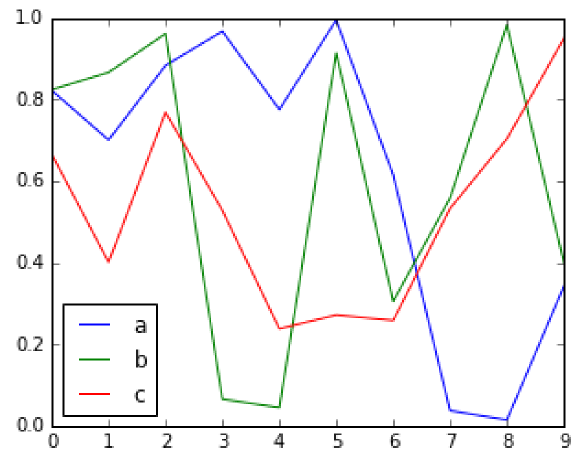
```
In[10]: # reset rcParams
    plt.rcParams.update(IPython_default);
```

Now let's see how it looks (Figure 4-85):

```
In[11]: hist_and_lines()
```

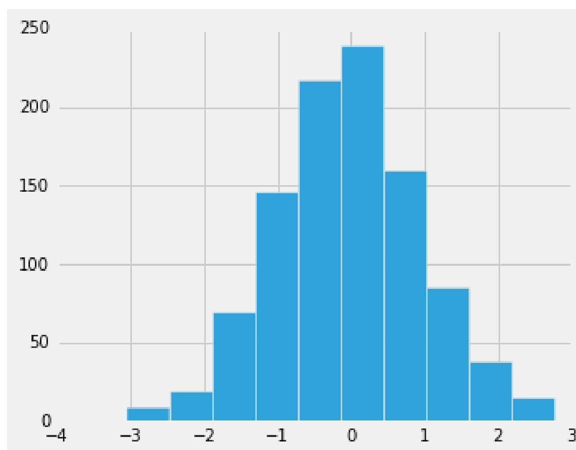
AMSCE-1101

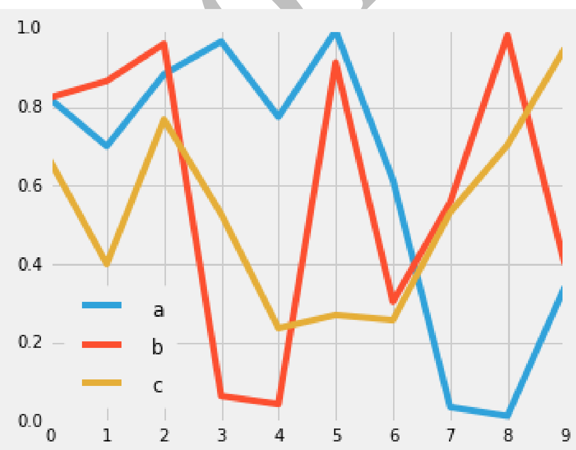




FiveThirtyEight style

```
In[12]: with plt.style.context('fivethirtyeight'):
        hist_and_lines()
```



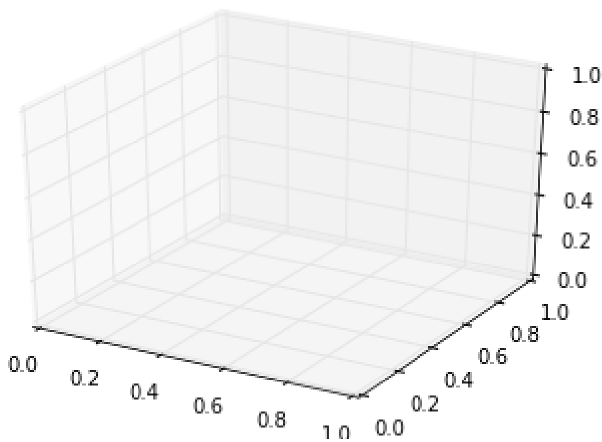


Similarly we have ggplot, Bayesian Methods for Hackers style, Dark background, Grayscale, Seaborn style

Three-Dimensional Plotting in Matplotlib

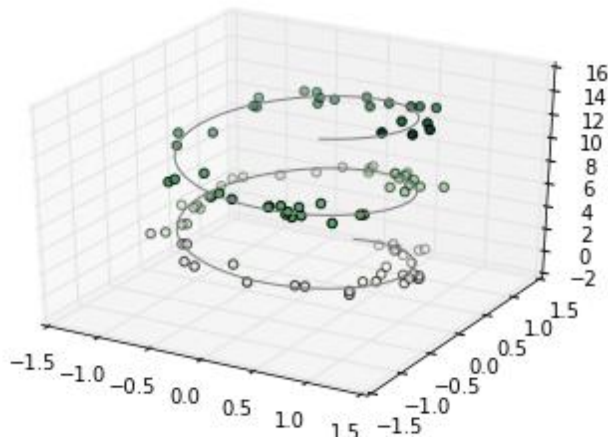
```
In[1]: from mpl_toolkitsimport mplot3d
```

```
In[2]: %matplotlib inline
import numpyas np
import matplotlib.pyplotas plt
In[3]: fig = plt.figure()
ax= plt.axes(projection='3d')
```



Three-Dimensional Points and Lines

```
In[4]: ax= plt.axes(projection='3d')
         # Data for a three-dimensional line
         zline= np.linspace(0, 15, 1000)
         xline= np.sin(zline)
         yline= np.cos(zline)
         ax.plot3D(xline, yline, zline, 'gray')
         # Data for three-dimensional scattered points
         zdata= 15 * np.random.random(100)
         xdata= np.sin(zdata) + 0.1 * np.random.randn(100)
         ydata= np.cos(zdata) + 0.1 * np.random.randn(100)
         ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens'); #scatter points
```



Geographic Data with Basemap

```
$ conda install basemap
```

```
In[1]: %matplotlib inline
import numpyas np
import matplotlib.pyplotas plt
         from mpl_toolkits.basemapimport Basemap
```

```
In[2]: plt.figure(figsize=(8, 8))
```

```
m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=-100)
        m.bluemarble(scale=0.5);
```

```
In[3]: fig = plt.figure(figsize=(8, 8))
        m = Basemap(projection='lcc', resolution=None,width=8E6, height=8E6,lat_0=45,
        lon_0=-100,)
        m.etopo(scale=0.5, alpha=0.5)        #satellite image
        # Map (long, lat) to (x, y) for plotting
        x, y = m(-122.3, 47.6) #lat and long
        plt.plot(x, y, 'ok', markersize=5)
        plt.text(x, y, ' Seattle', fontsize=12);
```

Drawing a Map Background

• Physical boundaries and bodies of water
drawcoastlines()
Draw continental coast lines
drawlsmask()
Draw a mask between the land and sea, for use with projecting images on one or the other

drawmapboundary()
Draw the map boundary, including the fill color for oceans
drawrivers()
Draw rivers on the map
fillcontinents()
Fill the continents with a given color; optionally fill lakes with another color
• Political boundaries
drawcountries()
Draw country boundaries
drawstates()
Draw US state boundaries
drawcounties()
Draw US county boundaries
• Map features
drawgreatcircle()
Draw a great circle between two points

AMSCI

drawparallels()
Draw lines of constant latitude
drawmeridians()
Draw lines of constant longitude
drawmapscale()
Draw a linear scale on the map
• Whole-globe images
bluemarble()
Project NASA's blue marble image onto the map
shadedrelief()
Project a shaded relief image onto the map
etopo()
Draw an etopo relief image onto the map
warpimage()
Project a user-provided image onto the map

Plotting Data on Maps

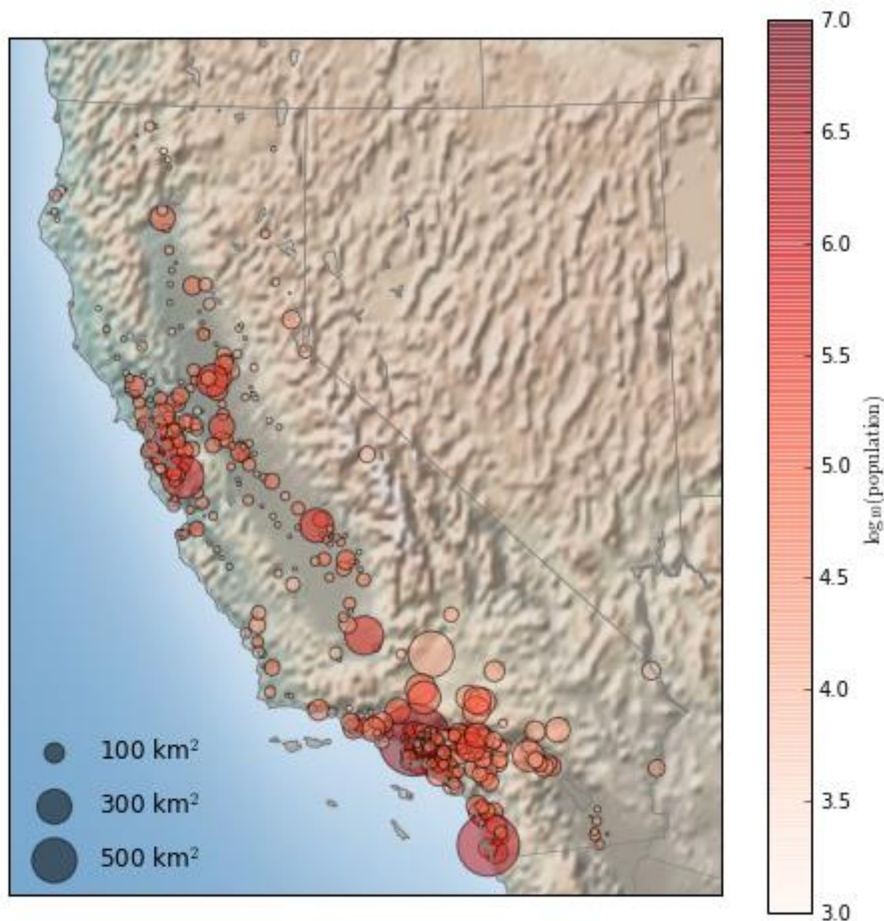
contour()/contourf()
Draw contour lines or filled contours
imshow()
Draw an image
pcolor()/pcolormesh()
Draw a pseudocolor plot for irregular/regular meshes
plot()
Draw lines and/or markers
scatter()
Draw points with markers
quiver()
Draw vectors
barbs()
Draw wind barbs
drawgreatcircle()
Draw a great circle

Example: California Cities

```
In[10]: import pandas as pd
        cities = pd.read_csv('data/california_cities.csv')
        # Extract the data we're interested in
        lat= cities['latd'].values
        lon= cities['longd'].values
        population = cities['population_total'].values
        area = cities['area_total_km2'].values

In[11]: # 1. Draw the map background
        fig = plt.figure(figsize=(8, 8))
        m = Basemap(projection='lcc', resolution='h', #map projection, resolution high
        lat_0=37.5, lon_0=-119,
        width=1E6, height=1.2E6)
        m.shadedrelief()        #draw shaded satellite image
```

AMSCE-1101

```
m.drawcoastlines(color='gray')
m.drawcountries(color='gray')
m.drawstates(color='gray')
# 2. scatter city data, with color reflecting population
# and size reflecting area
m.scatter(lon, lat, latlon=True,c=np.log10(population), s=area,cmap='Reds',
alpha=0.5)
# 3. create colorbar and legend
plt.colorbar(label=r'$\log_{10}({\rm population})$')
plt.clim(3, 7)   #Set the color limits of the current image.
# make legend with dummy points
for a in [100, 300, 500]:
plt.scatter([], [], c='k', alpha=0.5, s=a,
label=str(a) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False,
labelspacing=1, loc='lower left');
```

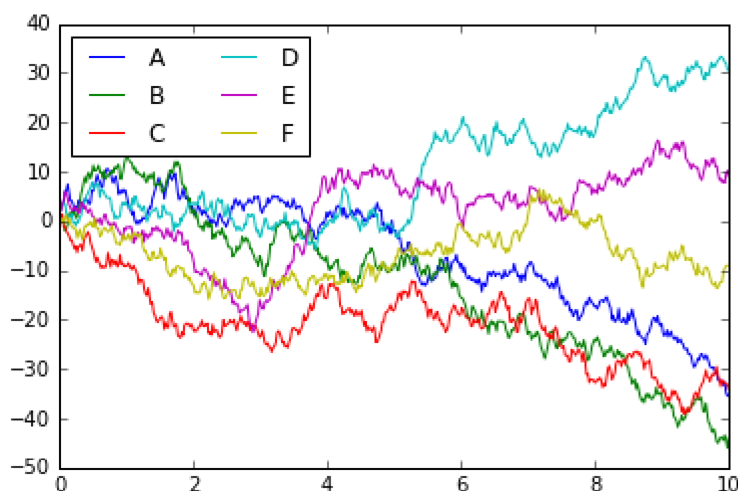


Visualization with Seaborn

Example of matplot lib classic plot.

```
In[1]: import matplotlib.pyplotas plt
       plt.style.use('classic')
       %matplotlib inline
```

```
        import numpyas np
        import pandas as pd
In[2]: # Create some data
        rng= np.random.RandomState(0)
        x = np.linspace(0, 10, 500)
        y = np.cumsum(rng.randn(500, 6), 0)#cumulative sum of elements (partial sum of
        sequence)
In[3]: # Plot the data with Matplotlib defaults
        plt.plot(x, y)
        plt.legend('ABCDEF', ncol=2, loc='upper left');
```



Seaborn image plot

```
In[4]: import seaborn as sns
        sns.set() #Seaborn's default settings to your plots,
In[5]: # same plotting code as above!
        plt.plot(x, y)
        plt.legend('ABCDEF', ncol=2, loc='upper left');
```

AMSCOL