PIE Tech

POLLACHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE and Affiliated to Anna University)

sky is the limit

Department of Computer Science and Engineering

Regulation 2021

II Year – III Semester

CS3352 FOUNDATIONS OF DATA SCIENCE

UNIT I - INTRODUCTION

Data

In computing, data is information that has been translated into a form that is efficient for movement or processing

Data Science

Data science is an evolutionary extension of statistics capable of dealing with the massive amounts of data produced today. It adds methods from computer science to the repertoire of statistics.

Benefits and uses of data science

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.
- 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.
- Nongovernmental organizations (NGOs) use it to rase money and defend their causes.
- 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.

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

Let's explore all these interesting data types.

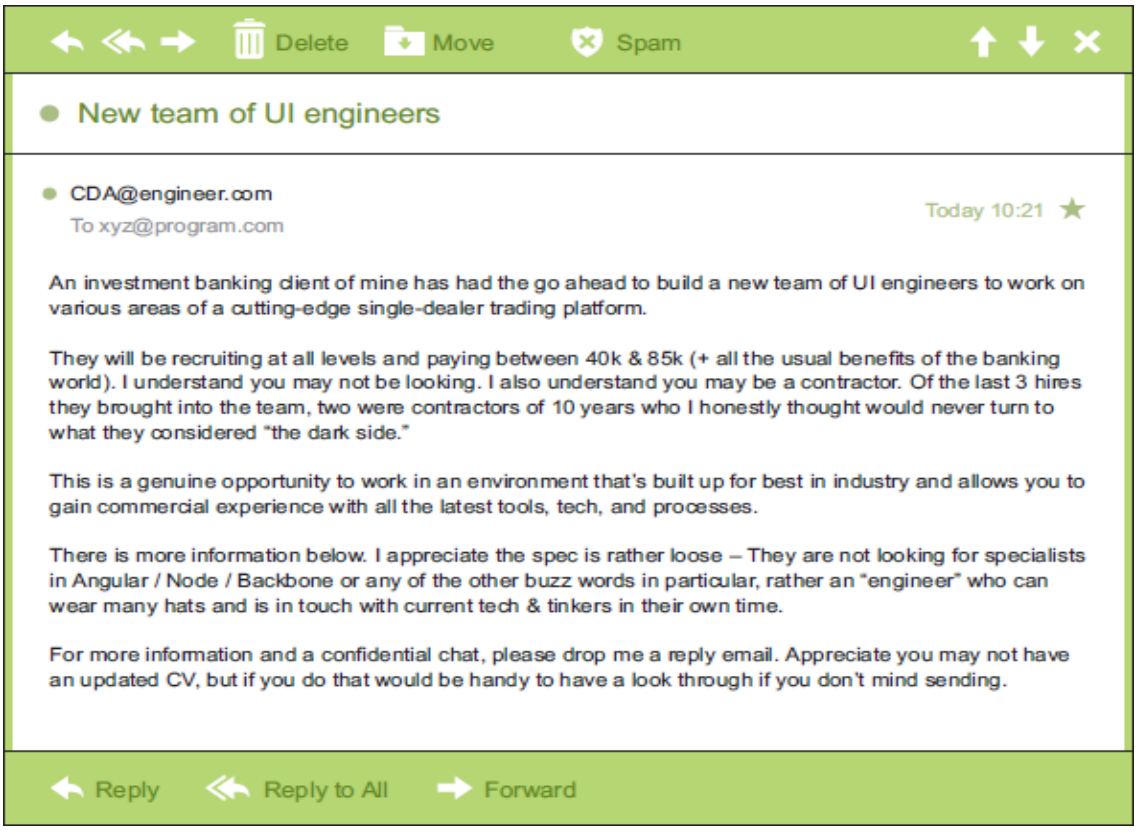
Structured data

- 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 files
- SQL, or Structured Query Language, is the preferred way to manage and query data that resides in databases.

	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%

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



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.

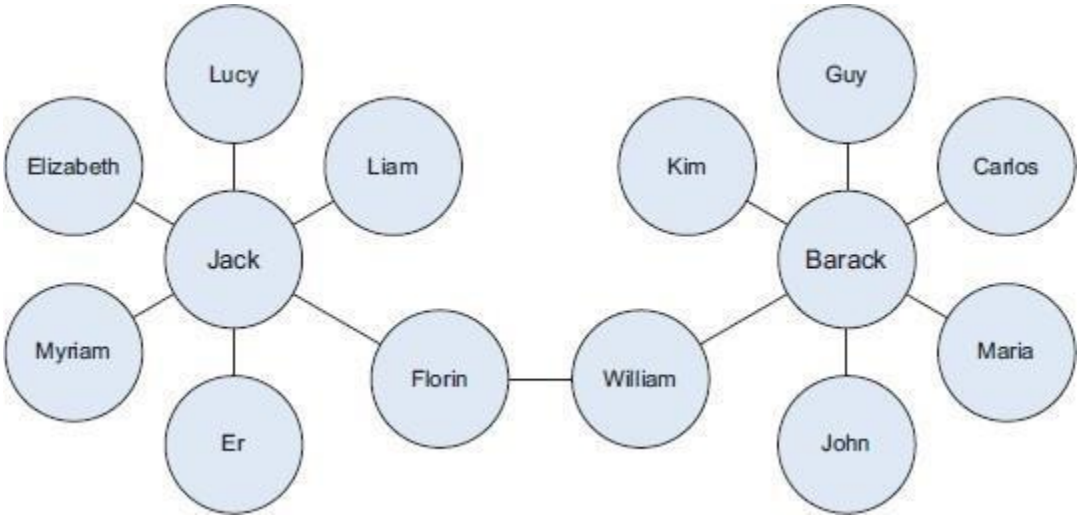
Machine-generated data

- 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.
- The analysis of machine data relies on highly scalable tools, due to its high volume and speed. Examples of machine data are web server logs, call detail records, network event logs, and telemetry.

```
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
```

Graph-based or network data

- "Graph data" can be a confusing term because any data can be shown in a graph
- 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.
- raph-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.



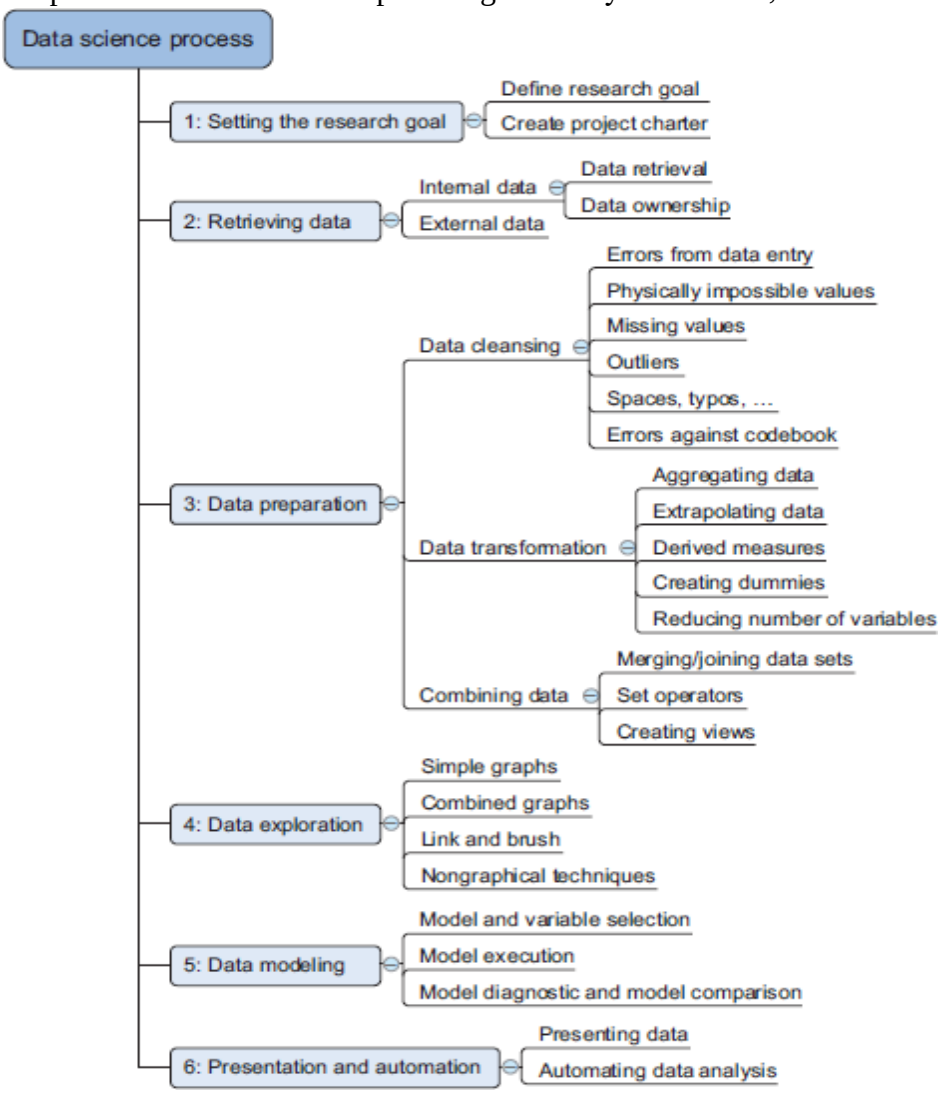
Streaming data

- The data flows into the system when an event happens instead of being loaded into a data store in a batch.
- Examples are the "What's trending" on Twitter, live sporting or music events, and the stock market.

Data Science Process

Overview of the data science process

he typical data science process consists of six steps through which you'll iterate, as shown in figure



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 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.

efining research goals

A project starts by understanding the *what*, the *why*, and the *how* of your project. 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.
- Continue asking questions and devising examples until you grasp the exact business expectations, identify how your project fits in the bigger picture, appreciate how your research is going to change the business, and understand how they'll use your results

Create 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

Retrieving data

- The next step in data science is to retrieve the required data. 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.
- More and more organizations are making even high-quality data freely available for public and commercial use.
- 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.

Start with data stored within the company (Internal data)

- 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.
- Data warehouses and data marts are home to preprocessed data, data lakes contain data in its natural or raw format.
- Finding data even within your own company can sometimes be a challenge. As companies grow, their data becomes scattered around many places. the data may be dispersed as people change positions and leave the company.
- Getting access to data is another difficult task. Organizations understand the value and sensitivity of data and often have policies in place so everyone has access to what they need and nothing more.
- These policies translate into physical and digital barriers called *Chinese walls*. These "walls" are mandatory and well-regulated for customer data in most countries.

External Data

- If data isn't available inside your organization, look outside your organizations. Companies provide data so that you, in turn, can enrich their services and ecosystem. Such is the case with Twitter, LinkedIn, and Facebook.
- More and more governments and organizations share their data for free with the world.
- 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

Data Preparation (Cleansing, Integrating, Transforming Data)

Your model needs the data in a specific format, so data transformation will always come into play. It's a good habit to correct data errors as early on in the process as possible. However, this isn't always possible in a realistic setting, so you'll need to take corrective actions in your program.

Cleansing data

Data cleansing is a sub process 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.

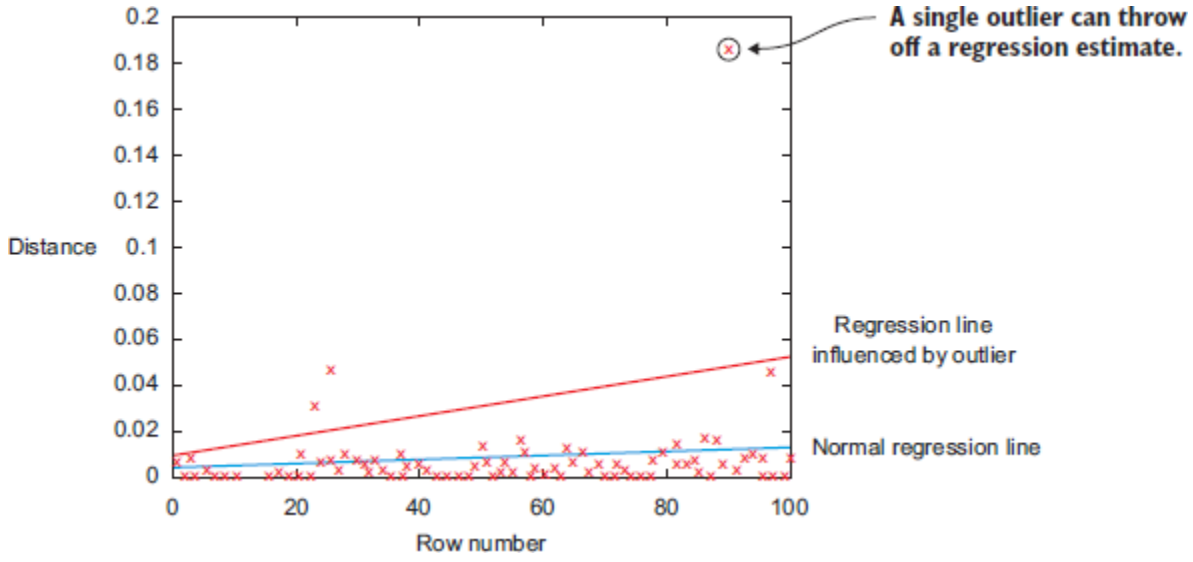
- The first type is the *interpretation error*, such as when you take the value in your data for granted, like saying that a person's age is greater than 300 years.
- The second type of error points to *inconsistencies* between data sources or against your company's standardized values.

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.

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

ometimes you'll use more advanced methods, such as simple modeling, to find and identify data errors; diagnostic plots can be especially insightful. For example, in figure we use a measure to identify data points that seem out of place. We do a regression to get acquainted with the data and detect the influence of individual observations on the regression line.



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 introduce an error into the chain.

- Data collected by machines or computers isn't free from errors. Errors can arise from human sloppiness, whereas others are due to machine or hardware failure.
- Detecting data errors when the variables you study don't have many classes can be done by tabulating the data with counts.
- When you have a variable that can take only two values: "Good" and "Bad", you can create a frequency table and see if those are truly the only two values present. In table the values "Godo" and "Bade" point out something went wrong in at least 16 cases.

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-thenelse rules:

```
f 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 whitespace cause the miss match in the string such as "FR " – "FR", dropping the observations that couldn't be matched.
- If you know to watch out for them, fixing redundant whitespaces is luckily easy enough in most programming languages. They all provide string functions that will remove the leading and trailing whitespaces. For instance, in Python you can use the 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".

In this case 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

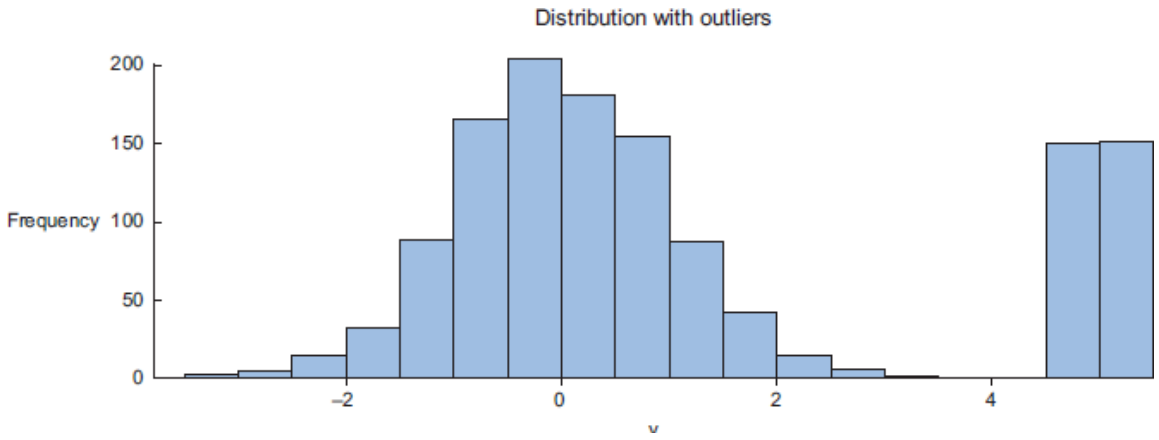
Here you check the value against physically or theoretically impossible values such as people taller than 3 meters or someone with an age of 299 years. Sanity checks can be directly expressed with rules:

```
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.

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.



Dealing with Missing Values

issing values aren't necessarily wrong, but you still need to handle them separately; certain modeling techniques can't handle missing values. They might be an indicator that something went wrong in your data collection or that an error happened in the ETL process. Common techniques data scientists use are listed in table

Integrating data

Your data comes from several different places, and in this substep we focus on integrating these different sources. Data varies in size, type, and structure, ranging from databases and Excel files to text documents.

The Different Ways of Combining Data

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

- Joining
- Appending or stacking

Joining Tables

- Joining tables allows you to combine the information of one observation found in one table with the information that you find in another table. The focus is on enriching a single observation.
- Let's say that the first table contains information about the purchases of a customer and the other table contains information about the region where your customer lives.
- Joining the tables allows you to combine the information so that you can use it for your model, as shown in figure.

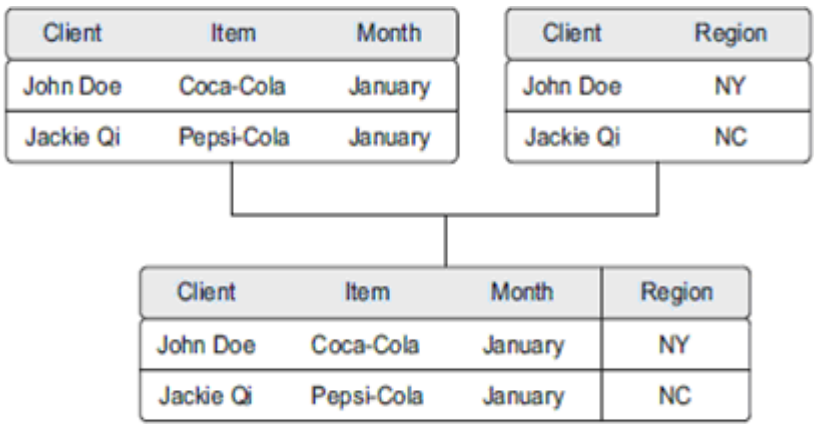


Figure. Joining two tables on the item and region key

To join tables, you use variables that represent the same object in both tables, such as a date, a country name, or a Social Security number. These common fields are known as keys. When these keys also uniquely define the records in the table they are called *primary keys*.
The number of resulting rows in the output table depends on the exact join type that you use

Appending Tables **Appending or stacking tables is effectively adding observations from one table to another table.**

- One table contains the observations from the month January and the second table contains observations from the month February. The result of appending these tables is a larger one with the observations from January as well as February.

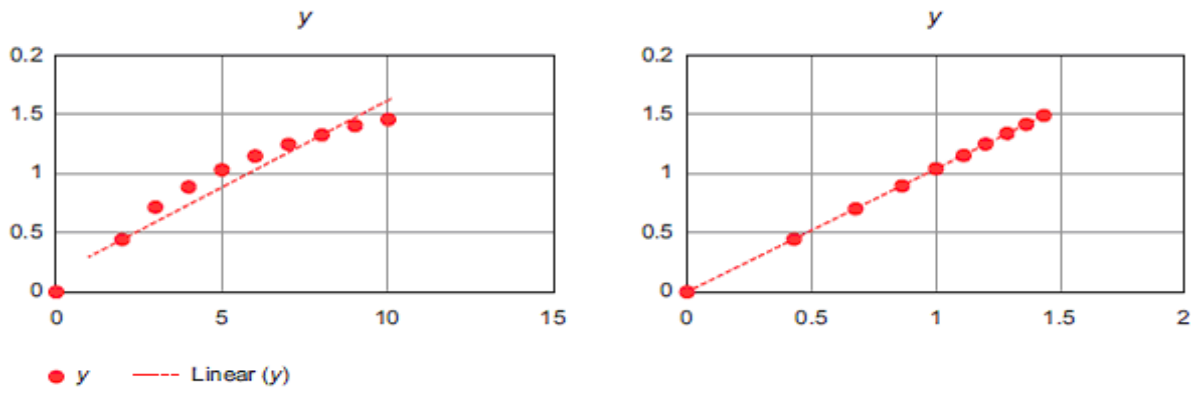
Figure. Appending data from tables is a common operation but requires an equal structure in the tables begin appended,

Transforming data

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

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. Transforming the input variables greatly simplifies the estimation problem. Other times you might want to combine two variables into a new variable.

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



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

- 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.
- Data scientists use special methods to reduce the number of variables but retain the maximum amount of data.

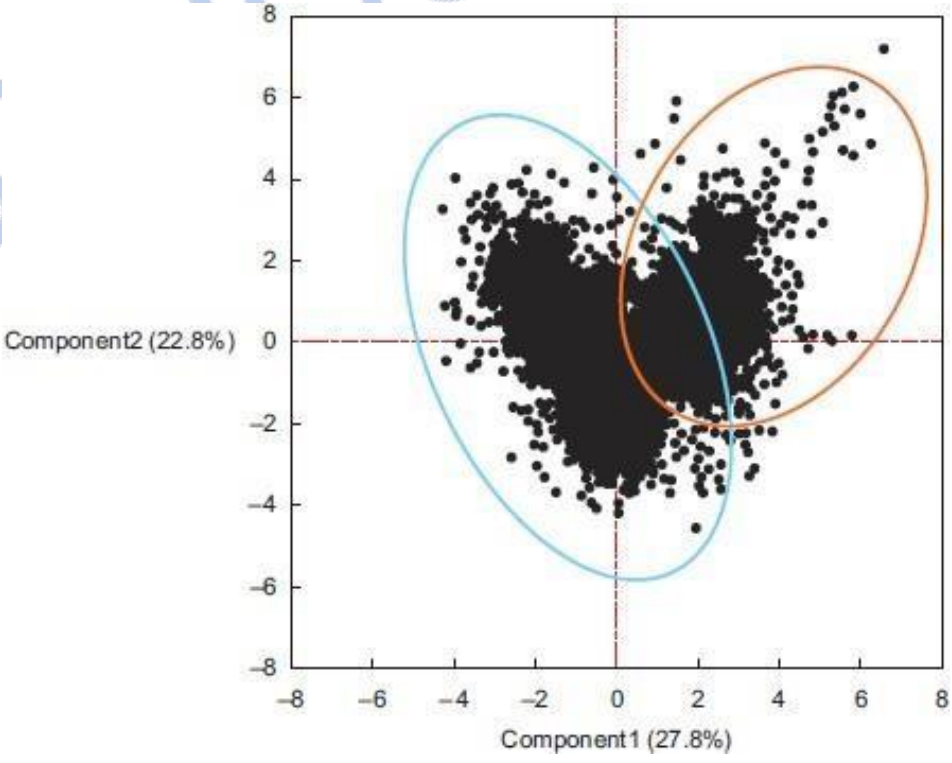


Figure shows how reducing the number of variables makes it easier to understand the key values. It also shows how two variables account for 50.6% of the variation within the data set (component1 = 27.8% + component2 = 22.8%). These variables, called "component1" and "component2," are both combinations of the original variables. They're the *principal components* of the underlying data structure

Turning Variables into Dummies

- *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.
- In this case you'll make separate columns for the classes stored in one variable and indicate it with 1 if the class is present and 0 otherwise.
- An example is turning one column named Weekdays into the columns Monday through Sunday. You use an indicator to show if the observation was on a Monday; you put 1 on Monday and 0 elsewhere.
- Turning variables into dummies is a technique that's used in modeling and is popular with, but not exclusive to, economists.

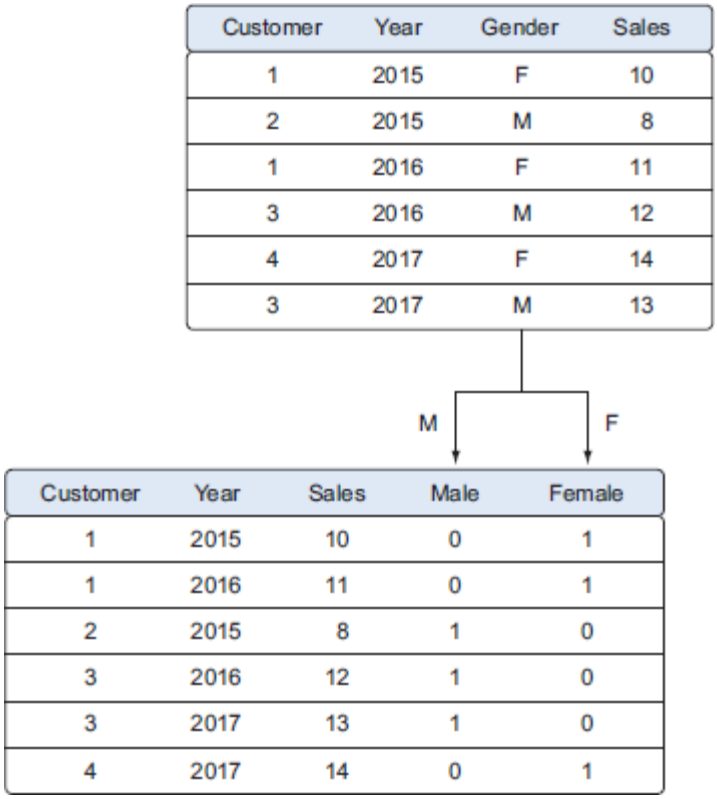
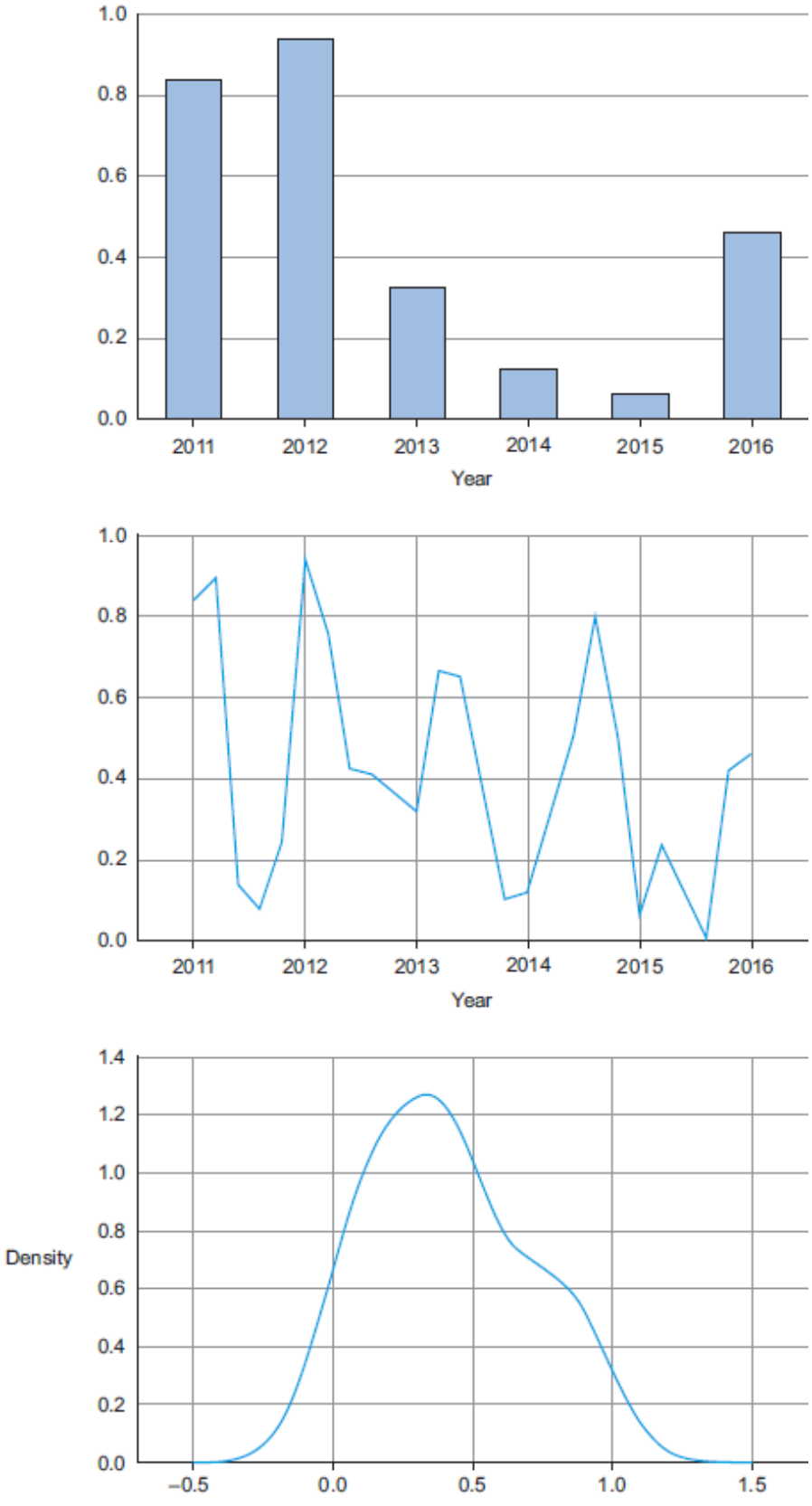


Figure. Turning variables into dummies is a data transformation that breaks a variable that has multiple classes into multiple variables, each having only two possible values: 0 or 1

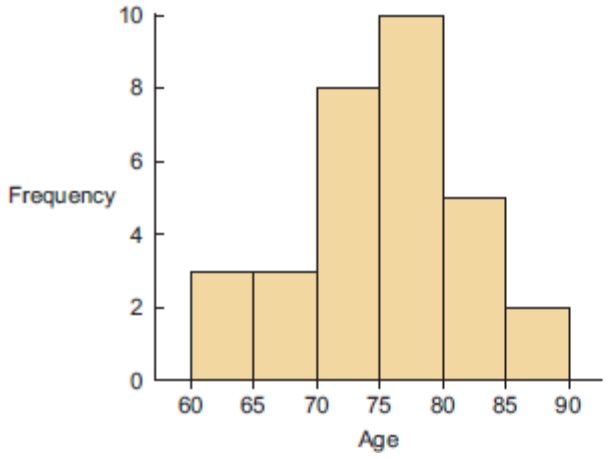
Exploratory data analysis

During exploratory data analysis you take a deep dive into the data (see figure below). 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.
The goal isn't to cleanse the data, but it's common that you'll still discover anomalies you missed before, forcing you to take a step back and fix them.



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 or histograms, as shown in below figure , to more complex diagrams such as Sankey and network graphs.
- Sometimes it's useful to compose a composite graph from simple graphs to get even more insight into the data Other times the graphs can be animated or made interactive to make it easier and, let's admit it, way more fun



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

The techniques we described in this phase are mainly visual, but in practice they're certainly not limited to visualization techniques. Tabulation, clustering, and other modeling techniques can also be a part of exploratory analysis. Even building simple models can be a part of this step.

Build the models

- With clean data in place and a good understanding of the content, you're ready to build models with the goal of making better predictions, classifying objects, or gaining an understanding of the system that you're modeling.
- This phase is much more focused than the exploratory analysis step, because you know what you're looking for and what you want the outcome to be.

Building a model is an iterative process. The way you build your model depends on whether you go with classic statistics or the somewhat more recent machine learning school, and the type of technique you want to use. Either way, most models consist of the following main steps:

- Selection of a modeling technique and variables to enter in the model
- Execution of the model
- Diagnosis and model comparison

Model and variable selection

You'll need to select the variables you want to include in your model and a modeling technique. 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?

Model execution

- Once you've chosen a model you'll need to implement it in code.

- Most programming languages, such as Python, already have libraries such as StatsModels or Scikit-learn. These packages use several of the most popular techniques.
- Coding a model is a nontrivial task in most cases, so having these libraries available can speed up the process. As you can see in the following code, it's fairly easy to use linear regression with StatsModels or Scikit-learn
- Doing this yourself would require much more effort even for the simple techniques. The following listing shows the execution of a linear prediction model.

```
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.

Model diagnostics and model comparison

- You'll be building multiple models from which you then choose the best one based on multiple criteria. Working with a holdout sample helps you pick the best-performing model.
- A holdout sample is a part of the data you leave out of the model building so it can be used to evaluate the model afterward.
- The principle here is simple: the model should work on unseen data. You use only a fraction of your data to estimate the model and the other part, the holdout sample, is kept out of the equation.
- 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 we show the general idea on comparing models. The error measure used in the example is the mean square error.

Formula for mean square error.

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

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.

	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

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

bove figure compares the performance of two models to predict the order size from the price. The first model is *size = 3 * price* and the second model is *size = 10*.

- 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.
- 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*.

Presenting findings and building applications

- 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.
- 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 is the process of discovering actionable information from large sets of data. Data mining uses mathematical analysis to derive patterns and trends that exist in data. Typically, these patterns cannot be discovered by traditional data exploration because the relationships are too complex or because there is too much data.

These patterns and trends can be collected and defined as a *data mining model*. Mining models can be applied to specific scenarios, such as:

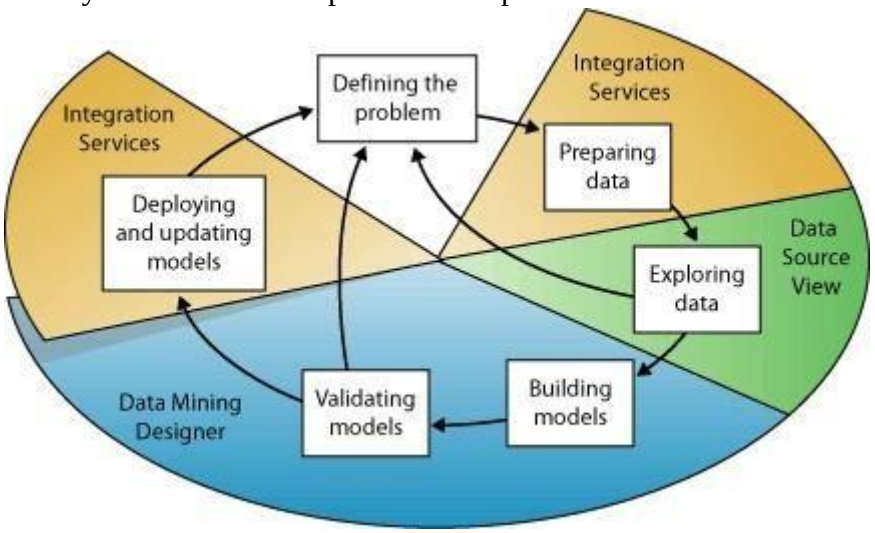
- **Forecasting**: Estimating sales, predicting server loads or server downtime

- **Risk and probability**: Choosing the best customers for targeted mailings, determining the probable break-even point for risk scenarios, assigning probabilities to diagnoses or other outcomes
- **Recommendations**: Determining which products are likely to be sold together, generating recommendations
- **Finding sequences**: Analyzing customer selections in a shopping cart, predicting next likely events
- **Grouping**: Separating customers or events into cluster of related items, analyzing and predicting affinities

Building a mining model is part of a larger process that includes everything from asking questions about the data and creating a model to answer those questions, to deploying the model into a working environment. This process can be defined by using the following six basic steps:

1. Defining the Problem
2. Preparing Data
3. Exploring Data
4. Building Models
5. Exploring and Validating Models
6. Deploying and Updating Models

The following diagram describes the relationships between each step in the process, and the technologies in Microsoft SQL Server that you can use to complete each step.



Defining the Problem

The first step in the data mining process is to clearly define the problem, and consider ways that data can be utilized to provide an answer to the problem.

This step includes analyzing business requirements, defining the scope of the problem, defining the metrics by which the model will be evaluated, and defining specific objectives for the data mining project. These tasks translate into questions such as the following:

- What are you looking for? What types of relationships are you trying to find?
- Does the problem you are trying to solve reflect the policies or processes of the business?
- Do you want to make predictions from the data mining model, or just look for interesting patterns and associations?
- Which outcome or attribute do you want to try to predict?

- What kind of data do you have and what kind of information is in each column? If there are multiple tables, how are the tables related? Do you need to perform any cleansing, aggregation, or processing to make the data usable?
- How is the data distributed? Is the data seasonal? Does the data accurately represent the processes of the business?

Preparing Data

- The second step in the data mining process is to consolidate and clean the data that was identified in the Defining the Problem step.
- Data can be scattered across a company and stored in different formats, or may contain inconsistencies such as incorrect or missing entries.
- Data cleaning is not just about removing bad data or interpolating missing values, but about finding hidden correlations in the data, identifying sources of data that are the most accurate, and determining which columns are the most appropriate for use in analysis

Exploring Data

Exploration techniques include calculating the minimum and maximum values, calculating mean and standard deviations, and looking at the distribution of the data. For example, you might determine by reviewing the maximum, minimum, and mean values that the data is not representative of your customers or business processes, and that you therefore must obtain more balanced data or review the assumptions that are the basis for your expectations. Standard deviations and other distribution values can provide useful information about the stability and accuracy of the results.

Building Models

The mining structure is linked to the source of data, but does not actually contain any data until you process it. When you process the mining structure, SQL Server Analysis Services generates aggregates and other statistical information that can be used for analysis. This information can be used by any mining model that is based on the structure.

Exploring and Validating Models

Before you deploy a model into a production environment, you will want to test how well the model performs. Also, when you build a model, you typically create multiple models with different configurations and test all models to see which yields the best results for your problem and your data.

Deploying and Updating Models

After the mining models exist in a production environment, you can perform many tasks, depending on your needs. The following are some of the tasks you can perform:

- Use the models to create predictions, which you can then use to make business decisions.
- Create content queries to retrieve statistics, rules, or formulas from the model.
- Embed data mining functionality directly into an application. You can include Analysis Management Objects (AMO), which contains a set of objects that your application can use to create, alter, process, and delete mining structures and mining models.
- Use Integration Services to create a package in which a mining model is used to intelligently separate incoming data into multiple tables.
- Create a report that lets users directly query against an existing mining model
- Update the models after review and analysis. Any update requires that you reprocess the models.
- Update the models dynamically, as more data comes into the organization, and making constant changes to improve the effectiveness of the solution should be part of the deployment strategy.

Data warehousing

Data warehousing is the process of constructing and using a data warehouse. A data warehouse is constructed by integrating data from multiple heterogeneous sources that support analytical reporting, structured and/or ad hoc queries, and decision making. Data warehousing involves data cleaning, data integration, and data consolidations.

Characteristics of data warehouse

The main characteristics of a data warehouse are as follows:

- Subject-Oriented

  A data warehouse is subject-oriented since it provides topic-wise information rather than the overall processes of a business. Such subjects may be sales, promotion, inventory, etc
- Integrated

  A data warehouse is developed by integrating data from varied sources into a consistent format. The data must be stored in the warehouse in a consistent and universally acceptable manner in terms of naming, format, and coding. This facilitates effective data analysis.
- Non-Volatile

  Data once entered into a data warehouse must remain unchanged. All data is read-only. Previous data is not erased when current data is entered. This helps you to analyze what has happened and when.
- Time-Variant

  The data stored in a data warehouse is documented with an element of time, either explicitly or implicitly. An example of time variance in Data Warehouse is exhibited in the Primary Key, which must have an element of time like the day, week, or month.

Database vs. Data Warehouse

Although a data warehouse and a traditional database share some similarities, they need not be the same idea. The main difference is that in a database, data is collected for multiple transactional purposes. However, in a data warehouse, data is collected on an extensive scale to perform analytics. Databases provide real-time data, while warehouses store data to be accessed for big analytical queries.

Data Warehouse Architecture

Usually, data warehouse architecture comprises a three-tier structure.

Bottom Tier

The bottom tier or data warehouse server usually represents a relational database system. Back-end tools are used to cleanse, transform and feed data into this layer.

Middle Tier

The middle tier represents an OLAP server that can be implemented in two ways.

The ROLAP or Relational OLAP model is an extended relational database management system that maps multidimensional data process to standard relational process.

The MOLAP or multidimensional OLAP directly acts on multidimensional data and operations.

Top Tier

This is the front-end client interface that gets data out from the data warehouse. It holds various tools like query tools, analysis tools, reporting tools, and data mining tools.

How Data Warehouse Works

Data Warehousing integrates data and information collected from various sources into one comprehensive database. For example, a data warehouse might combine customer information from an organization's point-of-sale systems, its mailing lists, website, and comment cards. It might also incorporate confidential

information about employees, salary information, etc. Businesses use such components of data warehouse to analyze customers.

Data mining is one of the features of a data warehouse that involves looking for meaningful data patterns in vast volumes of data and devising innovative strategies for increased sales and profits.

Types of Data Warehouse
There are three main types of data warehouse.

Enterprise Data Warehouse (EDW)
This type of warehouse serves as a key or central database that facilitates decision-support services throughout the enterprise. The advantage to this type of warehouse is that it provides access to cross-organizational information, offers a unified approach to data representation, and allows running complex queries.

Operational Data Store (ODS)
This type of data warehouse refreshes in real-time. It is often preferred for routine activities like storing employee records. It is required when data warehouse systems do not support reporting needs of the business.

Data Mart
A data mart is a subset of a data warehouse built to maintain a particular department, region, or business unit. Every department of a business has a central repository or data mart to store data. The data from the data mart is stored in the ODS periodically. The ODS then sends the data to the EDW, where it is stored and used.

Summary

In this chapter you learned the data science process consists of six steps:

- *Setting the research goal*—Defining the what, the why, and the how of your project in a project charter.
- *Retrieving data*—Finding and getting access to data needed in your project. This data is either found within the company or retrieved from a third party.
- *Data preparation*—Checking and remediating data errors, enriching the data with data from other data sources, and transforming it into a suitable format for your models.
- *Data exploration*—Diving deeper into your data using descriptive statistics and visual techniques.
- *Data modeling*—Using machine learning and statistical techniques to achieve your project goal.
- *Presentation and automation*—Presenting your results to the stakeholders and industrializing your analysis process for repetitive reuse and integration with other tools.

Unit – II
DESCRIBING DATA

Types of Data - Types of Variables -Describing Data with Tables and Graphs –Describing Data with Averages - Describing Variability - Normal Distributions and Standard (z) Scores

THREE TYPES OF 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.
- **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. To determine the type of data, focus on a single observation in any collection of observations

TYPES OF VARIABLES

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

- The weights can be described not only as quantitative data but also as observations for a quantitative variable, since the various weights take on different numerical values.
- By the same token, the replies can be described as observations for a qualitative variable, since the replies to the Facebook profile question take on different values of either Yes or No.
- Given this perspective, any single observation can be described as a constant, since it takes on only one value.

Discrete and Continuous Variables

Quantitative variables can be further distinguished as **discrete or continuous.**

A **discrete variable** consists of isolated numbers separated by gaps.

Discrete variables can only assume specific values that you cannot subdivide. Typically, you count discrete values, and the results are integers.

Examples

- Counts- such as the number of children in a family. (1, 2, 3, etc., but never 1.5)
- These variables cannot have fractional or decimal values. You can have 20 or 21 cats, but not 20.5
- The number of heads in a sequence of coin tosses.
- The result of rolling a die.
- The number of patients in a hospital.
- The population of a country.

While discrete variables have no decimal places, the average of these values can be fractional. For example, families can have only a discrete number of children: 1, 2, 3, etc. However, the average number of children per family can be 2.2.

A **continuous variable** consists of numbers whose values, at least in theory, have no restrictions.

Continuous variables can assume any numeric value and can be meaningfully split into smaller parts. Consequently, they have valid fractional and decimal values. In fact, continuous variables have an infinite number of potential values between any two points. Generally, you measure them using a scale.

Examples of continuous variables include weight, height, length, time, and temperature.

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

Independent Variable

In an experiment, an **independent variable** *is the treatment manipulated by the investigator.*

- Independent variables (IVs) are the ones that you include in the model to explain or predict changes in the dependent variable.
- Independent indicates that they stand alone and other variables in the model do not influence them.
- Independent variables are also known as predictors, factors, treatment variables, explanatory variables, input variables, x-variables, and right-hand variables—because they appear on the right side of the equals sign in a regression equation.
- It is a variable that stands alone and isn't changed by the other variables you are trying to measure.

For example, someone's age might be an independent variable. Other factors (such as what they eat, how much they go to school, how much television they watch)

The impartial creation of distinct groups, which differ only in terms of the independent variable, has a most desirable consequence. Once the data have been collected, any difference between the groups can be interpreted as being *caused* by the independent variable.

Dependent Variable

When a variable is believed to have been influenced by the independent variable, it is called a **dependent variable.** In an experimental setting, the dependent variable is measured, counted, or recorded by the investigator.

- The dependent variable (DV) is what you want to use the model to explain or predict. The values of this variable depend on other variables.
- It's also known as the response variable, outcome variable, and left-hand variable. Graphs place dependent variables on the vertical, or Y, axis.
- a dependent variable is exactly what it sounds like. It is something that depends on other factors.

For example the blood sugar test depends on what food you ate, at which time you ate etc.
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

An uncontrolled variable that compromises the interpretation of a study is known as a **confounding variable.** Sometimes a confounding variable occurs because it's impossible to assign subjects randomly to different conditions.

Describing Data with Tables and Graphs

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.
- When observations are sorted into classes of single values, as in Table 2.1, the result is referred to as a frequency distribution for ungrouped data.
- The frequency distribution shown in Table 2.1 is only partially displayed because there are more than 100 possible values between the largest and smallest observations.

Frequency distribution table is much more informative if possible

observed values is less then 20. If more entry is observed then grouped data is used.

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

Grouped Data

According to their frequency of occurrence. When observations are sorted into classes of more than one value result is referred to as a frequency for grouped data. (Shown in table 2.2)

- The general structure of this frequency distribution is the data's are grouped into class intervals with 10 possible values each.
- The frequency (f) column shows the frequency of observations in each class and, at the bottom, the total number of observations in all classes.

GUIDELINES

Table 2.3
FREQUENCY DISTRIBUTION WITH TOO MANY INTERVALS

WEIGHT	*f*
245–249	1
240–244	0
235–239	0
230–234	0
225–229	2
220–224	1
215–219	0
210–214	0
205–209	2
200–204	0
195–199	0
190–194	4
185–189	1
180–184	2
175–179	2
170–174	5
165–169	7
160–164	5
155–159	9
150–154	8
145–149	1
140–144	0
135–139	2
130–134	1
Total	53

Table 2.4
FREQUENCY DISTRIBUTION WITH TOO FEW INTERVALS

WEIGHT	*f*
200–249	6
150–199	43
100–149	4
Total	53

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,

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

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.***

OUTLIERS

An outlier is an extremely high or extremely low data point relative to the nearest data point and the rest of the neighboring co-existing values in a data graph or dataset you're working with.

Outliers are extreme values that stand out greatly from the overall pattern of values in a dataset or graph.

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.

This type of distribution is especially helpful when you must compare two or more distributions based on different total numbers of observations.

The conversion to relative frequencies allows a direct comparison of the shapes of two distributions without adjust other observations.

Constructing Relative Frequency Distributions

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

Table 2.5 illustrates a relative frequency distribution based on the weight distribution of Table 2.2.

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.*

Percentages or Proportions

Some people prefer to deal with percentages rather than proportions because percentages usually lack decimal points. A proportion always varies between 0 and 1, whereas a percentage always varies between 0 percent and 100 percent.

To convert the relative frequencies, multiply each proportion by 100; that is, move the decimal point two places to the right.

CUMULATIVE FREQUENCY DISTRIBUTIONS

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

Cumulative frequencies are usually converted, in turn, to cumulative percentages. Cumulative percentages are often referred to as percentile ranks.

Constructing Cumulative Frequency Distributions

To convert a frequency distribution into a cumulative frequency distribution, add to the frequency of each class the sum of the frequencies of all classes ranked below it.

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		

Cumulative Percentages

As has been suggested, if relative standing within a distribution is particularly important, then cumulative frequencies are converted to cumulative percentages

To obtain this cumulative percentage, the cumulative frequency of the class should be divided by the total frequency of the entire distribution.

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. Thus a weight has a percentile rank of 80 if equal or lighter weights constitute 80 percent of the entire distribution.

FREQUENCY DISTRIBUTIONS FOR QUALITATIVE (NOMINAL) DATA

Frequency distributions for qualitative data are easy to construct. Simply determine the frequency with which observations occupy Each class, and report these frequencies as shown in Table 2.7 for the Face book profile survey

Table 2.7
FACEBOOK PROFILE SURVEY

Response	*f*
Yes	56
No	27
Total	83

Qualitative data have an ordinal level of measurement because Observations can be ordered from least to most, that order should be preserved in the frequency table

Relative and Cumulative Distributions for Qualitative Data

Frequency distributions for qualitative variables can always be converted into relative frequency distributions.

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		

**To avoid a value of .00 for General, proportions are carried three places to the right of the decimal point.*

GRAPHS

Data can be described clearly and concisely with the aid of a well-constructed frequency distribution. And data can often be described even more vividly by converting frequency distributions into graphs.

GRAPHS FOR QUANTITATIVE DATA

Histograms

A bar-type graph for quantitative data. The common boundaries between adjacent bars emphasize the continuity of the data, as with continuous variables.

A histogram is a display of statistical information that uses rectangles to show the frequency of data items in successive numerical intervals of equal size.

Important features of histograms

- Equal units along the horizontal axis (the X axis, or abscissa) reflect the various class intervals of the frequency distribution.
- Equal units along the vertical axis (the Y axis, or ordinate) reflect increases in frequency. (The units along the vertical axis do not have to be the same width as those along the horizontal axis.)
- The intersection of the two axes defines the origin at which both numerical scales equal 0.
- Numerical scales always increase from left to right along the horizontal axis and from bottom to top along the vertical axis
- The body of the histogram consists of a series of bars whose heights reflect the frequencies for the various classes.
- The adjacent bars in histograms have common boundaries that emphasize the continuity of quantitative data for continuous variables.
- The introduction of gaps between adjacent bars would suggest an artificial disruption in the data more appropriate for discrete quantitative variables or for qualitative variables.

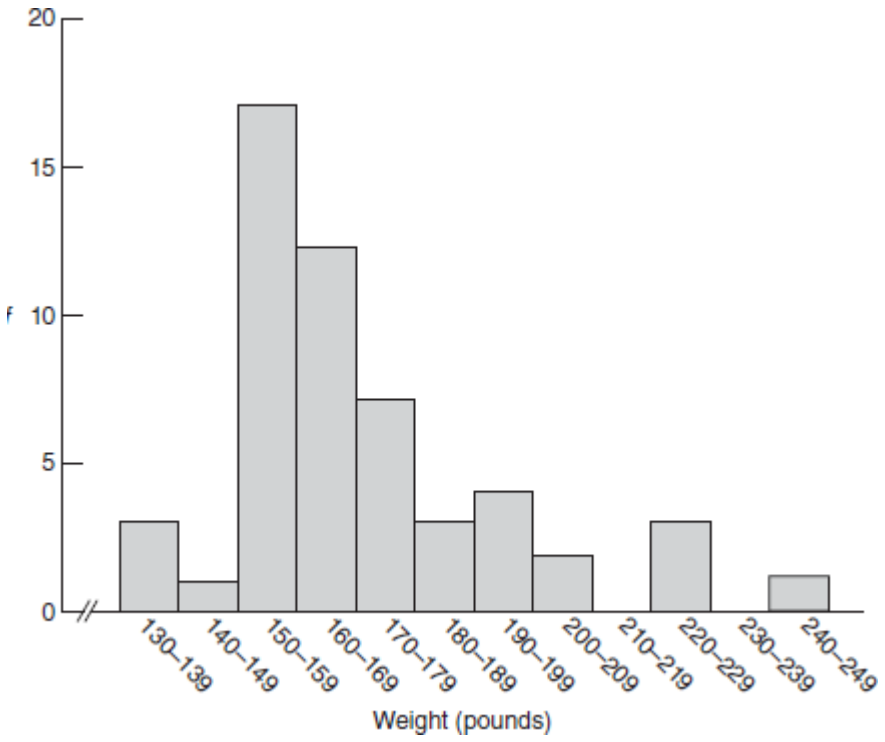


Figure: Histogram

Frequency Polygon

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

Step-by-step transformation of a histogram into a frequency polygon

A. This panel shows the histogram for the weight distribution.
B. Place dots at the midpoints of each bar top or, in the absence of bar tops, at midpoints for classes on the horizontal axis, and connect them with straight lines.
C. c. Anchor the frequency polygon to the horizontal axis. First, extend the upper tail to the midpoint of the first unoccupied class on the upper flank of the histogram. Then extend the lower tail to the midpoint of the first unoccupied class on the lower flank of the histogram. Now all of the area under the frequency polygon is enclosed completely.
D. Finally, erase all of the histogram bars, leaving only the frequency polygon.

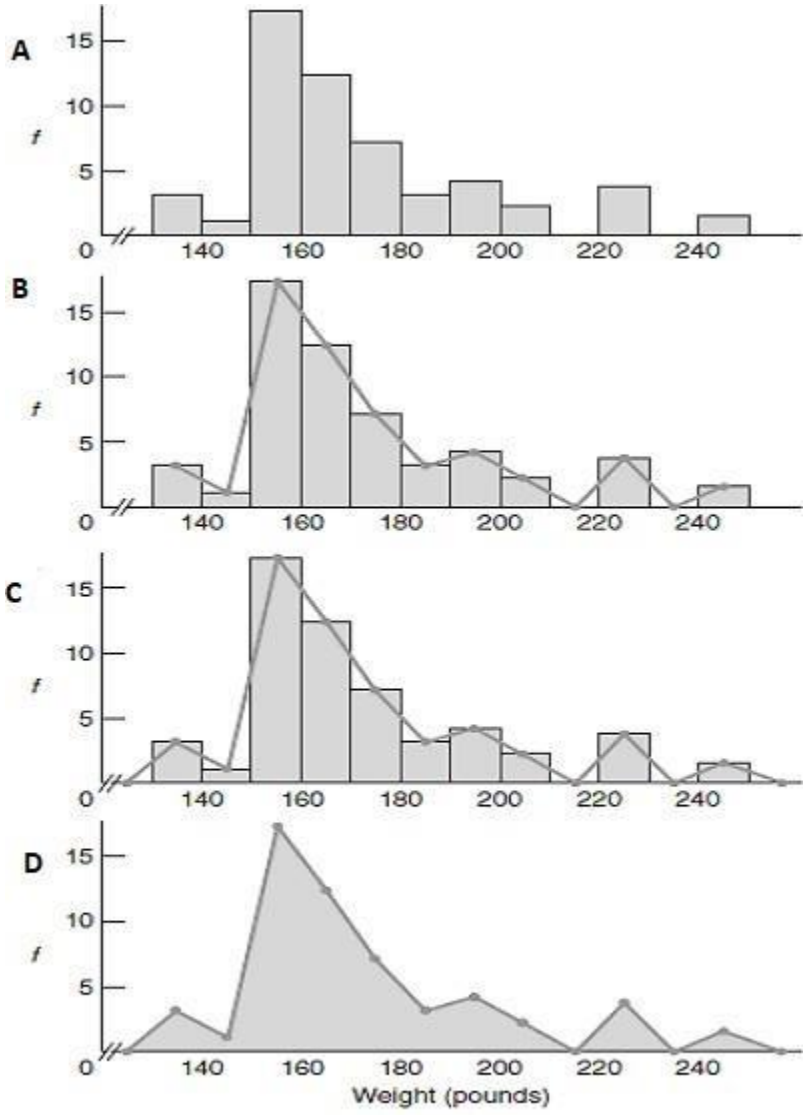


FIGURE
Transition from histogram to frequency polygon.

Stem and Leaf Displays

Another technique for summarizing quantitative data is a stem and leaf display. Stem and leaf displays are ideal for summarizing distributions, such as that for weight data, without destroying the identities of individual observations.

Constructing Stem and Leaf Display

The leftmost panel of table re-creates the weights.

To construct the stem and leaf display for the table given below, first note that, when counting by tens, the weights range from the 130s to the 240s.

Arrange a column of numbers, the stems, beginning with 13 (representing the 130s) and ending with 24 (representing the 240s). 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.

For example

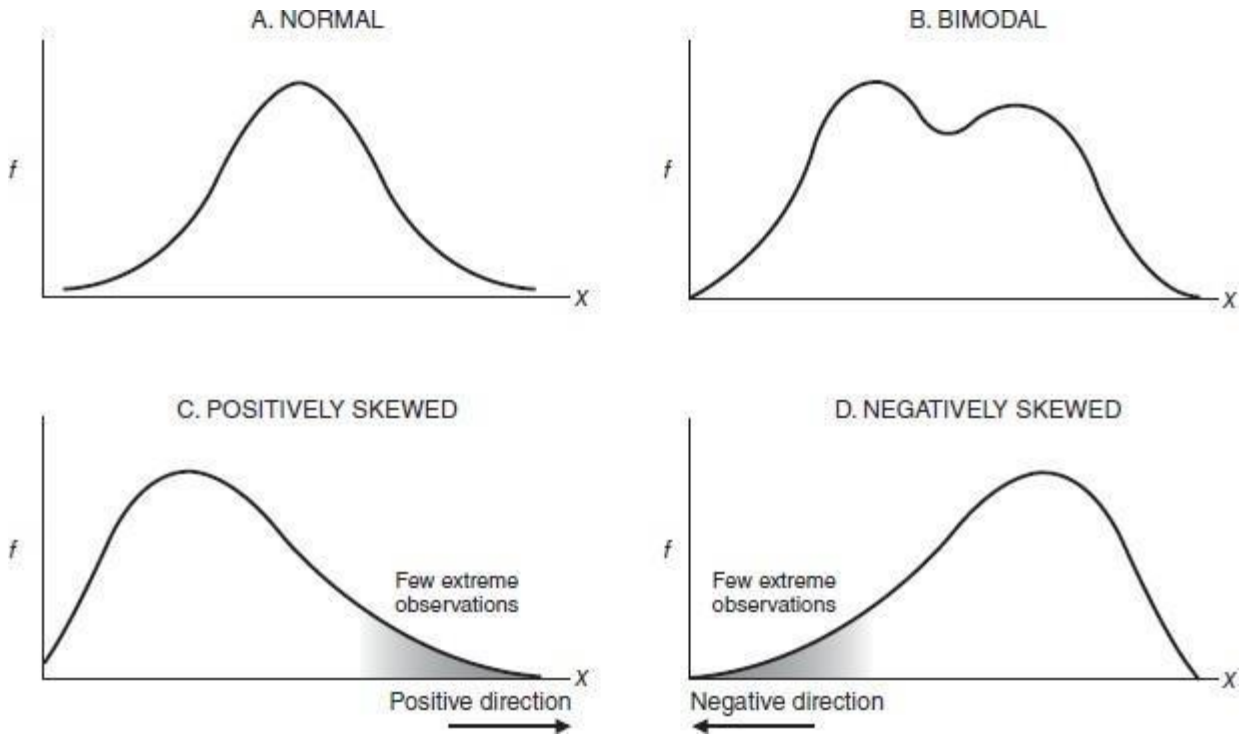
Enter each raw score into the stem and leaf display. As suggested by the shaded coding in Table 2.9, the first raw score of 160 reappears as a leaf of 0 on a stem of 16. The next raw score of 193 reappears as a leaf of 3 on a stem of 19, and the third raw score of 226 reappears as a leaf of 6 on a stem of 22, and so on, until each raw score reappears as a leaf on its appropriate stem.

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

RAW SCORES					STEM AND LEAF DISPLAY
160	165	135	175		
193	168	245	165	13	3 5 5
226	169	170	185	14	5
152	160	156	154	15	2 7 1 7 8 0 2 0 2 6 9 8 2 6 4 7 6
180	170	160	179	16	0 3 5 8 9 0 0 0 6 5 5 5
205	150	225	165	17	2 0 0 0 2 5 9
163	152	190	206	18	0 0 5
157	160	159	165	19	3 0 0 0
151	190	172	157	20	5 6
157	150	190	156	21	
220	133	166	135	22	6 0 5
145	180	158		23	
158	152	152		24	5
172	170	156			

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. Below figure shows some of the more typical shapes for smoothed frequency polygons (which ignore the inevitable irregularities of real data).

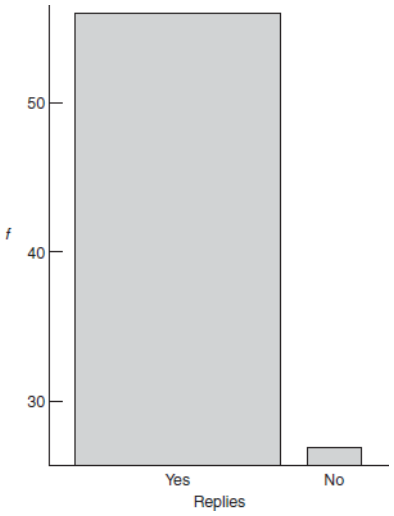


A GRAPH FOR QUALITATIVE (NOMINAL) DATA

- 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. Likewise, equal segments along the vertical axis reflect increases in frequency. The body of the bar graph consists of a series of bars whose heights reflect the frequencies for the various words or classes.
- A person's answer to the question "Do you have a Facebook profile?" is either Yes or No, not some impossible intermediate value, such as 40 percent Yes and 60 percent No.
- Gaps are placed between adjacent bars of bar graphs to emphasize the discontinuous nature of qualitative data.

MISLEADING GRAPHS

Graphs can be constructed in an unscrupulous manner to support a particular point of view.
Popular sayings says, including "Numbers don't lie, but statisticians do" and "There are three kinds of lies—lies, damned lies, and statistics."



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.***

steps

Describing Data with Averages

MODE

The mode reflects the value of the most frequently occurring score.
In other words
A mode is defined as the value that has a higher frequency in a given set of values. It is the value that appears the most number of times.
Example:
In the given set of data: 2, 4, 5, 5, 6, 7, the mode of the data set is 5 since it has appeared in the set twice.

Types of Modes

Bimodal, Trimodal & Multimodal (More than one mode)

- When there are two modes in a data set, then the set is called **bimodal**

For example, The mode of Set A = {2,2,2,3,4,4,5,5,5} is 2 and 5, because both 2 and 5 is repeated three times in the given set.

- When there are three modes in a data set, then the set is called **trimodal**

For example, the mode of set A = {2,2,2,3,4,4,5,5,5,7,8,8,8} is 2, 5 and 8

- When there are four or more modes in a data set, then the set is called **multimodal**

Example: The following table represents the number of wickets taken by a bowler in 10 matches. Find the mode of the given set of data.

Match No.	1	2	3	4	5	6	7	8	9	10
No. of Wickets	2	1	1	3	2	3	2	2	4	1

It can be seen that 2 wickets were taken by the bowler frequently in different matches. Hence, the mode of the given data is 2.

MEDIAN

The median reflects the middle value when observations are ordered from least to most.
The median splits a set of ordered observations into two equal parts, the upper and lower halves.

Finding the Median

- Order scores from least to most.
- If the total number of observation given is odd, then the formula to calculate the median is:

  Median = $\{(n+1)/2\}^{th}$ term / observation

- If the total number of observation is even, then the median formula is:

  Median = $1/2[(n/2)^{th}$ term + $\{(n/2)+1\}^{th}$ term]

Example 1:

Find the median of the following:

4, 17, 77, 25, 22, 23, 92, 82, 40, 24, 14, 12, 67, 23, 29

Solution:
n= 15
When we put those numbers in the order we have:

4, 12, 14, 17, 22, 23, 23, 24, 25, 29, 40, 67, 77, 82, 92,

Median = $\{(n+1)/2\}^{th}$ term
= (15+1)/2
=8
The 8^{th} term in the list is 24
The median value of this set of numbers is 24.

Example 2:
Find the median of the following:
9,7,2,11,18,12,6,4

Solution
n=8
When we put those numbers in the order we have:
2, 4, 6, 7, 9,11, 12, 18

Median = $1/2[(n/2)^{th}$ term + $\{(n/2)+1\}^{th}$ term]

= ½ [(8/2) term + ((8/2)+1)term]
=1/2[4^{th} term+5^{th} term] (in our list 4^{th} term is 7 and 5^{th} term is 9)
= ½[7+9]
=1/2(16)
=8
The median value of this set of numbers is 8.

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

Mean is the average of the given numbers and is calculated by dividing the sum of given numbers by the total number of numbers.

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

Types of means

- Sample mean
- Population mean

Sample Mean
The sample mean is a central tendency measure. The arithmetic average is computed using samples or random values taken from the population. It is evaluated as the sum of all the sample variables divided by the total number of variables.

SAMPLE MEAN

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

Population Mean

The population mean can be calculated by the sum of all values in the given data/population divided by a total number of values in the given data/population.

POPULATION MEAN

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

AVERAGES FOR QUALITATIVE AND RANKED DATA

Mode

he mode always can be used with qualitative data.

Median

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

Describing Variability

RANGE

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

The range in statistics for a given data set is the difference between the highest and lowest values. For example, if the given data set is {2,5,8,10,3}, then the range will be 10 – 2 = 8.

Example 1: Find the range of given observations: 32, 41, 28, 54, 35, 26, 23, 33, 38, 40.

Solution: Let us first arrange the given values in ascending order.
23, 26, 28, 32, 33, 35, 38, 40, 41, 54
Since 23 is the lowest value and 54 is the highest value, therefore, the range of the observations will be;
Range (X) = Max (X) – Min (X)
= 54 – 23
= 31

VARIANCE

Variance is a measure of how data points differ from the mean. A variance is a measure of how far a set of data (numbers) are spread out from their mean (average) value.

Formula

$\sigma = \Sigma(x-\mu)^2$ or

Variance = (Standard deviation)^2= σ^2 => $\sigma^2 = \Sigma(x-\mu)^2 /n$

the values of all scores must be added and then divided by the total number of scores.

Example
X = 5, 8, 6, 10, 12, 9, 11, 10, 12, 7
Solution
Mean = sum (x)/ n
n= 10
sum (x) = 5+8+6+10+12+9+11+10+12+ 7

= 90

Mean=> $\mu = 90 / 10 = 9$

Deviation from mean

$x - \mu = -4, -1, -3, 1, 3, 0, 2, 1, 3, -2$

$(x-\mu)^2 = 16, 1, 9, 1, 9, 0, 4, 1, 9, 4$

$\Sigma(x-\mu)^2 = 16+1+9+1+9+0+4+1+9+4$

$=54$

$\sigma^2 = \Sigma(x-\mu)^2 / n$

$=54/10$

$= 5.4$

TANDARD DEVIATION

The standard deviation, the square root of the mean of all squared deviations from the mean, that is,

Standard deviation = √variance

Standard Deviation: A rough measure of the average (or standard) amount by which scores deviate

Standard Deviation: A Measure of Distance

The mean is a measure of position, but the standard deviation is a measure f distance (on either side of the mean of the distribution).

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

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

$$SS = \sum(X - \mu)^2 \qquad (4.1)$$

"The sum of squares equals the sum of all squared deviation scores." You can reconstruct this formula by remembering the following three steps:

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, $\Sigma (X - \mu)2$.

VARIANCE FOR POPULATION

$$\sigma^2 = \frac{SS}{N} \qquad (4.5)$$

STANDARD DEVIATION FOR POPULATION

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{SS}{N}} \qquad (4.6)$$

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

	4	**5**
X	$X-\mu$	$(X-\mu)^2$
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** $\Sigma X=70$ **6** $SS=\Sigma (X-\mu)^2=22$

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

Sample notation can be substituted for population notation in the above two formulas without causing any essential changes:

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

$$SS=\Sigma(X-\overline{X})^2 \qquad (4.3)$$

VARIANCE FOR SAMPLE

$$s^2=\frac{SS}{n-1} \qquad (4.7)$$

STANDARD DEVIATION FOR SAMPLE

$$s=\sqrt{s^2}=\sqrt{\frac{SS}{n-1}} \qquad (4.8)$$

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 4.5
TWO ESTIMATES OF POPULATION VARIABILITY

WHEN μ IS UNKNOWN ($\bar{X} = 3$)

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

$\Sigma(X - \bar{X}) = 0$ $\Sigma(X - \bar{X})^2 = 30$

$df = n - 1 = 5 - 1 = 4$

$$s^2(df = n-1) = \frac{\Sigma(X - \bar{X})^2}{n-1} = \frac{30}{4} = 7.50$$

WHEN μ IS KNOWN ($\mu = 2$)

X	$X - \mu$	$(X - \mu)^2$
7	$7 - 2 = 5$	25
3	$3 - 2 = 1$	1
1	$1 - 2 = -1$	1
0	$0 - 2 = -2$	4
4	$4 - 2 = 2$	4

$\Sigma(X - \mu) = 5$ $\Sigma(X - \mu)^2 = 35$

$df = n = 5$

$$s^2(df = n) = \frac{\Sigma(X - \mu)^2}{n} = \frac{35}{5} = 7.00$$

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.
- Degrees of freedom are the number of independent variables that can be estimated in a statistical analysis. These values of these variables are without constraint, although the values do impost restrictions on other variables if the data set is to comply with estimate parameters.
- Degrees of Freedom (df) The number of values free to vary, given one or more mathematical restrictions.

Formula
Degree of freedom **df = n-1**

Example
Consider a data set consists of five positive integers. The sum of the five integers must be the multiple of 6. The values are randomly selected as 3, 8, 5, and 4.
The sum of this for values is 20. So we have to choose the fifth integer to make the sum divisible by 6. Therefore the fifth element is 10.

The number of degrees of Degrees of Freedom (df) The number of values free to vary, given one or more mathematical restrictions. Freedom—in the numerator, as in the formulas for s2 and s. In fact, we can use degrees of freedom to rewrite the formulas for the sample variance and standard deviation:

VARIANCE FOR SAMPLE

$$s^2 = \frac{SS}{n-1} = \frac{SS}{df} \qquad (4.9)$$

STANDARD DEVIATION FOR SAMPLE

$$s = \sqrt{\frac{SS}{n-1}} = \sqrt{\frac{SS}{df}} \qquad (4.10)$$

INTERQUARTILE RANGE (IQR)

The interquartile range (IQR), is simply the range for the middle 50 percent of the scores. More specifically, the IQR equals the distance between the third quartile (or 75th percentile) and the first quartile (or 25th percentile), that is, after the highest quarter (or top 25 percent) and the lowest quarter (or bottom 25 percent) have been trimmed from the original set of scores. Since most distributions are spread more widely in their extremities than their middle, the IQR tends to be less than half the size of the range.

Simply, The IQR describes the middle 50% of values when ordered from lowest to highest. To find the interquartile range (IQR), first find the median (middle value) of the lower and upper half of the data. These values are quartile 1 (Q1) and quartile 3 (Q3). The IQR is the difference between Q3 and Q1.

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
   ↑
   2 1
4. third quartile = 11
5. 7, 9, 9, 10, 11, 11, 13
   ↑
   1 2
6. first quartile = 9
7. IQR = 11 − 9 = 2

Normal Distributions and Standard (z) Scores

THE NORMAL CURVE

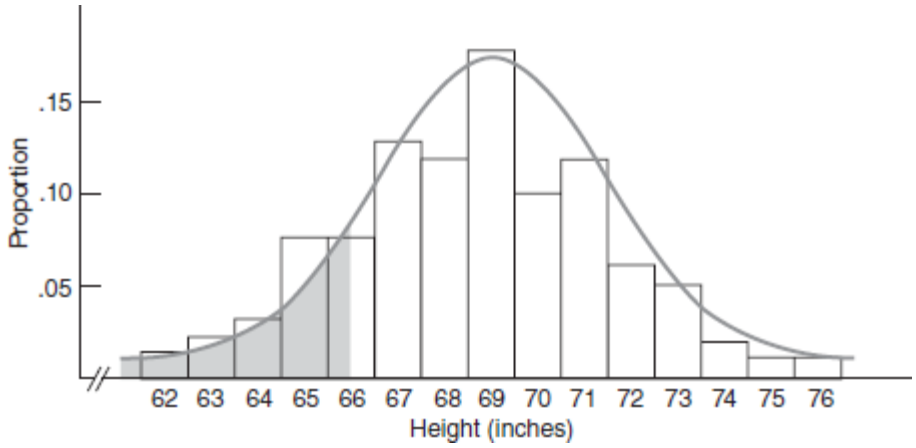
The normal distribution is a continuous probability distribution that is symmetrical on both sides of the mean, so the right side of the center is a mirror image of the left side.

Properties of the Normal Curve

- The normal curve is a theoretical curve defined for a continuous variable, as described in Section 1.6, and noted for its symmetrical bell-shaped form, as revealed in below figure
- Because the normal curve is symmetrical, its lower half is the mirror image of its upper half.
- 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.

Properties of a normal distribution

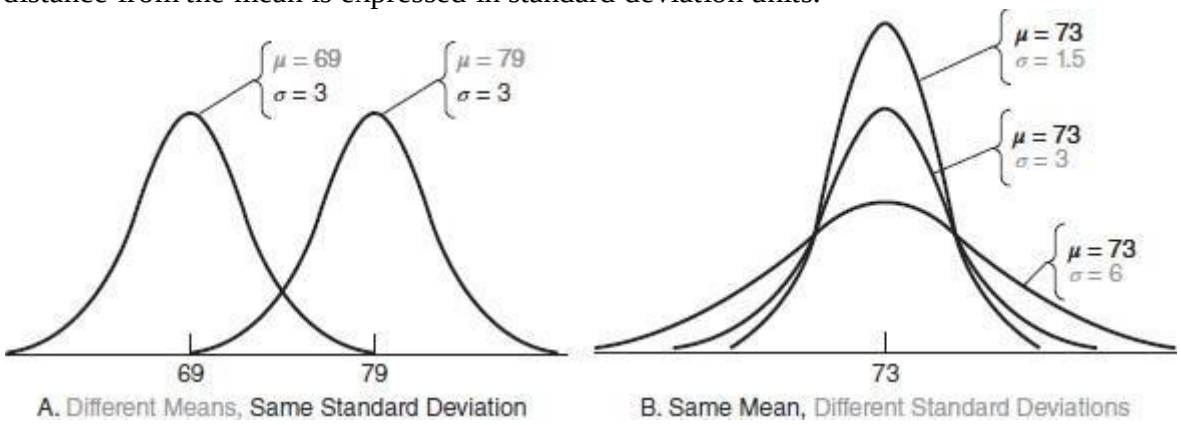
- The mean, mode and median are all equal.
- The curve is symmetric at the center (i.e. around the mean, μ).
- Exactly half of the values are to the left of center and exactly half the values are to the right.
- The total area under the curve is 1.



Different Normal Curves

As a theoretical exercise, it is instructive to note the various types of normal curves that are produced by an arbitrary change in the value of either the mean (μ) or the standard deviation (σ).

Obvious differences in appearance among normal curves are less important than you might suspect. 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.



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.

A z score can be defined as a measure of the number of standard deviations by which a score is below or above the mean of a distribution. In other words, it is used to determine the distance of a score from the mean. If the z score is positive it indicates that the score is above the mean. If it is negative then the score will be below the mean. However, if the z score is 0 it denotes that the data point is the same as the mean.

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),

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

Where X is the original score and μ and σ are the mean and the standard deviation, respectively, for the normal distribution of the original scores. Since identical units of measurement appear in both the numerator

and denominator of the ratio for z, the original units of measurement cancel each other and the z score emerges as a unit-free or standardized number, often referred to as a standard score.

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
Example
Suppose on a GRE test a score of 1100 is obtained. The mean score for the GRE test is 1026 and the population standard deviation is 209. In order to find how well a person scored with respect to the score of an average test taker, the z score will have to be determined.

The steps to calculate the z score are as follows:

- Step 1: Write the value of the raw score in the z score equation. $z = (1100-\mu) / \sigma$
- Step 2: Write the mean and standard deviation of the population in the z score formula. $z = (1100-1026) / 209$
- Step 3: Perform the calculations to get the required z score. $z = 0.345$
- Step 4: A z score table can be used to find the percentage of test-takers that are below the score of the person. Using the first two digits of the z score, determine the row containing these digits of the z table. Now using the 2nd digit after the decimal, find the corresponding column. The intersection of this row and column will give a value. As shown below, this value will be 0.6368 for the given example.
- Step 5: Use the value from step 5 and multiply it by 100 to get the required percentage. 0.6368 * 100 = 63.68%. This shows that 63.68% of test-takers scores are lesser than the given raw score.

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. This is the one normal curve for which a table is actually available**.**

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.

For a standard normal curve

Mean = 0

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

Standard deviation = 1

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

Standard Normal Table
The standard normal table consists of columns of z scores coordinated with columns of proportions

Using the Top Legend of the Table
Notice that columns are arranged in sets of three, designated as A, B, and C in the legend at the top of the table. When using the top legend, all entries refer to the upper half of the standard normal curve. The entries in column A are z scores, beginning with 0.00 and ending with 4.00

Given a z score of zero or more, columns B and C indicate how the z score splits the area in the upper half of the normal curve. As suggested by the shading in the top legend, column B indicates the proportion of area between the mean and the z score, and column C indicates the proportion of area beyond the z score, in the upper tail of the standard normal curve.

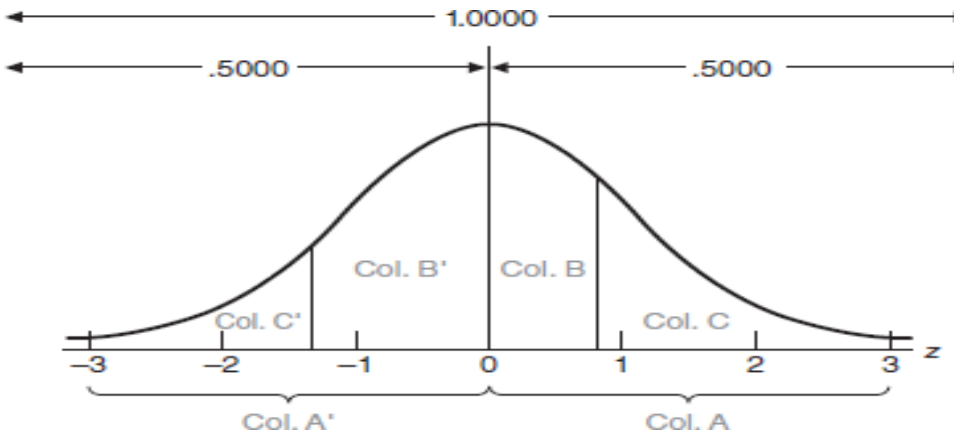
Using the Bottom Legend of the Table

Now the columns are designated as A′, B′, and C′ in the legend at the bottom of the table. When using the bottom legend, all entries refer to the lower half of the standard normal curve.

A negative z score, columns B′ and C′ indicate how that z score splits the lower half of the normal curve. As suggested by the shading in the bottom legend of the table, column B′ indicates the proportion of area between the mean and the negative z score, and column C′ indicates the proportion of area beyond the negative z score, in the lower tail of the standard normal curve.

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′

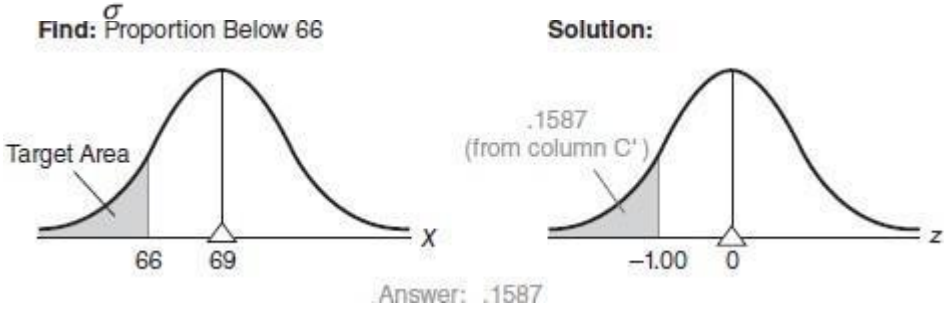


FINDING PROPORTIONS

Finding Proportions for One Score

- Sketch a normal curve and shade in the target area,
- Plan your solution according to the normal table.
- Convert *X* to *z*.

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



- Find the target area.

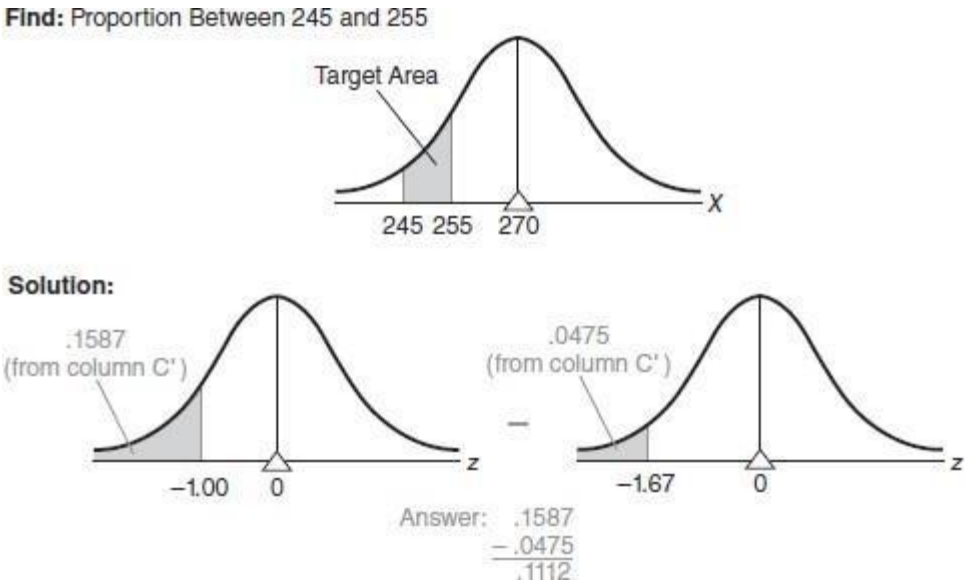
Finding Proportions between Two Scores

- Sketch a normal curve and shade in the target area, (example, find proportion between 245 to 255)
- Plan your solution according to the normal table.
- Convert *X* to *z* by expressing 255 as

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

and by expressing 245 as

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



- Find the target area.

FINDING SCORES

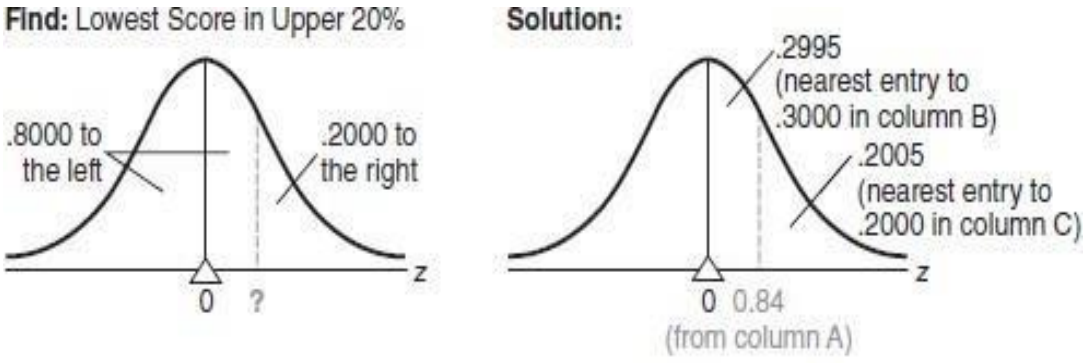
So far, we have concentrated on normal curve problems for which Table A must be consulted to find the unknown proportion (of area) associated with some known score or pair of known scores

Now we will concentrate on the opposite type of normal curve problem for which Table A must be consulted to find the unknown score or scores associated with some known proportion.

For this type of problem requires that we reverse our use of Table A by entering proportions in columns B, C, B′, or C′ and finding z scores listed in columns A or A′.

Finding One Score

- Sketch a normal curve and, on the correct side of the mean, draw a line representing the target score, as in figure



Answer: $X = \mu + (z)(\sigma)$
$= 230 + (0.84)(50)$
$= 230 + 42$
$= 272$

It's often helpful to visualize the target score as splitting the total area into two sectors—one to the left of (below) the target score and one to the right of (above) the target score

- Plan your solution according to the normal table.

In problems of this type, you must plan how to find the z score for the target score. Because the target score is on the right side of the mean, concentrate on the area in the upper half of the normal curve, as described in columns B and C.

- Find z.
- Convert z to the target score.

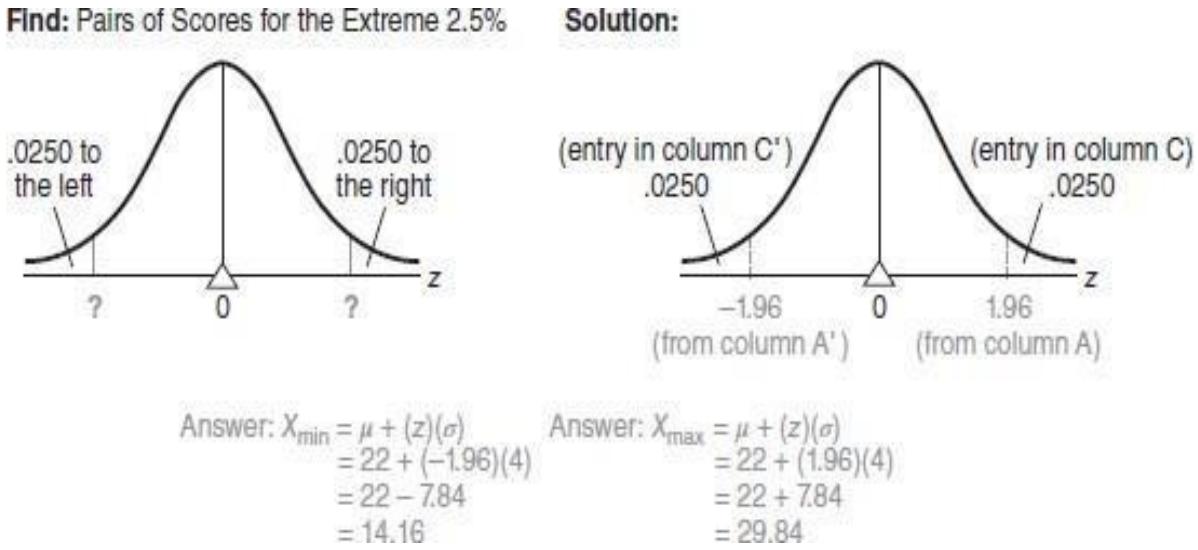
When converting z scores to original scores, you will probably find it more efficient to use the following equation

CONVERTING z SCORE TO ORIGINAL SCORE

$$X = \mu + (z)(\sigma)$$

Finding Two Scores

- Sketch a normal curve. On either side of the mean, draw two lines representing the two target scores, as in figure



Answer: $X_{min} = \mu + (z)(\sigma)$
$= 22 + (-1.96)(4)$
$= 22 - 7.84$
$= 14.16$

Answer: $X_{max} = \mu + (z)(\sigma)$
$= 22 + (1.96)(4)$
$= 22 + 7.84$
$= 29.84$

- Plan your solution according to the normal table.
- Find z.
- Convert z to the target score.

Points to Remember

1. range = largest value – smallest value in a list
2. class interval = range / desired no of classes
3. relative frequency = frequency (f)/ε(f)
4. Cumulative frequency - *add to the frequency of each class the sum of the frequencies of all classes ranked below it.*
5. Cumulative percentage = (f/cumulative f)*100
6. Histograms
7. Construction of frequency polygon
8. Stem and leaf display
9. Mode - *The value of the most frequent score.*
10. For odd no of terms **Median = {(n+1)/2}th term / observation**. For even no of terms **Median = 1/2[(n/2)th term + {(n/2)+1}th term]**
11. Mean = sum of all scores / number of scores

SAMPLE MEAN $\bar{X} = \frac{\sum X}{n}$ **POPULATION MEAN** $\mu = \frac{\sum X}{N}$

Variance $\sigma = \Sigma(x-\mu)^2$ or

Variance = (Standard deviation)^2= σ^2 $=> \sigma^2 = \Sigma(x-\mu)^2 /n$

12. Range (X) = Max (X) – Min (X)

STANDARD DEVIATION FOR POPULATION

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{SS}{N}} \qquad (4.6)$$

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

$$SS = \Sigma(X - \mu)^2 \qquad (4.1)$$

VARIANCE FOR SAMPLE

$$s^2 = \frac{SS}{n-1} \qquad (4.7)$$

STANDARD DEVIATION FOR SAMPLE

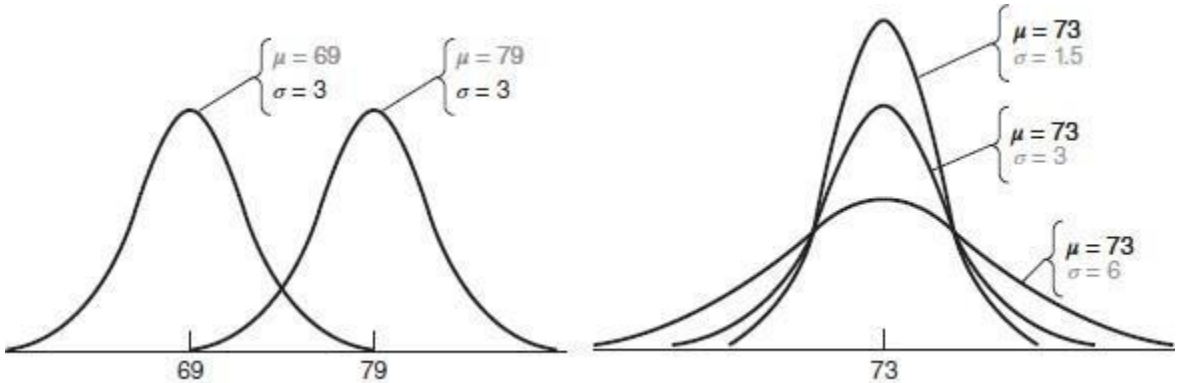
$$s = \sqrt{s^2} = \sqrt{\frac{SS}{n-1}} \qquad (4.8)$$

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

$$SS = \Sigma(X - \overline{X})^2 \qquad (4.3)$$

13. Degree of freedom **df = n-1**

14. Types of normal curve



A. Different Means, Same Standard Deviation

B. Same Mean, Different Standard Deviations

15. z – score

z SCORE

$$z = \frac{X - \mu}{\sigma} \qquad (5.1)$$

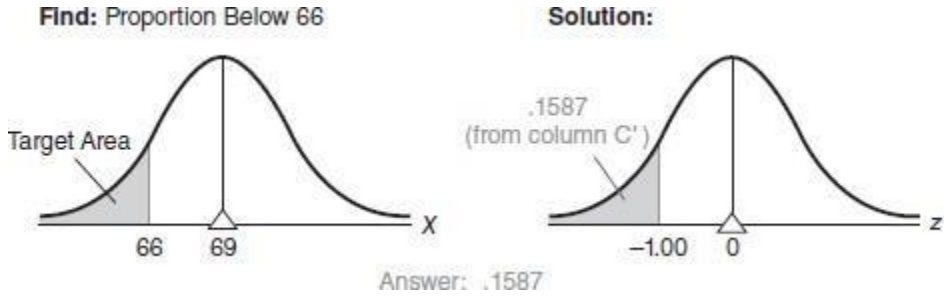
16. Standard normal curve; **mean = 0, standard deviation = 1**

17. Finding proportion

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

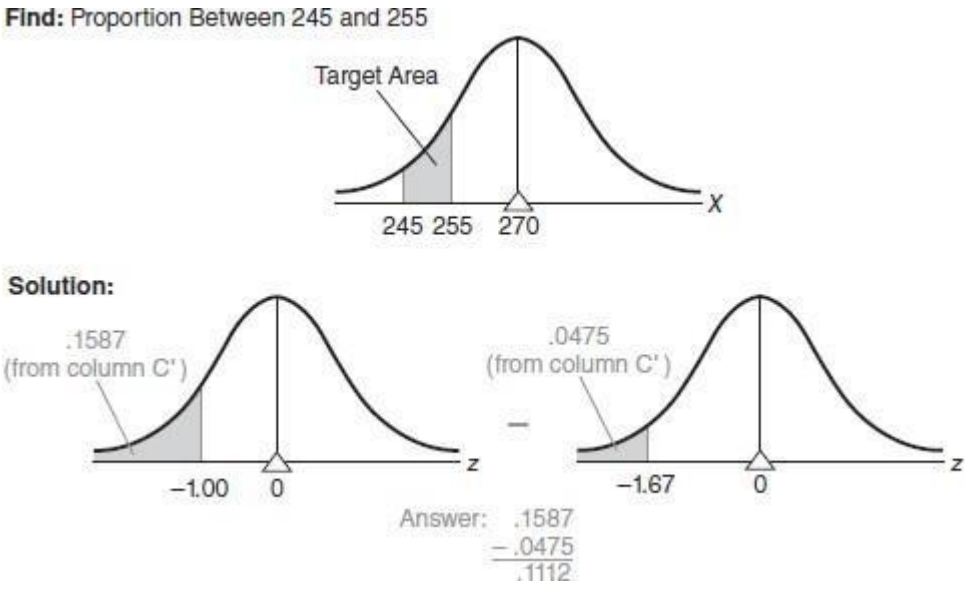
18. Finding proportion

1. For one score



Answer: .1587

2. For between two score



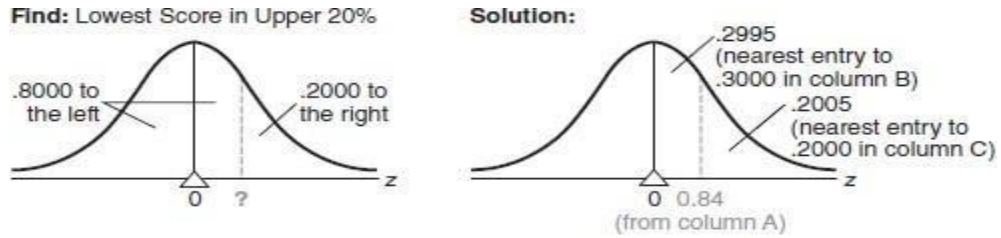
Answer: .1587
– .0475
.1112

19. Finding scores

CONVERTING z SCORE TO ORIGINAL SCORE

$$X = \mu + (z)(\sigma)$$

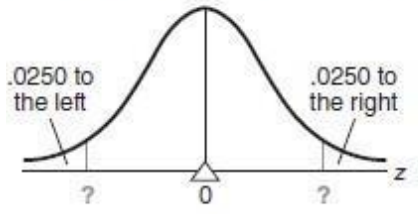
20. Finding scores – one score



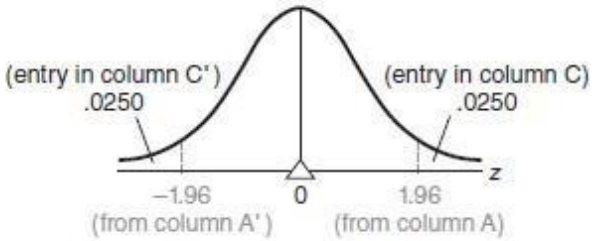
Answer: $X = \mu + (z)(\sigma)$
$= 230 + (0.84)(50)$
$= 230 + 42$
$= 272$

Two scores

Find: Pairs of Scores for the Extreme 2.5%



Solution:



Answer: $X_{min} = \mu + (z)(\sigma)$
$= 22 + (-1.96)(4)$
$= 22 - 7.84$
$= 14.16$

Answer: $X_{max} = \mu + (z)(\sigma)$
$= 22 + (1.96)(4)$
$= 22 + 7.84$
$= 29.84$

Unit – III
DESCRIBING RELATIONSHIPS

Correlation – Scatter plots – correlation coefficient for quantitative data – computational formula for correlation coefficient – Regression – regression line – least squares regression line – Standard error of estimate – interpretation of r2 – multiple regression equations – regression towards the mean

Correlation

Correlation refers to a process for establishing the relationships between two variables. You learned a way to get a general idea about whether or not two variables are related, is to plot them on a "scatter plot". While there are many measures of association for variables which are measured at the ordinal or higher level of measurement, correlation is the most commonly used approach.

Types of Correlation

- **Positive Correlation** – when the values of the two variables move in the same direction so that an increase/decrease in the value of one variable is followed by an increase/decrease in the value of the other variable.
- **Negative Correlation** – when the values of the two variables move in the opposite direction so that an increase/decrease in the value of one variable is followed by decrease/increase in the value of the other variable.
- **No Correlation** – when there is no linear dependence or no relation between the two variables.

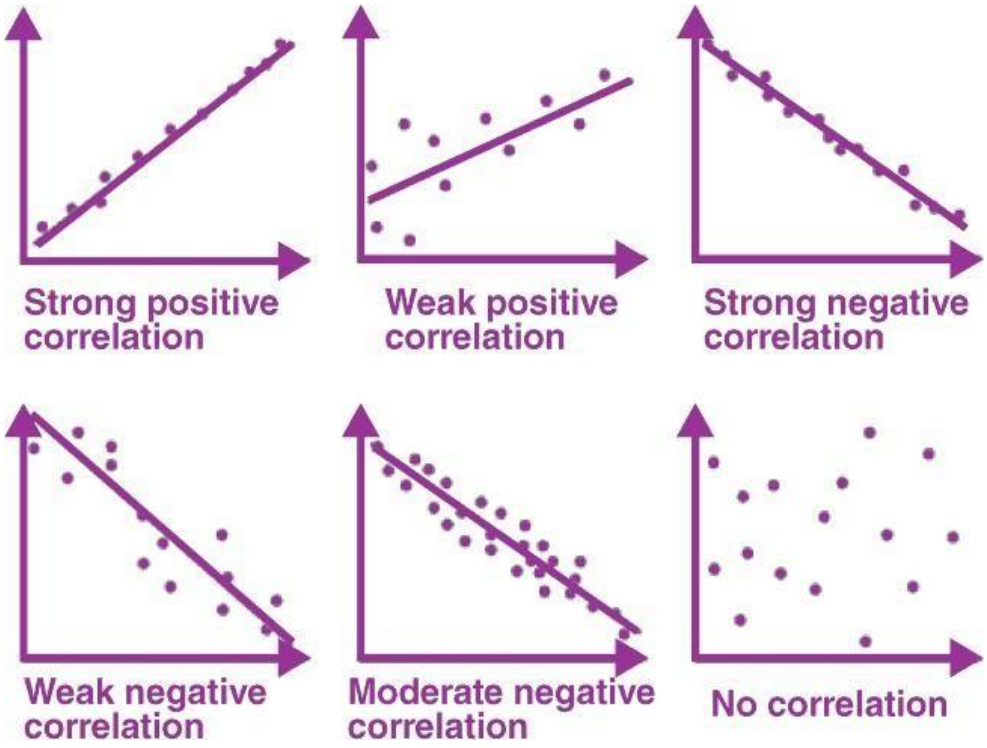


Table 6.2
THREE TYPES OF RELATIONSHIPS

A. POSITIVE RELATIONSHIP

FRIEND	SENT	RECEIVED
Doris	13	14
Steve	9	18
Mike	7	12
Andrea	5	10
John	1	6

B. NEGATIVE RELATIONSHIP

FRIEND	SENT	RECEIVED
Doris	13	6
Steve	9	10
Mike	7	14
Andrea	5	12
John	1	18

C. LITTLE OR NO RELATIONSHIP

FRIEND	SENT	RECEIVED
Doris	13	10
Steve	9	18
Mike	7	12
Andrea	5	6
John	1	14

SCATTERPLOTS

A scatter plot is a graph containing a cluster of dots that represents all pairs of scores. In other words Scatter plots are the graphs that present the relationship between two variables in a data-set. It represents data points on a two-dimensional plane or on a Cartesian system.

Construction of scatter plots

- The independent variable or attribute is plotted on the X-axis. Fig 6.1
- The dependent variable is plotted on the Y-axis.

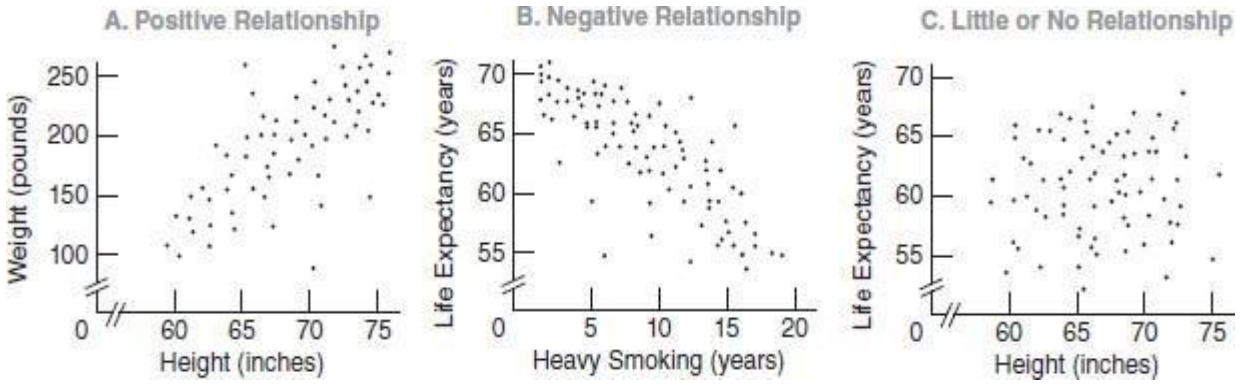
- Use each pair of scores to locate a dot within the scatter plot

Positive, Negative, or Little or No Relationship?

The first step is to note the tilt or slope, if any, of a dot cluster.
A dot cluster that has a slope from the lower left to the upper right, as in panel A of below figure reflects a **positive relationship.**

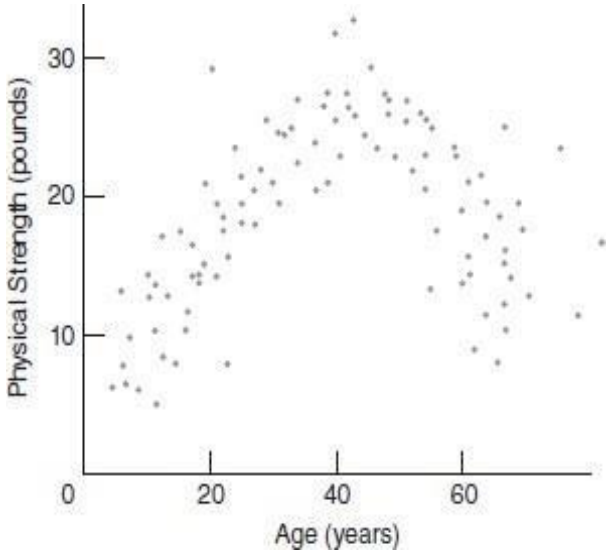
A dot cluster that has a slope from the upper left to the lower right, as in panel B of below figure reflects a **negative relationship.**

A dot cluster that lacks any apparent slope, as in panel C of below figure reflects **little or no relationship.**



Perfect Relationship
A dot cluster that equals (rather than merely approximates) a straight line reflects a perfect relationship between two variables.

Curvilinear Relationship
The previous discussion assumes that a dot cluster approximates a straight line and, therefore, reflects a **linear relationship.** But this is not always the case. Sometimes a dot cluster approximates a bent or curved line, as in below figure, and therefore reflects a **curvilinear relationship.**



A CORRELATION COEFFICIENT FOR QUANTITATIVE DATA : *r*

The correlation coefficient, r, is a summary measure that describes the extent of the statistical relationship between two interval or ratio level variables.

Properties of r

- The correlation coefficient is scaled so that it is always between -1 and +1.
- When r is close to 0 this means that there is little relationship between the variables and the farther away from 0 r is, in either the positive or negative direction, the greater the relationship between the two variables.
- The sign of r indicates the type of linear relationship, whether positive or negative.
- The numerical value of r, without regard to sign, indicates the strength of the linear relationship.
- A number with a plus sign (or no sign) indicates a positive relationship, and a number with a minus sign indicates a negative relationship

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

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

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

The sum of the products term in the numerator, *SPxy*, is defined in below formula

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

Or the formula is written as

$$r = \frac{n\left(\sum xy\right) - \left(\sum x\right)\left(\sum y\right)}{\sqrt{\left[n\sum x^2 - \left(\sum x\right)^2\right]\left[n\sum y^2 - \left(\sum y\right)^2\right]}}$$

Where n = Number of Information

Σx = Total of the First Variable Value

Σy = Total of the Second Variable Value

Σxy = Sum of the Product of first & Second Value

Σx^2 = Sum of the Squares of the First Value

Σy^2 = Sum of the Squares of the Second Value

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

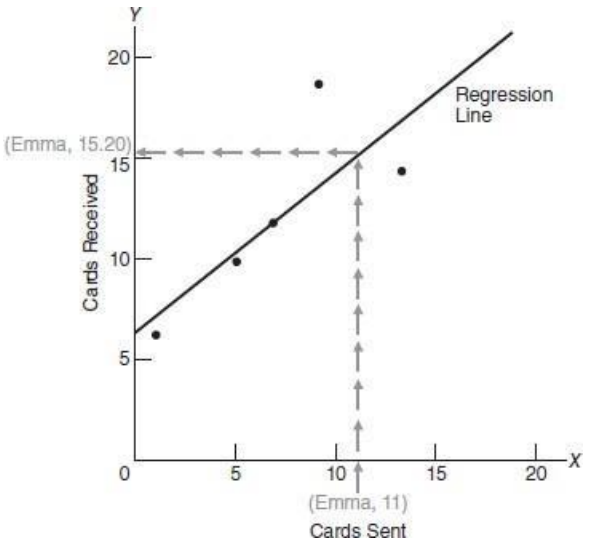
REGRESSION

A regression is a statistical technique that relates a dependent variable to one or more independent (explanatory) variables. A regression model is able to show whether changes observed in the dependent variable are associated with changes in one or more of the explanatory variables.

Regression captures the correlation between variables observed in a data set, and quantifies whether those correlations are statistically significant or not.

A Regression Line

a regression line is a line that best describes the behaviour of a set of data. In other words, it's a line that best fits the trend of a given data.



The purpose of the line is to describe the interrelation of a dependent variable (Y variable) with one or many independent variables (X variable). By using the equation obtained from the regression line an analyst can forecast future behaviours of the dependent variable by inputting different values for the independent ones.

Types of regression

The two basic types of regression are

- Simple linear regression

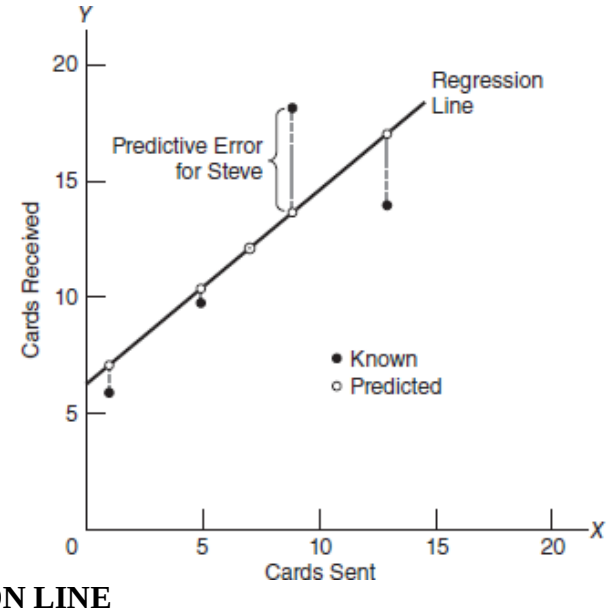
Simple linear regression uses one independent variable to explain or predict the outcome of the dependent variable Y

- Multiple linear regression

Multiple linear regressions use two or more independent variables to predict the outcome

Predictive Errors

Prediction error refers to the difference between the predicted values made by some model and the actual values.



LEAST SQUARES REGRESSION LINE

The placement of the regression line minimizes not the total predictive error 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 least squares regression line.

The Least Squares Regression Line is the line that minimizes the sum of the residuals squared. The residual is the vertical distance between the observed point and the predicted point, and it is calculated by subtracting ˆy from y.

Formula

$y' = bx+a$ b – slope , a – y intercept

$$b = \frac{N\Sigma(xy) - \Sigma x \Sigma y}{N\Sigma(x^2) - (\Sigma x)^2}$$

$b = \frac{\Sigma y - m \Sigma x}{N}$

Example

"x"	"y"
2	4
3	5
5	7
7	10
9	15

Step 1: For each (x,y) calculate x^2 and xy:

x	y	x^2	xy
2	4	4	8
3	5	9	15
5	7	25	35
7	10	49	70
9	15	81	135

Step 2: Sum x, y, x2 and xy (gives us Σx, Σy, Σx2 and Σxy):

Σx: 26 Σy: 41 Σx^2: 168 Σxy: 263

Step 3: Calculate Slope b

$$b = \frac{N \Sigma(xy) - \Sigma x \Sigma y}{N \Sigma(x^2) - (\Sigma x)^2}$$

$$= \frac{5 \times 263 - 26 \times 41}{5 \times 168 - 262}$$

$$= \frac{1315 - 1066}{840 - 676}$$

$$= \frac{249}{164}$$

$b = 1.5183.$

Step 4: Calculate Intercept a

$$a = \frac{\Sigma y - b \Sigma x}{N}$$

$$= \frac{41 - 1.5183 \times 26}{5}$$

$a = 0.3049.$

Step 5: $y' = bx+a$

$y' = 1.518x + 0.305$

x	y	y = 1.518x + 0.305	error
2	4	3.34	−0.66
3	5	4.86	−0.14
5	7	7.89	0.89
7	10	10.93	0.93
9	15	13.97	−1.03

To predict the y value we can assume any value for x.
Assume x = 8.
Then y = 1.518 x 8 + 0.305
= 12.45

STANDARD ERROR OF ESTIMATE ,$s_{y|x}$

The standard error of the estimate is a measure of the accuracy of predictions. The regression line is the line that minimizes the sum of squared deviations of prediction (also called the sum of squares error), and the standard error of the estimate is the square root of the average squared deviation.

The standard error of estimate and symbolized as $s_{y|x}$, this estimate of predictive error complies with the general format for any sample standard deviation, that is, the square root of a sum of squares term divided by its degrees of freedom.

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

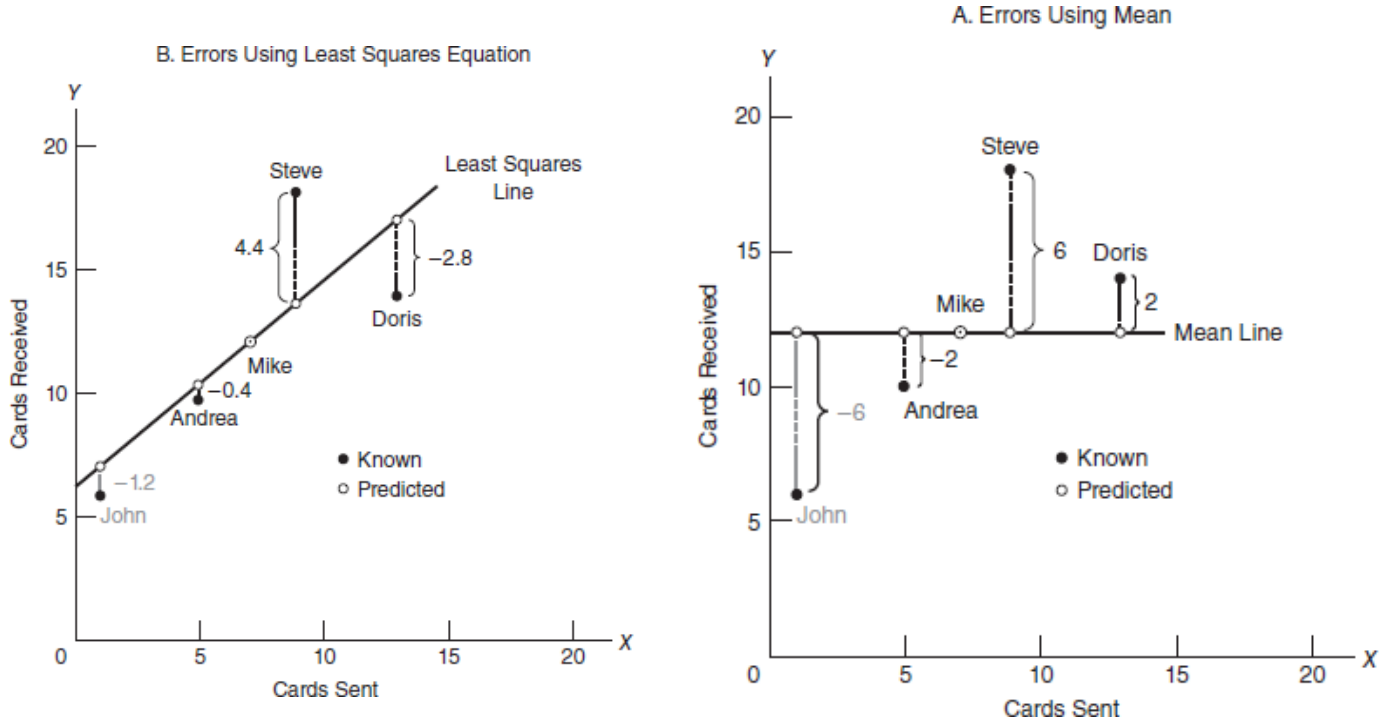


Fig. Predictive errors for five friends

Example

Calculate the standard error of estimate for the given X and Y values. X = 1,2,3,4,5 Y=2,4,5,4,5

Solution

Create five columns labeled x, y, y', y – y', (y – y')2 and N=5

x	y	x^2	xy	Y'= bx+a	y-y'	$(y – y')^2$
1	2	1	2	2.8	-0.8	0.64
2	4	4	8	3.4	0.6	0.36
3	5	9	15	4.0	1	1
4	4	16	16	4.6	-0.6	0.36
5	5	25	25	5.2	-0.2	0.04
Σx:15	**Σy:20**	**$Σx^2$:55**	**Σxy:66**			**$Σ(y – y')^2$ = 2.4**

Note: for finding b value we have to find xy and x^2, so add xy and x^2 column in table

$$b = \frac{N \Sigma(xy) - \Sigma x \Sigma y}{N \Sigma(x^2) - (\Sigma x)^2}$$

$$b = \frac{5(66) - 15x20}{5(55) - (15)^2}$$

$$= \frac{330 - 300}{275 - 225}$$

b= 30/50 = 0.6

$$a = \frac{\Sigma y - b \Sigma x}{N}$$

$$= \frac{20 - (0.6 \times 15)}{5}$$

$$= \frac{20 - 11}{5}$$

a= 9/5 = 2.2

SSy/x = $\sqrt{((y-y')^2 / n-2)}$

$$= \sqrt{(2.4/3)}$$

SSy/x = 0.894

INTERPRETATION OF r^2

R-Squared (R^2 or the coefficient of determination) is a statistical measure in a regression model that determines the proportion of variance in the dependent variable that can be explained by the independent variable. In other words, r-squared shows how well the data fit the regression model (the goodness of fit).

R-squared can take any values between 0 to 1. Although the statistical measure provides some useful insights regarding the regression model, the user should not rely only on the measure in the assessment of a statistical model.

In addition, it does not indicate the correctness of the regression model. Therefore, the user should always draw conclusions about the model by analyzing r-squared together with the other variables in a statistical model.

The most common interpretation of r-squared is how well the regression model explains observed data.

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

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

MULTIPLE REGRESSION EQUATIONS

Multiple regression is a statistical technique applied on datasets dedicated to draw out a relationship between one response or dependent variable and multiple independent variables.

Multiple regression works by considering the values of the available multiple independent variables and predicting the value of one dependent variable.

Example:

A researcher decides to study students' performance from a school over a period of time. He observed that as the lectures proceed to operate online, the performance of students started to decline as well. The parameters for the dependent variable "decrease in performance" are various independent variables like "lack of attention, more internet addiction, neglecting studies" and much more.

Formula to find multiple regression

$$y = b1x1 + b2x2 + \dots bnxn + a$$

REGRESSION TOWARD THE MEAN

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

In statistics, regression toward the mean (also called reversion to the mean, and reversion to mediocrity) is a concept that refers to the fact that if one sample of a random variable is extreme, the next sampling of the same random variable is likely to be closer to its mean.

Example

A military commander has two units return, one with 20% casualties and another with 50% casualties. He praises the first and berates the second. The next time, the two units return with the opposite results. From this experience, he "learns" that praise weakens performance and berating increases performance.

The Regression Fallacy

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

The regression fallacy can be avoided by splitting the subset of extreme observations into two 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

UNIT IV
PYTHON LIBRARIES FOR DATA WRANGLING

Basics of Numpy arrays –aggregations –computations on arrays –comparisons, masks, boolean logic – fancy indexing – structured arrays – Data manipulation with Pandas – data indexing and selection – operating on data – missing data – Hierarchical indexing – combining datasets – aggregation and grouping – pivot tables

NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.

We'll cover a few categories of basic array manipulations here:

Attributes of arrays
Determining the size, shape, memory consumption, and data types of arrays

Indexing of arrays
Getting and setting the value of individual array elements

Slicing of arrays
Getting and setting smaller subarrays within a larger array

Reshaping of arrays
Changing the shape of a given array

Joining and splitting of arrays
Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

- ndim (the number of dimensions),
- shape (the size of each dimension)
- size (the total size of the array)

Example

```
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
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)

print("dtype:", x3.dtype)

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

Array Indexing:

- Accessing Single Elements

Accessing Single Elements

- Indexing in NumPy will feel quite familiar like list indexing,

- In a one-dimensional array, you can access the ith value (counting from zero) by specifying the desired index in square brackets, just as with Python lists
- To index from the end of the array, you can use negative indices
- In a multidimensional array, you access items using a comma-separated tuple of indices
- Unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

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.
The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

x[start:stop:step]
start – starting array index
stop – array index to stop (last value will not be considered)
step – terms has to be printed from start to stop
Default to the values start=0, stop=size of dimension, step=1.

Example

```
x = np.arange(10)
x
```

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

```
x[:5] # prints first five elements
```
array([0, 1, 2, 3, 4])

```
x[5:] # elements after index 5
```
array([5, 6, 7, 8, 9])

```
x[4:7] # middle subarray(from 4th index to 6th index)
```
array([4, 5, 6])

While using negative indices the defaults for start and stop are swapped. This becomes a convenient way to reverse an array

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

```
x[5::-2] # reversed every other from index 5
```
array([5, 3, 1])

Multidimensional sub arrays

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

```
x2
array([[12, 5, 2, 4],
       [ 7, 6, 8, 8],
       [ 1, 6, 7, 7]])
```

```
x2[:2, :3] # two rows, three columns
array([[12, 5, 2],
       [ 7, 6, 8]])

x2[:3, ::2] # all rows, every other column(every second column)
array([[12, 2],
       [ 7, 8],
       [ 1, 7]])
```

Finally, sub array dimensions can even be reversed together

```
x2[::-1, ::-1]
array([[ 7, 7, 6, 1],
       [ 8, 8, 6, 7],
       [ 4, 2, 5, 12]])
```

Reshaping of Arrays

The most flexible way of doing this is with the **reshape()** method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
      [[1 2 3]
      [4 5 6]
      [7 8 9]]
```

Array Concatenation and Splitting

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines **np.concatenate, np.vstack,** and **np.hstack.** np.concatenate takes a tuple or list of arrays as its first argument.

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])

array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once

```
      z = [99, 99, 99]
      print(np.concatenate([x, y, z]))

[ 1 2 3 3 2 1 99 99 99]
```

np.concatenate can also be used for two-dimensional arrays

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
np.concatenate([grid, grid])

array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
```

```
       [4, 5, 6]])
```

Concatenate along the second axis (zero-indexed)

```
np.concatenate([grid, grid], axis=1)

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

np.vstack (vertical stack) functions

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])
p.vstack([x, grid])

array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

np.hstack (horizontal stack) functions

```
y = np.array([[99],
              [99]])
np.hstack([grid, y])

array([[ 9, 8, 7, 99],
       [ 6, 5, 4, 99]])
```

Splitting of arrays

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

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)

[1 2 3] [99 99] [3 2 1]
```

Notice that N split points lead to N + 1 subarrays. The related functions np.hsplit and np.vsplit are similar

```
grid = np.arange(16).reshape((4, 4))
grid
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15]])

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]]
```

```
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]]
```

Computation on NumPy Arrays: Universal Functions

Introducing UFuncs

NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a vectorized operation.

Vectorized operations in NumPy are implemented via ufuncs, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible—before we saw an operation between a scalar and an array, but we can also operate between two arrays

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: unary ufuncs, which operate on a single input, and binary ufuncs, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used.

```
x = np.arange(4)
print("x =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
```

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)

//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function.

- np.abs()
- np.absolute()

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

```
array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs

```
np.absolute(x)
array([2, 1, 0, 1, 2])

np.abs(x)
array([2, 1, 0, 1, 2])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions.

- np.sin()
- np.cos()
- np.tan()

inverse trigonometric functions

- np.arcsin()
- np.arccos()
- np.arctan()

Defining an array of angles: `theta = np.linspace(0, np.pi, 3)`

Compute some trigonometric functions like

```
print("theta = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials.

- np.exp(x) – calculate exponent of all elements in the input array ie e^x (e=2.7182)
- np.exp2(x) – calculate 2**x for all x being the array elements
- np.power(x,y) – calculates the power as x^y

```
x = [1, 2, 3]
print("x =", x)
print("e^x =", np.exp(x))
```

```
print("2^x =", np.exp2(x))
print("3^x =", np.power(3, x))
```

The inverse of the exponentials, the logarithms, are also available. The basic np.log gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm as .

- np.log(x) - is a mathematical function that helps user to calculate Natural logarithm of x where x belongs to all the input array elements
- np.log2(x) - to calculate Base-2 logarithm of x
- np.log10(x) - to calculate Base-10 logarithm of x

```
x = [1, 2, 4, 10]
print("x =", x)
print("ln(x) =", np.log(x))
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
```

Specialized ufuncs

NumPy has many more ufuncs available like

- Hyperbolic trig functions,
- Bitwise arithmetic,
- Comparison operators,
- Conversions from radians to degrees,
- Rounding and remainders, and much more

More specialized and obscure ufuncs is the submodule scipy.special. If you want to compute some obscure mathematical function on your data, chances are it is implemented in scipy.special.

- Gamma function

Advanced Ufunc Features

Specifying output

Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you'd like them to be. For all ufuncs, you can do this using the out argument of the function.

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

[0. 10. 20. 30. 40.]

Aggregates

To reduce an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

```
x = np.arange(1, 6)
np.add.reduce(x)
```

Similarly, calling reduce on the multiply ufunc results in the product of all array elements

```
np.multiply.reduce(x)
```

120

If we'd like to store all the intermediate results of the computation, we can instead use Accumulate

```
np.add.accumulate(x)
array([ 1, 3, 6, 10, 15])
```

Outer products

ufunc can compute the output of all pairs of two different inputs using the outer method. This allows you, in one line, to do things like create a multiplication table.

```
x = np.arange(1, 6)
np.multiply.outer(x, x)

array([[ 1, 2, 3, 4, 5],
       [ 2, 4, 6, 8, 10],
       [ 3, 6, 9, 12, 15],
       [ 4, 8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

Aggregations: Min, Max, and Everything in Between

Minimum and Maximum

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

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

- np.min() – finds the minimum (smallest) value in the array
- np.max() – finds the maximum (largest) value in the array

Example

```
x=[1,2,3,4]
np.min(x)
1
np.max(x)
4
```

Multidimensional aggregates

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

By default, each NumPy aggregation function will return the aggregate over the entire array. ie. If we use the np.sum() it will calculates the sum of all elements of the array.

Example

```
m = np.random.random((3, 4))
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 ]]
```

```
M.sum()
6.0850555667307118
```

Aggregation functions take an additional argument specifying the axis along which the aggregate is computed. The axis normally takes either 0 or 1. if the axis = 0 then it runs along with columns, if axis =1 it runs along with rows.

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

```
M.min(axis=0)
array([ 0.66859307, 0.03783739, 0.19544769, 0.06682827])
```

Similarly, we can find the maximum value within each row

```
M.max(axis=1)
array([ 0.8967576 , 0.99196818, 0.6687194 ])
```

Other aggregation functions

NumPy provides many other aggregation functions most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value.

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

Computation on Arrays: Broadcasting

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

For arrays of the same size, binary operations are performed on an element-by-element basis.

```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

```
array([5, 6, 7])
```

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

```
a + 5
```

```
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place.

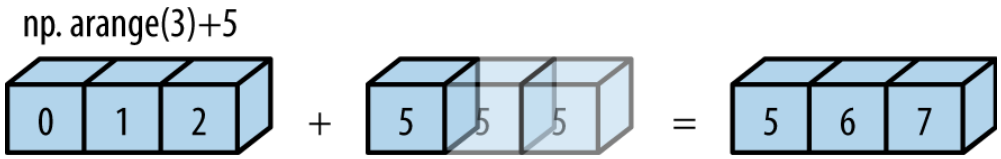
We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array.

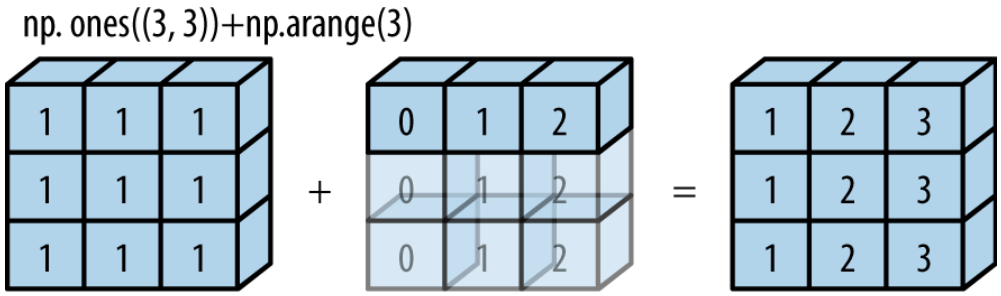
Example

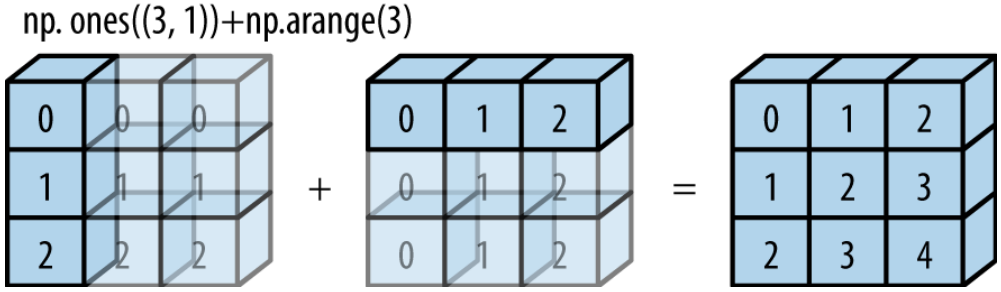
```
M = np.ones((3, 3))
M

array([ [ 1., 1., 1.],
        [ 1., 1., 1.],
        [ 1., 1., 1.]])

M + a

array([[ 1., 2., 3.],
       [ 1., 2., 3.],
       [ 1., 2., 3.]])
```

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

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched both a and b to match a common shape, and the result is a two dimensional array.







The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays.

- 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.

Broadcasting example 1

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

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shapes of the arrays are:

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

We see 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 dimension to match:

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

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

```
M + a
 array([[ 1., 2., 3.],
        [ 1., 2., 3.]])
```

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

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

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

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

And 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)
```

Because the result matches, these shapes are compatible. We can see this here:

```
a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Comparisons, Masks, and Boolean Logic

Comparison Operators as ufuncs.

We saw that using +, -, *, /, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs.
The result of these comparison operators is always an array with a Boolean data type.
All six of the standard comparison operations are available:

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

x <= 3 # less than or equal
array([ True, True, True, False, False], dtype=bool)

x >= 3 # greater than or equal
array([False, False, True, True, True], dtype=bool)

x != 3 # not equal
array([ True, True, False, True, True], dtype=bool)

x == 3 # equal
array([False, False, True, False, False], dtype=bool)
```

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

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example

```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x

array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

```
x < 6

array([[ True, True, True, True],
       [False, False, True, True],
       [ True, True, False, False]], dtype=bool)
```

The result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

- np.count_nonzero()
- np.sum()
- np.sum(x , axis)
- np.any()
- np.all()
- np.all(x , axis)

Boolean operators

Operator	Equivalent ufunc
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Example

```
np.sum((inches > 0.5) & (inches < 1))
inches > (0.5 & inches) < 1
np.sum(~( (inches <= 0.5) | (inches >= 1) ))
```

Boolean Arrays as Masks

A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our x array from before, suppose we want an array of all values in the array that are less than, say, 5

We can obtain a Boolean array for this condition easily, as we've already seen

Example

```
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])

x < 5
array([[False, True, True, True],
       [False, False, True, False],
       [ True, True, False, False]], dtype=bool)
```

Masking operation

To select these values from the array, we can simply index on this Boolean array; this is known as a masking operation.

```
x[x < 5]
```

```
array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True.

Fancy Indexing

Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once.

Types of fancy indexing.

- Indexing / accessing more values
- Array of indices
- In multi dimensional
- Standard indexing

Example

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Indexing / accessing more values

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

```
[x[3], x[7], x[2]]
```

```
[71, 86, 14]
```

Array of indices

We can pass a single list or array of indices to obtain the same result.

```
ind = [3, 7, 4]
x[ind]
```

```
array([71, 86, 60])
```

In multi dimensional

Fancy indexing also works in multiple dimensions. Consider the following array.

```
X = np.arange(12).reshape((3, 4))
X
```

```
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

Standard indexing

Like with standard indexing, the first index refers to the row, and the second to the column.

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
```

```
X[row,  col]

array ([ 2, 5, 11])
```

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

Example array

```
print(X)
        [[ 0 1 2 3]
        [ 4 5 6 7]
        [ 8 9 10 11]]
```

- Combine fancy and simple indices

```
X[2, [2, 0, 1]]
array([10, 8, 9])
```

- Combine fancy indexing with slicing

```
X[1:, [2, 0, 1]]

array([[ 6, 4, 5],
[10, 8, 9]])
```

- Combine fancy indexing with masking

```
mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask]

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.

Change some value in an array

Modify particular element by index

For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value.

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)

[ 0 99 99 3 99 5 6 7 99 9]
```

Using assignment operator

We can use any assignment-type operator for this. For example

```
x[i] -= 10
print(x)
```

```
[ 0 89 89 3 89 5 6 7 89 9]
```

Using at()

Use the at() method of ufuncs for other behavior of modifications.

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
```

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

Sorting Arrays

Sorting in NumPy: np.sort and np.argsort

Python has built-in sort and sorted functions to work with lists, we won't discuss them here because NumPy's np.sort function turns out to be much more efficient and useful for our purposes. By default np.sort uses an O[*N* log *N]*, quicksort algorithm, though mergesort and heapsort are also available. For most applications, the default quicksort is more than sufficient.

Sorting without modifying the input.

To return a sorted version of the array without modifying the input, you can use np.sort

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

```
array([1, 2, 3, 4, 5])
```

Returns sorted indices

A related function is argsort, which instead returns the *indices* of the sorted elements

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

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

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

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

```
np.sort(X, axis=0)
```

```
array([[2, 1, 4, 0, 1, 5],
```

```
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])

np.sort(X, axis=1)

array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the K smallest values in the array. NumPy provides this in the np.partition function. np.partition takes an array and a number K; the result is a new array with the smallest K values to the left of the partition, and the remaining values to the right, in arbitrary order

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)

array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Partitioning in multidimensional array

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array.

```
np.partition(X, 2, axis=1)

array([[3, 4, 6, 7, 6, 9],
       [2, 3, 4, 7, 6, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 9, 5]])
```

Structured Arrays

This section demonstrates the use of NumPy's structured arrays and record arrays, which provide efficient storage for compound, heterogeneous data.

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

Consider if we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays.

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

Creating structured array

NumPy can handle this through structured arrays, which are arrays with compound data types. create a structured array using a compound data type specification as follows.

```
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
print(data.dtype)
```

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

U10 - Unicode string of maximum length 10
i4 - 4-byte (i.e., 32 bit) integer
f8 - 8-byte (i.e., 64 bit) float

Now we can fill the array with our lists of values

```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)('Doug', 19, 61.5)]
```

Refer values through index or name

The handy thing with structured arrays is that you can now refer to values either by index or by name.

```
i. data['name']# by name
```

```
array(['Alice', 'Bob', 'Cathy', 'Doug'],dtype='<U10')
```

```
ii. data[0]# by index
```

```
('Alice', 25, 55.0)
```

Using Boolean masking

This allows to do some more sophisticated operations such as filtering on any fields.

```
data[data['age'] < 30]['name']
```

```
array(['Alice', 'Doug'],dtype='<U10')
```

Creating Structured Arrays

Dictionary method

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})
```

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

Numerical types can be specified with Python types

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})

dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

List of tuples

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])

dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

Specify the types alone

```
np.dtype('S10,i4,f8')

dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

Data Manipulation with Pandas

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame. DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.

Pandas, and in particular its Series and DataFrame objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time.

Here we will focus on the mechanics of using Series, DataFrame, and related structures effectively.

Introducing Pandas Objects

Pandas objects can be thought of as enhanced verions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

Pandas provide a host of useful tools, methods, and functionality on top of the basic data structures.

Three fundamental Pandas data structures: the Series, DataFrame, and Index

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0 0.25
1 0.50
2 0.75
3 1.00
    dtype: float64
```

- **Finding values**

  The values are simply a familiar NumPy array

```
data.values
```

```
array([ 0.25, 0.5 , 0.75, 1. ])
```

- **Finding index**

  The index is an array-like object of type pd.Index

```
data.index
```

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

- **Access by index**
  Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation

```
data[1]

0.5

data[1:3]

1 0.50
2 0.75
    dtype: float64
```

Series as generalized NumPy array

the NumPy array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.
This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type.
For example, if we wish, we can use strings as an index.

Strings as an index

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
data

a 0.25
b 0.50
c 0.75
d 1.00
dtype: float64
```

Noncontiguous or non sequential indices.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=[2, 5, 3, 7])
data

2 0.25
5 0.50
3 0.75
7 1.00
dtype: float64
```

Series as specialized dictionary

A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure that maps typed keys to a set of typed values.
just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

We can make the Series-as-dictionary analogy even more clear by constructing a Series object directly from a Python dictionary.
For example

```
sub1={'sai':90,'ram':85,'kasim':92,'tamil':89}
mark=pd.Series(sub1)
mark
```

```
sai        90
ram        85
kasim      92
tamil      89
dtype: int64
```

Dictionary-style item access

```
Mark['ram']
85
```

Array-style slicing

```
Mark[ 'sai':'kasim']
sai        90
ram        85
kasim      92
```

Constructing Series objects

- **List or NumPy array**

```
pd.Series([2, 4, 6])
```

```
0 2
1 4
2 6
dtype: int64
```

- **Repeated to fill the specified index**

```
pd.Series(5, index=[100, 200, 300])
```

```
100 5
200 5
300 5
dtype: int64
```

- **Data can be a dictionary, in which index defaults to the sorted dictionary keys**

```
pd.Series({2:'a', 1:'b', 3:'c'})
```

```
1 b
2 a
3 c
dtype: object
```

- **The index can be explicitly set if a different result is preferred**

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

```
3 c
2 a
dtype: object
```

The Pandas DataFrame Object

The fundamental structure in Pandas is the DataFrame. The DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

DataFrame as a generalized NumPy array

A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a DataFrame as a sequence of aligned Series objects. Here, by "aligned" we mean that they share the same index.

To demonstrate this, let's first construct a new Series listing the marks of subject2.

```
sub2={'sai':91,'ram':95,'kasim':89,'tamil':90}
```

We can use a dictionary to construct a single two-dimensional object containing this information.

```
result=pd.DataFrame({'DS':sub1,'FDS':sub2})
result
```

```
        DS    FDS
sai     90    91
ram     85    95
kasim 92 89
tamil 89 90
```

DataFrame has an index attribute

Like the Series object, the DataFrame has an index attribute that gives access to the index labels

```
result.index
```

```
Index(['sai', 'ram', 'kasim', 'tamil'], dtype='object')
```

DataFrame has a columns attribute.

The DataFrame has a columns attribute, which is an Index object holding the column labels.

```
result.columns
```

```
Index(['DS', 'FDS'], dtype='object')
```

DataFrame as specialized dictionary

We can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data.

```
result['DS']
```

```
sai 90
ram 85
kasim 92
```

```
tamil 89
Name: DS, dtype: int64
```

Note

In a two-dimensional NumPy array, data[0] will return the first row. For a DataFrame, data['col0'] will return the first column. Because of this, it is probably better to think about DataFrames as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful.

Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

- **From a single Series object.**
- **From a list of dicts.**
- **From a dictionary of Series objects.**
- **From a two-dimensional NumPy array.**
- **From a NumPy structured array.**

From a single Series object.

A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series.

```
sub1=pd.Series({'sai':90,'ram':85,'kasim':92,'tamil':89})
pd.DataFrame(sub1,columns=['DS'])
```

```
          DS
sai       90
ram       85
kasim     92
tamil     89
```

From a list of dicts.

Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data

```
data = [{'a': i, 'b': 2 * i}
for i in range(3)]
pd.DataFrame(data)
```

```
 a b
0 0 0
1 1 2
2 2 4
```

```
Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e.,"not a number") values.

pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
 a b c
0 1.0 2 NaN
1 NaN 3 4.0
```

From a dictionary of Series objects.

As we saw before, a DataFrame can be constructed from a dictionary of Series objects as well.

```
pd.DataFrame({'DS':sub1,'FDS':sub2})
```

```
          DS    FDS
sai       90    91
ram       85    95
kasim 92  89
tamil 89  90
```

From a two-dimensional NumPy array.

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each.

```
pd.DataFrame(np.random.rand(3, 2),
columns=['food', 'water'],
index=['a', 'b', 'c'])
```

```
  food    water
a 0.865257 0.213169
b 0.442759 0.108267
c 0.047110 0.905718
```

From a NumPy structured array.

A Pandas DataFrame operates much like a structured array, and can be created directly.

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
```

```
array([(0, 0.0), (0, 0.0), (0, 0.0)],
dtype=[('A', '<i8'), ('B', '<f8')])
```

```
pd.DataFrame(A)
```

```
  A B
0 0 0.0
1 0 0.0
2 0 0.0
```

The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit index that lets you reference and modify data. This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set.

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

- **Index as immutable array**

The Index object in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices.

```
ind[1]
3
ind[::2]
Int64Index([2, 5, 11], dtype='int64')
```

- **Index as ordered set**

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic.
The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way.

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
indA & indB # intersection

Int64Index([3, 5, 7], dtype='int64')

indA | indB # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

indA ^ indB # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

Data Indexing and Selection

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. It will help us to understand the patterns of data indexing and selection in these arrays.

- Series as dictionary
- Series as one-dimensional array
- Indexers: loc, iloc, and ix

- **Series as dictionary**

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

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
data

a 0.25
b 0.50
c 0.75
d 1.00
dtype: float64

data['b']
```

```
0.5
```

Examine the keys/indices and values

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values

i.
```
'a' in data
```
```
True
```

ii.
```
data.keys()
```
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

iii.
```
list(data.items())
```
```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Modifying series object

Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value.

```
data['e'] = 1.25
data
```

```
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.

Slicing by explicit index

```
data['a':'c']
```

```
a 0.25
b 0.50
c 0.75
dtype: float64
```

Slicing by implicit integer index

```
data[0:2]
```

```
a 0.25
b 0.50
dtype: float64
```

Masking

```
data[(data > 0.3) & (data < 0.8)]
```

```
b 0.50
c 0.75
```

```
dtype: float64
```

Fancy indexing

```
data[['a', 'e']]

a 0.25
e 1.25
dtype: float64
```

- **Indexers: loc, iloc, and ix**

Pandas provides some special indexer attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the Series.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data

1 a
3 b
5 c
dtype: object
```

loc - the loc attribute allows indexing and slicing that always references the explicit index.

```
data.loc[1]
'a'

data.loc[1:3]
1 a
3 b
dtype: object
```

iloc - The iloc attribute allows indexing and slicing that always references the implicit Python-style index.

```
data.iloc[1]
'b'

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

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

Data Selection in DataFrame

- **DataFrame as a dictionary**
- **DataFrame as two-dimensional array**
- **Additional indexing conventions**

DataFrame as a dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects.

The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name.

Dictionary-style indexing of the column name.

```
result=pd.DataFrame({'DS':sub1,'FDS':sub2})
result['DS']
```

```
          DS
sai       90
ram       85
kasim     92
tamil     89
```

Attribute-style access with column names that are strings

```
result.DS
```

```
          DS
sai       90
ram       85
kasim     92
tamil     89
```

Comparing attribute style and dictionary style accesses

```
result.DS is result['DS']
```

```
True
```

Modify the object

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

```
result['TOTAL']=result['DS']+result['FDS']
result
```

	DS	FDS	TOTAL
sai	90	91	181
ram	85	95	180
kasim 92	89	181	
tamil 89	90	179	

DataFrame as two-dimensional array

- **Transpose**

We can transpose the full DataFrame to swap rows and columns.

```
result.T
```

	sai	ram	kasim	tamil
DS	90	85	92	89
FDS	91	95	89	90
TOTAL 181	180	181	179	

Pandas again uses the loc, iloc, and ix indexers mentioned earlier. Using the iloc indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the DataFrame index and column labels are maintained in the result

- **loc**

  result.loc[: 'ram', : 'FDS']

	DS	FDS
sai	90	91
ram	85	95

- **iloc**

  result.iloc[:2, :2]

	DS	FDS
sai	90	91
ram	85	95

- **ix**

  result.ix[:2, :'FDS']

	DS	FDS
sai	90	91
ram	85	95

- **Masking and Fancy indexing**

In the loc indexer we can combine masking and fancy indexing as in the following:

result.loc[result.total>180,['DS', 'FDS']]

	DS	FDS
sai	90	91
kasim 92	89	

- **Modifying values**

Indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy.

result.iloc[1,1] =70

		DS	FDS	TOTAL
sai		90	91	181
ram		85	**70**	180
kasim	92	89	181	
tamil	89	90	179	

Additional indexing conventions

Slicing row wise

result['sai':'kasim']

	DS	FDS	TOTAL
sai	90	91	181
ram	85	**70**	180
kasim 92	89	181	

Such slices can also refer to rows by number rather than by index:

```
result[1:3]
```

	DS	FDS	TOTAL
ram	85	**70**	180
kasim 92	89	181	

Masking row wise

```
result[result.total>180]
```

	DS	FDS	TOTAL
sai	90	91	181
kasim 92	89	181	

Operating on Data in Pandas

Pandas inherits much of this functionality from NumPy, and the ufuncs. So Pandas having the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.).
For unary operations like negation and trigonometric functions, these ufuncs will preserve index and column labels in the output.
For binary operations such as addition and multiplication, Pandas will automatically align indices when passing the objects to the ufunc.
Here we are going to see how the universal functions are working in series and DataFrames by

- **Index preservation**
- **Index alignment**

Index Preservation

Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects. We can use all arithmetic and special universal functions as in NumPy on pandas. In outputs the index will preserved (maintained) as shown below.

For series

```
x=pd.Series([1,2,3,4])
x
```

```
0 1
1 2
2 3
3 4
dtype: int64
```

For DataFrame

```
df=pd.DataFrame(np.random.randint(0,10,(3,4)),
                                columns=['a','b','c','d'])
```

```
df
```

```
   a  b  c  d
0  1  4  1  4
1  8  4  0  4
2  7  7  7  2
```

For universal function. (here we use exponent as example)

Ufuncs for series

```
np.exp(ser)
```

```
0 8103.083928
1 54.598150
2 403.428793
3 20.085537
dtype: float64
```

Ufuncs for Data Frame

```
np.exp(df)
```

	a	b	c	d
0	2.718282	54.598150	2.718282	54.598150
1	2980.957987	54.598150	1.000000	54.598150
2	1096.633158	1096.633158	1096.633158	7.389056

Index Alignment

Pandas will align indices in the process of performing the operation. This is very convenient when you are working with incomplete data, as we'll.

Index alignment in Series

suppose we are combining two different data sources, then the index will aligned accordingly.

```
x=pd.Series([2,4,6],index=[1,3,5])
y=pd.Series([1,3,5,7],index=[1,2,3,4])
x+y
```

```
1 3.0
2 NaN
3 9.0
4 NaN
5 NaN
dtype: float64
```

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.
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 as missing data.

Fill value in missing data (fill_value)
If using NaN values is not the desired behavior, we can modify the fill value using appropriate object methods in place of the operators.

```
x.add(y,fill_value=0)
```

```
1 3.0
2 3.0
3 9.0
4 7.0
5 6.0
dtype: float64
```

Index alignment in DataFrame
A similar type of alignment takes place for both columns and indices when you are performing operations on DataFrames.

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),columns=list('AB'))
A
```

```
 A  B
0 1 11
1 5 1
```

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B
```

```
 B A C
0 4 0 9
1 5 8 0
2 9 2 6
```

```
A + B
```

```
   A    B   C
0  1.0  15.0 NaN
1  13.0 6.0  NaN
2  NaN  NaN  NaN
```

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with Series, we can use the associated object's arithmetic method and pass any desired fill_value to be used in place of missing entries. Here we'll fill with the mean of all values in A.

```
fill = A.stack().mean()
A.add(B, fill_value=fill)

  A   B    C
0 1.0  15.0 13.5
1 13.0 6.0  4.5
2 6.5  13.5 10.5
```

Mapping between Python operators and Pandas methods.

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Operations between Data Frame and Series

When you are performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained. Operations between a DataFrame and a Series are similar to operations between a two-dimensional and one-dimensional NumPy array.

```
A = rng.randint(10, size=(3, 4))
A
array([[3, 8, 2, 4],
[2, 6, 4, 8],
[6, 1, 3, 8]])

A - A[0]
array([[ 0, 0, 0, 0],
[-1, -2, 2, 4],
[ 3, -7, 1, 4]])
```

Handling Missing Data

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 indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its NumPy package, which does not have a built-in notion of NA values for non floating- point data types.

NumPy supports fourteen basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package.

Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floatingpoint NaN value, and the Python None object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic missing data

The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because None is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects)

This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects.

NaN: Missing numerical data

NaN is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation.

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

```
dtype('float64')
```

You should be aware that NaN is a bit like a data virus—it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN

```
1 + np.nan
```

```
nan
```

```
0 * np.nan
```

```
Nan
```

NaN and None in Pandas

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably.

```
pd.Series([1, np.nan, 2, None])
```

```
0 1.0
1 NaN
2 2.0
3 NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to np.nan, it will automatically be upcast to a floating-point type to accommodate the NA

```
x = pd.Series(range(2), dtype=int)
x
0 0
1 1
dtype: int64
x[0] = None
x
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.

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

Note : In Pandas, string data is always stored with an object dtype**.**

Operating on Null Values

there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- isnull() - Generate a Boolean mask indicating missing values
- 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().

isnull()

```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()

0 False
1 True
2 False
3 True
dtype: bool
```

notnull()

```
data.notnull()

0 True
1 False
 True
```

```
3 False
dtype: bool
```

Dropping null values

dropna()

```
data.dropna()
```

```
0 1
2 hello
dtype: object
```

Dropping null values in dataframe

```
df = pd.DataFrame([[1, np.nan, 2],
[2, 3, 5],
[np.nan, 4, 6]])
Df
```

```
 0  1   2
0 1.0 NaN 2
1 2.0 3.0 5
2 NaN 4.0 6
```

```
df.dropna()
```

```
0 1 2
1 2.0 3.0 5
```

Drop values in column or row

We can drop NA values along a different axis; axis=1 drops all columns containing a null value.

```
df.dropna(axis='columns')
```

```
0 2
1 5
2 6
```

Rows or columns having all null values

You can also specify how='all', which will only drop rows/columns that are all null values.

```
df[3] = np.nan
df
```

```
0 1 2 3
0 1.0 NaN 2 NaN
1 2.0 3.0 5 NaN
2 NaN 4.0 6 NaN
```

```
df.dropna(axis='columns', how='all')
```

```
0 1 2
0 1.0 NaN 2
1 2.0 3.0 5
2 NaN 4.0 6
```

Specific no of null values (thresh)

the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept

```
df.dropna(axis='rows', thresh=3)
```

```
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. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the isnull() method as a mask, but because it is such a common operation Pandas provides the fillna() method, which returns a copy of the array with the null values replaced.

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

```
a 1.0
b NaN
c 2.0
d NaN
```

Fill with single value

We can fill NA entries with a single value, such as zero

```
data.fillna(0)
```

```
a 1.0
b 0.0
c 2.0
d 0.0
e 3.0
dtype: float64
```

Fill with previous value

We can specify a forward-fill to propagate the previous value forward

```
data.fillna(method='ffill')
```

```
a 1.0
b 1.0
c 2.0
d 2.0
e 3.0
dtype: float64
```

Fill with next value

We can specify a back-fill to propagate the next values backward.

```
data.fillna(method='bfill')
```

```
a 1.0
b 2.0
c 2.0
d 3.0
e 3.0
dtype: float64
```

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and twodimensional data, stored in Pandas Series and DataFrame objects, respectively. Often it is useful to go beyond this and store higher-dimensional data—that is, data indexed by more than one or two keys.

Pandas does provide Panel and Panel4D objects that natively handle three-dimensional and four-dimensional, 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.

In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

Here we'll explore the direct creation of MultiIndex objects; considerations around indexing, slicing, and computing statistics across multiply indexed data; and useful routines for converting between simple and hierarchically indexed representations of your data.

A Multiply Indexed Series

Pandas MultiIndex

Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndex type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows

```
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
```

```
(California, 2000) 33871648
(California, 2010) 37253956
(New York, 2000) 18976457
(New York, 2010) 19378102
(Texas, 2000) 20851820
```

```
#creating multi index
index = pd.MultiIndex.from_tuples(index)
index
```

```
MultiIndex(levels=[['California','New York','Texas'],[2000, 2010]],
labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Hierarchical representation of the data

```
pop = pop.reindex(index)
pop
```

```
California 2000 33871648
                2010 37253956
New York        2000 18976457
                2010 19378102
Texas    2000 20851820
                2010 25145561
dtype: int64
```

Here the first two columns of the Series representation show the multiple index values, while the third column shows the data.

Access all data with second index

```
pop[:, 2010]
```

```
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.

```
pop_df = pop.unstack()
pop_df
```

```
                2000  2010
California 33871648 37253956
New York        18976457 19378102
Texas    20851820 25145561
```

The **stack()** method provides the opposite operation.

```
pop_df.stack()
```

```
California 2000 33871648
2010 37253956
New York 2000 18976457
2010 19378102
Texas 2000 20851820
```

```
2010 25145561
dtype: int64
```

Add a new column in multi dimensional data frame.

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

```
                          total    under18
California      2000 33871648 9267089
                2010 37253956 9284094
New York        2000 18976457 4687374
                2010 19378102 4318033
Texas           2000 20851820 5906301
                2010 25145561 6879014
```

Universal functions

All the ufuncs and other functionality work with hierarchical indices.

```
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

```
                2000    2010
California      0.273594        0.249211
New York        0.247010        0.222831
Texas           0.283251        0.273568
```

Methods of Multi Index Creation

To construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor.

```
df = pd.DataFrame(np.random.rand(4, 2),
index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
columns=['data1', 'data2'])
df
```

```
data1 data2
a 1 0.554233 0.356072
2 0.925244 0.219474
b 1 0.441759 0.610054
2 0.171495 0.886688
```

if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default.

```
data = {        ('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
```

```
                ('New York', 2010): 19378102}
pd.Series(data)

California      2000 33871648
                2010 37253956
New York        2000 18976457
                2010 19378102
Texas           2000 20851820
                2010 25145561
dtype: int64
```

Explicit MultiIndex constructors

You can construct the MultiIndex from a simple list of arrays, giving the index values within each level.

```
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

MultiIndex(levels=[['a', 'b'], [1, 2]],
labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Multi index from a list of tuples,

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Multi index from Cartesian product.

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

MultiIndex level names

It is convenient to name the levels of the MultiIndex. You can accomplish this by passing the names argument to any of the above MultiIndex constructors, or by setting the names attribute of the index after the fact.

```
pop.index.names = ['state', 'year']
pop

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.

```
# 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
```

```
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

Indexing and slicing on a MultiIndex is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed Series, and then multiply indexed DataFrames.

Multiply indexed Series

```
        Pop
```

```
state            year
California       2000 33871648
                      2010 37253956
New York               2000 18976457
                      2010 19378102
Texas            2000 20851820
                      2010 25145561
dtype: int64
```

- Access single elements
  We can access single elements by indexing with multiple terms

```
        pop['California', 2000]
```

```
        33871648
```

- Partial indexing
  The MultiIndex also supports partial indexing, or indexing just one of the levels in the index

```
        pop['California']
```

```
year
2000 33871648
2010 37253956
dtype: int64
```

- Partial slicing
  Partial slicing is available as well, as long as the MultiIndex is sorted.

```
pop.loc['California':'New York']
```

```
state       year
California  2000 33871648
            2010 37253956
New York    2000 18976457
            2010 19378102
dtype: int64
```

- Sorted indices
  With sorted indices, we can perform partial indexing on lower levels by passing an empty slice in the first index

```
pop[:, 2000]
```

```
state
California  33871648
New York    18976457
Texas       20851820
dtype: int64
```

- Other types of indexing and selection
  Selection based on Boolean masks

```
pop[pop > 22000000]
```

```
state       year
California  2000 33871648
            2010 37253956
Texas       2010 25145561
dtype: int64
```

Selection based on fancy indexing

```
pop[['California', 'Texas']]
```

```
state       year
California  2000 33871648
            2010 37253956
Texas       2000 20851820
            2010 25145561
dtype: int64
```

Rearranging Multi-Indices

We saw a brief example of this in the stack() and unstack() methods, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

- **Sorted and unsorted indices**

We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
```

```
char int
a    1    0.003001
     2    0.164974
c    1    0.741650
```

Pandas provides a number of convenience routines to perform this type of sorting; examples are the sort_index() and sortlevel() methods of the DataFrame. We'll use the simplest, sort_index(), here:

```
data = data.sort_index()
data
```

```
char int
a    1    0.003001
     2    0.164974
b    1    0.001693
     2    0.526226
c    1    0.741650
     2    0.569264
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
data['a':'b']
```

```
char int
a    1    0.003001
     2    0.164974
b    1    0.001693
     2    0.526226
dtype: float64
```

- **Stacking and unstacking indices**

it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use.

```
pop.unstack(level=0)
```

```
state California New York Texas
year
2000 33871648 18976457 20851820
2010 37253956 19378102 25145561
```

```
pop.unstack(level=1)
```

```
year            2000    2010
state
California      33871648 37253956
New York              18976457 19378102
Texas           20851820 25145561
```

The opposite of unstack() is stack(), which here can be used to recover the original series:

```
pop.unstack().stack()
```

```
state year
California      2000 33871648
                2010 37253956
New York              2000 18976457
                2010 19378102
Texas           2000 20851820
                2010 25145561
dtype: int64
```

- **Index setting and resetting**

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the reset_index method. Calling this on the population dictionary will result in a DataFrame with a state and year column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation.

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

```
        state    year population
0 California    2000 33871648
1 California    2010 37253956
2 New York      2000 18976457
3 New York      2010 19378102
4 Texas               2000 20851820
5 Texas               2010 25145561
```

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as mean(), sum(), and max(). For hierarchically indexed data, these can be passed a level parameter that controls which subset of the data the aggregate is computed on.

For example, let's return to our health data: (you can create your own data frame / series)

```
health_data
```

```
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
```

Calculate the average as follows

```
data_mean = health_data.mean(level='year')
data_mean
```

```
subject          Bob              Guido Sue
type     HR Temp         HR Temp         HR Temp
year
2013     37.5 38.2 41.0 35.85 32.0 36.95
2014     38.5 37.6 43.5 37.55 56.0 36.70
```

By further making use of the axis keyword, we can take the mean among levels on the columns as well:

```
data_mean.mean(axis=1, level='type')
```

```
type     HR               Temp
year
2013     36.833333 37.000000
2014     46.000000 37.283333
```

Combining Datasets

Concat and Append

Simple Concatenation with pd.concat

Pandas has a function, pd.concat(), which has a similar syntax to np.concatenate but contains a number of options that we'll discuss momentarily

pd.concat() can be used for a simple concatenation of Series or DataFrame objects, just as np.concatenate() can be used for simple concatenations of arrays

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1 A
2 B
3 C
4 D
5 E
6 F
dtype: object
```

Concatenation in data frame.

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
print(df1); print(df2); print(pd.concat([df1, df2]))
```

```
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

One important difference between np.concatenate and pd.concat is that Pandas concatenation preserves indices, even if the result will have duplicate indices! Consider this simple example.

```
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
```

The append() method

Series and DataFrame objects have an append method that can accomplish the same thing in fewer keystrokes. For example, rather than calling pd.concat([df1, df2]), you can simply call df1.append(df2):

```
        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
```

Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations.

Categories of Joins

- One-to-one joins
- Many-to-one joins
- Many-to-many joins

One – to – one joins

The simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation.

```
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
```

To combine this information into a single DataFrame, we can use the pd.merge() function

```
df3 = pd.merge(df1, df2)
df3
```

```
      employee      group          hire_date
0     Bob           Accounting     2008
1     Jake    Engineering     2012
2     Lisa    Engineering     2004
3     Sue           HR                     2014
```

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate.

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                                'supervisor': ['Carly', 'Guido', 'Steve']})
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
```

The resulting DataFrame has an additional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example.

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'], 'skills':
['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', '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

Aggregation and Grouping

Computing aggregations like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

Simple Aggregation in Pandas

As with a one dimensional NumPy array, for a Pandas Series the aggregates return a single value.

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

```
0 0.374540
1 0.950714
2 0.731994
3 0.598658
4 0.156019
dtype: float64
```

Sum

```
ser.sum()
```

```
2.8119254917081569
```

Mean

```
ser.mean()
```

```
0.56238509834163142
```

The same operations also performed in DataFrame

Listing of Pandas aggregation methods

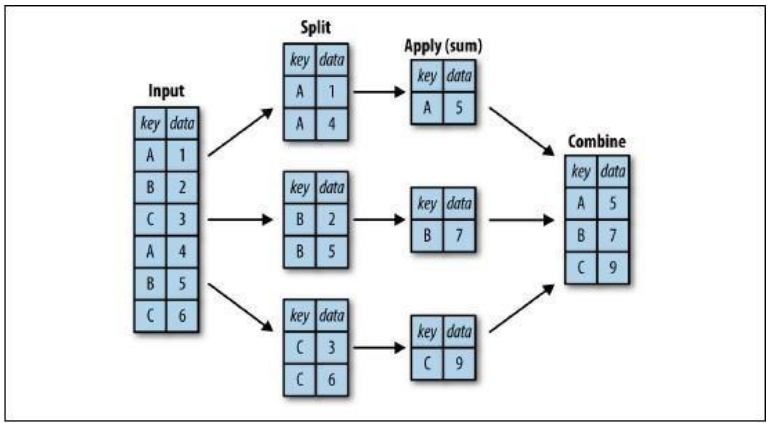
Aggregation	Description
count()	Total number of items
first(), last()	First and last item
mean(), median()	Mean and median
min(), max()	Minimum and maximum
std(), var()	Standard deviation and variance
mad()	Mean absolute deviation
prod()	Product of all items
sum()	Sum of all items

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the socalled groupby operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: split, apply, combine.

- The split step involves breaking up and grouping a DataFrame depending on the value of the specified key.
- The apply step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

• The combine step merges the results of these operations into an output array.



Example

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
Df
```

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The GroupBy object

The GroupBy object is a very flexible abstraction. The most important operations made available by a GroupBy are aggregate, filter, transform, and apply.

Groupby supports the basic operations like.

- Column indexing.
- Iteration over groups.
- Dispatch methods.
- Aggregate, filter, transform, apply

Column indexing.

The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object. For example

```
df=pd.read_csv('D:\iris.csv')
df.groupby('variety')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at
0x0000023BAADE84C0>
```

```
df.groupby(' variety)[' petal.length'']
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x0000023BAADE8490>
```



```
df.groupby(' variety ')[ “petal.length'']].sum()
```

```
     variety Setosa 73.1
     Versicolor     213.0
     Virginica      277.6
     Name: petal.length, dtype: float64
```

Iteration over groups.
The GroupBy object supports direct iteration over the groups, returning each group as a Series or DataFrame.
This can be useful for doing certain things manually, though it is often much faster to use the built-in apply functionality, which we will discuss momentarily.

Dispatch methods.
Through some Python class magic, any method not explicitly implemented by the GroupBy object will be passed through and called on the groups, whether they are DataFrame or Series objects. For example, you can use the describe() method of DataFrames to perform a set of aggregations that describe each group in the data.

Example

```
df.groupby('variety')['petal.length'].describe().unstack()
```

```
     variety
count      Setosa    50.000000
           Versicolor     50.000000
           Virginica      50.000000
mean       Setosa         1.462000
           Versicolor     4.260000
           Virginica      5.552000
std        Setosa         0.173664
           Versicolor     0.469911
           Virginica      0.551895
min        Setosa         1.000000
           Versicolor     3.000000
           Virginica      4.500000
25%        Setosa         1.400000
           Versicolor     4.000000
           Virginica      5.100000
50%        Setosa         1.500000
           Versicolor     4.350000
           Virginica      5.550000
75%        Setosa         1.575000
           Versicolor     4.600000
           Virginica      5.875000
max        Setosa         1.900000
           Versicolor     5.100000
           Virginica      6.900000
dtype: float64
```

Aggregate, filter, transform,and apply

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
```

```
'data1': range(6),
'data2': rng.randint(0, 10, 6)},
columns = ['key', 'data1', 'data2'])
df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation.

We're now familiar with GroupBy aggregations with sum(), median(), and the like, but the aggregate() method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once. Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Filtering.

A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value.

The filter() function should return a Boolean value specifying whether the group passes the filtering.

Transformation.

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

The apply() method.

The apply() method lets you apply an arbitrary function to the group results. The function should take a DataFrame, and return either a Pandas object (e.g., DataFrame, Series) or a scalar; the combine operation will be tailored to the type of output returned.

Pivot Tables

A pivot table is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple column wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a multidimensional version of GroupBy aggregation. That is, you split apply- combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Pivot Table Creation

```
import numpy as np
import pandas as pd
df=pd.read_csv('D:\diabetes.csv')
df.pivot_table('preg',index='age',columns='Class').sample(10)
```

#here diabetes data set has large no of rows so we use sample()

Class	tested_negative	tested_positive
age		
63	5.500000	NaN
28	3.440000	2.000000
61	7.000000	4.000000
69	5.000000	NaN
45	7.285714	7.375000
62	6.500000	1.000000
53	2.000000	6.250000
68	8.000000	NaN
23	1.516129	1.857143

Class	tested_negative	tested_positive
age		
52	13.000000	3.428571

UNIT V
DATA VISUALIZATION

Importing Matplotlib – Line plots – Scatter plots – visualizing errors – density and contour plots – Histograms – legends – colors – subplots – text and annotation – customization – three dimensional plotting - Geographic Data with Basemap - Visualization with Seaborn.

Simple Line Plots

The simplest of all plots is the visualization of a single function y = f x . Here we will take a first look at creating a simple plot of this type.

The *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.

The *axes* (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

Line Colors and Styles

- The first adjustment you might wish to make to a plot is to control the line colors and styles.
- To adjust the color, you can use the color keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways
- If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines

Different forms of color representation.

specify color by name	- color='blue'
short color code (rgbcmyk)	- color='g'
Grayscale between 0 and 1	- color='0.75'
Hex code (RRGGBB from 00 to FF)	- color='#FFDD44'
RGB tuple, values 0 and 1	- color=(1.0,0.2,0.3)
all HTML color names supported	- color='chartreuse'

- We can adjust the line style using the linestyle keyword.

  Different line styles

```
linestyle='solid'
linestyle='dashed'
linestyle='dashdot'
linestyle='dotted'
```

  Short assignment

```
linestyle='-' # solid
linestyle='--' # dashed
linestyle='-.' # dashdot
linestyle=':' # dotted
```

- linestyle and color codes can be combined into a single nonkeyword argument to the plt.plot() function

```
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
```

Axes Limits

- The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methods

Example

```
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```

- The plt.axis() method allows you to set the x and y limits with a single call, by passing a list that specifies [xmin, xmax, ymin, ymax]

```
plt.axis([-1, 11, -1.5, 1.5]);
```

- Aspect ratio equal is used to represent one unit in x is equal to one unit in y. `plt.axis('equal')`

Labeling Plots

The labeling of plots includes titles, axis labels, and simple legends.

Title - `plt.title()`

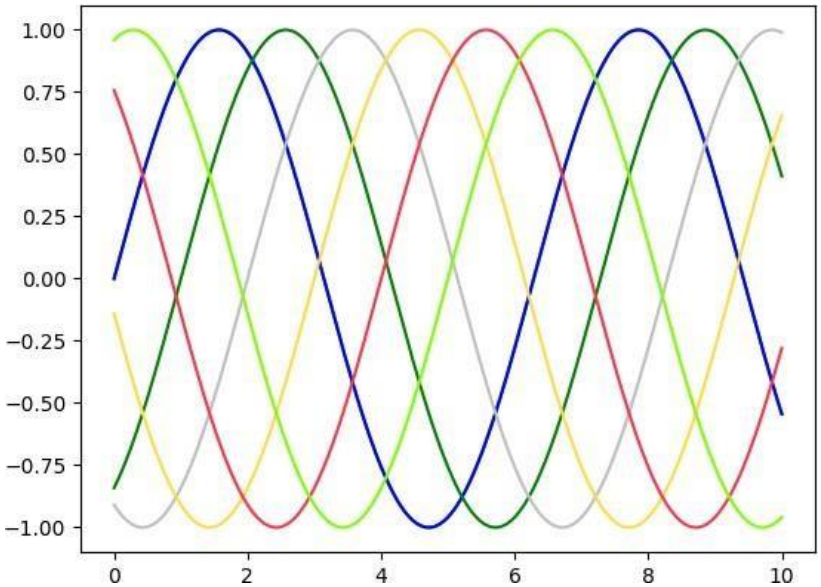
Label - `plt.xlabel()`
`plt.ylabel()`

Legend - `plt.legend()`

Example programs

Line color

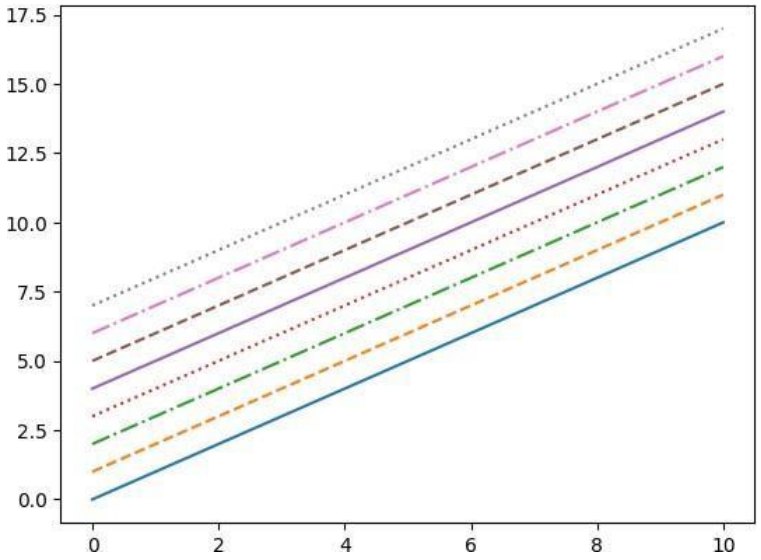
```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
plt.plot(x, np.sin(x - 0), color='blue') # specify color by name
plt.plot(x, np.sin(x - 1), color='g') # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), 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
```



Line style

```
import matplotlib.pyplot as plt
```

```
import numpy as np
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
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
```



Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

Syntax

```
plt.plot(x, y, 'type of symbol ', color);
```

Example

```
plt.plot(x, y, 'o', color='black');
```

- The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-' and '--' to control the line style, the marker style has its own set of short string codes.

Example

- Various symbols used to specify ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']

- Short hand assignment of line, symbol and color also allowed.

```
plt.plot(x, y, '-ok');
```

- Additional arguments in plt.plot()
  We can specify some other parameters related with scatter plot which makes it more attractive. They are color, marker size, linewidth, marker face color, marker edge color, marker edge width, etc

Example

```
plt.plot(x, y, '-p', color='gray',
markersize=15, linewidth=4,
markerfacecolor='white',
markeredgecolor='gray',
markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

Scatter Plots with plt.scatter

- A second, more powerful method of creating scatter plots is the plt.scatter function, which can be used very similarly to the plt.plot function
  ```
  plt.scatter(x, y, marker='o');
  ```
- The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.
- Notice that the color argument is automatically mapped to a color scale (shown here by the colorbar() command), and the size argument is given in pixels.
- Cmap – color map used in scatter plot gives different color combinations.

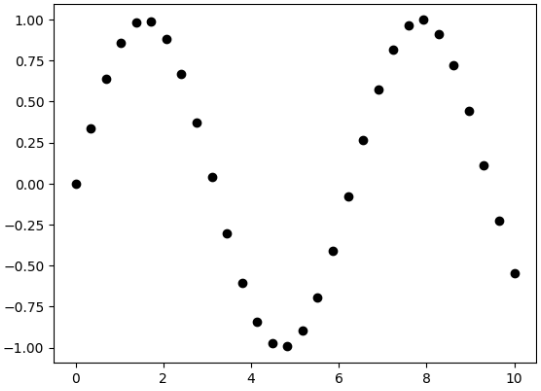
```
Perceptually Uniform Sequential
['viridis', 'plasma', 'inferno', 'magma']
Sequential
['Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds', 'YlOrBr', 'YlOrRd',
'OrRd', 'PuRd', 'RdPu', 'BuPu', 'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn',
'YlGn']
Sequential (2)
['binary', 'gist_yarg', 'gist_gray', 'gray', 'bone', 'pink', 'spring', 'summer',
'autumn', 'winter', 'cool', 'Wistia', 'hot', 'afmhot', 'gist_heat', 'copper']
```

```
Diverging
['PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu', 'RdYlBu', 'RdYlGn', 'Spectral',
'coolwarm', 'bwr', 'seismic']
Qualitative
['Pastel1', 'Pastel2', 'Paired', 'Accent', 'Dark2', 'Set1', 'Set2', 'Set3',
'tab10', 'tab20', 'tab20b', 'tab20c']
Miscellaneous
['flag', 'prism', 'ocean', 'gist_earth', 'terrain', 'gist_stern', 'gnuplot',
'gnuplot2', 'CMRmap', 'cubehelix', 'brg', 'hsv', 'gist_rainbow', 'rainbow',
'jet', 'nipy_spectral', 'gist_ncar']
```

Example programs.

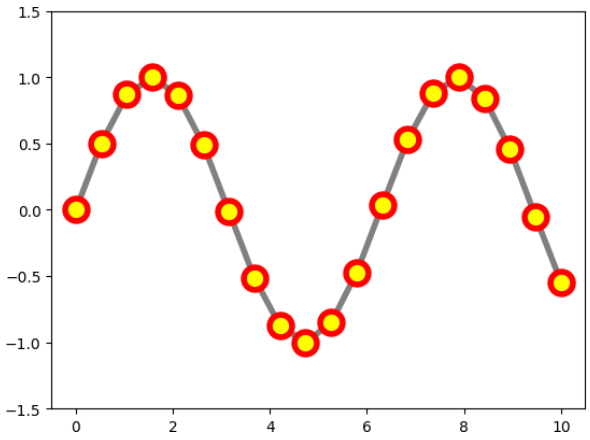
Simple scatter plot.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black');
```



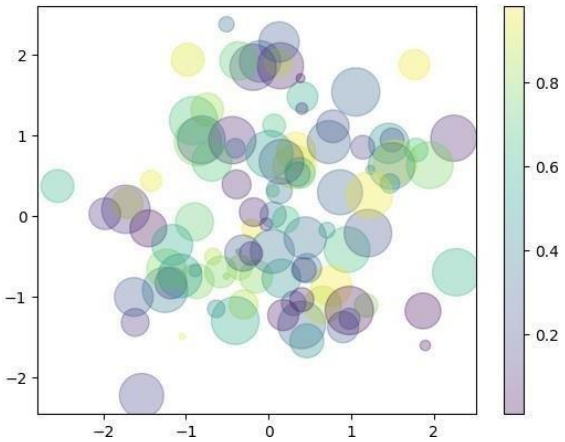
Scatter plot with edge color, face color, size, and width of marker. (Scatter plot with line)

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 20)
y = np.sin(x)
plt.plot(x, y, '-o', color='gray',
markersize=15, linewidth=4,
markerfacecolor='yellow',
markeredgecolor='red',
markeredgewidth=4)
plt.ylim(-1.5, 1.5);
```



Scatter plot with random colors, size and transparency

```
import numpy as np
import matplotlib.pyplot as plt
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, map='viridis')
plt.colorbar()
```



Visualizing Errors

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself. For example, imagine that I am using some astrophysical observations to estimate the Hubble Constant, the local measurement of the expansion rate of the Universe.
In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

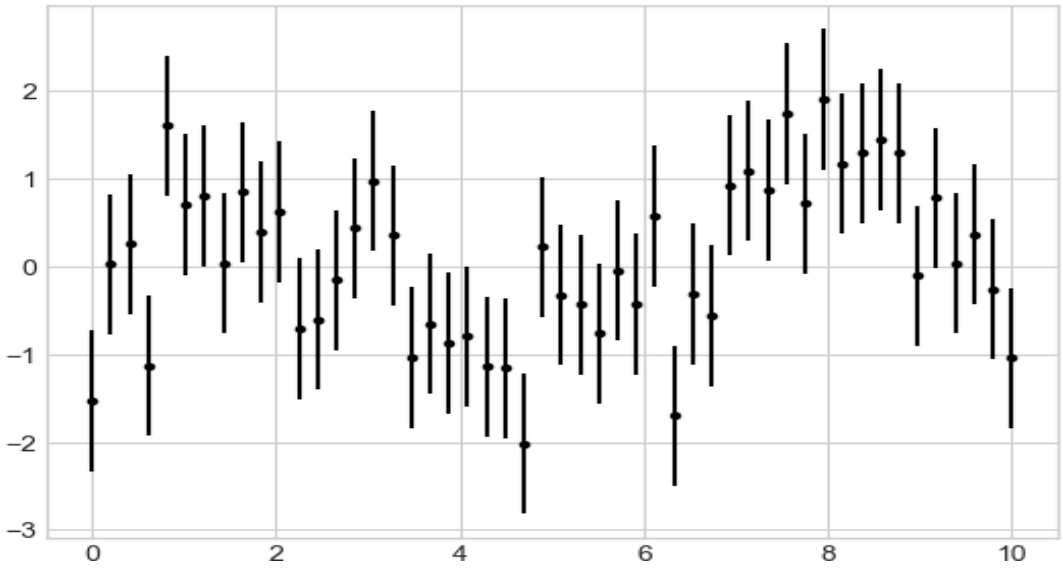
Types of errors

- Basic Errorbars
- Continuous Errors

Basic Errorbars

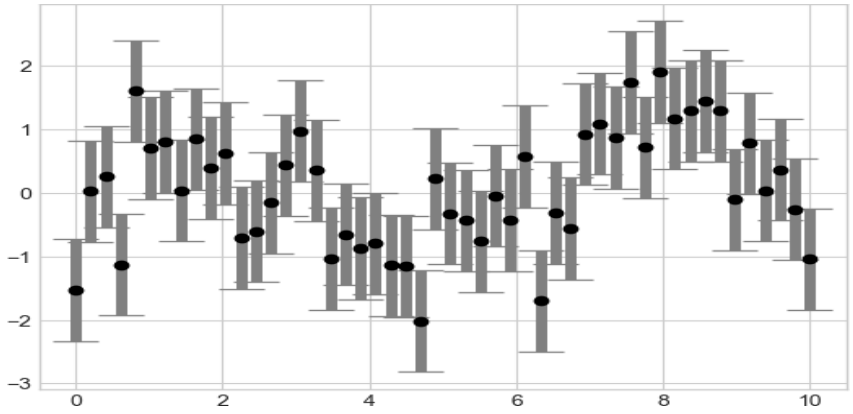
A basic errorbar can be created with a single Matplotlib function call.

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



- Here the fmt is a format code controlling the appearance of lines and points, and has the same syntax as the shorthand used in plt.plot()
- In addition to these basic options, the errorbar function has many options to fine tune the outputs. Using these additional options you can easily customize the aesthetics of your errorbar plot.

```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',ecolor='lightgray', elinewidth=3, capsize=0);
```



Continuous Errors

- In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like plt.plot and plt.fill_between for a useful result.
- Here we'll perform a simple Gaussian process regression (GPR), using the Scikit-Learn API. This is a method of fitting a very flexible nonparametric function to data with a continuous measure of the uncertainty.

Density and Contour Plots

To display three-dimensional data in two dimensions using contours or color-coded regions.
There are three Matplotlib functions that can be helpful for this task:

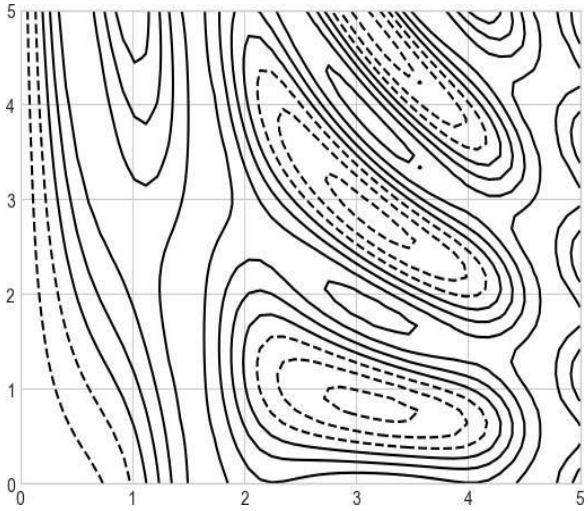
- plt.contour for contour plots,
- plt.contourf for filled contour plots, and
- plt.imshow for showing images.

Visualizing a Three-Dimensional Function

A contour plot can be created with the plt.contour function. It takes three arguments:

- a grid of x values,
- a grid of y values, and
- a grid of z values.

The x and y values represent positions on the plot, and the z values will be represented by the contour levels.
The way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays:



Example

```
def f(x, y):
        return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='black');
```

- Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.
- Alternatively, you can color-code the lines by specifying a colormap with the cmap argument.
- We'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range.

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

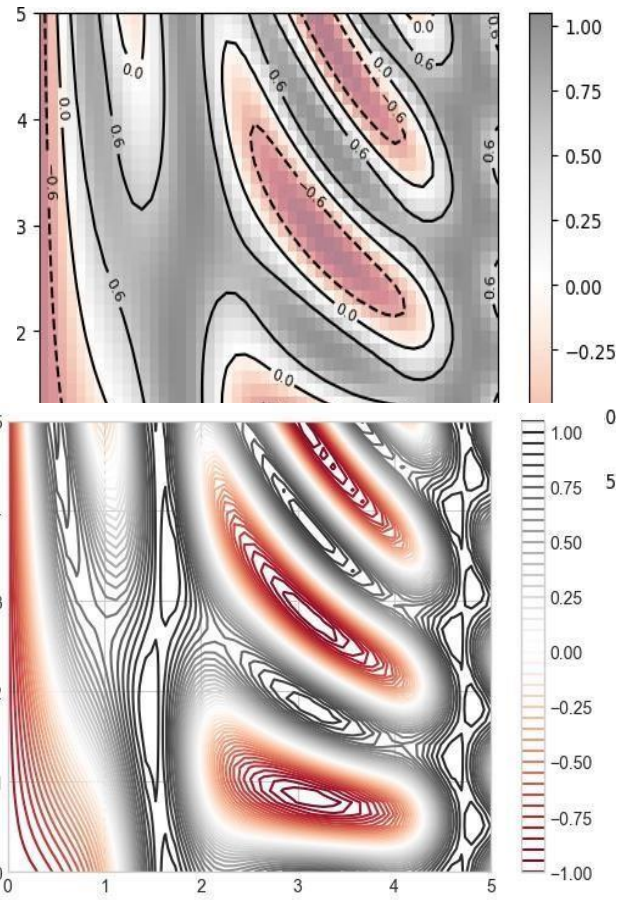
- One potential issue with this plot is that it is a bit "splotchy." That is, the color steps are discrete rather than continuous, which is not always what is desired.
- You could remedy this by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level.
- A better way to handle this is to use the plt.imshow() function, which interprets a two-dimensional grid of data as an image.

There are a few potential gotchas with imshow().

- plt.imshow() doesn't accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin, ymax] of the image on the plot.
- plt.imshow() by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
- plt.imshow() will automatically adjust the axis aspect ratio to match the input data; you can change this by setting, for example, plt.axis(aspect='image') to make x and y units match.

Finally, it can sometimes be useful to combine contour plots and image plots. we'll use a partially transparent background image (with transparency set via the alpha parameter) and over-plot contours with labels on the contours themselves (using the plt.clabel() function):

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
cmap='RdGy', alpha=0.5)
plt.colorbar();
```



Example Program

```
import numpy as np
import matplotlib.pyplot as
pltdef f(x, y):
return np.sin(x) ** 10 + np.cos(10 + y * x) *
np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.imshow(Z, extent=[0, 10, 0, 10],
origin='lower', cmap='RdGy')
plt.colorbar()
```

Histograms

- Histogram is the simple plot to represent the large data set. A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval.

Parameters

- plt.hist( ) is used to plot histogram. The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

- bins - A histogram displays numerical data by grouping data into "bins" of equal width. Each bin is plotted as a bar whose height corresponds to how many data points are in that bin. Bins are also sometimes called "intervals", "classes", or "buckets".
- normed - Histogram normalization is a technique to distribute the frequencies of the histogram over a wider range than the current range.
- x - (n,) array or sequence of (n,) arrays Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.
- histtype - {'bar', 'barstacked', 'step', 'stepfilled'}, optional
  The type of histogram to draw.

  - 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
  - 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
  - 'step' generates a lineplot that is by default unfilled.
  - 'stepfilled' generates a lineplot that is by default filled.

  Default is 'bar'
- align - {'left', 'mid', 'right'}, optional
  Controls how the histogram is plotted.

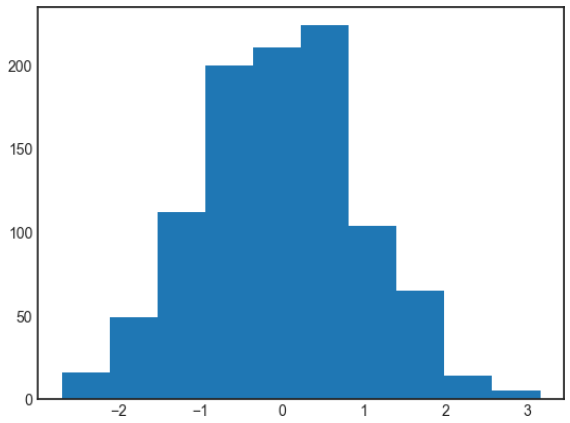
  - 'left': bars are centered on the left bin edges.
  - 'mid': bars are centered between the bin edges.
  - 'right': bars are centered on the right bin edges.

  Default is 'mid'
- orientation - {'horizontal', 'vertical'}, optional
  If 'horizontal', barh will be used for bar-type histograms and the bottom kwarg will be the left edges.
- color - color or array_like of colors or None, optional
  Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

  Default is None
- label - str or None, optional. Default is None

Other parameter

- **kwargs - Patch properties, it allows us to pass a variable number of keyword arguments to a python function. ** denotes this type of function.



Example

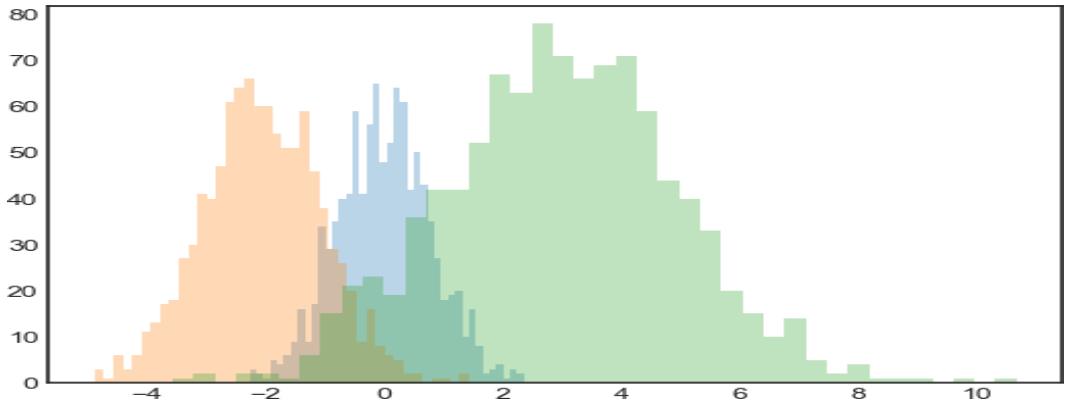
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
data = np.random.randn(1000)
plt.hist(data);
```

The hist() function has many options to tune both the calculation and the display; here's an example of a more customized histogram.

```
plt.hist(data, bins=30, alpha=0.5,histtype='stepfilled', color='steelblue',edgecolor='none');
```

The plt.hist docstring has more information on other customization options available. I find this combination of histtype='stepfilled' along with some transparency alpha to be very useful when comparing histograms of several distributions

```
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, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

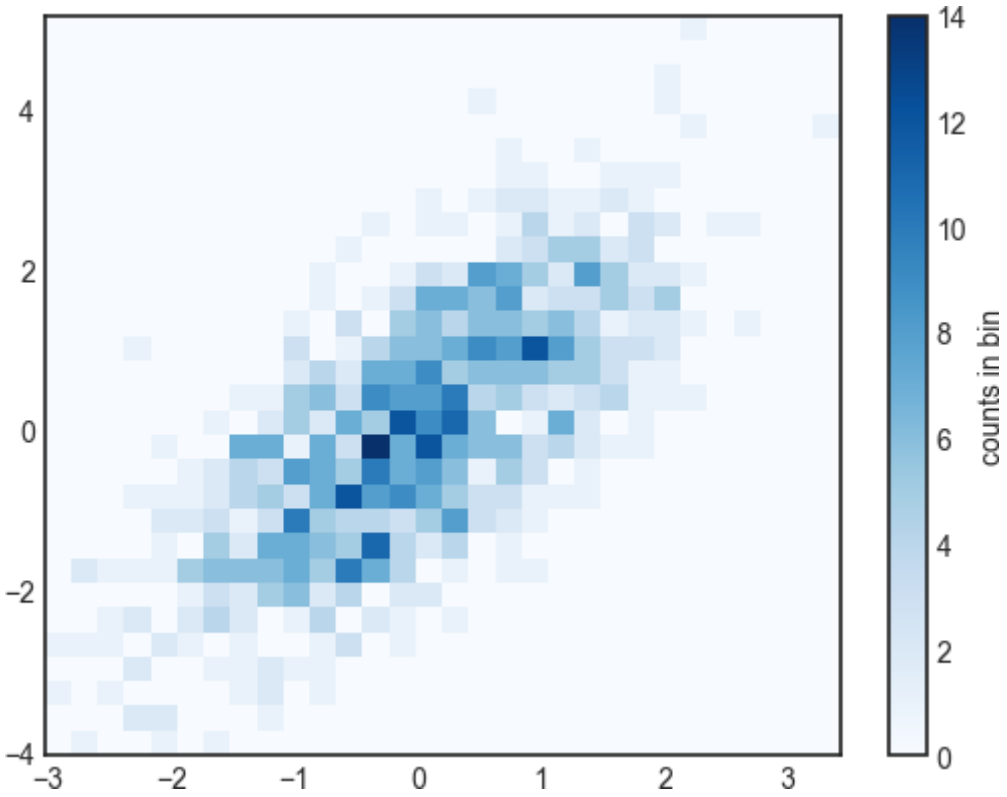


Two-Dimensional Histograms and Binnings

- We can create histograms in two dimensions by dividing points among two dimensional bins.
- We would define x and y values. Here for example We'll start by defining some data—an x and y array drawn from a multivariate Gaussian distribution:
- Simple way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2d() function

Example

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 1000).T
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib

```
plt.plot(x, np.sin(x), '-b', label='Sine')
plt.plot(x, np.cos(x), '--r', label='Cosine')
plt.legend();
```

Customizing Plot Legends

Location and turn off the frame - We can specify the location and turn off the frame. By the parameter loc and framon.

```
ax.legend(loc='upper left', frameon=False)
fig
```

Number of columns - We can use the ncol command to specify the number of columns in the legend.

```
ax.legend(frameon=False, loc='lower center', ncol=2)
fig
```

Rounded box, shadow and frame transparency

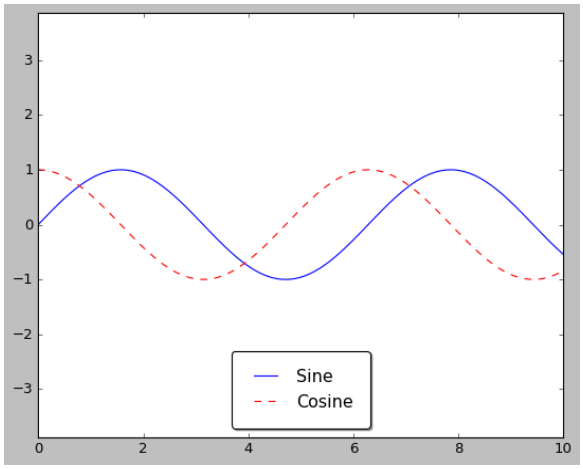
We can use a rounded box (fancybox) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text.

```
ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
fig
```

Choosing Elements for the Legend

- The legend includes all labeled elements by default. We can change which elements and labels appear in the legend by using the objects returned by plot commands.
- The plt.plot() command is able to create multiple lines at once, and returns a list of created line instances. Passing any of these to plt.legend() will tell it which to identify, along with the labels we'd like to specify

```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
lines = plt.plot(x, y)
plt.legend(lines[:2],['first','second']);

# Applying label individually.
plt.plot(x, y[:, 0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])
plt.legend(framealpha=1, frameon=True);
```



Multiple legends

It is only possible to create a single legend for the entire plot. If you try to create a second legend using plt.legend() or ax.legend(), it will simply override the first one. We can work around this by creating a new legend artist from scratch, and then using the lower-level ax.add_artist() method to manually add the second artist to the plot

Example

```
import matplotlib.pyplot as plt
plt.style.use('classic')
import numpy as np
x = np.linspace(0, 10, 1000)
ax.legend(loc='lower center', frameon=True, shadow=True,borderpad=1,fancybox=True)
fig
```

Color Bars

In Matplotlib, a color bar is a separate axes that can provide a key for the meaning of colors in a plot. For continuous labels based on the color of points, lines, or regions, a labeled color bar can be a great tool.
The simplest colorbar can be created with the plt.colorbar() function.

Customizing Colorbars

Choosing color map.

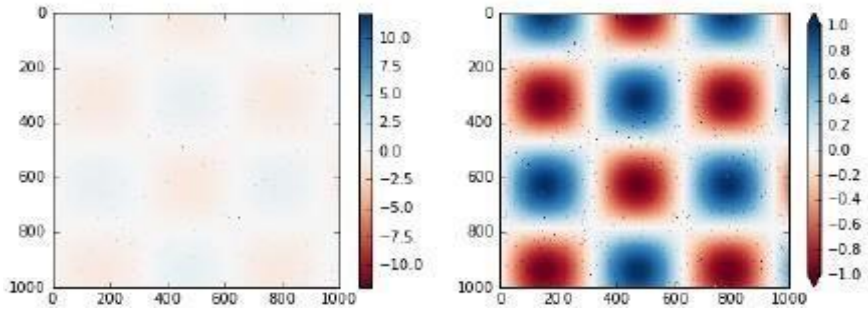
We can specify the colormap using the cmap argument to the plotting function that is creating the visualization. Broadly, we can know three different categories of colormaps:

- **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., RdBu or PuOr).
- **Qualitative colormaps** - These mix colors with no particular sequence (e.g., rainbow or jet).

Color limits and extensions

- Matplotlib allows for a large range of colorbar customization. The colorbar itself is simply an instance of plt.Axes, so all of the axes and tick formatting tricks we've learned are applicable.
- We can narrow the color limits and indicate the out-of-bounds values with a triangular arrow at the top and bottom by setting the extend property.

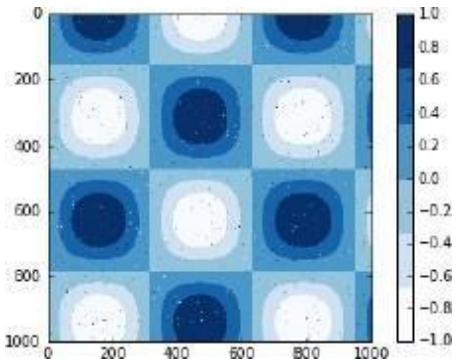
```
plt.subplot(1, 2, 2)
plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1);
```



Discrete colorbars

Colormaps are by default continuous, but sometimes you'd like to represent discrete values. The easiest way to do this is to use the plt.cm.get_cmap() function, and pass the name of a suitable colormap along with the number of desired bins.

```
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
plt.colorbar()
plt.clim(-1, 1);
```



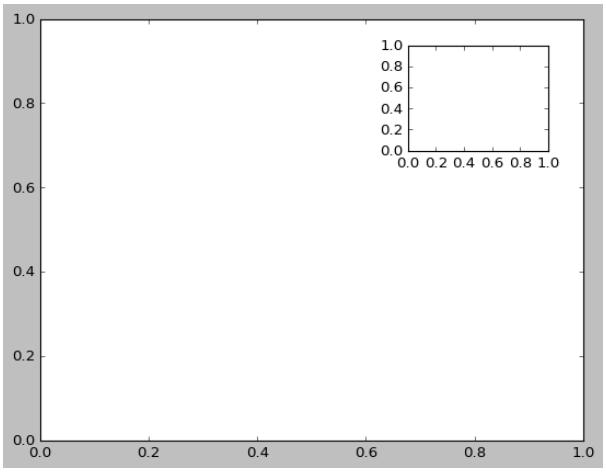
Subplots

- Matplotlib has the concept of subplots: groups of smaller axes that can exist together within a single figure.
- These subplots might be insets, grids of plots, or other more complicated layouts.
- We'll explore four routines for creating subplots in Matplotlib.
    - plt.axes: Subplots by Hand
    - plt.subplot: Simple Grids of Subplots
    - plt.subplots: The Whole Grid in One Go
    - plt.GridSpec: More Complicated Arrangements

plt.axes: Subplots by Hand

- The most basic method of creating an axes is to use the plt.axes function. As we've seen previously, by default this creates a standard axes object that fills the entire figure.
- plt.axes also takes an optional argument that is a list of four numbers in the figure coordinate system.
- These numbers represent [bottom, left, width,height] in the figure coordinate system, which ranges from 0 at the bottom left of the figure to 1 at the top right of the figure.

For example,
we might 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).




```
import matplotlib.pyplot as plt
import numpy as np
ax1 = plt.axes() # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

Vertical sub plot

The equivalent of plt.axes() command within the object-oriented interface is ig.add_axes(). Let's use this to create two vertically stacked axes.

```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
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));
```

- We now have two axes (the top with no tick labels) that are just touching: the bottom of the upper panel (at position 0.5) matches the top of the lower panel (at position 0.1+ 0.4).
- If the axis value is changed in second plot both the plots are separated with each other, example ax2 = fig.add_axes([0.1, 0.01, 0.8, 0.4

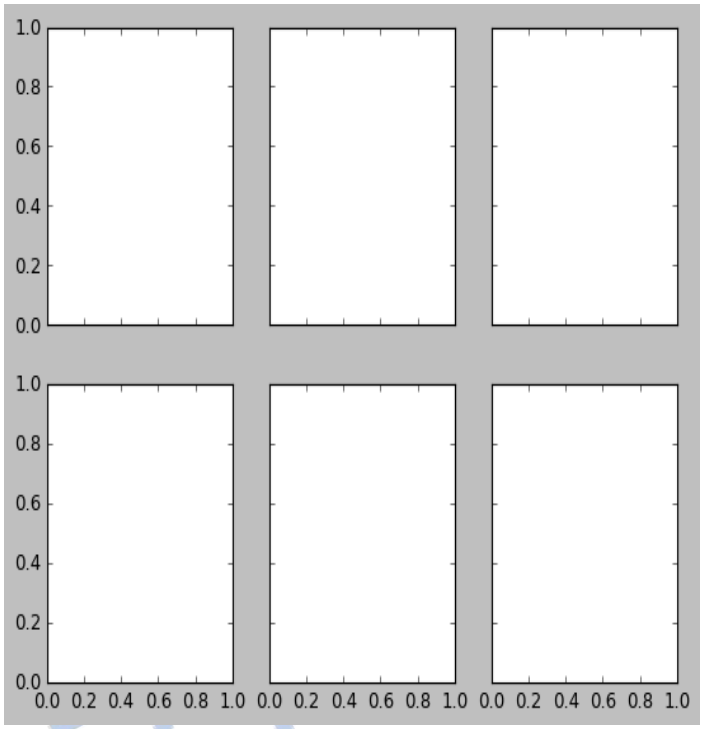
plt.subplot: Simple Grids of Subplots

- Matplotlib has several convenience routines to align columns or rows of subplots.
- The lowest level of these is plt.subplot(), which creates a single subplot within a grid.
- This command takes three integer arguments—the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right

```
for i in range(1, 7):
plt.subplot(2, 3, i)
plt.text(0.5, 0.5, str((2, 3, i)),
fontsize=18, ha='center')
```

plt.subplots: The Whole Grid in One Go

- The approach just described can become quite tedious when you're creating a large grid of subplots, especially if you'd like to hide the x- and y-axis labels on the inner plots.
- For this purpose, plt.subplots() is the easier tool to use (note the s at the end of subplots).
- Rather than creating a single subplot, this function creates a full grid of subplots in a single line, returning them in a NumPy array.
- The arguments are the number of rows and number of columns, along with optional keywords sharex and sharey, which allow you to specify the relationships between different axes.
- Here we'll create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale



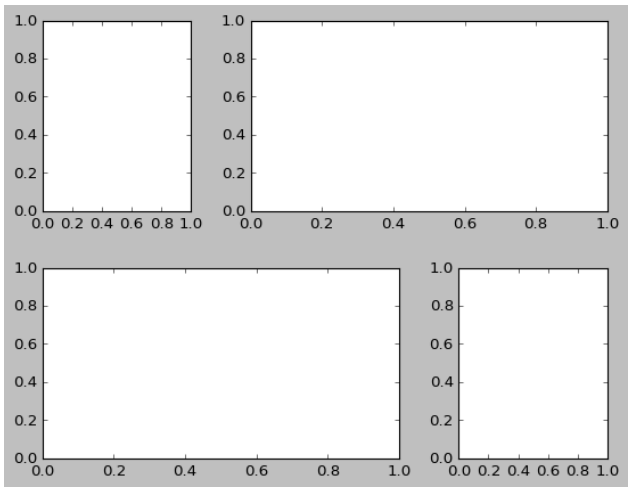
```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```

Note that by specifying sharex and sharey, we've automatically removed inner labels on the grid to make the plot cleaner.

plt.GridSpec: More Complicated Arrangements

To go beyond a regular grid to subplots that span multiple rows and columns, plt.GridSpec() is the best tool. The plt.GridSpec() object does not create a plot by itself; it is simply a convenient interface that is recognized by the plt.subplot() command.

For example, a gridspec for a grid of two rows and three columns with some specified width and height space looks like this:



```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

From this we can specify subplot locations and extents

```
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2]);
```

Text and Annotation

- The most basic types of annotations we will use are axes labels and titles, here we will see some more visualization and annotation information's.

- Text annotation can be done manually with the plt.text/ax.text command, which will place text at a particular x/y value.
- The ax.text method takes an x position, a y position, a string, and then optional keywords specifying the color, size, style, alignment, and other properties of the text. Here we used ha='right' and ha='center', where ha is short for horizontal alignment.

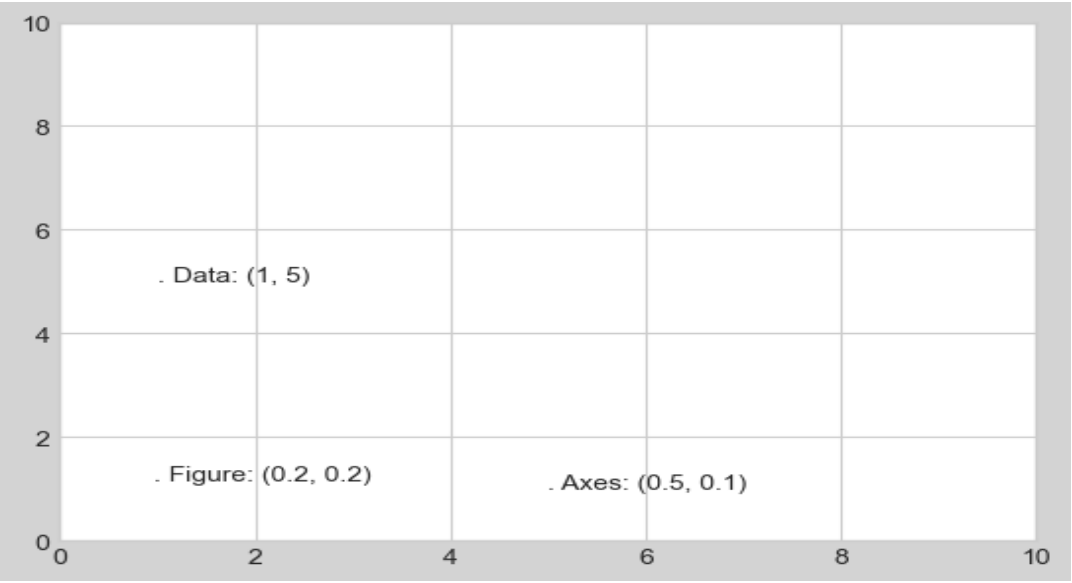
Transforms and Text Position

- We anchored our text annotations to data locations. Sometimes it's preferable to anchor the text to a position on the axes or figure, independent of the data. In Matplotlib, we do this by modifying the transform.
- Any graphics display framework needs some scheme for translating between coordinate systems.
- Mathematically, such coordinate transformations are relatively straightforward, and Matplotlib has a well-developed set of tools that it uses internally to perform them (the tools can be explored in the matplotlib.transforms submodule).
- There are three predefined transforms that can be useful in this situation.

    - 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)

Example

```
import matplotlib.pyplot as plt
import matplotlib as mpl
plt.style.use('seaborn-whitegrid')
import numpy as np
import pandas as pd
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)
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```



Note that by default, the text is aligned above and to the left of the specified coordinates; here the "." at the beginning of each string will approximately mark the given coordinate location.

The transData coordinates give the usual data coordinates associated with the x- and y-axis labels. The transAxes coordinates give the location from the bottom-left corner of the axes (here the white box) as a fraction of the axes size.

The transfigure coordinates are similar, but specify the position from the bottom left of the figure (here the gray box) as a fraction of the figure size.
Notice now that if we change the axes limits, it is only the transData coordinates that will be affected, while the others remain stationary.

Arrows and Annotation

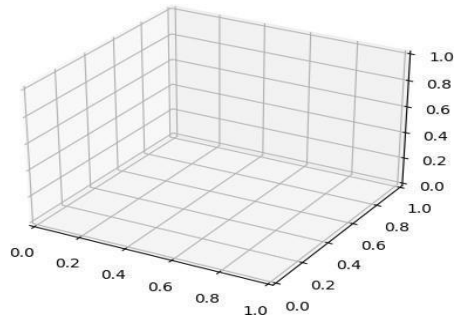
- Along with tick marks and text, another useful annotation mark is the simple arrow.
- Drawing arrows in Matplotlib is not much harder because there is a plt.arrow() function available.
- The arrows it creates are SVG (scalable vector graphics)objects that will be subject to the varying aspect ratio of your plots, and the result is rarely what the user intended.
- The arrow style is controlled through the arrowprops dictionary, which has numerous options available.

Three-Dimensional Plotting in Matplotlib

We enable three-dimensional plots by importing the mplot3d toolkit, included with the main Matplotlib installation.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
fig = plt.figure()
ax = plt.axes(projection='3d')
```

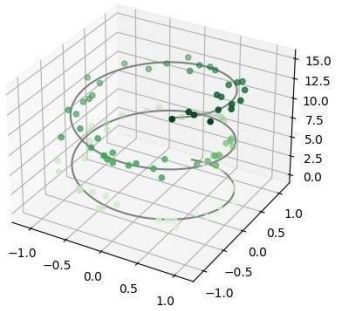
With this 3D axes enabled, we can now plot a variety of three-dimensional plot types.



Three-Dimensional Points and Lines

The most basic three-dimensional plot is a line or scatter plot created from sets of (x, y, z) triples.
In analogy with the more common two-dimensional plots discussed earlier, we can create these using the ax.plot3D and ax.scatter3D functions

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
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)
```



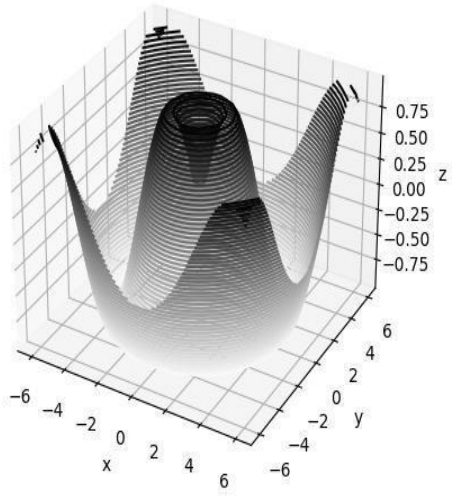
```
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
plt.show()
```

Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page.

Three-Dimensional Contour Plots

- mplot3d contains tools to create three-dimensional relief plots using the same inputs.
- Like two-dimensional ax.contour plots, ax.contour3D requires all the input data to be in the form of two-dimensional regular grids, with the Z data evaluated at each point.
- Here we'll show a three-dimensional contour diagram of a three dimensional sinusoidal function

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```



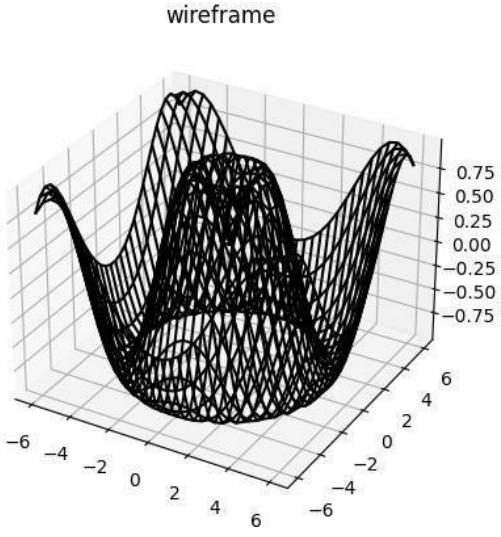
Sometimes the default viewing angle is not optimal, in which case we can use the view_init method to set the elevation and azimuthal angles.

```
ax.view_init(60, 35)
fig
```

Wire frames and Surface Plots

- Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots.
- These take a grid of values and project it onto the specified threedimensional surface, and can make the resulting three-dimensional forms quite easy to visualize.

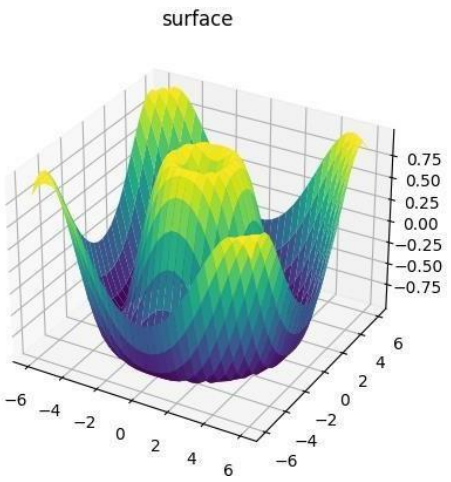
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
plt.show()
```



- A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon.

- Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized

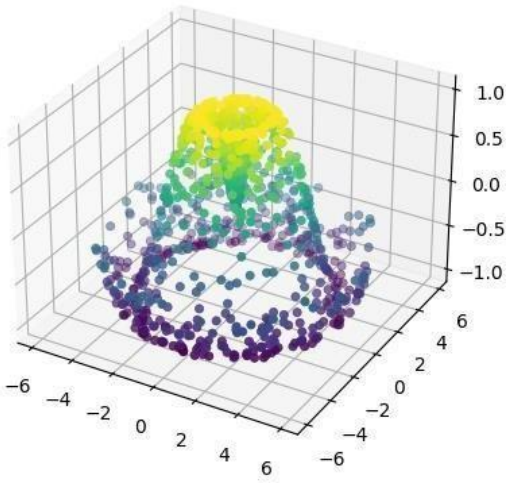
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
ax.set_title('surface')
plt.show()
```



Surface Triangulations

- For some applications, the evenly sampled grids required by the preceding routines are overly restrictive and inconvenient.
- In these situations, the triangulation-based plots can be very useful.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
theta = 2 * np.pi * np.random.random(1000)
r = 6 * np.random.random(1000)
x = np.ravel(r * np.sin(theta))
y = np.ravel(r * np.cos(theta))
z = f(x, y)
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5)
```



Geographic Data with Basemap

- One common type of visualization in data science is that of geographic data.
- Matplotlib's main tool for this type of visualization is the Basemap toolkit, which is one of several Matplotlib toolkits that live under the mpl_toolkits namespace.
- Basemap is a useful tool for Python users to have in their virtual toolbelts
- Installation of Basemap. Once you have the Basemap toolkit installed and imported, geographic plots also require the PIL package in Python 2, or the pillow package in Python 3.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```

- Matplotlib axes that understands spherical coordinates and allows us to easily over-plot data on the map

- We'll use an etopo image (which shows topographical features both on land and under the ocean) as the map background

Program to display particular area of the map with latitude and longitude lines

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
from itertools import chain
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)
def draw_map(m, scale=0.2):
# draw a shaded-relief image
   m.shadedrelief(scale=scale)
# lats and longs are returned as a dictionary
lats = m.drawparallels(np.linspace(-90, 90, 13))
lons = m.drawmeridians(np.linspace(-180, 180, 13))
# keys contain the plt.Line2D instances
lat_lines = chain(*(tup[1][0] for tup in lats.items()))
lon_lines = chain(*(tup[1][0] for tup in lons.items()))
all_lines = chain(lat_lines, lon_lines)
# cycle through these lines and set the desired style
for line in all_lines:
   line.set(linestyle='-', alpha=0.3, color='r')
```

Map Projections

The Basemap package implements several dozen such projections, all referenced by a short format code. Here we'll briefly demonstrate some of the more common ones.

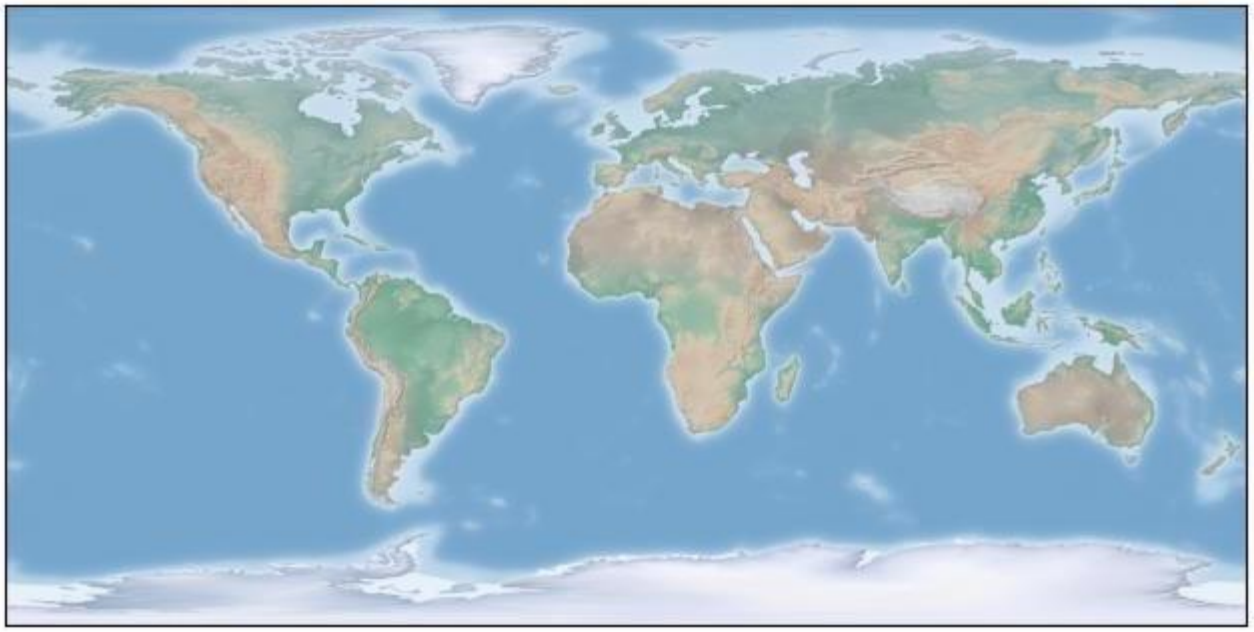
- Cylindrical projections
- Pseudo-cylindrical projections
- Perspective projections
- Conic projections

Cylindrical projection

- The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitude are mapped to horizontal and vertical lines, respectively.
- This type of mapping represents equatorial regions quite well, but results in extreme distortions near the poles.
- The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles.
- Other cylindrical projections are the Mercator (projection='merc') and the cylindrical equal-area (projection='cea') projections.
- The additional arguments to Basemap for this view specify the latitude (lat) and longitude (lon) of the lower-left corner (llcrnr) and upper-right corner (urcrnr) for the desired map, in units of degrees.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
```

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
llcrnrlat=-90, urcrnrlat=90,
llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)
```



Pseudo-cylindrical projections

- Pseudo-cylindrical projections relax the requirement that meridians (lines of constant longitude) remain vertical; this can give better properties near the poles of the projection.
- The Mollweide projection (projection='moll') is one common example of this, in which all meridians are elliptical arcs
- It is constructed so as to
- preserve area across the map: though there are distortions near the poles, the area of small patches reflects the true area.
- Other pseudo-cylindrical projections are the sinusoidal (projection='sinu') and Robinson (projection='robin') projections.
- The extra arguments to Basemap here refer to the central latitude (lat_0) and longitude (lon_0) for the desired map.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None,
lat_0=0, lon_0=0)
draw_map(m)
```

Perspective projections

- Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lies within the Earth!).

- One common example is the orthographic projection (projection='ortho'), which shows one side of the globe as seen from a viewer at a very long distance.
- Thus, it can show only half the globe at a time.
- Other perspective-based projections include the gnomonic projection (projection='gnom') and stereographic projection (projection='stere').
- These are often the most useful for showing small portions of the map.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
lat_0=50, lon_0=0)
draw_map(m);
```

Conic projections

- A conic projection projects the map onto a single cone, which is then unrolled.
- This can lead to very good local properties, but regions far from the focus point of the cone may become very distorted.
- One example of this is the Lambert conformal conic projection (projection='lcc').
- It projects the map onto a cone arranged in such a way that two standard parallels (specified in Basemap by lat_1 and lat_2) have well-represented distances, with scale decreasing between them and increasing outside of them.
- Other useful conic projections are the equidistant conic (projection='eqdc') and the Albers equal-area (projection='aea') projection

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
lon_0=0, lat_0=50, lat_1=45, lat_2=55, width=1.6E7, height=1.2E7)
draw_map(m)
```

Drawing a Map Background

The Basemap package contains a range of useful functions for drawing borders of physical features like continents, oceans, lakes, and rivers, as well as political boundaries such as countries and US states and counties.

The following are some of the available drawing functions that you may wish to explore using IPython's help features:

• 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
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

- The Basemap toolkit is the ability to over-plot a variety of data onto a map background.
- There are many map-specific functions available as methods of the Basemap instance.

Some of these map-specific methods are:

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

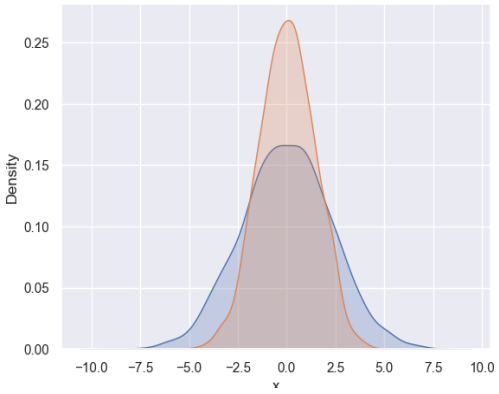
Visualization with Seaborn

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Histograms, KDE, and densities

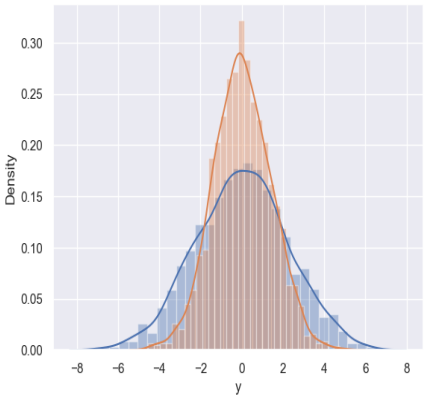
- In statistical data visualization, all you want is to plot histograms and joint distributions of variables. We have seen that this is relatively straightforward in Matplotlib
- Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with sns.kdeplot

```
import pandas as pd
import seaborn as sns
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])
for col in 'xy':
        sns.kdeplot(data[col], shade=True)
```



- Histograms and KDE can be combined using distplot

```
sns.distplot(data['x'])
sns.distplot(data['y']);
```



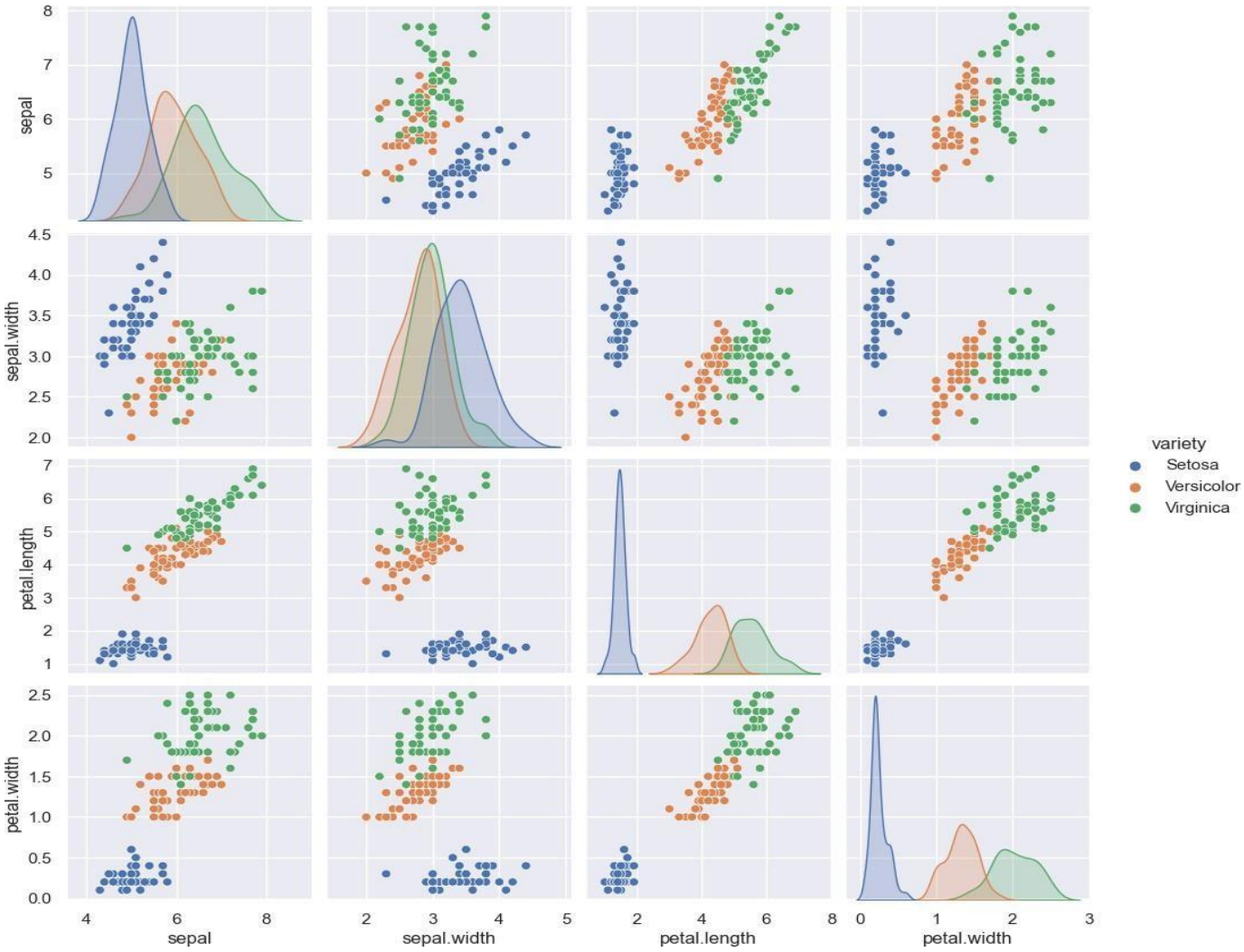
- If we pass the full two-dimensional dataset to kdeplot, we will get a two-dimensional visualization of the data.
- We can see the joint distribution and the marginal distributions together using sns.jointplot.

Pair plots

When you generalize joint plots to datasets of larger dimensions, you end up with pair plots. This is very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

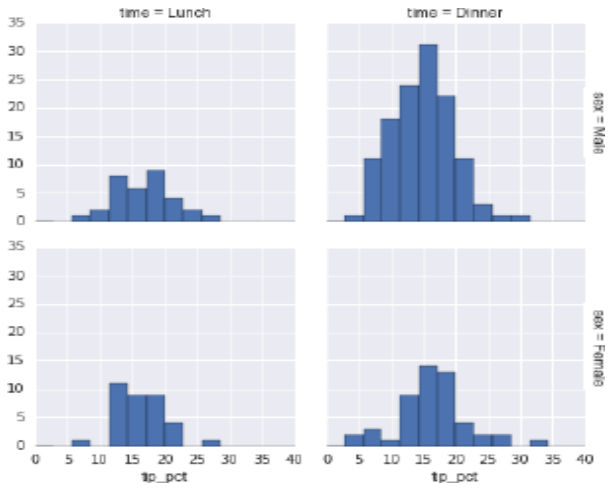
We'll demo this with the Iris dataset, which lists measurements of petals and sepals of three iris species:

```
import seaborn as sns
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue='species', size=2.5);
```



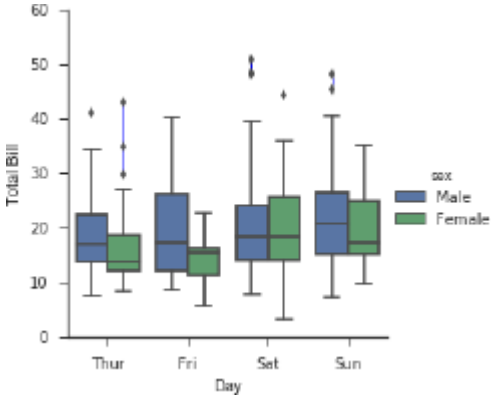
Faceted histograms

- Sometimes the best way to view data is via histograms of subsets. Seaborn's FacetGrid makes this extremely simple.
- We'll take a look at some data that shows the amount that restaurant staff receive in tips based on various indicator data



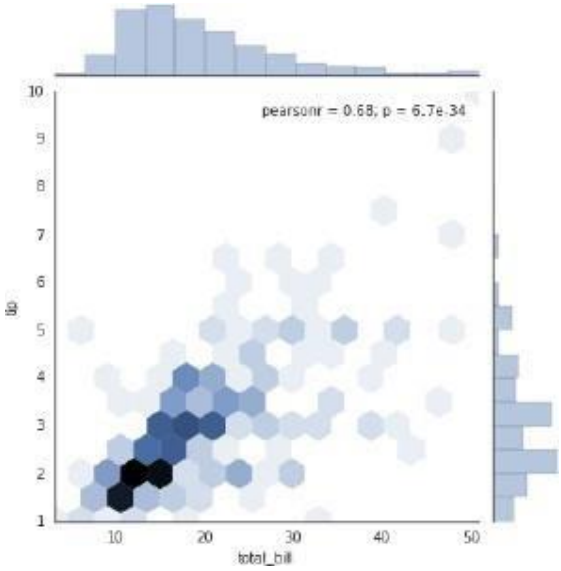
Factor plots

Factor plots can be useful for this kind of visualization as well. This allows you to view the distribution of a parameter within bins defined by any other parameter.



Joint distributions

Similar to the pair plot we saw earlier, we can use sns.jointplot to show the joint distribution between different datasets, along with the associated marginal distributions.



Bar plots

Time series can be plotted with sns.factorplot.