



THE DATA SCIENCE HANDBOOK

The Only Resource You Will Ever Need for Data Science



Python Intro

Programming in Python

Concepts

- When we give a computer a set of instructions, we say that we're **programming** it. To program a computer, we need to write the instructions in a special language, which we call a **programming language**.
- Python has **syntax** rules, and each line of instruction must comply with these rules. For example, `print(23 + 7) print(10 - 6) print(12 + 38)` doesn't comply with Python's syntax rules and raises a **syntax error**.
- The instructions we send to the computer are collectively known as **code**. Each line of instruction is known as a **line of code**.
- When we write code, we *program* the computer to do something. For this reason, we also call the code we write a **computer program**, or a **program**.
- The code we write serves as **input** to the computer. The result of executing the code is called **output**.
- The sequence of characters that follows the `#` symbol is called a **code comment**. We can use code comments to stop the computer executing a line of code or add information about the code we write.

Syntax

- Displaying the output of a computer program:

```
print(5 + 10)
print(5 * 10)
```
- Ignoring certain lines of code by using code comments:

```
# print(5 + 10)
print(5 * 10)
# This program will only print 50
```
- Performing arithmetical operations:

```
1 + 2
4 - 5
30 * 1
```

`20 / 3`

`4**3`

`(4 * 18)**2 / 10`

Variables and Data Types

Concepts

- We can store values in the computer memory. Each storage location in the computer's memory is called a **variable**.
- There are two syntax rules we need to be aware of when we're naming variables:
 - We must use only letters, numbers, or underscores (we can't use apostrophes, hyphens, whitespace characters, etc.).
 - Variable names can't start with a number.
- Whenever the syntax is correct, but the computer still returns an error for one reason or another, we say we got a **runtime error**.
- In Python, the `=` operator tells us that the value on the right is **assigned** to the variable on the left. It doesn't tell us anything about equality. We call `=` an **assignment operator**, and we read code like `x = 5` as "five is assigned to x" or "x is assigned five", but not "x equals five".
- In computer programming, values are classified into different **types**, or **data types**. The type of a value offers the computer the required information about how to handle that value. Depending on the type, the computer will know how to store a value in memory, or what operations can and can't be performed on a value.
- In this mission, we learned about three data types: integers, floats, and strings.
- The process of linking two or more strings together is called **concatenation**.

Syntax

- Storing values to variables:

```
twenty = 20
result = 43 + 2**5
currency = 'USD'
```
- Updating the value stored in a variable:

```
x = 30
x += 10 # this is the same as x = x + 10
```
- Rounding a number:

```
round(4.99) # the output will be 5
```
- Using quotation marks to create a string:

```
app_name = "Clash of Clans"
```

```
app_rating = '3.5'
```

- Concatenating two or more strings:

```
print('a' + 'b') # prints 'ab'
```

```
print('a' + 'b' + 'c') # prints 'abc'
```

- Converting between types of variables:

```
int('4')
```

```
str(4)
```

```
float('4.3')
```

```
str('4.3')
```

- Finding the type of a value:

```
type(4)
```

```
type('4')
```

Resources

- More on [Strings in Python](#).

Lists and For Loops

Concepts

- A **data point** is a value that offers us some information.
- A set of data points make up a **data set**. A table is an example of a data set.
- **Lists** are data types which we can use to store data sets.
- Repetitive process can be automated using **for loops**.

Syntax

- Creating a list of data points:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_2 = ['Instagram', 0.0, 'USD', 2161558, 4.5]
```

- Creating a list of lists:

```
data = [row_1, row_2]
```

- Retrieving an element of a list:

```
first_row = data[0]
first_element_in_first_row = first_row[0]
first_element_in_first_row = data[0][0]
last_element_in_first_row = first_row[-1]
last_element_in_first_row = data[0][-1]
```

- Retrieving multiple list elements and creating a new list:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
rating_data_only = [row_1[3], row_1[4]]
```

- Performing list slicing:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
second_to_fourth_element = row_1[1:4]
```

- Opening a data set file and using it to create a list of lists:

```
opened_file = open('AppleStore.csv')
from csv import reader
read_file = reader(opened_file)
apps_data = list(read_file)
```

- Repeating a process using a for loop:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]

for data_point in row_1:

    print(data_point)
```

Resources

- [Python Lists](#)
- [Python For Loops](#)
- [More on CSV files](#)
- [A list of keywords in Python](#) — **for** and **in** are examples of keywords (we used **for** and **in** to write for loops)

Conditional Statements

Concepts

- We can use an **if statement** to implement a condition in our code.
- An **elif** clause is executed if the preceding **if** statement (or the other preceding **elif** clauses) resolves to **False** and the condition specified after the **elif** keyword evaluates to **True**.
- **True** and **False** are **Boolean values**.
- **and** and **or** are **logical operators**, and they bridge two or more Booleans together.
- We can compare a value **A** to value **B** to determine whether:
 - **A is equal to B** and vice versa (**B** is equal to **A**) — **==**.
 - **A is not equal to B** and vice versa — **!=**.
 - **A is greater than B** or vice versa — **>**.
 - **A is greater than or equal to B** or vice versa — **>=**.
 - **A is less than B** or vice versa — **<**.
 - **A is less than or equal to B** or vice versa — **<=**.

Syntax

- Using an **if** statement to control your code:

```
if True:  
    print(1)  
  
if 1 == 1:  
    print(2)  
    print(3)
```

- Combining multiple conditions:

```
if 3 > 1 and 'data == data':  
    print('Both conditions are true!')  
  
if 10 < 20 or 4 <= 5:  
    print('At least one condition is true.')
```

- Building more complex **if** statements:

```
if (20 > 3 and 2 != 1) or 'Games' == 'Games':  
    print('At least one condition is true.')
```

- Using the else clause:

```
if False:  
    print(1)  
  
else:  
    print('The condition above was false.')
```

- Using the elif clause:

```
if False:  
    print(1)  
  
elif 30 > 5:  
    print('The condition above was false.')
```

Resources

- [If Statements in Python](#)

Dictionaries and Frequency Tables

Concepts

- The index of a dictionary value is called a **key**. In `'4+': 4433`, the dictionary key is `'4+'`, and the dictionary value is `4433`. As a whole, `'4+': 4433` is a **key-value pair**.
- Dictionary values can be of any data type: strings, integers, floats, Booleans, lists, and even dictionaries. Dictionary keys can be of almost any data type we've learned so far, excepting lists and dictionaries. If we use lists or dictionaries as dictionary keys, the computer raises an error.
- We can check whether a certain value exist in the dictionary as a key using an the **in** operator. An **in** expression always returns a Boolean value.
- The number of times a unique value occurs is also called **frequency**. Tables that map unique values to their frequencies are called **frequency tables**.
- When we iterate over a dictionary with a **for** loop, the looping is done by default over the dictionary keys.

Syntax

- Creating a dictionary:

```
# First way
dictionary = {'key_1': 1, 'key_2': 2}

# Second way
dictionary = {}
dictionary['key_1'] = 1
dictionary['key_2'] = 2
```

- Retrieving individual dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}
dictionary['key_1'] # Outputs 100
dictionary['key_2'] # Outputs 200
```

- Checking whether a certain value exist in the dictionary as a key:

```
dictionary = {'key_1': 100, 'key_2': 200}
'key_1' in dictionary # Outputs True
'key_5' in dictionary # Outputs False
100 in dictionary # Outputs False
```

- Updating dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}
dictionary['key_1'] += 600 # This will change the value to 700
```

- Creating a frequency table for the unique values in a column of a data set:

```
frequency_table = {}

for row in a_data_set:
    a_data_point = row[5]

    if a_data_point in frequency_table:
        frequency_table[a_data_point] += 1
    else:
        frequency_table[a_data_point] = 1
```

Functions: Fundamentals

Concepts

- Generally, a function displays this pattern:
 1. It takes in an input.
 2. It does something to that input.
 3. It gives back an output.
- In Python, we have **built-in functions** (like `sum()`, `max()`, `min()`, `len()`, `print()`, etc.) and functions that we can create ourselves.
- Structurally, a function is composed of a header (which contains the `def` statement), a body, and a `return` statement.
- Input variables are called **parameters**, and the various values that parameters take are called **arguments**. In `def square(number)`, the `number` variable is a parameter.
In `square(number=6)`, the value `6` is an argument that is passed to the parameter `number`.
- Arguments that are passed by name are called **keyword arguments** (the parameters give the name). When we use multiple keyword arguments, the order we use doesn't make any practical difference.
- Arguments that are passed by position are called **positional arguments**. When we use multiple positional arguments, the order we use matters.
- **Debugging** more complex functions can be a bit more challenging, but we can find the **bugs** by reading the **traceback**.

Syntax

- Creating a function with a single parameter:

```
def square(number):  
    return number**2
```

- Creating a function with more than one parameter:

```
def add(x, y):  
    return x + y
```

- Reusing a function within another function's definition:

```
def add_to_square(x):  
    return square(x) + 1000 # we defined square() above
```

Resources

- [Functions in Python](#)

Functions: Intermediate

Concepts

- We need to avoid using the name of a built-in function to name a function or a variable because this overwrites the built-in function.
- Each built-in function is well documented in [the official Python documentation](#).
- Parameters and return statements are not mandatory when we create a function.

```
def print_constant():
    x = 3.14
    print(x)
```

- The code inside a function definition is executed only when the function is called.
- When a function is called, the variables defined inside the function definition are saved into a temporary memory that is erased immediately after the function finishes running. The temporary memory associated with a function is isolated from the memory associated with the main program (the main program is the part of the program outside function definitions).
- The part of a program where a variable can be accessed is often called scope. The variables defined in the main program are said to be in the global scope, while the variables defined inside a function are in the local scope.
- Python searches the global scope if a variable is not available in the local scope, but the reverse doesn't apply — Python won't search the local scope if it doesn't find a variable in the global scope. Even if it searched the local scope, the memory associated with a function is temporary, so the search would be pointless.

Syntax

- Initiating parameters with **default arguments**:

```
def add_value(x, constant=3.14):
    return x + constant
```

- Using **multiple return statements**:

```
def sum_or_difference(a, b, do_sum):
    if do_sum:
        return a + b
    return a - b
```

- Returning **multiple variables**:

```
def sum_and_difference(a, b):
```

```
a_sum = a + b
difference = a - b
return a_sum, difference
sum_1, diff_1 = sum_and_difference(15, 10)
```

Resources

- [Python official documentation](#)
- [Style guide for Python code](#)

Learn and Install Jupyter Notebook

Concepts

- Jupyter Notebook, often referred to as Jupyter, is much more complex than a code editor. Jupyter Notebook allows us to:
 - Type and execute code
 - Add accompanying text to our code (including math equations)
 - Add visualizations
- Jupyter can run in a browser and is often used to create compelling data science projects that can be easily shared with other people.
- A notebook is a file created using Jupyter notebooks. Notebooks can easily be shared and distributed so people can view your work.
- Types of modes in Jupyter:
 - Jupyter is in edit mode whenever we type in a cell — a small pencil icon appears to the right of the menu bar.
 - Jupyter is in command mode whenever we press Esc or whenever we click outside of the cell — the pencil to the right of the menu bar disappears.
- State refers to what a computer remembers about a program.
- We can convert a code cell to a Markdown cell to add text to explain our code. Markdown syntax allows us to use keyboard symbols to format our text.
- Installing the Anaconda distribution will install both Python and Jupyter on your computer.

Syntax

MARKDOWN SYNTAX

- Adding italics and bold:
Italics
****Bold****
- Adding headers (titles) of various sizes:
header one
header two
- Adding hyperlinks and images:
[Link] (<http://a.com>)

- Adding block quotes:

`> Blockquote`

- Adding lists:

`*`

`*`

`*`

- Adding horizontal lines:

`---`

- Adding inline code:

`'Inline code with backticks'`

- Adding code blocks

`````

`code`

`````

JUPYTER NOTEBOOK SPECIAL COMMANDS

- Displaying the code execution history:

`%history -p`

Keyboard Shortcuts

- Some of the most useful keyboard shortcuts we can use in command mode are:

- Ctrl + Enter: run selected cell
- Shift + Enter: run cell, select below
- Alt + Enter: run cell, insert below
- Up: select cell above
- Down: select cell below
- Enter: enter edit mode
- A: insert cell above
- B: insert cell below
- D, D (press D twice): delete selected cell
- Z: undo cell deletion
- S: save and checkpoint

- Y: convert to code cell
- M: convert to Markdown cell
- Some of the most useful keyboard shortcuts we can use in edit mode are:
 - Ctrl + Enter: run selected cell
 - Shift + Enter: run cell, select below
 - Alt + Enter: run cell, insert below
 - Up: move cursor up
 - Down: move cursor down
 - Esc: enter command mode
 - Ctrl + A: select all
 - Ctrl + Z: undo
 - Ctrl + Y: redo
 - Ctrl + S: save and checkpoint
 - Tab : indent or code completion
 - Shift + Tab: tooltip

Resources

- [Markdown syntax](#)
- [Installing Anaconda](#)

Cleaning and Preparing Data in Python

Concepts

- When working with comma separated value (CSV) data in Python, it's common to have your data in a 'list of lists' format, where each item of the internal lists are strings.
- If you have numeric data stored as strings, sometimes you will need to remove and replace certain characters before you can convert the strings to numeric types like `int` and `float`.
- Strings in Python are made from the same underlying data type as lists, which means you can index and slice specific characters from strings like you can lists.

Syntax

TRANSFORMING AND CLEANING STRINGS

- Replace a substring within a string:

```
green_ball = "red ball".replace("red", "green")
```

- Remove a substring:

```
friend_removed = "hello there friend!".replace(" friend", "")
```

- Remove a series of characters from a string:

```
bad_chars = ["'", ",", ".", "!"]
```

```
string = "We'll remove apostrophes, commas, periods, and exclamation marks!"
```

```
for char in bad_chars:
```

```
    string = string.replace(char, "")
```

- Convert a string to title cases:

```
Hello = "hello".title()
```

- Check a string for the existence of a substring:

```
if "car" in "carpet":
```

```
    print("The substring was found.")
```

```
else:
```

```
    print("The substring was not found.")
```

- Split a string into a list of strings:

```
Split_on_dash = "1980-12-08".split("-")
```

- Slice characters from a string by position:

```
last_five_chars = "This is a long string."[:5]
```

- Concatenate strings:

```
superman = "Clark" + " " + "Kent"
```

Resources

- [Python Documentation: String Methods](#)

Python Data Analysis Basics

Concepts

- The `str.format()` method allows you to insert values into strings without explicitly converting them.
- The `str.format()` method also accepts optional format specifications which you can use to format values, so they are more easily read.

Syntax

STRING FORMATTING AND FORMAT SPECIFICATIONS

- Insert values into a string in order:

```
continents = "France is in {} and China is in {}".format("Europe",
    "Asia")
```

- Insert values into a string by position:

```
squares = "{0} times {0} equals {1}".format(3,9)
```

- Insert values into a string by name:

```
population = "{name}'s population is {pop}
million".format(name="Brazil", pop=209)
```

- Format specification for precision of two decimal places:

```
two_decimal_places = "I own {:.2f}% of the
company".format(32.5548651132)
```

- Format specification for comma separator:

```
india_pop = The approximate population of {} is
{}".format("India",1324000000)
```

- Order for format specification when using precision and comma separator:

```
balance_string = "Your bank balance is {:.2f}].format(12345.678)
```

Resources

- [Python Documentation: Format Specifications](#)
- [PyFormat: Python String Formatting Reference](#)

Object-Oriented Python

Concepts

- In **Object-Oriented Programming**, the fundamental building blocks are objects.
 - It differs from **Procedural** programming, where sequential steps are executed.
- An **object** is an entity that stores data.
- A **class** describes an object's type. It defines:
 - What data is stored in the object, known as attributes
 - What actions the object can do, known as methods
- An **attribute** is a variable that belongs to an instance of a class.
- A **method** is a function that belongs to an instance of a class.
- Attributes and methods are accessed using **dot notation**. Attributes do not use parentheses, whereas methods do.
- An **instance** describes a specific example of a class. For instance, in the code `x = 3`, `x` is an instance of the type `int`.
 - When an object is created, it is known as **instantiation**.
- A **class definition** is code that defines how a class behaves, including all methods and attributes.
- The `init` method is a special method that runs at the moment an object is instantiated.
 - The `init` method (`__init__()`) is one of a number of special methods that Python defines.
- All methods must include `self`, representing the object instance, as their first parameter.
- It is convention to start the name of any attributes or methods that aren't intended for external use with an underscore.

Syntax

- Define an empty class:

```
class MyClass():  
    pass
```
- Instantiate an object of a class:

```
class MyClass():  
    pass  
  
mc_1 = MyClass()
```

- Define an init function in a class to assign an attribute at instantiation:

```
class MyClass():

    def __init__(self, param_1):

        self.attribute_1 = param_1

mc_2 = MyClass("arg_1")
```

- Define a method inside a class and call it on an instantiated object:

```
class MyClass():

    def __init__(self, param_1):

        self.attribute_1 = param_1

    def add_20(self):

        self.attribute_1 += 20

mc_3 = MyClass(10) # mc.attribute is 10

mc.add_20() # mc.attribute is 30
```

Resources

- [Python Documentation: Classes](#)

Working with Dates and Times in Python

Concepts

- The `datetime` module contains the following classes:
 - `datetime.datetime` — For working with date and time data
 - `datetime.time` — For working with time data only
 - `datetime.timedelta` — For representing time periods
- Time objects behave similarly to `datetime` objects for the following reasons:
 - They have attributes like `time.hour` and `time.second` that you can use to access individual time components.
 - They have a `time.strftime()` method, which you can use to create a formatted string representation of the object.
- The `timedelta` type represents a period of time, e.g. 30 minutes or two days.
- Common format codes when working with `datetime.datetime.strptime`:

Strftime Code	Meaning	Examples
<code>%d</code>	Day of the month as a zero-padded number ¹	<code>04</code>
<code>%A</code>	Day of the week as a word ²	<code>Monday</code>
<code>%m</code>	Month as a zero-padded number ¹	<code>09</code>
<code>%Y</code>	Year as a four-digit number	<code>1901</code>
<code>%y</code>	Year as a two-digit number with zero-padding ^{1, 3}	<code>01</code> (2001) <code>88</code> (1988)
<code>%B</code>	Month as a word ²	<code>September</code>

Strftime Code	Meaning	Examples
%H	Hour in 24 hour time as zero-padded number ¹	05 (5 a.m.) 15 (3 p.m.)
%p	a.m. or p.m. ²	AM
%I	Hour in 12 hour time as zero-padded number ¹	05 (5 a.m., or 5 p.m. if AM/PM indicates otherwise)
%M	Minute as a zero-padded number ¹	07

- 1. The strftime parser will parse non-zero padded numbers without raising an error.
- 2. Date parts containing words will be interpreted using the locale settings on your computer, so strftime won't be able to parse 'febrero' (february in Spanish) if your locale is set to an english language locale.
- 3. Year values from 00-68 will be interpreted as 2000-2068, with values 70-99 interpreted as 1970-1999.

- Operations between timedelta, datetime, and time objects (datetime can be substituted with time):

Operation	Explanation	Resultant Type
datetime - datetime	Calculate the time between two specific dates/times	timedelta
datetime - timedelta	Subtract a time period from a date or time.	datetime
datetime + timedelta	Add a time period to a date or time.	datetime
timedelta + timedelta	Add two periods of time together	timedelta
timedelta - timedelta	Calculate the difference between two time periods.	timedelta

Syntax

IMPORTING MODULES AND DEFINITIONS

- Importing a whole module:

```
import csv
```

```
csv.reader()
```

- Importing a whole module with an alias:

```
import csv as c
c.reader()
```

- Importing a single definition:

```
from csv import reader
reader()
```

- Importing multiple definitions:

```
from csv import reader, writer
reader()
writer()
```

- Importing all definitions:

```
from csv import *
```

WORKING WITH THE DATETIME MODULE

- All examples below presume the following import code:

```
import datetime as dt
```

- Creating datetime.datetime string given a month, year, and day:

```
eg_1 = dt.datetime(1985, 3, 13)
```

- Creating a datetime.datetime object from a string:

```
eg_2 = dt.datetime.strptime("24/12/1984", "%d/%m/%Y")
```

- Converting a datetime.datetime object to a string:

```
dt_object = dt.datetime(1984, 12, 24)
```

```
dt_string = dt_object.strftime("%d/%m/%Y")
```

- Instantiating a datetime.time object:

```
eg_3 = datetime.time(hour=0, minute=0, second=0, microsecond=0)
```

- Retrieving a part of a date stored in the datetime.datetime object:

```
eg_1.day
```

- Creating a date from a datetime.datetime object:

```
d2_dt = dt.datetime(1946, 9, 10)
d2 = d2_dt.date()
```

- Creating a datetime.date object from a string:

```
d3_str = "17 February 1963"  
d3_dt = dt.datetime.strptime(d3_str, "%d %B %Y")  
d3 = d3_dt.date()
```

- Instantiating a `datetime.timedelta` object:

```
eg_4 = dt.timedelta(weeks=3)
```

- Adding a time period to a `datetime.datetime` object:

```
d1 = dt.date(1963, 2, 26)
```

```
d1_plus_1wk = d1 + dt.timedelta(weeks=1)
```

Resources

- [Python Documentation - Datetime module](#)
- [Python Documentation: Strftime/Strptime Codes](#)
- [strftime.org](#)

Data Analysis and Visualization

Introduction to NumPy

Concepts

- Python is considered a high-level language because we don't have to manually allocate memory or specify how the CPU performs certain operations. A low-level language like C gives us this control and lets us improve specific code performance, but a tradeoff in programmer productivity is made. The NumPy library lets us write code in Python but take advantage of the performance that C offers. One way NumPy makes our code run quickly is **vectorization**, which takes advantage of **Single Instruction Multiple Data (SIMD)** to process data more quickly.
- A list in NumPy is called a 1D Ndarray and a list of lists is called a 2D Ndarray. NumPy ndarrays use indices along both rows and columns and is the primary way we select and slice values.

Syntax

SELECTING ROWS, COLUMNS, AND ITEMS FROM AN NDARRAY

- Convert a list of lists into a ndarray:

```
import numpy as np  
  
f = open("nyc_taxis.csv", "r")  
  
taxi_list = list(csv.reader(f))  
  
taxi = np.array(converted_taxi_list)
```

- Selecting a row from an ndarray:

```
second_row = taxi[1]
```

- Selecting multiple rows from an ndarray:

```
all_but_first_row = taxi[1:]
```

- Selecting a specific item from an ndarray:

```
fifth_row_second_column = taxi[4,1]
```

SLICING VALUES FROM AN NDARRAY

- Selecting a single column:

```
second_column = taxi[:,1]
```

- Selecting multiple columns:

```
second_third_columns = taxi[:,1:3]
```

```
cols = [1,3,5]
second_fourth_sixth_columns = taxi[:, cols]
```

- Selecting a 2D slice:

```
twod_slice = taxi[1:4, :3]
```

VECTOR MATH

- `vector_a + vector_b` - Addition
- `vector_a - vector_b` - Subtraction
- `vector_a * vector_b` - Multiplication (this is unrelated to the vector multiplication used in linear algebra).
- `vector_a / vector_b` - Division
- `vector_a % vector_b` - Modulus (find the remainder when `vector_a` is divided by `vector_b`)
- `vector_a ** vector_b` - Exponent (raise `vector_a` to the power of `vector_b`)
- `vector_a // vector_b` - Floor Division (divide `vector_a` by `vector_b`, rounding down to the nearest integer)

CALCULATING STATISTICS FOR 1D NDARRAYS

- [`ndarray.min\(\)` to calculate the minimum value](#)
- [`ndarray.max\(\)` to calculate the maximum value](#)
- [`ndarray.mean\(\)` to calculate the mean average value](#)
- [`ndarray.sum\(\)` to calculate the sum of the values](#)

CALCULATING STATISTICS FOR 2D NDARRAYS

- Max value for an entire 2D Ndarray:
`taxi.max()`
- Max value for each row in a 2D Ndarray (returns a 1D Ndarray):
`taxi.max(axis=1)`
- Max value for each column in a 2D Ndarray (returns a 1D Ndarray):
`taxi.max(axis=0)`

ADDING ROWS AND COLUMNS TO NDARRAYS

- Joining a sequence of arrays:

```
np.concatenate([a1, a2], axis=0)
```

- Expanding the shape of an array:

```
np.expand_dims([1, 2], axis=0)
```

SORTING

- Sorting a 1D Ndarray:

```
np.argsort(taxi[0])
```

- Sorting a 2D NDarray by a specific column:

```
sorted_order = np.argsort(taxi[:,15])
```

```
taxi_sorted = taxi[sorted_order]
```

Resources

- [Arithmetic functions from the NumPy documentation.](#)
- [NumPy ndarray documentation](#)

Boolean Indexing with NumPy

Concepts

- Selecting values from a Ndarray using Boolean arrays is very powerful. Using Boolean arrays helps us think in terms of filters on the data, instead of specific index values (like we did when working with Python lists).

Syntax

READING CSV FILES WITH NUMPY

- Reading in a CSV file:

```
import numpy as np  
  
taxi = np.loadtxt('nyctaxis.csv', delimiter=',', skip_header=1)
```

BOOLEAN ARRAYS

- Creating a Boolean array from filtering criteria:

```
np.array([2,4,6,8]) < 5
```

- Boolean filtering for 1D Ndarray:

```
a = np.array([2,4,6,8])  
  
filter = a < 5  
  
a[filter]
```

- Boolean filtering for 2D Ndarray:

```
tip_amount = taxi[:,12]  
  
tip_bool = tip_amount > 50  
  
top_tips = taxi[tip_bool, 5:14]
```

ASSIGNING VALUES

- Assigning values in a 2D Ndarray using indices:

```
taxi[28214,5] = 1  
  
taxi[:,0] = 16  
  
taxi[1800:1802,7] = taxi[:,7].mean()
```

- Assigning values using Boolean arrays:

```
taxi[taxi[:, 5] == 2, 15] = 1
```

Resources

- [Reading a CSV file into NumPy](#)
- [Indexing and selecting data](#)

Introduction to Pandas

Concepts

- NumPy provides fundamental structures and tools that makes working with data easier, but there are several things that limit its usefulness as a single tool when working with data:
 - The lack of support for column names forces us to frame the questions we want to answer as multi-dimensional array operations.
 - Support for only one data type per ndarray makes it more difficult to work with data that contains both numeric and string data.
 - There are lots of low level methods, however there are many common analysis patterns that don't have pre-built methods.
- The **pandas** library provides solutions to all of these pain points and more. Pandas is not so much a replacement for NumPy as an extension of NumPy. The underlying code for pandas uses the NumPy library extensively. The main objects in pandas are **Series** and **Dataframes**. Series is equivalent to a 1D Ndarray while a dataframe is equivalent to a 2D Ndarray.
- Different label selection methods:

Select by Label	Explicit Syntax	Shorthand Convention	Other Shorthand
Single column from dataframe	<code>df.loc[:, "col1"]</code>	<code>df["col1"]</code>	<code>df.col1</code>
List of columns from dataframe	<code>df.loc[:, ["col1", "col7"]]</code>	<code>df[["col1", "col7"]]</code>	
Slice of columns from dataframe	<code>df.loc[:, "col1": "col4"]</code>		
Single row from dataframe	<code>df.loc["row4"]</code>		
List of rows from dataframe	<code>df.loc[["row1", "row8"]]</code>		
Slice of rows from dataframe	<code>df.loc["row3": "row5"]</code>	<code>df["row3": "row5"]</code>	

Select by Label	Explicit Syntax	Shorthand Convention	Other Shorthand
Single item from series	s.loc["item8"]	s["item8"]	s.item8
List of items from series	s.loc[["item1","item7"]]	s[["item1","item7"]]	
Slice of items from series	s.loc["item2":"item4"]	s["item2":"item4"]	

Syntax

PANDAS DATAFRAME BASICS

- Reading a file into a dataframe:

```
f500 = pd.read_csv('f500.csv',index_col=0)
```

- Returning a dataframe's data types:

```
col_types = f500.dtypes
```

- Returning the dimensions of a dataframe:

```
dims = f500.shape
```

SELECTING VALUES FROM A DATAFRAME

- Selecting a single column:

```
f500["rank"]
```

- Selecting multiple columns:

```
f500[["country", "rank"]]
```

- Selecting the first n rows:

```
first_five = f500.head(5)
```

- Selecting rows from a dataframe by label:

```
drink_companies = f500.loc[["Anheuser-Busch InBev", "Coca-Cola", "Heineken Holding"]]

big_movers = f500.loc[["Aviva", "HP", "JD.com", "BHP Billiton"], ["rank", "previous_rank"]]

middle_companies = f500.loc["Tata Motors":"Nationwide", "rank": "country"]
```

DATA EXPLORATION METHODS

- Describing a Series object:

```
revs = f500["revenues"]

summary_stats = revs.describe()
```

- Unique Value Counts for a Column:

```
country_freqs = f500['country'].value_counts()
```

ASSIGNMENT WITH PANDAS

- Replacing a specific column with a new Series object:

```
f500["revenues_b"] = f500["revenues"] / 1000
```

- Replacing a specific value in a dataframe:

```
f500.loc["Dow Chemical","ceo"] = "Jim Fitterling"
```

BOOLEAN INDEXING IN PANDAS

- Filtering a dataframe down on a specific value in a column:

```
kr_bool = f500["country"] == "South Korea"
```

```
top_5_kr = f500[kr_bool].head()
```

- Updating values using Boolean filtering:

```
f500.loc[f500["previous_rank"] == 0, "previous_rank"] = np.nan
```

```
prev_rank_after =
f500["previous_rank"].value_counts(dropna=False).head()
```

Resources

- [Dataframe.loc\[\]](#)
- [Indexing and Selecting Data](#)

Exploring Data with pandas

Concepts

- To select values by axis labels, use `loc[]`. To select values by integer locations, use `iloc[]`. When the label for an axis is just its integer position, these methods can be mostly used interchangeably.
- To take advantage of vectorization in pandas but think and speak in filtering criteria (instead of integer index values), you'll find yourself expressing many computations as Boolean masks and filtering series and dataframes. Because using a loop doesn't take advantage of vectorization, it's important to avoid doing so unless you absolutely have to. Boolean operators are a powerful technique to further take advantage of vectorization when filtering because you're able to express more granular filters.

Syntax

USING ILOC[] TO SELECT BY INTEGER POSITION

- Selecting a value:

```
third_row_first_col = df.iloc[2,0]
```

- Selecting a row:

```
second_row = df.iloc[1]
```

CREATING BOOLEAN MASKS USING PANDAS METHODS

- String method that returns Boolean series object:

```
is_california = usa["hq_location"].str.endswith("CA")
```

- Filtering using Boolean series object:

```
df_where_filter_true = usa[is_california]
```

- Selecting only the non-null values in a column:

```
f500[f500["previous_rank"].notnull()]
```

BOOLEAN OPERATORS

- Multiple required filtering criteria:

```
filter_big_rev_neg_profit = (f500["revenues"] > 100000) &  
(f500["profits"] < 0)
```

- Multiple optional filtering criteria:

```
filter_big_rev_neg_profit = (f500["revenues"] > 100000) |  
(f500["profits"] < 0)
```

Resources

- [Boolean Indexing](#)
- [iloc vs loc](#)

Data Cleaning Basics

Concepts

- Computers, at their lowest levels, can only understand binary. Encodings are systems for representing all other values in binary so a computer can work with them. The first standard was ASCII, which specified 128 characters. Other encodings popped up to support other languages, like Latin-1 and UTF-8. UTF-8 is the most common encoding and is very friendly to work with in Python 3.
- When converting text data to numeric data, we usually follow the following steps:
 - Explore the data in the column.
 - Identify patterns and special cases.
 - Remove non-digit characters.
 - Convert the column to a numeric dtype.
 - Rename column if required.

Syntax

READING A CSV IN WITH A SPECIFIC ENCODING

- Reading in a CSV file using Latin encoding:

```
laptops = pd.read_csv('laptops.csv', encoding='Latin-1')
```

- Reading in a CSV file using UTF-8:

```
laptops = pd.read_csv('laptops.csv', encoding='UTF-8')
```

- Reading in a CSV file using Windows-1251:

```
laptops = pd.read_csv('laptops.csv', encoding='Windows-1251')
```

MODIFYING COLUMNS IN A DATAFRAME

- Renaming An Existing Column:

```
laptops.rename(columns={'MANUFACTURER' : 'manufacturer'}, inplace=True)
```

- Converting A String Column To Float:

```
laptops["screen_size"] =  
laptops["screen_size"].str.replace("'",'').astype(float)
```

- Converting A String Column To Integer:

```
laptops["ram"] = laptops["ram"].str.replace('GB','')  
laptops["ram"] = laptops["ram"].astype(int)
```

STRING COLUMN OPERATIONS

- Extracting Values From The Beginning Of Strings:

```
laptops["gpu_manufacturer"] =  
(laptops["gpu"].str.split(n=1,expand=True).iloc[:,0] )
```

- Extracting Values From The End Of Strings:

```
laptops["cpu_speed_ghz"] =  
(laptops["cpu"].str.replace("GHz","").str.rsplit(n=1,expand=True).iloc[  
:,1].astype(float) )
```

- Reordering Columns And Exporting Cleaned Data:

```
specific_order = ['manufacturer', 'model_name', 'category',  
'screen_size_inches', 'screen', 'cpu', 'cpu_manufacturer', 'cpu_speed',  
'ram_gb', 'storage_1_type', 'storage_1_capacity_gb', 'storage_2_type',  
'storage_2_capacity_gb', 'gpu', 'gpu_manufacturer', 'os', 'os_version',  
'weight_kg', 'price_euros']  
  
reordered_df = laptops[specific_order]  
  
reordered_df.to_csv("laptops_cleaned.csv", index=False)
```

FIXING VALUES

- Replacing Values Using A Mapping Dictionary:

```
mapping_dict = {'Android': 'Android',  
  
                'Chrome OS': 'Chrome OS',  
  
                'Linux': 'Linux',  
  
                'Mac OS': 'macOS',  
  
                'No OS': 'No OS',  
  
                'Windows': 'Windows',  
  
                'macOS': 'macOS' }  
  
laptops["os"] = laptops["os"].map(mapping_dict)
```

- Dropping Missing Values:

```
laptops_no_null_rows = laptops.dropna(axis=0)
```

- Filling Missing Values:

```
laptops.loc[laptops["os"] == "No OS", "os_version"] = "No OS"
```

Resources

- [Python Encodings](#)
- [Indexing and Selecting Data](#)

Line Charts

Concepts

- To create line charts, we use the [matplotlib](#) library, which allows us to: quickly create common plots using high-level functions, extensively tweak plots, and create new kinds of plots from the ground up.
- By default, matplotlib displays a coordinate grid with the x-axis and y-axis values ranging from -0.6 to 0.6, no grid lines, and no data.
- Visual representations use visual objects like dots, shapes, and lines on a grid.
- Plots are a category of visual representation that allows us to easily understand the representation between variables.

Syntax

- Importing the pyplot module:

```
import matplotlib.pyplot as plt
```

- Displaying the plot in a Jupyter Notebook cell:

```
%matplotlib inline
```

- Generating and displaying the plot:

```
plt.plot()
```

```
plt.show()
```

- Generating a line chart:

```
plt.plot(first_twelve['DATE'], first_twelve['VALUE'])
```

- To rotate axis ticks:

```
plt.xticks(rotation=90)
```

- To add axis labels:

```
plt.xlabel('Month')
```

```
plt.ylabel('Unemployment Rate')
```

- To add a plot label:

```
plt.title('Monthly Unemployment Trends, 1948')
```

Resources

- [Documentation for pyplot](#)
- [Types of plots](#)

Multiple plots

Concepts

- A figure acts as a container for all of our plots and has methods for customizing the appearance and behavior for the plots within that container.
- Pyplot uses the following when we create a single plot:
 - A container for all plots was created (returned as a Figure object.)
 - A container for the plot was positioned on a grid (the plot returned as an Axes object.)
 - Visual symbols were added to the plot (using the Axes methods.)
- With each subplot, matplotlib generates a coordinate grid that was similar to the one we generated using the plot() function:
 - The x-axis and y-axis values ranging from 0.0 to 1.0.
 - No gridlines.
 - No data.

Syntax

- Creating a figure using the pyplot module:

```
fig = plt.figure()
```
- Adding a subplot to an existing figure with 2 plots and 1 column, one above the other:
 - Returns a new Axes object, which needs to be assigned to a variable:

```
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)
```
- Generating a line chart within an Axes object:

```
ax1.plot(unrate['DATE'][:12], unrate['VALUE'][:12])
ax2.plot(unrate['DATE'][12:24], unrate['VALUE'][12:24])
```
- Changing the dimensions of the figure with the figsize parameter (width x height):

```
fig = plt.figure(figsize=(12, 5))
```
- Specifying the color for a certain line using the c parameter:

```
plt.plot(unrate[0:12]['MONTH'], unrate[0:12]['VALUE'], c='red')
```

- Creating a legend using the pyplot module and specifying its location:

```
plt.legend(loc="upper left")
```

- Setting the title for an Axes object:

```
ax.set_title('Unemployment Trend, 1948')
```

Resources

- [Methods to Specify Color in Matplotlib](#)
- [Lifecycle of a Plot](#)

Bar Plots And Scatter Plots

Concepts

- A bar plot uses rectangular bars whose lengths are proportional to the values they represent.
 - Bar plots help us to locate the category that corresponds to the smallest or largest value.
 - Bar plots can either be horizontal or vertical.
 - Horizontal bar plots are useful for spotting the largest value.
- A scatter plot helps us determine if 2 columns are weakly or strongly correlated.

Syntax

- Generating a vertical bar plot:

```
pyplot.bar(bar_positions, bar_heights, width)
```

OR

```
Axes.bar(bar_positions, bar_heights, width)
```

- Using arange to return evenly separated values:

```
bar_positions = arange(5) + 0.75
```

- Using Axes.set_ticks(), which takes in a list of tick locations:

```
ax.set_ticks([1, 2, 3, 4, 5])
```

- Using Axes.set_xticklabels(), which takes in a list of labels:

```
Ax.set_xticklabels(['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars'], rotation = 90)
```

- Rotating the labels:

```
ax.set_xticklabels(['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars'], rotation = 90)
```

- Using Axes.scatter() to create a scatter plot:

```
ax.scatter(norm_reviews["Fandango_Ratingvalue"],
norm_reviews["RT_user_norm"])
```

Resources

- [Documentation for Axes.scatter\(\)](#)
- [Correlation Coefficient](#)

Histograms And Box Plots

Concepts

- Frequency distribution consists of unique values and corresponding frequencies.
- Bins are intervals of fixed length to cover all possible values.
- Histogram shows the distribution over numerical data.
- Quartiles divide the range of numerical values into four different regions.
- Box plot visually shows quartiles of a set of data as well as any outliers.
- Outliers are abnormal values that affect the overall observation of the data set due to their very high or low values.

Syntax

- Creating a frequency distribution:
`norm_reviews['Fandango_RatingValue'].value_counts()`
- Creating a histogram:
`ax.hist(norm_reviews['Fandango_RatingValue'])`
- Specifying the lower and upper range of bins within a histogram:
`ax.hist(norm_reviews['Fandango_RatingValue'], range=(0,5))`
- Setting y-axis limits:
`ax.set_ylim(0,50)`
- Setting number of bins for a histogram:
`ax.hist(norm_reviews['Fandango_RatingValue'], bins = 20)`
- Creating a box plot:
`ax.boxplot(norm_reviews["RT_user_norm"])`
- Creating a boxplot for multiple columns of data:
`num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue']
ax.boxplot(norm_reviews[num_cols].values)`

Resources

- [Documentation for histogram](#)
- [Documentation for boxplot](#)

- [Various ways to show distributions](#)

Improving Plot Aesthetics

Concepts

- Charjunk refers to all visual elements that does not help with understanding the data.
- Data-Ink Ratio is the fractional amount of ink used in showing data compared to the ink used to display the actual graphic.

Syntax

- Turning ticks off:

```
ax.tick_params(bottom="off", left="off", top="off", right="off")
```

- Removing Spines for the right axis:

```
ax.spines["right"].set_visible(False)
```

- Removing Spines for all axes:

```
for key, spine in ax.spines.items():  
    spine.set_visible(False)
```

Resources

- [5 Data Vizualition Best Practices](#)
- [Data-Ink Ratio](#)
- [Three Types of Chart Junk](#)

Color, Layout, and Annotations

Concepts

- RGB (Red, Green, Blue) color model describes how the three primary colors combine in any proportion to form any secondary color.
- You can specify a color using RGB values with each R, G, and B values out of one. For example: (0/255, 107/255, 164/255).

Syntax

- Using the linewidth parameter to alter width:

```
ax.plot(women_degrees['Year'], women_degrees[stem_cats[sp]],
        c=cb_dark_blue, label='Women', linewidth=3)
```

- Annotating using Axes.text():

- First parameter is the x coordinate.
 - Second parameter is the y coordinate.
 - Third parameter is a string of text.

```
ax.text(1970, 0, "Starting Point")
```

Resources

- [RGB Color Codes](#)
- [Documentation for pyplot module](#)
- [Documentation for Axes.Text](#)

Conditional Plots

Concepts

- Seaborn:
 - Is built on top of matplotlib.
 - Has good support for more complex plots.
 - Attractive default styles.
 - Integrates well with the pandas library.
- Seaborn creates a matplotlib figure or adds to the current existing figure.
- Seaborn stylesheets:
 - **darkgrid**: Coordinate grid displayed, dark background color.
 - **whitegrid**: Coordinate grid displayed, white background color.
 - **dark**: Coordinate grid hidden, dark background color.
 - **white**: Coordinate grid hidden, white background color.
 - **ticks**: Coordinate grid hidden, white background color, ticks visible.

Syntax

- Importing the seaborn module:

```
import seaborn as sns
```
- Creating a distribution plot:

```
sns.distplot(titanic['Fare'])
```
- Creating a kernel density plot:

```
sns.kdeplot(titanic['Fare'])
```
- Shading underneath the curve of a kernel density plot:

```
sns.kdeplot(titanic['Fare'], shade=True)
```
- Setting the Seaborn Style Sheet:

```
sns.set_style("white")
```
- Removing the Spines
 - Default: Only top and right Spines are removed.

```
sns.despine(left=True, bottom=True)
```

- Generating a grid of data containing a subset of the data for different values:

```
g = sns.Facetgrid(titanic, col = "Pclass", size = 6)
```
- Generating a grid of data with multiple subsets of the data:

```
g = sns.Facetgrid(titanic, col = "Pclass", row = "Survived", size = 6)
```
- Adding different colors for unique values using hue:

```
g = sns.Facetgrid(titanic, col = "Pclass", row = "Survived", hue =
"Sex", size = 6)
```
- Using a grid of data and mapping it onto a Seaborn object:

```
g.map(sns.kdeplot, "Age", shade=True)
```
- Adding a legend to the grid of data:

```
g.add_legend()
```

Resources

- [Different Seaborn Plots](#)
- [3D Surface Plots](#)
- [Why is Seaborn sns](#)

Visualizing Geographic Data

Concepts

- Latitude runs North to South and ranges from -90 to 90 degrees.
- Longitude runs East to West and ranges from -180 to 180 degrees.
- Map projections project points on a sphere onto a 2D plane.
- Basemap makes it easy to work with geographical data.
- You'll want to import matplotlib.pyplot because Basemap is an extension of matplotlib.
- Matplotlib classes can be used to customize the appearance of map.
- Great Circle is the shortest circle connecting 2 points on a sphere, and it shows up as a line on a 2D projection.

Syntax

- Importing Basemap:

```
from mpl_toolkits.basemap import Basemap
```

- Using the Basemap constructor:

```
m = Basemap('merc', -80, 80, -180, 180)
```

- Converting Longitude and Latitude to Cartesian:

- Use basemap constructor for conversion.

- Only accepts list values.

```
x, y = m(airports["longitude"].tolist(), airports["latitude"].tolist())
```

- Using the scatter attribute of Basemap:

```
m.scatter(x, y)
```

- Adjusting the size of the scatter marker:

```
m.scatter(x, y, s=5)
```

- Drawing coastlines on the Basemap object:

```
m.drawcoastlines()
```

- Drawing Great Circles on the Basemap object:

```
m.drawgreatcircles(startlon, startlat, endlon, endlat)
```

Resources

- [Geographic Data with Basemap](#)

- [Basemap Toolkit Documentation](#)
- [Plotting Data on a Map](#)

Data Aggregation

Concepts

- **Aggregation** is applying a statistical operation to groups of data. It reduces dimensionality so that the dataframe returned will contain just one value for each group. The aggregation process can be broken down into three steps:
 - Split the dataframe into groups.
 - Apply a function to each group.
 - Combine the results into one data structure.
- The **groupby** operation optimizes the split-apply-combine process. It can be broken down into two steps:
 - Create a GroupBy object.
 - Call an aggregation function.
- Creating the GroupBy object is an intermediate step that allows us to optimize our work. It contains information on how to group the dataframe, but nothing is actually computed until a function is called.
- The **DataFrame.pivot_table()** method can also be used to aggregate data. We can also use it to calculate the grand total for the aggregation column.

Syntax

GROUPBY OBJECTS

- Create a GroupBy object:

```
df.groupby('col_to_groupby')
```
- Select one column from a GroupBy object:

```
df.groupby('col_to_groupby')['col_selected']
```
- Select multiple columns from a GroupBy object:

```
df.groupby('col_to_groupby')[['col_selected1', 'col_selected2']]
```

COMMON AGGREGATION METHODS

- `mean()`: Calculates the mean of groups
- `sum()`: Calculates the sum of group values
- `size()`: Calculates the size of groups
- `count()`: Calculates the count of values in groups

- `min()`: Calculates the minimum of group values
- `max()`: Calculates the maximum of group values

GROUPBY.AGG() METHOD

- Apply one function to a GroupBy object:

```
df.groupby('col_to_groupby').agg(function_name)
```

- Apply multiple functions to a GroupBy object:

```
df.groupby('col_to_groupby').agg([function_name1, function_name2,
function_name3])
```

- Apply a custom function to a GroupBy object:

```
df.groupby('col_to_groupby').agg(custom_function)
```

AGGREGATION WITH THE DATAFRAME.PIVOT_TABLE METHOD

- Apply only one function:

```
df.pivot_table(values='Col_to_aggregate', index='Col_to_group_by',
aggfunc=function_name)
```

- Apply multiple functions:

```
df.pivot_table('Col_to_aggregate', 'Col_to_group_by',
aggfunc=[function_name1, function_name2, function_name3])
```

- Aggregate multiple columns:

```
df.pivot_table(['Col_to_aggregate1', 'Col_to_aggregate2'],
'Col_to_group_by', aggfunc = function_name)
```

- Calculate the grand total for the aggregation column:

```
df.pivot_table('Col_to_aggregate', 'Col_to_group_by',
aggfunc=function_name, margins=True)
```

Combining Data with Pandas

Concepts

- A key or join key is a shared index or column that is used to combine dataframes together.
- There are four kinds of joins:
 - Inner: Returns the intersection of keys, or common values.
 - Outer: Returns the union of keys, or all values from each dataframe.
 - Left: Includes all of the rows from the left dataframe, along with any rows from the right dataframe with a common key. The result retains all columns from both of the original dataframes.
 - Right: Includes all of the rows from the right dataframe, along with any rows from the left dataframe with a common key. The result retains all columns from both of the original dataframes. This join type is rarely used.
- The **pd.concat()** function can combine multiple dataframes at once and is commonly used to "stack" dataframes or combine them vertically (axis=0). The **pd.merge()** function uses keys to perform database-style joins. It can only combine two dataframes at a time and can only merge dataframes horizontally (axis=1).

Syntax

CONCAT() FUNCTION

- Concatenate dataframes vertically (axis=0):
`pd.concat([df1, df2])`
- Concatenate dataframes horizontally (axis=1):
`pd.concat([df1, df2], axis=1)`
- Concatenate dataframes with an inner join:
`pd.concat([df1, df2], join='inner')`

MERGE() FUNCTION

- Join dataframes on index:
`pd.merge(left=df1, right = df2, left_index=True, right_index=True)`
- Customize the suffix of columns contained in both dataframes:
`pd.merge(left=df1, right=df2, left_index=True, right_index=True, suffixes=('left_df_suffix', 'right_df_suffix'))`
- Change the join type to left, right, or outer:

```
pd.merge(left= df1, right=df2, how='join_type', left_index=True,  
right_index=True)
```

- Join dataframes on a specific column:

```
pd.merge(left=df1, right=df2, on='Column_Name')
```

Resources

- [Merge and Concatenate](#)

Transforming Data with Pandas

Concepts

- The **Series.apply()** and **Series.map()** methods can be used to apply a function element-wise to a *series*. The **DataFrame.applymap()** method can be used to apply a function element-wise to a *dataframe*.
- The **DataFrame.apply()** method has different capabilities than the **Series.apply()** method. Instead of applying functions element-wise, the **df.apply()** method applies functions along an axis, either column-wise or row-wise. When we create a function to use with **df.apply()**, we set it up to accept a Series, most commonly a column.
- Use the **apply()** method when a vectorized function does not exist because a vectorized function can perform an equivalent task faster than the **apply()** method. Sometimes, it may be necessary to reshape a dataframe to use a vectorized method.

Syntax

APPLYING FUNCTIONS ELEMENT-WISE

- Apply a function element-wise to a series:

```
df[col_name].apply(function_name)  
df[col_name].map(function_name)
```

- Apply a function element-wise to a dataframe:

```
df.applymap(function_name)
```

APPLYING FUNCTIONS ALONG AN AXIS

- Apply a function along an axis, column-wise:

```
df.apply(function_name)
```

RESHAPING DATAFRAMES

- Reshape a dataframe:

```
pd.melt(df, id_vars=[col1, col2], value_vars=[col3, col4])
```

Working with Strings In Pandas

Concepts

- Pandas has built in several vectorized methods that perform the same operations for strings in Series as Python string methods.
- A regular expression is a sequence of characters that describes a search pattern. In pandas, regular expressions is integrated with vectorized string methods to make finding and extracting patterns of characters easier.

Syntax

REGULAR EXPRESSIONS

- To match multiple characters, specify the characters between "[]":
`pattern = r"[Nn]ational accounts"`
 - This expression would match "national accounts" and "National accounts".
- To match a range of characters or numbers, use:
`pattern = r"[0-9]"`
 - This expression would match any number between 0 and 9.
- To create a capturing group, or indicate that only the character pattern matched should be extracted, specify the characters between "()" :
`pattern = r"([1-2][0-9][0-9][0-9])"`
 - This expression would match years.
- To repeat characters, use "{}". To repeat the pattern "[0-9]" three times:
`pattern = r"([1-2][0-9){3})"`
 - This expression would also match years.
- To name a capturing group, use:
`pattern = r"(?P<Years>[1-2][0-9]{3})"`
 - This expression would match years and name the capturing group "Years".

VECTORIZED STRING METHODS

- Find specific strings or substrings in a column:

```
df[col_name].str.contains(pattern)
```

- Extract specific strings or substrings in a column:
`df[col_name].str.extract(pattern)`
- Extract more than one group of patterns from a column:
`df[col_name].str.extractall(pattern)`
- Replace a regex or string in a column with another string:
`df[col_name].str.replace(pattern, replacement_string)`

Resources

- [Working with Text Data](#)
- [Regular Expressions](#)

Working with Missing and Duplicate Data

Concepts

- Missing or duplicate data may exist in a data set for many reasons. Sometimes, they may exist because of user input errors or data conversion issues; other times, they may be introduced while performing data cleaning tasks. In the case of missing values, they may also exist in the original data set to purposely indicate that data is unavailable.
- In pandas, missing values are generally represented by the `NaN` value or the `None` value.
- To handle missing values, first check for errors made while performing data cleaning tasks. Then, try to use available data from other sources (if it exists) to fill them in. Otherwise, consider dropping them or replacing them with other values.

Syntax

IDENTIFYING MISSING VALUES

- Identify rows with missing values in a specific column:
`missing = df[col_name].isnull()`
`df[missing]`
- Calculate the number of missing values in each column:
`df.isnull().sum()`

REMOVING MISSING VALUES

- Drop rows with any missing values:
`df.dropna()`
- Drop specific columns:
`df.drop(columns_to_drop, axis=1)`
- Drop columns with less than a certain number of non-null values:
`df.dropna(thresh = min_nonnull, axis=1)`

REPLACING MISSING VALUES

- Replace missing values in a column with another value:
`df[col_name].fillna(replacement_value)`

VISUALIZING MISSING DATA

- Use a heatmap to visualize missing data:

```
import seaborn as sns  
sns.heatmap(df.isnull(), cbar=False)
```

CORRECTING DUPLICATE VALUES

- Identify duplicates values:

```
dups = df.duplicated()  
df[dups]
```

- Identify rows with duplicate values in only certain columns:

```
dups = df.duplicated([col_1, col_2])  
df[dups]
```

- Drop duplicate values. Keep the first duplicate row:

```
df.drop_duplicates()
```

- Drop rows with duplicate values in only certain columns. Keep the last duplicate row:

```
combined.drop_duplicates([col_1, col_2], keep='last')
```

Resources

- [Working with Missing Data](#)

Data Cleaning Walkthrough

Concepts

- A data science project usually consists of either an exploration and analysis of a set of data or an operational system that generates predictions based on data the updates continually.
- When deciding on a topic for a project, it's best to go with something you're interested in.
- In real-world data science, you may not find an ideal dataset to work with.

Syntax

- Combining dataframes:

```
z = pd.concat([x,y])
```

- Copying or adding columns:

```
survey["new_column"] = survey["old_column"]
```

- Filtering to keep columns:

```
survey_fields = ["DBN", "rr_s", "rr_t"]
survey = survey.loc[:,survey_fields]
```

- Adding 0s to the front of the string until the string has desired length:

```
zfill(5)
```

- Applying function to Series:

```
data["class_size"]["padded_csd"] =
data["class_size"]["CSD"].apply(pad_csd)
```

- Converting a column to numeric data type:

```
data["sat_results"]["SAT Math Avg. Score"] =
pd.to_numeric(data["sat_results"]["SAT Math Avg. Score"])
```

Resources

- [Data.gov](https://www.data.gov)
- [/r/datasets](https://www.reddit.com/r/datasets)
- [Awesome datasets](https://www.awesomedatasets.com/)
- rs.io

Data Cleaning Walkthrough: Combining the Data

Concepts

- Merging data in Pandas supports four types of joins -- **left**, **right**, **inner**, and **outer**.
- Each of the join types dictates how pandas combines the rows.
- The strategy for merging affects the number of rows we end up with.
- We can use one or multiple aggregate functions on a grouped dataframe.

Syntax

- Resetting the index:

```
class_size.reset_index(inplace=True)
```

- Grouping a dataframe by column:

```
class_size=class_size.groupby("DBN")
```

- Aggregating a grouped Dataframe:

```
class_size = class_size.agg(numpy.mean)
```

- Displaying column types:

```
data["ap_2010"].dtypes
```

- Performing a left join:

```
combined.merge(data["ap_2010"], on="DBN", how="left")
```

- Displaying the shape of the dataframe (row, column):

```
combined.shape
```

- Performing an inner join:

```
combined = combined.merge(data[class_size], on="DBN", how="inner")
```

- Filling in missing values:

```
combined.fillna(0)
```

Resources

- [Dataframe.groupby\(\)](#)
- [agg\(\) documentation](#)

Data Cleaning Walkthrough: Analyzing and Visualizing the Data

Concepts

- An r value measures how closely two sequences of numbers are correlated.
- An r value ranges from **-1 to 1**.
- An r value closer to **-1** tells us the two columns are negatively correlated while an r value closer to **1** tells us the columns are positively correlated.
- The r value is also called Pearson's correlation coefficient.
- Keyword arguments for **scatter()** method:
 - **s** : Determines the size of the point that represents each school on the map.
 - **zorder** : Determines where the method draws the points on the z axis. In other words, it determines the order of the layers on the map.
 - **latlon** : A Boolean value that specifies whether we're passing in latitude and longitude coordinates instead of x and y plot coordinates.

Syntax

- Finding correlations between columns in a dataframe:
`combined.corr()`
- Specifying a plot type using Dataframe.plot():
`combined.plot.scatter(x='total_enrollment', y='sat_score')`
- Creating a map of New York City:

```
from mpl_toolkits.basemap import Basemap  
  
m = Basemap( projection='merc', llcrnrlat=40.496044,  
urcrnrlat=40.915256, llcrnrlon=-74.255735, urcrnrlon=-73.700272,  
resolution='i' )  
  
m.drawmapboundary(fill_color='#85A6D9')  
  
m.drawcoastlines(color='#6D5F47', linewidth=.4)  
  
m.drawrivers(color='#6D5F47', linewidth=.4)
```
- Converting a Pandas series to list:
`longitudes = combined["lon"].tolist()`

- Making a scatterplot using Basemap:

```
m.scatter(longitudes, latitudes, s=20, zorder=2, latlon=True)
```

Resources

- [R value](#)
- [pandas.DataFrame.plot\(\)](#)
- [Correlation](#)
- [Guess the Correlation](#)

The Command Line

Command Line Basics

Concepts

- Both Linux and OS X are based on an operating system called UNIX and have similar terminals.
- Before GUIs (Graphical User Interfaces) came along, the most common way for a person to interact with their computer was through the command line interface.
- A command line interface lets us navigate folders and launch programs by typing commands.
- Computers store files in directories, or folders.
- The root directory, represented by a forward slash, is the top-level directory of any UNIX system.
- An absolute path always begins with a forward slash that's written in relation to the root directory.
- A relative path is relative to the directory we're in.
- Verbose mode displays the specific action of a Bash command when it is executed.
- Commands have options that can modify their behavior.

Syntax

- Print working directory
`pwd`
- Change directories
`cd`
- Check logged in user
`whoami`
- Switch to home directory
`cd ~`
- Switch to root directory
`cd /`
- Make folder in directory
`mkdir [directory name]`

- Turn on 'verbose' mode for mkdir
`mkdir -v [directory name]`
- Help for any command
`[command] --help`
- List all the files in folders in a directory
`ls`
- Remove a directory
`rmdir [directory name]`

Resources

- [Command line options](#)
- [Run UNIX/Linux Commands on Windows](#)

Working with Files

Concepts

- Every program writes to standard output and receives input through standard output.
- If the program throws an error while running, it writes it to standard error.
- **stderr** and **stdout** usually display on the monitor, while **stdin** is the input from the keyboard.
- **stdout**, **stderr**, and **stdin** exist because these standard streams allow the interfaces to be abstract.
- We can redirect standard streams to connect them to different sources.
- In Unix, every file and folder has certain permissions associated with it. These permissions have three scopes:
 - **owner** : The user who created the file or folder
 - **group** : Users in the owner's group (on Unix systems, an owner can place users in groups)
 - **everyone** : All other users on the system who aren't the user or in the user's group
- Each scope can have any of three permissions (a scope can have multiple permissions at once):
 - **read** : The ability to see what's in a file (if defined on a folder, the ability to see what files are in a folder)
 - **write** : The ability to modify a file (if a folder, the ability to delete, modify, and rename files in the folder)
 - **execute** : The ability to run a file (some files are executable, and need this permission to run)
- Each permission can be granted or denied to each scope.
- The character for read is **r**, the character for write is **w**, and the character for execute is **x**.
- If a scope doesn't have a permission, a dash takes the place of it instead.
- We can use octal notation to represent permissions for all scopes with 4 digits.
 - **---** : No permissions; corresponds to 0
 - **--x** : Execute only permission; corresponds to 1
 - **-w-** : Write only permissions; corresponds to 2

- **-wx** : Write and execute permissions; corresponds to 3
 - **r--** : Read only permissions; corresponds to 4
 - **r-x** : Read and execute permissions; corresponds to 5
 - **rw-** : Read and write permissions; corresponds to 6
 - **rwx** : Read, write, and execute permissions; corresponds to 7
- Files typically have extensions like .txt and .csv that indicate the file type.
- Rather than relying on extensions to determine file type, Unix-based operating systems like Linux use media types, which are also called MIME types.
- The root user has all permissions and access to all files by default.

Syntax

- Create a file
`touch [name of file]`
- Print text
`echo [string of text]`
- Write text to file
`echo [string of text] > [name of file]`
- Edit a file without redirection
`nano [name of file]`
- View permissions on files and folders
`ls -l`
- Get info about a file
`stat [file name]`
- Modify file permissions
`chmod [octal notation integer] [file name]`
- Move file
`mv [file name] [destination path]`
- Copy file
`cp [file name] [new file name]`
- Delete file
`rm [name of file]`

- Switch and run as root user

sudo

Resources

- [Standard streams](#)
- [Octal](#)

Working with Programs

Concepts

- A shell is a way to access and control a computer.
- Bash is the most popular of the UNIX shells, and the default on most Linux and OS X computers.
- Command line shells allow us the option of running commands and viewing the results, as opposed to clicking an icon in a graphical shell.
- Bash is essentially a language, which allows us to run other programs.
- A command language is a special kind of programming language through which we can control applications and the system as a whole.
- Quotations around strings in Bash are optional, unless they contain a space.
- We can set variables on the command line by assigning values to them.
- In the command line environment, variables consist entirely of uppercase characters, numbers, and underscores.
- We can assign any data type to a variable.
- Accessing the value of a variable is not much different than Python -- we just need to prepend it with a dollar sign (\$).
- Environment variables are variables you can access outside the shell.
- We can run many programs from Bash, including Python.
- `os.environ` is a dictionary containing all of the values for the environment variables.
- The `PATH` environment variable is configured to point to several folders.
- Programs are similar to functions, and can have any number of arguments.
- Programs can also have optional flags, which modify program behavior.
- We can chain multiple flags that have single, short, character names.

Syntax

- Assign a variable

```
OS = Linux
```

OR

```
OS = "Linux"
```

- Print a variable

- Create an environment variable
`echo $OS`
- Run python inside of Bash
`python`
- Access environment variables
`import os`
`print(os.environ["LINUX"])`
- See what folders are in PATH
`echo $PATH`
- List files in directory in long mode
`ls -l`
- Specify a longer flag with two dashes
`ls --ignore`

Resources

- [UNIX Shells](#)
- [Environment Variables](#)

Command Line Python Scripting

Concepts

- Command line python interpreter good for testing snippets of code quickly, as well as debugging.
- Command line python interpreter not good for developing Python programs.
- Common way to develop with Python: use an IDE or text editor to create Python files, and then run from command line.
- You can enter the default Python executable using **python**.
- We can access Python 3 using the **python3** executable.
- Packages are an important way to extend Python's functionality.
- Pip is the best way to install packages from the command line.
- Virtual Environments allows us to have a certain version of Python and other packages without worrying about how it will affect other projects on our system.
- By default, virtualenv will use the **python** executable.
- We can import functions from a package into a file as well as functions and classes into another file.

Syntax

- Install Python packages
`pip install [package name]`
- Upgrade pip
`pip install --upgrade pip`
- Install the virtualenv tool
`pip install virtualenv`
 - Note: Python 3 should already have the venv module.
- Create a virtual environment
`virtualenv [name of environment]`
- Change the Python interpreter
`virtualenv -p [path to desired Python] [name of environment]`
- Activate virtual environment
`source [name of environment]/bin/activate`

- Check Python version
`python -v`
- Check installed packages and version
`pip freeze`
- Switch a virtualenv off
`deactivate`

Resources

- [Python Package Index](#)
- [Python Virtual Environments - A Primer](#)

Working with Jupyter console

Concepts

- Jupyter is an enhanced Python interpreter that makes working with data easier.
- Shells are useful for when you need to quickly test some code, explore datasets, and perform basic analysis.
- The main difference between Jupyter console and Jupyter notebook is that the console functions in interactive mode.
- Magics are special Jupyter commands that always start with %. Jupyter magics enable to you to access Jupyter-specific functionality, without Python executing your commands.
- Autocomplete makes it quicker to write code and lead to discovery of new methods. Trigger autocomplete by pressing the TAB key while typing a variable's name. Press TAB after typing variable name to show the methods.

Syntax

- Opening the Jupyter console:
`ipython`
- Getting an overview of IPython's features:
`?`
- Accessing Python's help system:
`help()`
- Displaying the documentation for an object:
`help(obj)`
- Exiting the Jupyter console:
`exit`
- Running an external Python script:
`%run test.py`
- Opening a file editor:
`%edit`
- Opening an interactive debugger:
`%debug`
- Showing the last few commands:

```
%history
```

- Saving the last few commands:

```
%save
```

- Printing all variable names:

```
%who
```

- Resetting the IPython session:

```
%reset
```

- Showing the contents of the current directory:

```
!ls
```

- Executing code from the clipboard:

```
%paste
```

- Opening editing area where you can paste in code from your clipboard:

```
%cpaste
```

Resources

- [IPython Documentation](#)
- [Jupyter magics](#)

Piping and redirecting output

Concepts

- The ? wildcard character is used to represent a single, unknown character.
- The * wildcard character is used to represent any number of characters.
- We can use the pipe character (|) to send the standard output of one command to the standard output of another command.
- Escape characters tell the shell to treat the character coming directly after it as a plain character.

Syntax

- Redirecting standard output to overwrite a file:

```
echo "Dataquest is best!" > best.txt
```

- Redirecting standard output to add text to a file:

```
echo "Dataquest is awesome!" >> awesome.txt
```

- Sorting the lines of a file in alphabetical order:

```
sort < beer.txt
```

- Sorting the lines of a file in reverse alphabetical order:

```
sort -r < beer.txt
```

- Searching through contents of a file to find a specific instance of text:

```
grep "pass" beer.txt
```

- Using a wildcard character to represent a single character when searching:

```
grep "beer" beer?.txt
```

- Using a wildcard character to represent any number of characters:

```
grep "beer" *.txt
```

- Redirecting standard input to standard output:

```
tail -n 10 logs.txt | grep "Error"
```

- Printing the contents of a file:

```
cat beer.txt
```

- Running two commands sequentially:

```
echo "All the beers are gone" >> beer.txt && cat beer.txt
```

- Using a backslash escape character to add a quote to a file:

```
echo "\"Get out of here,\" said Neil Armstrong to the moon people." >>  
famous_quotes.txt
```

Resources

- [Escape Characters](#)
- [Wildcard Characters](#)

Data Cleaning and Exploration Using Csvkit

Concepts

- **csvkit** supercharges your workflow by adding command line tools specifically for working with **CSV** files.
- **csvstack** stacks rows from multiple CSV files.
- **csvlook** renders CSV in pretty table format.
- **csvcut** selects specific columns from a CSV file.
- **csvstat** calculates descriptive statistics for some or all columns.
- **csvgrep** filters tabular data using specific criteria

Syntax

- Installing CSVkit:
`sudo pip install csvkit`
- Consolidating rows from multiple CSV files into one new file:
`csvstack file1.csv file2.csv file3.csv > final.csv`
- Adding a new column:
`csvstack -n origin file1.csv file2.csv file3.csv > final.csv`
- Specifying a grouping value for each filename:
`csvstack -g 1,2,3 file2.csv file3.csv > final.csv`
- Returning standard input in a formatted table representation:
`head -10 final.csv | csvlook`
- Displaying all column names along with a unique integer identifier:
`csvcut -n Combined_hud.csv`
- Displaying the first five values of a specific column:
`csvcut -c 1 Combined_hud.csv | head -5`
- Calculating summary statistics for a column:
`csvcut -c 2 Combined_hud.csv | csvstat`
- Calculating the mean value for all columns:

```
csvstat --mean Combined_hud.csv
```

- Finding all rows in a column that match a specific pattern:

```
csvgrep -c 2 -m -9 Combined_hud.csv
```

- Selecting rows that do not match a specific pattern:

```
csvgrep -c 2 -m -9 -i Combined_hud.csv
```

Resources

- [CSVkit documentation](#)
- [Working with CSVkit](#)

Introduction to Git

Concepts

- Distributed version control systems will "merge" changes together intelligent and exist to enable multiple developers to work on a project at the same time.
- A repository (or "repo") tracks multiple versions of the files in the folder, enabling collaboration.
- While there are multiple distributed version control systems, Git is the most popular.
- Files and folders with a period prefix are typically private.
- Commits are checkpoints that you store after adding files and/or making changes.
- A diff are the changes between commits.
- Files can have one of three states in Git:
 - **committed** : The current version of the file has been added to a commit, and Git has stored it.
 - **staged** : The file has been marked for inclusion in the next commit, but hasn't been committed yet (and Git hasn't stored it yet). You might stage one file before working on a second file, for example, then commit both files at the same time when you're done.
 - **modified** : The file has been modified since the last commit, but isn't staged yet.

Syntax

- Getting started with Git
`git`
- Initializing a repo
`git init`
- Check the state of each file
`git status`
- Add files to staging area
`git add`
- Configure identity in Git
 - Configure email
`git config --global user.email "your.email@domain.com"`
 - Configure name
`git config --global user.name "Your name"`

- Making a commit
`git commit -m "Commit message here"`
- Viewing the diff
 - View the diff before staged
`git diff`
 - View the diff after staged
`git diff --staged`
- View repo's commit history
`git log`

Resources

- [Git Documentation](#)
- [GitHub's Blog](#)

Git Remotes

Concepts

- Pushing code to remote repositories allows us to:
 - Share code with others and build a portfolio.
 - Collaborate with others on a project and build code together.
 - Download and use code others have created.
- GitHub is the most useful way to use Git.
- Markdown allows us to create lists and other complex but useful structures in plain text.
- Most GitHub projects contain a README Markdown file, which helps people understand what the project is and how to install it.
- A branch contains a slightly different version of the code, and are created when developers want to work on a new feature for a project.
- The master branch is the main branch of the repo and is usually the most up-to-date shared version of any code project.
- A remote repo will almost always have the name origin.
- Each commit has a unique commit hash so we can refer to it later.

Syntax

- To clone a repo
`git clone https://github.com/amznlabs/amazon-dsstne.git`
- View all the branches in the repo
`git branch`
- To push local repo to remote repo
`git push origin [branchname]`
- List all the repo's remotes
`git remote`
- See full commit history
`git log`
- See the specific change in a commit
`git show [hash]`

- Switch between commits in local repo
`git reset`
- Update current branch with latest commits
`git pull`

Resources

- [GitHub](#)
- [Anatomy of a Git Commit](#)

Git Branches

Concepts

- Branches allow us to create several different work areas within the same repo.
- Switching branches is useful when we want to work on changes to a project that require different amounts of development time.
- Git will prevent you from switching to a different branch if there is a potential merge conflict with the branch we're switching to.
- Git uses HEAD to refer to the current branch, as well the branch with the latest commit in that branch.
- Django is a popular Web framework for Python that programmers develop using Git and GitHub.
- In order to merge branch B into branch A, switch to branch A then merge the branch.
- Pull requests will show us the differences between branches in an attractive interface, and allow other developers to add comments.
- Typical branch names:
 - Feature : **feature/happy-bot**
 - Fix : **fix/remove-error**
 - Chore : **chore/add-analytics**

Syntax

- Create a branch
`git branch [branch name]`
- Switch to branch
`git checkout [branch name]`
- Create and switch to branch
`git checkout -b [branch name]`
- Show all the remote branches
`git branch -r`
- Show all the branches available locally
`git branch -a`
- Merge a branch
`git merge`

- Delete a branch
`git branch -d [branch name]`
- Update Git's list of branch names and commits
`git fetch`

Resources

- [Pull Requests](#)
- [Django](#)

Merge Conflicts

Concepts

- Git is designed to preserve everyone's work.
- Merge conflicts arise when you have commits in your current branch that aren't in your other branch.
- Git adds markup lines to the problem files where the conflicts occur.
- Aborting resets the working directory and Git history to the state before the merge.
- With multi-line conflicts, they're placed into a code block in a single conflict.
- To resolve a merge conflict remove the markup and any conflicting lines we don't want to keep.
- The period character at the end of the checkout commands is a wildcard that means all files.
- .gitignore is a file that contains a list of all files that Git should ignore when adding to the staging area and committing
- Removing files from the Git cache will prevent Git from tracking changes to those files, and adding them to future commits.
- A wildcard character is a placeholder represented by a single character, which can be represented as any number of characters.

Syntax

RESOLVE CONFLICT METHODS

- Abort the merge
`git merge --abort`
- Use graphical merge tools
`git mergetool`
- Declare whose files to keep
`git checkout --ours .`
OR
`git checkout --theirs .`

GIT CACHE

- Remove files from Git cache
`git rm --cached`

Resources

- [How Git creates and handles merge conflicts](#)
- [Examples of wildcard characters](#)

Working with Data Sources

Introduction to SQL

Concepts

- A database is a data representation that lives on disk, and can be queried, accessed, and updated without using much memory.
- A database management system (DBMS) can be used to interact with a database. Examples include Postgres and SQLite. SQLite is the most popular database in the world and is lightweight enough that the SQLite DBMS is included as a module in Python.
- To work with data stored in a database, we instead use a language called SQL (or structured query language).

Syntax

- Returning first 10 rows from a table:

```
SELECT *
FROM recent_grads
LIMIT 10;
```

- Filtering return results:

```
SELECT Major, ShareWomen
FROM recent_grads
WHERE ShareWomen < 0.5;
```

- Filtering results using multiple criteria:

```
SELECT Major, Major_category, Median, ShareWomen
FROM recent_grads
WHERE ShareWomen > 0.5 AND Median > 50000;
```

- Filtering results using the OR clause:

```
SELECT Major, Median, Unemployed
FROM recent_grads
WHERE Median >= 10000 OR Unemployed <= 1000
LIMIT 20;
```

- Grouping using AND and OR with parentheses:

```
SELECT Major, Major_category, ShareWomen, Unemployment_rate
FROM recent_grads
WHERE (Major_category = 'Engineering') AND (ShareWomen > 0.5 OR
Unemployment_rate < 0.051);
```

- Sorting results:

```
SELECT Major, ShareWomen, Unemployment_rate
FROM recent_grads
WHERE ShareWomen > 0.3 AND Unemployment_rate < 0.1
ORDER BY ShareWomen DESC;
```

Resources

- [W3 Schools](#)
- [SQL Zoo](#)

Summary Statistics

Concepts

- Summary statistics are used to summarize a set of observations.
- Everything is considered a table in SQL. One advantage of this simplification is that it's a common and visual representation that makes SQL approachable for a much wider audience.
- Datasets and calculations that aren't well suited for a table representation must be converted to be used in a SQL database environment.
- Aggregate functions are applied over columns of values and return a single value.
- The **COUNT** clause can be used on any column while aggregate functions can only be used on numeric columns.
- SQL supports standard arithmetic operators (+, -, *, /).
- Arithmetic between two floats returns a float.
- Arithmetic between a float and an integer returns a float.
- Arithmetic between two integers returns an integer.

Syntax

- Returning a count of rows in a table:

```
SELECT COUNT(Major)  
FROM recent_grads
```

- Returning the minimum value of a table:

```
SELECT MIN(ShareWomen)  
FROM recent_grads;
```

- Computing the sum of a column as an integer:

```
SELECT SUM(Total)  
FROM recent_grads
```

- Computing the sum of a column as a float value:

```
SELECT TOTAL(Total)  
FROM recent_grads
```

- Specifying a name for a column in the results:

```
SELECT (*) AS "Total Majors" # also works without AS
```

```
FROM recent_grads
```

- Returning the unique values of a column:

```
SELECT DISTINCT Major_category
```

```
FROM recent_grads
```

- Performing an arithmetic operation on a table:

```
SELECT P75th - P25th quartile_spread
```

```
FROM recent_grads
```

Resources

- [Aggregate Functions](#)
- [Summary Statistics](#)

Group Summary Statistics

Concepts

- The **GROUP BY** clause allows you to compute summary statistics by group.
- The **HAVING** clause filters on the virtual column that **GROUP BY** generates.
- **WHERE** filters results before the aggregation, whereas **HAVING** filters after aggregation.
- The **ROUND** function rounds the results to desired decimal places.
- **PRAGMA TABLE_INFO()** returns the type, along with other information for each column.
- The **CAST** function in SQL converts data from one data type to another. For example, we can use the **CAST** function to convert numeric data into character string data.

Syntax

- Computing summary statistics by a unique value in a row:

```
SELECT SUM(Employed)  
FROM recent_grads  
GROUP BY Major_category;
```

- Filtering results after aggregation:

```
SELECT Major_category, AVG(Employed) / AVG(Total) AS share_employed  
FROM recent_grads  
GROUP BY Major_category  
HAVING share_employed > 0.8;
```

- Rounding a column to two decimal places:

```
SELECT Major_category, ROUND(ShareWomen, 2) AS rounded_share_women  
FROM recent_grads;
```

- Generating information about each column in a table:

```
PRAGMA TABLE_INFO(recent_grads);
```

- Converting, known as casting, a column to a float type:

```
SELECT CAST(Women as Float) / CAST(Total as Float)  
FROM recent_grads;
```

Resource

- [PRAGMA TABLE_INFO](#)

- [Core functions of SQLite](#)

Subqueries

Concepts

- SQL is a declarative-programming language. Designers of SQL wants its users to focus on expressing computations over variable names.
- A subquery is a query nested within another query and must always be contained within parentheses.
- A subquery can exist within the **SELECT**, **FROM** or **WHERE** clause.
- We can use **IN** to specify a list of values we want to match against.
- When writing queries that have subqueries, we'll want to write our inner queries first.
- The subquery gets executed first whenever the query gets ran.

Syntax

- Writing subqueries:

```
SELECT Major, ShareWomen FROM recent_grads WHERE ShareWomen >
    (SELECT AVG(ShareWomen)
        FROM recent_grads
    )
```

- Returning rows that match a list of values:

```
SELECT Major, Major_category FROM recent_grads
WHERE Major_category IN ('Business', 'Engineering')
```

Resources

- [SQLite Documentaion](#)
- [Subqueries](#)

Querying SQLite from Python

Concepts

- SQLite is a database that doesn't require a standalone server and stores an entire database on a single computer.
- We can interact with SQLite database in two ways:
 - With the sqlite3 Python module.
 - With the SQLite shell.
- A Connection instance maintains the connection to the database we want to work with.
- When connected to a database, SQLite locks the database file.
- We use a Cursor class to:
 - Run a query against the database.
 - Parse the results from the database.
 - Convert the results to native python objects.
 - Store the results within the Cursor instance as a local variable.
- A tuple is a core data structure that Python uses to represent a sequence of values, similar to a list.

Syntax

- Importing the sqlite3 module:

```
import sqlite3
```
- Connecting to a SQLite database:

```
conn = sqlite3.connect("job.db")
```
- Creating an empty tuple:

```
t = ()
```
- Accessing the first value of a tuple:

```
apple = t[0]
```
- Returning a Cursor class:

```
cursor = conn.cursor()
```
- Executing a query:

```
cursor.execute("SELECT * FROM recent_grads;")
```

- Fetching the full results set as a list of tuples:

```
results = cursor.fetchall()
```

- Fetching one result and then the next result:

```
first_result = cursor.fetchone()  
second_result = cursor.fetchone()
```

- Fetching the first five results:

```
five_results = cursor.fetchmany(5)
```

- Closing a sqlite3 connection:

```
conn.close()
```

Resources

- [Connection instance](#)
- [SQLite version 3](#)

Joining Data in SQL

Concepts

- We use joins to combine multiple tables within a query.
- A schema diagram shows the tables in the database, the columns within them, and how they are connected.
- The **ON** statement tells the SQL engine what columns to use to join the tables.
- Joins come after the **FROM** clause.
- An inner join is the most common way to join data using SQL. An inner join includes only rows that have a match as specified by the **ON** clause.
- A left join includes all rows from an inner join, plus any rows from the first table that don't have a match in the second table.
- A right join includes all rows from the second table that don't have a match in the first table, plus any rows from an inner join.
- A full outer join includes all rows from both joined tables.
- SQLite doesn't support full outer joins or right joins.

Syntax

- Joining tables using an INNER JOIN:

```
SELECT [column_names] FROM [table_name_one]
    INNER JOIN [table_name_two] ON [join_constraint];
```

- Joining tables using a LEFT JOIN:

```
SELECT * FROM facts
    LEFT JOIN cities ON cities.facts_id = facts.id;
```

- Joining tables using a RIGHT JOIN:

```
SELECT f.name country, c.name city
    FROM cities c
    RIGHT JOIN facts f ON f.id = c.facts_id
    LIMIT 5;
```

- Joining tables using a FULL OUTER JOIN:

```
SELECT f.name country, c.name city
    FROM cities c
```

```

    FULL OUTER JOIN facts f ON f.id = c.facts_id
    LIMIT 5;

• Sorting a column without specifying a column name:

SELECT name, migration_rate FROM FACTS
ORDER BY 2 desc; # 2 refers to migration_rate column

• Using a join within a subquery:

SELECT c.name capital_city, f.name country
FROM facts f
INNER JOIN (
    SELECT * FROM cities
    WHERE capital = 1
) c ON c.facts_id = f.id
LIMIT 10;

```

Resources

- [SQL Joins](#)
- [Quora - Difference in Joins](#)

Intermediate Joins in SQL

Concepts

- A schema diagram helps us understand the available columns and the structure of the data.
- In a schema diagram, relationships are shown using lines between tables.
- Each row's primary key must be unique.
- A recursive join is joining a table to itself.
- The SQL engine will concatenate multiple columns and columns with a string. Also, the SQL engine also handles converting different types where needed.
- We can use the pipe operator (||) to concatenate columns.
- You can use the **LIKE** statement for partial matches:
 - **%Jen** : will match Jen at the end of a string, e.g., Sarah-Jen.
 - **Jen%** : will match Jen at the start of a string, e.g., Jenny.
 - **%Jen%** : will match Jen anywhere within the string, e.g., Kris Jenner.
- **LIKE** in SQLite is case insensitive but it may be case sensitive for other flavors of SQL.
 - You might need to use the **LOWER()** function in other flavors of SQL if is case sensitive.

Syntax

- Joining data from more than two tables:

```
SELECT [column_names] FROM [table_name_one]
    [join_type] JOIN [table_name_two] ON [join_constraint]
    [join_type] JOIN [table_name_three] ON [join_constraint]
    ...
    ...
    ...
    [join_type] JOIN [table_name_three] ON [join_constraint]
```

- Combining columns into a single column:

```
SELECT
    album_id,
    artist_id,
    "album id is " || album_id col_1,
```

```
"artist id is " || artist_id col2,  
album_id || artist_id col3  
FROM album LIMIT 3;
```

- Matching a part of a string:

```
SELECT  
    first_name,  
    last_name,  
    phone  
FROM customer  
WHERE first_name LIKE "%Jen%";
```

- Using if/then logic in SQL:

```
CASE  
    WHEN [comparison_1] THEN [value_1]  
    WHEN [comparison_2] THEN [value_2]  
    ELSE [value_3]  
END  
AS [new_column_name]
```

Resources

- [LOWER function](#)
- [Database Schema](#)

Building and Organizing Complex Queries

Concepts

- A few tips to help make your queries more readable:
 - If a select statement has more than one column: put each selected column on a new line, indented from the select statement.
 - Always capitalize SQL function names and keywords.
 - Put each clause of your query on a new line.
 - Use indenting to make subqueries appear logically separate.
- A **WITH** statement helps a lot when your main query has some slight complexities.
- A view is a permanently defined **WITH** statement that you can use in all future queries.
- Redefining a view requires having to delete or drop the existing view.
- Statements before and after **UNION** clause must have the same number of columns, as well as compatible data types.
- Comparison of **UNION**, **INTERSECT**, and **EXCEPT**:

Operator	What it Does	Python Equivalent
UNION	Selects rows that occur in either statement.	or
INTERSECT	Selects rows that occur in both statements.	and
EXCEPT	Selects rows that occur in the first statement, but don't occur in the second statement.	and not

Syntax

- Using the **WITH** clause:

```
WITH track_info AS
```

```
(
```

```

SELECT
    t.name,
    ar.name artist,
    al.title album_name,
FROM track t
INNER JOIN album al ON al.album_id = t.album_id
INNER JOIN artist ar ON ar.artist_id = al.artist_id
)
SELECT * FROM track_info
WHERE album_name = "Jagged Little Pill";

```

- Creating a view:

```

CREATE VIEW chinook.customer_2 AS
SELECT * FROM chinook.customer;

```

- Dropping a view

```
DROP VIEW chinook.customer_2;
```

- Selecting rows that occur in one or more SELECT statements:

```

[select_statement_one]
UNION
[select_statement_two];

```

- Selecting rows that occur in both SELECT statements:

```

SELECT * from customer_usa
INTERSECT
SELECT * from customer_gt_90_dollars;

```

- Selecting rows that occur in the first SELECT statement but not the second SELECT statement:

```

SELECT * from customer_usa
EXCEPT
SELECT * from customer_gt_90_dollars;

```

- Chaining WITH statements:

```

WITH
usa AS
(

```

```
SELECT * FROM customer
WHERE country = "USA"
),
last_name_g AS
(
SELECT * FROM usa
WHERE last_name LIKE "G%"
),
state_ca AS
(
SELECT * FROM last_name_g
WHERE state = "CA"
)
SELECT
first_name,
last_name,
country,
state
FROM state_ca
```

Resources

- [SQL Style Guide](#)
- [Set Operations](#)

Table Relations and Normalization

Concepts

- A semicolon is necessary to end your queries in the SQLite shell.
- SQLite comes with several dot commands to work with databases.
- You run dot commands are ran within SQLite shell.
- SQLite uses **TEXT**, **INTEGER**, **REAL**, **NUMERIC**, **BLOB** data types behind the scenes.
- A breakdown of SQLite data types and equivalent data types from other types of SQL.

Type	Commonly Used For	Equivalent Types
TEXT	Names Email Addresses Dates and Times Phone Numbers	CHARACTER VARCHAR NCHAR NVARCHAR DATETIME
INTEGER	IDs Quantities	INT SMALLINT BIGINT INT8
REAL	Weights Averages	DOUBLE FLOAT
NUMERIC	Prices Statuses	DECIMAL BOOLEAN
BLOB	Binary Data	BLOB

- A primary key is a unique identifier for each row
- A foreign key describes how the column is related to a foreign table
- Database normalization optimizes the design of databases, allowing for stronger data integrity. For example, it helps you avoid data duplication if a record being stored multiple times, and it helps avoid data modification if you need to update several rows after removing duplicate records.

- A compound primary key is when two or more columns combine to form a primary key.

Syntax

- Launching the SQLite shell:

```
sqlite3 chinook.db
```

- Switching column headers on:

```
.headers on
```

- Switching to column mode:

```
.mode column
```

- Displaying help text:

```
.help
```

- Displaying a list of all tables and views:

```
.tables
```

- Running BASH shell commands:

```
.shell [command]
```

- Viewing table schema:

```
.schema [table_name]
```

- Quitting the SQLite Shell:

```
.quit
```

- Creating a table:

```
CREATE TABLE [table_name] (
    [column1_name] [column1_type],
    [column2_name] [column2_type],
    [column3_name] [column3_type],
    [...]
);
```

- Creating a table with a primary and a foreign key:

```
CREATE TABLE purchase (
    purchase_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    purchase_date TEXT,
    total NUMERIC,
```

```
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

- Creating a compound primary key:

```
CREATE TABLE [table_name] (
    [column_one_name] [column_one_type],
    [column_two_name] [column_two_type],
    [column_three_name] [column_three_type],
    [column_four_name] [column_four_type],
    PRIMARY KEY (column_one_name, column_two_name)
);
```

- Inserting values into a table:

```
INSERT INTO [table_name] (
    [column1_name],
    [column2_name],
    [column3_name]
) VALUES (
    [value1],
    [value2],
    [value3]
);
OR
INSERT INTO [table_name] VALUES ([value1], [value2], [value3]);
```

- Deleting selected rows from a table:

```
DELETE FROM [table_name]
WHERE [expression];
```

- Adding a column:

```
ALTER TABLE [table_name]
ADD COLUMN [column_name] [column_type];
```

- Changing values for existing rows:

```
UPDATE [table_name]
SET [column_name] = [expression]
WHERE [expression]
```

Resources

- [SQLite Shell](#)
- [Database Normalization](#)

Using PostgreSQL

Concepts

- SQLite doesn't allow for restricting access to a database.
- PostgreSQL is the most commonly used database engine. It is powerful and open source (free to download and use).
- PostgreSQL allows you to create multiple databases.
- PostgreSQL consists of a server and clients.
 - A server is a program that manages databases and handles queries.
 - Clients communicate back and forth to the server. Multiple clients can communicate with the server at the same time.
- The most common Python client for PostgreSQL is called psycopg2`.
- PostgreSQL uses SQL transactions to prevent changes made in the database if any of the transactions fail.

Syntax

- Connecting to a PostgreSQL database called "postgres" with a user called "postgres":

```
import psycopg2  
  
conn = psycopg2.connect("dbname=postgres user=postgres")
```

- Initializing a Cursor object:

```
conn.cursor()
```

- Closing the database connection:

```
conn.close()
```

- Creating a table:

```
CREATE TABLE tableName(  
    column1 dataType1 PRIMARY KEY,  
    column2 dataType2,  
    column3 dataType3,  
    ...  
) ;
```

- Executing a query:

- ```
cur.execute("SELECT * FROM notes;")
```
- Applying the changes to the database:  
`conn.commit()`
  - Removing a SQL transaction:  
`conn.rollback()`
  - Activating autocommit:  
`conn.autocommit = True`
  - Fetching one result:  
`cur.fetchone()`
  - Fetching all rows in the table:  
`cur.fetchall()`
  - Inserting rows into a table:  
`INSERT INTO tableName  
VALUES (value1, value2, ...);`
  - Specifying an owner when creating a database:  
`CREATE DATABASE income OWNER dq;`
  - Removing a database:  
`DROP DATABASE income;`

## Resources

- [PostgreSQL](#)
- [Why Use PostgreSQL](#)

# Command line PostgreSQL

## Concepts

- The PostgreSQL command line tool is called **psql**.
- **psql** connects to a running PostgreSQL server process, which enables you to:
  - Run queries.
  - Manage users and permissions.
  - Manage databases.
  - See PostgreSQL system information.
- Queries in **psql** must end with a semicolon (;) or they won't be performed.
- When users are created, they don't have any ability, or permissions, to access tables in existing databases.
- You can grant or revoke multiple permissions by separating them with commas.
- You can grant or revoke users ability to use the **SELECT**, **INSERT**, **UPDATE**, or **DELETE** clauses on a table.
- A superuser can perform any function in a database.

## Syntax

- Starting the PostgreSQL command line tool:  
**psql**
- Exiting the PostgreSQL command line tool:  
**\q**
- Creating a database:  
**CREATE DATABASE dbName;**
- Listing databases:  
**\l**
- Listing all tables in the current database:  
**\dt**
- Listing the users that have access to the database:  
**\du**
- Connecting to a specified database:

- ```
psql -d dbName
```
- Creating a user:
`CREATE ROLE userName;`
 - Allowing a user to login to PostgreSQL and run queries:
`CREATE ROLE userName WITH LOGIN;`
 - Creating a password for a user:
`CREATE ROLE userName WITH LOGIN PASSWORD 'password';`
 - Allowing a user to create databases:
`CREATE ROLE userName WITH CREATEDB LOGIN PASSWORD 'password';`
 - Allowing a user to create other users:
`CREATE ROLE userName WITH CREATEROLE LOGIN PASSWORD 'password';`
 - Making the user a superuser:
`CREATE ROLE userName WITH LOGIN PASSWORD 'password' SUPERUSER;`
 - Granting a user permissions to access a table:
`GRANT SELECT ON tableName TO userName;`
 - Granting a user complete control of a table:
`GRANT ALL PRIVILEGES ON tableName to userName;`
 - Displaying what privileges have been granted to users:
`\dp tableName`
 - Removing permissions from a user:
`REVOKE SELECT ON tableName FROM userName;`
 - Removing all permissions from a user:
`REVOKE ALL PRIVILEGES on tableName FROM userName;`

Resources

- [psql documentation](#)
- [17 Practical psql commands](#)

Introduction to Indexing

Concepts

- Your query in SQLite is tokenized and parsed to look for any syntax errors before returning the results to you. If there are any syntax errors, the query execution halts and an error message is returned to you.
- You should minimize the amount of disk reads necessary when working with a database stored on disk.
- The query optimizer generates cost estimates for the various ways to access the underlying data, factoring in the schema of the tables and the operations the query requires. The optimizer quickly assesses the various ways to access the data and generate a best guess for the fastest query plan.
- SQLite still scans the entire table. A full table scan has time complexity $O(n)$ where n is the number of total rows in the table.
- Binary search of a table using the primary key would be $O(\log n)$ where n is the number of total rows in the table. Binary search on a primary key would be over a million times faster when working on a database with millions of rows compared to doing a full table scan.
- Either **SCAN** or **SEARCH** will always appear at the start of the query explanation for **SELECT** queries.
- An index table is optimized for lookups by the primary key.

Syntax

- Listing what SQLite is doing to return our results:
`EXPLAIN QUERY PLAN SELECT * FROM facts;`
- Creating an index:
`CREATE INDEX index_name ON table_name(column_name);`
- Creating an index if it does not exist:
`CREATE INDEX IF NOT EXISTS area_idx ON facts(area);`

Resources

- [What is an index?](#)
- [Query Plan](#)
- [Time Complexity](#)

Multi-column indexing

Concepts

- When there are two possible indexes available, SQLite tries to estimate which index will result in better performance. However, SQLite is not good estimating and will often end up picking an index at random.
- Use a multi-column index when data satisfying multiple conditions, in multiple columns, is to be retrieved.
- When creating a multi-column index, the first column in the parentheses becomes the primary key for the index.
- A covering index contains all the information necessary to answer a query.
- Covering indexes don't apply just to multi-column indexes.

Syntax

- Creating a multi-column index:

```
CREATE INDEX index_name ON table_name(column_name_1, column_name_2);
```

Resources

- [Multi-Column Indexes](#)
- [SQLite Index](#)

Working with APIs

Concepts

- An application program interface (API) is a set of methods and tools that allow different applications to interact with each other. APIs are hosted on web servers.
- Programmers use APIs to retrieve data as it becomes available, which allows the client to quickly and effectively retrieve data the changes frequently.
- JavaScript Object Notation (JSON) format is the primary format for sending and receiving data through APIs. JSON encodes data structures like lists and dictionaries as strings to ensure that machines can read them easily.
- The JSON library has two main methods:
 - **dumps** - Takes in a Python object, and converts it to a string.
 - **loads** - Takes in a JSON string, and converts it to a Python object.
- We use the **requests** library to communicate with the web server and retrieve the data.
- An endpoint is a server route for retrieving specific data from an API.
- Web servers return status codes every time they receive an API request.
- Status codes that are relevant to GET requests:
 - **200** - Everything went okay, and the server returned a result.
 - **301** - The server is redirecting you to a different endpoint. This can happen with a company switches domain names or an endpoint's name has changed.
 - **401** - The server thinks you're not authenticated. This happens when you don't supply the right credentials.
 - **400** - The server thinks you made a bad request. This can happen when you don't send the information the API requires to process your request.
 - **403** - The resource you're trying to access is forbidden; you don't have the right permissions to see it.
 - **404** - The server didn't find the resource you tried to access.

Syntax

- Accessing the content of the data the server returns:
`response.content`
- Importing the JSON library:

```
import json
```

- Getting the content of a response as a Python object:
`response.json()`
- Accessing the information on how the server generated the data, and how to decode the data:
`response.headers`

Resources

- [Requests library Documentation](#)
- [JSON Library Documentation](#)

Intermediate APIs

Concepts

- APIs use authentication to perform rate limiting. Rate limiting ensures the user can't overload the API server by making too many requests too fast, which allows the API server to be available and responsive for all users.
- Rate limiting ensures the user can't overload the API server by making too many requests too fast.
- APIs requiring authentication use an access token. An access token is a string the API can read and associate with your account. Tokens are preferable to a username and password for the following security reasons:
 - Typically, someone accesses an API from a script. If you put your username and password in a script and someone manages to get their hands on it, they can take over your account. If someone manages to get their hands on the access token, you can revoke the access token.
 - Access tokens can also have scopes in specific permissions. You can generate multiple tokens that give different permissions to give you more control over security.
- Pagination allows for a certain number of records per page.
- Different API endpoints choose what types of requests they will accept.
 - We use POST requests to send information and to create objects on the API's server. POST requests almost always includes data so the server can create the new object. A successful POST request will return a 201 status code.
 - A successful PATCH request will return a 200 status code.
 - A PUT request will send the object we're revising as a replacement for the server's existing version.
 - A DELETE request removes objects from the server. A successful DELETE request will return a 204 status code.

Syntax

- Passing in a token to authorize API access:

```
{"Authorization": "token 1f36137fbbe1602f779300dad26e4c1b7fbab631"}
```

- Defining pagination query parameters:

```
{"per_page": 50, "page": 1}
```

- Using a POST request to create a repository:

```
payload = {"name": "test"}
```

```
requests.post("https://api.github.com/user/repos", json=payload)
```

- Using a PATCH request to change a repository:

```
payload = {"description": "The best repository ever!", "name": "test"}  
response = requests.patch("https://api.github.com/repos/VikParuchuri/test", json=payload)
```

- Using a DELETE request to delete a repository:

```
response = requests.delete("https://api.github.com/repos/VikParuchuri/learning-about-apis")
```

Resources

- [Github API documentation](#)
- [Understanding REST](#)

Web Scraping

Concepts

- A lot of data is not accessible through data sets or APIs; they exist on the Internet as Web pages. We can use a technique called web scraping to access the data without waiting for the provider to create an API.
- We can use the **requests** library to download a web page, and **Beautifulsoup** to extract the relevant parts of the web page.
- Web pages use HyperText Markup Language (HTML) as the foundation for the content on the page, and browsers such as Google Chrome and Mozilla Firefox reads the HTML to determine how to render and display the page.
- The **head** tag in HTML contains information that's useful to the Web browser that's rendering the page. The **body** section contains the bulk of the content the user interacts with on the page. The **title** tag tells the Web browser what page title to display in the toolbar.
- HTML allows elements to have IDs so we can use them to refer to specific elements since IDs are unique.
- **Cascading Style Sheets**, or **CSS**, is a language for adding styles to HTML pages.
- We can also use CSS selectors to select elements when we do web scraping.

Syntax

- Importing BeautifulSoup:
`from bs4 import BeautifulSoup`
- Initializing the HTML parser:
`parser = BeautifulSoup(content, 'html.parser')`
- Getting the inside text of a tag:
`title_text = title.text`
- Returning a list of all occurrences of a tag:
`head.find_all("title")`
- Getting the first instance of a tag:
`title=head[0].find_all("title")`
- Creating an example page using HTML:
`<html>`
`<head>`

```
<title>
</head>
<body>
    <p>Here is some simple content for this page.<p>
</body>
</html>
```

- Using CSS to make all of the text inside all paragraphs red:

```
p{
    color: red
}
```

- Using CSS selectors to style all elements with the class "inner-text" red:

```
.inner-text{
    color: red
}
```

- Working with CSS selectors:

```
parser.select(".first-item")
```

Resources

- [HTML basics](#)
- [HTML element](#)
- [BeautifulSoup Documentation](#)

Probability and Statistics

Sampling

Concepts

- The set of *all* individuals relevant to a particular statistical question is called a **population**. A smaller group selected from a population is called a **sample**. When we select a smaller group from a population, we do **sampling**.
- A **parameter** is a metric specific to a population, and a **statistic** is a metric specific to a sample. The difference between a statistic and its corresponding parameter is called **sampling error**. If the sampling error is low, then the sample is **representative**.
- To make our samples representative we can try different sampling methods:
 - **Simple random sampling**.
 - **Stratified sampling**.
 - **Cluster sampling**.
- When we describe a sample or a population, we do **descriptive statistics**. When we try to use a sample to draw conclusions about a population, we do **inferential statistics** (we *infer* information from the sample about the population).

Syntax

- Sampling randomly a **Series** object:

```
### Sampling 10 sample points ###  
sample_10 = Series.sample(10)
```

```
### Sampling 500 sample points ###  
sample_500 = Series.sample(500)
```

- Making the generation of random numbers predictable using the **random_state** parameter:

```
### Sampling 10 sample points in a reproducible way ###  
sample_10 = Series.sample(10, random_state = 1)
```

```
### Using a different value for `random_state`
```

```
### sample_10_different = Series.sample(10, random_state = 2)
```

Resources

- [The Wikipedia entry](#) on sampling.
- [The Wikipedia entry](#) on samples.
- [The Wikipedia entry](#) on populations.

Variables in Statistics

Concepts

VARIABLES

- A property whose values can vary from individual to individual is called a **variable**. For instance, *height* is a property whose value can vary from individual to individual — hence *height* is a variable.
- Variables can be divided into two categories:
 - **Quantitative** variables, which describe a *quantity*. Examples include: height, age, temperature, time, etc.
 - **Qualitative** variables, which describe a *quality*. Examples include: name, team, t-shirt number, color, zip code, etc.

SCALES OF MEASUREMENT

- The system of rules that define how a variable is measured is called **scale of measurement**. We learned about four scales of measurement: nominal, ordinal, interval, and ratio.
- The **nominal** scale is specific to qualitative variables. If a variable is measured on a nominal scale:
 - We can tell whether two individuals are different or not.
 - We can't tell the direction of the difference.
 - We can't tell the size of the difference.
- The **ordinal** scale is specific to quantitative variables. If a variable is measured on an ordinal scale:
 - We can tell whether two individuals are different or not.
 - We can tell the direction of the difference.
 - We can't tell the size of the difference.
- **Interval** and **ratio** scales are both specific to quantitative variables. If a variable is measured on an interval or ratio scale:
 - We can tell whether two individuals are different or not.
 - We can tell the direction of the difference.
 - We can tell the size of the difference.
- For an interval scale, the **zero point** doesn't mean the absence of a quantity — this makes it impossible to measure the difference between individuals in terms of **ratios**. In contrast, for a

ratio scale, the zero point means the absence of a quantity, which makes it possible to measure the difference between individuals in terms of ratios.

- Variables measured on an interval and ratio scales can be divided further into:
 - **Discrete** variables — there's no possible intermediate value between any two adjacent values of a discrete variable.
 - **Continuous** variables — there's an infinity of values between any two values of a continuous variable.

Resources

- [The Wikipedia entry](#) on the four scales of measurement we learned about.
- [A brief intuitive introduction](#) to discrete and continuous variables.

Frequency Distributions

Concepts

- A table that shows the frequency for each unique value in a distribution is called a **frequency distribution table**.
- The frequencies can be expressed as:
 - Absolute counts (**absolute frequencies**).
 - Proportions or percentages (**relative frequencies**).
- The percentage of values that are equal or less than a value x is called the **percentile rank** of x . For instance, if the percentile rank of a value of 32 is 57%, 57% of the values are equal to or less than 32.
- If a value x has a percentile rank of $p\%$, we say that x is the p th **percentile**. For instance, if 32 has a percentile rank of 57%, we say that 32 is the 57th percentile.
- Frequency distribution tables can be grouped in **class intervals** to form **grouped frequency distribution tables**. As a rule of thumb, 10 is a good number of class intervals to choose because it offers a good balance between information and comprehensibility.

Syntax

- Generating a frequency distribution table for a **Series**:

```
frequency_table = Series.value_counts()
```
- Sorting the values of frequency distribution table:

```
### In an ascending order (default) ###
freq_table_asc = Series.value_counts().sort_index()

### In a descending order ###
freq_table_desc = Series.value_counts().sort_index(ascending = False)
```
- Finding proportions and percentages in a frequency distribution table:

```
### Proportions ###
proportions = Series.value_counts(normalize = True)

### Percentages ###
percentages = Series.value_counts(normalize = True) * 100
```
- Finding the percentile rank of a value (score) in some array:

```
from scipy.stats import percentileofscore
```

- ```
percentile_rank = percentileofscore(a = some_array, score = some_score,
kind = 'weak')
```
- Finding percentiles:

```
Only the quartiles
quartiles = Series.describe()

Any percentile we want
percentiles = Series.describe(percentiles = [.1, .15, .33, .5, .592,
.9])
```
  - Generating a grouped frequency table:

```
With 5 class intervals
gr_freq_table_5 = Series.value_counts(bins = 5)

With 10 class intervals
gr_freq_table_10 = Series.value_counts(bins = 10)
```

## Resources

- [An intuitive introduction](#) to frequency distribution tables.
- [An intuitive introduction](#) to grouped frequency distribution tables.
- [The Wikipedia entry](#) on frequency distributions.

# Visualizing Frequency Distributions

## Concepts

- To visualize frequency distributions for *nominal* and *ordinal* variables, we can use:
  - **Bar plots.**
  - **Pie charts.**
- To visualize frequency distributions for variables measured on an interval or ratio scale, we can use a **histogram**.
- Depending on the shape of the histogram, we can have:
  - **Skewed** distributions:
    - Left skewed (negatively skewed) — the tail of the histogram points to the left.
    - Right skewed (positively skewed) — the tail of the histogram points to the right.
  - **Symmetrical** distributions:
    - **Normal** distributions — the values pile up in the middle and gradually decrease in frequency toward both ends of the histogram.
    - **Uniform** distributions — the values are distributed uniformly across the entire range of the distribution.

## Syntax

- Generating a bar plot for a frequency distribution table:

```
Vertical bar plot ###
Series.value_counts().plot.bar()

Horizontal bar plot ###
Series.value_counts().plot.barch()
```
- Generating a pie chart for a frequency distribution table:

```
Using the defaults ###
Series.value_counts().plot.pie()

Making the pie chart a circle and adding percentages labels ###
import matplotlib.pyplot as plt

Series.value_counts().plot.pie(figsize = (6,6), autopct = '%.1f%%')
plt.ylabel('') # removes the label of the y-axis
```

- Generating a histogram for a **Series**:

```
Series.plot.hist()
```

## Resources

- [An introduction](#) to bar plots.
- [An introduction](#) to pie charts.
- [An introduction](#) to histograms.
- [An introduction](#) to skewed distributions.
- [More details](#) on the normal distribution.

# Comparing Frequency Distributions

## Concepts

- To compare visually frequency distributions for nominal and ordinal variables we can use **grouped bar plots**.
- To compare visually frequency distributions for variables measured on an interval or ratio scale, we can use:
  - **Step-type histograms**.
  - **Kernel density plots**.
  - **Strip plots**.
  - **Box plots**.
- A value that is much lower or much larger than the rest of the values in a distribution is called an **outlier**. A value is an outlier if:
  - It's larger than the upper quartile by 1.5 times the interquartile range.
  - It's lower than the lower quartile by 1.5 times the interquartile range.

## Syntax

- Generating a grouped bar plot:

```
import seaborn as sns

sns.countplot(x = 'column_name_1', hue = 'column_name_2', data =
some_dataframe)
```

- Generating only the shape of the histogram for two **Series** objects:

```
Series_1.plot.hist(histtype = 'step')
Series_2.plot.hist(histtype = 'step')
```

- Generating kernel density plots for two **Series** objects:

```
Series_1.plot.kde()
Series_2.plot.kde()
```

- Generating strip plots:

```
import seaborn as sns
sns.stripplot(x = 'column_name_1', y = 'column_name_2', data =
some_dataframe)
```

- Generating multiple box plots:

```
import seaborn as sns
sns.boxplot(x = 'column_name_1', y = 'column_name_2', data =
some_dataframe)
```

## Resources

- [A seaborn tutorial](#) on grouped bar plots, strip plots, box plots, and more.
- [A seaborn tutorial](#) on kernel density plots, histograms, and more.

# The Mean

## Concepts

- We can summarize the distribution of a numerical variable by computing its **mean**.
- The mean is a single value and is the result of taking into account **equally** each value in the distribution.
- The mean is **the balance point** of a distribution — the total distance of the values below the mean is equal to the total distance of the values above the mean.
- The mean  $\mu$  of a population can be defined algebraically in several equivalent ways:

$$\mu = \frac{x_1 + x_2 + \dots + x_N}{N} = \frac{\sum_{i=1}^N x_i}{N} = \frac{1}{N} \left( \sum_{i=1}^N x_i \right)$$

- The mean  $\bar{x}$  of a sample can be defined algebraically in several equivalent ways:

$$\bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n} = \frac{\sum_{i=1}^n x_i}{n} = \frac{1}{n} \left( \sum_{i=1}^n x_i \right)$$

- The sample mean  $\bar{x}$  is an unbiased estimator for the population mean  $\mu$ .

## Syntax

- Computing the mean of any numerical array:

```
Pure Python ###
mean = sum(array) / len(array)

Using numpy ###
from numpy import mean
mean_numpy = mean(array)
```

- Computing the mean of a **Series**:

```
mean = Series.mean()
```

## ***Resources***

- [The Wikipedia entry](#) on the mean.
- Useful documentation:
  - [numpy.mean\(\)](#)
  - [Series.mean\(\)](#)

# The Weighted Mean and the Median

## Concepts

- When data points bear different weights, we need to compute **the weighted mean**. The formulas for the weighted mean are the same for both samples and populations, with slight differences in notation:

$$\bar{x} = \frac{\sum_{i=1}^n x_i w_i}{\sum_{i=1}^n w_i} = \frac{x_1 w_1 + x_2 w_2 + \dots + x_n w_n}{w_1 + w_2 + \dots + w_n}$$

$$\mu = \frac{\sum_{i=1}^N x_i w_i}{\sum_{i=1}^N w_i} = \frac{x_1 w_1 + x_2 w_2 + \dots + x_N w_N}{w_1 + w_2 + \dots + w_N}$$

- It's difficult to define the median algebraically. To compute the median of an array, we need to:
  - Sort the values in an ascending order.
  - Select the middle value as the median. If the distribution is even-numbered, we select the middle two values, and then compute their mean — the result is the median.
- The median is ideal for:
  - Summarizing numerical distributions that have **outliers**.
  - Open-ended** distributions.
  - Ordinal data**.

## Syntax

- Computing the weighted mean for a distribution `distribution_x` with weights `weights_x`:

```
Using numpy
```

```
from numpy import average
```

```

weighted_mean_numpy = average(distribution_X, weights = weights_X)

By coding a function from scratch

def weighted_mean(distribution, weights):
 weighted_sum = []
 for mean, weight in zip(distribution, weights):
 weighted_sum.append(mean * weight)
 return sum(weighted_sum) / sum(weights)

weighted_mean_function = weighted_mean(distribution_X, weights_X)

```

- Finding the median for a `Series`:

```
median = Series.median()
```

- Finding the median for any numerical array:

```
from numpy import median
```

```
median_numpy = median(array)
```

## Resources

- [An intuitive introduction](#) to the weighted mean.
- [The Wikipedia entry](#) on the weighted mean.
- [The Wikipedia entry](#) on the median.
- Useful documentation:
  - [numpy.average\(\)](#)
  - [Series.median\(\)](#)
  - [numpy.median\(\)](#)

# The Mode

## Concepts

- The **most frequent** value in the distribution is called **the mode**.
- A distribution can have:
  - One mode (**unimodal** distribution).
  - Two modes (**bimodal** distribution).
  - More than two modes (**multimodal** distribution).
  - **No mode** (as for a perfectly uniform distribution or the distribution of a continuous variable).
- The mode is an ideal summary metric for:
  - **Nominal** data.
  - **Ordinal** data (especially when the values are represented using words).
  - **Discrete** data (when we need to communicate the average value to a non-technical audience).
- The location of the mean, median, and mode is usually predictable for certain kinds distributions:
  - **Left-skewed** distributions: the mode is on the far right, the median is to the left of the mode, and the mean is to the left of the median.
  - **Right-skewed** distributions: the mode is on the far left, the median is to the right of the mode, and the mean is to the right of the median.
  - **Normal** distributions: the mean, the median, and the mode are all in the center of the distribution.
  - **Uniform** distributions: the mean and the median are at the center, and there's no mode.
  - Any **symmetrical** distribution: the mean and the median are at the center, while the position of the mode may vary, and there can also be symmetrical distributions having more than one mode (see example in the mission).

## Syntax

- Computing the mode of **Series**:  
`mode = Series.mode()`
- Coding from scratch a function that computes the mode of an array:

```
def mode(array):
 counts = {}
 for value in array:
 if value in counts:
 counts[value] += 1
 else:
 counts[value] = 1
 return max(counts, key = counts.get)
```

## Resources

- [The Wikipedia entry](#) on the mode.
- [Paul von Hippel's paper](#) addressing patterns in skewed distributions.

# Measures of Variability

## Concepts

- There are many ways we can measure the **variability** of a distribution. These are some of the measures we can use:
  - **The range.**
  - **The mean absolute deviation.**
  - **The variance.**
  - **The standard deviation.**
- Variance and standard deviation are the most used metrics to measure variability. To compute the standard deviation  $\sigma$  and the variance  $\sigma^2$  for a **population**, we can use the formulas:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (x_i - \mu)^2}{N}}$$

$$\sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N}$$

- To compute the standard deviation  $s$  and the variance  $s^2$  for a **sample**, we need to add the **Bessel's correction** to the formulas above:

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n - 1}}$$

$$s^2 = \frac{\sum_{i=1}^n (x_i - \mu)^2}{n - 1}$$

- **Sample variance**  $s^2$  is the only unbiased estimator we learned about, and it's unbiased only when we sample with replacement.

## Syntax

- Writing a function that returns the range of an array:

```
def find_range(array):
 return max(array) - min(array)
```

- Writing a function that returns the mean absolute deviation of an array:

```
def mean_absolute_deviation(array):
 reference_point = sum(array) / len(array)
 distances = []
 for value in array:
 absolute_distance = abs(value - reference_point)
 distances.append(absolute_distance)
 return sum(distances) / len(distances)
```

- Finding the variance of an array:

```
If the array is a `Series` object

sample_variance = Series.var(ddof = 1)
population_variance = Series.var(ddof = 0)

If the array is not a `Series` object

from numpy import var

sample_variance = var(a_sample, ddof = 1)
```

```
population_variance = var(a_population, ddof = 0)
```

- Finding the standard deviation of an array:

```
If the array is a `Series` object ###
sample_stdev = Series.std(ddof = 1)
population_stdev = Series.std(ddof = 0)
If the array is not a `Series` object ###
from numpy import std
sample_stdev = std(a_sample, ddof = 1)
population_stdev = std(a_population, ddof = 0)
```

## Resources

- [An intuitive introduction to variance and standard deviation.](#)
- Useful documentation:
  - [numpy.var\(\)](#)
  - [numpy.std\(\)](#)
  - [Series.var\(\)](#)
  - [Series.std\(\)](#)

# Z-scores

## Syntax

- Writing a function that converts a value to a z-score:

```
def z_score(value, array, bessel = 0):
 mean = sum(array) / len(array)
 from numpy import std
 st_dev = std(array, ddof = bessel)
 distance = value - mean
 z = distance / st_dev
 return z
```

- Standardizing a **Series**:

```
standardized_distro = Series.apply(lambda x: (x - Series.mean()) /
Series.std())
```

- Transforming a standardized distribution to a different distribution, with a predefined mean and standard deviation:

```
mean = some_mean
st_dev = some_standard_deviation
standardized_distro = Series.apply(lambda z: z * st_dev + mean)
```

## Concepts

- A **z-score** is a number that describes the location of a value within a distribution. Non-zero z-scores (+1, -1.5, +2, -2, etc.) consist of two parts:
  - *A sign*, which indicates whether the value is above or below the mean.
  - *A value*, which indicates the number of standard deviations that a value is away from the mean.
- The z-score of the mean is 0.
- To compute the z-score  $z$  for a value  $x$  coming from a population with mean  $\mu$  and standard deviation  $\sigma$ , we can use this formula:

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

- To compute the z-score  $z$  for a value  $x$  coming from a sample with mean  $\bar{x}$  and standard deviation  $s$ , we can use this formula:
- We can **standardize** any distribution by transforming all its values to z-scores. The resulting distribution will have a mean of 0 and a standard deviation of 1. Standardized distributions are often called **standard distributions**.

$$z = \frac{x - \bar{x}}{s}$$

- Standardization is useful for **comparing values** coming from distributions with different means and standard deviations.
- We can transform any population of z-scores with mean  $\mu_z=0$  and  $\sigma_z=1$  to a distribution with any mean  $\mu$  and any standard deviation  $\sigma$  by converting each z-score  $z$  to a value  $x$  using this formula:

$$x = z\sigma + \mu$$

- We can transform any sample of z-scores with mean  $\bar{x}_z=0$  and  $s_z=1$  to a distribution with any mean  $\bar{x}$  and any standard deviation  $s$  by converting each z-score  $z$  to a value  $x$  using this formula:

$$x = zs + \bar{x}$$

## Resources

- [The `z-score\(\)` function from `scipy.stats.mstats`](#) — useful for standardizing distributions.
- [The Wikipedia entry on z-scores.](#)

# Introduction to probability

## Concepts

- Probability is the percentage chance of an event or sequence of events occurring.
- Conjunctive probability is the probability involves a sequence of events.
- Disjunctive probability is the probability involves mutually exclusive events.
- Independent events are not affected by the previous event. Dependent events are affected by the previous event.

## Resources

- [Probability](#)
- [Basic Probability](#)

# Calculating probabilities

## Concepts

- The probability of three heads when flipping three coins is  $0.5 * 0.5 * 0.5$ , which equals **0.125**.
- Probability follows a pattern. A given outcome happening all the time or none of the time, can only occur in one combination. The next step lower, a given outcome happening every time except once, or a given outcome only happening once, can happen in as many combinations as there are total events.
- A factorial means "multiply every number from 1 to this number together" so  $4! = 4 * 3 * 2 * 1 = 24$ .
- We can calculate the number of combinations in which an outcome can occur in a set of events using:  
$$\frac{N!}{k!(N-k)!}$$
  - **k** is the number of times we want the desired outcome to occur.
  - **N** is the total number of events we have.
- The probability of a single combination occurring is given by  $p^k q^{N-k}$  where:
  - **p** is the probability of an outcome will occur.
  - **q** is the complimentary probability the outcome will not happen.
  - **k** is the number of times we want the desired outcome to occur.
  - **N** is the total number of events we have.
- Statistical significance is the question of whether a result happened as the result of something we changed, or whether a result is a matter of random chance. Typically, researchers will use **5%** as a significance threshold to determine if an event is statistically significant or not.

## Syntax

- Finding probability of an event:

```
days_over_threshold = bikes[bikes["cnt"] > 4000].shape[0] # number of days that satisfies condition

total_days = bikes.shape[0] # total number of days

probability_over_4000 = days_over_threshold / total_days # proportion of condition

satisfied:total number of days
```

- Accessing the factorial method using the math module:

```
import math
math.factorial(5)
```

## ***Resources***

- [Binomial Distribution](#)
- [Factorial](#)
- [Statistical Significance](#)

# Probability distributions

## Concepts

- Binomial probabilities are the chance of a certain outcome happening in a sequence.
- One way to visualize binomials is a binomial distribution. Given N events, it plots the probabilities of getting different numbers of successful outcomes. The binomial distribution parameters are:
  - **N**: The total number of events.
  - **p**: The probability of the outcome we're interested in seeing.
- Formula for binomial probability:

$$(p^k * q^{N-k}) * \frac{N!}{k!(N-k)!}$$

- The probability mass function (pmf) gives us the probability of each k occurring, and takes in the following parameters:
  - **x**: The list of outcomes.
  - **n**: The total number of events.
  - **p**: The probability of the outcome we're interested in seeing.
- A probability distribution can only tell us which values are likely, and how likely they are.
- We can calculate the expected probability of a probability distribution using  $N*p$ , where N is the total number of events, and p is the probability of the outcome we're interested in seeing.
- The formula for standard deviation, or a measure of how much the values vary from the mean, of a probability distribution is  $\sqrt{N*p*q}$  where N is the total number of events, p is the probability of the outcome we're interested in seeing, and q is the probability of the outcome not happening.
- The cumulative density function is the probability that k or less events will occur.
- The z-score is the number of standard deviations away from the mean and used to find the percentage of values to the left or right.
- We can calculate the mean ( $\mu$ ) and standard deviation ( $\sigma$ ) using the following formulas:
  - $\mu = N * p$
  - $\sigma = \sqrt{N * p * q}$
- We can figure out the z-score of a value using the following formula:

- $$\text{z-score} = \frac{k-\mu}{\sigma}$$

## Syntax

- Using the probability mass function from SciPy:

```
from scipy import linspace
from scipy.stats import binom
outcome_counts = linspace(0,30,31)
dist = binom.pmf(outcome_counts,30,0.39)
```

- Using the probability mass function from SciPy:

```
from scipy import linspace
from scipy.stats import binom
outcome_counts = linspace(0,30,31)
dist = binom.cdf(outcome_counts,30,0.39)
```

- Getting the sum of all the probabilities to the left of k, including k:

```
left = binom.cdf(k,N,p)
```

- Getting the sum of all the probabilities to the right of k:

```
right = 1 - left
```

## Resources

- [Probability Mass Function](#)
- [SciPy Documentation](#)
- [Documentation for scipy.stats.binom](#)
- [Cumulative Density Function](#)

# Significance Testing

## Concepts

- A hypothesis is a pattern or rule that can be tested. We use hypothesis testing to determine if a change we made had a meaningful impact.
  - A null hypothesis describes what is happening currently.
  - While we use an alternative hypothesis to compare with the null hypothesis to decide which describes the data better.
- In a blind experiment, none of the participants knows which group they're in. Blind experiments help reduce potential bias.
- If there is a meaningful difference in the results, we say the result is statistically significant.
- A test statistic is a numerical value that summarizes the data; we can use it in statistical formulas.
- The permutation test is a statistical test that involves simulating rerunning the study many times and recalculating the test statistic for each iteration.
- A sampling distribution approximates the full range of possible test statistics under the null hypothesis.
- A p-value can be considered to be a measurement of how unusual an observed event is. The lower the p-value, the more unusual the event is.
- Whether we reject or fail to reject the null hypothesis or alternative hypothesis depends on the p-value threshold, which should be set before conducting the experiment.
- The most common p-value threshold is **0.05** or **5%**. The p-value threshold can affect the conclusion you reach.

## Syntax

- Visualizing a sampling distribution:

```
mean_differences = []
for i in range(1000):
 group_a = []
 group_b = []
 for value in all_values:
 assignment_chance = np.random.rand()
 if assignment_chance >= 0.5:
```

```
group_a.append(value)

else:

 group_b.append(value)

iteration_mean_difference = np.mean(group_b) -
np.mean(group_a)

mean_differences.append(iteration_mean_difference)

plt.hist(mean_differences)

plt.show()
```

## ***Resources***

- [What P-Value Tells You](#)
- [Difference Between A/B Testing and Hypothesis Testing](#)
- [Type I and Type II Errors](#)

# Chi-squared tests

## Concepts

- The chi-squared test enables us to quantify the difference between sets of observed and expected categorical values to determine statistical significance.
- To calculate the chi-squared test statistic, we use the following formula:  $\text{observed} - \text{expected}^2 / \text{expected}$ .
- A p-value allows us to determine whether the difference between two values is due to chance, or due to an underlying difference.
- Chi-squared values increase as sample size increases, but the chance of getting a high chi-squared value decreases as the sample gets larger.
- A degree of freedom is the number of values that can vary without the other values being "locked in."

## Syntax

- Calculating the chi-squared test statistic and creating a histogram of all the chi-squared values:

```
chi_squared_values = []

from numpy.random import random

import matplotlib.pyplot as plt

for i in range(1000):

 sequence = random((32561,))

 sequence[sequence < .5] = 0

 sequence[sequence >= .5] = 1

 male_count = len(sequence[sequence == 0])

 female_count = len(sequence[sequence == 1])

 male_diff = (male_count - 16280.5) ** 2 / 16280.5

 female_diff = (female_count - 16280.5) ** 2 / 16280.5

 chi_squared = male_diff + female_diff

 chi_squared_values.append(chi_squared)

plt.hist(chi_squared_values)
```

- Calculating a chi-squared sampling distribution with four degrees of freedom:

```
import numpy as np
```

```
from scipy.stats import chisquare
observed = np.array([5, 10, 15])
expected = np.array([7, 11, 12])
chisquare_value, pvalue = chisquare(observed, expected) # returns a
list
```

## Resources

- [Chi-Square Test](#)
- [Degrees of Freedom](#)
- [Scipy Chi-Square documentation](#)

# Multi category chi-squared tests

## Concepts

- In a multiple category chi-squared test, we calculate expected values across our whole dataset.
- We can calculate the chi-squared value by using the following steps:
  1. Subtract the expected value from the observed value.
  2. Subtract the difference.
  3. Divide the squared difference by the expected value.
  4. Repeat for all observed and expected values and add up all the values.
- Formula for chi-squared:

$$\sum \frac{(\text{observed} - \text{expected}^2)}{\text{expected}}$$

- Finding that a result isn't significant doesn't mean that no association between the columns exists. Finding a statistically significant result doesn't imply anything about what the correlation is.
- Chi-squared tests can only be applied in the case where each possibility within a category is independent.

## Syntax

- Calculating the chi-squared value:

```
observed = [6662, 1179, 15128, 9592]
expected = [5249.8, 2597.4, 16533.5, 8180.3]
values = []
for i, obs in enumerate(observed):
 exp = expected[i]
 value = (obs - exp) ** 2 / exp
 values.append(value)
chisq_gender_income = sum(values)
```

- Finding the chi-squared value and p-value using `scipy.stats.chisquare`:

```
import numpy as np
from scipy.stats import chisquare
```

```
observed = np.array([6662, 1179, 15128, 9592])
expected = np.array([5249.8, 2597.4, 16533.5, 8180.3])
chisq_value, pvalue_gender_income = chisquare(observed, expected)
```

- Using the pandas.crosstab function to print a table that shows frequency counts:

```
import pandas

table = pandas.crosstab(income["sex"], [income["high_income"]])

print(table)
```

- Using the scipy.stats.chi2\_contingency function to generate the expected values:

```
import numpy as np

from scipy.stats import chi2_contingency

observed = np.array([[5, 5], [10, 10]])

chisq_value, pvalue, df, expected = chi2_contingency(observed)
```

## Resources

- [Chi-squared test of association](#)
- [Documentation for scipy.stats.chi2\\_contingency function](#)
- [Documentation for pandas.crosstab function](#)

# Machine Learning

## Introduction to K-Nearest Neighbors

### Concepts

- Machine learning is the process of discovering patterns in existing data to make a prediction.
- In machine learning, a feature is an individual measurable characteristic.
- When predicting a continuous value, the main similarity metric that's used is Euclidean distance.
- K-nearest neighbors computes the Euclidean Distance to find similarity and average to predict an unseen value.
- Let  $q_1$  to  $q_n$  represent the feature values for one observation, and  $p_1$  to  $p_n$  represent the feature values for the other observation then the formula for Euclidean distance is as follows:

$$d = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2}$$

- In the case of one feature (univariate case), the Euclidean distance formula is as follows:

$$d = \sqrt{(q_1 - p_1)^2}$$

### Syntax

- Randomizing the order of a DataFrame:

```
import numpy as np
np.random.seed(1)
np.random.permutation(len(dc_listings))
```

- Returning a new DataFrame containing the shuffled order:

```
dc_listings = dc_listings.loc[np.random.permutation(len(dc_listings))]
```

- Applying string methods to replace a comma with an empty character:

```
stripped_commas = dc_listings['price'].str.replace(',', '')
```

- Converting a Series object to a float datatype:

```
dc_listings['price'] = dc_listings['price'].astype('float')
```

### Resources

- [K-Nearest Neighbors](#)

- [Five Popular Similarity Measures](#)

# Evaluating Model Performance

## Concepts

- A machine learning model outputs a prediction based on the input to the model.
- When you're beginning to implement a machine learning model, you'll want to have some kind of validation to ensure your machine learning model can make accurate predictions on new data.
- You can test the quality of your model by:
  - Splitting the data into two sets:
    - The training set, which contains a majority of the rows (75%).
    - The test set, which contains the remaining rows (25%).
  - Using the rows in the training set to predict the values for the rows in the test set.
  - Comparing the actual values with the predicted values to see how accurate the model is.
- To quantify how good the predictions are for the test set, you would use an error metric. The error metric quantifies the difference between each predicted and actual value and then averaging those differences.
  - This is known as the mean error but isn't effective in most cases because positive and negative differences are treated differently.
- The MAE computes the absolute value of each error before we average all the errors.
- Let n be the number of observations then the MAE equation is as follows:

$$MAE = \frac{1}{n} \sum_{k=1}^n |(actual_k - predicted_k)| + \dots + |(actual_n - predicted_n)|$$

- The MSE makes the gap between the predicted and actual values clearer by squaring the difference of the two values.
- Let n be the number of observations then the MSE equation is as follows:

$$MSE = \frac{1}{n} \sum_{k=1}^n (actual_k - predicted_k)^2 + \dots + (actual_n - predicted_n)^2$$

- RMSE is an error metric whose units are the base unit, and is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n} \sum_{k=1}^n (actual_k - predicted_k)^2 + \dots + (actual_n - predicted_n)^2}$$

- In general, the MAE value is expected to be much less than the RMSE value due the sum of the squared differences before averaging.

## Syntax

- Calculating the (mean squared error) MSE:

```
test_df['squared_error'] = (test_df['predicted_price'] -
test_df['price'])**2

mse = test_df['squared_error'].mean()
```

- Calculating the (mean absolute error) MAE:

```
test_df['squared_error'] = np.absolute(test_df['predicted_price'] -
test_df['price'])

mae = test_df['squared_error'].mean()
```

- Calculating the root mean squared error (RMSE) :

```
test_df['squared_error'] = (test_df['predicted_price'] -
test_df['price'])**2

mse = test_df['squared_error'].mean()

rmse = mse ** (1/2)
```

## Resources

- [MAE and RMSE comparison](#)
- [About Train, Validation, and Test Sets in Machine Learning](#)

# Multivariate K-Nearest Neighbors

## Concepts

- To reduce the RMSE value during validation and improve accuracy, you can:
  - Select the relevant attributes a model uses. When selecting attributes, you want to make sure you're not working with a column that doesn't have continuous values. The process of selecting features to use in a model is known as feature selection.
  - Increase the value of **k** in our algorithm.
- We can normalize the columns to prevent any single value having too much of an impact on distance. Normalizing the values to a standard normal distribution preserves the distribution while aligning the scales. Let  $x$  be a value in a specific column,  $\mu$  be the mean of all values within a single column, and  $\sigma$  be the standard deviation of the values within a single column, then the mathematical formula to normalize the values is as follows:

$$x = \frac{x - \mu}{\sigma}$$

- The **distance.euclidean()** function from **scipy.spatial** expects:
  - Both of the vectors to be represented using a list-like object (Python list, NumPy array, or pandas Series).
  - Both of the vectors must be 1-dimensional and have the same number of elements.
- The scikit-learn library is the most popular machine learning library in Python. Scikit-learn contains functions for all of the major machine learning algorithms implemented as a separate class. The workflow consists of four main steps:
  - Instantiate the specific machine learning model you want to use.
  - Fit the model to the training data.
  - Use the model to make predictions.
  - Evaluate the accuracy of the predictions.
- One main class of machine learning models is known as a regression model, which predicts numerical value. The other main class of machine learning models is called classification, which is used when we're trying to predict a label from a fixed set of labels.
- The fit method accepts list-like objects while the predict method accepts matrix like objects.
- The **mean\_squared\_error()** function takes in two inputs:
  - A list-like object representing the actual values.
  - A list like object representing the predicted values using the model.

## Syntax

- Displaying the number of non-null values in the columns of a DataFrame:

```
dc_listings.info()
```

- Removing rows from a DataFrame that contain a missing value:

```
dc_listings.dropna(axis=0, inplace=True)
```

- Normalizing a column using pandas:

```
first_transform = dc_listings['maximum_nights'] -
dc_listings['maximum_nights'].mean()
```

```
normalized_col = first_transform / first_transform.std()
```

- Normalizing a DataFrame using pandas:

```
normalized_listings = (dc_listings - dc_listings.mean()) /
(dc_listings.std())
```

- Calculating Euclidean distance using SciPy:

```
from scipy.spatial import distance

first_listing = [-0.596544, -0.439151]

second_listing = [-0.596544, 0.412923]

dist = distance.euclidean(first_listing, second_listing)
```

- Using the KNeighborsRegressor to instantiate an empty model for K-Nearest Neighbors:

```
from sklearn.neighbors import KNeighborsRegressor

knn = KNeighborsRegressor()
```

- Using the fit method to fit the K-Nearest Neighbors model to the data:

```
train_df = normalized_listings.iloc[0:2792]

test_df = normalized_listings.iloc[2792:]

train_features = train_df[['accommodates', 'bathrooms']]

train_target = train_df['price']

knn.fit(train_features, train_target)
```

- Using the predict method to make predictions on the test set:

```
predictions = knn.predict(test_df[['accommodates', 'bathrooms']])
```

- Calculating MSE using scikit-learn:

```
from sklearn.metrics import mean_squared_error

two_features_mse = mean_squared_error(test_df['price'], predictions)
```

## ***Resources***

- [Scikit-learn library](#)
- [K-Neighbors Regressor](#)

# Hyperparameter Optimization

## Concepts

- Hyperparameters are values that affect the behavior and performance of a model that are unrelated to the data. Hyperparameter optimization is the process of finding the optimal hyperparameter value.
- Grid search is a simple but common hyperparameter optimization technique, which involves evaluating the model performance at different k values and selecting the k value that resulted in the lowest error. Grid search involves:
  - Selecting a subset of the possible hyperparameter values.
  - Training a model using each of these hyperparameter values.
  - Evaluating each model's performance.
  - Selecting the hyperparameter value that resulted in the lowest error value.
- The general workflow for finding the best model is:
  - Selecting relevant features to use for predicting the target column.
  - Using grid search to find the optimal hyperparameter value for the selected features.
  - Evaluate the model's accuracy and repeat the process.

## Syntax

- Using Grid Search to find the optimal k value:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
cols = ['accommodates', 'bedrooms', 'bathrooms', 'number_of_reviews']
hyper_params = [x for x in range(5)]
mse_values = list()
for value in hyper_params:
 knn = KNeighborsRegressor(n_neighbors=value, algorithm='brute')
 knn.fit(train_df[cols], train_df['price'])
 predictions = knn.predict(test_df[cols])
 mse = mean_squared_error(test_df['price'], predictions)
 mse_values.append(mse)
```

- Plotting to visualize the optimal k value:

```

features = ['accommodates', 'bedrooms', 'bathrooms',
'number_of_reviews']

hyper_params = [x for x in range(1, 21)]

mse_values = list()

for hp in hyper_params:

 knn = KNeighborsRegressor(n_neighbors=hp, algorithm='brute')

 knn.fit(train_df[features], train_df['price'])

 predictions = knn.predict(test_df[features])

 mse = mean_squared_error(test_df['price'], predictions)

 mse_values.append(mse)

plt.scatter(hyper_params, mse_values)

plt.show()

```

## **Resources**

- [Difference Between Parameter and Hyperparameter](#)
- [Hyperparameter Optimization](#)

# Cross Validation

## Concepts

- Holdout validation is a more robust technique for testing a machine learning model's accuracy on new data the model wasn't trained on. Holdout validation involves:
  - Splitting the full data set into two partitions:
    - A training set.
    - A test set.
  - Training the model on the training set.
  - Using the trained model to predict labels on the test set.
  - Computing an error to understand the model's effectiveness.
  - Switching the training and test sets and repeat.
  - Averaging the errors.
- In holdout validation, we use a 50/50 split instead of the 75/25 split from train/test validation to eliminate any sort of bias towards a specific subset of data.
- Holdout validation is a specific example of k-fold cross-validation, which takes advantage of a larger proportion of the data during training while still rotating through different subsets of the data, when **k** is set to two.
- K-fold cross-validation includes:
  - Splitting the full data set into **k** equal length partitions:
    - Selecting **k-1** partitions as the training set.
    - Selecting the remaining partition as the test set.
  - Training the model on the training set.
  - Using the trained model to predict labels on the test fold.
  - Computing the test fold's error metric.
  - Repeating all of the above steps **k-1** times, until each partition has been used as the test set for an iteration.
  - Calculating the mean of the **k** error values.
- The parameters for the KFold class are:
  - **n\_splits**: The number of folds you want to use.

- **shuffle**: Toggle shuffling of the ordering of the observations in the data set.
  - **random\_state**: Specify the random seed value if **shuffle** is set to **True**.
- The parameters for using **cross\_val\_score** are:
  - **estimator**: Scikit-learn model that implements the **fit** method (e.g. instance of **KNeighborsRegressor**).
  - **X**: The list or 2D array containing the features you want to train on.
  - **y**: A list containing the values you want to predict (target column).
  - **scoring**: A string describing the scoring criteria.
  - **cv**: The number of folds. Here are some examples of accepted values:
    - An instance of the **KFold** class.
    - An integer representing the number of folds.
- The workflow for k-fold cross-validation with scikit-learn includes:
  - Instantiating the scikit-learn model class you want to fit.
  - Instantiating the **KFold** class and using the parameters to specify the k-fold cross-validation attributes you want.
  - Using the **cross\_val\_score()** function to return the scoring metric you're interested in.
- Bias describes error that results in bad assumptions about the learning algorithm. Variance describes error that occurs because of the variability of a model's predicted value. In an ideal world, we want low bias and low variance when creating machine learning models.

## Syntax

- Implementing holdout validation:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
train_one = split_one
test_one = split_two
train_two = split_two
test_two = split_one
model = KNeighborsRegressor()
model.fit(train_one[["accommodates"]], train_one["price"])
test_one["predicted_price"] = model.predict(test_one[["accommodates"]])
```

```

iteration_one_rmse = mean_squared_error(test_one["price"],
test_one["predicted_price"])**(1/2)

model.fit(train_two[["accommodates"]], train_two["price"])

test_two["predicted_price"] = model.predict(test_two[["accommodates"]])

iteration_two_rmse = mean_squared_error(test_two["price"],
test_two["predicted_price"])**(1/2)

avg_rmse = np.mean([iteration_two_rmse, iteration_one_rmse])

```

- Implementing k-fold cross validation:

```

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error

model = KNeighborsRegressor()

train_iteration_one = dc_listings[dc_listings["fold"] != 1]

test_iteration_one = dc_listings[dc_listings["fold"] == 1].copy()

model.fit(train_iteration_one[["accommodates"]],
train_iteration_one["price"])

labels = model.predict(test_iteration_one[["accommodates"]])

test_iteration_one["predicted_price"] = labels

iteration_one_mse = mean_squared_error(test_iteration_one["price"],
test_iteration_one["predicted_price"])

iteration_one_rmse = iteration_one_mse ** (1/2)

```

- Instantiating an instance of the KFold class from sklearn.model\_selection:

```

from sklearn.model_selection import cross_val_score, KFold

kf = KFold(5, shuffle=True, random_state=1)

```

- Implementing cross\_val\_score along with the KFold class:

```

from sklearn.model_selection import cross_val_score

model = KNeighborsRegressor()

mses = cross_val_score(model, dc_listings[["accommodates"]],
dc_listings["price"], scoring="neg_mean_squared_error", cv=kf)

```

## Resources

- [Accepted values for scoring criteria](#)
- [Bias-variance Trade-off](#)
- [K-Fold cross-validation documentation](#)

# Understanding Linear and Nonlinear Functions

## Concepts

- Calculus helps us:
  - Understand the steepness at various points.
  - Find the extreme points in a function.
  - Determine the optimal function that best represents a dataset.
- A linear function is a straight line.
- If m and b are constant values where x and y are variables then the function for a linear function is:

$$y = mx + b$$

- In a linear function, the m value controls of steep a line is while the b value controls a line's y-intercept or where the line crosses the y axis.
- One way to think about slope is as a rate of change. Put more concretely, slope is how much the y axis changes for a specific change in the x axis. If  $(x_1, y_1)$  and  $(x_2, y_2)$  are 2 coordinates on a line, the slope equation is:

$$m = \frac{y_1 - y_2}{x_1 - x_2}$$

- When  $x_1$  and  $x_2$  are equivalent, the slope is undefined because the division of 0 has no meaning in mathematics.
- Nonlinear functions represent curves, and there output values (y) are not proportional to their input values (x).
- Examples of non-linear functions include:

- $y = x^3$
- $y = x^3 + 3x^2 + 2x - 1$
- $y = \frac{1}{-x^2}$
- $y = \sqrt{x}$

- A line that intersects two points on a curve is known a secant line.
- The slope between any two given points is known as the instantaneous rate of change. For linear functions, the rate of change at any point on the line is the same. For nonlinear function,

the instantaneous rate of change describes the slope of the line that's perpendicular to the nonlinear function at a specific point.

- The line that is perpendicular to the nonlinear function at a specific point is known as the tangent line, and only intersects the function at one point.

## Syntax

- Generating a NumPy array containing 301 values:

```
import numpy as np
np.linspace(0, 3, 100)
```

- Plotting  $y = -(x^2) + 3x - 1$ :

```
import numpy as np
np.linspace(0, 3, 100)

y = -1 * (x ** 2) + 3*x - 1

plt.plot(x,y)
```

- Plotting a secant line:

```
def draw_secant(x_values):

 x = np.linspace(-20,30,100)

 y = -1*(x**2) + x*3 - 1

 plt.plot(x,y)

 x_0 = x_values[0]

 x_1 = x_values[1]

 y_0 = -1*(x_0**2) + x_0*3 - 1

 y_1 = -1*(x_1**2) + x_1*3 - 1

 m = (y_1 - y_0) / (x_1 - x_0)

 b = y_1 - m*x_1

 y_secant = x*m + b

 plt.plot(x, y_secant, c='green')

 plt.show()
```

## Resources

- [Calculus](#)
- [Instantaneous Rate of Change](#)
- [Secant Line](#)

- [Division by zero](#)

# Understanding Limits

## Concepts

- A limit describes the value a function approaches when the input variable to the function approaches a specific value. A function at a specific point may have a limit even though the point is undefined.
- The following mathematical notation formalizes the statement "As  $x_2$  approaches 3, the slope between  $x_1$  and  $x_2$  approaches -3" using a limit:

$$\lim_{x_2 \rightarrow 3} \frac{f(x_2) - f(x_1)}{x_2 - x_1} = -3$$

- A defined limit can be evaluated by substituting the value into the limit. Whenever the resulting value of a limit is defined at the value the input variable approaches, we say that limit is defined.
- The SymPy library has a suite of functions that let us calculate limits. When using SymPy, it's critical to declare the Python variables you want to use as symbols as SymPy maps the Python variables directly to variables in math when you pass them through `sympy.symbols()`.
- The `sympy.limit()` function takes in three parameters:
  - The function we're taking the limit for.
  - The input variable.
  - The value the input variable approaches.
- Properties of Limits:
  - Sum Rule:  $\lim_{x \rightarrow a} [f(x) + g(x)] = \lim_{x \rightarrow a} f(x) + \lim_{x \rightarrow a} g(x)$
  - Difference Rule:  $\lim_{x \rightarrow a} [f(x) - g(x)] = \lim_{x \rightarrow a} f(x) - \lim_{x \rightarrow a} g(x)$
  - Constant Function Rule:  $\lim_{x \rightarrow a} [cf(x)] = c \lim_{x \rightarrow a} f(x)$

## Syntax

- Importing Sympy and declaring the variables as symbols:

```
import sympy
x, y = sympy.symbols('x y')
```
- Using SymPy to calculate a limit:

```
limit_one = sympy.limit(x**2 + 1, x, 1)
```

## Resources

- [sympy.symbols\(\) Documentation](#)

- [Proofs of Properties of Limits](#)

# Finding Extreme Points

## Concepts

- A derivative is the slope of the tangent line at any point along a curve.
- Let  $x$  be a point on the curve and  $h$  be the distance between two points, then the mathematical formula for the slope as  $h$  approaches zero is given as:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- Differentiation is the process of finding a function's derivative.
- Finding the derivative of:  $f(x) = -(x)^2 + 3x - 1$ :

$$\begin{aligned} \bullet \quad y' &= \lim_{h \rightarrow 0} \frac{(-(x+h)^2 + 3(x+h) - 1) - (-(x)^2 + 3x - 1)}{h} \\ \bullet \quad y' &= \lim_{h \rightarrow 0} \frac{-x^2 - 2xh - h^2 + 3x + 3h - 1 + x^2 - 3x + 1}{h} \\ \bullet \quad y' &= \lim_{h \rightarrow 0} \frac{h(-2x - h + 3)}{h} \\ \bullet \quad y' &= \lim_{h \rightarrow 0} -2x - h + 3 \\ \bullet \quad y' &= -2x + 3 \end{aligned}$$

- Three ways of notating a curve's derivative:
  - $y' = -2x + 3$
  - $f'(x) = -2x + 3$  \*Only use if derivative is a function
  - $\frac{d}{dx}[-x^2 + 3x - 1] = -2x + 3$
- A critical point is a point where the slope changes direction from negative slope to positive slope or vice-versa. Critical points represent extreme values, which can be classified as a minimum or maximum value.
- Critical points are found by setting the derivative function to 0 and solving for  $x$ .
- Critical point classification:
  - When the slope changes direction from positive to negative it can be a maximum value.
  - When the slope changes direction from negative to positive, it can be a minimum value.
  - If the slope doesn't change direction, like at  $x=0$  for  $y=x^3$ , then it can't be a minimum or maximum value.
- Each maximum or minimum value points are known as local extrema.
- Classifying local extrema:

- A point is a relative minimum if a critical point is the lowest point in a given interval.
  - A point is a relative maximum if a critical point is the highest point in a given interval.
- Instead of using the definition of the derivative, we can apply derivative rules to easily calculate the derivative functions.
- Derivative rules:
  - Power rule: Let  $r$  be some power, then  $f'(x) = rx^{r-1}$ 
    - Example: Let  $f(x) = x^2$ . In our function,  $r$  would be 2. Using the power rule, its derivative would be  $f'(x) = 2x^{2-1}$  or  $f'(x) = 2x$
  - Sum rule:  $\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}[f(x)] + \frac{d}{dx}[g(x)]$ 
    - Example:  $\frac{d}{dx}[-x^3 + x^2] = \frac{d}{dx}[-x^3] + \frac{d}{dx}[x^2] = -3x^2 + 2x$
  - Constant factor rule:  $\frac{d}{dx}[3x] = 3\frac{d}{dx}x = 3 \cdot 1 = 3$
- Derivative of  $x$  is always 1 and derivative of 1 is always 0.
- Once you found the critical points of a function, you can analyze the direction of the slope around the points using a sign chart to classify the point as a minimum or maximum. We can test points around our points of interest to see if there is a sign change as well as what the change is.

## **Resources**

- [Derivative rules](#)
- [Sign chart](#)

# Linear Systems

## Concepts

- Linear algebra provides a way to represent and understand the solutions to systems of linear equations. We represent linear equations in the general form of  $Ax+By=c$ .
- A system of linear equations consists of multiple, related functions with a common set of variables. The point where the equations intersect is known as a solution to the system.
- The elimination method involves representing one of our variables in terms of a desired variable and substituting the equation that is in terms of the desired variable.
  - Suppose we have the equations  $y=1000+30x$  and  $y=100+50x$ . Since both are equal to  $y$ , we can substitute in the second function with the first function. The following are the steps to solve our example using the elimination method:
    - $1000 + 30x = 100 + 50x$
    - $900 = 20x$
    - $45 = x$
- A matrix uses rows and columns to represent only the coefficients in a linear system, and it's similar to the way data is represented in a spreadsheet or a DataFrame.
- Gaussian elimination is used to solve systems of equation that are modeled by many variables and equations.
- In an augmented matrix, the coefficients from the left side of the function are on the left side of the bar ( $|$ ), while the constraints from the right sides of the function are on the right side.
- To preserve the relationships in the linear system, we can use the following row operations:
  - Any two rows can be swapped.
  - Any row can be multiplied by a nonzero constant.
  - Any row can be added to another row.
- To solve an augmented matrix, you'll have to rearrange the matrix into echelon form. In this form, the values on the diagonal are all equal to 1 and the values below the diagonal are equal to 0.

## Syntax

- Representing a matrix as an array:

```
import numpy as np
matrix_one = np.asarray([
```

```
[0, 0, 0],
[0, 0, 0]
, dtype=np.float32)

• Multiplying a row by a nonzero constant:

matrix[1] = 2*matrix[1]

• Adding one row to another row:

matrix[1] = matrix[1] + matrix[0]

• Combining and chaining row operations:

matrix[1] = 0.5*matrix[2] + matrix[1] + matrix[3]
```

## Resources

- [General form](#)
- [Elimination method](#)
- [Gaussian Elimination](#)
- [Linear algebra](#)

# Vectors

## Syntax

- Visualizing vectors in matplotlib:

```
import matplotlib.pyplot as plt
plt.quiver(0, 0, 1, 2)
```

- Setting the color of each vector:

```
plt.quiver(0, 0, 1, 2, angles='xy', scale_units='xy', scale=1,
 color='blue')
```

- Multiplying and adding vectors:

```
vector_one = np.asarray([
 [1],
 [2],
 [1]
, dtype=np.float32)

vector_two = 2*vector_one + 0.5*vector_one
```

- Computing the dot product:

```
vector_dp = np.dot(vector_one[:,0], vector_two)
```

## Concepts

- When referring to matrices, the convention is to specify the number of rows first then the number of columns. For example, a matrix containing two rows and three columns is known as a 2x3 matrix.
- A list of numbers in a matrix is known as a vector, a row from a matrix is known as a row vector, and a column from a matrix is known as a column vector.
- A vector can be visualized on a coordinate grid when a vector contains two or three elements. Typically, vectors are drawn from the origin (0,0) to the point described by the vector.
- Arrows are used to visualize individual vectors because they emphasize two properties of a vector — direction and magnitude. The direction of a vector describes the way it's pointing while the magnitude describes its length.
- The `pyplot.quiver()` function takes in four required parameters: **X**, **Y**, **U**, and **V**. **X** and **Y** correspond to the (x,y) coordinates we want the vector to start at while **U** and **V** correspond the (x,y) coordinate we want to draw the vector from.

- The optional parameters: **angles**, **scale\_units** and **scale** always want to be used when plotting vectors. Setting angles to '**xy**' lets matplotlib know we want the angle of the vector to be between the points we specified. The **scale\_units** and **scale**parameters lets us specify custom scaling parameters for the vectors.
- Similar to rows in a matrix, vectors can be added or subtracted together. To add or subtract vectors, you add the corresponding elements in the same position. Vectors can also be scaled up by multiplying the vector by a real number greater than 1 or less than -1. Vectors can also be scaled down by multiplying the vector by a number between -1 and 1.
- To compute the dot product, we need to sum the products of the 2 values in each position in each vector. The equation to compute the dot product is:

$$\vec{a} * \vec{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

where a and b are vectors.

- A linear combination are vectors that are scaled up and then added or subtracted.
- The arithmetic representation of the matrix equation is  $Ax \rightarrow = b$  where where A represents the coefficient matrix, x → represents the solution vector, and b → represents the constants. Note that b → can't be a vector containing all zeros, also known as the zero factor and represented using 0.

## **Resources**

- [Vector operations](#)
- [plt.quiver\(\)](#)

# Matrix Algebra

## Concepts

- Many operations that can be performed on vectors can also be performed on matrices. With matrices, we can perform the following operations:
  - Add and subtract matrices containing the same number of rows and calculations.
  - Multiply a matrix by a scalar value.
  - Multiply a matrix with a vector and other matrices. To multiply a matrix by a vector or another matrix, the number of columns must match up with the number of rows in the vector or matrix. The order of multiplication does matter when multiplying matrices and vectors.
- Taking the transpose of a matrix switches the rows and columns of a matrix. Mathematically, we use the notation  $A^T$  to specify the transpose operation.
- The identity matrix contains 1 along the diagonal and 0 elsewhere. The identity matrix is often represented using  $I_n$  where  $n$  is the number of rows and columns.
- When we multiply with any vector containing two elements, the resulting vector matches the original vector exactly.
- To transform  $A$  into the identity matrix  $I$  in  $Ax \rightarrow = b \rightarrow$ , we multiply each side by the inverse of matrix  $A$ .
- The inverse of a  $2 \times 2$  matrix can be found using the following:

$$\text{If } A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \text{ then } A^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- If the determinant of a  $2 \times 2$  matrix or  $ad-bc$  is equal to 0, the matrix is invertible. We can compute the determinant and matrix inverse only for matrices that have the same number of rows in columns. These matrices are known as square matrices.
- To compute the determinant of a matrix other than a  $2 \times 2$  matrix, we can break down the full matrix into minor matrices.

## Syntax

- Using NumPy to multiply a matrix and a vector:

```
matrix_a = np.asarray([
 [0.7, 3, 9],
 [1.7, 2, 9],
 [0.7, 9, 2]
```

```
], dtype=np.float32)

vector_b = np.asarray([
[1], [2], [1]
], dtype=np.float32)

ab_product = np.dot(matrix_a, vector_b)
```

- Using NumPy to compute the transpose of a matrix:

```
matrix_b = np.asarray([
[113, 3, 10],
[1, 0, 1],
], dtype=np.float32)

transpose_b = np.transpose(matrix_b)
```

- Using NumPy to create an identity matrix:

```
np.identity(3) # creates the 3x3 identity matrix
```

- Using NumPy to compute the inverse of a matrix:

```
matrix_c = np.asarray([
[30, -1],
[50, -1]
])

matrix_c_inverse = np.linalg.inv(matrix_c)
```

- Using NumPy to compute the determinant of a matrix:

```
matrix_22 = np.asarray([
[8, 4],
[4, 2]
])

det_22 = np.linalg.det(matrix_22)
```

## Resources

- [Documentation for the dot product of two arrays](#)
- [Identity matrix](#)
- [Documentation for the transpose of an array](#)

# Solution Sets

## Concepts

- An inconsistent system has two or more equations that no solution exists when the augmented matrix is in reduced echelon form.

Example of a inconsistent system: 
$$\left[ \begin{array}{cc|c} 8 & 4 & 5 \\ 4 & 2 & 5 \end{array} \right]$$

- When the determinant is equal to zero, we say the matrix is singular or it contains no inverse.

- Example of a singular matrix: 
$$\left[ \begin{array}{cc} 8 & 4 \\ 4 & 2 \end{array} \right]$$
- The formula for the determinant of a  $2 \times 2$  square matrix is:

$$\det(A) = ad - bc$$

- If we substitute in the values, we get a determinant of zero:

$$\det(A) = ad - bc = 8 \cdot 2 - 4 \cdot 4 = 16 - 16 = 0$$

- A nonhomogenous system is a system where the constants vector ( $b \rightarrow$ ) doesn't contain all zeros.

Example of a nonhomogenous system: 
$$\left[ \begin{array}{cc|c} 8 & 4 & 5 \\ 4 & 2 & 5 \end{array} \right]$$

- A homogenous system is a system where the constants vector ( $b \rightarrow$ ) us equal to the zero vector.

Example of a omogenous system: 
$$\left[ \begin{array}{cc|c} 8 & 4 & 0 \\ 4 & 2 & 0 \end{array} \right]$$

A homogenous system always contains the trivial solution: the zero vector.

- For a nonhomogenous system that contains the same number of rows and columns, there are 3 possible solutions:
  - No solution.
  - A single solution.
  - Infinitely many solutions.
- For rectangular (nonsquare, nonhomogenous) systems, there are two possible solutions:
  - No solution.

- Infinitely many solutions.
- If  $Ax=b$  is a linear system, then every vector  $x \rightarrow$  which satisfies the system is said to be a solution vector of the linear system. The set of solution vectors of the system is called the solution space of the linear system.
- When the solution is a solution space (and not just a unique set of values), it's common to rewrite it into parametric vector form.

Example a vector in parametric vector form:  $\vec{x} = x_3 * \begin{bmatrix} 4/3 \\ 0 \\ 1 \end{bmatrix}$

## **Resources**

- [Consistent and Inconsistent equations](#)
- [Solution Spaces of Homogenous Linear Systems](#)

# The Linear Regression Model

## Concepts

- An instance-based learning algorithm, such as K-nearest neighbors, relies completely on previous instances to make predictions. K-nearest neighbors doesn't try to understand or capture the relationship between the feature columns and the target column.
- Parametric machine learning, like linear regression and logistic regression, results in a mathematical function that best approximates the patterns in the training set. In machine learning, this function is often referred to as a model. Parametric machine learning approaches work by making assumptions about the relationship between the features and the target column.
- The following equation is the general form of the simple linear regression model:

$$\hat{y} = a_1 x_1 + a_0$$

where  $y^{\wedge}$  represents the target column while  $x_1$  represents the feature column we chose to use in our model.  $a_0$  and  $a_1$  represent the parameter values that are specific to the dataset.

- The goal of simple linear regression is to find the optimal parameter values that best describe the relationship between the feature column and the target column.
- We minimize the model's residual sum of squares to find the optimal parameters for a linear regression model. The equation for the model's residual sum of squares is as follows:

$$RSS = (y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + \dots + (y_n - \hat{y}_n)^2$$

where  $y_n^{\wedge}$  is our target column and  $y$  are our true values.

- A multiple linear regression model allows us to capture the relationship between multiple feature columns and the target column. The formula for multiple linear regression is as follows:

$$\hat{y} = a_0 + a_1 x_1 + a_2 x_2 + \dots + a_n x_n$$

where  $x_1$  to  $x_n$  are our feature columns, and the parameter values that are specific to the data set are represented by  $a_0$  along with  $a_1$  to  $a_n$ .

- In linear regression, it is a good idea to select features that are a good predictor of the target column.

## Syntax

- Importing and instantiating a linear regression model:

```
from sklearn.linear_model import LinearRegression
```

- ```
lr = LinearRegression()
```
- Using the `LinearRegression`
 - class to fit a linear regression model between a set of columns:

```
lr.fit(train[['Gr Liv Area']], train['SalePrice'])
```
 - Returning the a1 and a0 parameters for $y=a0+a1x1$:

```
a0 = lr.intercept_
a1 = lr.coef_
```
 - Predicting the labels using the training data:

```
test_predictions = lr.predict(test[['Gr Liv Area']])
```
 - Calculating the correlation between pairs of columns:

```
train[['Garage Area', 'Gr Liv Area', 'Overall Cond',
'SalePrice']].corr()
```

Resources

- [Linear Regression Documentation](#)
- [pandas.DataFrame.corr\(\) Documentation](#)

Feature Selection

Concepts

- Once we select the model we want to use, selecting the appropriate features for that model is the next important step. When selecting features, you'll want to consider correlations between features and the target column, correlation with other features, and the variance of features.
- Along with correlation with other features, we need to also look for potential collinearity between some of the feature columns. Collinearity is when two feature columns are highly correlated and have the risk of duplicating information.
- We can generate a correlation matrix heatmap using Seaborn to visually compare the correlations and look for problematic pairwise feature correlations.
- Feature scaling helps ensure that some columns aren't weighted more than others when helping the model make predictions. We can rescale all of the columns to vary between 0 and 1. This is known as min-max scaling or rescaling. The formula for rescaling is as follows:

$$\frac{x - \min(x)}{\max(x) - \min(x)}$$

where x is the individual value, $\min(x)$ is the minimum value for the column x belongs to, and $\max(x)$ is the maximum value for the column x belongs to.

Syntax

- Using Seaborn to generate a correlation matrix heatmap:

```
sns.heatmap(DataFrame)
```

- Rescaling the features for a model:

```
data = pd.read_csv('AmesHousing.txt', delimiter="\t")
train = data[0:1460]
unit_train = (train[['Gr Liv Area']] - train['Gr Liv
Area'].min())/(train['Gr Liv Area'].max() - train['Gr Liv Area'].min())
```

Resources

- [seaborn.heatmap\(\) documentation](#)
- [Feature scaling](#)

Gradient Descent

Syntax

- Implementing gradient descent for 10 iterations:

```
a1_list = [1000]
alpha = 10
for x in range(0, 10):
    a1 = a1_list[x]
    deriv = derivative(a1, alpha, xi_list, yi_list)
    a1_new = a1 - alpha*deriv
    a1_list.append(a1_new)
```

Concepts

- The process of finding the optimal unique parameter values to form a unique linear regression model is known as model fitting. The goal of model fitting is to minimize the mean squared error between the predicted labels made using a given model and the true labels. The mean squared error is as follows:

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

- Gradient descent is an iterative technique for minimizing the squared error. Gradient descent works by trying different parameter values until the model with the lowest mean squared error is found. Gradient descent is a commonly used optimization technique for other models as well. An overview of the gradient descent algorithm is as follows:

- Select initial values for the parameter a_1 .
- Repeat until convergence (usually implemented with a max number of iterations):
 - Calculate the error (MSE) of the model that uses current parameter value:
 $MSE(a_1) = \frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2$
 - Calculate the derivative of the error (MSE) at the current parameter value:
 $\frac{d}{da_1} MSE(a_1)$
 - Update the parameter value by subtracting the derivative times a constant (α , called the learning rate): $a_1 := a_1 - \alpha \frac{d}{da_1} MSE(a_1)$
- Univariate case of gradient descent:

- The function that we optimize through minimization is known as a cost function or as the loss function. In our case, the loss function is:

$$MSE(a_1) = \frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2$$

- Applying calculus properties to simplify the derivative of the loss function:

- Applying the linearity of differentiation property, we can bring the constant term outside the summation:

$$\cdot \frac{d}{da_1} MSE(a_1) = \frac{1}{n} \sum_{i=1}^n \frac{d}{da_1} (a_1 x_1^{(i)} - y^{(i)})^2$$

- Using the power rule and the chain rule to simplify:

$$\cdot \frac{d}{da_1} MSE(a_1) = \frac{1}{n} \sum_{i=1}^n 2(a_1 x_1^{(i)} - y^{(i)}) \frac{d}{da_1} (a_1 x_1^{(i)} - y^{(i)})$$

- Because we're differentiating with respect to a_1 , we treat $y^{(i)}$ and $x_1^{(i)}$ as constants.

$$\cdot \frac{d}{da_1} MSE(a_1) = \frac{2}{n} \sum_{i=1}^n x_1^{(i)} (a_1 x_1^{(i)} - y^{(i)})$$

- For every iteration of gradient descent:

- The derivative is computed using the current a_1 value.
- The derivative is multiplied by the learning rate (α): $\alpha \frac{d}{da_1} MSE(a_1)$. The result is subtracted from the current parameter value and assigned as the new parameter value: $a_1 := a_1 - \alpha \frac{d}{da_1} MSE(a_1)$

- Multivariate case of gradient descent:

- When we have two parameter values (a_0 and a_1), the cost function is now a function of two variables instead of one. Our new cost function is:

$$MSE(a_0, a_1) = \frac{1}{n} \sum_{i=1}^n (a_0 + a_1 x_1^{(i)} - y^{(i)})^2$$

- We also need two update rules:

$$\cdot a_0 := a_0 - \alpha \frac{d}{da_0} MSE(a_0, a_1)$$

$$\cdot a_1 := a_1 - \alpha \frac{d}{da_1} MSE(a_0, a_1)$$

- Computed derivative for the multivariate case:

$$\cdot \frac{d}{da_1} MSE(a_0, a_1) = \frac{2}{n} \sum_{i=1}^n x_1^{(i)} (a_0 + a_1 x_1^{(i)} - y^{(i)})$$

- Gradient descent scales to as many variables as you want. Keep in mind each parameter value will need its own update rule, and it closely matches the update for a_1 . The derivative for other parameters are also identical.
- Choosing good initial parameter values and choosing a good learning rate are the main challenges with gradient descent.

Resources

- [Mathematical Optimization](#)
- [Loss Function](#)

Ordinary Least Squares

Concepts

- The ordinary least squares estimation provides a clear formula for directly calculating the optimal values that maximizes the cost function.
- The OLS estimation formula that results in optimal vector a :

$$a = (X^T X)^{-1} X^T y$$

- OLS estimation provides a closed form solution to the problem of finding the optimal parameter values. A closed form solution is where a solution can be computed arithmetically with a predictable amount of mathematical operations.
- The error for OLS estimation is often represented using the Greek letter for E . Since the error is the difference between the predictions made using the model and the actual labels, it's represented as a vector:

$$\epsilon = \hat{y} - y$$

- We can use the error metric to define y :

$$y = Xa - \epsilon$$

- The cost function in matrix form:

$$J(a) = \frac{1}{n} (Xa - y)^T (Xa - y)$$

- The derivative of the cost function:

$$\frac{dJ(a)}{da} = 2X^T Xa - 2X^T y$$

- Minimizing the cost function, $J(a)$.

- Set the derivative equal to 0 and solve for a :

- $2X^T Xa - 2X^T y = 0$

- Compute the inverse of X and multiply both sides by the inverse:

- $a = (X^T X)^{-1} X^T y$

- The biggest limitation of OLS estimation is that it's computationally expensive when the data is large. Computing the inverse of a matrix has a computational complexity of approximately $O(n^3)$.
- OLS is computationally expensive, and so is commonly used when the numbers of elements in the dataset is less than a few million elements.

Syntax

- Finding the optimal parameter values using ordinary least squares (OLS) :

```
first_term = np.linalg.inv(  
    np.dot(  
        np.transpose(X),  
        X  
    )  
)  
  
second_term = np.dot(  
    np.transpose(X),  
    Y  
)  
  
a = np.dot(first_term, second_term)  
print(a)
```

Resources

- [Walkthrough of the derivative of the cost function](#)
- [Ordinary least squares](#)

Processing and Transforming Features

Syntax

- Converting any column to the categorical data type:

```
train['Utilities'] = train['Utilities'].astype('category')
```

- Accessing the underlying numerical representation of a column:

```
train['Utilities'].cat.codes
```

- Applying dummy coding for all of the text columns:

```
dummy_cols = pd.get_dummies()
```

- Replace all missing values in a column with a specified value:

```
fill_with_zero = missing_floats.fillna(0)
```

Concepts

- Feature engineering is the process of processing and creating new features. Feature engineering is a bit of an art and having knowledge in the specific domain can help create better features.
- Categorical features are features that can take on one of a limited number of possible values.
- A drawback to converting a column to the categorical data type is that one of the assumptions of linear regression is violated. Linear regression operates under the assumption that the features are linearly correlated with the target column.
- Instead of converting to the categorical data type, it's common to use a technique called dummy coding. In dummy coding, a dummy variable is used. A dummy variable that takes the value of 0 or 1 to indicate the absence or presence of some categorical effect that may be expected to shift the outcome.
- When values are missing in a column, there are two main approaches we can take:
 - Removing rows that contain missing values for specific columns:
 - Pro: Rows containing missing values are removed, leaving only clean data for modeling.
 - Con: Entire observations from the training set are removed, which can reduce overall prediction accuracy.
 - Imputing (or replacing) missing values using a descriptive statistic from the column:
 - Pro: Missing values are replaced with potentially similar estimates, preserving the rest of the observation in the model.
 - Con: Depending on the approach, we may be adding noisy data for the model to learn.

Resources

- [Feature Engineering](#)
- [Dummy Coding](#)
- [pandas.DataFrame.fillna\(\)](#)

Logistic regression

Concepts

- In classification, our target column has a finite set of possible values, which represent different categories a row can belong to.
- In binary classification, there are only two options for values:
 - **0** for the False condition.
 - **1** for the True condition.
- Categorical values are used to represent different options or categories. Classification focuses on estimating the relationship between the independent variables and the dependent categorical variable.
- One technique of classification is called logistic regression. While a linear regression model outputs a real number as the label
- The logistic function is a version of the linear function that is adapted for classification. Mathematically, the logistic function is represented as the following:

$$\sigma(t) = \frac{e^t}{1 + e^t}$$

where e^t is the exponential transformation to transform all values to be positive, and $\frac{t}{1+e^t}$ is the normalization transformation to transform all values between **0** and **1**.

Syntax

- Defining the logistic function:

```
def logistic(x):
    """
    np.exp(x) raises x to the exponential power e^x. e ~= 2.71828
    """
    return np.exp(x) / (1 + np.exp(x))
```

- Instantiating a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
linear_model = LogisticRegression()
```

- Training a logistic regression model:

```
logistic_model.fit(admissions[["gpa"]], admissions["admit"])
```

- Returning predicted probabilities for a column:

```
pred_probs = logistic_model.predict_proba(admission[["gpa"]])
```

- , a logistic regression model outputs a probability value.

Resources

- [Documentation for the LogisticRegression class](#)
- [Documentation for the predict_proba method](#)

Introduction to Evaluating Binary Classifiers

Concepts

- Prediction accuracy is the simplest way to determine the effectiveness of a classification model. Prediction accuracy can be calculated by the number of labels correctly predicted divided the total number of observations:

$$Accuracy = \frac{\# \text{ of Correctly Predicted}}{\# \text{ of Observations}}$$

- A discrimination threshold is used to determine what labels are assigned based on their probability. Scikit-learn sets the discrimination threshold to 0.5 by default when predicting labels.
 - For example, if the predicted probability is greater than 0.5, the label for that observation is 1. If it is less than 0.5, the label for that observation is 0.
- There are four different outcomes of a binary classification model:
 - True Positive: The model correctly predicted the label as positive.
 - True Negative: The model correctly predicted the label as negative.
 - False Positive: The model falsely predicted the label as positive.
 - False Negative: The model falsely predicted the label as negative.
- Sensitivity, or True Positive Rate, is the proportion of labels that were correctly predicted as positive. Mathematically, this is written as:

$$TPR = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- Sensitivity helps of answer "How effective is this model at identifying positive outcomes?"
- Specificity, or True Negative Rate, is the proportion of of labels that were correctly predicted as negative. Mathematically, this is written as:

$$TNR = \frac{\text{True Negatives}}{\text{False Positives} + \text{True Negatives}}$$

- Specificity helps us answer "How effective is the model at identifying negative outcomes?"

Resources

- [Sensitivity and Specificity](#)

- [Discrimination threshold](#)

Multiclass classification

Concepts

- In the instance where two values are just two different labels, it is safer to turn the discrete values into categorical variables.
- Dummy variables are for columns that represent categorical values.
- A problem is a multiclass classification problem when there are three or more categories or classes. There are existing multiclassification techniques that be categorized into the following:
 - Transformation to Binary: Reducing the problem to multiple binary classification problems.
 - Extension from Binary: Extending existing binary classifiers to solve multi-class classification problems.
 - Hierarchical classification: Dividing the output into a tree where each parent node is divided into multiple child nodes and the process is continued until each child node represents only one class.
- The one-versus-all method, which is a transformation to binary technique, is a technique where we choose a single category as the Positive case and group the rest of the categories as the False case.

Syntax

- Returning a DataFrame containing binary columns:
`dummy_df = pd.get_dummies(cars["cylinders"])`
- Concatenating DataFrames:
`cars = pd.concat([cars, dummy_df], axis=1)`
- Returning a Series containing the index of the maximum value:
`predicted_origins=testing_probs.idxmax(axis=1)`

Resources

- [Documentation for idxmax\(\)](#)
- [Multiclass Classification](#)

Overfitting

Concepts

- Bias and variance are at the heart of understanding overfitting.
- Bias describes error that results in bad assumptions about the learning algorithm. Variance describes error that occurs because of the variability of a model's predicted values.
- We can approximate the bias of a model by training a few different models using different features on the same class and calculating their error scores.
- To detect overfitting, you can compare the in-sample error and the out-of-sample error, or the training error with the test error.
 - To calculate the out-of-sample error, you need to test the data on a test set of data. If you don't have a separate test data set, you can use cross-validation.
 - To calculate the in-sample-error, you can test the model over the same data it was trained on.
- When the out-of-sample error is much higher than the in-sample error, this is a clear indicator the trained model doesn't generalize well outside the training set.

Resources

- [Bias-variance tradeoff](#)
- [Blog post on the bias-variance tradeoff](#)

Clustering basics

Concepts

- Two major types of machine learning are supervised and unsupervised learning. In supervised learning, you train an algorithm to predict an unknown variable from known variables. In unsupervised learning, you're finding patterns in data as opposed to making predictions.
- Unsupervised learning is very commonly used with large data sets where it isn't obvious how to start with supervised machine learning. It's a good idea to try unsupervised learning to explore a data set before trying to use supervised machine learning models.
- Clustering is one of the main unsupervised learning techniques. Clustering algorithms group similar rows together and is a key way to explore unknown data.
- We can use the Euclidean distance formula to find the distance between two rows to group similar rows. The formula for Euclidean distance is:

$$d = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2}$$

where q_n and p_n are observations from each row.

- The k-means clustering algorithm uses Euclidean distance to form clusters of similar items.

Syntax

- Computing the Euclidean distance in Python:

```
from sklearn.metrics.pairwise import euclidean_distances  
euclidean_distances(votes.iloc[0,3:], votes.iloc[1,3:])
```

- Initializing the KMeans class from scikit-learn:

```
from sklearn.cluster import KMeans  
kmeans_model = KMeans(n_clusters=2, random_state=1)
```

- Calculating the distance between observations and the clusters:

```
senator_distances = kmeans_model.fit_transform(votes.iloc[:, 3:])
```

- Computing a frequency table of two or more factors:

```
labels = kmeans_model.labels_  
print(pd.crosstab(labels, votes["party"]))
```

Resources

- [Documentation for sklearn.cluster.KMeans](#)
- [Unsupervised Machine learning](#)
- [Redefining NBA Basketball Positions](#)

K-means clustering

Concepts

- Centroid-based clustering works well when the clusters resemble circles with centers.
- K-Means clustering is a popular centroid-based clustering algorithm. The K refers to the number of clusters we want to segment our data into. K-Means clustering is an iterative algorithm that switches between recalculating the centroid of each cluster and the items that belong to each cluster.
- Euclidean distance is the most common technique used in data science for measuring distance between vectors. The formula for distance in two dimensions is:

$$\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2}$$

where q and p are the two vectors we are comparing.

- Example: If **q** is (5,2) and **p** is (3,1), the distance comes out to:

$$\sqrt{(5 - 3)^2 + (2 - 1)^2} = \sqrt{5} \approx 2.23607$$

- If clusters look like they don't move a lot after every iteration, this means two things:
 - K-Means clustering doesn't cause massive changes in the makeup of clusters between iterations, meaning that it will always converge and become stable.
 - Where we pick the initial centroids and how we assign elements to clusters initially matters a lot because K-Means clustering is conservative between iterations.
- To counteract the problems listed above, the **sklearn** implementation of K-Means clustering does some intelligent things like re-running the entire clustering process lots of times with random initial centroids so the final results are a little less biased.

Syntax

- Computing the Euclidean distance in Python:

```
def calculate_distance(vec1, vec2):  
    root_distance = 0  
    for x in range(0, len(vec1)):  
        difference = centroid[x] - player_values[x]  
        squared_difference = difference**2
```

```

        root_distance += squared_difference

    euclid_distance = math.sqrt(root_distance)

    return euclid_distance

• Assigning observations to clusters:

def assign_to_cluster(row):
    lowest_distance = -1
    closest_cluster = -1

    for cluster_id, centroid in centroids_dict.items():

        df_row = [row['ppg'], row['atr']]

        euclidean_distance = calculate_distance(centroid, df_row)

        if lowest_distance == -1:
            lowest_distance = euclidean_distance
            closest_cluster = cluster_id

        elif euclidean_distance < lowest_distance:
            lowest_distance = euclidean_distance
            closest_cluster = cluster_id

    return closest_cluster

• Initializing the KMeans class from scikit-learn:

from sklearn.cluster import KMeans
kmeans_model = KMeans(n_clusters=2, random_state=1)

```

Resources

- [Sklearn implementation of K-Means clustering](#)
- [Implementing K-Means clustering from scratch](#)

Introduction to Decision Trees

Concepts

- Decision trees are a powerful and popular machine learning technique. The decision machine learning algorithm enables us to automatically construct a decision tree that tells us what outcomes we should predict in certain situations.
- Decision trees can pick up nonlinear interactions between variables in the data that linear regression cannot.
- A decision tree is made up of a series of nodes and branches. A node is where we split the data based on a variable, and a branch is one side of the split. The tree accumulates more levels as the data is split based on variables.
- A tree is n levels deep where n is one more than the number of nodes. The nodes at the bottom of the tree are called terminal nodes, or leaves.
- When splitting the data, you aren't splitting randomly; there is an objective to make a prediction on future data. To meet complete our objective, each leaf must have only one value for our target column.
- One type of algorithm used to construct decision trees is called the ID3 algorithm. There are other algorithms like CART that use different metrics for the split criterion.
- A metric used to determine how "together" different values are is called entropy, which refers to disorder. For example, if there were many values "mixed together", the entropy value would be high while a dataset consisting of one value would have low entropy.
- The formula for entropy is $\sum_{i=1}^c P(x_i) \log_b P(x_i)$ where i is a unique value in a single column, $P(x_i)$ is the probability of the value occurring in our data, b is the base of our logarithm, and c is the number of unique values in a single column.
- You can use information gain to tell which split will reduce entropy the most.
- The formula for information gain is the following:

$$IG(T, A) = Entropy(T) - \sum_{v \in A} \frac{|T_v|}{|T|} \cdot Entropy(T_v)$$

IG is information gain, T is our target variable, A is the variable you are splitting on, and T_v is the number of times a unique value is the target variable.

Syntax

- Converting a categorical variable to a numeric value:

```
col = pandas.Categorical(income["workclass"])
```
- Retrieving the numeric value of the categorical variable:

- col.codes
 - Using Python to calculate entropy:
- ```
def calc_entropy(column):
 """
 Calculate entropy given a pandas series, list, or numpy array.
 """
 counts = numpy.bincount(column)
 probabilities = counts / len(column)
 entropy = 0
 for prob in probabilities:
 if prob > 0:
 entropy += prob * math.log(prob, 2)
 return -entropy
```
- Using Python to calculate information gain:
- ```
def calc_information_gain(data, split_name, target_name):
    """
    Calculate information gain given a data set, column to split on, and
    target
    """
    original_entropy = calc_entropy(data[target_name])
    column = data[split_name]
    median = column.median()
    left_split = data[column <= median]
    right_split = data[column > median]
    to_subtract = 0
    for subset in [left_split, right_split]:
        prob = (subset.shape[0] / data.shape[0])
        to_subtract += prob * calc_entropy(subset[target_name])
    return original_entropy - to_subtract
```

Resources

- [Pandas categorical class documentation](#)
- [Information Theory](#)

- [Entropy](#)

Building a Decision Tree

Syntax

- Using Python to calculate entropy:

```
def calc_entropy(column):  
    """  
    Calculate entropy given a pandas series, list, or numpy array.  
    """  
  
    counts = numpy.bincount(column)  
    probabilities = counts / len(column)  
    entropy = 0  
  
    for prob in probabilities:  
        if prob > 0:  
            entropy += prob * math.log(prob, 2)  
  
    return -entropy
```

- Using Python to calculate information gain:

```
def calc_information_gain(data, split_name, target_name):  
    """  
    Calculate information gain given a data set, column to split on, and  
    target.  
    """  
  
    original_entropy = calc_entropy(data[target_name])  
  
    column = data[split_name]  
    median = column.median()  
  
    left_split = data[column <= median]  
    right_split = data[column > median]  
    to_subtract = 0  
  
    for subset in [left_split, right_split]:  
        prob = (subset.shape[0] / data.shape[0])  
        to_subtract += prob * calc_entropy(subset[target_name])  
  
    return original_entropy - to_subtract
```

- Finding the best column to split on:

```

def find_best_column(data, target_name, columns):
    """
    Find the best column to split on given a data set, target variable, and
    list of columns.

    """
    information_gains = []
    for col in columns:
        information_gain = calc_information_gain(data, col,
                                                "high_income")
        information_gains.append(information_gain)
    highest_gain_index =
    information_gains.index(max(information_gains))
    highest_gain = columns[highest_gain_index]
    return highest_gain

```

- Applying a function to a data frame:

```
df.apply(find_best_column, axis=0)
```

Concepts

- Pseudocode is a piece of plain-text outline of a piece of code explaining how the code works. Exploring the pseudocode is a good way to understand it before trying to code it.
- Pseudocode for the ID3 algorithm:

```

def id3(data, target, columns):
    1 Create a node for the tree
    2 If all values of the target attribute are 1, Return the node,
    with label = 1
    3 If all values of the target attribute are 0, Return the node,
    with label = 0
    4 Using information gain, find A, the column that splits the data
    best
    5 Find the median value in column A
    6 Split column A into values below or equal to the median (0),
    and values above the median (1)
    7 For each possible value (0 or 1), vi, of A,
    8 Add a new tree branch below Root that corresponds to rows of
    data where A = vi

```

```

    9 Let Examples(vi) be the subset of examples that have the value
vi for A

    10 Below this new branch add the subtree id3(data[A==vi], target,
columns)

    11 Return Root

```

- We can store the entire tree in a nested dictionary by representing the root node with a dictionary and branches with keys for the left and right node.
- Dictionary for a decision tree:

```

{
    "left": {
        "left": {
            "left": {
                "number": 4,
                "label": 0
            },
            "column": "age",
            "median": 22.5,
            "number": 3,
            "right": {
                "number": 5,
                "label": 1
            }
        },
        "column": "age",
        "median": 25.0,
        "number": 2,
        "right": {
            "number": 6,
            "label": 1
        }
    },
    "column": "age",
    "median": 37.5,
}

```

```
"number":1,  
"right":{  
    "left":{  
        "left":{  
            "number":9,  
            "label":0  
        },  
        "column":"age",  
        "median":47.5,  
        "number":8,  
        "right":{  
            "number":10,  
            "label":1  
        },  
        "column":"age",  
        "median":55.0,  
        "number":7,  
        "right":{  
            "number":11,  
            "label":0  
        }  
    },  
},  
}
```

Resources

- [Recursion](#)
- [ID3 Algorithm](#)

Applying Decision Trees

Concepts

- Scikit-learn includes the **DecisionTreeClassifier** class for classification problems, and **DecisionTreeRegressor** for regression problems.
- AUC (area under the curve) ranges from 0 to 1 and a measure of how accurate our predictions are, which makes it ideal for binary classification. The higher the AUC, the more accurate our predictions. AUC takes in two parameters:
 - **y_true**: true labels.
 - **y_score**: predicted labels.
- Trees overfit when they have too much depth and make overly complex rules that match the training data but aren't able to generalize well to new data. The deeper the tree is, the worse it typically performs on new data.
- Three ways to combat overfitting:
 - "Prune" the tree after we build it to remove unnecessary leaves.
 - Use ensambling to blend the predictions of many trees.
 - Restrict the depth of the tree while we're building it.
- We can restrict tree depth by adding a few parameters when we initialize the **DecisionTreeClassifier**:
 - **max_depth**: Globally restricts how deep the tree can go.
 - **min_samples_split**: The minimum number of rows a node should have before it can be split; if this is set to 2 then nodes with two rows won't be split and will become leaves instead.
 - **min_samples_leaf**: The minimum number of rows a leaf must have.
 - **min_weight_fraction_leaf**: The fraction of input rows a leaf must have.
 - **max_leaf_nodes**: The maximum number of total leaves; this will limit the number of leaf nodes as the tree is being built.
 - However, some parameters aren't compatible. For example, we can't use **max_depth** and **max_leaf_nodes** together.
- Underfitting occurs when our model is too simple to explain the relationships between the variables.

- High bias can cause underfitting while high variance can cause overfitting. We call this bias-variance tradeoff because decreasing one characteristic will usually increase the other. This is a limitation of all machine learning algorithms.
- The main advantages of using decision trees is that decision trees are:
 - Easy to interpret.
 - Relatively fast to fit and make predictions.
 - Able to handle multiple types of data.
 - Able to pick up nonlinearities in data and fairly accurate.
- The most powerful way to reduce decision tree overfitting is to create ensembles of trees. The random forest algorithm is a popular choice for doing this. In cases where prediction accuracy is the most important consideration, random forests usually perform better.

Syntax

- Instantiating the scikit-learn decision tree classifier:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=1)
```

- Fitting data using the decision tree classifier:

```
columns = ["age", "workclass", "education_num", "marital_status"]
clf.fit(income[columns], income["high_income"])
```

- Shuffling the rows of a data frame and re-indexing:

```
income = income.reindex(numpy.random.permutation(income.index))
```

- Calculating the area under curve (AUC) using scikit-learn:

```
from sklearn.metrics import roc_auc_score
error = roc_auc_score(test["high_income"], clf.predict(test[columns]))
```

Resources

- [DecisionTreeClassifier class documentation](#)
- [Area under the curve](#)
- [Documentation for roc_auc_score](#)

Introduction to Random Forests

Concepts

- The random forest algorithm is a powerful tool to reduce overfitting in decision trees.
- Random forest is an ensemble algorithm that combines the predictions of multiple decision trees to create a more accurate final prediction.
- There are many methods to get from the output of multiple models to a final vector of predictions. One method is majority voting. In majority voting, each decision tree classifier gets a "vote" and the most commonly voted value for each row "wins."
- To get ensemble predictions, use the **predict_proba** method on the classifiers to generate probabilities, take the mean of the probabilities for each row, and then round the result.
- The more dissimilar the models we use to construct an ensemble are, the stronger their combined predictions will be. For example, ensembling a decision tree and a logistic regression model will result in stronger predictions than ensembling two decision trees with similar parameters. However, ensembling similar models will result in a negligible boost in the accuracy of the model.
- Variation in the random forest will ensure each decision tree is constructed slightly differently and will make different predictions as a result. Bagging and random forest subsets are two main ways to introduce variation in a random forest.
- With bagging, we train each tree on a random sample of the data or "bag". When doing this, we perform sampling with replacement, which means that each row may appear in the "bag" multiple times. With random forest subsets, however, only a constrained set of features that is selected randomly will be used to introduce variation into the trees.
- **RandomForestClassifier** has an **n_estimators** parameter that allows you to indicate how many trees to build. While adding more trees usually improves accuracy, it also increases the overall time the model takes to train. The class also includes the **bootstrap** parameter which defaults to **True**. "Bootstrap aggregation" is another name for bagging.
- **RandomForestClassifier** has a similar interface to **DecisionTreeClassifier** and we can use the **fit()** and **predict()** methods to train and make predictions.
- The main strengths of a random forest are:
 - Very accurate predictions: Random forests achieve near state-of-the-art performance on many machine learning tasks.
 - Resistance to overfitting: Due to their construction, random forests are fairly resistant to overfitting.
- The main weaknesses of using a random forest are:

- They're difficult to interpret: Since we've averaging the results of many trees, it can be hard to figure out why a random forest is making predictions the way it is.
- They take longer to create: Making two trees takes twice as long as making one, making three trees takes three times as long, and so on.

Syntax

- Instantiating the RandomForestClassifier:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=5, random_state=1,
min_samples_leaf=2)
```

- Predicting the probability that a given class is correct for a row:

```
RandomForestClassifier.predict_proba()
```

- Computing the area under the curve:

```
print(roc_auc_score(test["high_income"], predictions))
```

- Introducing variation through bagging:

```
bag_proportion = .6
predictions = []
for i in range(tree_count):
    bag = train.sample(frac=bag_proportion, replace=True,
random_state=i)
    clf = DecisionTreeClassifier(random_state=1, min_samples_leaf=2)
    clf.fit(bag[columns], bag["high_income"])
    predictions.append(clf.predict_proba(test[columns])[:,1])
combined = numpy.sum(predictions, axis=0) / 10
rounded = numpy.round(combined)
```

Resources

- [Majority Voting](#)
- [RandomForestClassifier documentation](#)

Representing Neural Networks

Concepts

- Neural networks are usually represented as **graphs**. A graph is a data structure that consists of nodes (represented as circles) that are connected by edges (represented as lines between the nodes).
- Graphs are a highly flexible data structure; you can even represent a list of values as a graph. Graphs are often categorized by their properties, which act as constraints. You can read about the many different ways graphs can be categorized [on Wikipedia](#).
- Neural network models are represented as a **computational graph**. A computational graph uses nodes to describe variables and edges to describe how variables are combined.
- In a simple neural network:
 - each feature column in a data set is represented as an **input neuron**
 - each weight value is represented as an arrow from the feature column it multiples to the **output neuron**
- Inspired by biological neural networks, an **activation function** determines if the neuron *fires* or not. In a neural network model, the activation function transforms the weighted sum of the input values.

Syntax

- Generating data with specific properties using scikit learn:
 - `sklearn.datasets.make_regression()`
 - `sklearn.datasets.make_classification()`
 - `sklearn.datasets.make_moons()`
- Generating a regression data set with 3 features, 1000 observations, and a random seed of 1:

```
from sklearn.datasets import make_regression
data = make_regression(n_samples=1000, n_features=3, random_state=1)
```
- Returning a tuple of two NumPy objects that contain the generated data:

```
print(type(data))
tuple
```
- Retrieving the features of the generated data:

```
print(data[0])
array([[ 0.93514778, 1.81252782, 0.14010988],
```

```

[-3.06414136, 0.11537031, 0.31742716],
[-0.42914228, 1.20845633, 1.1157018 ],
...,
[-0.42109689, 1.01057371, 0.20722995],
[ 2.18697965, 0.44136444, -0.10015523],
[ 0.440956 , 0.32948997, -0.29257894])

```

- Retrieving the first row of data:

```

print(data[0][0])
array([ 0.93514778, 1.81252782, 0.14010988])

```

- Retrieving the labels of the data:

```

print(data[1])
array([ 2.55521349e+02, -2.24416730e+02, 1.77695808e+02,
       -1.78288470e+02, -6.31736749e+01, -5.52369226e+01,
       -2.33255554e+01, -8.81410996e+01, -1.75571964e+02,
       1.06048917e+01, 7.07568627e+01, 2.86371625e+02,
       ...,
       -7.38320267e+01, -2.38437890e+02, -1.23449719e+02,
       3.36130733e+01, -2.67823475e+02, 1.21279169e+00,
       -2.62440408e+02, 1.32486453e+02, -1.93414037e+02,
       -2.75702376e+01, -1.00678877e+01, 2.05169507e+02,
       -1.52978767e+02, 1.18361239e+01, -2.97505169e+02,
       2.40169605e+02, 7.33158364e+01, 2.18888903e+02,
       3.92751308e+01])

```

- Retrieving the first label of the data:

```

print(data[1][0])
255.52134901495128

```

- Creating a dataframe:

```

features = pd.DataFrame(data[0])

```

Resources

- [Graph Theory on Wikipedia](#)
- [Directed Acyclic Graph on Wikipedia](#)

- [Feedforward Neural Network on Wikipedia](#)
- [Calculus on Computational Graphs](#)
 - Explores how computational graphs can be used to organize derivatives.

Nonlinear Activation Functions

Concepts

TRIGONOMETRIC FUNCTIONS

- Trigonometry is short for triangle geometry and provides formulas, frameworks, and mental models for reasoning about triangles. Triangles are used extensively in theoretical and applied mathematics and build on mathematical work done over many centuries.
- A triangle is a [polygon](#) that has the following properties:
 - 3 edges
 - 3 vertices
 - angles between edges add up to 180 degrees
- Two main ways that triangles can be classified is by the internal angles or by the edge lengths.
- A trigonometric function is a function that inputs an angle value (usually represented as θ) and outputs some value. These functions compute ratios between the edge lengths.
- The three main trigonometric functions:

$$\begin{aligned}\bullet \sin(\theta) &= \frac{\text{opposite}}{\text{hypotenuse}} \\ \bullet \cos(\theta) &= \frac{\text{adjacent}}{\text{hypotenuse}} \\ \bullet \tan(\theta) &= \frac{\text{opposite}}{\text{adjacent}}\end{aligned}$$

- Right triangle terminology:
 - Hypotenuse describes the line that isn't touching the right angle.
 - Opposite refers to the line opposite the angle.
 - Adjacent refers to the line touching the angle that *isn't* the hypotenuse.
- In a neural network model, we're able massively expand the expressivity of the model by adding one or more layers, each with multiple linear combinations and nonlinear transformations.
- The three most commonly used activation functions in neural networks are:
 - the sigmoid function
 - the ReLU function
 - the tanh function

Syntax

- ReLU function:

```
def relu(values):  
    return np.maximum(x, 0)
```

- Tan function:

- [np.tan](#)

- Tanh function:

- [np.tanh](#)

Resources

- [Medium Post on Activation Functions](#)
- [Activation Function on Wikipedia](#)
- [Hyperbolic Tangent on Wikipedia](#)
- [Numpy documentation for tan](#)
- [Numpy documentation for tanh](#)

Hidden Layers

Concepts

- The intermediate layers are known as **hidden layers**, because they aren't directly represented in the input data or the output predictions. Instead, we can think of each hidden layer as intermediate features that are learned during the training process. This is actually very similar to how decision trees are structured. The branches and splits represent some intermediate features that are useful for making predictions and are analogous to the hidden layers in a neural network.
- Each of these hidden layers has its own set of weights and biases, which are discovered during the training process. In decision tree models, the intermediate features in the model represented something more concrete we can understand (feature ranges).
- The number of hidden layers and number of neurons in each hidden layer are hyperparameters that act as knobs for the model behavior.

Syntax

- Training a classification neural network:

```
from sklearn.neural_network import MLPClassifier  
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='logistic')
```

- Training a regression neural network:

```
from sklearn.neural_network import MLPRegressor  
mlp = MLPRegressor(hidden_layer_sizes=(1,), activation='relu')
```

- Specifying hidden layers as a tuple object:

```
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='relu')  
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='relu')
```

- Specifying the activation function:

```
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='relu')  
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='logistic')  
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='tanh')
```

- Generating data with nonlinearity:

```
from sklearn.datasets import make_moons  
data = make_moons()
```

Resources

- [Sklearn Hyperparameter Optimization](#)
- [Neural Network Hyperparameter Optimization](#)
- [Deep Learning on Wikipedia](#)

Advanced Python

Memory and Unicode

Concepts

- Hard drives are referred to as magnetic storage because they store data on magnetic strips.
- Magnetic strips can only contain a series of two values — ups and downs.
- Using binary, we can store strings in magnetic ups and downs. In binary, the only valid numbers are **0** and **1** so it's easy to store binary values on a hard drive.
- Binary is referred to as base two because there are only two possible digits — 0 and 1. We refer to digits as **0-9** as base 10 because there are 10 possible digits.
- We can add numbers in binary like we can in base 10.
- Computers store strings in binary. Strings are split into single characters and then converted to integers. Those integers are then converted to binary and stored.
- ASCII characters are the simple characters — upper and lowercase English letters, digits, and punctuation symbols. ASCII only supports 255 characters.
- As a result of ASCII's limitation, the tech community adopted a new standard called Unicode. Unicode assigns "code points" to characters.
- An encoding system converts code points to binary integers. The most common encoding system for Unicode is UTF-8.
- UTF-8 supports all Unicode characters and all ASCII characters.
- Bytes are similar to a string except that it contains encoded byte values.
- Hexadecimal is base 16. The valid digits in hexadecimal are **0-9** and **A-F**.
 - A: 10
 - B: 11
 - C: 12
 - D: 13
 - E: 14
 - F: 15

- Hexadecimal is used because it represents a byte efficiently. For example, the integer 255 in base 10 is represented by 11111111 in binary while it can be represented in hexadecimal using the digits FF.

Syntax

- Converting a number to binary:

```
int("100", 2)
```

- Encoding a string into bytes:

```
batman.encode("utf-8")
```

- Converting a string of length one to a Unicode code point:

```
ord("a")
```

- Converting integer to binary string:

```
bin(100)
```

- Decoding a bytes object into a string:

```
morgan_freeman.decode()
```

- Counting how many times a string occurs in a list:

```
from collections import Counter
fruits = ["apple", "apple", "banana", "orange"]
fruit_count = Counter(fruits)
```

Resources

- [Binary numbers](#)
- [Unicode](#)

Algorithms

Concepts

- An algorithm is a well-defined series of steps for performing a task. Algorithms usually have an input and output, and can either be simple or complicated.
- A linear search algorithm searches for a value in a list by reviewing each item in the list.
- When using more complex algorithms, it's important to make sure the code remains modular. Modular code consists of smaller chunks that we can reuse for other things.
- Abstraction is the idea that someone can use our code to perform in operation without having to worry about how it was written or implemented.
- When choosing from multiple algorithms, a programmer has to decide which algorithm best suits their needs. The most common factor to consider is time complexity. Time complexity is a measurement of how much time an algorithm takes with respect to its input size.
- An algorithm of constant time takes the same amount of time to complete, regardless of input size.
 - An example would be an algorithm to return the first element of a list.
- We refer to the time complexity of an algorithm that has to check n elements as linear time.
- Big-O Notation is the most commonly used notation when discussing time complexity. The following are most commonly used when discussing time complexity:
 - Constant time: $O(n)$
 - Linear time: $O(n)$
 - Quadratic: $O(n^2)$
 - Exponential: $O(2^n)$
 - Logarithmic: $O(\log(n))$
- An algorithm with lower-order time complexities are more efficient. In other words, an algorithm of constant time is more efficient than linear time algorithms. Similarly, an algorithm which has $O(n^2)$ complexity is more efficient than an algorithm with (O^3) complexity.

Syntax

- Example of a constant time algorithm:

```
def blastoff(message):  
    count = 10  
    for i in range(count):
```

```
    print(count - i)  
    print(message)
```

- Example of a linear time algorithm:

```
def is_empty_1(ls):  
    if length(ls) == 0:  
        return True  
  
    else:  
        return False
```

Resources

- [Time complexity](#)
- [Big-O notation](#)

Binary Search

Concepts

- Binary search helps us find an item efficiently if we know the list is ordered. Binary search works by checking the middle element of the list, comparing it to the item we're looking for and repeating the process.
- Pseudo-code is a powerful, easy-to-use tool that will help you train your ability to develop and visualize algorithms. Pseudo-code comments reflect the code we want to write and describes in high-level human language.
- Pseudo-code for binary search:

```
If the name comes before our guess  
    Adjust the bounds as needed  
Else if the name comes after our guess  
    Adjust the bounds as needed  
Else  
    Player found, so return first guess
```

- Binary search runs in logarithmic time, which we denote as $O(\log(n))$.

Syntax

- Implementing logic for binary search:

```
def player_age(name):  
    name = format_name(name)  
    first_guess_index = math.floor(length/2)  
    first_guess = format_name(nba[first_guess_index][0])  
    if name < first_guess:  
        return "earlier"  
    elif name > first_guess:  
        return "later"  
    else:  
        return "found"
```

- Implementing binary search until our answer is found:

```
def player_age(name):  
    name = format_name(name)
```

```
upper_bound = length - 1
lower_bound = 0
index = math.floor((lower_bound + upper_bound) / 2)
guess = format_name(nba[index][0])
while name != guess:
    if name < guess:
        upper_bound = index - 1
    else:
        lower_bound = index + 1
    index = math.floor((lower_bound + upper_bound) / 2)
    guess = format_name(nba[index][0])
return "found"
```

Resources

- [Binary search algorithm](#)

Data Structures

Concepts

- A data structure is a way of organizing data. Specifically, data structures are concerned with organization of data within a program. Lists and dictionaries are some examples of data structures, but there are many more.
- An array is a list that can contain items, which occupy a specific slot. Arrays cannot expand beyond their initial size. For example, if we create an array of size 10, it can only hold 10 elements.
- When we delete or add an element to the array, each element has to be shifted, which can make those operations quite costly.
- Dynamic arrays are a type of array that we can expand to fit as many elements as we'd like. Dynamic arrays are much more useful in data science than fixed-size arrays.
- A list is a one-dimensional array because they only go in one direction. One-dimensional arrays only have a length and no other dimension.
- A two-dimensional array has a height and a width and has two indexes.
- In data science, we call one-dimensional arrays vectors and two-dimensional arrays matrices.
- The time complexity of a two-dimensional array traversal is $O(m*n)$ where **m** is the height of our array, and **n** is the width.
- A hash table is a data structure that stores data based on key-value pairs. A dictionary is the most common form of a hash table.
- A hash table is a large array; however, a hash table is a clever construct that takes advantages of accessing elements by index and converts the keys to indexes using a hash function.
- A hash function accepts a key as input and converts it to an integer index.
- Accessing and storing data in hash tables is very quick; however, using a hash table uses a lot of memory.

Syntax

- Inserting an element into a list at the given index:
`players.insert(0, "Jeff Adrien")`
- Retrieving an element in a 2D array:
`arr[2][3]`
- Retrieving the value of a key within a dictionary:
`city_population["Boston"]`

Resources

- [Hash Table](#)

Recursion and Advanced Data Structures

Concepts

- Recursion is the method of repeating code without using loops. An example of recursion would be the factorial function in mathematics.
 - We denote a factorial using the ! sign. For example $n!$ denotes multiplying n by all the positive integers less than n . However, $0!$ is defined as 1.
 - For example: $5! = 5 * 4 * 3 * 2 * 1$.
- A linked list is made up of many nodes. In a singly linked list, each node contains its data as well as the next node.
- A linked list is a type a recursive data structure since each node contains the data and then points to another linked list.
- An advantage of using linked lists is that you need to modify very few nodes when inserting or deleting because the update only requires a constant amount of changes.

Syntax

- Using recursion to return the n^{th} Fibonacci number:

```
def fib(n):  
    if n == 0 or n == 1:  
        return 1  
    return fib(n - 1) + fib(n - 2)
```

- Getting the length of the linked list using iteration:

```
def length_iterative(ls):  
    count = 0  
    while not ls.is_empty():  
        count = count + 1  
        ls = ls.tail()  
    return count
```

- Using recursion to find the length of a linked list:

```
def length_recursive(ls):  
    if ls.is_empty():
```

```
    return 0

return 1 + length_recursive(ls.tail())
```

Resources

- [Recursion](#)
- [Linked Lists](#)

Object-Oriented Programming

Concepts

- In object-oriented programming, everything is an object. Classes and instances are known as objects and they're a fundamental part of object-oriented programming.
- The special `__init__` function runs whenever a class is instantiated. The `__init__` function can take in parameters, but `self` is always the first one. `Self` is just a reference to the instance of the class and is automatically passed in when you instantiate an instance of the class.
- Inheritance enables you to organize classes in a tree-like hierarchy. Inheriting from a class means that the new class can exhibit behavior of the inherited class but also define its own additional behavior.
- Class methods act on an entire class rather than a particular instance of a class. We often use them as utility functions.
- Overloading is a technique used to modify a inherited class to ensure not all behavior is inherited. Overloading methods gives access to powerful functions without having to implement tedious logic.

Syntax

- Creating a class:

```
class Class:  
  
    def __init__(self, team_name):  
        self.team_name = team_name
```

- Creating an instance of a class:

```
spurs = Team("San Antonio Spurs")
```

- Defining a class method:

```
@classmethod  
  
def older_team(self, team1, team2):  
    return "Not yet implemented"
```

Resources

- [Object-oriented Programming](#)
- [Documentation for classmethod](#)

Exception Handling

Concepts

- Errors can be quite useful to us because they tell us what went wrong with our code.
- Exception handling comes into play when we want to handle errors gracefully so our program doesn't crash.
- An exception is a broad characterization of what can go wrong with a program. Exceptions occur during the execution of a program whereas syntax errors will cause your code not to run at all.
- In a **try-except** block, Python will attempt to execute the try section of the statement. If Python raises an exception, the code in the **except** statement will execute.
- While you have the ability to catch any exception without specifying a particular error in the **except:** section, not specifying an error is sometimes dangerous as you won't be able to execute exception-specific logic.

Syntax

- Handling an exception using a try-except block:

```
try:  
    impossible_value = int("Not an integer")  
except ValueError:  
    print("Cannot convert string to integer")
```

- Catching multiple types of exceptions:

```
try:  
    f = open("data.txt", "r")  
    s = f.readline()  
    i = float(s)  
except ValueError:  
    print("Cannot convert data to floating point value")  
except IOError:  
    print("Could not read file")
```

Resources

- [Why a try-except block is useful in Python](#)
- [Errors and Exceptions](#)

Lambda Functions

Concepts

- We can slice or select fragments of string objects using the index of the string. Slicing a string extracts a chunk or substring from the original string. You can either slice by specifying a single number to select a specific character or a range of values to select multiple characters.
- When extracting the first four characters in a string, you would specify the starting index as **0** but the ending index as **4**. Python uses the ending index to stop iterating and doesn't return the character at the ending index.
- Python's flexibility with ranges allow you to omit the starting or ending index to extract strings.
- Lambda functions, or anonymous functions, are used when you want to run a function once and don't need to save it for reuse.

Syntax

- Accessing the first character in a string:
`s = "string"[0]`
- Accessing the first four characters of a string:
`stri = "string"[0:4]`
- Omitting the ending index to access the last five characters of a string:
`sword = "password"[3:]`
- Skipping indexes in a slice:
`hlo = "hello world":5:2`
- Stepping backwards in a string:
`olleh = "hello world":4:-1`
- Applying a function to a list of items:
`list(map(func, my_list))`
- Filtering through a list of items using a function:

```
def is_palindrome(my_string):
    return my_string == my_string[::-1]
palindrome_passwords = list(filter(is_palindrome, passwords))
```
- Using a lambda function to filter a list:
`numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

```
evens = list(filter(lambda x : x % 2 == 0, numbers))
```

Resources

- [Documentation for map, filter, and reduce](#)
- [Anonymous function](#)

Introduction to Computer Architecture

Concepts

- A computer must be able to do four things:
 - Take input.
 - Produce output.
 - Store data.
 - Perform computation.
- Keyboards and mice are examples of input devices while screens, monitors, and speakers are examples of output devices.
- Computers can store data either in random-access memory or in disk storage.
- We can think of data as occupying "slots" in large, linearly-arranged pieces of hardware. We refer to each "slot" as one byte.
- Small integers and characters can be stored in one byte while larger date types, like strings, require multiple bytes.
- Low-level languages often interact with portions of memory explicitly; therefore, data types have very predictable memory usage. On the other hand, high-level languages empower you to express logic quickly and easily.
- We use a base-10 number system, which means each digit corresponds to a power of 10.

Number	Base 10 Expression
008	$0 * (10^2) + 0 * (10^1) + 8 * (10^0)$
091	$0 * (10^2) + 9 * (10^1) + 1 * (10^0)$
453	$4 * (10^2) + 5 * (10^1) + 3 * (10^0)$

- Binary is a number system where every digit is either **0** or **1**. This is sometimes referred to as a base-2 number system.
- In binary, the conversion is the exact same except each digit corresponds to a power of 2.

Binary	Base 2 Expression	Base 10 Value

001	$0 * (2^2) + 0 * (2^1) + 1 * (10^0)$	1
010	$0 * (2^2) + 1 * (2^1) + 0 * (2^0)$	2
100	$1 * (2^2) + 0 * (2^1) + 0 * (2^0)$	4
101	$1 * (2^2) + 0 * (2^1) + 1 * (2^0)$	5

- Characters are stored as binary and each character has its own binary number.
- A central processing unit (CPU) is a chip in the computer that can perform any computation.
- When a computer executes a program, the program is stored in memory as a sequence of machine instructions. The CPU reads the program like a book keeping a program counter and it points to the next instruction a CPU should execute.
- Once the CPU executes an instruction, the program counter moves to the instruction that's adjacent in memory. However, control flow statements can alter how the program counter traverses' instructions in memory. Functions can also change the order of statement execution.
- A processing unit that executes one instruction at a time is called a core. A multi-core processor can execute more than one set of instructions at a time.

Syntax

- Accessing the address for a variable's location in storage:

```
my_int = 5
```

```
id(my_int)
```

- Checking how many of bytes of memory a variable occupies:

```
my_int = 200
```

```
size_of_my_int = sys.getsizeof(my_int)
```

- Returning the current processor time in seconds:

```
import time
```

```
time.clock()
```

Resources

- [Binary numbers](#)
- [Central Processing Unit](#)

Parallel Processing

Concepts

- A multi-core central processing unit (CPU) has the ability to run multiple instructions simultaneously.
- Parallel processing is the technique of taking advantage of modern multi-core CPUs to run multiple programs at once.
- Immutable variables, such as integers, cannot be changed. Mutable variables, such as dictionaries and lists, are mutable. Mutable variables are especially useful in parallel processing because you often want to share and edit the same data between different processes.
- Multithreading refers to the technique of running multiple processes at once.
- A thread refers to any one path of execution in a program.
- Blocking refers to waiting for a condition to execute. For example, the main thread will wait until the other thread has finished executing.
- A program is deterministic if we can precisely predict its output for a particular input. On the other hand, a program is nondeterministic if we can't reliably predict the outcome of running a piece of code.
- By nature, multithreading is nondeterministic; however, we can use **threading.Lock** to combat this. A lock is a way of conditionally blocking the execution of some threads. At any given time, a lock is either available or acquired.
- Atomic operations finish executing before any operations can occur. On the other hand, nonatomic operations can run simultaneously while other operations are occurring.

Syntax

- Creating and starting new threads:

```
import threading  
  
counter = Counter()  
  
count_thread = threading.Thread(target=count_up_100000, args=[counter])  
count_thread.start()
```

- Joining a thread with the main thread:

```
thread.join()
```

- Using `threading.Lock`:

```
def conduct_trial():  
    counter = Counter()
```

```
lock = threading.Lock()

count_thread = threading.Thread(target=count_up_100000,
args=[counter, lock])

count_thread.start()

intermediate_value = counter.get_count()

count_thread.join()

return intermediate_value
```

Resources

- [Documentation for threading library](#)
- [Multithreading](#)

Advanced Topics

Getting Started with Kaggle

Concepts

- Kaggle is a site where people create algorithms to compete against machine learning practitioners around the world.
- Each Kaggle competition has two key data files to work with — a training set and a testing set. The training set contains data we can use to train our model whereas the testing set contains all of the same feature columns, but is missing the target value column.
- Along with the data files, Kaggle competitions include a data dictionary, which explains the various columns that make up the data set.
- Acquiring domain knowledge is thinking about the topic you are predicting, and it's one of the most important determinants for success in machine learning.
- Logistic regression is often the first model you want to train when performing classification.
- The scikit-learn library has many tools that make performing machine learning easier. The scikit-learn workflow consists of four main steps:
 - Instantiate (or create) the specific machine learning model you want to use.
 - Fit the model to the training data.
 - Use the model to make predictions.
 - Evaluate the accuracy of the predictions.
- Before submitting, you'll need to create a submission file. Each Kaggle competition can have slightly different requirements for the submission file.
- You can start your submission to Kaggle by clicking the blue 'Submit Predictions' button on the competitions page.

Syntax

- Cutting bin values into discrete intervals:

```
pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
```
- Splitting our data using scikit-learn:

```
from sklearn.model_selection
```

```
import train_test_split train_X, test_X, train_y, test_y =  
train_test_split( all_X, all_y, test_size=0.2,random_state=0)
```

- Calculating accuracy using scikit-learn:

```
from sklearn.metrics import accuracy_score  
  
accuracy = accuracy_score(test_y, predictions)
```

- Calculating cross validation accuracy score using scikit-learn:

```
from sklearn.model_selection import cross_val_score  
  
cross_val_score(estimator, X, y, cv=None)
```

Resources

- [Kaggle](#)
- [Documentation for pandas.cut](#)

Feature Preparation, Selection and Engineering

Concepts

- We can focus on two main areas to boost the accuracy of our predictions:
 - Improving the features we train our model on.
 - Improving the model itself.
- Feature selection involves selecting features that are incorporated into the model. Feature selection is important because it helps to exclude features which are not good predictors or features that are closely related to each other.
- A model that is overfitting fits the training data too closely and is unlikely to predict well on unseen data.
- A model that is well-fit captures the underlying pattern in the data without the detailed noise found in the training set. The key to creating a well-fit model is to select the right balance of features.
- Rescaling transforms features by scaling each feature to a given range.
- Feature engineering is the practice of creating new features from existing data. Feature engineering results in a lot of accuracy boosts.
- A common technique to engineer a feature is called binning. Binning is when you take a continuous feature and separate it out into several ranges to create a categorical feature.
- Collinearity occurs where more than one feature contains data that are similar. If you have some columns that are collinear, you may get great results on your test data set, but then the model performs worse on unseen data.

Syntax

- Returning the descriptive statistics of a data frame:

```
df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
                     'numeric': [1, 2, 3],
                     'categorical': pd.Categorical(['d', 'e', 'f'])})
df.describe(include='all')
```

- Rescaling data:

- ```

from sklearn.preprocessing import minmax_scale
columns = ["column one", "column two"]
data[columns] = minmax_scale(data[columns])

```
- Returning a NumPy array of coefficients from a LogisticRegression model:

```

lr = LogisticRegression()
lr.fit(train_X,train_y)
coefficients = lr.coef_

```

  - Creating a horizontal bar plot:

```

ordered_feature_importance = feature_importance.abs().sort_values()
ordered_feature_importance.plot.barh()
plt.show()

```

  - Creating bins:

```

pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)

```

  - Extracting groups from a match of a regular expression:

```

s = Series(['a1', 'b2', 'c3'])
s.str.extract(r'([ab])(\d)')

```

  - Producing a correlation matrix:

```

import seaborn as sns
correlations = train.corr()
sns.heatmap(correlations)
plt.show()

```

  - Using recursive feature elimination for feature selection:

```

from sklearn.feature_selection import RFECV
lr = LogisticRegression()
selector = RFECV(lr, cv=10)
selector.fit(all_X, all_y)
optimized_columns = all_X.columns[selector.support_]

```

## Resources

- [Documentation for cross validation score](#)
- [Documentation for recursive feature elimination with cross validation](#)

- [Mastering feature engineering](#)

# Model Selection and Tuning

## Concepts

- Model selection is the process of selecting the algorithm which gives the best predictions for your data. Each algorithm has different strengths and weaknesses and we need to select the algorithm that works best with our specific set of data.
- The k-nearest neighbors algorithm finds the observations in our training set most similar to the observation in our test set and uses the average outcome of those 'neighbor' observations to make a prediction. The 'k' is the number of neighbor observations used to make the prediction.
- Hyperparameter optimization, or hyperparameter tuning, is the process of varying the parameters of each model to optimize accuracy.
  - Grid search is one method of hyperparameter optimization.
  - Grid search trains a number of models across a "grid" of values and then searches for the model that gives the highest accuracy.
- Random forests is a specific type of decision tree algorithm. Decision tree algorithms attempt to build the most efficient decision tree based on the training data.

## Syntax

- Instantiating a KNeighborsClassifier:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

- Using the range class to iterate through values:

```
for k in range(1,8,2):
 print(k)
```

- Performing grid search using scikit-learn:

```
from sklearn.model_selection import GridSearchCV
knn = KNeighborsClassifier()
hyperparameters = {
 "n_neighbors": range(1,50,2)
}
grid = GridSearchCV(knn, param_grid=hyperparameters, cv=10)
grid.fit(all_X, all_y)
print(grid.best_params_)
```

```
print(grid.best_score_)

• Fitting and making predictions using the RandomForestClassifier:

from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=1)
clf.fit(train_X,train_y)
predictions = clf.predict(test_X)
```

## ***Resources***

- [Cross Validation and Grid Search](#)
- [Hyperparameter optimization](#)

# Naive Bayes for Sentiment Analysis

## Concepts

- The Naive Bayes classifier figures out how likely data attributes are associated with a certain class.
- The classifier is based on Bayes' theorem, which is

$$P(A | B) = \frac{P(B | A)(P(A))}{P(B)}, P(B) \neq 0$$

where:

- A and B are events.
- $P(A | B)$  is a conditional probability. Specifically, the likelihood of event A occurring given the B is true.
- $P(B | A)$  is also a conditional probability. Specifically, the likelihood of event B occurring given the A is true.
- $P(A)$  and  $P(B)$  are the probabilities of observing A and B independently of each other.
- Bayes' Theorem describes the probability of an event based on prior knowledge of conditions that might be related to the event.
- Naive Bayes extends Bayes' theorem to handle the case of multiple data points by assuming each data point is independent.
- The formula for the classifier is the following

$$P(y | x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1, \dots, x_n)}$$

- To find the "right classification", we find out which classification ( $P(y | x_1, \dots, x_n)$ ) has the highest probability.

## Resources

- [Bayes' theorem](#)
- [Probability theory](#)

# Introduction to Natural Language Processing

## Concepts

- Natural language processing is the study of enabling computers to understand human languages. Natural language processing including applications such as scoring essays, inferring grammatical rules, and determine emotions associated with text.
- A bag of words model represents each piece as a numerical vector.
- Tokenization is the process of breaking up pieces of text into individual words.
- Stop words don't tell anything about the document content and don't add anything relevant. Examples of stop words are 'the', 'an', 'and', 'a', and there are many others.
- To calculate prediction error, we can use the mean squared error. The mean square error penalized errors further away because the errors are squared. We often use the MSE because we'd like all our predictions to be relatively close to the actual values.

## Syntax

- Splitting an array into random train and test subsets:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(counts,
submissions["upvotes"], test_size=0.2, random_state=1)
```

- Initializing the LinearRegression class:

```
from sklearn.linear_model import LinearRegression

clf = LinearRegression()
```

- Predicting using a LinearRegression model:

```
predictions = clf.predict(X_test)
```

## Resources

- [Natural Language Processing](#)
- [Bag-of-words model](#)

# Spark and Map-Reduce

## Introduction to Spark

### Syntax

- Loading a data set into a resilient distributed data set (RDD) :

```
raw_data = sc.textFile("daily_show.tsv")
```

- Printing out the first five elements of the RDD:

```
raw_data.take(5)
```

- Mapping a function to every element in the RDD:

```
daily_show = raw_data.map(lambda line: line.split('\t'))
```

- Merging the values for each similar key:

```
tally = daily_show.map(lambda x: (x[0], 1)).reduceByKey(lambda x,y: x+y)
```

- Retuning the count of an RDD:

```
tally.take(tally.count())
```

- Filtering an RDD:

```
rdd.filter(lambda x: x % 2 == 0)
```

### Concepts

- MapReduce is a paradigm that efficiently distributes calculations over hundreds or thousands of computers to calculate the result in parallel.
- Hadoop is an open source project that is the dominant processing toolkit for big data. There are pros and cons to Hadoop including:
  - Hadoop made it possible to analyze large data sets; however, it had to rely on disk storage for computation rather than memory.
  - Hadoop wasn't a great solution for calculations that require multiple passes over the same data or require many intermediate steps.
  - Hadoop had suboptimal support for SQL and machine learning implementations.
- To improve speeds of many data processing workloads, UC Berkeley AMP lab developed Spark.
- Spark's main core data structure is a RDD.

- An RDD is a data set that's distributed across the RAM of a cluster of many machines.
  - An RDD is essentially a collection of elements we can use to hold objects.
- PySpark is a Python API that allows us to interface with RDDs in Python.
- Spark's RDD implementation lets us evaluate code "lazily", which means we can postpone running a calculation until absolutely necessary.
- Calculations in Spark are a series of steps we can chain together and run in succession to form a pipeline. Pipelining is the key idea to understand when working with Spark.
- The main types of methods in Spark are:
  - Transformations: **map()**, **reduceByKey()**.
  - Actions: **take()**, **reduce()**, **saveAsTextFile()**, **collect()**.
- RDD objects are immutable, which means that we can't change their values once we created them.

## ***Resources***

- [MapReduce](#)
- [PySpark documentation](#)

# Transformations and Actions

## Concepts

- **yield** is a Python technique that allows the interpreter to generate data on the fly and pull it when necessary as opposed to storing to the memory immediately.
- Spark takes advantage of 'yield' to improve the speed of computations.
- **flatMap()** is useful when you want to generate a sequence of values from an RDD.

## Syntax

- Generate a sequence of values from an RDD:

```
def hamlet_speaks(line):
 id = line[0]
 speaketh = False

 if "HAMLET" in line:
 speaketh = True

 if speaketh:
 yield id, "hamlet speaketh!"

hamlet_spoken = split_hamlet.flatMap(lambda x: hamlet_speaks(x))
```

- Return the number of elements in an RDD:

```
hamlet_spoken_lines.count()
```

- Return a list representation of an RDD:

```
hamlet_spoken_lines.collect()
```

## Resources

- [Python yield](#)
- [Difference between map and flatMap in Apache Spark](#)

# Spark DataFrames

## Concepts

- The Spark DataFrame:
  - Is a feature that allows you to create and work with dataframe objects.
  - Combines the scale and speed of Spark with the familiar query, filter, and analysis capabilities of pandas.
  - Allows you to modify and reuse existing pandas code to much larger data sets.
  - Has better support for various data formats.
  - Is immutable.
- The Spark SQL class gives Spark more information about that data structure you're using and the computation you want to perform.
- When you read data into the SQLContext object, Spark:
- Instantiates a Spark DataFrame object.
- Infers the schema from the data and associates it with the DataFrame.
- Reads in the data and distributes it across clusters (if multiple clusters are available).
- Returns the DataFrame object.
- Because the Spark dataframe is influenced by the pandas DataFrame, it has some of the same method implementations such as `agg()`, `join()`, `sort()`, and `where()`.
- To handle the shortcomings of the Spark library, we can convert a Spark DataFrame to a pandas DataFrame.

## Syntax

- Instantiating the SQLContext class:

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```
- Reading in JSON data:

```
df = sqlCtx.read.json("census_2010.json")
```
- Using the show() method to print the first five rows:

```
df.show(5)
```
- Using the head method and a for loop to return the first five rows of the DataFrame:

```
first_five = df.head(5) for r in first_five: print(r.age)
```

- Using the show method to display columns:

```
df.select('age', 'males', 'females')
```

- Converting a Spark DataFrame to a pandas DataFrame:

```
pandas_df = df.toPandas()
```

## Resources

- [Spark programming guide](#)
- [Pandas and Spark DataFrames](#)

# Spark SQL

## Concepts

- Spark maintains a virtual database within a SQLContext object. This makes it possible to use Spark's SQL interface to query and interact with the data.
- Spark's uses a type of SQL that is identical to SQLite to query a table.
- Spark SQL allows you to run join queries across data from multiple file types.
- Spark SQL supports the functions and operators from SQLite. Supported functions/operators are as follows:
- COUNT()
- AVG()
- SUM()
- AND
- OR

## Syntax

- Registering an RDD as a temporary table:

```
from pyspark.sql import SQLContext

sqlCtx = SQLContext(sc)

df = sqlCtx.read.json("census_2010.json")

df.registerTempTable('census2010')
```

- Returning a list of tables:

```
tables = sqlCtx.tableNames()
```

- Querying a table in Spark:

```
sqlCtx.sql('select age from census2010').show()
```

- Calculating summary statistics for a DataFrame:

```
query = 'select males,females from census2010'
sqlCtx.sql(query).describe().show()
```

## Resources

- [Spark SQL](#)
- [Purpose of Spark SQL](#)

# Production Databases

## Intro to Postgres

### **Concepts**

- Data engineers needs to have the skills to build a data pipeline that connects all the pieces of the data ecosystem together and keep it running.
- The parts of a data pipeline are the following:
  - Collecting
  - Short-Term Storage
  - Processing
  - Long-Term Storage
  - Presenting
- Relational databases are the most common storage used for web content, large business storage, and for data platforms.
- Postgres (or PostgreSQL) is one of the biggest open source relational databases.
- Postgres is one of the best options for data analysts.
- Postgres is a more robust engine that is implemented as a server. Postgres can also handle multiple connections and can implement more advanced querying features.
- **psycopg2** is an open source library that implements the Postgres protocol to connect to our Postgres server.
- SQL transactions prevent loss of data by ensuring all queries in a transaction block are executed at the same time. If any transactions fail then the whole group fails, and no changes are made to the database.
- A new transaction will automatically be created when we open a Connection in **psycopg2**.
- When a commit is called, the PostgreSQL engine will run all the queries at once. Not calling a commit or rollback will cause the transaction to stay in a pending state, and the changes will not be made.

### **Syntax**

- Connecting to a database using psycopg2:

```
import psycopg2
```

```
conn = psycopg2.connect("dbname=postgres user=postgres")
```

- Creating a table:

```
CREATE TABLE tableName (
 column1 dataType1 PRIMARY KEY,
 column2 dataType2,
 column3 dataType3,
 ...
);
```

- Dropping a table from a database:

```
DROP TABLE tableName
```

OR

```
DROP TABLE IF EXISTS tableName
```

- Inserting values using psycopg2:

```
import psycopg2
conn = psycopg2.connect("dbname=dq user=dq")
cur = conn.cursor()

insert_query = "INSERT INTO users VALUES {}".format("(10, 'hello@dataquest.io', 'Some Name', '123 Fake St.')")

cur.execute(insert_query)
conn.commit()
```

OR

```
import psycopg2
conn = psycopg2.connect("dbname=dq user=dq")
cur = conn.cursor()

cur.execute("INSERT INTO users VALUES (%s, %s, %s, %s)", (10, 'hello@dataquest.io', 'Some Name', '123 Fake St.'))
conn.commit()
```

- Deleting data from a table:

```
DELETE from tableName
```

- Loading in a file using psycopg2:

```
conn = psycopg2.connect('dbname=postgres user=postgres')
cur = conn.cursor()
```

```
sample_file.csv has a header row.

with open('sample_file.csv', 'r') as f:
 # Skip the header row.
 next(f)
 cur.copy_from(f, 'sample_table', sep=',')
```

- Returning the first result:

```
cur.fetchone()
```

- Returning each row in a table:

```
cur.fetchall()
```

## ***Resources***

- [Comparison of Relational Databases](#)
- [Psycopg2 documentation](#)

# Creating Tables

## Concepts

- Using data types will save space on the database server which provides exponentially faster read and writes. In addition, having proper data types will ensure that any errors in the data will be caught and the data can be queried the way you expect.
- The description property outputs column information from the table. Within the column information, you will find the column data type, name, and other meta information.
- Numeric data types that Postgres supports:

| Name             | Storage Size | Description                     | Range                                                                                    |
|------------------|--------------|---------------------------------|------------------------------------------------------------------------------------------|
| smallint         | 2 bytes      | small-range integer             | -32768 to +32767                                                                         |
| integer          | 4 bytes      | typical choice for integer      | -2147483648 to +2147483647                                                               |
| bigint           | 8 bytes      | large-range integer             | -9223372036854775808 to 9223372036854775807                                              |
| decimal          | variable     | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| numeric          | variable     | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| real             | 4 bytes      | variable-precision, inexact     | 6 decimal digits precision                                                               |
| double precision | 8 bytes      | variable-precision, inexact     | 15 decimal digits precision                                                              |
| serial           | 4 bytes      | autoincrementing integer        | 1 to 2147483647                                                                          |

|           |         |                                |                          |
|-----------|---------|--------------------------------|--------------------------|
| bigserial | 8 bytes | large autoincrementing integer | 1 to 9223372036854775807 |
|-----------|---------|--------------------------------|--------------------------|

- **REAL, DOUBLE PRECISION, DECIMAL, and NUMERIC** can store float-like numbers such as: 1.23123, 8973.1235, and 100.00.
- The difference between **REAL** and **DOUBLE PRECISION** is that the **REAL** type is up to 4 bytes, whereas the **DOUBLE PRECISION** type is up to 8 bytes.
- The **DECIMAL** type works as follows: The precision value which is the maximum amount of digits before and/or after the decimal point, whereas the scale is the maximum amount of digits after the decimal number where scale must be less than or equal to precision.
- The **NUMERIC** and **DECIMAL** types are equivalent in Postgres.
- Corrupted data is unexpected data that has been entered into the data set.
- String-like data types that Postgres supports:

| Name                             | Description                |
|----------------------------------|----------------------------|
| character varying(n), varchar(n) | variable-length with limit |
| character(n), char(n)            | fixed-length, blank padded |
| text                             | variable unlimited length  |

- The difference between **CHAR(N)** and **VARCHAR(N)** is that **CHAR(N)** will pad any empty space of a character with whitespace characters while **VARCHAR(N)** does not.
- The **BOOLEAN** type can accept any of the following:
- The "true" state: True, 't' 'true', 'y', 'yes', 'no', '1'.
- The "false" state: False, 'f', 'false', 'n', 'no', 'off', '0'.
- Date/Time data types that Postgres supports:

| Name                                    | Storage Size | Description                       | Low Value | High Value | Resolution                |
|-----------------------------------------|--------------|-----------------------------------|-----------|------------|---------------------------|
| timestamp [ (p) ] [ without time zone ] | 8 bytes      | both date and time (no time zone) | 4713 BC   | 294276 AD  | 1 microsecond / 14 digits |

|                                    |          |                                    |                  |                 |                           |
|------------------------------------|----------|------------------------------------|------------------|-----------------|---------------------------|
| timestamp [ (p) ] with time zone   | 8 bytes  | both date and time, with time zone | 4713 BC          | 294276 AD       | 1 microsecond / 14 digits |
| date                               | 4 bytes  | date (no time of day)              | 4713 BC          | 5874897 AD      | 1 day                     |
| time [ (p) ] [ without time zone ] | 8 bytes  | time of day (no date)              | 00:00:00         | 24:00:00        | 1 microsecond / 14 digits |
| time [ (p) ] with time zone        | 12 bytes | times of day only, with time zone  | 00:00:00+1459    | 24:00:00-1459   | 1 microsecond / 14 digits |
| interval [ fields ] [ (p) ]        | 16 bytes | time interval                      | -178000000 years | 178000000 years | 1 microsecond / 14 digits |

## Syntax

- Returning the description of a table:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")
cur = conn.cursor()
cur.execute('SELECT * FROM users LIMIT 0')
print(cur.description)
```

## Resources

- [The cursor class](#)
- [PostgreSQL data types](#)

# Managing Created Tables

## Concepts

- It's always a good idea to remove redundant columns as tables taking unnecessary disk space can cause queries to be slower.
- You cannot change the data type of a column to another that it is not compatible with.
- Adding a column will result in that column containing null entries.
- We can concatenate strings using `||`. `||` is similar to `+` in Python, and it is part of Postgres' built-in functions that can be used to create new entries from a combination or already declared columns.

## Syntax

- Renaming a table:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

cur.execute('ALTER TABLE old_table RENAME TO new_table')

conn.commit()
```

- Dropping a column:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

cur.execute('ALTER TABLE example_table DROP COLUMN redundant_column')

conn.commit()
```

- Mapping a value to the data type:

```
import psycopg2

from psycopg2 import extensions as ext

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

cur.execute('SELECT id FROM ign_reviews')

id_column = cur.description[0]

print(id_column.type_code in ext.INTEGER.values) ## prints False
```

- Changing a columns data type:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

Assume `other_type` is of type `INTEGER`.

cur.execute('ALTER TABLE example_table ALTER COLUMN other_type TYPE
BIGINT')

conn.commit()
```

- Renaming a column:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

cur.execute('ALTER TABLE example_table RENAME COLUMN bad_name TO
relevant_name')

conn.commit()
```

- Adding a column:

```
import psycopg2

conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

cur.execute('ALTER TABLE example_table ADD COLUMN id INTEGER PRIMARY
KEY')

conn.commit()
```

- Adding a column with a default value:

```
ALTER TABLE ign_reviews ADD COLUMN release_date DATE DEFAULT 01-01-1991
```

- Adding or updating entries in a column:

```
UPDATE ign_reviews SET editors_choice = 'F' WHERE id > 5000
```

- Creating a date type from text:

```
to_date('01-01-1991', 'DD-MM-YYYY')
```

## Resources

- [Data Type Formatting Functions](#)
- [Documentation for Alter Table](#)

# Loading and Extracting Data with Tables

## Concepts

- A prepared statement helps performance by signaling what table and how many values will be altered so that the Postgres engine can be ready for them.
- The **mogrify()** statement returns a **utf-8** character encoded string.
- The difference between **copy\_from()** and **copy\_expert()** is that **copy\_expert** contains a parameter for the file descriptor.
- As table size increases, it requires even more memory and disk space to load and store the files.

## Syntax

- Using a prepared statement to insert values:

```
conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

with open('ign.csv', 'r') as f:

 next(f)

 reader = csv.reader(f)

 for row in reader:

 cur.execute(

 "INSERT INTO ign_reviews VALUES (%s, %s, %s, %s, %s,
 %s, %s, %s, %s)", row

)

 conn.commit()
```

- Returning the exact string that would be sent to the database:

```
cur.mogrify("INSERT INTO test (num, data) VALUES (%s, %s)", (42,
'bar'))

"INSERT INTO test (num, data) VALUES (42, E'bar')"
```

- Copying data between a file and a table:

```
COPY example_table FROM STDIN
```

- Specifying the type of file when copying using copy expert:

```
conn = psycopg2.connect("dbname=dq user=dq")

cur = conn.cursor()

with open('ign.csv', 'r') as f:
```

```
 cur.copy_expert('COPY ign_reviews FROM STDIN WITH CSV HEADER', f)
 conn.commit()
```

- Extracting a Postgres table to any Python file object:

```
cur = conn.cursor()

with open('example.txt', 'w') as f:
 cur.copy_expert('COPY ign_reviews to STDOUT', f)
```

## *Resources*

- [Formatted SQL with Psycopg's mogrify](#)
- [PostGreSQL COPY method](#)

# User and Database Management

## Concepts

- A superuser is like an administrator who has full access to the Postgres engine and can issue any command on every database, table, and user. You must enter a password when using a superuser to connect to the Postgres server.
- Privileges are rules that allow a user to run commands such as **SELECT, INSERT, DELETE TAVLE**. Privileges can either be granted or revoked by the server owner or by database superusers.
- Not revoking certain privileges allow unaware users to issue the wrong command and destroy the entire database.
- The most common practice when creating users is to create them then revoke all privileges, and then choose the privileges you want to grant.

## Syntax

- Connecting to the Postgres server with a secured user:

```
import psycopg2

conn = psycopg2.connect(user="postgres", password="abc123")
```
- Creating a user with a password that can create other users and databases:

```
conn = psycopg2.connect(dbname='dq', user = 'postgres',
password='password')

cur = conn.cursor()

cur.execute("CREATE USER data_viewer WITH CREATEROLE CREATEUSER CREATEDB PASSWORD
'somepassword'")

conn.commit()
```
- Revoking privileges from a user on a table:

```
conn = psycopg2.connect(dbname="dq", user="dq")

cur = conn.cursor()

cur.execute("REVOKE ALL ON user_accounts FROM data_viewer")

conn.commit()
```
- Grating the privileges to a user on a table:

```
conn = psycopg2.connect(dbname="dq", user="dq")

cur = conn.cursor()
```

```
cur.execute("GRANT SELECT ON user_accounts TO data_viewer")
conn.commit()
```

- Creating a group:

```
CREATE GROUP some_group NOLOGIN
```

- Assigning a user to a group:

```
GRANT some_group TO new_user
```

- Creating a database with a specified owner:

```
CREATE DATABASE accounts OWNER postgres
```

## ***Resources***

- [User privileges](#)
- [Groups](#)

# Exploring Postgres Internals

## Concepts

- In every Postgres engine, there are a set of internal tables Postgres uses to manage its entire structure. These contain all the information about data, names of tables, and types stored in a Postgres database.
- We can use the **information\_schema** table to get a high-level overview of what tables are stored in the database.
- The **information\_schema.tables** structure is as follows:

| Name                         | Data Type      | Description                                                                                                                                                                     |
|------------------------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| table_catalog                | sql_identifier | Name of the database that contains the table (always the current database)                                                                                                      |
| table_schema                 | sql_identifier | Name of the schema that contains the table                                                                                                                                      |
| table_name                   | sql_identifier | Name of the table                                                                                                                                                               |
| table_type                   | character_data | Type of the table: BASE TABLE for a persistent base table (the normal table type), VIEW for a view, FOREIGN TABLE for a foreign table, or LOCAL TEMPORARY for a temporary table |
| self_referencing_column_name | sql_identifier | Applies to a feature not available in PostgreSQL                                                                                                                                |
| reference_generation         | character_data | Applies to a feature not available in PostgreSQL                                                                                                                                |
| user_defined_type_catalog    | sql_identifier | If the table is a typed table, the name of the database that contains the underlying data type (always the current database), else null.                                        |
| user_defined_type_schema     | sql_identifier | If the table is a typed table, the name of the schema that contains the underlying data type, else null.                                                                        |

|                        |                |                                                                                                                                                                           |
|------------------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| user_defined_type_name | sql_identifier | If the table is a typed table, the name of the underlying data type, else null.                                                                                           |
| is_insertable_into     | yes_or_no      | YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)                                                           |
| is_typed               | yes_or_no      | YES if the table is a typed table, NO if not                                                                                                                              |
| commit_action          | character_data | If the table is a temporary table, then PRESERVE, else null. (The SQL standard defines other commit actions for temporary tables, which are not supported by PostgreSQL.) |

- In Postgres, schemas are used as a namespace for tables with the distinct purpose of separating them into isolated groups or sets within a database.
- **AsIs** keeps the valid SQL representation of a non-string quoted instead of converting it.
- Using an internal table, we can accurately map the types for every column in a table.

## Syntax

- Getting all the tables within a Postgres database:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="eRqg123EEk1")

cur = conn.cursor()

cur.execute("SELECT table_name FROM information_schema.tables WHERE
table_schema='public' ORDER BY table_name")
```

- Converting the name of a table to a Postgres string:

```
from psycopg2.extensions import AsIs

table_name = "state_info"

proper_interpolation = cur.mogrify("SELECT * FROM %s LIMIT 0",
[AsIs(table_name)])

cur_execute(proper_interpolation)
```

## Resources

- [The Information schema](#)
- [System catalogs](#)
- [pg\\_type table description](#)

- [pg\\_aggregate table description](#)

# Debugging Postgres Queries

## Concepts

- Path of a query:
  - The query is parsed for correct syntax. If there are any errors, the query does not execute, and you receive an error message. If error-free, then the query is transformed into a query tree.
  - A rewrite system takes the query tree and checks against the system catalog internal tables for any special rules. Then, if there are any rules, it rewrites them into the query tree.
  - The rewritten query tree is then processed by the planner/optimizer which creates a query plan to send to the executor. The planner ensures that this is the fastest possible route for query execution.
  - The executor takes in the query plan, runs each step, then returns back any rows it found.
- The **EXPLAIN** command examines the query at the third step in the path.
- For any query, there are multiple paths leading to the same answer and the paths keep increasing as the complexity of a query grows.
- Query plans are a **Seq Scan**, which means the executor will loop through every row one at a time.
- You can format the query plan in the following formats:
  - Text
  - XML
  - JSON
  - YAML
- Both **Startup Cost** and **Total Cost** are estimated values that are measured as an arbitrary unit of time.
  - **Startup Cost** represents the time it takes before a row can be returned.
  - **Total Cost** includes **Startup Cost** and is the total time it takes to run the node plan until completion.
- Joins are computationally expensive to perform and the biggest culprit in delaying execution time.

- Before a join can occur, a **Seq Scan** is performed on each joined table. These operations can quickly become inefficient as the size of the tables increase.

## Syntax

- Returning a query plan:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute('EXPLAIN SELECT * FROM homeless_by_coc')

print(cur.fetchall())
```

- Formatting the query plan of a query:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("EXPLAIN (format json) SELECT COUNT(*) FROM homeless_by_coc
WHERE year > '2012-01-01'")
```

- Returning actual run time statistics of a query:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("EXPLAIN (ANALYZE, FORMAT json) SELECT COUNT(*) FROM
homeless_by_coc WHERE year > '2012-01-01'")
```

- Reverting any changes made to the database:

```
import psycopg2

conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

Modifying change to the database.

cur.execute("EXPLAIN ANALYZE ALTER TABLE state_info ADD COLUMN notes
text")

Reverting the change.

conn.rollback()
```

- Executing an inner join in Postgres:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()
```

```
cur.execute("EXPLAIN (ANALYZE, FORMAT json) SELECT hbc.state, hbc,
coc_number, hbc.coc_name, si.name FROM homeless_by_coc as hbc,
state_info as si WHERE hbc.state = si.postal")
```

## Resources

- [Postgres EXPLAIN statement](#)
- [Full table scan](#)

# Using an Index

## Concepts

- Indexes can speed up queries to run in  $O(\log(n))$  complexity as opposed to  $O(n)$ .
- Postgres will use an **Index Scan** instead of **Seq Scan** When filtering the table on a primary key column.
- When performing an index scan, Postgres will:
  - Use binary search to find the first row that matches a filter condition and store the row in a temporary collection.
  - Advance to the next row to look for and more rows that matches a filter condition and add those rows to the temporary collection.
  - Return the final collection of rows that matched.
- When filtering a table on value a in a primary key column, Postgres ensures that it is also unique. Postgres knows to stop search when it finds the instance that match the primary key value.
- A composite primary key is created when more than one column is used to specify the primary key of a table.
- Indexes give us tremendous speed benefits at the cost of space.

## Syntax

- Creating a table with a composite primary key:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("""
CREATE TABLE state_idx (
 state CHAR(2),
 homeless_id INT,
 PRIMARY KEY (state, homeless_id)
)
""")

cur.execute("INSERT INTO state_idx SELECT state, id FROM
homeless_by_coc")

conn.commit()
```

- Creating a Postgres index:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("CREATE INDEX state_idx ON homeless_by_coc(state)")

conn.commit()
```

- Deleting an index:

```
DROP INDEX state_idx

OR

DROP INDEX IF EXISTS state_idx
```

## ***Resources***

- [Constraints](#)
- [Indexes](#)

# Advanced Indexing

## Concepts

- Indexes create a B-Tree structure on a column, which allows filtered queries to perform binary search.
- Indexes reduce query speeds from  $O(n)$  complexity to  $O(\log(n))$ .
- A **Bitmap Heap Scan** occurs when Postgres encounters two, or more, columns that contain an index. A heap scan follows these steps:
  - Runs through the indexed column and selects all the rows that match a filter.
  - Creates a **Bitmap Heap** that is used as the temporary index.
  - Scans through the **Bitmap Heap** and selects all rows that match a different filter.
  - Returns the results.
- A **Bitmap Heap Scan** is more efficient than a pure **Seq Scan** because the number of filtered rows in an index will always be less than or equal to the number of rows in the full table.
- Postgres allows up to 32 columns to be indexed.
- Along with passing in additional options when creating an index, an index can be created using any Postgres function.
- Partial indexes restrict an index to a range of data and can be independent from the column that is being queried.

## Syntax

- Creating an index:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("CREATE INDEX state_idx ON homeless_by_coc(state)")

conn.commit()
```

- Creating a multi-column index:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("CREATE INDEX state_year_idx ON homeless_by_coc(state,
year)")
```

- Creating a partial index:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("CREATE INDEX state_count_idx ON homeless_by_coc(state)
WHERE count > 0")

conn.commit()
```

## Resources

- [Partial indexes](#)
- [Indexes on expressions](#)

# Vacuuming Postgres Databases

## Concepts

- When running a **DELETE** query on a table, Postgres marks rows as dead, which means they will be eventually removed as opposed to removing them entirely.
- Postgres transactions follow a set of properties called ACID. ACID stands for:
  - Atomicity: If one thing fails in the transaction the whole transaction fails.
  - Consistency: A transaction will move the database from one valid set to another.
  - Isolation: concurrent effects to the database will be followed through as sequential changes.
  - Durability: Once the transaction is committed, it will stay that way regardless of crash, power outage, or some other catastrophic event.
- Postgres uses multi-version control that a user keeps a consistent version of her expected database state during the transaction.
- Vacuuming a table will remove the marked dead rows and reclaim the space they took from the table.
- A transaction block is a sequence of commands wrapped between **BEGIN** and **COMMIT**.
- **BEGIN** starts a transaction block, **COMMIT** commits the current transaction, and **ROLLBACK** aborts the current transaction.
- No insert, update, or delete queries can be issued against the table during the vacuum duration. Select queries on the table are considerably slowed down to the point where they are unusable.
- Postgres offers a feature called autovacuum and it runs periodically on tables to ensure the dead rows are removed and your statistics are up-to-date.

## Syntax

- Deleting rows from table:

```
conn = psycopg2.connect(dbname="dq", user="hud_admin",
password="abc123")

cur = conn.cursor()

cur.execute("DELETE FROM homeless_by_coc")
```

- Removing dead rows from a table:

```
Vacuum a single table

VACUUM homeless_by_coc
```

**OR**

```
Vacuum each user created table
VACUUM
```

- Aborting the transaction:

```
conn.rollback()
```

- Committing changes immediately:

```
conn.autocommit = True
```

- Updating table statistics:

```
VACUUM ANALYZE table_name
```

- Reclaiming lost space for a table:

```
VACUUM FULL homeless_by_coc
```

## **Resources**

- [Postgres BEGIN](#)
- [Autovacuum](#)

# Handling Big Data Sets

## Optimizing Dataframe Memory Footprint

### Concepts

- The BlockManager class is responsible for maintaining the mapping between the row and column indexes and the blocks of values of the same data type.
- Pandas uses the ObjectBlock class to represent blocks containing string columns and the FloatBlock class to represent blocks containing float columns.
- Pandas represent numeric values as NumPy ndarrays, whereas pandas represent string values as Python string objects.
- A kilobyte is equivalent to 1,024 bytes and a megabyte is equivalent to 1,048,576 bytes.
- Many types in pandas have multiple subtypes that can use fewer bytes to represent each value. For example, the **float** type has the **float16**, **float32**, **float64**, and **float128** subtypes. The number portion of a type's name indicates the number of bits that type uses to represent values.
- Subtypes for the most common panda types:

| object | bool | float    | int   | datetime   |
|--------|------|----------|-------|------------|
| object | bool | float16  | int8  | datetime64 |
|        |      | float32  | int16 |            |
|        |      | float64  | int32 |            |
|        |      | float128 | int64 |            |

- The category datatype uses integer values under the hood to represent the values in a column, rather than the raw values. Categoricals are useful whenever a column contains a limited set of values.

### Syntax

- Returning an estimate for the amount of memory a dataframe consumes:

```
DataFrame.info()
```

- Retrieving the internal BlockManager object:

```
DataFrame._data
```

- Retrieving the amount of memory, the values in a column consume:

```
Series nbytes
```

- Returning the number of values in a dataframe:

```
DataFrame.size
```

- Returning the true memory footprint of a dataframe:

```
DataFrame.info(memory_usage="deep")
```

- Returning the amount of memory each column consumes:

```
DataFrame.memory_usage(deep=True)
```

- Finding the minimum and maximum values for each integer subtype:

```
import numpy as np

int_types = ["int8", "int16", "int32", "int64"]

for it in int_types:
 print(np.iinfo(it))
```

- Finding the minimum and maximum values for each float subtype:

```
import numpy as np

float_types = ["float16", "float32", "float64", "float128"]

for ft in float_types:
 print(np.finfo(ft))
```

- Converting a column to a specific datatype:

```
Series.astype()
```

- Converting a column to the most space efficient subtype:

```
pd.to_numeric(Series, downcast='integer')
```

- Converting a column to the datetime type:

```
pd.to_datetime(Series)
```

- Converting a column to the category datatype:

```
Series.astype('Category')
```

- Specify the column types when reading in data:

```
import numpy as np
col_types = {"id": np.int32}
df = pd.read_csv('data.csv', dtypes=col_types)
```

## Resources

- [Documentation for the BlockManager class](#)
- [Documentation for the pd.read\\_csv\(\) function](#)

# Processing Dataframes in Chunks

## Concepts

- We can use chunking to load in and process dataframes in chunks when working with data sets that don't fit into memory.
- Breaking a task down, processing the different parts separately, and combining them later on is an important workflow in batch processing.
- We can cut down the overall running time by only loading in the columns we're examining.

## Syntax

- Specifying the number of rows we want each chunk of a dataframe to contain:

```
chunk_iter = pd.read_csv("data.csv", chunksize = 10000)
```

- Printing each chunk:

```
for chunk in chunk_iter:
 print(chunk)
```

- Combining pandas objects:

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

- Creating a GroupBy object:

```
s4 = s3.groupby(s3.index)
```

- Observing the groups in a GroupBy object:

```
for key, item is s4:
 print(key.get_group(key))
```

- Timing chunking and processing:

```
%%timeit

lifespans = []

chunk_iter = pd.read_csv("moma.csv", chunksize=250,
dtype={"ConstituentBeginDate": "float", "ConstituentEndDate": "float"},
usecols=['ConstituentBeginDate', 'ConstituentEndDate'])

for chunk in chunk_iter:
 lifespans.append(chunk['ConstituentEndDate'] -
 chunk['ConstituentBeginDate'])

lifespans_dist = pd.concat(lifespans)
```

## ***Resources***

- [Batch Processing](#)
- [Documentation for the Series.groupby method](#)

# Augmenting Pandas with SQLite

## Concepts

- Pandas stores and work with data in memory, whereas a database like SQLite can represent data on disk.
- While pandas is limited by the amount of available memory, SQLite is limited only by the amount of available disk space.
- SQLite data types:

| Type    | Description                                                                                                 |
|---------|-------------------------------------------------------------------------------------------------------------|
| NULL    | The value is a NULL value                                                                                   |
| INTEGER | The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes, depending on the magnitude of the value |
| REAL    | The value is a floating point value, stored as an 8-byte IEEE floating point number                         |
| TEXT    | The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE)                |
| BLOB    | The value is a blob of data, stored exactly as it was entered                                               |

- Selecting the correct types in SQLite reduces the disk footprint of the database file, and can make some SQLite operations faster.
- Generating a pandas dataframe using SQL allows us to do data selection with SQL, but the iterative exploration and analysis using pandas.
- Pandas has several advantages over SQLite such as:
  - Pandas has a large suite of functions and methods for performing common operations.
  - Pandas has a diverse type system we can use to save space and improve code running speed.
  - Pandas works in memory and will be quicker for most tasks.
- Querying data in SQL and working with batches of the results set will help you get the most out of SQL and pandas.

## Syntax

- Appending rows to a SQLite database table:

```
import sqlite3

import pandas as pd

conn = sqlite3.connect('moma.db')

moma_iter = pd.read_csv('moma.csv', chunksize=1000)

for chunk in moma_iter:

 chunk.to_sql("exhibitions", conn, if_exists='append',
index=False)
```

- Using pandas to query a SQLite database:

```
conn = sqlite3.connect('test.db')

df = pd.DataFrame({'A': [0,1,2], 'B': [3,4,5]})

df.to_sql('test', conn)

pd.read_sql('select A from test', conn)
```

## *Resources*

- [SQLite Datatypes](#)
- [Limits in SQLite](#)

# CPU Bound Programs

## Concepts

- A memory limitation is when a data set won't fit into memory available on the computer.
- A program bound is similar to a limitation in that it affects how you're able to process your data.  
The two primary ways a program can be bound are:
  - CPU-bound — The program will be dependent on your CPU to execute quickly. The faster your processor is, the faster your program will be.
  - I/O-bound — The program will be dependent on external resources, like files on disk and network services to execute quickly. The faster these external resources can be accessed, the faster your program will run.
- The more efficient you make your code, the less back-and-forth trips will need to be made, and the faster your code will run.
- Big O notation expresses time complexity in terms of the length of the input variable, represented as  $n$ .
- Big O notation is a great way for estimating the time complexity of algorithms where:
  - You can easily trace all of the function calls and understand any nested time complexity.
  - The code is relatively straightforward.
- The general process behind refactoring code is:
  - Measure how long the current code takes to run.
  - Rewriting the code so that the algorithm you want is nicely isolated from the rest of the code.
  - Rewriting the algorithm to reduce time complexity.
  - Benchmarking the new algorithm to see if it's faster.
  - Rinse and repeat as needed.
- Space complexity indicates how much additional space in memory our code uses over and above the input arguments.

## Syntax

- Finding duplicate values in columns:  
`DataFrame.duplicated()`
- Seeing how long a piece of code takes to run:

```
import time

start = time.time()

We're timing how long this line takes to run.

duplicates = []

elapsed = time.time() - start

print("Took {} seconds to run.".format(elapsed))
```

- Calling your program from the command line using the time command:

```
time python script.py
```

- Using the cProfile function to display timing statistics of a program or a line of code:

```
import cProfile

cProfile.run('print(10)')
```

## Resources

- [Documentation for pandas duplicated method](#)
- [Big O notation](#)
- [cProfile](#)
- [Unix time command](#)
- [Contexttimer](#)
- [lineprofiler](#)
- [Complexity Analysis Introduction](#)

# I/O Bound Programs

## Concepts

- CPU bound tasks are tasks where our Python program is executing something. CPU bound tasks will:
  - Execute faster if you optimize the algorithm.
  - Execute faster if your processor has a higher clock speed (can execute more operations).
- I/O bound tasks aren't using your CPU at all and waiting for something else to finish. I/O bound tasks are tasks where:
  - Our program is reading from an input (like a CSV file).
  - Our program is writing to an output (like a text file).
  - Our program is waiting for another program to execute something (like a SQL query).
  - Our program is waiting for another server to execute something (like an API request).
- A task is blocked when it's waiting for something to happen. When a thread is blocked, it isn't running any operations on the CPU.
- The hard drive is the slowest way to do I/O because it reads in data more slowly than memory and is much farther away from the CPU than memory.
- Threading allows us to execute tasks that are I/O bound more quickly. Threading makes CPU usage more efficient because when one thread is waiting around for a query to finish, another thread can process the result.
- Locking ensures that only one thread is accessing a shared resource at any time.  
The **threading.Lock.acquire()** method acquires the Lock and prevents any other thread from proceeding until it can also acquire the lock. The **threading.Lock.release()** method releases the Lock so other threads can acquire it.
- Examples of shared resources are:
  - The system stdout.
  - SQL databases.
  - APIs.
  - Objects in memory.

## Syntax

- Initializing an in-memory database:

```
memory = sqlite3.connect(':memory:')
```

- Reading the contents of the disk database:

```
disk = sqlite3.connect('lahman2015.sqlite')

dump = "" .join(line for line in disk.iterdump())
```

- Copying the database from disk into memory:

```
memory.executescript(dump)
```

- Creating a new thread:

```
thread = threading.Thread(target=task, args=(team,))

thread.start()
```

- Joining threads:

```
t1 = threading.Thread(target=task, args=(team,))

t2 = threading.Thread(target=task, args=(team,))

t3 = threading.Thread(target=task, args=(team,))

Start the first three threads

t1.start()

t2.start()

t3.start()

t1.join() # Wait until t1 finishes.

t2.join() # Wait until t2 finishes. If it already finished, then keep
going.

t3.join() # Wait until t3 finishes. If it already finished, then keep
going.
```

- Creating a lock:

```
lock = threading.Lock()

def task(team):

 lock.acquire()

 # This code cannot be executed until a thread acquires the lock.

 print(team)

 lock.release()

t1 = threading.Thread(target=task, args=(team,))

t2 = threading.Thread(target=task, args=(team,))

t1.start()

t2.start()
```

## ***Resources***

- [threading.Thread class](#)
- [Global Interpreter Lock](#)
- [Threading](#)

# Overcoming the Limitations Of Threads

## Concepts

- The GIL (Global Interpreter Lock) in Cpython only allows one thread at a time to execute Python code using a locking mechanism.
- Python enables us to write at a high abstraction layer, which means that code can be extremely terse, but still achieve a lot.
- Threading can speed up I/O bound programs since the GIL only applies to executing Python code.
- The GIL gets released when we do I/O operations but can also get released in situations where you're calling external libraries that have significant components written in other languages that aren't bounded by the GIL.
- Threads are good for situations where you have long-running I/O bound tasks, but they aren't so good where you have CPU-bound tasks, or you have tasks that will run very quickly.
- Processes are best when your task is CPU bound or when your task will take long enough.
- Threads run inside processes and each process has its own memory, and all the threads inside share the same memory.
- One thread can be running inside each Python interpreter at a time, so starting multiple processes enables us to avoid the GIL.
- Creating a process is a relatively "heavy" operation and takes time. Threads, since they're inside processes, are much faster to make.
- Deadlocks happen when two threads or processes both require a lock that the other process has before proceeding.

## Syntax

- Turning a Python object into bytecode:

```
def myfunc(alist):
 return len(alist)

dis.dis(myfunc)
```

- Processing objects to represent activity that is run in a separate process:

```
import multiprocessing

def task(email):
 print(email)
```

```
process = multiprocessing.Process(target=task, args=(email,))
process.start()
process.join()
```

- Using the Pipe class to pipe data between processes:

```
import multiprocessing

def echo_email(email, conn):
 # Sends the email through the pipe to the parent process.
 conn.send(email)

 # Close the connection, since the process will terminate.
 conn.close()

 # Creates a parent connection (which we'll use in this thread), and a
 # child connection (which we'll pass in).
 parent_conn, child_conn = multiprocessing.Pipe()

 # Pass the child connection into the child process.

 p = multiprocessing.Process(target=echo_email, args=(email,
 child_conn,))

 # Start the process.
 p.start()

 # Block until we get data from the child.
 print(parent_conn.recv())

 # Wait for the process to finish.

 p.join()
```

- Creating a Pool of processes:

```
from multiprocessing import Pool

Create a pool of workers.

p = Pool(5)
```

## Resources

- [CPython](#)
- [Global Interpreter Lock](#)
- [Multiprocessing library](#)

# Quickly Analyzing Data with Parallel Processing

## Concepts

- The **threading** and **multiprocessing** packages are widely used and gives you more low-level control.
- The **concurrent.futures** package allows for a simple and consistent interface for both threads and processes.
- **concurrent.features.ThreadPoolExecutor.map()** method returns a generator, but you can just call it using **list** to force it to evaluate.
- Threads and processes are using a paradigm called MapReduce, which is utilized in data processing tools like Apache Hadoop and Apache Spark.

## Syntax

- Creating a pool of threads:

```
import concurrent.futures

def word_length(word):
 return len(word)

pool = concurrent.futures.ThreadPoolExecutor(max_workers=10)

lengths = pool.map(word_length, ["Hello", "are", "you", "thinking", "of", "becoming", "a",
"polar", "bear", "?"])
```

- Creating a pool of processes:

```
import concurrent.futures

def word_length(word):
 return len(word)

pool = concurrent.futures.ProcessPoolExecutor(max_workers=10)

lengths = pool.map(word_length, ["Hello", "are", "you", "thinking", "of", "becoming", "a",
"polar", "bear", "?"])
```

## Resources

- [Debugging using the multiprocessing module](#)
- [Documentation for concurrent.features module](#)

# Processing Tasks with Stacks and Queues

## Concepts

- You should use Worker pools work well when you have a fixed amount of work; they do not well when you have work that keeps coming in.
- A stack is a data structure that takes in new elements. Stacks are a way to implement the theoretically more efficient method of prioritization by following a particular order in which the operations are performed.
- A queue is a first in first out system, where the tasks that arrived first get processed first.
- Queues are more "fair" than a stack — all tasks get the same priority, and none of them get processed early.
- Queues are generally best when you want all tasks processed at about the same pace, and queues usually have a fairly low maximum wait time for processing tasks.
- Stacks are generally best when you are okay waiting around while tasks finish processing. Stacks have a fairly high maximum wait time.
- When adding more elements to stacks:
  - Items added to a stack towards the end are processed much faster than items added towards the beginning.
  - Some stack tasks are finished almost immediately after they're added.
  - The worst-case queue time in a stack is equivalent to waiting for every single task to be processed first.
- When adding more elements to queues:
  - Items added to a queue towards the end are processed more slowly than items added earlier (this depends strongly on the throughput of the task processor).
  - Only the first item added to a queue is processed instantly (given that tasks are added faster than they can be processed).
  - The worst-case queue time for a queue depends on the throughput of the task processor.

## Syntax

- Adding an element to the "top" of a stack:

```
stack = [1,2]
```

```
stack.insert(0,3)
```

- Removing an element from the "top" of a stack:

```
stack.pop(0)
```

- Creating a stack:

```
class Stack():

 def __init__(self):
 self.items = []

 def push(self, value):
 self.items.insert(0, value)

 def pop(self):
 return self.items.pop(0)

 def count(self):
 return len(self.items)
```

- Adding an element to the bottom of a queue:

```
queue = [1,2]

queue.append(3)

queue.append(4)
```

- Removing the top element of a queue:

```
queue.pop(0)
```

- Creating a queue:

```
class Queue():

 def __init__(self):
 self.items = []

 def push(self, value):
 self.items.append(value)

 def pop(self):
 return self.items.pop(0)

 def count(self):
 return len(self.items)
```

## Resources

- [Stack Data Structure](#)

- [Queue Data Structure](#)

# Effectively Using Arrays and Lists

## Concepts

- Binary is a convenient way to store data since you only need to store two "positions".
- Python lists are implemented as C arrays, but lists don't have a fixed size, and they can store elements of any type.
- A pointer is a special kind of variable in C that points to the value of another variable in memory. Pointers allow us to refer to the same value in multiple places without having to copy the value.
- The NumPy array type is based on a C array and behaves very similarly, so it's a better choice for implementing an array class than a Python list.
- Linked lists allow you to flexibly add as many elements as desired. Linked lists achieve this by storing links between items.
- Linked lists don't allow for directly indexing an item like in an array. Instead, we need to scan through the list to find the item we want.
- Linked lists have the following characteristics:
  - You don't have to specify how many nodes you want upfront — you can store as many values as you want.
  - Data isn't restricted to a single type — you can store any data you want in any node.
  - Finding an element in a linked list has time complexity  $O(n)$  time since we need to iterate through all of the elements to find the one, we want.
  - Insertions and deletions are fast since we don't need to copy anything — we just need to find the insertion or deletion point.
- Linked lists are better if you're storing values, and you don't know how many you want to store. Linked lists are also easier to combine and shuffle, which makes them very useful when gathering data.
- Arrays are better when you need to access data quickly but won't be changing it much. Arrays are usually much better for computation, such as when you're analyzing data.

## Syntax

- Converting an integer to binary:  
`bin(10)`
- Converting a number as a string to a decimal integer:  
`int("1010", 2)`

- Retrieving the hexadecimal memory address of any variable:

```
hex(id(1))
```

- Implementing an array as a class:

```
import numpy as np

class Array():

 def __init__(self, size):
 self.array = np.zeros(size, dtype=np.float64)
 self.size = size
```

- Accessing and setting items in a class:

```
def __getitem__(self, key):
 return self.array[key]

def __setitem__(self, key):
 return self.array[key]
```

- Inserting items to an list:

```
list = [1,2,3]

list.insert(1,5) # list is now [1,5,2,3]
```

## Resources

- [Python List Implementation](#)
- [NumPy Arrays](#)
- [Linked Lists](#)
- [Arrays](#)

# Sorting Arrays And Lists

## Concepts

- The default sorting behavior isn't ideal in the following cases:
  - We have a custom data structure, and we want to sort it. For example, we want to sort a set of JSON files.
  - We're working with data that's too large to fit into memory, but we still want to ensure that everything is sorted. This may require splitting the data across multiple machines to sort, and then combining the sorted results.
  - We want a custom ordering — for example, we want to sort locations based on their proximity to one or more cities. We can't sort by simple distance to the closest city, since we want to take distance to multiple cities into account.
- There are a variety of different sorting techniques that have different time and space complexity trade-offs.
- The basic unifying factor behind most sorting algorithms is the idea of the swap. Most sorting algorithms differ only in which order different items are swapped.
- Pseudocode for making a swap:
  - Select the two elements you want to swap.
  - Copy the first element to an external variable.
  - Replace the first element with the value of the second.
  - Replace the second element with the value of the external variable.
- How each type of sort works:
  - The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
  - A bubble sort works by making passes across an array and "bubbling" values until the sort is done.
  - The insertion sort works by looping through each element in the array and "inserting" it into a sorted list at the beginning of an array.
- Time complexity for each sorting algorithm:
  - Selection sort has  $O(n^2)$  time complexity.
  - A bubble sort has time complexity of  $O(n)$  when the array is already sorted but has time complexity of  $O(n^2)$  otherwise.

- An insertion sort has time complexity of  $O(n)$  when the array is already sorted but has time complexity of  $O(n^2)$  otherwise.
- Space complexity for each sorting algorithm:
  - Selection sort has  $O(1)$  space complexity.
  - Bubble sort has  $O(1)$  space complexity.
  - Insertion sort also has  $O(1)$  space complexity.

## Syntax

- Sorting a list:

```
list.sort()
```

- Implementing a swap:

```
def swap(array, pos1, pos2):
 store = array[pos1]
 array[pos1] = array[pos2]
 array[pos2] = store
```

- Implementing selection sort:

```
def selection_sort(array):
 for i in range(len(array)):
 lowest_index = i
 for z in range(i, len(array)):
 if array[z] < array[lowest_index]:
 lowest_index = z
 swap(array, lowest_index, i)
```

- Implementing bubble sort:

```
def bubble_sort(array):
 swaps = 1
 while swaps > 0:
 swaps = 0
 for i in range(len(array) - 1):
 if array[i] > array[i+1]:
 swap(array, i, i+1)
 swaps += 1
```

- Implementing insertion sort:

```
def insertion_sort(array):
 for i in range(len(array)):
 j = i
 while j > 0 and array[j - 1] > array[j]:
 swap(array, j, j-1)
 j -= 1
```

## *Resources*

- [Selection Sort](#)
- [Bubble Sort](#)
- [Insertion Sort](#)
- [Sorting Terminology](#)

# Searching Arrays and Lists

## Concepts

- You'll want to implement your own searching logic in some cases. Example cases include:
  - You want to find all occurrences of a term.
  - You have custom search logic across multiple fields in a row.
  - You have a data structure that doesn't have built-in search, like a linked list.
  - You want a higher-performance search algorithm for your use case.
- Time complexity of a linear search if you're only looking for the first element that matches your search:
  - In the best case, when the item you want to find is first in the list, the complexity is  $O(1)$ .
  - In the average case, when the item you want is in the middle of the list, the complexity is  $O(n/2)$ , which simplifies to  $O(n)$ .
  - In the worst case, when the item you want is at the end of the list, the complexity is  $O(n)$ .
- Space complexity of a linear search:
  - When searching for multiple elements, linear search has  $O(n)$  space complexity.
  - When searching for the first matching element, it has space complexity of  $O(1)$ .
- The binary search algorithm looks for the midpoint of a given range and keeps narrowing the window in its search until the value is found.
- Binary search performs better than a linear search since it doesn't have to search every single element of the array.
- In general, you should use a linear search if:
  - You only need to search once.
  - You don't need to sort the list for another reason (like viewing items in order).
  - You have complex search criteria or require external lookups.
- You should use binary search if:
  - The data is already sorted, or you need to sort it for another reason.
  - You need to perform multiple searches.
  - You can distribute the sort across multiple machines, so it runs faster.

## Syntax

- Retrieving an index of a list:

```
list.index("item")
```

- Implementing linear search:

```
def linear_search(array, search):
 indexes = []
 for i, item in enumerate(array):
 if item == search:
 indexes.append(i)
 return indexes

sevens = linear_search(times, 7)
```

- Implementing a more complex linear search:

```
def linear_multi_search(array, search):
 indexes = []
 for i, item in enumerate(array):
 if item == search:
 indexes.append(i)
 return indexes

transactions = [[times[i], amounts[i]] for i in range(len(amounts))]
results = linear_multi_search(transactions, [56, 10.84])
```

- Implementing binary search:

```
def binary_search(array, search):
 counter = 0
 insertion_sort(array)
 m = 0
 i = 0
 z = len(array) - 1
 while i <= z:
 counter += 1
 m = math.floor(i + ((z - i) / 2))
 if array[m] == search:
```

```
 return m

elif array[m] < search:
 i = m + 1

elif array[m] > search:
 z = m - 1

return counter
```

## *Resources*

- [List of Unicode characters](#)
- [Binary Search](#)

# Hash Tables

## Concepts

- A hash table allows us to store data with a key that is associated with a value.
- Time complexity used to look up a value when using a key is O(1).
- A hash table stores their data in an array.
- A modulo, denoted with "%" in Python, is a mathematical operator that finds the remainder after a number is divided by another. For example:
  - $11 \% 10 = 1$
  - $10 \% 10 = 0$
  - $20 \% 3 = 2$
  - $23 \% 9 = 5$
- A hash collision occurs when two different values results in the same hash.
- One way to avoid collisions is to store a list of values at each array position.
- The time complexity is O(n) when all the elements in the hash table are in a single list.
- The Python **dict** class is implemented to optimize for speed, but the following principles apply:
  - Insertion is generally O(1), but worst-case can be O(n).
  - Indexing is generally O(1), but worst-case can be O(n).
- A dictionary is an excellent data structure since hash tables can be combined fairly easily. It's very commonly used when doing distributed computation, and understanding the memory and lookup time constraints can be very helpful.

## Syntax

- Retrieving an unicode character code:

```
ord(string)
```

- Creating a simple hash function:

```
def simple_hash(key):
 key = str(key)
 return ord(key[0])
```

- Implementing a hash table:

```
class HashTable():
```

```

def __init__(self, size):
 self.array = np.zeros(size, dtype=np.object)
 self.size = size

def __getitem__(self, key):
 ind = hash(key) % self.size
 for k,v in self.array[ind]:
 if key == k:
 return v

def __setitem__(self, key, value):
 ind = hash(key) % self.size
 if not isinstance(self.array[ind], list):
 self.array[ind] = []
 replace = None
 for i,data in enumerate(self.array[ind]):
 if data[0] == key:
 replace = i
 if replace is None:
 self.array[ind].append((key,value))
 else:
 self.array[ind][replace] = (key, value)

```

- Returning the hash value of an object:

`hash(object)`

## Resources

- [Unicode HOWTO](#)
- [Python dictionary implementation](#)

# Overview of Recursion

## Concepts

- Recursion is commonly defined as a function that calls itself.
- A terminating case makes sure recursion from continuing forever. The terminating case is also known as the base case.
- The base case is necessary to ensure your program doesn't run out of memory.
- A call stack is a stack data structure that stores information about the active subroutines of a computer program.
- A stack overflow occurs if the call stack pointer exceeds the stack bound. Stack overflow is a common cause of infinite recursion.
- Divide and conquer involves splitting the problem into a set of smaller sub problems that are easier. After reaching the easier terminal case, the values that will solve the general problem at hand gets returned.
- The goal of the merge sort algorithm is to first divide up an unsorted list into a bunch of smaller sorted lists and then merge them all together to create a sorted list.
- The time complexity for merge sort is  $O(n\log_2 n)$ .

## Syntax

- Implementing recursion:

```
example_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def recursive_summation(values):
 if len(values) <= 1:
 return values[0]
 return values[0] + recursive_summation(values[1:])

recursive_summation(example_list)
```

- Implementing divide and conquer:

```
f = open('random_integers.txt', 'r')

random_integers = [int(line) for line in f.readlines()]

def summation(values):
 if len(values) == 0:
 return 0
 if len(values) == 1:
```

```

 return values[0]

 midpoint = len(values) // 2

 return summation(values[:midpoint]) +
 summation(values[midpoint:])

divide_and_conquer_sum = summation(random_integers)

```

- Implementing merge sort:

```

def merge(left_list, right_list):
 sorted = []
 while left_list and right_list:
 if left_list[0] < right_list[0]:
 sorted.append(left_list.pop(0))
 else:
 sorted.append(right_list.pop(0))
 sorted += left_list
 sorted += right_list
 return sorted

```

## *Resources*

- [Recursion](#)
- [Merge Sort Algorithm](#)

# Introduction to Binary Trees

## Concepts

- A tree data structure is naturally described using recursion.
- A node is an abstract data type that contains references to left and right nodes.
  - Nodes can be **None** or can store other data, like an integer.
  - The topmost node is known as the root node.
  - A node that has left and right references of type **None** is known as a leaf.
- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- A child node is a node that is added and that isn't the root.
- A parent node is the node the child node references. The root node is the only parent node with no parent.
- Level order insert works the following way:
  - Given a list of integers, start at the first element and make it the root node.
  - Continue down the list adding the following two nodes at the second level, then the next four as the third level, etc. The pattern is that at level  $k$ , you will have  $2^{k-1}$  nodes.
- The value of every left side node corresponds to  $2 * \text{index of referrer} + 1$ .
- The value of every right-side node corresponds to  $2 * \text{index of referrer} + 2$ .
- A node is an interior node if the node has a parent *and* it has both a left node or a right mode.
- Traversal methods:
  - Preorder Traversal:
    - Handles the value of the current node.
    - Recursively traverse all the nodes on the left side.
    - Recursively traverse all the nodes on the right side.
  - In order Traversal:
    - Recursively traverse all the nodes on the left side.
    - Handles the value of the current node.
    - Recursively traverse all the nodes on the right side.
  - Post order Traversal:

- Recursively traverse all the nodes on the left side.
  - Recursively traverse all the nodes on the right side.
  - Handles the value of the current node.
- The depth of a tree is defined as the maximum level of a tree. We can find the depth of the tree by recursively traversing it.
  - A balanced tree is a tree in which for every node, a subtree's height does not differ by more than one.
  - A complete binary tree is a tree that has all levels completely filled except the last level.
  - Binary heaps allow us to query the top values of extremely large data sets.

## Syntax

- Implementing a binary tree:

```
class Node:

 def __init__(self, value=None):
 self.value = value
 self.left = None
 self.right = None

class BinaryTree:

 def __init__(self, root=None):
 self.root = root

 def insert(self, value):
 if not self.root:
 self.root = Node(value=value)
 elif not self.root.left:
 self.root.left = Node(value = value)
 elif not self.root.right:
 self.root.right = Node(value = value)
```

## Resources

- [Binary trees](#)
- [Tree data structures for beginners](#)

# Implementing a Binary Heap

## Concepts

- A binary heap is a version of a complete binary tree.
- A binary tree is complete if the tree's levels are filled in except for the last that has nodes filled in from left to right.
- A binary heap, or heap, requires the two following conditions to hold:
  - It must be a complete binary tree.
  - The value of a parent is greater than or equal to OR less than or equal to any of its child nodes.
- Categories of heaps:
  - If it is a max-heap, than each of the parent nodes is greater than or equal to any of its child nodes.
  - If it is a min-heap, each of the parent nodes is less than or equal to any of its child nodes.
- **heapq** is Python's own heap implementation and works both as a max-heap and min-heap.

## Syntax

- Implementing a max-heap:

```
class MaxHeap:

 def __init__(self):
 self.nodes = []

 def insert(self, value):
 self.nodes.append(value)
 index = len(self.nodes) - 1
 parent_index = math.floor((index-1)/2)

 while index > 0 and self.nodes[parent_index] <
 self.nodes[index]:
 self.nodes[parent_index], self.nodes[index] =
 self.nodes[index], self.nodes[parent_index]

 index = parent_index
 parent_index = math.floor((index-1)/2)

 return self.nodes
```

- Returning a list of the top 100 elements:

```
heap = MaxHeap()

class MaxHeap(BaseMaxHeap):

 def top_n_elements(self, n):

 return [self.pop() for _ in range(n)]

heap = MaxHeap()

heap.insert_multiple(heap_list)

top_100 = heap.top_n_elements(100)
```

- Using Python's heap to return the top 100 elements:

```
top_100 = heapq.nlargest(100, heap_list, key=lambda x: x[1])
```

- Using Python's heap to return the bottom 100 elements:

```
bottom_100 = heapq.nsmallest(100, heap_list, key=lambda x: x[1])
```

## ***Resources***

- [Binary Heap](#)
- [Heap queue algorithm](#)

# Working with Binary Search Trees

## Concepts

- A BST provides the ability to run efficient range queries on data sets. A BST requires the following conditions to hold:
  - Every value in a nodes' left sub-tree has a value that is less than or equal to the parent node.
  - Every value in a nodes' right sub-tree has a value that is greater than or equal to the parent node.
- Every new node in a BST is inserted in sorted order.
- Searching for an item in a balanced binary tree has time complexity of  $O(\log n)$ . On the other hand, searching for an item in an unbalanced binary tree has time complexity  $O(n)$ .
- A BST that stays balanced for every insert is called a self-balancing BST.
- A tree rotation operation involves changing the structure of the tree while maintaining the order of the elements.

## Syntax

- Implementing a binary search tree (BST) :

```
class BST:

 def __init__(self):
 self.node = None

 def insert(self, value=None):
 node = Node(value=value)

 if not self.node:
 self.node = node
 self.node.right = BST()
 self.node.left = BST()
 return

 if value > self.node.value:
 if self.node.right:
 self.node.right.insert(value=value)
 else:
 self.node.right.node = node
 else:
```

```

 return

 if self.node.left:

 self.node.left.insert(value=value)

 else:

 self.node.left.node = node

def inorder(self, tree):

 if not tree or not tree.node:

 return []

 return (self.inorder(tree.node.left) + [tree.node.value] +
self.inorder(tree.node.right))

```

- Searching a BST:

```

class BST(BaseBST):

 def search(self, value):

 if not self.node:

 return False

 if value == self.node.value:

 return True

 result = False

 if self.node.left:

 result = self.node.left.search(value)

 if self.node.right:

 result = self.node.right.search(value)

 return result

```

- Performing rotation operations:

```

class BST(BaseBST):

 def left_rotate(self):

 old_node = self.node

 new_node = self.node.right.node

 if not new_node:

 return

 new_right_sub = new_node.left.node

 self.node = new_node

```

```
 old_node.right.node = new_right_sub
 new_node.left.node = old_node

def right_rotate(self):
 old_node = self.node
 new_node = self.node.left.node
 if not new_node:
 return
 new_left_sub = new_node.right.node
 self.node = new_node
 old_node.left.node = new_left_sub
 new_node.right.node = old_node
```

## Resources

- [Binary Search Tree](#)
- [Tree Rotations](#)

# Performance Boosts of Using a B-Tree

## Concepts

- A B-Tree is a sorted and balanced tree that contains nodes with multiple keys and children.
- An index is a data structure that contains a key and a direct reference to a row of data.
- The degree of a B-Tree is a property of the tree designed to bound the number of keys in a tree.
  - The minimum degree must be greater than or equal to two.
- The maximum number of children we can have per node is  $2t$  where  $t$  is the degree of the tree. We call this property the order of the tree.
- The height of the B-Tree is given by the equation  $\log_m(n)=h$ , where  $m$  is order of the tree,  $n$  is the number of entries, and  $h$  is the height of the tree.
- The time complexity for inserting data into a B-Tree is  $O(\log_m(n))$ .

## Syntax

- Retrieving a row of data from a file:

```
import linecache

row = linecache.getline(file_name, line_number)

print(row)
```

- Creating a simple B-Tree and inserting into a node:

```
class Node:

 def __init__(self, keys=None, children=None):
 self.keys = keys or []
 self.children = children or []

 def is_leaf(self):
 return len(self.children) == 0

 def __repr__(self):
 # Helpful method to keep track of Node keys.

 Return "{}.format(self.keys)

class BTtree:

 def __init__(self, t):
 self.t = t
 self.root = None
```

```
def insert(self, key):
 self.insert_non_full(self.root, key)
```

- Searching a B-Tree:

```
class BTree(BaseBTree):

 def search(self, node, term):

 if not self.root:

 return False

 index = 0

 for key in node.keys:

 if key == term:

 return True

 if term > key:

 index += 1

 if node.is_leaf():

 return False

 return self.search(node.children[index], term)
```

## Resources

- [B-Tree](#)
- [Degree and order of a tree](#)

# Performance Boosts of Using a B-Tree II

## Concepts

- Time complexity for loading in data into a B-Tree is  $O(n \log(n))$ .
- The pickle module allows us to save a B-Tree model and then load it every time we want to run a range query.
- The pickle module serializes Python objects into a series of bytes and then writes it out into a Python readable format (but not human readable).

## Syntax

- Creating a simple B-Tree:

```
class Node:

 def __init__(self, keys=None, children=None):
 self.keys = keys or []
 self.children = children or []

 def is_leaf(self):
 return len(self.children) == 0

 def __repr__(self):
 # Helpful method to keep track of Node keys.
 return "{}.format(self.keys)

class BTTree:

 def __init__(self, t):
 self.t = t
 self.root = None

 def insert_multiple(self, keys):
 for key in keys:
 self.insert(key)
```

- Loading the contents of a csv into a B-Tree:

```
with open('amounts.csv', 'r') as f:
 reader = csv.reader(f)
 next(reader)

 values = [float(l[0]) for l in reader]
```

```
btree = BTree(5)
btree.insert_multiple(values)
```

- Using the pickle module to save and load a B-tree:

```
import pickle

Save the model.

with open('btree_example.pickle', 'wb') as f:
 pickle.dump(btree, f)

Load the model.

with open('btree_example.pickle', 'rb') as f:
 new_btree = pickle.load(f)
```

## Resources

- [Pickle module](#)

# Data Pipelines

## Functional Programming

### Concepts

- A data pipeline is a sequence of tasks. Each task takes in input, and then returns an output that is used in the next task.
- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- The benefit of using pure functions over impure functions is the reduction of side effects. Side effects occur when there are changes performed within a function's operation that are outside its scope.
- We can use a lambda expression instead of the `def` syntax.
- Functions that allow for the ability to pass in functions as arguments are called first-class functions.

### Syntax

- Using both pure and impure functions:

```
Create a global variable `A`.

A = 5

def impure_sum(b):
 # Adds two numbers, but uses the
 # global `A` variable.

 return b + A

def pure_sum(a, b):
 # Adds two numbers, using
 # ONLY the local function inputs.

 return a + b
```

- Using a lambda function:

```
new_add = lambda a, b: a + b
```

- Mapping a function to every element in a list:

```
values = [1, 2, 3, 4, 5]
```

- ```
add_10 = list(map(lambda x: x + 10, values))
```
- Filtering elements in a list:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even = list(filter(lambda x: x % 2 == 0, values))
```
- Using reduce to sum elements in a list:

```
values = [1, 2, 3, 4]
summed = reduce(lambda a, b: a + b, values)
```
- Using list comprehensions instead of map and filter:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Map
add_10 = [x + 10 for x in values]

# Filter
even = [x for x in values if x % 2 == 0]
```
- Returning a new function with the default inputs:

```
def(a, b):
    return a + b

add_two = partial(add, 2)
add_ten = partial(add, 10)
```
- Creating a composition of functions:

```
def add_two(x):
    return x + 2

def multiply_by_four(x):
    return x * 4

def subtract_seven(x):
    return x - 7

composed = compose(
    add_two # + 2
    multiply_by_four, # * 4
    subtract_seven, # - 7
)
```

Resources

- [Object-Oriented Programming vs. Functional Programming](#)
- [Functools Module](#)

Pipeline Tasks

Concepts

- File streaming works by breaking a file into small sections, and then loaded one at a time into memory.
- A generator is an iterable object that is created from a generator function.
- A generator differs from a regular function in two important ways:
 - A generator uses **yield** instead of **return**.
 - Local variables are kept in memory until the generator completes.
- The **yield** expression:
 - Lets the Python interpreter know that the function is a generator.
 - Suspends the function execution, keeping the local variables in memory until the next call.
- Once the final **yield** in the generator is executed, the generator will have exhausted all of its elements.

Syntax

- Calculating the squares of each number using a generator:

```
def squares(N):  
    for i in range(N):  
        yield i * i
```

- Fetching the next element in an iterable:

```
next(iterable)
```

- Turning a list comprehension into a generator comprehension:

```
# list comprehension  
  
squared_list = [i * i for i in range(20)]  
  
# generator comprehension  
  
squared_gen = (i * i for i in range(20))
```

- Writing to a file using the csv module:

```
import csv  
  
rows = [('a', 'b', 'c'), ('a1', 'b1', 'c1')]  
  
# Open file with read and write permissions.
```

```
file = open('example_file.csv', 'r+')
writer = csv.writer(file, delimiter=',')
writer.writerows(rows)

• Combining iterables:

import itertools
import random

numbers = [1, 2]
letters = ('a', 'b')

# Random number generator.

randoms = (random.random() for _ in range(2))

for ele in itertools.chain(numbers, letters, randoms):
    print(ele)
```

Resources

- [itertools module](#)

Building a Pipeline Class

Concepts

- An inner function is a function within a function. The benefit of these inner functions is that they are encapsulated in the scope of the parent function.
- A closure is defined by an inner function that has access to its parent's variables. We can pass any number of arguments from the parent function down to the inner function using the * character.
- A decorator is a Python callable object that modifies the behavior of any function, method, or class.
- The **StringIO** object mimics a file-like object that keeps a file-like object in memory.

Syntax

- Adding an arbitrary number of arguments:

```
def add(*args):  
    parent_args = args  
  
    def inner(*inner_args):  
        return sum(parent_args + inner_args)  
  
    return inner  
  
add_nine = add(1, 3, 5)  
  
print(add_nine(2, 4, 6))  
# prints 21
```

- Using a decorator:

```
def logger(func):  
    def inner(*args):  
        print('Calling function: {}'.format(func.__name__))  
        print('With args: {}'.format(args))  
        return func(*args)  
  
    return inner  
  
@logger  
  
def add(a, b):  
    return a + b  
  
print(add(1, 2))
```

```
# 'Calling function: add'  
# 'With args: (1, 2)'  
# 3
```

Resources

- [io module](#)
- [A guide to Python's function decorators](#)

Multiple Dependency Pipeline

Concepts

- A pipeline that handles multiple branching is called a Directed Acyclic Graph (DAG).
- Breaking down the terminology of a DAG:
 - Graph: A data structure that is composed of vertices and edges.
 - Directed: Each edge of a vertex points only in one direction.
 - Acyclic: The graph does not have any cycles, meaning that it cannot point to a vertex more than once.
- When using a DAG, we can implement task scheduling in linear time, $O(V+E)$, where V and E are the numbers of vertcies and edges.
- The time complexity for finding the longest path is $O(n^2)$ and $O(n\log n)$ for sorting by the longest paths.
- The number of in-degrees is what makes the root node different than any other node.
 - The number of in-degrees is the total count of edges pointing toward the node.
 - Each root node will always have zero in-degrees.
- A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v , u comes before v in the ordering.

Syntax

- Building a DAG class:

```
class DAG:  
    def __init__(self):  
        self.root = Vertex()  
  
class Vertex:  
    def __init__(self):  
        self.to = []  
        self.data = None
```

- Integrating a DAG into a pipeline:

```
class Pipeline:  
    def __init__(self):  
        self.tasks = DAG()
```

```

def task(self, depends_on=None):
    def inner(f):
        self.tasks.add(f)
        if depends_on:
            self.tasks.add(depends_on, f)
        return f
    return inner

def run(self):
    scheduled = self.tasks.sort()
    completed = {}
    for task in scheduled:
        for node, values in self.tasks.graph.items():
            if task in values:
                completed[task] = task(completed[node])
        if task not in completed:
            completed[task] = task()
    return completed

```

Resources

- [Deque module](#)
- [Kahn's Algorithm](#)