Data science class note

Blog titles

There are seven blog posts that we'll ask you to complete. That is a graduation requirement - without the seven required blog posts, you won't be able to graduate from the program. Please feel free to write more posts, although in general a "reasonable quantity and great quality" is probably the best strategy so you can have 1-2 posts that you can point potential employers to that really show how you can write for both technical and non-technical audiences. The risk of too many posts is that employers will click onto some of the weaker ones if you didn't have the time to make them all great.

Here are the seven required blog topics:

1. Why did you decide to learn data science?
2. Write a post on a visualization technique of your choice (line plot, histogram, heat-map, etc). How is it being used to answer analytical problems and what are it’s strengths and weaknesses?
3. Write about a topic you are finding particularly challenging. Do it in the form of a tutorial to help another aspiring data scientist to learn that topic.
4. Write a tutorial (with a data set and code sample) on something that we did not cover during the course that you think might be interesting to other people taking the course. It can be a topic we didn’t cover at all, or can just go deeper into a topic that we did cover.
5. Write about your experience with model selection, model validation and hyper-parameter tuning for a data set you’ve worked with on the program. Include hints and tips that would help another data science student do a better job of model selection, validation and tuning.
6. Pick a data science paper written in the last 18 months and rewrite it to explain it to a non-technical business stakeholder. Focus on why it’s important and why it would matter to them.
7. Pick one of the most influential papers in data science (ask us for a list!) and rewrite it to explain it to a non-technical business stakeholder. Focus on why it’s important and why it would matter to them.

These posts should be written in order. The first post requires no experience in data science. The second one assumes you've completed the first couple of sections in module 1, question 5 is one you won't be ready to write about until the end of module 3 and you won't want to dig into the last couple until you're in mod 4 or even in project mode in module 5.

In terms of length, there isn't a set requirement. That said, it'd be hard to do justice to most of these topics (except perhaps the first one) in under 800-1000 words. Equally, you should try to make it a posting - not a novel. If you're blowing through 3000 or 4000 words, it's going to be hard to get someone to read the post, so you might want to constrain the scope so it can be fitted into (say) 1000-3000 words (yes, 3500 words is fine if you need it!)

**How to Blog**

If you already have a blog for work related topics, feel free just to add your posts to that. If you have a personal blog, you might want to set up a separate "work" blog. It's not required, but do consider that employers will probably look at multiple posts from any blog you share, so if you mix personal and business postings, make sure they're equally high quality and would reflect well on you when being considered for a job. Finally, if you've never blogged before, not to worry! Pick your favorite blogging site/software and start blogging. Don't worry too much about which platform you pick - just pick something and focus on writing great content.

What data scientists do

## 1. How much or how many - Regression analysis

Regression analysis is used to predict a continuous value - such as the number of staff you'll need for a busy shift or the likely sale price of a house. Regression is a handy technique to forecast sales, monitor marketing campaigns and create future plans etc. Traditional trend analysis only looks at how one business entity changes with respect to another. Regression analyses can provide insight into how an outcome will change when several other variables are modified.

## 2. Which category - Classification analysis

Classification analysis is like a regression since it allows you to predict something. The difference is that a classifier is used to predict which category something will fall into. If you're trying to figure out whether a client is likely to default on a loan (a binary classifier - default or no default) or which of your products a customer is likely to prefer, you're going to need a classifier.

## 3. Is this weird? - Anomaly detection[¶](https://nyc-56-55642.ide-proxy.ide.learn.co/notebooks/dsc-problems-data-science-can-solve-online-ds-ft-061019/index.ipynb#3.-Is-this-weird?---Anomaly-detection)

Anomaly detection is a data science technique used to find unusual patterns that do not conform to expected behavior. Anomaly detection is a common analysis technique. It has many applications in such businesses, from intrusion detection (identifying strange patterns in network traffic that could signal a hack) to fraud detection in credit card transactions to fault detection in operating environments.

## 4. Which option should be taken? - Recommender Systems

Recommender systems are one of the most popular applications of data science today. They are used to predict user preferences towards a product/service. Almost every major tech company (Amazon, Netflix, Google, Facebook) has applied them in some form or the other. You might have noticed phrases like "If you like this product, you may also like ...", "Users who bought this item also bought ..." and "Based on your preferences, we recommend following products to you ...". You got it, these are all recommender systems in action.

Recommender systems can help a business retain customers by providing them with tailored suggestions specific to their needs. They can help increase sales and create brand loyalty through relevant personalization. When a customer feels as though they are understood by your brand, they are more likely to stay loyal and continue purchasing through your site. According to a recent study by McKinsey, up to 75% of what consumers watch on Netflix comes from the company’s recommender system. Retail giant Amazon credits recommender systems with 35% of their revenue. Best Buy decided to focus on their online sales, and in 2016’s second quarter they reported a 23.7% increase, thanks in part to their recommender system.

Data science life cycle

Business understanding—data mining--data claeaning--data exploration--feature engineering—predictive modeling—data visualization.

Business understanding/ Domain knowledge

Some of the questions that the data scientist might be asked to answer could include:

* How much or how many? E.g. Identifying the number of new customers likely to join your company in the next quarter. (Regression analysis)
* Which category? E.g. Assigning a document to a given category for a document management system. (Classification analysis)
* Which group? E.g. Creating a number of groups (segments) of your customers based on their monetary value. (Clustering)
* Is this weird? E.g. Detecting suspicious activities of customers by a credit card company to identify potential fraud. (Anomaly detection)
* Which option should be taken? E.g. Recommending new products (such as movies, books or music) to existing customers (Recommendation systems)

**Data Mining**

After agreeing with the analytical question(s), identifying the objective for your analysis, the next stage of analysis is to identify and gather the required data.

Data mining is a process of identifying and collecting data of interest from different sources - databases, text files, APIs, the Internet, and even printed documents. Some of the questions that you may ask yourself at this stage are:

* What data items do I need in order to answer my analytical question?
* Where can I find this data?
* How can I obtain the data from the data source?
* How do I sample from this data?
* Are there any privacy/ legal issues that I must consider prior to data usage?

**Data Cleaning**

Data cleaning and wrangling is usually the most time-consuming stage within the data science process. This stage may take up to 50-80% of a data scientist's time as there are a vast number of possible problems that make the data "dirty" and unsuitable for analysis. Some of the problems you may see in data are as follows:

* Inconsistencies in data
* Mis-spelled text data
* Outliers
* Imbalanced data
* Invalid/outdated data
* Missing data

The data cleaning stage requires the development of a strategy on how to deal with these issues. Such a strategy may vary substantially between different analysis experiments depending on the nature of problems being solved.

**Data Exploration**

Data exploration or Exploratory Data Analysis is the stage that follows data cleaning. Exploratory analysis help highlight the patterns, relations, variance and bias of available data. Exploratory analysis may involve the following activities:

* Selecting a subset of a bigger dataset for exploration
* Calculating basic descriptive statistics such as the mean, the median, and the mode
* Creating a range of plots including histograms, scatter plots and distribution curves to identify trends in the data
* Other interactive visualization with filtering to focus on a specific segment of data

## Feature Engineering

A "Feature" is a measurable attribute of the phenomenon being observed - the number of bedrooms in a house or the weight of a vehicle. Based on the nature of the analytical question asked in the first step, a data scientist may have to engineer additional features not found in the original dataset. Feature engineering is the process of using expert knowledge to transform raw data into meaningful features that directly address the problem you are trying to solve. For example, taking weight and height to calculate Body Mass Index for the individuals in the dataset. This stage will substantially influence the accuracy of the predictive model you construct in the next stage.

## Data Visualization

After deriving the required results from a statistical model, visualizations are normally used to summarize and present the findings of the analysis process in a form which is easily understandable by non-technical decision makers.

Data visualization could be thought of as an evolution of visual communication techniques as it deals with the visual representation of data. There are a wide range of different data visualization techniques, from bar graphs, line graphs and scatter plots to alluvial diagrams and spatio-temporal visualizations, each of which will work better for presenting certain types of information.

## Cloning Learn.co lessons to your local hard drive

One of the many useful features of the command line will be using git to clone (download) a local copy of the curriculum hosted on learn.co. This will allow you to work offline and to save changes as you work through exercises and start programming!

To start, **for Mac users, open the terminal application**, **for Windows, open “Git Bash”**.

## pwd

The first command to try out is **pwd** which stands for **print working directory**. This will tell you where you currently are in the computer's directory structure. Try it out.

## cd

The next essential command is **cd** which stands for **change directory**. This allows you to navigate to different folders on your computer's hard drive. Typing cd by itself will automatically take you to your home directory. Typing cd and a folder name will take you to that folder. Typing cd .. will move you one folder up in the hierarchy. Play around and trying moving between folders for a minute or two.

## Tab Completion

An extraordinarly useful feature when working on the command line is tab completion. This allows you to hit the tab button to autocomplete names once you have made a unique specification.

For example, if you navigate to your root directory by running the command **cd**, if you're on a Mac, you will probably have 2 folders within your root directory named "Downloads" and "Documents" (these are standard folder names created by default in most systems, although you may have renamed them, or your system may be different). With these, or longer folder names, it can sometimes become cumbersome to type the full folder name. instead, you can start typing the command and folder name such as **"cd Dow"** and then press **tab** to autocomplete. Like magic, the command line should complete the statement correctly to be **cd Downloads**. (Note: this will not work if you have another folder that begins with "Dow". Similarly, if you just typed **cd D** or **cd Do** followed by **tab**, the command line will not autocomplete, as the selection is not unique, because **D** or **Do** could both refer to either **Documents** or **Downloads**. Also note that these commands are case sensitive, and folder capitalization much be matched exactly.

**ls**

Continuing with navigating the computer's hard drive, it's useful to know how to **list files**. This is done with the **ls** command, short for list.

You can also pass optional parameters to ls such as **ls -a** which list **all files** (including hidden files), **ls -l** which will give a **long listing** of files (including file size and last edit times) or you can pass multiple parameters simultaneously such as **ls -al** to produce a detailed listing of all files.

**\* the wildcard parameter**

Also very useful is the wildcard parameter. For example, if you wanted to list all files in the current working directory that begin with a, you could type **ls a\***. Here, the asterix (\*) denotes anything is allowed following the a. Similarly, to list all pdf files in the current working directory we could type **ls \*.pdf**, or to list all text files, we could type **ls \*.txt**.

**mkdir**

## Finally, as you continue to navigate the file directory from the command line it can be useful to be able to create new folders. To do this, use the mkdir command, which stands for make directory. Try it out with mkdir NewFolderName. Afterward, use the ls command to see that there is indeed a new folder, and if you wish, move into the new folder using the cd command.

## Cloning GitHub repositories and Learn.co lessons

Now that you can navigate the file directory using the command line, you're ready to download some course materials from the web to your local environment.

* Create a folder on your computer for your course materials and navigate into it.
* Then create a subfolder titled "section01" and navigate into that.
* Return to your web browser and navigate to the lesson you want to download.
* Click the GitHub icon

**Click the fork button**, as shown in order to create a copy to your personal account which you can edit and update.

## Finally:

* Press **cmd+L** to highlight the url bar and **cmd+c** to copy the url (whenever we use cmd to refer to holding down the command key on a Mac computer, on a Windows computer, hold down the control key instead)
* Return to the terminal (you should be in your "section01" folder)
* Type: **git clone** and paste your repo url (**cmd + v** for Mac; for Windows, in git bash, **shift + insert**)
* Et Voila! The repository and all of its contents will be downloaded locally to your computer!
* Remember that we then need to **cd** into the new repository once we've downloaded it
* Our next step is to open our Jupyter Notebook locally (**not on learn**) using the command line

## Jupyter Notebook

So, now that we understand a bit about how to use our terminal and how to clone GitHub repositories from our learn lessons, we should talk about the Jupyter Notebooks that will run most of the content in this course.

Make sure to activate your conda virtual environment in your terminal first by typing source activate learn-env. Then type jupyter notebook in your command line and press enter. Next, your default browser will open a new window or tab and you will see the list of files that are in your current directory (remember we want to be in the GitHub repo directory that we just downloaded).

**Note:** To stop a Jupyter Notebook, go to your command line where the notebook is running and press the ***control*** key + the letter ***C*** (**ctrl**+**c**).

Second step is to click on the index.ipynb file which is the Jupyter Notebook we will be using in this and future labs and lessons. This will open a new tab where we will see the same content from learn!

### Background on Jupyter

Jupyter is a web application that allows someone to create and work with documents that have live code. It's a very popular tool among data scientists, as it allows for both explanations of thinking behind code as well as the code itself.

### Introduction to cells

The notebook itself consists of cells. Double click on this content to see what I mean. Once we double click on a cell, we are in insert mode. This means that we are able to edit the cells, just as you would if this were a word document. We can tell that we are in insert mode because of the green border around the cell.

After entering insert mode for this cell, change some content. Don't worry about what you change as we can always undo it. We can revert our changes to a cell by making sure that we are still in insert mode and by pressing command + z on a Mac or control + z on Windows.

To get out of insert mode and see the effect of our changes, press shift + enter.

### Adding and Deleting Cells

We have already seen, to alter the contents of a cell we simply double click on that cell, bringing us into insert mode, and then change the contents. Now let's see how to add, and delete cells from escape mode.

#### Adding cells

If we wish to quickly add a new cell we can do so with the following steps:

* Make sure we are not in insert mode, but in escape mode
  + Remember we can tell we are in insert mode when we have a green border around our cell.
  + To get out of insert mode and into escape mode, press shift + enter. Another option is to press the escape key.
  + You will no longer see a cell bordered in green.
* Then press the letter b to create a new cell.

#### Deleting cells

To delete a cell we once again should be in escape mode, and then press the x key.

Of course, we'll want a way to undo our deletion. From escape mode, you can press z to undo deletion of a cell. Note that this is different from cmd + z. Pressing cmd + z while in insert mode undoes any changes inside of a cell while, whether these changes be deletions or text insertions. Pressing z from escape mode undoes the deletion of a cell.

Go to escape mode and press x. This cell disappears!

Then bring it back with z.

### types of Cells

The current cell and every other cell in this lesson has been a markdown cell, meaning that it allows us to write text and stylize that text. For example, if you surround some text with two asterisks (\*\*) on both sides, the text **becomes bold**. That's markdown.

Cells can also have a type of code. If we are writing in a cell that is for Python code, everything in that cell must be valid Python or we will see an error.

So, a cell must either be of type markdown or of type code, in which case all of the contents must be valid Python. It cannot be both. We can quickly change a cell from markdown to code with some keyboard shortcuts.

From escape mode, we change a cell to type code by pressing the letter y. From escape mode, we change a cell to type markdown by pressing the letter m.

Anytime we create a new cell, say with the shortcut key b, the new cell will default to code mode. We can switch to escape mode and press the letter m to change the cell from code to markdown.

#### View All Shortcuts

Press the key h while in escape mode to view the menu for all of Jupyter's shortcuts.

### Working with Python in Jupyter

Ok, now that we know a little bit about adding and deleting cells, as well as changing cell types from markdown to code, let's focus on working with Python in Jupyter. We'll go into a large amount of detail about working with a Jupyter Notebook in Python, but the main takeaway is this: if we see a Python cell, we should press shift + enter on that cell.

The major gotcha in working with Python code is that we must execute the cells in order for Python to register the code in them. So for example, just seeing the cell where we define name to 'bob' below does not write that cell to memory.

## Setup to clone

Remember from the last lesson, you need to:

1. Click on the "GitHub" logo in the top right of the learn.co page
2. Fork the repository on GitHub so you have your own copy there
3. Copy the URL of your repository
4. Open a terminal window (terminal on a Mac, Git Bash on Windows)
5. Make sure to activate your conda virtual environment so you have the right version of Python and all of the necessary packages. On a mac or in Git Bash on Windows, type source activate learn-env. (If you have to use the conda shell on windows, type
6. activate learn-env instead).
7. Clone (download) the files to your hard drive by typing git clone and then pasting the URL of your repo you saved in step 3.
8. Type cd followed by the name of the directory you just created (running the ls command will show you the name of the directory you downloaded)
9. Run the jupyter notebook command to start up Jupyter, and in the browser window that opens, navigate to and click on the index.ipynb notebook.

Import pandas

*import pandas as pd*

*import matplotlib.pyplot as plt*

*%matplotlib inline*

## Command Versus Edit Mode

You should also start to notice that when you are in a cell writing code (or notes), the cell is highlighted in **green** meaning you are in **edit mode**.

Alternatively, if you **press esc**, the cursor will be in **blue** inidicating that you are in **command mode**.

### Edit Mode

Edit mode is the standard mode for editing cells, whether its writing code or notes. To enter edit mode from command mode simply hit enter, or double click on a cell.

### Command Mode

In command mode, you can delete cells, add cells, copy cells, paste cells, change cell types, and more. You can also do these tasks in a more cumbersome (and time consuming) manner by using the various headers in the menu bar at top.

*First coding*

## Variables

The other thing that happened in our block of code above was that we defined a **variable**.

This happened in the first line of code:  
df = pd.read\_csv('lego\_sets.csv')

As we saw, we used the built in read\_csv method from the pandas package which we imported under the alias pd.

The output of this method was then assigned to the variable df. This is the standard syntax for declaring any variable. You do not have to specify variable types, as in many other programming languages. Simply:

variable\_name = what\_to\_stor\_in\_the\_variable

You can also check what type of object something is using the built in **type()**method. This can be useful when determining how to work with an object that you are unfamiliar with.

## Common DataFrame Methods

As you can see, our variable **df** is a DataFrame object (which is part of the Pandas core package). Here's some other common methods you will want to become familiar with when working with Pandas dataframes:

* df.head()
  + Preview the first 5 rows of a dataframe. Pass a number for more/less rows
* df.tail(10)
  + Preview last 10 rows (default 5 if no number given)
* df.info()
  + Returns column names and details about each column
* df.columns
  + Return column names. Note that there is no parentheses for this. This is becuase this is an attribute as opposed to a method

## Pandas Series

While the entire spreadsheet is called a **dataframe**, each individual column is known as a **series**. You can access a specific column of a pandas dataframe one of two ways:

df['col\_name']

or

df.col\_name

First note that in df['col\_name'] we need 'quotations' around the column name. The quotations denote the column name is a **string**, python's built in variable type for storing text. This can alternatively be replaced with double quotes df["col\_name"]. In general, anything in quotations is a string in python code. Occassionally, with very ill formatted column names with quotations in the names themselves, you may even need to wrap a name in triple quotes df["""col\_name"""] . This will rarely happen in this particular context, but is also the general pattern for dealing with messy strings.

Note that the second method, df.col\_name, will only work if there are no spaces within the name of the column. Similarly to tab completion with the command line, this is a primary reason why programers use dashes (-) and underscores (\_) in lieu of whitespace in their variable and file names. Also note that no quotations are used when using this format. (The column names have been stored as attributes to the DataFrame object!)

Eg

df.head()

this give us the first five rows

df.ages.head()

this give us the first five rows of the column name age which is similar to df.age[:5] ## List and Series Slices

Above, we introduced an entirely new programming pattern called a slice.

The syntax for a slice is `[start:end]`.

You can also pass an additional third parameter `[start:end:count\_by]` which will allow you to:

count every other: `[start:end:2]`

count backwards: `start:end:-1]`

or potentially much more cryptic patterns, depending on what you pass.

While we could have also used `df.State.head()`, slicing works for many more datatypes. This includes the previously mentioned \*strings\* as well as \*\*lists\*\* and other iterable objects. \*\*Series\*\*, the columns of the pandas DataFrame are similar to python's built in \*lists\*, but also have additional \*methods\* built in to them that we will continue to investigate.

## List and Series Slices

Above, we introduced an entirely new programming pattern called a slice.

The syntax for a slice is `[start:end]`.

You can also pass an additional third parameter `[start:end:count\_by]` which will allow you to:

count every other: `[start:end:2]`

count backwards: `start:end:-1]`

or potentially much more cryptic patterns, depending on what you pass.

While we could have also used `df.State.head()`, slicing works for many more datatypes. This includes the previously mentioned \*strings\* as well as \*\*lists\*\* and other iterable objects. \*\*Series\*\*, the columns of the pandas DataFrame are similar to python's built in \*lists\*, but also have additional \*methods\* built in to them that we will continue to investigate.

## Common Series Methods

Some very useful series methods include:

* df.col\_name.value\_counts()
* df.col\_name.astype()
* series.mean() #Changing notation here: series refers to df.col\_name (which is a series)!
* series.median()
* series.min()
* series.max()
* series.std()
* series.unique()
* series.nunique()
* series.sample()
* series.sort\_values()

There are also many more but this is a very good starting point. Start pulling up some docstrings (using shift+tab within parentheses!) and getting acquainted with reading them yourself before we go through a few of these methods together.

## Graphing

Our bread and butter tool for graphing in python will be **matplotlib**. There are a number of other packages you can also check out for visualization later on such as plotly, folium, and bokeh to name a few, but matplotlib is an industry standard used widely. As with the pandas package, we must import the matplotlib package to use its built in functions and methods. Rather then import the entire package however, we import only the piece we intend to use (hence the dot notation) and again alias this under the shorthand plt. Finally, the iPython magic command **% matplotlib inline** makes these graphs appear within our jupyter notebook.

#import a subset of the matplotlib package under the alias 'plt'

import matplotlib.pyplot as plt

#ipython magic command for displaying graphs within the notebook

%matplotlib inline # df is the name of the file

to\_graph = df.theme\_name.value\_counts()[:5] # assign one of the col of the file df to to\_graph

to\_graph.plot(kind='barh') # horizontal bar graph

## Adding labels

The graph above is a good start, but we should be sure to add some labels! To do this we make successive calls to the plt package we imported. Some common methods you should be familiar with include:

* plt.title()
* plt.xlabel()
* plt.ylabel()
* plt.legend()
* plt.xticks()
* plt.yticks()

to\_graph = df.theme\_name.value\_counts()[:5]

to\_graph.plot(kind='barh') #lots of other optional parameters can be passed such as color

plt.title('Top 5 Lego Themes', fontsize=16) #fontsize is optional

plt.xlabel('Number of Lego Sets') #you could also pass in fontsize if you wanted here

plt.ylabel('Theme') #you could also rotate text if you wanted

assigning variables

variable=data

## Declaring variables without assignment

Sometimes we wish to declare a variable without assigning it to data. In Python, that's a little tricky to do. As we just saw with name, declaring variables without assignment throws an error. Thankfully, Python has a special type for us that represents nothing at all. None is a data type in Python that represents nothing. So, if we do not know the type of a variable and want to have the data to the variable be assigned later, we can assign that variable to None.

example

address = None

Notice that address is now assigned, but it is assigned to None.

"Send an email to " + full\_name + " at " + email + " to say how nice it was meeting yesterday."

## What are Strings?

A lot of information in the world is in the form of text. To capture this information and operate on it in Python we take this text and make it into the **String** (str) data type.

**Note:** double quotes and single quotes can be used interchangeably in Python; however, for readability it is important that we stay consistent.

Below, we have the name of a cartoon character, Homer Simpson. By putting quotes ("") or ('') around the name, we create a string.

"Homer Simpson"

When programmers say *string*, what they mean is text. When programmers say *data type*, they just mean type of data. We can think of 'Homer Simpson' as an instance of the string data type.

Here are a few other types of data in Python that we will talk more about in later lessons:

100 *# Integer*

10.0 *# Float*

**True** *# Boolean*

Since there are several types of data in Python we can discover the type of any piece of data by calling, or executing, the type() function. By calling or executing a function, we mean running the function so that it executes the code within it.

A string is just text. We indicate to Python that we are writing a string by surrounding our content with quotation marks. Once we do this, we can operate on this string by calling methods like upper or endswith. We identified a general pattern for calling methods on datatypes: 'instance of a datatype-dot-method name-parentheses'.

The second thing we learned was different mechanisms for learning about methods. We saw the importance of guessing and experimentation, and how doing so can give us error messages, which provide clues. We also saw how to ask questions about a datatype by calling 'help' followed by the datatype name like help(str). Finally, we saw we can ask Google. This mechanism of exploration is a skill we'll build up over time and this course will provide guidance and practice on along the way.

example

print(name) name = "Bart Simpson"

name = name.replace('Simpson', 'Flanders') # this replace the name Simpson by Falnders

print(name)

## Summary

In this section, we introduced two new types of data: numbers and booleans. We saw that numbers allow us to perform standard math operations, and we saw that booleans answer whether something is True or False, and serve as a way our program or different methods can respond to questions.

We have seen almost all of our Python datatypes, we talked about how to choose a datatype, and we talked about how to switch between datatypes. We said that we choose a datatype based on the capabilities that we want to give to that data: should it answer whether it is larger or smaller, or does it make sense to capitalize? The goal of this discussion is to begin thinking about why we decide to put data in specific types (i.e. string, number, boolean). We also introduced coercion methods like bool and str that switch between datatypes.

let's create a string that uses both of these variables to tell us how many weeks we will be traveling in our travel month. The string should read "I will be travelling 5 weeks starting in the month of January". Interpolate the num\_of\_weeks and travel\_month to get the correct string.

**Remember:** We can interpolate strings in the following ways:

* "Start of string" + variable\_to\_interpolate1 + "middle" + variable\_to\_interpolate2 + "end of string"
* "Start of string {variable1} middle {variable2} end of string".format(variable1=variable\_to\_interpolate, variable2=variable\_to\_interpolate)
* f"Start of string {variable\_to\_interpolate1} middle {variable\_to\_interpolate2} end of string"

example

number\_of\_week=5

travel\_month=January

travelling\_schedule = f"I will be travelling {number\_of\_weeks} weeks starting in the month of {travel\_month}"

travelling\_schedule

conditional

# reset variable to 0

vacation\_days = 0

# this does the same as above

vacation\_days += 1

vacation\_days += 1

# print how many vacation days there are

print(vacation\_days)

A block is any code that is grouped together. With conditionals, we indicate that something is part of the block by indenting. So the line vacation\_days += 1 is indented to ensure that it is run as a part of the conditional argument below. To end the block we simply stop indenting.

vacation\_days = 1

if False:

# code does not run since conditional argument is False

vacation\_days += 1

print("vacation\_days = ", vacation\_days)

print("we incremented vacation days")

elif True:

print("We are now in our elif statement!")

print("This means that we exceeded our goals this quarter")

print("We will increase our vacation days by two")

vacation\_days += 2

print("vacation\_days = ", vacation\_days)

else:

print("vacation\_days = ", vacation\_days)

print("we did NOT increment vacation days")

Boolean values (True and False) can also be used in mathematical equations. True is set to 1 and False is set to 0.

Eg

True + 5 + True =7

True-False-False+True=2

## Summary

In this lesson, we saw how conditionals allow us to make decisions with our code by only executing code under the if statement when the conditional argument is True or truthy. We then saw how we can use the else statement to only run code when the conditional argument is False or falsy, and as we know, code that is not in a conditional block is still run as normal. Next We examined what is truthy or falsy, and saw that None, 0, empty strings and lists are all falsy. If we are unsure, we can use the bool function to see a the boolean value of a piece of data. Finally, we brought it all together and used if, elif, and elsestatements together to make decisions based on the conditions of our problem.

## What Are Lists?

A list is our first form of a collection. A collection is just a way of grouping multiple pieces of data together. For example, let's consider the top cities for travel according to the magazine Travel and Leisure. Here is how we usually see a list of travel locations in a document or on a website.

We indicate that we are initializing a list with an opening bracket, [, and we end the list with a closing bracket ]. We separate each list item, also called an element, with a comma.

Slicing: accessing multiple elements

Eg

top\_two = top\_travel\_cities[0:2]

In programming terms, we would say that slicing elements is non-destructive, because it does not change the underlying data structure. We can do it as many times as we like, and our top\_travel\_cities array remains unchanged. If we wish to store that slice of elements, we can store it in another variable.

.append() to add on the list

append is destructive. That is, it changes our underlying data structure. Every time we execute the append method, another element is added to our list. Now what if we accidentally add 'San Antonio' a second time to our list.

.pop()

Listname.pop

To remove the last item

Set (name of list) to make the list unique(no repetition)

A set is just like a list, except elements do not have order and each element appears just once.we can change to a list by assigning as a list eg

unique\_travel\_cities = list(unique\_travel\_cities)

len

## Summary[¶](https://nyc-54-53787.ide-proxy.ide.learn.co/notebooks/dsc-lists-online-ds-ft-061019/index.ipynb#Summary)

In this section we saw how to associate data together in a collection, called a list. A list is similar to a list in the real world - it implies the data has some connection, and that it has an order to it. We initialize a list with the brackets, [], and separate each element by a comma. To access elements from a list, we use the bracket accessor followed by the index of the element we want to retrieve, and our indices begin at zero and increase from there. To add a new element to the end of the list we use the append method, and to remove an element from the end of a list we use pop. We can change elements anywhere between by first accessing the elements and then reassigning them.

# Working With Dictionaries

## Introduction

After introducing and working with Lists, you might be wondering if there are other kinds of collections in Python that we should know about. Well, there are! In this lesson, we will introduce **dictionaries**. As we know, lists represent a collection of information that is ordered, like a list of the most watched TV shows. However, in different situations, we may want our data to represent attributes of an entity, such as the various attributes of a single TV show like its name, genre, starring actors, etc. For scenarios where the stored objects have no definitive order, but need to be retrieved, a **dictionary** is more natural. Dictionaries are unordered collections of key-value pairs. Rather then specifying a positional index as with lists, we specify a key for a dictionary and are returned with the value associated with that key. For example, in a list, we could retrieve the third item with ExList[2] (remember indexing starts at 0), while in a dictionary, there is no specific third item. Instead, we would have to specify a key such as AcronymnDict['GDPR'] to retrieve the associated value attached to that key. This is similar to traditional dictionaries: you look up a specific word (the key) to find its associated definition (the value).

## Why Use a Dictionary When We Have Lists?

While lists are great, for listing information like we mentioned earlier, they can actually become very messy when we are trying to use them to organize data which is more a bit more complex. Let's look at a brief example of a person.

Every person has a **name**, **age**, **height** (in inches), **weight** (in lbs), and **fav\_lang**. How we would represent a person using a list?

terrance = ["Terrance", 25, "6'00", 165, "Python"]

Now, that looks fine but what do we do if we want to tell someone Terrance's fav programming language? We just have to **remember** that Terrance's favorite programming language comes fifth in his list of information? What if he has more attributes than just the five that are listed (i.e. native\_language, hometown, etc.)? What if his attributes are in a different order than we expected? We can see that this list would easily breakdown and cause more problems than it solves.

However, if we use a dictionary, we can more neatly organize this information and make it easier for us to use as the dictionary grows. Let's see what Terrance's information would look like using a dictionary.

terrance = {'name': "Terrance", 'age': 25, 'weight': 72, 'height': 165, 'fav\_lang': "Python"}

This dictionary definitely has more text in it, but we can see a direct association between the attribute or **key** and its correlated **value** (i.e. {"key": "value"}). This datatype makes it easier to store and access information, such as the attributes of a person or other entity. Note that dictionaries are unordered, so trying to access information using an index number will not work! Accessing information is always done by calling the associated **key**.

Let's take a deeper look at how dictionaries are built and how they work.

To access an element in dictionary friends

Friends[‘name’]

Friends[‘seasons’]=4 this assign a new key element season with value 4

Keys can be numbers eg friends[14]=’some\_value’

To delete the key

del friends[14]

help(dict) to know more methods

accessing list inside dictinary

david = friends['creators'][0] in the friends dict the list creator

As programmers, we tend not to get much smarter over time. Instead, we develop skills for making problems easier to solve. Taking the problem in steps, and checking our work at each of these steps is a technique we should continue to lean on. It's the mark of a skilled developer.

## Summary

In this section, we saw a new type of collection, the dictionary. A dictionary is an unordered collection of key-value pairs. We mark the start and end of a dictionary with curly braces, `{}`, and then follow the pattern of `'key':'value'` for each of the associated attributes, with each attribute separated by a comma: `dictionary = {'key1':'value1', 'key2':'value2'}`.

We retrieve a specific value from a dictionary by using the bracket accessor in combination with the key, so `dictionary['key2]'` returns `'value2'`. We can also add a new attribute with the format `dictionary['key3'] = 'value3'

## Summary

In this lesson, we learned how to use loops to iterate through a collection of elements. We started with iterating through a list of numbers, and performed the same operation on each number. Then we saw how we can loop through the numbers and have each number be used to access a successive element from a separate list, like countries. We then saw that to ensure that our list of numbers matched the indices of a our other list, we had to use the expression, for element in list(range(0, len(list))). Finally, we introduced a naming convention that is commonly used when naming the variable for our loops when iterating over a collection that is a list of common elements (i.e. ice\_cream\_flavorfor a list of ice\_cream\_flavors).

# The Bash Shell

## pwd

The first command to try out is **pwd** which stands for **print working directory**. This will tell you where you currently are in the computer's directory structure. Try it out in terminal or git bash.

## cd

The next essential command is **cd** which stands for **change directory**. This allows you to navigate to different folders on your computer's hard drive. Typing cd by itself will automatically take you to your home directory. Typing cd and a folder name will take you to that folder. Typing cd .. will move you one folder up in the hierarchy. Play around and trying moving around folders for a minute or two.

## Tab completion

An extraordinarily useful feature when working on the command line is tab completion. This allows you to hit the tab button to autocomplete names once you have made a unique specification.

For example, if you navigate to your root directory by running the command **cd**, you will probably have 2 folders within your root directory named "Downloads" and "Documents" (these are standard folder names created by default in most systems, although you may have renamed them, or your system may be different). With these or longer folder names, it can sometimes become cumbersome to type the full folder name. instead, you can start typing the command and folder name such as **"cd Dow"** and then press **tab** to autocomplete. Like magic, the command line should complete the statement correctly to be **cd Downloads**. (Note: this will not work if you have another folder that begins with "Dow". Similarly, if you just typed **cd D** or **cd Do** followed by **tab**, the command line will not autocomplete, as the selection is not unique, because **D** or **Do** could both refer to either **Documents** or **Downloads**. Also note that these commands are case sensitive, and folder capitalization much be matched exactly.

## ls

Continuing with navigating the computer's hard drive, it's useful to know how to **list files**. This is done with the **ls** command, short for list.

You can also pass optional parameters to ls such as **ls -a** which list **all files**(including hidden files), **ls -l** which will give a **long listing** of files (including file size and last edit times) or you can pass multiple parameters simultaneously such as **ls -al** to produce a detailed listing of all files.

## \* the wildcard parameter

Also very useful is the wildcard parameter. For example, if you wanted to list all files in the current working directory that begin with a, you could type **ls a\***. Here, the asterix () denotes anything is allowed following the a. Similarly, to list all pdf files in the current working directory we could type \*\*ls .pdf**, or to list all text files, we could type**ls .txt\*\*.

## mkdir

Finally, as you continue to navigate the file directory from the command line it can be useful to be able to create new folders. To do this, use the **mkdir**command, which stands for **make directory**. Try it out with **mkdir NewFolderName**. Afterward, use the **ls** command to see that there is indeed a new folder, and if you wish, move into the new folder using the **cd** command.

## Additional Resources

* [More Basic Shell Commands](http://www.ks.uiuc.edu/Training/Tutorials/Reference/unixprimer.html)
* [Linux Bash Man Page](https://linux.die.net/man/1/bash)
* [Detailed Bash](https://tiswww.case.edu/php/chet/bash/bashref.html)

## Summary

In this lesson, we looked at some basic bash commands in order to navigate through files and folders on your computer. From here, we're ready to get started with git, an important version control system used by many programmers, developers and data scientists.