

# PYTHON FOUNDATIONS

## FOR DATA ANALYSIS

★★★★★ *With Expert Python Instructor Chris Bruehl*



\*Copyright Maven Analytics, LLC

# COURSE STRUCTURE

---



This is a **project-based** course for students looking for a practical, hands-on, and highly engaging approach to learning Python essentials for data analysis

*Additional resources include:*

- ★ **Downloadable PDF** to serve as a helpful reference when you're offline or on the go
- ★ **Quizzes & Assignments** to test and reinforce key concepts, with step-by-step solutions
- ★ **Interactive demos** to keep you engaged and apply your skills throughout the course

# COURSE OUTLINE

---

1

## Why Python?

Introduce the Python analytics ecosystem and key reasons why it's the programming language of choice for many data analysts

2

## Jupyter Notebooks

Install Anaconda & create your first Jupyter Notebook, a user-friendly Python coding environment designed for data analysis and visualization

3

## Data Types

Introduce native Python data types, common use cases, type conversion methods, and key concepts like iteration and mutability

4

## Variables

Learn how to name and store values in memory using variables, as well as how to overwrite, delete and track them

5

## Numeric Data

Learn how to work with numeric data, and use numeric functions to perform a range of arithmetic operations

6

## Strings

Learn how to manipulate text via indexing and slicing, calculate string lengths, apply various string methods, and print f-strings to include variables

# COURSE OUTLINE

---

7

## Conditional Logic

*Learn how to use IF statements and Boolean operators to establish conditional logic and control the flow of your programs*

8

## Sequence Data Types

*Learn how to create, modify, and nest lists, tuples, and ranges, all of which allow you to store many values within a single variable*

9

## Loops

*Understand the logic behind For and While loops and learn how to refine loop logic and handle common errors*

10

## Dictionaries & Sets

*Address the limitations of working with lists and explore common scenarios for using dictionaries and sets in their place*

11

## Functions

*Learn how to create custom functions in Python to boost productivity, and how to import external functions stored in modules or packages*

12

## Manipulating Excel Sheets

*Import the openpyxl package and manipulate data from an Excel worksheet using the Python skills you've learned throughout the course*

# INTRODUCING THE COURSE PROJECT



## THE SITUATION

You've just been hired as a Data Analytics Intern for **Maven Ski Shop**, the world's #1 store for skis, snowboards, and winter gear. The team is beginning to use Python for data analysis, and needs your help getting up to speed.



## THE ASSIGNMENT

To prepare for scalable growth, the business is transitioning to Python as their primary tool for tracking **inventory**, **pricing**, and **promotions**.

As one of your first tasks as an intern, you've been asked to help analyze sales data from the shop's recent Black Friday promotion.



## THE OBJECTIVES

### Use Python to:

- Process missing data fields
- Reshape and aggregate transactional data
- Calculate KPIs and deliver insights on Black Friday Sales
- Build a simple data pipeline and export the processed data to Excel to share with leadership



**MAVEN**  
SKI SHOP

# SETTING EXPECTATIONS

---



This is **NOT** a Python course for software development

- We'll cover data types, variables, conditional logic, loops, and custom functions applied to common analytics use cases, and won't dive deep into object-oriented programming or full-stack frameworks



We'll focus on **core concepts** necessary for data analysis

- Our goal is to get you familiar with basic Python syntax and programming concepts in order to prepare you to use Python's wide variety of data analysis libraries (which we'll cover in depth in separate courses)



We'll use **Jupyter Notebooks** as our primary coding environment

- Jupyter Notebooks are free to use, and the industry standard for conducting data analysis with Python (we'll introduce Google Colab as an alternative, cloud-based environment as well)



You do **NOT** need prior coding experience to take this course

- We'll start from the beginning by covering essential building blocks and programming concepts, then practice applying them using Python's user-friendly syntax and functionality

# SETTING EXPECTATIONS

---



## Who this is for:

- Analysts or BI professionals looking to learn Python for data analysis
- Students looking to learn the most popular open-source analytics tool
- Anyone who wants to understand the core fundamentals of the Python language and syntax



## Who this is NOT for:

- Experienced Python programmers or advanced users
- Students looking to learn Python for Software or Web Development
- Anyone who would rather copy and paste code or run packages without building the foundational skills

# WHY PYTHON?

# MEET PYTHON

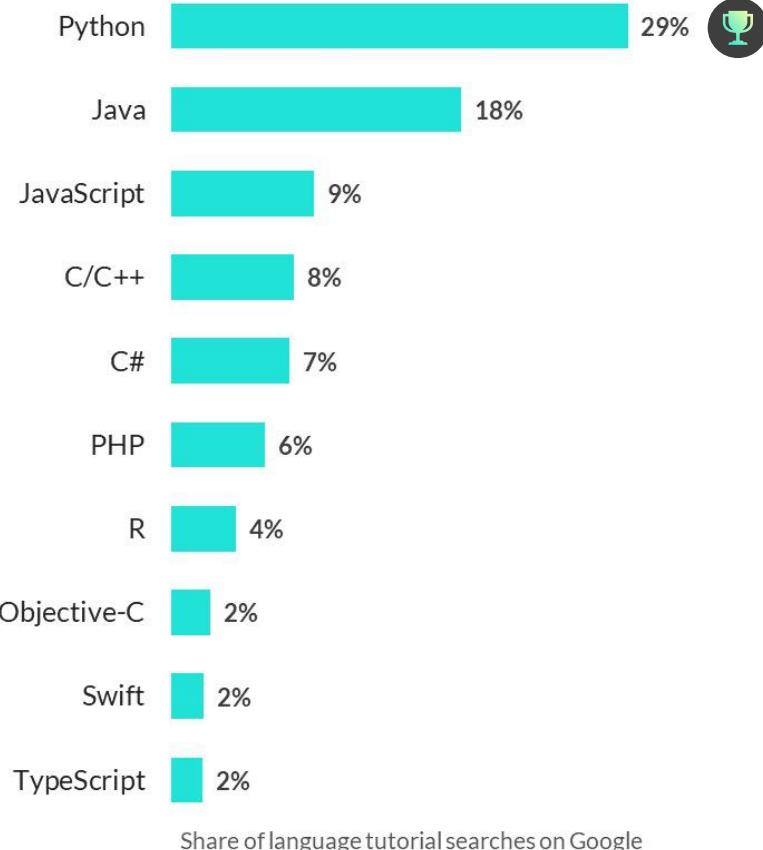


**Python** is a free, open-source programming language that is both powerful and easy to learn.

It has become one of the most popular languages in the world due to its accessibility, flexibility, ease of use, and wide range of applications, including:

- Data Analytics
- Software Development
- Machine Learning
- Web Scraping
- Game Development
- And more!

Popularity of Programming Language (PYPL):



# WHY PYTHON FOR ANALYTICS?



## Scalability

Unlike most analytics tools or self-service Business Intelligence platforms, Python is **open source, free to use, and built for scale**



## Versatility

With powerful libraries and frameworks, Python can add value at **every stage of the analytics workflow**, from data prep and analysis to machine learning and visualization



## Community

Become part of a **large and active Python user community**, where you can share resources, get help, offer support, and connect with other users



## Automation

Python can **automate complex tasks and workflows out of the box**, without complicated integrations or custom plug-ins



## Demand

Python skills are **valuable and highly sought after**, and are becoming increasingly popular among analytics and Business Intelligence professionals



When it comes to data analytics, each tool has unique strengths and weaknesses; while Python shouldn't be the *only* tool in your stack, it can add tremendous value when combined with other tools like **Excel, SQL, Power BI & Tableau**

# PYTHON ANALYTICS ECOSYSTEM

## General Purpose Programming



Mastering **base Python** will give you a solid foundational understanding of the language, which is essential for using packages and libraries effectively

## Data Manipulation & Analysis



**Pandas** helps us structure our data into dataset formats similar to that which you'd see in SQL or Excel. It also provides us with an arsenal of analytical functions that help us manipulate data and calculate the metrics we need to understand our data

## Data Visualization



**Matplotlib** and **Seaborn** can create a wide array of visually appealing, static visualizations

**Plotly** can be used to create interactive visualizations and dynamic dashboards

## Machine Learning



**Scikit learn** is among the most popular tools for building and testing machine learning models

**Statsmodels** provides a suite of tools for model building and statistical analysis

**TensorFlow** is the industry standard for developing deep learning models

# DATA ROLES USING PYTHON

## BI / DATA ANALYST

**Data Analysts** often use Python due to its cost effectiveness, or in collaboration with data science teams

Analysts are typically well-versed in base Python, Pandas, and at least one visualization library

 19% of data analyst jobs require Python skills (25% in California)

## DATA ENGINEER

**Data Engineers** commonly use Python to automate complex ETL processes or interact with APIs.

DBAs often use Pandas to manipulate data before storing it in a database or data warehouse.

 72% of data engineer jobs require Python skills

## DATA VIZ SPECIALIST

**Data Visualization Specialists** may use Python to design custom visuals that standard templates can't support.

They often utilize integrations with tools like Power BI or Tableau.

 33% of data viz roles mention Python as a required or desired skill

## DATA SCIENTIST

**Data Scientists** are most likely to use the 'full stack' of Python data tools.

They leverage packages like Pandas, Scikit learn, Statsmodels & TensorFlow to build and deploy ML models.

 71% of data science and machine learning jobs require Python skills



# JUPYTER NOTEBOOKS

# JUPYTER NOTEBOOKS



In this section we'll install Anaconda and introduce **Jupyter Notebooks**, a user-friendly coding environment where we'll write our first Python program

## TOPICS WE'LL COVER:

Installation & Setup

Notebook Interface

Comments & Markdown

The Print Function

Google Colab

Helpful Resources

## GOALS FOR THIS SECTION:

- Install Anaconda and launch Jupyter Notebooks
- Get comfortable with the Jupyter Notebook environment and interface
- Learn some very basic Python syntax and write our first simple program



# INSTALLING ANACONDA (MAC)

Installation & Setup

Notebook Interface

Comments & Markdown

The Print Function

Google Colab

Helpful Resources

1) Go to [anaconda.com/products/distribution](https://anaconda.com/products/distribution) and click

Download

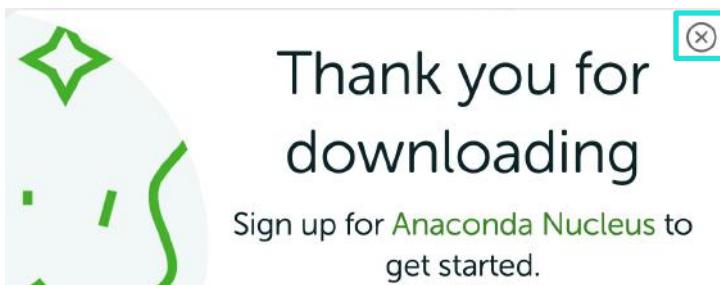
Individual Edition is now

## ANACONDA DISTRIBUTION

The world's most popular open-source Python distribution platform



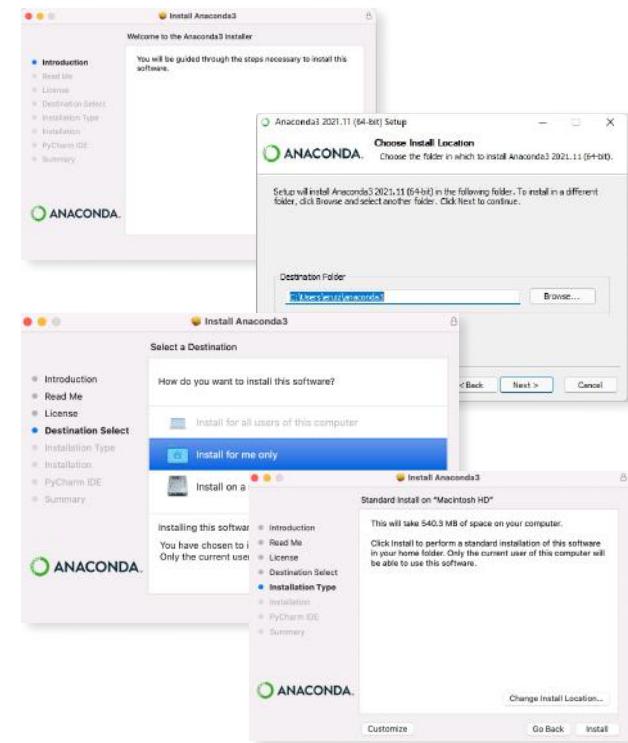
2) Click **X** on the Anaconda Nucleus pop-up  
(no need to launch)



3) Launch the downloaded Anaconda **pkg** file



4) Follow the **installation steps**  
(default settings are OK)





# INSTALLING ANACONDA (PC)

Installation & Setup

Notebook Interface

Comments & Markdown

The Print Function

Google Colab

Helpful Resources

1) Go to [anaconda.com/products/distribution](https://anaconda.com/products/distribution) and click 

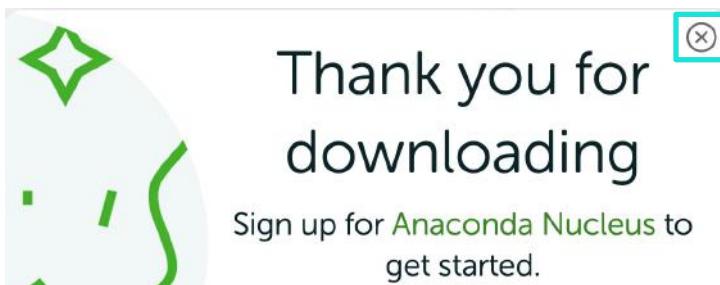
Individual Edition is now

## ANACONDA DISTRIBUTION

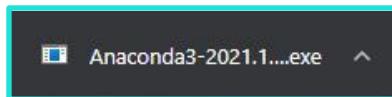
The world's most popular open-source Python distribution platform



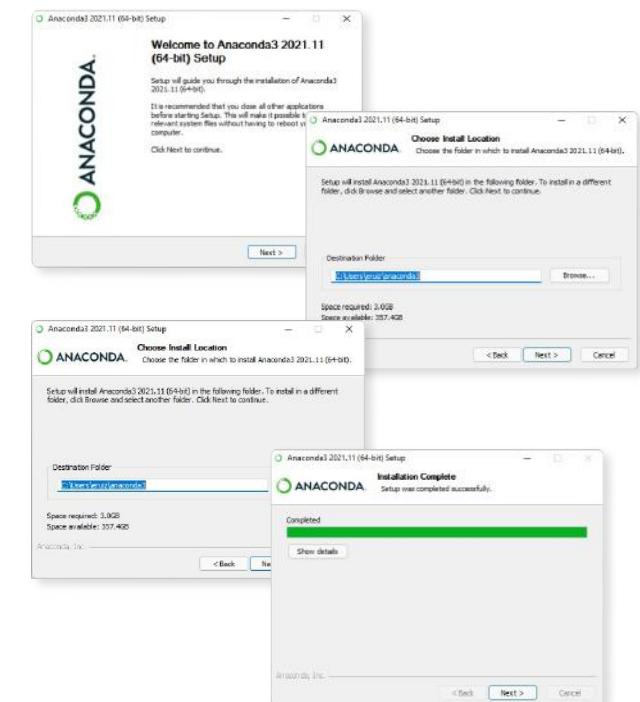
2) Click **X** on the Anaconda Nucleus pop-up  
(no need to launch)



3) Launch the downloaded Anaconda **exe** file



4) Follow the **installation steps**  
(default settings are OK)





# LAUNCHING JUPYTER

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

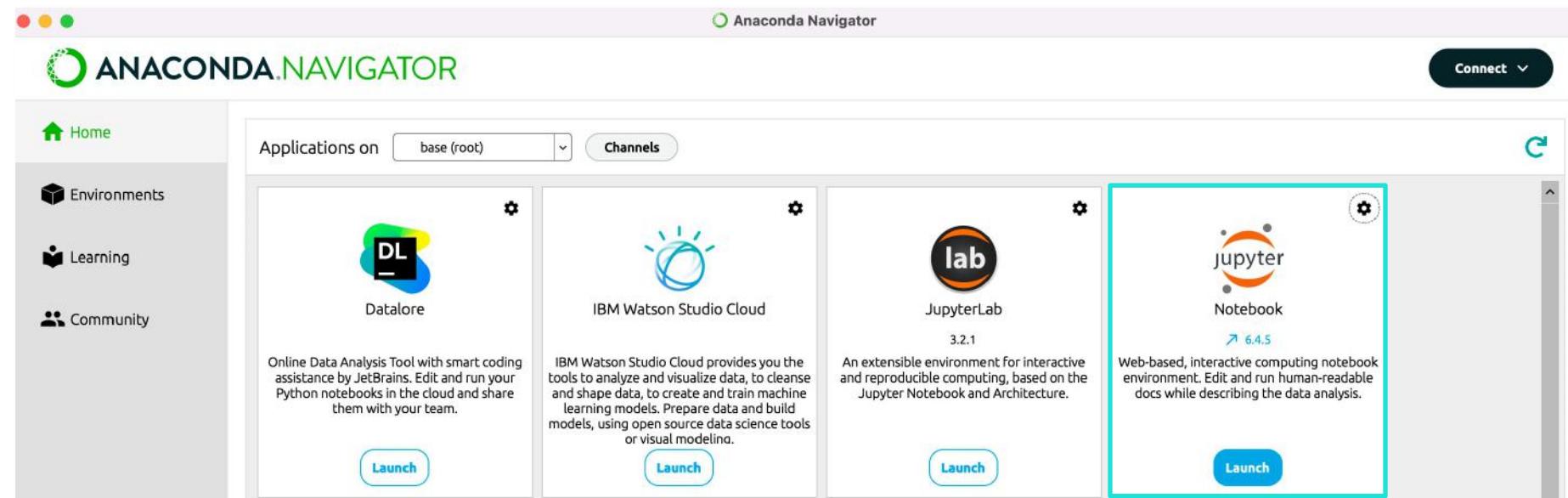
The Print  
Function

Google Colab

Helpful  
Resources

1) Launch **Anaconda Navigator**

2) Find **Jupyter Notebook** and click 





# YOUR FIRST JUPYTER NOTEBOOK

Installation & Setup

Notebook Interface

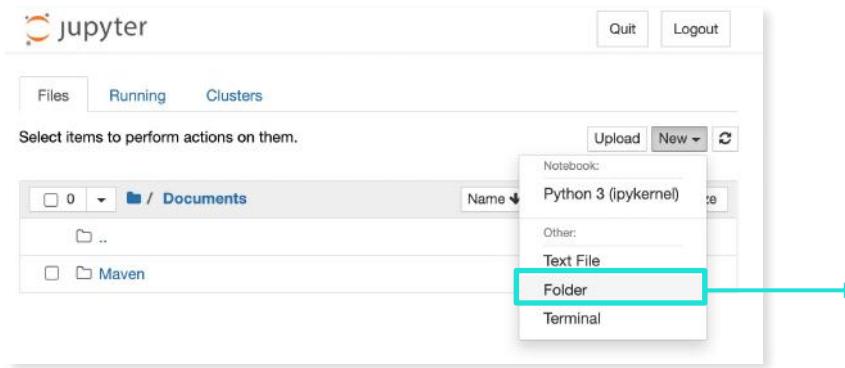
Comments & Markdown

The Print Function

Google Colab

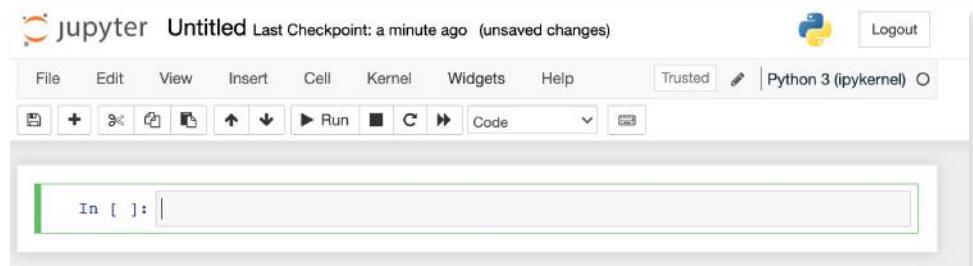
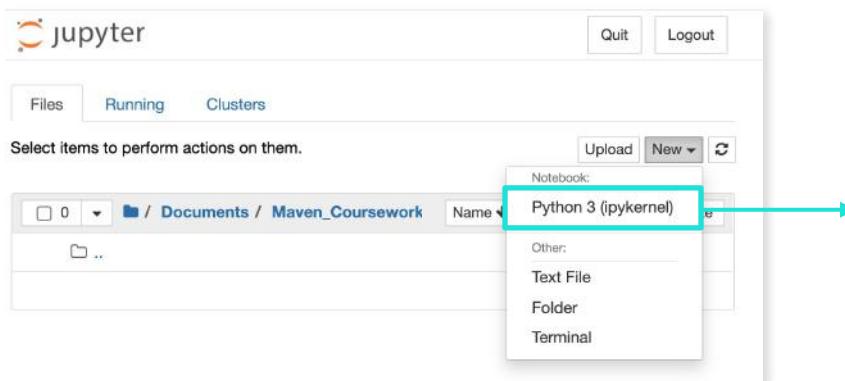
Helpful Resources

- Once inside the Jupyter interface, **create a folder** to store your notebooks for the course



**NOTE:** You can rename your folder by clicking "Rename" in the top left corner

- Open your new coursework folder and **launch your first Jupyter notebook!**



**NOTE:** You can rename your notebook by clicking on the title at the top of the screen



# THE NOTEBOOK SERVER

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

**NOTE:** When you launch a Jupyter notebook, a terminal window may pop up as well; this is called a **notebook server**, and it powers the notebook interface

```
Last login: Tue Jan 25 14:04:12 on ttys002
(base) chrisb@Chriss-MBP ~ % jupyter notebook
[I 2022-01-26 08:45:53.886 LabApp] JupyterLab extension loaded from /Users/chrisb/opt/anaconda3/lib/python3.9/site-packages/jupyterlab
[I 2022-01-26 08:45:53.886 LabApp] JupyterLab application directory is /Users/chrisb/opt/anaconda3/share/jupyter/lab
[I 08:45:53.890 NotebookApp] Serving notebooks from local directory: /Users/chrisb
[I 08:45:53.890 NotebookApp] Jupyter Notebook 6.4.5 is running at:
[I 08:45:53.890 NotebookApp] http://localhost:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[I 08:45:53.890 NotebookApp] or http://127.0.0.1:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[I 08:45:53.890 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 08:45:53.893 NotebookApp]

To access the notebook, open this file in a browser:
file:///Users/chrisb/Library/Jupyter/runtime/nbserver-27175-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
or http://127.0.0.1:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[W 08:46:05.829 NotebookApp] Notebook Documents/Maven_Coursework/Python_Intro.ipynb
```



If you close the server window,  
**your notebooks will not run!**

Depending on your OS, and method of launching Jupyter, one may not open. As long as you can run your notebooks, don't worry!



# THE NOTEBOOK INTERFACE

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

## Menu Bar

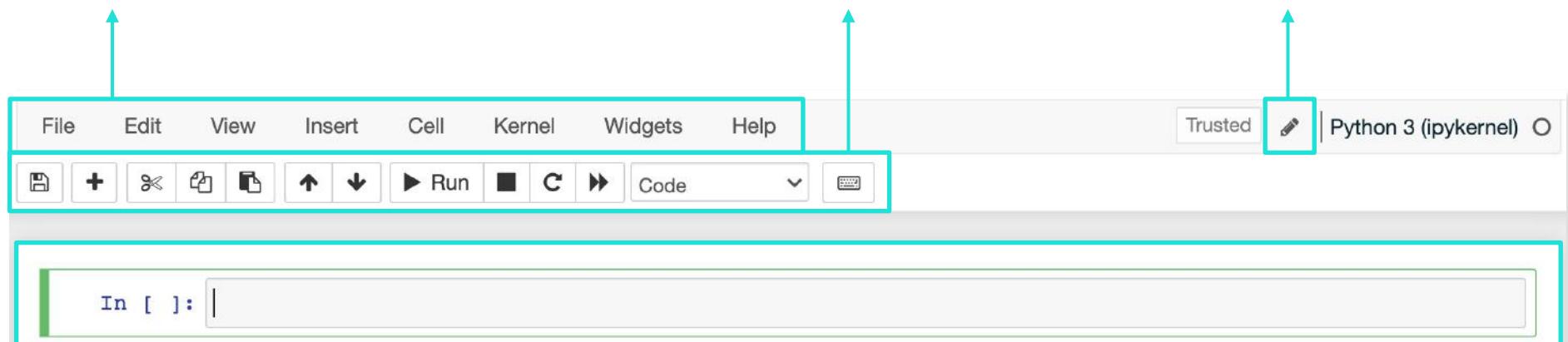
Options to manipulate the way  
the notebook functions

## Toolbar

Buttons for the most-used  
actions within the notebook

## Mode Indicator

Displays whether you are in **Edit**  
Mode or **Command** Mode



## Code Cell

Input field where you will write and  
edit new code to be executed



# MENU OPTIONS

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

## File

Save or revert, make a  
copy, open a notebook,  
download, etc.

- File Edit View Insert
- New Notebook ▾
- Open...
- Make a Copy...
- Save as...
- Rename...
- Save and Checkpoint ⌘S
- Revert to Checkpoint ▾
- Print Preview
- Download as ▾
- Trusted Notebook
- Close and Halt

## Edit

Edit cells within your  
notebook (while in  
command mode)

- Edit View Insert
- Cut Cells X
- Copy Cells C
- Paste Cells Above ↑V
- Paste Cells Below V
- Paste Cells & Replace
- Delete Cells D, D
- Undo Delete Cells Z
- Split Cell ⌘↑Minus
- Merge Cell Above
- Merge Cell Below
- Move Cell Up
- Move Cell Down
- Edit Notebook Metadata
- Find and Replace

## View

Edit cosmetic options for  
your notebook.

- View Insert Cell ▾
- Toggle Header
- Toggle Toolbar
- Toggle Line Numbers ↑L
- Cell Toolbar ▾

## Insert

Insert new cells into your  
notebook

- Insert Cell Kernel
- Insert Cell Above A
- Insert Cell Below B

# MENU OPTIONS



Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

## Cell

Access options for  
running the cells in your  
notebook

Cell   Kernel   Widgets   Help

- Run Cells
- Run Cells and Select Below
- Run Cells and Insert Below
- Run All
- Run All Above
- Run All Below
- Cell Type
- Current Outputs
- All Output

## Kernel

Interact with the  
instance of Python that  
runs your code

Kernel   Widgets   Help

- Interrupt
- Restart
- Restart & Clear Output
- Restart & Run All
- Reconnect
- Shutdown
- Change kernel

## Widgets

Manage interactive  
elements, or 'widgets' in  
your notebook

Widgets   Help

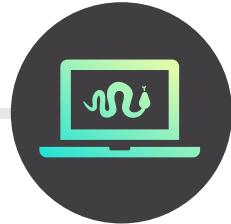
- Save Notebook Widget State
- Clear Notebook Widget State
- Download Widget State
- Embed Widgets

## Help

View or edit keyboard  
shortcuts and access  
Python reference pages

Help

- User Interface Tour
- Keyboard Shortcuts
- Edit Keyboard Shortcuts
- Notebook Help
- Markdown
- Python Reference
- IPython Reference
- NumPy Reference
- SciPy Reference
- Matplotlib Reference
- SymPy Reference
- pandas Reference
- About



# THE TOOLBAR

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

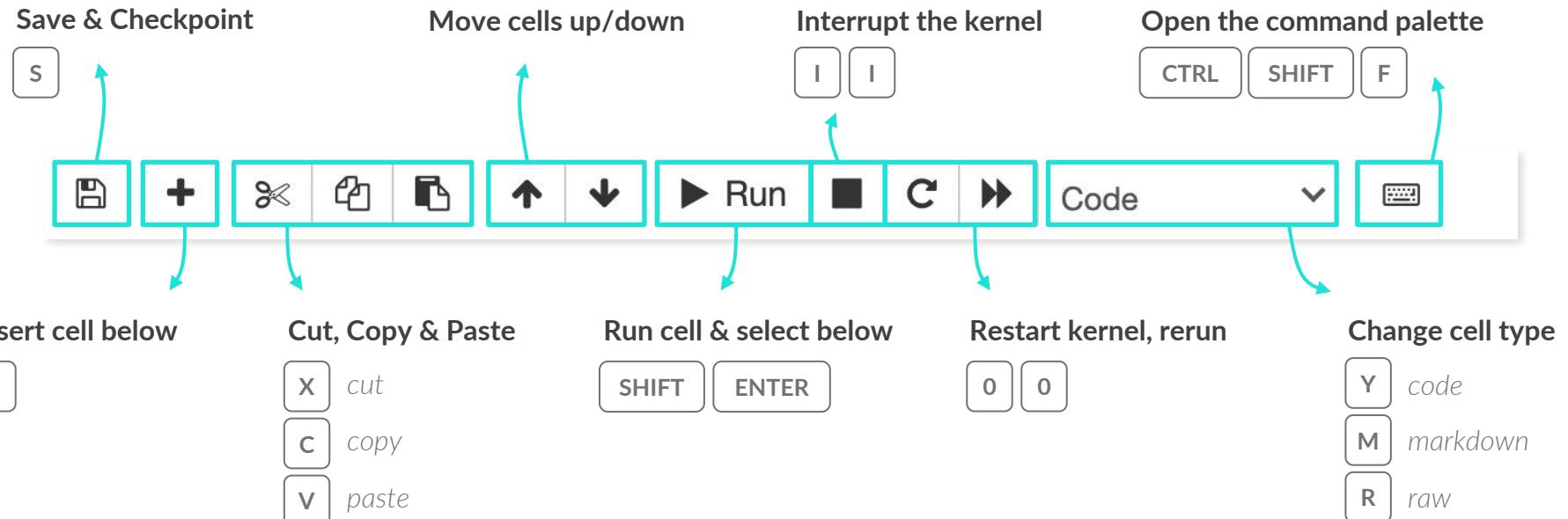
The Print  
Function

Google Colab

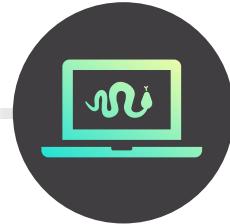
Helpful  
Resources

The **toolbar** provides easy access to the most-used notebook actions

- These actions can also be performed using hotkeys (keyboard shortcuts)



Shortcuts may differ depending on **which mode you are in**



# EDIT & COMMAND MODES

Installation &  
Setup

Notebook  
Interface

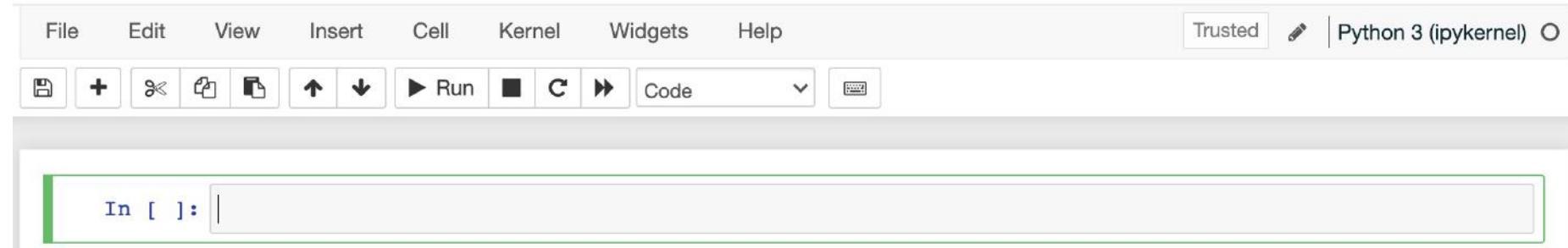
Comments &  
Markdown

The Print  
Function

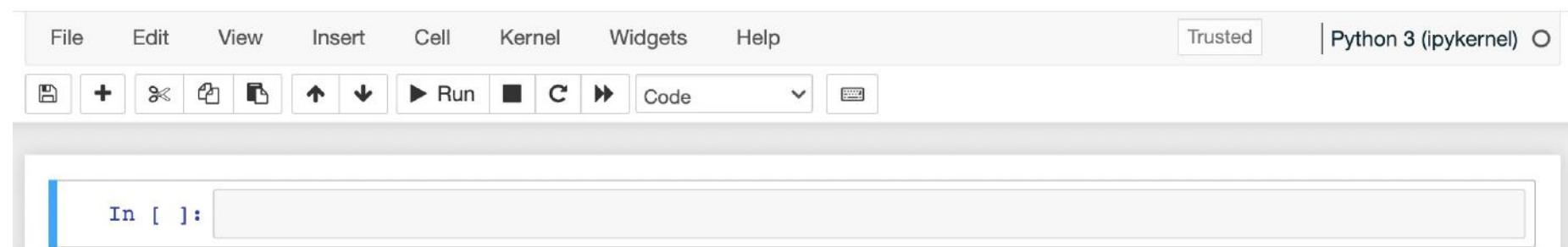
Google Colab

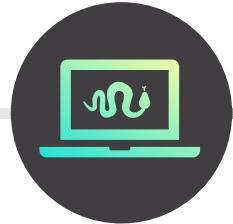
Helpful  
Resources

**EDIT MODE** is for editing **content within cells**, and is indicated by **green** highlights and a icon



**COMMAND MODE** is for editing **the notebook**, and is indicated by **blue** highlights and no icon





# THE CODE CELL

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

The **code cell** is where you'll write and execute Python code

In [ ]: |

In [1]: 5 + 5

Out[1]: 10

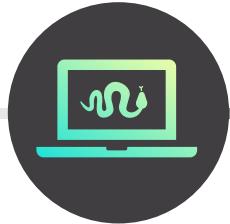
Type some code, and click **Run** to execute

- **In [ ]:** Our code (*input*)
- **Out [ ]:** What our code produced (*output*)\*

\*Note: not all code has an output!



**Congratulations**, you just became  
a Python programmer!



# THE CODE CELL

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

The **code cell** is where you'll write and execute Python code

In [1]: `5 + 5`

Out[1]: 10

In [2]: `5 + 5`

Out[2]: 10

Note that our output hasn't changed, but the number in the brackets increased from **1** to **2**.

This is a **cell execution counter**, which indicates how many cells you've run in the current session.

If the cell is still processing, you'll see **In[\*]**

Click back into the cell (or use the up arrow) and press **SHIFT + ENTER** to rerun the code

In [2]: `5 + 5`

Out[2]: 10

In [3]: `5 - 5`

Out[3]: 0

The cell counter will continue to increment as you run additional cells



# COMMENTING CODE

Installation & Setup

Notebook Interface

Comments & Markdown

The Print Function

Google Colab

Helpful Resources

**Comments** are lines of code that start with '# and do not run

- They are great for explaining portions of code for others who may use or review it
- They can also serve as reminders for yourself when you revisit your code in the future

```
In [4]: # I'm subtracting five from five. Add a space between your hash and comment.  
5 - 5
```

```
Out[4]: 0
```

```
In [5]: 5 - 5 # change the second 5 to a 6 tomorrow
```

```
Out[5]: 0
```

```
# This notebook is about teaching the basics of Jupyter notebook.  
# Should i define what a jupyter notebook is here?  
# I'm subtracting five from five. Add a space between your hash and comment.  
5 - 5 # 5 is the fifth integer greater than zero. It is also the number of fingers on our hand  
# 5 is a very interesting number  
# so is 0, which is the output
```

```
0
```

Think about your audience when commenting your code (you may not need to explain basic arithmetic to an experienced Python programmer)

Be conscious of over-commenting, which can actually make your code even more difficult to read



Comments should explain **individual cells or lines of code**, NOT your entire workflow – we have better tools for that!



# MARKDOWN CELLS

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

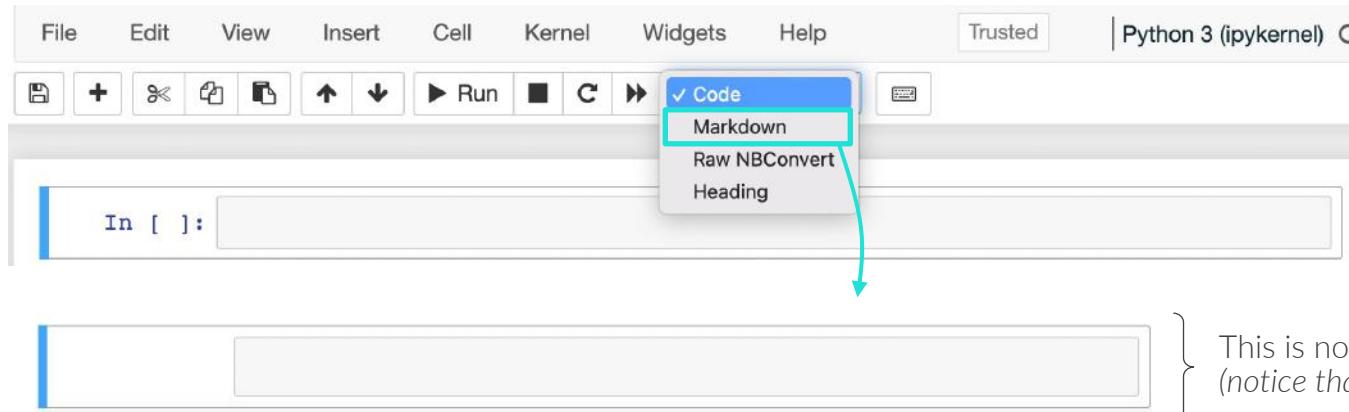
Google Colab

Helpful  
Resources

**Markdown cells** let you write structured text passages to explain your workflow, provide additional context, and help users navigate the notebook

## To create a markdown cell:

1. Create a new cell above the top cell (press **A** with the cell selected)
2. Select “**Markdown**” in the cell menu (or press **M**)



} This is now a markdown cell  
(notice that the `In [ ]:` disappeared)



# MARKDOWN SYNTAX

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

Markdown cells use a special **text formatting syntax**

## # Jupyter Notebook Intro

### ## Section 1: Markdown Basics

This is body text. I can use this area to provide more in depth explanations of my:

- \* Thought process
- \* Overall workflow
- \* etc

Anything that would be too much text for comments.

To create bulleted lists, begin a line with \*.

Numbered lists can be created by beginning a line with 1., 2., and so on.

Markdown has a **\*\*lot\*\*** of capabilities, and could be a course on its own. You will learn more as you build more notebooks and look at the work of others.

The Essentials to get started are:

1. Create headers with # (one is biggest, six is the smallest header)
2. **\*\*Bold\*\***, **\*italicize\***, **\*\*\*Bold AND Italic\*\*\***
3. Creating bulleted or numbered lists (begin a new line with \* for bullets 1. for numbers).
4. Code highlighting e.g. ``my_variable = 5``. Use the backtick, not apostrophe.

To explore further, I highly recommend [\[this guide.\]](https://www.markdownguide.org/basic-syntax/)(<https://www.markdownguide.org/basic-syntax/>)



# MARKDOWN SYNTAX

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

Markdown cells use a special **text formatting syntax**

## Jupyter Notebook Intro

### Section 1: Markdown Basics

This is body text. I can use this area to provide more in depth explanations of my:

- Thought process
- Overall workflow
- etc

Anything that would be too much text for comments.

To create bulleted lists, begin a line with \*.

Numbered lists can be created by beginning a line with 1., 2., and so on.

Markdown has a **lot** of capabilities, and could be a course on its own. You will learn more as you build more notebooks and look at the work of others.

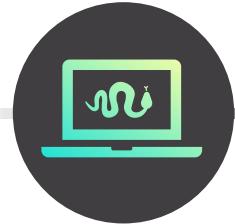
The Essentials to get started are:

1. Create headers with # (one is biggest, six is the smallest header)
2. **Bold**, *italicize*, **Bold AND Italic**
3. Creating bulleted or numbered lists (begin a new line with \* for bullets 1. for numbers).
4. Code highlighting e.g. `my_variable = 5`. Use the backtick, not apostrophe.

To explore further, I highly recommend [this guide](#).

In [4]: `5 + 5`

Out[4]: 10



# THE PRINT FUNCTION

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

The **print()** function will display a specified value

In [11]: `print('Hello World!')`

Hello World!

Simply specify the value  
you want to print inside  
the parenthesis

In [16]: `print(5, 5 + 5)`

5 10

You can print multiple  
values by separating  
them with **commas**

Note that this does not say **Out[16]**: as a printed output is  
different from the standard output returned by Python



Besides `print()`, Python has **many built-in functions** along with tools  
for **creating our own custom functions** (more on that later!)



**PRO TIP:** Add a "?"  
after a function name to  
access documentation

In [18]: `print?`

**Docstring:**  
`print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`  
**Prints the values to a stream, or**  
**Optional keyword arguments:**  
`file:` a file-like object (stream)  
`sep:` string inserted between values  
`end:` string appended after the last value  
`flush:` whether to forcibly flush the stream  
**Type:** `builtin_function_or_method`



# ALTERNATIVE: GOOGLE COLAB

Installation &  
Setup

Notebook  
Interface

Comments &  
Markdown

The Print  
Function

Google Colab

Helpful  
Resources

**Google Colab** is Google's cloud-based version of Jupyter Notebooks

To create a Colab notebook:

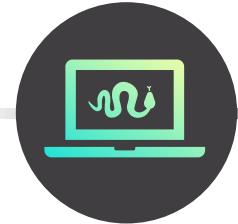
1. Log in to a Gmail account
2. Go to **colab.research.google.com**
3. Click “**new notebook**”



Colab is very similar to Jupyter Notebooks (*they even share the same file extension*); the main difference is that you are connecting to **Google Drive** rather than your machine, so files will be stored in Google's cloud

The screenshot shows the Google Colab interface. At the top, there are tabs for 'Examples', 'Recent', 'Google Drive', 'GitHub', and 'Upload'. The 'Recent' tab is selected. Below the tabs, there is a 'Filter notebooks' input field and a table with columns for 'Title', 'Last opened', and 'First opened'. Two notebooks are listed: 'Welcome To Colaboratory' (last opened 1:34 PM on January 4) and 'Scratch\_Work.ipynb' (last opened January 25, first opened January 4).

The screenshot shows the Google Drive interface. On the left, there is a sidebar with options like 'New', 'Priority', 'My Drive', 'Shared drives', 'Shared with me', 'Recent', 'Starred', 'Trash', and 'Storage'. In the center, there is a 'Suggested' section with a thumbnail for '1\_Python\_Intro.ipynb'. On the right, there is a modal window titled 'New notebook' with a 'Cancel' button.



# HELPFUL RESOURCES

Installation & Setup

Notebook Interface

Comments & Markdown

The Print Function

Google Colab

Helpful Resources

why is my output not printing python

In Python what is it called when you see the output of a variable without printing it?

Built-in Functions

Creating a better dashboard with Python, Dash, and Plotly

**Google** your questions – odds are someone else has asked the same thing and it has been answered (*include Python in the query!*)

**Stack Overflow** is a public coding forum that will most likely have the answers to most of the questions you'll search for on Google

<https://stackoverflow.com/>

The **Official Python Documentation** is a great “cheat sheet” for library and language references

<https://docs.python.org/3/>

There are many quality **Python & Analytics Blogs** on the web, and you can learn a lot by subscribing and reviewing the concepts and underlying code

<https://towardsdatascience.com/>

# KEY TAKEAWAYS

---



## Jupyter Notebooks

- Jupyter notebooks are popular among analysts and data scientists, since they allow you to create and document entire analytical workflows and render outputs and visualizations on screen



## Code cells

- Make sure that you know how to run, add, move, and remove cells, as well as how to restart your kernel or stop the code from executing



## Use comments and markdown cells for documentation

- Comments should be used to explain specific portions of your code, and markdown should be used to document your broader workflow and help users navigate the notebook



## Google Colab

- Colab and Jupyter notebooks are very similar and share the same file extension, but Colab files are stored in Google Drive instead of on your machine

# DATA TYPES

# DATA TYPES



In this section we'll introduce Python **data types**, review their properties, and cover how to identify and convert between them

## TOPICS WE'LL COVER:

Data Types

The Type Function

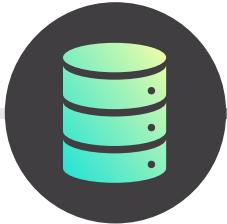
Type Conversion

Iterables

Mutability

## GOALS FOR THIS SECTION:

- Review the basics of Python data types
- Learn how to determine an object's data type
- Learn to convert between compatible data types
- Understand the concepts of iterability & mutability



# PYTHON DATA TYPES

Data Types

The Type Function

Type Conversion

Iterables

Mutability

Numeric



Text



Boolean



None



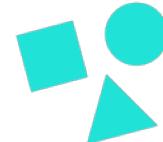
Sequence



Mapping



Set





# PYTHON DATA TYPES

Data Types

The Type Function

Type Conversion

Iterables

Mutability

## Single value (simple):



### Numeric

Represents numeric values

- Integer(`int`) - 5
- Float(`float`) - 5.24
- Complex(`complex`) - 5 + 2j



### Text

Represents sequences of characters, usually text

- String(`str`) - 'snowboard'



### Boolean

Represents True and False values

- Boolean(`bool`) - True, False



### None

Represents the absence of a value

- `NoneType` - None

## Multiple values:



### Sequence

Represents sequences of values, usually text or numeric

- List(`list`) - [1, 3, 5, 7, 9]
- Tuple(`tuple`) - ('snowboard', 'skis')
- Range(`range`) - range(1, 10, 2)



### Mapping

Maps keys to values for efficient information retrieval

- Dictionary(`dict`) - { 'snowboard': 24, 'skis': 17 }



### Set

Represents a collection of unique, non-duplicate values

- Set(`set`) - { 'snowboard', 'skis' }
- Frozen Set(`frozenset`) - { 'snowboard', 'skis' }



# THE TYPE FUNCTION

Data Types

The Type Function

Type Conversion

Iterables

Mutability

`type(1)`

`int`

`type('snowboard')`

`str`

`type(True)`

`bool`

`type(None)`

`NoneType`

`type([1, 3, 5, 7])`

`list`

`type({'a': 3, 'b': 5})`

`dict`



**PRO TIP:** Use `type()` if you are getting a `TypeError` or unexpected behavior when passing data through a function to make sure the data type is correct; it's not uncommon for values to be stored incorrectly



# TYPE CONVERSION

Data Types

The Type Function

Type Conversion

Iterables

Mutability

Convert data types by using “<data type>(object)”

## EXAMPLE

Converting data into an integer data type

```
int('123')
```

123

Using `int` converts the text string of '123' into an integer data type



Errors in Python will **cause the code to stop running**, so it's important to learn to diagnose and fix them

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

`TypeError`

Using `int` attempts to convert the list into an integer data type, but returns a `TypeError` instead

This operation isn't valid because the list has multiple values, while an integer can only be one value

`TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'`



# ITERABLES

Data Types

The Type Function

Type Conversion

Iterables

Mutability

**Iterables** are data types that can be iterated, or looped through, allowing you to move from one value to the next

These data types are considered iterable:

**Sequence**



**Mapping**



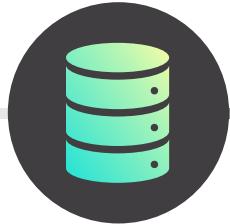
**Set**



**Text**



While text strings are treated as a single value, individual characters in a text string can be iterated through



# MUTABILITY

Data Types

The Type Function

Type Conversion

Iterables

Mutability

## Mutable Data Types

*Flexible – can add, remove, change values in the object*

- Lists
- Dictionaries
- Sets

## Immutable Data Types

*To modify a value must delete and recreate the entire object*

- Integers
- Floats
- Strings
- Booleans
- Tuples
- Frozenset

*Note that all simple data types are immutable*

# KEY TAKEAWAYS

---



Python has a wide variety of **data types** with different properties

- *It's important to understand them at a high level for now, as we'll dive deeper into each later in the course*



The **type()** function allows you to determine an object's data type

- *You can convert between data types if the underlying values are compatible*



**Iterables** are data types with values that you can loop through

- *Text strings are iterable despite containing a single value, as you can iterate through its characters*



Data types can be **mutable** or **immutable**

- *Mutable data types can be modified after their creation, while immutable data types cannot be changed without being overwritten*

# VARIABLES

# VARIABLES

---



In this section we'll introduce the concept of **variables**, including how to properly name, overwrite, delete, and keep track of them

## TOPICS WE'LL COVER:

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

## GOALS FOR THIS SECTION:

- Learn to assign variables in Python
- Understand the behavior for overwriting variables
- Learn the rules & best practices for naming variables



# VARIABLE ASSIGNMENT

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

**Variables** are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

variable\_name = value

*Intuitive label assigned to the variable*

*Examples:*

- price
- city
- heights

*Initial value assigned to the variable*

*Examples:*

- 10.99
- 'Los Angeles'
- [180, 173, 191]



Make sure you add the **space** on both sides of the equal sign!



# VARIABLE ASSIGNMENT

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

**Variables** are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

## EXAMPLE

*Creating a price variable*

```
price = 5  
print(price)
```

5

We're creating a variable named **price** and assigning it a value of 5, then we're printing the variable, returning its value



**PRO TIP:** Only assign variables for values that can change or will be used repeatedly; if you're not sure, it's likely a good idea to assign a variable

```
price = 5  
print(price + 1)
```

6

Any operation that can be performed on the value of a variable can be performed using the variable name  
Here we're printing the result of adding 1, a hard coded value, to the **price**



# VARIABLE ASSIGNMENT

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

**Variables** are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

## EXAMPLE

*Creating a price list variable*

```
price_list = [2.50, 4.99, 10, None, 'PROMO']  
  
print_price_list = print(price_list)  
  
print_price_list
```

```
[2.5, 4.99, 10, None, 'PROMO']
```

First, we're creating a variable named **price\_list** and assigning it to list of values

Then we're assigning a call to the **print** function with our **price\_list** variable as input



Note that assigning the **print()** function to a variable here doesn't necessarily improve our program over simply calling **print()** outside of it, but it's worth knowing **even a function call can be assigned to a variable**

# ASSIGNMENT: VARIABLE ASSIGNMENT

 **NEW MESSAGE**  
February 3, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Tax Calculation**

Hi there, welcome to the Maven Ski Company!

For your first task, I'd like you to change the code that adds a dollar in tax to our price before printing.

The tax will no longer be fixed at one dollar, so:

- Create a 'tax' variable and assign it a value of 2
- Create a 'total' variable equal to 'price' + 'tax'
- Print 'total'

Thanks in advance!

 [tax\\_code.ipynb](#)

 [Reply](#)    [Forward](#)

## *Results Preview*

```
print(total)
```

7

# ASSIGNMENT: VARIABLE ASSIGNMENT

 **NEW MESSAGE**  
February 3, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Tax Calculation

Hi there, welcome to the Maven Ski Company!

For your first task, I'd like you to change the code that adds a dollar in tax to our price before printing.

The tax will no longer be fixed at one dollar, so:

- Create a 'tax' variable and assign it a value of 2
- Create a 'total' variable equal to 'price' + 'tax'
- Print 'total'

Thanks in advance!

 [tax\\_code.ipynb](#)

 Reply    Forward

## Solution Code

```
price = 5
tax = 2
total = price + tax

print(total)
```

7



# OVERWRITING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

```
price = 5  
price = 6  
price = 7  
  
print(price)
```

7

```
price = 5  
new_price = 6  
  
print(price, new_price)
```

5 6

Python stores the value of 5 in memory when it is assigned to **price**

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory



Memory



Consider creating a new variable for a new value rather than overwriting

When you overwrite a variable, its previous value will be lost and cannot be retrieved



# OVERWRITING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

```
price = 5  
price = 6  
price = 7  
  
print(price)
```

7

```
old_price = 5  
price = 6  
  
print(old_price, price)
```

5 6

Python stores the value of 5 in memory when it is assigned to **price**

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory

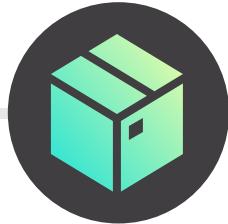


Memory



When you overwrite a variable, its previous value will be lost and cannot be retrieved

Or create a new variable for the old value



# OVERWRITING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

```
price = 5  
price = 6  
price = 7  
  
print(price)
```

7



**PRO TIP:** Use variables like 'new\_price' and 'old\_price' when testing programs with different values to make sure you don't lose important data – once the code works as needed, remove any extra variables!

Python stores the value of 5 in memory when it is assigned to **price**

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory



Memory



When you overwrite a variable, its previous value will be lost and cannot be retrieved



# OVERWRITING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

Variables can also be **assigned as values to other variables**

- The underlying value is assigned, but there is no remaining association between the variables

```
price = 5  
new_price = 6  
price = new_price  
new_price = 7  
  
print(price)
```

6

Python stores the value of 5 in memory when it is assigned to **price**  
Then stores the value of 6 when it is assigned to **new\_price**  
Then assigns the value of **new\_price**, which is 6, to **price**, 5 is no longer assigned to a variable, so gets removed  
Then assigns the value of 7 to **new\_price**  
Note that **price** is still equal to 6, it's not tied to the value of **new\_price**!



Memory

price	6	5
new_price	7	



# DELETING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

The **del** keyword will permanently remove variables and other objects

```
price = 5  
del price
```

```
print(price)
```

```
NameError
```

```
Traceback (most recent call last)  
/var/folders/f8/075hbnj13wb0f9yzh9k4ny  
z00000gn/T/ipykernel_36120/432900761.p  
y in <module>  
    2 del price  
    3  
----> 4 print(price)
```

```
NameError: name 'price' is not defined
```

Python stores the value of 5 in memory when it is assigned to **price**

Then **del** removes it from memory

When we try to print **price**, we get an **Error**, as the variable no longer exists

Memory

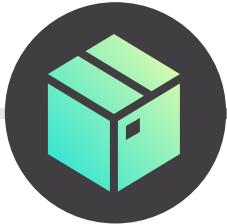
price

5

**NameError** occurs when we reference a variable or object name that isn't defined



**PRO TIP:** Deleting objects is generally unnecessary, and mostly used for large objects (like datasets with 10k+ rows); in most cases, reassign the variables instead and Python will get rid of the old value!



# NAMING RULES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

Variables have some basic **naming rules**



Variable names **can**:

- Contain letters (case sensitive!)
- Contain numbers
- Contain underscores
- Begin with a letter or underscore



Variable names **cannot**:

- Begin with a number
- Contain spaces or other special characters (\*, &, ^, -, etc.)
- Be reserved Python keywords like **del** or **list**



**PRO TIP:** “Snake case” is the recommended naming style for Python variables, which is all lowercase with words separated by underscores (first\_second\_third, new\_price, etc.)



# NAMING RULES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

Variables have some basic **naming rules**



**Valid** variable names:

- price\_list\_2019
- \_price\_list\_2019
- PRICE\_LIST\_2019
- pl2019



**Invalid** variable names

- 2019\_price\_list (*starts with a number*)
- price\_list-2019 (*has special characters*)
- 2019 price list (*has spaces*)
- list (*reserved Python keyword*)



**PRO TIP:** Give your variables intuitive names to make understanding your code easier – instead of PL19, consider something like price\_list\_2019 or prices\_2019



# TRACKING VARIABLES

Variable Assignment

Overwriting & Deleting

Naming Conventions

Tracking Variables

```
price = 10
product = 'Super Snowboard'
Date = '10-Jan-2021'
dimensions = [160, 25, 2]
```

```
%who
```

```
Date      dimensions      price      product
```

```
%whos
```

Variable	Type	Data/Info
----------	------	-----------

Date	str	10-Jan-2021
------	-----	-------------

dimensions	list	n=3
------------	------	-----

price	int	10
-------	-----	----

product	str	Super Snowboard
---------	-----	-----------------

*%who* returns variable names

*%whos* returns variable names, types, and information on the data contained



Magic commands that start with "%" only work in the iPython environments, which applies to Jupyter and Colab

# ASSIGNMENT: FIXING VARIABLES

 **NEW MESSAGE**  
February 3, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Price List Creation**

Hi there, thanks for you help on the tax code!

The file attached contains the attempts of our previous intern to create variables representing our 2018, 2019, and 2020 price lists, among others.

Please make sure these variables are given names in the format 'price\_list\_year' by filling in the correct year and cleaning up any unnecessary variable names.

Thanks in advance!

 [price\\_lists.ipynb](#)

 [Reply](#)    [Forward](#)

## Results Preview

Variable	Type	Data/Info
price_list_2018	list	n=5
price_list_2019	list	n=5
price_list_2020	list	n=5

# ASSIGNMENT: FIXING VARIABLES

 **NEW MESSAGE**  
February 3, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Price List Creation

Hi there, thanks for you help on the tax code!

The file attached contains the attempts of our previous intern to create variables representing our 2018, 2019, and 2020 price lists, among others.

Please make sure these variables are given names in the format 'price\_list\_year' by filling in the correct year and cleaning up any unnecessary variable names.

Thanks in advance!

 [price\\_lists.ipynb](#)

 Reply    Forward

## Solution Code

```
price_list_2018 = [4.99, 8.99, 17.99, 22.99, 94.99]
```

```
price_list_2019 = [5.99, 9.99, 19.99, 24.99, 99.99]
```

```
price_list_2020 = [6.49, 10.49, 19.99, 26.99, 104.99]
```

```
%whos
```

Variable	Type	Data/Info
<hr/>		
price_list_2018	list	n=5
price_list_2019	list	n=5
price_list_2020	list	n=5

# KEY TAKEAWAYS

---



Variables are containers used to **store values** from any data type

- *These values are stored in memory and can be referenced repeatedly in your code*



**Overwrite variables** by assigning new values to them

- *The old value will be lost unless it's assigned to another variable*



Variable names must follow Python's **naming rules**

- *"Snake case" is the recommended naming style (all lowercase with underscores separating each word)*

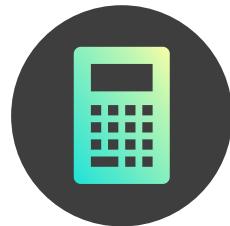


Give variables **intuitive** names

- *Even though you can use magic commands to track variables, good names save a lot of time & confusion*

# NUMERIC DATA

# NUMERIC DATA



In this section we'll cover **numeric data types**, including how to convert between them, perform arithmetic operations, and apply numeric functions

## TOPICS WE'LL COVER:

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

## GOALS FOR THIS SECTION:

- Review the different numeric data types
- Learn to convert between data types
- Perform moderately complex arithmetic operations



# NUMERIC DATA TYPES

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

There are 3 types of **numeric data types** in Python:

1. **Integers (int)** – whole numbers without decimal points
  - Examples: `3`, `42`, `-14`
2. **FLOATS (float)** – real numbers with decimal points
  - Examples: `3.14`, `-1.20134`, `0.088`
3. **Complex (complex)** – numbers with real and imaginary portions
  - Examples: `3 + 2j`, `2.4 - 1j`, `-2j`



Never worked with complex numbers before? Don't worry! They are rarely used in **data analytics**



# NUMERIC TYPE CONVERSION

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

Use “<data type>(object)” to **convert any number** into a numeric data type

You can convert floats into integers:

```
number = 1.9999  
type(number)
```

float

```
fixed_number = int(number)  
fixed_number, type(fixed_number)
```

(1, int)

Note that converting to int *rounds down*

Integers into floats:

```
number = 1  
fixed_number = float(number)  
fixed_number, type(fixed_number)
```

(1.0, float)

And even strings into floats (or integers):

```
text = '1.2'  
type(text)
```

str

```
fixed_number = float(text)  
fixed_number, type(fixed_number)
```

(1.2, float)

As long as they only contain numeric characters:

```
number = 'abc123'  
fixed_number = int(number)
```

Note that this string  
*has letters*

*ValueError: invalid literal for int() with base 10: 'abc123'*

*ValueError* occurs when a function receives an input with an inappropriate value



# ARITHMETIC OPERATORS

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

## **+** Addition

Adds two values

$$5 + 3 = 8$$

## **-** Subtraction

Subtracts one value from another

$$5 - 3 = 2$$

## **\*** Multiplication

Multiplies two values

$$5 * 3 = 15$$

## **/** Division

Divides one value by another

$$5 / 3 = 1.6666666667$$

## **//** Floor Division

Divides one value by another, then rounds down to the nearest integer

$$5 // 3 = 1$$

## **%** Modulo

Returns the remainder of a division

$$5 \% 3 = 2$$

## **\*\*** Exponentiation

Raises one value to the power of another

$$5 ** 3 = 125$$



# ORDER OF OPERATIONS

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

Python uses the standard PEMDAS **order of operations** to perform calculations

1. **P**arentheses
2. **E**xponentiation
3. **M**multiplication & **D**ivision (including Floor Division & Modulo), from left to right
4. **A**ddition & **S**ubtraction, from left to right

## Without Parentheses

$$3 * 6 + 5 - \underline{2^{**} 3} - 1$$

Exponentiation first

$$\underline{3 * 6} + 5 - 8 - 1$$

Then multiplication

$$18 + 5 - 8 - 1$$

Finally, addition & subtraction

14

## With Parentheses (to control execution)

$$3 * (\underline{(6 + 5)} - 2^{**}(\underline{3 - 1}))$$

Addition & subtraction in inner parentheses first

$$3 * (11 - \underline{2^{**} 2})$$

Then exponentiation in outer parenthesis

$$3 * (11 - 4)$$

Then subtraction in outer parenthesis

$$3 * 7$$

Finally, multiplication

21

# ASSIGNMENT: ARITHMETIC OPERATORS

 NEW MESSAGE  
February 4, 2022

From: Ricardo Nieva (Financial Planner)  
Subject: Financial Calculations

Hi there!  
Can you help calculate the following numbers?

- Gross profit from selling a snowboard
- Gross margin from selling a snowboard
- Price needed for a gross margin of 70%
- Sales tax on a snowboard sale
- Amount of money if the gross profit from selling 5 snowboards is invested for one year

There are more details in the file attached.

 financial\_calculations.ipynb     Reply     Forward

## Results Preview

```
print(gross_profit)
print(gross_margin)
print(price_needed_for70)
print(sales_tax)
print(amount_after_1yr)
```

```
300.0
0.6000120002400048
666.6333333333332
39
1575.0
```

# ASSIGNMENT: ARITHMETIC OPERATORS



**NEW MESSAGE**

February 4, 2022

From: **Ricardo Nieva** (Financial Planner)  
Subject: **Financial Calculations**

Hi there!

Can you help calculate the following numbers?

- Gross profit from selling a snowboard
- Gross margin from selling a snowboard
- Price needed for a gross margin of 70%
- Sales tax on a snowboard sale
- Amount of money if the gross profit from selling 5 snowboards is invested for one year

There are more details in the file attached.

financial\_calculations.ipynb

Reply

Forward

## Solution Code

```
snowboard_price = 499.99  
snowboard_cost = 199.99  
  
gross_profit = snowboard_price - snowboard_cost  
gross_profit
```

```
300.0  
gross_margin = gross_profit / snowboard_price  
gross_margin
```

```
0.6000120002400048  
  
desired_margin = .7  
  
price_needed_for70 = snowboard_cost / (1 - desired_margin)
```

```
price_needed_for70
```

```
666.633333333332
```

```
tax_rate = .08  
  
sales_tax = tax_rate * snowboard_price  
  
sales_tax
```

```
39.9992  
  
amount_invested = 5 * gross_profit  
interest_rate = .05  
  
amount_after_lyr = amount_invested + (amount_invested * interest_rate)  
  
amount_after_lyr
```

```
1575.0
```



# NUMERIC FUNCTIONS

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

## Single number:

**round**

*Rounds a number to a specified number of digits*

**round(number, # of digits)**

**abs**

*Returns the absolute value of a number*

**abs(number)**

## Multiple numbers:

**sum**

*Sums all numbers in an iterable*

**sum(iterable)**

**min**

*Returns the smallest value in an iterable*

**min(iterable)**

**max**

*Returns the largest value in an iterable*

**max(iterable)**



# ROUND & ABS

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

The **round()** function rounds a number to a specified number of digits

The **abs()** function returns the absolute value of a number

`round(3.1415926535, 2)`

3.14

`round(3.1415926535)`

3

`round(3.51)`

4

This rounds the number to 2 decimal places

`abs(-3)`

3

If number of digits isn't supplied, it will round to the nearest integer

It rounds down decimals < 0.5 and rounds up decimals >= 0.5

This will always return a positive number



# SUM, MIN & MAX

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

The **sum()**, **min()**, and **max()** functions perform sum, minimum, and maximum calculations on iterable data types that only contain numeric values

`sum( ( 3, 4, 5 ) )`

12

} This sums the numbers in the tuple

`min([ 3, 4, 5 ])`

3

} This returns the smallest value from the list

`max({ 3, 4, 5 } )`

5

} This returns the largest value from the set

# ASSIGNMENT: NUMERIC FUNCTIONS

 **NEW MESSAGE**  
February 4, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Customer Questions**

Hi there!  
Could you quickly run these numbers for me?

- What is the highest priced item we sell?
- What is the lowest priced item we sell?
- How much would it cost for a customer to purchase two of every item, rounded to the nearest dollar?

Thanks in advance! (the price list is attached)

 *customer\_questions.ipynb*

 **Reply**    **Forward**

## *Results Preview*

```
price_list = [129.99, 99.99, 119.19, 99.99, 89.99, 79.99, 49.99]
```

```
print(lowest_price, highest_price)
```

```
49.99 129.99
```

```
1338
```

# ASSIGNMENT: NUMERIC FUNCTIONS

 **NEW MESSAGE**  
February 4, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** Customer Questions

Hi there!  
Could you quickly run these numbers for me?

- What is the highest priced item we sell?
- What is the lowest priced item we sell?
- How much would it cost for a customer to purchase two of every item, rounded to the nearest dollar?

Thanks in advance! (the price list is attached)

 [customer\\_questions.ipynb](#)

[Reply](#) [Forward](#)

## *Solution Code*

```
price_list = [129.99, 99.99, 119.19, 99.99, 89.99, 79.99, 49.99]
```

```
lowest_price = min(price_list)  
highest_price = max(price_list)
```

```
print(lowest_price, highest_price)
```

```
49.99 129.99
```

```
round(2 * sum(price_list))
```

```
1338
```

# KEY TAKEAWAYS

---



- Data analysts typically work with **integer** & **float** numeric data types
  - Complex numbers are the third numeric data type, but it's unlikely you'll ever use them



- Arithmetic operations follow the **PEMDAS** order of operations
  - Use parentheses to control the order in which the operations are performed



- Python has **built-in functions** that work with specifically with numbers
  - Round, abs, sum, min, and max help perform simple operations that you'd otherwise have to code yourself



# STRINGS

# STRINGS



In this section we'll review the fundamentals of **strings**, a data type used to store text, and cover functions and methods to manipulate them

## TOPICS WE'LL COVER:

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## GOALS FOR THIS SECTION:

- Create strings from scratch or from other data types
- Learn to manipulate strings using arithmetic, indexing, slicing, and string methods
- Use f-strings to incorporate variables into strings and make them dynamic



# STRING BASICS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

**Strings** are sequences of character data commonly used to store text

You can create a string by using quotation marks, or converting from numbers:

*Single*  
quotes

```
new_string = 'This is a "string".'  
print(new_string)
```

This is a "string".

*Double*  
quotes

```
newer_string = "Double quotes allow us to use ' inside."  
print(newer_string)
```

Double quotes allow us to use ' inside.

*Triple*  
quotes

```
# triple quotes  
newerer_string = '''Triple quotes allow us to  
write multiline strings. We also don't  
have to worry about "errors" due to using  
single or double quotes. Pretty cool!  
''  
print(newerer_string)
```

Triple quotes allow us to  
write multiline strings. We also don't  
have to worry about "errors" due to using  
single or double quotes. Pretty cool!

*Convert*  
to string

```
number = 22.5  
string = str(number)  
  
string
```

'22.5'

The quotes indicate  
this is a string

print(string)

22.5

Using print() cleans  
up the quotes



# STRING ARITHMETIC

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Some **arithmetic operators** are compatible with strings

You can **concatenate** strings using **+**

```
'Snow' + 'board'
```

```
'Snowboard'
```

```
subject = 'My'  
noun = 'Maven snowboard'  
verb = 'cost'  
price = 22.55
```

```
subject + ' ' + noun + ' ' + verb + ' $' + str(price) + '!'
```

```
'My Maven snowboard cost $22.55!'
```

You can concatenate any number of strings  
The addition of '' to add spaces, as well as  
adding punctuation like '!' or '\$' is common

You can **repeat** strings using **\***

```
'Repeat 3 times: ' + ("I will not steal my coworker's pen! ") * 3
```

```
"Repeat 3 times: I will not steal my coworker's pen! I will not steal  
my coworker's pen! I will not steal my coworker's pen! "
```

Repeating strings is not  
very common in practice



# STRING INDEXING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Strings, along with sequence data types, can be navigated via their **index**

- Python is **0-indexed**, which means that the first item in a sequence has an index of 0, not 1

```
message = 'I hope it snows tonight!'
```

message[0]

'I'

message[1]

' '

message[2]

'h'

*String[index]* returns a specified character:

- String[0] returns the first character
- String[1] returns the second character
- String[2] returns the third character

message

0	I
1	h
2	o
3	p
...	
22	t
23	!



Indexing is a critical concept to master for data analysis, and while the 0-index can be confusing at first, with practice it will be second nature



# STRING INDEXING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Strings, along with sequence data types, can be navigated via their **index**

- Python is **0-indexed**, which means that the first item in a sequence has an index of 0, not 1

```
message = 'I hope it snows tonight!'
```

```
message[-1]
```

```
'!'
```



Indexing a value greater than the length of the string or object results in an **IndexError**

```
message[52]
```

**IndexError: string index out of range**



message

-24	I
-23	
...	
-5	i
-4	g
-3	h
-2	t
-1	!

You can also access individual characters in a text string by specifying their **negative** index

Negative indices begin at the end of the object and work backwards

Since there is no negative 0, this starts at -1

# ASSIGNMENT: STRING INDEXING

 **NEW MESSAGE**  
February 5, 2022

From: **Sierra Schnee** (Security Consultant)  
Subject: **Password Retrieval**

Hi there!

I need your help deciphering a hidden message. Our previous intern created a complex password storage procedure.

I've determined one of his passwords is 'Maven' , but what I'm interested in is the index of the letters used to spell 'Maven' in the attached text messages, as this will help unlock new passwords.

Thanks in advance!

 [suspicious\\_texts.ipynb](#)

## Results Preview

```
text1 = 'Your friend Mark'  
text2 = 'was'  
text3 = 'having'  
text4 = 'a great day'  
text5 = 'on the mountain'
```

```
maven = text1 + text2 + text3 + text4 + text5
```

```
print('The secret password is ' + maven)
```

The secret password is Maven

# ASSIGNMENT: STRING INDEXING

 **NEW MESSAGE**  
February 5, 2022

From: **Sierra Schnee** (Security Consultant)  
Subject: Password Retrieval

Hi there!

I need your help deciphering a hidden message. Our previous intern created a complex password storage procedure.

I've determined one of his passwords is 'Maven' , but what I'm interested in is the index of the letters used to spell 'Maven' in the attached text messages, as this will help unlock new passwords.

Thanks in advance!

 [suspicious\\_texts.ipynb](#)

## Solution Code

```
text1 = 'Your friend Mark'  
text2 = 'was'  
text3 = 'having'  
text4 = 'a great day'  
text5 = 'on the mountain'
```

```
maven = text1[-4] + text2[1] + text3[2] + text4[4] + text5[-1]
```

```
print('The secret password is ' + maven)
```

The secret password is Maven



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

**Slicing** returns multiple elements in a string, or sequence, via indexing

[start:stop:step size]

Index value  
to start with  
(default is 0)

Index value to stop at,  
not inclusive  
(default is all the values  
after the start)

Amount to increment each  
subsequent index  
(default is 1)



As with indexing, **slicing is a critical concept to master for data analysis**, as it allows you to manipulate a wide variety of data formats



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'
message[0:6:1]
```

' I hope '



How does this code work?

- **Start: 0** – start with the first element in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'
message[:6]
```

' I hope '



How does this code work?

- **Start: 0** – start with the first element in the string by default
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range by default



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'
message[2:6]
```

' hope '



How does this code work?

- **Start: 2** – start with the third element (index of 2) in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range by default



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'  
message[6:]
```

' it snows tonight! '



How does this code work?

- **Start: 6** – start with the seventh element (index of 6) in the string
- **Stop: 25** – stop at the end of the string
- **Step size: 1** – return every single element in the range by default



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'
message[0:6:2]
```

'Ihp'



How does this code work?

- **Start: 0** – start with the first element in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 2** – return every other element in the range



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'
message[:::-1]
```

' !thginot swons ti epoh I'



How does this code work?

- **Start:** `-1` – start with the last element in the string by default, due to the negative step size
- **Stop:** `-25` – stop at the beginning of the string
- **Step size:** `-1` – return every single element in the range, going backwards



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'  
message[-1:-9:-1]
```

' !thginot'



How does this code work?

- **Start: -1** – start with the last element in the string
- **Stop: -9** – stop at the ninth element from the end of the string without including it
- **Step size: -1** – return every single element in the range, going backwards



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'  
message[-9:-1:1]
```

' tonight'



How does this code work?

- **Start:** `-9` – start with the ninth element from the end of the string
- **Stop:** `-1` – stop at last element in the string without including it
- **Step size: 1** – return every single element in the range



# STRING SLICING

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

## EXAMPLE

Grabbing components of a text string

```
message = 'I hope it snows tonight!'  
message[-9::]
```

' tonight! '



How does this code work?

- **Start:** `-9` – start with the ninth element from the end of the string
- **Stop:** `25` – stop at the end of the string by default
- **Step size:** `1` – return every single element in the range by default

# ASSIGNMENT: STRING SLICING

 NEW MESSAGE  
February 5, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: Help Drafting Copy

Hi there!  
I need help drafting some copy.  
We have a long customer testimonial and I want to view several options for a good testimonial quote.  
The full details are in the file attached.  
Looking forward to working with you on this!

 marketing\_copy.ipynb     Reply     Forward

## Results Preview

```
print(short_testimonial)  
print(happy_customer)
```

I love skiing. It's my life.  
I love my life!

# ASSIGNMENT: STRING SLICING



NEW MESSAGE

February 5, 2022

From: **Alfie Alpine** (Marketing Manager)

Subject: Help Drafting Copy

Hi there!

I need help drafting some copy.

We have a long customer testimonial and I want to view several options for a good testimonial quote.

The full details are in the file attached.

Looking forward to working with you on this!

marketing\_copy.ipynb

Reply

Forward

## Solution Code

```
short_testimonial = testimonial[0:23] + testimonial[-7:-2]
```

```
print(short_testimonial)
```

I love skiing. It's my life.

```
happy_customer = short_testimonial[:7] + short_testimonial[-8:-1] + '!"
```

```
print(happy_customer)
```

I love my life!



# THE LENGTH FUNCTION

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The **len()** function returns the number of elements in an iterable

```
# len returns the length of strings  
len('snowboard')
```

9

} When used on a string, it returns the number of characters

```
# len will return the length of any iterable object  
len(['beginner', 'enthusiast', 'pro'])
```

3

} When used on a list, it returns the number of elements within it (more on lists later!)

# ASSIGNMENT: STRING LENGTH

 **NEW MESSAGE**  
February 5, 2022

**From:** Sierra Schnee (Security Consultant)  
**Subject:** Cryptography

Hi, great work last time!

I can't tell you much, but I have attached one more message for you to help me decode using Python.

I think in this batch the length of each message represents the letter in the alphabet for each character in the password.

Can you send me the password once you crack it?

Thanks in advance!

 suspicious2.ipynb     Reply     Forward

## Results Preview

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'  
print(len('Hello'))  
print(alphabet[5])  
print(password)
```

5  
f  
maven

# ASSIGNMENT: STRING LENGTH

 **NEW MESSAGE**  
February 5, 2022

From: **Sierra Schnee** (Security Consultant)  
Subject: **Cryptography**

Hi, great work last time!

I can't tell you much, but I have attached one more message for you to help me decode using Python.

I think in this batch the length of each message represents the letter in the alphabet for each character in the password.

Can you send me the password once you crack it?

Thanks in advance!

 suspicious2.ipynb     Reply     Forward

## Solution Code

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'  
message1 = 'Hello World!'  
message2 = ''  
message3 = 'It snowed a lot today'  
message4 = 'eeee'  
message5 = 'I love snow!!'
```

```
password = (alphabet[len(message1)]  
+ alphabet[len(message2)])  
+ alphabet[len(message3)]  
+ alphabet[len(message4)]  
+ alphabet[len(message5)])  
  
password  
  
'maven'
```



# METHODS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

**Methods** are functions that only apply to a specific data type or object

- We call them by following the object with a period then the method name

```
message = 'I hope it snows tonight!'
```

```
message.replace('snow', 'rain')
```

```
'I hope it rains tonight!'
```

} replace is a **string method** being applied to message, which is a string

```
number = 256  
number.replace(5, 9)
```

```
AttributeError: 'int' object has no attribute 'replace'
```

} If you try to apply a string method to an integer, you will receive an **AttributeError**



# STRING METHODS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

**find**

Returns the index of the first instance of a given string

`.find(string)`

**upper**

Converts all characters in a string to uppercase

`.upper()`

**lower**

Converts all characters in a string to lowercase

`.lower()`

**strip**

Removes leading and trailing spaces, or any other character specified, from a string

`.strip(optional string)`

**replace**

Substitutes a given string of characters with another

`.replace(string to replace, replacement)`

**split**

Splits a list based on a given separator (space by default)

`.split(optional string)`

**join**

Joins list elements into a single string separated by a delimiter

`delimiter.join(list)`



# FIND

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The `.find()` method searches for a specified value within a string and returns the index of its first instance

```
message = 'I hope it snows tonight!'
```

```
message.find('snow')
```

10

*'snow'* first appears in the 11<sup>th</sup> character (index value of 10)

- `.find(string)`

```
message.find('rain')
```

-1

*-1* is returned if the specified value isn't found



# UPPER & LOWER

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The `.upper()` and `.lower()` methods return a given string with all its characters in uppercase or lowercase respectively

```
message = 'I hope it snows tonight!'
```

```
print(message.upper())
```

I HOPE IT SNOWS TONIGHT!

```
print(message.lower())
```

i hope it snows tonight!



# STRIP

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The **.strip()** method removes leading and trailing spaces (by default), or any other specified character, from a string

- **.lstrip()** removes all leading spaces, and **.rstrip()** removes all trailing spaces

```
message = "      I love the space bar      "  
message.strip()
```

'I love the space bar'

This removes leading and trailing spaces by default

- **.strip(optional string)**

```
message = '.Remove the punctuation.'  
message.strip('.')
```

'Remove the punctuation'

This removes leading and trailing periods



# REPLACE

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The **.replace()** method substitutes a specified string of characters with another

- This is a common method to remove or change punctuation while cleaning text

```
message = 'I hope it snows tonight!'
```

```
message.replace('snow', 'rain')
```

'I hope it rains tonight!'

} This replaces all instances of 'snow' with 'rain'

- **.replace(string to replace, replacement)**

```
message.replace(' ', '')
```

'Ihopeitsnowstonight!'

} This replaces all **spaces** with an **empty string**, which creates one continuous word



# SPLIT & JOIN

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The **.split()** method separates a string into a list based on a delimiter  
(space by default)

```
message = 'I hope it snows tonight!'
word_list = message.split(' ')
word_list
```

```
['I', 'hope', 'it', 'snows', 'tonight!']
```

A new, separate element is created each time a space is encountered

- **.split(optional string)**

The **.join()** method combines individual list elements into a single string, separated by a given delimiter

```
' '.join(word_list)
```

```
'I hope it snows tonight!'
```

This takes the elements in `word_list` and combines them into a single string, separated by a single space

- **delimiter.join(list)**



# CHAINING METHODS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Methods can be **chained together** to create powerful expressions

```
message = 'I hope it snows TONIGHT!'
message[6::].lower().split()
['it', 'snows', 'tonight!']
```



message[6::] → 'it snows TONIGHT!'

'it snows TONIGHT!'.lower() → 'it snows tonight!'

'it snows tonight!'.split() → ['it', 'snows', 'tonight!']



How does this code work?

- The string is **sliced** to remove the first two words ('I hope')
- Then it's converted to all lowercase letters with **.lower()**
- And finally divided into a list of separate words with **.split()**

```
len(message[6::].lower().split())
```

3

You can even wrap this in a len function to return the number of words after 'I hope'



# CHAINING METHODS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Methods can be **chained together** to create powerful expressions

```
message = 'I hope it snows tonight!'
```



**PRO TIP:** Converting text to all upper or lowercase can make many operations, like finding a keyword or joining data much easier

```
message.find('SNOW')
```

-1

} Find is case-sensitive, so it won't find '**SNOW**' in 'I hope it **snows** TONIGHT'

```
message.upper().find('SNOW')
```

10

} By converting the message to uppercase, it finds the match and returns the index



# STRING METHOD CHEATSHEET



```
message = 'I hope it snows tonight!'
```

**.find(string)** Returns the index of the first instance of a given string

```
message.find('snow')
```

```
10
```

**.upper()** Converts all characters in a string to uppercase

```
message.upper()
```

```
'I HOPE IT SNOWS TONIGHT!'
```

**.lower()** Converts all characters in a string to lowercase

```
message.lower()
```

```
i hope it snows tonight!
```

**.strip(optional string)** Removes leading or trailing spaces or characters

```
message.strip('!')
```

```
'I hope it snows tonight'
```

**.replace(string to replace, replacement)** Substitutes a string with another

```
message.replace('snow', 'rain')
```

```
'I hope it rains tonight!'
```

**.split(optional string)** Splits a string into a list based on spaces or a delimiter

```
message.split()
```

```
['I', 'hope', 'it', 'snows', 'tonight!']
```

**delimiter.join(list)** Combines list elements into a string, separated by a delimiter

```
' '.join(word_list)
```

```
'I hope it snows tonight!'
```

**NOTE:** There are many more string methods than covered; to review those, or get additional detail on the methods covered, visit the official Python documentation:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

# ASSIGNMENT: STRING METHODS



## NEW MESSAGE

February 5, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: **Copy Edits**

We want to draft more copy from the file attached:

- Combine the text messages and call it 'full\_text'
- Remove leading spaces and convert to lowercase, call it 'fixed\_text'
- Change 'on the mountain' to 'at the ski shop' and call it 'new\_text'
- Look up a method to count the number of spaces in 'new\_text'; we pay by word so I want to know how much our ad will cost. Word count equals the number of spaces plus 1.

marketing\_copy2.ipynb

Reply

Forward

## Results Preview

```
text1 = 'Your friend Mark'  
text2 = 'was'  
text3 = 'having'  
text4 = 'a great day'  
text5 = 'on the mountain'
```

full\_text

```
'Your friend Mark was having a great day on the mountain'
```

fixed\_text

```
'your friend mark was having a great day on the mountain'
```

new\_text

```
'your friend mark was having a great day at the ski shop'
```

word\_count

12

# ASSIGNMENT: STRING METHODS



## NEW MESSAGE

February 5, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: **Copy Edits**

We want to draft more copy from the file attached:

- Combine the text messages and call it 'full\_text'
- Remove leading spaces and convert to lowercase, call it 'fixed\_text'
- Change 'on the mountain' to 'at the ski shop' and call it 'new\_text'
- Look up a method to count the number of spaces in 'new\_text'; we pay by word so I want to know how much our ad will cost. Word count equals the number of spaces plus 1.

marketing\_copy2.ipynb

Reply

Forward

## Solution Code

```
text1 = 'Your friend Mark'  
text2 = 'was'  
text3 = 'having'  
text4 = 'a great day'  
text5 = 'on the mountain'
```

```
full_text = text1 + ' ' + text2 + ' ' + text3 + ' ' + text4 + ' ' + text5
```

```
full_text
```

```
' Your friend Mark was having a great day on the mountain'
```

```
fixed_text = full_text.strip().lower()  
fixed_text
```

```
'your friend mark was having a great day on the mountain'
```

```
# let's remove this text and replace it with 'at the ski shop'  
# Slice up to the portion to remove and add the new text  
  
new_text = fixed_text[:fixed_text.find('on the mountain')] + 'at the ski shop'  
  
# OR, use replace  
  
new_text = fixed_text.replace('on the mountain', 'at the ski shop')  
new_text
```

```
'your friend mark was having a great day at the ski shop'
```

```
word_count = new_text.count(' ') + 1  
word_count
```



# F-STRINGS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

You can include variables in strings by using **f-strings**

- This allows you to change the output text based on changing variable values

```
name = 'Chris'  
role = 'employee'  
  
print(f"{name} is our favorite {role}, of course!")
```

Chris is our favorite employee, of course!

Add an **f** before the quotation marks to indicate an f-string, and place the variable names into the string by using curly braces {}

```
name = 'Anand'  
role = 'customer'  
  
print(f"{name} is our favorite {role}, of course!")
```

Anand is our favorite customer, of course!

Note that the f-string code doesn't change, but the output does change if the variables get assigned new values

# ASSIGNMENT: F-STRINGS

 NEW MESSAGE  
February 5, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Price Display Message

Hi again,  
Our website is currently displaying static prices, so we need to manually change them every time a price changes, and write a new one every time we get a new product  
Can you write a piece of code for a string that takes product and price as variables? Thanks.  
P.S. Don't work with Sierra anymore, she's not an employee

 price\_display.ipynb     Reply     Forward

## Results Preview

```
price = 499.99  
product = 'snowboard'
```

# your print statement here

The snowboard costs \$499.99.

```
price = 19.99  
product = 'scarf'
```

# exact copy of print statement in first cell

The scarf costs \$19.99.

These two lines of code should be identical

# ASSIGNMENT: F-STRINGS

 NEW MESSAGE

February 5, 2022

From: **Sally Snow** (Ski Shop Manager)

Subject: Price Display Message

Hi again,

Our website is currently displaying static prices, so we need to manually change them every time a price changes, and write a new one every time we get a new product

Can you write a piece of code for a string that takes product and price as variables? Thanks.

P.S. Don't work with Sierra anymore, she's not an employee

 price\_display.ipynb

## *Solution Code*

```
price = 499.99
product = 'snowboard'

print(f'The {product} costs ${price}.')
```

The snowboard costs \$499.99.

```
price = 19.99
product = 'scarf'

print(f'The {product} costs ${price}.')
```

The scarf costs \$19.99.

# KEY TAKEAWAYS

---



Strings are used to store **text** and other sequences of characters

- *Analysts interact with strings in categorical data, or in large bodies of text that need to be mined for insight*



Access single characters with **indexing**, and multiple characters with **slicing**

- *Indexing & slicing are used with many other data types beyond strings, so they are critical to master*



Learn and leverage **string methods** to facilitate working with strings

- *You don't need to memorize every method and their syntax on day 1, but it's important to be aware of them*

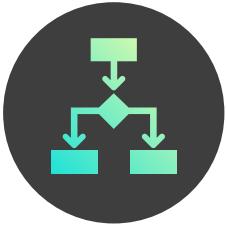


Use **f-strings** to incorporate variables into your strings

- *This will make your text strings dynamic and allow you to avoid writing repetitive code*

# CONDITIONAL LOGIC

# CONDITIONAL LOGIC



In this section we'll cover **conditional logic**, and write programs that make decisions based on given criteria using true or false expressions

## TOPICS WE'LL COVER:

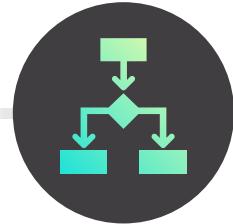
The Boolean Data Type

Boolean Operators

Conditional Control Flow

## GOALS FOR THIS SECTION:

- Understand the properties of the Boolean data type
- Learn to write true or false expressions using Boolean operators
- Learn to control the flow of the program using conditional logic statements



# THE BOOLEAN DATA TYPE

The Boolean  
Data Type

Boolean  
Operators

Conditional  
Control Flow

The **Boolean data type** has two possible values: **True** & **False**

- **True** is equivalent to **1** in arithmetic operations
- **False** is equivalent to **0** in arithmetic operations

**True \* 1**

1

**False \* 1**

0

The **not** keyword inverts Boolean values:

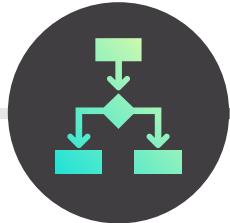
- **not True** is equivalent to **False**
- **not False** is equivalent to **True**

**(not True) \* 1**

0

**(not False) \* 1**

1



# COMPARISON OPERATORS

The Boolean  
Data Type

Boolean  
Operators

Conditional  
Control Flow

True

False

`==` Equal

`5 == 5`

`5 == 3`

`!=` Not Equal

`'Hello' != 'world!'`

`10 != 10`

`<` Less Than

`5 < 6 < 7`

`21 < 12`

`>` Greater Than

`10 > 5`

`10 > 5 > 7`

`<=` Less Than or Equal

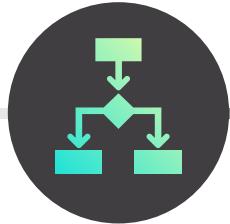
`5 <= 5`

`(5 + 5) <= 8`

`>=` Greater Than or Equal

`11 >= 9 >= 9`

`len('I') >= len('am')`



# MEMBERSHIP TESTS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

**Membership tests** check if a value exists inside an iterable data type

- The keywords **in** and **not in** are used to conduct membership tests

```
message = 'I hope it snows tonight!'
'snow' in message
```

**True**

```
'rain' in message
```

**False**

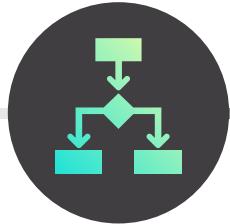
```
message = 'I hope it snows tonight!'
'snow' not in message
```

**False**

'snow' is in the string assigned to **message**, so the membership test returns **True**

'rain' is NOT in the string assigned to **message**, so the membership test returns **False**

'snow' is in the string assigned to **message**, so the membership test checking if it is NOT in message returns **False**



# BOOLEAN OPERATORS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

**Boolean operators** allow you to combine multiple comparison operators

- The **and** operator requires *all* statements to be true
- The **or** operator requires *one* statement to be true

One is True and the other False

```
5 > 4 and 7 > 8
```

False

```
5 > 4 or 7 > 8
```

True

You can chain together any number of Boolean expressions

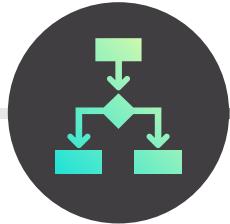
```
15 > 10 and 11 > 10 and 12 > 10
```

True



**PRO TIP:** If the first expression chained by **and** is **False**, or by **or** is **True**, the rest of the clauses will not be evaluated

Place simple clauses first to eliminate the need to run complex clauses, making your program more efficient



# BOOLEAN OPERATORS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

**Boolean operators** allow you to combine multiple comparison operators

- The **and** operator requires *all* statements to be true
- The **or** operator requires *one* statement to be true

Use parentheses to control the order of evaluation:

```
price = 105  
item = 'skis'  
  
(price <= 100 and item == 'ski poles') or (item == 'skis')
```

True

Because **item == 'skis'**  
the **or** operator makes  
the expression True

```
price <= 100 and (item == 'ski poles' or item == 'skis')
```

False

Because **price <= 100**  
returns false the **and**  
operator makes the  
expression False

# ASSIGNMENT: BOOLEAN OPERATORS

 **NEW MESSAGE**  
February 6, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Inventory Logic

We need develop logic that displays Boolean values based on our inventory levels (current inventory = 5):

1. Check if inventory is equal to 0
2. Check if inventory is greater than 5
3. If #2 is True, return a price of \$99, 0.0 if not
4. Check if inventory is positive and price is less than \$100
5. Check if the customer's name is 'Chris' and the product is 'super snowboard', or if the inventory is greater than 0

 [inventory\\_logic.ipynb](#) Reply Forward

## Results Preview

```
inventory_count = 5
```

```
False
```

```
False
```

```
0.0
```

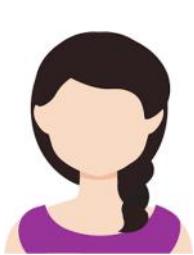
```
price = 99.99
```

```
True
```

```
customer_name = 'Barry'  
inventory_count = 0  
product = 'super_snowboard'
```

```
False
```

# ASSIGNMENT: BOOLEAN OPERATORS



## NEW MESSAGE

February 6, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Inventory Logic**

We need develop logic that displays Boolean values based on our inventory levels (current inventory = 5):

1. Check if inventory is equal to 0
2. Check if inventory is greater than 5
3. If #2 is True, return a price of \$99, 0.0 if not.
4. Check if inventory is positive and price is less than \$100
5. Check if the customer's name is 'Chris' and the product is 'super snowboard', or if the inventory is greater than 0

inventory\_logic.ipynb

Reply

Forward

## Solution Code

```
inventory_count = 5
```

```
inventory_count == 0
```

```
False
```

```
inventory_count > 5
```

```
False
```

```
(inventory_count > 5) * 99.99
```

```
0.0
```

```
price = 99.99
```

```
inventory_count > 0 and price < 100
```

```
True
```

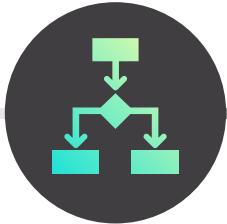
```
customer_name = 'Barry'
```

```
inventory_count = 0
```

```
product = 'super_snowboard'
```

```
inventory_count > 0 or (customer_name == 'Chris' and product == 'super_snowboard')
```

```
False
```



# THE IF STATEMENT

The Boolean Data Type

Boolean Operators

Conditional Control Flow

The **if statement** runs lines of indented code when a given logical condition is met

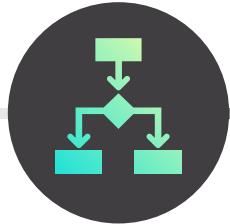
**if** condition:

Indicates an if statement

A logical expression that evaluates to True or False

*Examples:*

- *price > 100*
- *product == 'skis'*
- *inventory > 0 and inventory <= 10*



# THE IF STATEMENT

The Boolean  
Data Type

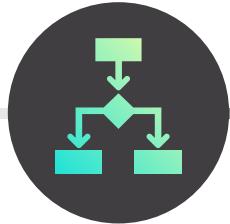
Boolean  
Operators

Conditional  
Control Flow

The **if statement** runs lines of indented code when a given logical condition is met

```
if condition:  
    do this
```

*Code to run if the logical  
expression returns True  
(must be indented!)*



# THE IF STATEMENT

The Boolean Data Type

Boolean Operators

Conditional Control Flow

## EXAMPLE

Determining experience level for a snowboard

```
price = 999.99

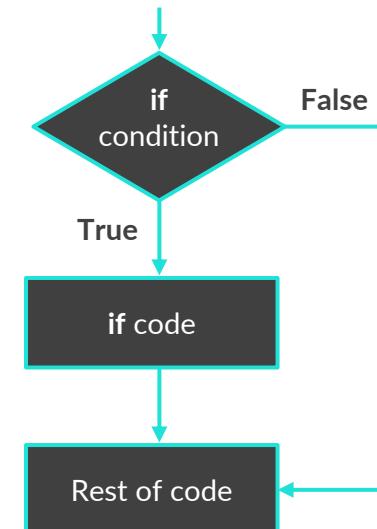
if price > 500:
    print('This snowboard is designed for experienced users.')

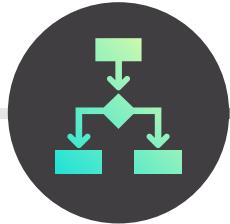
print('This code will run whether or not the if statement is True.)
```

This product is for experienced users.  
This code will run whether or not the if statement is True.



How does this code work?





# CONTROL FLOW

The Boolean Data Type

Boolean Operators

Conditional Control Flow

**Control flow** is a programming concept that allows you to choose which lines to execute, rather than simply running all the lines from top to bottom

There are three conditional statements that help achieve this: **if**, **else**, and **elif**

## EXAMPLE

Determining experience level for a snowboard

```
price = 499.99  
  
if price > 500:  
    print('This product is for experienced users.')  
  
print('This code will run whether or not the if statement is True.')
```

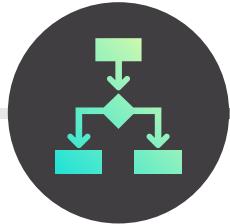
This code will run whether or not the if statement is True.

Python uses **indentation** to depart from a linear flow

In this case, the first print statement only runs if price is greater than 500



Press tab, or use four spaces, to indent your code when writing if statements or you will receive an **IndentationError**



# THE ELSE STATEMENT

The Boolean Data Type

Boolean Operators

Conditional Control Flow

The **else statement** runs lines of indented code when none of the logical conditions in an if or elif statements are met

## EXAMPLE

Determining experience level for a snowboard

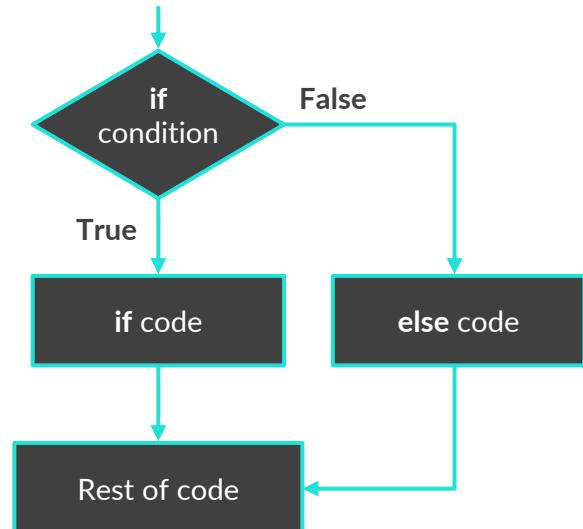
```
price = 499.99
price_threshold = 500

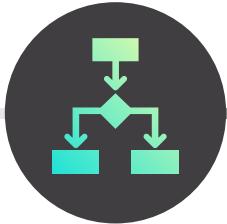
if price > price_threshold:
    print('This snowboard is for experienced users.')
else:
    print('This board is suitable for a beginner.)
```

This board is suitable for a beginner.



How does this code work?





# THE ELIF STATEMENT

The Boolean Data Type

Boolean Operators

Conditional Control Flow

The **elif statement** lets you specify additional criteria to evaluate when the logical condition in an if statement is not met

## EXAMPLE

Determining experience level for a snowboard

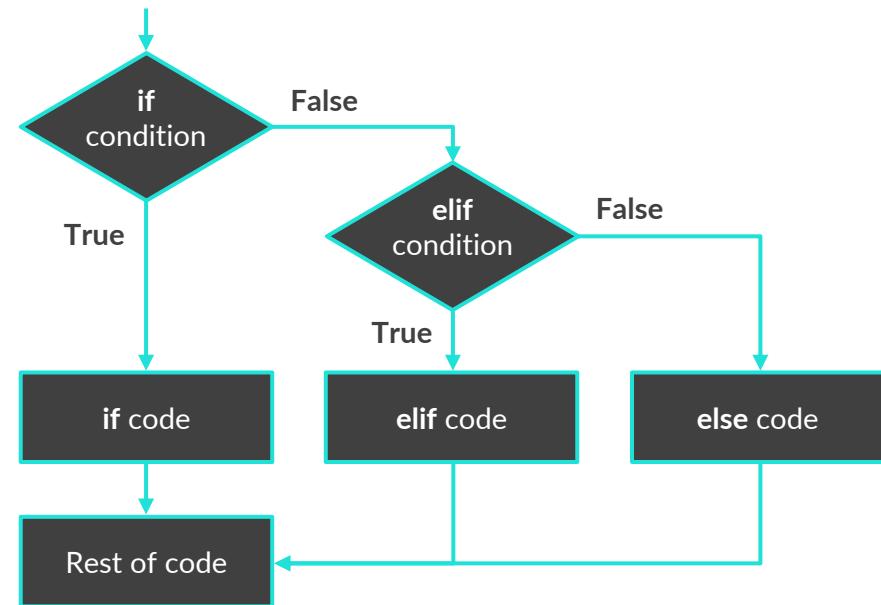
```
price = 499.99
expert_threshold = 500
intermediate_threshold = 300

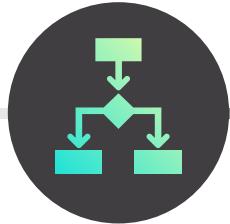
if price > expert_threshold:
    print('This board is for experienced users.')
elif price > intermediate_threshold:
    print('This board is good for intermediate_users.')
else:
    print('This should be suitable for a beginner.)
```

This board is good for intermediate\_users.



How does this code work?





# THE ELIF STATEMENT

The Boolean  
Data Type

Boolean  
Operators

Conditional  
Control Flow

Any number of **elif statements** can be used between the if and else statements  
As more logical statements are added, it's important to be careful with their order

*Will the elifs below ever run?*

```
price = 1499.99
luxury_threshold = 1000
expert_threshold = 500
intermediate_threshold = 300

if price > intermediate_threshold:
    print('This board is good for intermediate users')
elif price > expert_threshold:
    print('This board is for experienced users')
elif price > luxury_threshold:
    print("The gold doesn't improve the ride but it looks great!")
else:
    print('This should be suitable for a beginner.')
```

This board is good for intermediate users

**No**, because any value that is true for them will also be true for the if statement above them

Either put the most restrictive conditions first, or be more explicit with your logic

# ASSIGNMENT: CONTROL FLOW

 NEW MESSAGE  
February 6, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Inventory Query**

Hi there, great work on the last assignment!  
Time to take it a step further:

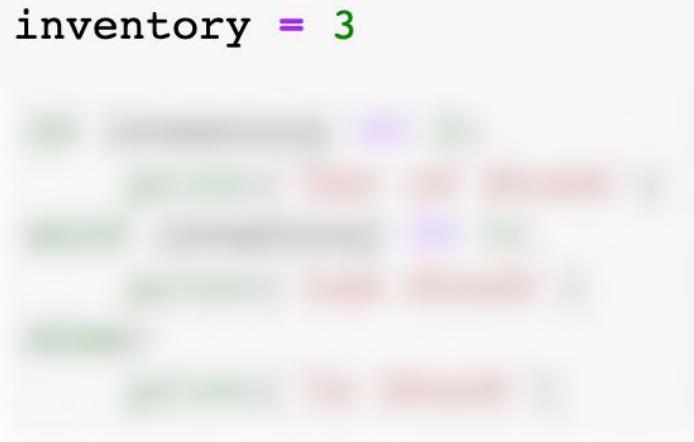
- If inventory is 0 or less, print 'OUT OF STOCK'
- If inventory is between 1-5, print 'Low Stock'
- If inventory is over 5, print 'In Stock'

Make sure to test your logic with several values.  
Can't wait to see your work!

 [inventory\\_query.ipynb](#)     Reply     Forward

## Results Preview

```
inventory = 3
```



**Low Stock**

# ASSIGNMENT: CONTROL FLOW

 NEW MESSAGE  
February 6, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Inventory Query**

Hi there, great work on the last assignment!  
Time to take it a step further:

- If inventory is 0 or less, print 'OUT OF STOCK'
- If inventory is between 1-5, print 'Low Stock'
- If inventory is over 5, print 'In Stock'

Make sure to test your logic with several values.  
Can't wait to see your work!

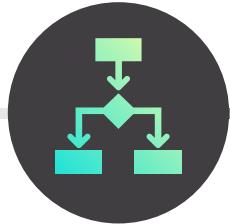
inventory\_query.ipynb     Reply     Forward

## *Solution Code*

```
inventory = 3

if inventory <= 0:
    print('Out of Stock')
elif inventory <= 5:
    print('Low Stock')
else:
    print('In Stock')
```

Low Stock



# NESTED IF STATEMENTS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

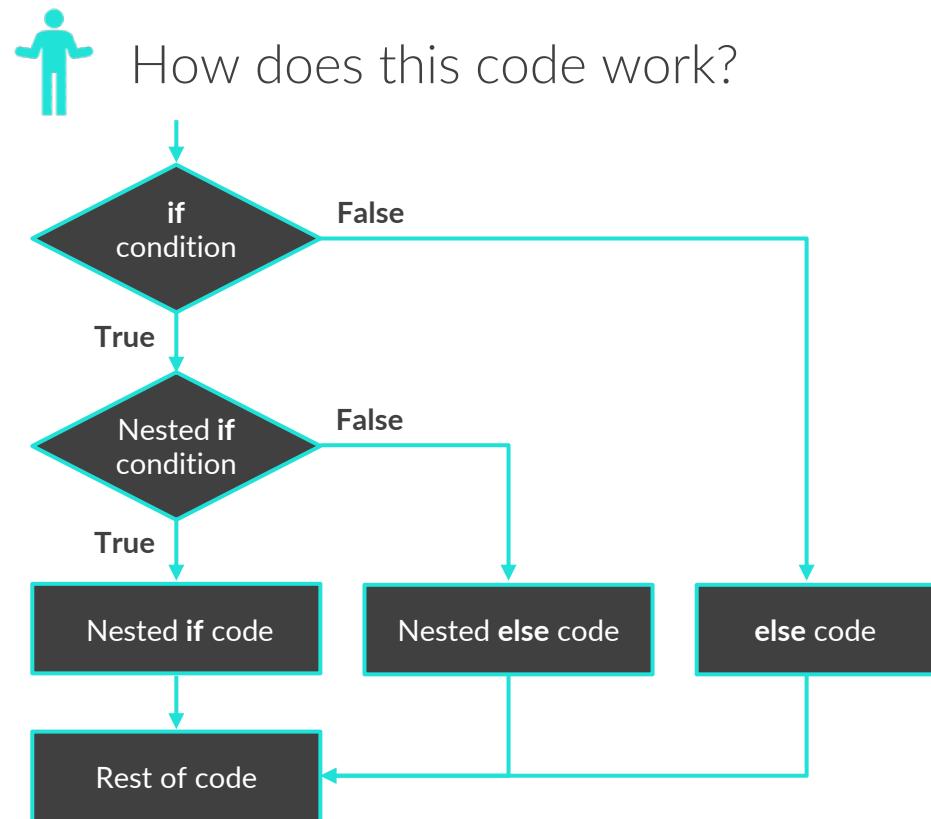
**Nested if statements** let you specify additional criteria to evaluate after a logical condition in an if statement is met

## EXAMPLE

Trying to purchase a product

```
price = 499.99
budget = 500
inventory = 0

if budget > price:
    if inventory > 0:
        print('You can afford this and we have it in stock!')
    else: # equivalent to if inventory <= 0
        print("You can afford this but it's out of stock.")
else:
    print(f'Unfortunately, this board costs more than ${budget}.')
You can afford this but it's out of stock.
```



# ASSIGNMENT: NESTED IF STATEMENTS

 **NEW MESSAGE**  
February 6, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Inventory Query Edit**

Hi there, we need to make one final change to our inventory logic for Chris, our owner:

- If inventory is 0 or less, and if the customer's name is 'Chris', print 'You can have the display model, sir'
- Otherwise, print 'Out of stock'

The rest of the logic should stay the same.

Thanks again!

 [inventory\\_query2.ipynb](#)

 [Reply](#)    [Forward](#)

**Results Preview**

```
inventory = 0
customer_name = 'Chris'

You can have the display model, sir
```

# ASSIGNMENT: NESTED IF STATEMENTS

 NEW MESSAGE  
February 6, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Inventory Query Edit**

Hi there, we need to make one final change to our inventory logic for Chris, our owner:

- If inventory is 0 or less, and if the customer's name is 'Chris', print 'You can have the display model, sir'
- Otherwise, print 'Out of stock'

The rest of the logic should stay the same.

Thanks again!

 inventory\_query2.ipynb     Reply     Forward

## *Solution Code*

### *With Nesting:*

```
inventory = 0
customer_name = 'Chris'

if inventory <= 0:
    if customer_name == 'Chris':
        print('You can have the display model, sir')
    else:
        print('Out of stock')
elif inventory <= 5:
    print('Low Stock')
else:
    print('In Stock')
```

You can have the display model, sir

### *Without Nesting:*

```
inventory = 0
customer_name = 'Chris'

if customer_name == 'Chris' and inventory <= 0:
    print('I can sell you the display model, sir.')
elif inventory <= 0:
    print('Out of stock')
elif inventory <= 5:
    print('Low Stock')
else:
    print('In Stock')
```

I can sell you the display model, sir.

# KEY TAKEAWAYS

---



## The Boolean data type is the foundation of conditional logic

- The only two values for the Boolean data type are True and False, which are equivalent to 1 and 0, respectively, when performing arithmetic operations



## You can write logical statements using comparison operators, membership tests and Boolean operators

- Comparison operators return a Boolean value and include equal, not equal, greater than, and less than
- Membership tests determine if a specified value is in an iterable, and return a Boolean value
- Boolean operators (and, or) let you evaluate multiple logical statements as a single condition



## Conditional control flow lets you determine which lines of code to execute

- Conditional statements include IF, ELIF, and ELSE, which can be nested for multiple layers of logic

# SEQUENCE DATA TYPES

# SEQUENCE DATA TYPES



In this section we'll cover **sequence data types**, including lists, tuples, and ranges, which are capable of storing many values

## TOPICS WE'LL COVER:

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

## GOALS FOR THIS SECTION:

- Learn to create lists and modify list elements
- Apply common list functions and methods
- Understand the different methods for copying lists
- Review the benefits of using tuples and ranges



# LISTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

**Lists** are iterable data types capable of storing many individual elements

- List elements can be any data type, and can be added, changed, or removed at any time

```
random_list = ['snowboard', 10.54, None, True, -1]
```

Lists are created with square brackets []

List elements are separated by commas

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
```

} While lists can contain multiple data types, they often contain **one data type**

```
empty_list = []
empty_list
[]
```

} You can create an empty list and add elements later

```
list('Hello')
['H', 'e', 'l', 'l', 'o']
```

} You can create lists from other iterable data types



# MEMBERSHIP TESTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Since lists are iterable data types, you can conduct **membership tests** on them

```
sizes_in_stock = ['XS', 'S', 'L', 'XL', 'XXL']  
'M' in sizes_in_stock
```

False

{ 'M' is not an element in *sizes\_in\_stock*, so False is returned }

```
if 'M' in sizes_in_stock:  
    print("I'll take the medium please.")  
elif 'S' in sizes_in_stock:  
    print("It'll be tight but small will do.")  
else:  
    print("I'll wait until you have medium.")
```

It'll be tight but small will do.

{ 'M' is not an element in *sizes\_in\_stock*, but 'S' is, so the print function under elif is executed }



# LIST INDEXING

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Elements in a list can be accessed via their **index**, or position within the list

- Remember that Python is 0-indexed, so the first element has an index of 0, not 1

```
item_list = [ 'Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings' ]  
item_list[0]  
'Snowboard'
```

An index of 0 grabs the first list element

```
item_list[3]  
'Goggles'
```

An index of 3 grabs the fourth list element



# LIST SLICING

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

**Slice notation** can also be used to access portions of lists

[start: stop: step size]

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
```

```
item_list[:3]
```

```
['Snowboard', 'Boots', 'Helmet']
```

*list[3]* grabs elements 0, 1, and 2

Remember that 'stop' is non-inclusive!

```
item_list[::-2]
```

```
['Snowboard', 'Helmet', 'Bindings']
```

*list[::2]* grabs every other element in the list

```
item_list[2:4]
```

```
['Helmet', 'Goggles']
```

*list[2:4]* grabs the 3<sup>rd</sup> (index of 2) and 4<sup>th</sup> (index of 3) elements in the list



# UNPACKING LISTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Lists elements can be **unpacked** into individual variables

```
item_list = ['Snowboard', 'Boots', 'Helmet']
s, b, h = item_list
print(s, b, h)
```

Snowboard Boots Helmet

*s, b, and h are now string variables*



You need to specify the same number of variables as list elements or you will receive a **ValueError**

# ASSIGNMENT: LIST OPERATIONS

 **NEW MESSAGE**  
February 7, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Customer Lists

Hi there!

Can you create a list from the customers that made purchases on Black Friday (attached) and:

- Return 99.99 if C00009 made a purchase; otherwise return 0.0
- Create separate lists for the 5<sup>th</sup> and 6<sup>th</sup> customers, every third customer, and the last 2 customers, we're conducting market research.

Thanks again!

 *black\_friday\_customers.ipynb* Reply Forward

## Results Preview

```
customer_list = [C00001, C00003, C00004, C00005, C00006, C00007, C00018, C00010, C00013, C00014, C00015, C00016, C00029]
```

\* 99.99  
0.0

```
fifth_sixth = [C00006, C00007]
```

```
every_third = [C00001, C00005, C00018, C00014, C00029]
```

```
last_two = [C00016, C00020]
```

# ASSIGNMENT: LIST OPERATIONS

 **1 NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** Customer Lists

Hi there!

Can you create a list from the customers that made purchases on Black Friday (attached) and:

- Return 99.99 if C00009 made a purchase; otherwise return 0.0
- Create separate lists for the 5<sup>th</sup> and 6<sup>th</sup> customers, every third customer, and the last 2 customers, we're conducting market research.

Thanks again!

 black\_friday\_customers.ipynb Reply Forward

## Solution Code

```
customer_list = ['C00001', 'C00003', 'C00004', 'C00005', 'C00006',
                 'C00007', 'C00018', 'C00010', 'C00013', 'C00014',
                 'C00015', 'C00016', 'C00029']
```

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00018',
 'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00029']
```

```
('C00009' in customer_list) * 99.99
```

```
0.0
```

```
fifth_sixth = customer_list[4:6]
```

```
fifth_sixth
```

```
['C00006', 'C00007']
```

```
every_third = customer_list[::3]
```

```
every_third
```

```
['C00001', 'C00005', 'C00018', 'C00014', 'C00029']
```

```
last_two = customer_list[-2:]
last_two
```

```
['C00016', 'C00020']
```



# CHANGING LIST ELEMENTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Lists elements can be **changed**, but not added, by using indexing

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items[1] = 'Gloves'
new_items
['Coat', 'Gloves', 'Snowpants']
```

The second element in the list has changed from 'Backpack' to 'Gloves'

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items[3] = 'Gloves'
new_items
```

The list only has 3 elements, so an index of 3 (the fourth element) does not exist

**IndexError: list assignment index out of range**



# ADDING LIST ELEMENTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

You can **.append()** or **.insert()** a new element to a list

- **.append(element)** – adds an element to the end of the list
- **.insert(index, element)** – adds an element to the specified index in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_list.append('Coat')
print(item_list)
```

```
['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat']
```

'Coat' was added as the last element in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_list.insert(3, 'Coat')
item_list
```

```
['Snowboard', 'Boots', 'Helmet', 'Coat', 'Goggles', 'Bindings']
```

'Coat' was added as the fourth element in the list



# COMBINING & REPEATING LISTS

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

Lists can be **combined**, or concatenated, with **+** and **repeated** with **\***

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
new_items = ['Coat', 'Backpack', 'Snowpants']

all_items = item_list + new_items
print(all_items)

['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat',
'Backpack', 'Snowpants']
```

```
new_items * 3

['Coat',
'Backpack',
'Snowpants',
'Coat',
'Backpack',
'Snowpants',
'Coat',
'Backpack',
'Snowpants']
```



# REMOVING LIST ELEMENTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

There are two ways to **remove** lists elements:

1. The **del** keyword deletes the selected elements from the list

```
new_items = ['Coat', 'Backpack', 'Snowpants']
del new_items[1:3]
new_items
['Coat']
```

The second (index of 1) and third (index of 2) elements were deleted from the list

- **del list\_name(slice)**

2. The **.remove()** method deletes the first occurrence of the specified value from the list

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items.remove('Coat')
new_items
['Backpack', 'Snowpants']
```

'Coat' was removed from the list

- **.remove(value)**

# ASSIGNMENT: ADDING LIST ELEMENTS

 **NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Adding Customers**

Hi again!

We just found a receipt for customer C00009 in the warehouse, can you add them to the customer list?

While you're at it, we decided to extend the sale through Saturday since things were so crazy.

Can you add the customers in saturday\_list to customer\_list?

Thank you!

 [updated\\_customer\\_lists.ipynb](#)

 [Reply](#)    [Forward](#)

## Results Preview

Adding C00009:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',  
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Adding saturday\_list:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',  
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009',  
'C00004', 'C00017', 'C00019', 'C00002', 'C00008', 'C00021', 'C00022']
```

# ASSIGNMENT: ADDING LIST ELEMENTS

 NEW MESSAGE  
February 7, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Adding Customers

Hi again!

We just found a receipt for customer C00009 in the warehouse, can you add them to the customer list?

While you're at it, we decided to extend the sale through Saturday since things were so crazy.

Can you add the customers in saturday\_list to customer\_list?

Thank you!

 updated\_customer\_lists.ipynb

## *Solution Code*

Adding C00009 with .append():

```
customer_list.append('C00009')

print(customer_list)

['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
 'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Adding C00009 with .insert():

```
customer_list.insert(7, 'C00009')

print(customer_list)

['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
 'C00009', 'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020']
```

Adding saturday\_list

```
customer_list = customer_list + saturday_list

print(customer_list)

['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
 'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009',
 'C00004', 'C00017', 'C00019', 'C00002', 'C00008', 'C00021', 'C00022']
```

# ASSIGNMENT: REMOVING LIST ELEMENTS

 **NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Change of plan!**

Hi there!

Sorry to do this to you, but we decided to keep the Friday and Saturday sales separate for now. Can you remove Saturday's customers from the customer list?

Also, C00004 is a friend of the owner and should not be counted in the Black Friday sales, so please remove them from the list.

Thanks again!

 [updated\\_updated\\_customer\\_lists.ipynb](#)

 [Reply](#)    [Forward](#)

## *Results Preview*

Removing Saturday's customers:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Removing C00004:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00005', 'C00006', 'C00007', 'C00008', 'C00010',
'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

# ASSIGNMENT: REMOVING LIST ELEMENTS

 **NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Change of plan!**

Hi there!

Sorry to do this to you, but we decided to keep the Friday and Saturday sales separate for now. Can you remove Saturday's customers from the customer list? Also, C00004 is a friend of the owner and should not be counted in the Black Friday sales, so please remove them from the list.

Thanks again!

 *updated\_updated\_customer\_lists.ipynb*

 Reply     Forward

## *Solution Code*

Removing Saturday's customers (two methods):

```
del customer_list[-7:]  
  
print(customer_list)  
  
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',  
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

```
customer_list = customer_list[:14]  
  
print(customer_list)  
  
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',  
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Removing C00004:

```
customer_list.remove('C00004')  
  
print(customer_list)  
  
['C00001', 'C00003', 'C00005', 'C00006', 'C00007', 'C00008', 'C00010',  
'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```



# LIST FUNCTIONS

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

**len(list\_name)** Returns the number of elements in a list

```
len(transactions)
```

6

**min(list\_name)** Returns the smallest element in the list

```
min(transactions)
```

2.07

**sum(list\_name)** Returns the sum of the elements in the list

```
sum(transactions)
```

1483.88

**max(list\_name)** Returns the largest element in the list

```
max(transactions)
```

1242.66



# SORTING LISTS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

There are two ways to **sort** lists elements:

1. The **.sort()** method sorts the list permanently (*in place*)

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]  
transactions.sort()  
transactions  
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]
```



**PRO TIP:** Don't sort in place until you're positive the code works as expected and you no longer need to preserve the original list

2. The **sorted** function returns a sorted list, but does not change the original (*not in place*)

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]  
sorted(transactions)  
  
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]  
  
transactions  
[10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```



# LIST METHODS

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

**.index(*value*)** Returns the index of a specified value within a list (returns -1 if not found)

```
transactions.index(200.14)
```

2

**.count()** Counts the number of times a given value occurs in a list

```
transactions.count(200.14)
```

1

**.reverse()** Reverses the order of the list elements in place

```
transactions.reverse()  
transactions
```

[8.01, 2.07, 1242.66, 200.14, 20.56, 10.44]



**PRO TIP:** Use a negative slice to reverse the order “not in place”

# ASSIGNMENT: LIST FUNCTIONS & METHODS

 **NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Customer C00010 Transactions**

I've attached a file with two lists: customer\_IDs and subtotals. The index for customer ID matches the index for the subtotal on each of their transactions.  
(*customer\_ids[0] made the transaction subtotals[0]*)

Can you look into customer C00010's behavior?

1. Calculate the average value of a transaction
2. Report how many transactions C00010 made
3. Look up the value of their first transaction
4. Compare it to the average value
5. Print a list of sorted IDs (don't change the list!)

 *customer\_transactions.ipynb*

Reply

Forward

## Results Preview

Average transaction value:

323.3877777777776

C00010's transactions:

2

First transaction index:

5

First transaction value:

599.99

Sorted ID list:

```
['C00001', 'C00001', 'C00001', 'C00002', 'C00003', 'C00004',
'C00004', 'C00005', 'C00006', 'C00006', 'C00007', 'C00008',
'C00008', 'C00010', 'C00010', 'C00013', 'C00014', 'C00015',
'C00016', 'C00016', 'C00017', 'C00018', 'C00018', 'C00019',
'C00020', 'C00021', 'C00022']
```

# ASSIGNMENT: LIST FUNCTIONS & METHODS

 **NEW MESSAGE**  
February 7, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Customer C00010 Transactions

I've attached a file with two lists: customer\_IDs and subtotals. The index for customer ID matches the index for the subtotal on each of their transactions.  
(*customer\_ids[0] made the transaction subtotals[0]*)

Can you look into customer C00010's behavior?

1. Calculate the average value of a transaction
2. Report how many transactions C00010 made
3. Look up the value of their first transaction
4. Compare it to the average value
5. Print a list of sorted IDs (don't change the list!)

 customer\_transactions.ipynb

Reply

Forward

## Solution Code

Average transaction value:

```
sum(subtotals)/len(subtotals)
```

323.3877777777776

C00010's transactions:

```
customer_ids.count('C00010')
```

2

First transaction index:

```
customer_ids.index('C00010')
```

5

First transaction value:

```
subtotals[5]
```

599.99



Looks higher than the average!

Sorted ID list:

```
print(sorted(customer_ids))
```

```
['C00001', 'C00001', 'C00001', 'C00002', 'C00003', 'C00004',
'C00004', 'C00005', 'C00006', 'C00006', 'C00007', 'C00008',
'C00008', 'C00010', 'C00010', 'C00013', 'C00014', 'C00015',
'C00016', 'C00016', 'C00017', 'C00018', 'C00018', 'C00019',
'C00020', 'C00021', 'C00022']
```

Using .sort() would have changed the list permanently!



# NESTED LISTS

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

Lists stored as elements of another list are known as **nested lists**

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
# grab the second element (a list)
list_of_lists[1]
['d', 'e', 'f']
```

Referencing a single index value will return an entire nested list

```
# grab the second element of the second element
list_of_lists[1][1]
'e'
```

Adding a second index value will return individual elements from nested lists



# NESTED LISTS

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

List **methods** & **functions** still work with nested lists

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
list_of_lists[1].append('k')
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f', 'k'], ['g', 'h', 'i']]
```

```
list_of_lists[2].count('h')
1
```

'k' is being added as the last element to the second list

'h' appears once in the third list



# COPYING LISTS

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

There are 3 different ways to **copy** lists:

1. **Variable assignment** – assigning a list to a new variable creates a “view”
  - Any changes made to one of the lists will be reflected in the other
2. **.copy()** – applying this method to a list creates a ‘shallow’ copy
  - Changes to entire elements (nested lists) will not carry over between original and copy
  - Changes to individual elements within a nested list will still be reflected in both
3. **deepcopy()** – using this function on a list creates entirely independent lists
  - Any changes made to one of the lists will NOT impact the other



**deepcopy** is overkill for the vast majority of uses cases, but worth being aware of



# VARIABLE ASSIGNMENT

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

Copying a list via **variable assignment** creates a “view” of the list

- Both variables point to the same object in memory
- Changing an element in one list will result in a change to the other

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]

# copy list using variable assignment
list_of_lists2 = list_of_lists

# change element in original list
list_of_lists[1] = ['x', 'y', 'z']

# The change will be reflected in the copy made by assignment
list_of_lists2

[['a', 'b', 'c'], ['x', 'y', 'z'], ['g', 'h', 'i']]
```

The change made to `list_of_lists` is reflected in `list_of_lists2`



# COPY

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Copying a list with the `.copy()` method creates a copy of the list

- Changes to entire elements (nested lists) will not carry over between original and copy
- Changes to individual elements within a nested list will still be reflected in both

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]

# use the copy method to create a copy
list_of_lists2 = list_of_lists.copy()

# replace the entire nested listed at index 0
list_of_lists[0] = ['x', 'y', 'z']

list_of_lists2
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

Since the entire nested list at index 0 was replaced, the change is NOT reflected in the copy

```
# change the second element of the first nested list
list_of_lists[0][1] = 'Oh no!'

list_of_lists2
[['a', 'Oh no!', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

Since a single element (index of 1) within the nested list was modified, the change IS reflected in the copy



# DEEPCOPY

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

Copying a list with the **deepcopy()** function creates a separate copy of the list

- Any changes made to one of the lists will NOT impact the other

This function is not part of base Python, so the *copy* library must be imported

```
from copy import deepcopy  
  
list_of_lists = [['a', 'b', 'c'],  
                 ['d', 'e', 'f'],  
                 ['g', 'h', 'i']]  
  
list_of_lists2 = deepcopy(list_of_lists)  
  
list_of_lists[0][1] = 'Oh no!'  
  
list_of_lists2  
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

The change is NOT reflected in the copy, even though a single element (index of 1) within the nested list was modified

# ASSIGNMENT: NESTED LISTS & COPYING

 NEW MESSAGE  
February 7, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Customer C00010 Transactions**

I've attached a file with five lists, each for a specific customer's orders. We're interested in these customers because they have multiple transactions.

Can you help:

- Create a list with these 5 lists as elements
- Print the second and third orders for the customer that made three total orders
- Create a copy of this list and change the transaction values for C00004 (friend of the owner) to 0.0 – without changing the original!

 multi\_order\_customers.ipynb

Reply

Forward

## Results Preview

```
orders_c00001 = [1799.94, 29.98, 99.99]
orders_c00004 = [15.98, 119.99]
orders_c00006 = [24.99, 24.99]
orders_c00008 = [649.99, 99.99]
orders_c00010 = [599.99, 399.97]
```

```
vip_list =  
  
vip_list  
[[1799.94, 29.98, 99.99],  
 [15.98, 119.99],  
 [24.99, 24.99],  
 [649.99, 99.99],  
 [599.99, 399.97]]
```

```
vip_list  
[29.98, 99.99]
```

```
revenue_adjusted_list =
```

```
revenue_adjusted_list  
[[1799.94, 29.98, 99.99],  
 [0.0, 0.0],  
 [24.99, 24.99],  
 [649.99, 99.99],  
 [599.99, 399.97]]
```

# ASSIGNMENT: NESTED LISTS & COPYING

 NEW MESSAGE  
February 7, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Customer C00010 Transactions

I've attached a file with five lists, each for a specific customer's orders. We're interested in these customers because they have multiple transactions.

Can you help:

- Create a list with these 5 lists as elements
- Print the second and third orders for the customer that made three total orders
- Create a copy of this list and change the transaction values for C00004 (friend of the owner) to 0.0 – without changing the original!

 multi\_order\_customers.ipynb

Reply

Forward

## Solution Code

```
orders_c00001 = [1799.94, 29.98, 99.99]
orders_c00004 = [15.98, 119.99]
orders_c00006 = [24.99, 24.99]
orders_c00008 = [649.99, 99.99]
orders_c00010 = [599.99, 399.97]
```

```
vip_list = [orders_c00001, orders_c00004, orders_c00006, orders_c00008, orders_c00010]
```

```
vip_list
```

```
[[1799.94, 29.98, 99.99],
 [15.98, 119.99],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```

```
vip_list[0][1:]
```

```
[29.98, 99.99]
```

```
revenue_adjusted_list = vip_list.copy()
```

```
revenue_adjusted_list[1] = [0.0,0.0]
```

```
revenue_adjusted_list
```

```
[[1799.94, 29.98, 99.99],
 [0.0, 0.0],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```

```
# our original list should be unchanged
vip_list
```

```
[[1799.94, 29.98, 99.99],
 [15.98, 119.99],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```



# TUPLES

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

**Tuples** are iterable data types capable of storing many individual items

- Tuples are very similar to lists, except they are **immutable**
- Tuple items can still be any data type, but they CANNOT be added, changed, or removed once the tuple is created

```
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')  
item_tuple  
( 'Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings' )
```

Tuples are created with parenthesis (), or the *tuple()* function

```
item_tuple[3]
```

```
'Goggles'
```

```
item_tuple[:3]
```

```
( 'Snowboard', 'Boots', 'Helmet' )
```

```
len(item_tuple)
```

List operations that don't modify their elements work with tuples as well



# WHY USE TUPLES?

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

## 1. Tuples require **less memory** than a list

```
import sys

item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')

print(f"The list is {sys.getsizeof(item_list)} + 'Bytes'")
print(f"The tuple is {sys.getsizeof(item_tuple)} + 'Bytes'")
```

```
The list is 120 Bytes
The tuple is 80 Bytes
```

The tuple is **33% smaller** than the list

For heavy data processing, this can be a big difference

## 2. Operations **execute quicker** on tuples than on lists

```
import timeit

# calculate time of summing list 10000 times
print(timeit.timeit("sum([10.44, 20.56, 200.14, 1242.66, 2.07, 8.01])",
                     number=10000))

# calculate time of summing tuple 10000 times
print(timeit.timeit("sum((10.44, 20.56, 200.14, 1242.66, 2.07, 8.01))",
                     number=10000))
```

```
0.0076123750004626345
0.003229583000575076
```

The tuple executed over **twice as fast** as the list



# WHY USE TUPLES?

List Basics

List Operations

Modifying Lists

List Functions  
& Methods

Nested Lists

Copying Lists

Tuples

Ranges

3. Tuples **reduce user error** by preventing modification to data
  - There are cases in which you explicitly do not want others to be able to modify data

4. Tuples are common output in **imported functions**

```
a = 1
b = 2
c = 3

a, b, c
```

```
(1, 2, 3)
```

Even asking to return multiple variables in a code cell returns them as a tuple

# ASSIGNMENT: TUPLES

 **NEW MESSAGE**  
February 7, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Sales Tax**

Hi there, we need to apply sales tax to each of customer C00001's transactions.

Can you grab these from multi\_order\_list, create a tuple, and unpack the tuple into three variables: transaction1, transaction2, and transaction3?

Then we just need to multiply them by 0.08 to calculate the sales tax of 8%, and round to the nearest cent.

Thanks in advance!

 [sales\\_tax.ipynb](#)

 [Reply](#)    [Forward](#)

## Results Preview



144.0  
2.4  
8.0

# ASSIGNMENT: TUPLES

 **NEW MESSAGE**  
February 7, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Sales Tax**

Hi there, we need to apply sales tax to each of customer C00001's transactions.

Can you grab these from multi\_order\_list, create a tuple, and unpack the tuple into three variables: transaction1, transaction2, and transaction3?

Then we just need to multiply them by 0.08 to calculate the sales tax of 8%, and round to the nearest cent.

Thanks in advance!

 sales\_tax.ipynb

Reply

Forward

## *Solution Code*

```
transaction1, transaction2, transaction3 = tuple(multi_order_customers[0])  
  
print(round(transaction1 * .08, 2))  
print(round(transaction2 * .08, 2))  
print(round(transaction3 * .08, 2))
```

144.0

2.4

8.0



# RANGES

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

**Ranges** are sequences of integers generated by a given start, stop, and step size

- They are more memory efficient than tuples, as they don't generate the integers until needed
- They save time, as you don't need to write the list of integers manually in the code
- They are commonly used with loops (*more on that later!*)

```
example_range = range(1, 5, 1)
```

```
print(example_range)
```

```
range(1, 5)
```

```
print(list(example_range))
```

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

```
print(tuple(range(len('Hello'))))
```

```
(0, 1, 2, 3, 4)
```

Note that printing a range does NOT return the integers

You can retrieve them by converting to a list or tuple

# ASSIGNMENT: RANGES

  **NEW MESSAGE**  
February 7, 2022

**From:** **Alfie Alpine** (Marketing Manager)  
**Subject:** Customer Giveaway

We are running a special email campaign to give products away to lucky customers who make a purchase.

We need the following integers generated:

- Even numbers in our first 10 customers (starting with our 2<sup>nd</sup> customer). They will receive a coffee mug.
- Odd numbers in our first 10 customers (starting with our 1<sup>st</sup> customer). They will get keychains.
- Every seventh customer in our first 100 customers (starting with our 7<sup>th</sup> customer). They get beanies!

 [customer\\_promo\\_generator.ipynb](#) Reply Forward

## Results Preview

Even Customers:

```
print
```

```
[2, 4, 6, 8, 10]
```

Odd Customers:

```
print
```

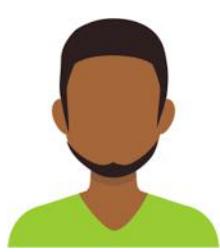
```
[1, 3, 5, 7, 9]
```

Every 7<sup>th</sup> Customer:

```
print
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

# ASSIGNMENT: RANGES



## NEW MESSAGE

February 7, 2022

From: **Alfie Alpine** (Marketing Manager)

Subject: Customer Giveaway

We are running a special email campaign to give products away to lucky customers who make a purchase.

We need the following integers generated:

- Even numbers in our first 10 customers (starting with our 2<sup>nd</sup> customer). They will receive a coffee mug.
- Odd numbers in our first 10 customers (starting with our 1<sup>st</sup> customer). They will get keychains.
- Every seventh customer in our first 100 customers (starting with our 7<sup>th</sup> customer). They get beanies!

customer\_promo\_generator.ipynb

Reply

Forward

## Solution Code

Even Customers:

```
print(list(range(2, 12, 2)))
```

```
[2, 4, 6, 8, 10]
```

Odd Customers:

```
print(list(range(1, 11, 2)))
```

```
[1, 3, 5, 7, 9]
```

Every 7<sup>th</sup> Customer:

```
print(list(range(7, 100, 7)))
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

# KEY TAKEAWAYS

---



Lists and tuples are sequence data types capable of storing many values

- *Lists allow you to add, remove, and change elements, while tuples do not*



Use **indexing** and **slicing** to access & modify list elements

- *Make sure you are comfortable with this syntax, as you will use it frequently in data analysis libraries as well*



Built-in **functions** and **methods** make working with sequences easier

- *All the list functions covered work with tuples as well, and tuples have many identical methods, like index() and count(). Take a minute to skim through the available methods in the Python documentation.*



Ranges help create a **sequence of integers** efficiently

- *These will become particularly powerful when combined with loops*

# LOOPS

# LOOPS



In this section we'll introduce the concept of **loops**, review different loop types and their components, and cover control statements for refining their logic and handling errors

## TOPICS WE'LL COVER:

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## GOALS FOR THIS SECTION:

- Understand different types of loop structures, their components, and common use cases
- Learn to loop over multiple iterable data types in single and nested loop scenarios
- Explore common control statements for modifying loop logic and handling errors



# LOOP BASICS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **loop** is a block of code that will repeat until a given condition is met

There are **two types** of loops:

## FOR LOOPS

- Run a specified number of times
- “**For**this many times”
- This often corresponds with the length of a list, tuple, or other iterable data type
- Should be used when you know how many times the code should run

## WHILE LOOPS

- Run until a logical condition is met
- “**While**this is TRUE”
- You usually don’t know how many times this loop will run, which can lead to infinite loop scenarios  
*(more on that later!)*
- Should be used when you don’t know how many times the code should run



# LOOP BASICS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **loop** is a block of code that will repeat until a given condition is met

## EXAMPLE

Converting elements in a price list from USD to Euros

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]

euro_list = [round(usd_list[0] * exchange_rate, 2),
            round(usd_list[1] * exchange_rate, 2),
            round(usd_list[2] * exchange_rate, 2),
            round(usd_list[3] * exchange_rate, 2),
            round(usd_list[4] * exchange_rate, 2)]

euro_list

[5.27, 8.79, 17.59, 21.99, 87.99]
```

In this example we're taking each element in our **usd\_list** and multiplying it by the exchange rate to convert it to euros

- Notice that we had to write a line of code for *each element in the list*
- Imagine if we had hundreds or thousands of prices!



# LOOP BASICS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **loop** is a block of code that will repeat until a given condition is met

## EXAMPLE

Converting elements in a price list from USD to Euros

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

print(euro_list)

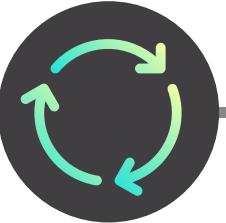
[5.27, 8.79, 17.59, 21.99, 87.99]
```

Now we're using a **For Loop** to cycle through each element in the **usd\_list** and convert it to Euros

- We only used two *lines of code* to process the entire list!



Don't worry if the code looks confusing now (we'll cover loop syntax shortly), the key takeaway is that we wrote the conversion **one time** and it was applied until we looped through **all the elements** in the list



# FOR LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **for loop** will run a specified number of times

- This often corresponds with the length of a list, tuple, or other iterable data type

`for item in iterable:`

Indicates a  
For Loop

A variable name for each  
item in the iterable

Iterable object to  
loop through

*Examples:*

- Price
- Product
- Customer

*Examples:*

- List
- Tuple
- String
- Dictionary
- Set



# FOR LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **for loop** will run a specified number of times

- This often corresponds with the length of a list, tuple, or other iterable data type

```
for item in iterable:  
    do this
```

*Code to run until the loop terminates (must be indented!)*

*Run order:*

1. *item = iterable[0]*
2. *item = iterable[1]*
3. *item = iterable[2]*
- 
- 
- n. item = iterable[len(iterable)-1]*



# FOR LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

*Printing individual letters in a string*

```
iterable = 'Maven'  
for item in iterable:  
    print(item)
```

M ← item = iterable[0]  
a ← item = iterable[1]  
v ← item = iterable[2]  
e ← item = iterable[3]  
n ← item = iterable[4]



How does this code work?

- Since '**Maven**' is a string, each letter is an **item** we'll iterate through
- **iterable** = ['M', 'a', 'v', 'e', 'n']
- The code will run and **print** each **item** in the **iterable**



# FOR LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing individual letters in a string

```
word = 'Maven'  
for letter in word:  
    print(letter)
```

M ← letter = word[0]  
a ← letter = word[1]  
v ← letter = word[2]  
e ← letter = word[3]  
n ← letter = word[4]



How does this code work?

- Since '**Maven**' is a string, we'll iterate through each **letter**
- **word** = ['M', 'a', 'v', 'e', 'n']
- The code will run and **print** each **letter** in the **word**



**PRO TIP:** Give the components of your loop intuitive names so they are easier to understand



# LOOPING OVER ITEMS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

**Looping over items** will run through the items of an iterable

- The loop will run as many times as there are items in the iterable

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

print(euro_list)

[5.27, 8.79, 17.59, 21.99, 87.99]
```

The for loop here is looping over the items, (elements) of **usd\_list**, so the loop code block runs 5 times (length of the list):

1. price = usd\_list[0] = 5.99
2. price = usd\_list[1] = 9.99
3. price = usd\_list[2] = 19.99
4. price = usd\_list[3] = 24.99
5. price = usd\_list[4] = 99.99



**PRO TIP:** To create a new list (or other data type) with loops, first create an empty list, then append values as your loop iterates



# LOOPING OVER INDICES

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

**Looping over indices** will run through a range of integers

- You need to specify a range (usually the length of an iterable)
- This range can be used to navigate the indices of iterable objects

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for i in range(len(usd_list)):
    euro_list.append(round(usd_list[i] * exchange_rate, 2))

print(euro_list)
[5.27, 8.79, 17.59, 21.99, 87.99]
```

The index is selecting the elements in the list

The for loop here is looping over indices in a range the size of the *euro\_list*, meaning that the code will run 5 times (length of the list):

1.  $i = 0$
2.  $i = 1$
3.  $i = 2$
4.  $i = 3$
5.  $i = 4$



**PRO TIP:** If you only need to access the elements of a single iterable, it's a best practice to loop over items instead of indices



# LOOPING OVER MULTIPLE ITERABLES

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

Looping over indices can help with **looping over multiple iterables**, allowing you to use the same index for items you want to process together

## EXAMPLE

Printing the price for each inventory item

```
euro_list = [5.27, 8.79, 17.59, 21.99, 87.99]
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

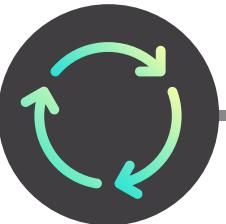
for i in range(len(euro_list)):
    print(f"The {item_list[i].lower()} costs {euro_list[i]} euros.")
```

```
The snowboard costs 5.27 euros.
The boots costs 8.79 euros.
The helmet costs 17.59 euros.
The goggles costs 21.99 euros.
The bindings costs 87.99 euros.
```

The for loop here is looping over indices in a range the size of the **euro\_list**, meaning that the code will run 5 times (length of the list)

For the first run:

- $i = 0$
- $item\_list[i] = \text{Snowboard}$
- $euro\_list[i] = 5.27$



# PRO TIP: ENUMERATE

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

The **enumerate** function will return both the index and item of each item in an iterable as it loops through

```
for index, element in enumerate(euro_list):
    print(index, element)
```

```
0 5.27
1 8.79
2 17.59
3 21.99
4 87.99
```

Use the index for  
multiple lists!

```
euro_list = [5.27, 8.79, 17.59, 21.99, 87.99]
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

for index, element in enumerate(euro_list):
    print(item_list[index], element)
```

```
Snowboard 5.27
Boots 8.79
Helmet 17.59
Goggles 21.99
Bindings 87.99
```



**PRO TIP:** Use enumerate if you want to loop over an index; it is slightly more efficient and considered best practice, as we are looping over an index derived from the list itself, rather than generating a new object to do so.

We're using the **index** of the euro\_list to access each element from the item\_list, and then printing each **element** of the euro\_list

# ASSIGNMENT: FOR LOOPS

 **NEW MESSAGE**  
February 8, 2022

**From:** Sally Snow (Ski Shop Manager)  
**Subject:** Sales Tax

Good morning,

Our billing system failed to apply sales tax to our transactions! Can you add a sales tax of 8% to each of the subtotals?

Once you've calculated tax, calculate the total by adding the tax to the subtotal, and create lists to store both the tax amounts and the totals.

Thanks in advance!

 taxable\_transactions.ipynb     Reply     Forward

## Results Preview

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99, 24.99, 1799.94, 99.99]
```

```
taxes = []
totals = []
```

```
print(taxes)
print(totals)
```

```
[1.28, 72.0, 64.0, 9.44, 0.48, 48.0, 2.0, 144.0, 8.0]
[17.26, 971.97, 863.97, 127.4, 6.47, 647.99, 26.99, 1943.94, 107.99]
```

1.28 = round(subtotal[0] \* .08, 2)

17.26 = round(1.28 + subtotal[0], 2)

# ASSIGNMENT: FOR LOOPS

 **NEW MESSAGE**

February 8, 2022

**From:** Sally Snow (Ski Shop Manager)

**Subject:** Sales Tax

Good morning,

Our billing system failed to apply sales tax to our transactions! Can you add a sales tax of 8% to each of the subtotals?

Once you've calculated tax, calculate the total by adding the tax to the subtotal, and create lists to store both the tax amounts and the totals.

Thanks in advance!

## Solution Code

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99, 24.99, 1799.94, 99.99]
```

```
taxes = []
totals = []

for subtotal in subtotals:
    tax = round(subtotal * .08, 2)
    total = round(subtotal + tax, 2)
    taxes.append(tax)
    totals.append(total)

print(taxes)
print(totals)
```

```
[1.28, 72.0, 64.0, 9.44, 0.48, 48.0, 2.0, 144.0, 8.0]
[17.26, 971.97, 863.97, 127.4, 6.47, 647.99, 26.99, 1943.94, 107.99]
```

# ASSIGNMENT: ENUMERATE

 **NEW MESSAGE**  
February 8, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Sales Tax By State**

Hi again!

I forgot to mention we need to apply different tax rates to different locations. The list attached has the location of each transaction. Our Sun Valley store is taxed at 8%, Stowe at 6%, and Mammoth at 7.75%.

Can you adjust your tax calculator code to apply the correct tax to each location?

Thanks!

 taxable\_transactions2.ipynb     Reply     Forward

## Results Preview

```
location = ['Sun Valley', 'Stowe', 'Mammoth',
            'Stowe', 'Sun Valley', 'Mammoth',
            'Mammoth', 'Mammoth', 'Sun Valley']
```

```
taxes = []
totals = []

for i, subtotal in enumerate(subtotals):
```

```
    print(taxes)
    print(totals)
```

```
[1.28, 54.0, 62.0, 7.08, 0.48, 46.5, 1.94, 139.5, 8.0]
[17.26, 953.97, 861.97, 125.04, 6.47, 646.49, 26.93, 1939.44, 107.99]
```

# ASSIGNMENT: ENUMERATE

 NEW MESSAGE  
February 8, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Sales Tax By State**

Hi again!

I forgot to mention we need to apply different tax rates to different locations. The list attached has the location of each transaction. Our Sun Valley store is taxed at 8%, Stowe at 6%, and Mammoth at 7.75%.

Can you adjust your tax calculator code to apply the correct tax to each location?

Thanks!

 taxable\_transactions2.ipynb

Reply

Forward

## Solution Code

```
location = ['Sun Valley', 'Stowe', 'Mammoth',
            'Stowe', 'Sun Valley', 'Mammoth',
            'Mammoth', 'Mammoth', 'Sun Valley']
```

```
taxes = []
totals = []

for i, subtotal in enumerate(subtotals):
    if location[i] == 'Sun Valley':
        tax = round(subtotal * .08, 2)
    elif location[i] == 'Stowe':
        tax = round(subtotal * .06, 2)
    else:
        tax = round(subtotal * .0775, 2)
    total = round(subtotal + tax, 2)
    taxes.append(tax)
    totals.append(total)

print(taxes)
print(totals)
```

```
[1.28, 54.0, 62.0, 7.08, 0.48, 46.5, 1.94, 139.5, 8.0]
[17.26, 953.97, 861.97, 125.04, 6.47, 646.49, 26.93, 1939.44, 107.99]
```



# WHILE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

**While loops** run until a given logical expression becomes FALSE

- In other words, the loop runs *while* the expression is TRUE

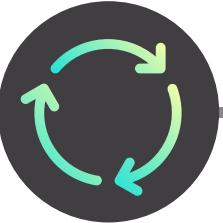
**while** logical expression:

Indicates a  
While Loop

A logical expression that  
evaluates to TRUE or FALSE

*Examples:*

- *counter < 10*
- *inventory > 0*
- *revenue > cost*



# WHILE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

**While loops** run until a given logical expression returns FALSE

- In other words, the loop runs *while* the expression is TRUE

**while** logical expression:  
do this

*Code to run while the logical  
expression is TRUE (must be  
indented!)*



# WHILE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

While loops often include **counters** that grow with each iteration

- Counters help us track how many times our loop has run
- They can also serve as a backup condition to exit a loop early (*more on this later!*)

```
counter = 0
while counter < 10:
    counter += 1
    print(counter)
```

This is an “addition assignment”, which adds a given number to the existing value of a variable:

`counter = counter + 1`

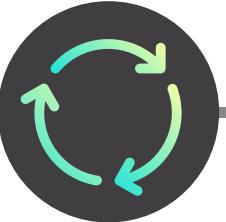
1 ← Counter increases to 1 in the first iteration

2 ← Counter increases to 2 in the second iteration

.

.

10 ← When the counter increments to 10, our condition becomes False, and exits the loop



# WHILE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Run a calculation until a goal is reached

```
# starting portfolio balance is 800000
stock_portfolio = 800000
year_counter = 0

while stock_portfolio < 1000000:
    # calculate annual investment income
    investment_income = stock_portfolio * .05  # 5% interest rate

    # add income to end of year portfolio balance
    stock_portfolio += investment_income

    # add one each year
    year_counter += 1

    print(f'At the end of year {year_counter}: '
          + f'My balance is ${round(stock_portfolio, 2)}')
```

At the end of year 1: My balance is \$840000.0

At the end of year 2: My balance is \$882000.0

At the end of year 3: My balance is \$926100.0

At the end of year 4: My balance is \$972405.0

At the end of year 5: My balance is \$1021025.25

The while loop here will run while stock\_portfolio is less than 1m

stock\_portfolio **starts at 800k** and **increases by 5%** of its value in each run:

1. 800k < 1m is TRUE
2. 840k < 1m is TRUE
3. 882k < 1m is TRUE
4. 926k < 1m is TRUE
5. 972k < 1m is TRUE
6. 1.02m < 1m is FALSE (**exit**)



# WHILE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Calculating bank balance until we're out of money.

```
bank_balance = 5000
month_counter = 0

while bank_balance > 0:
    spending = 1000
    bank_balance -= spending
    month_counter += 1
    print(f'At the end of month {month_counter}: '
          + f'My balance is ${round(bank_balance, 2)}')
```

At the end of month 1: My balance is \$4000  
At the end of month 2: My balance is \$3000  
At the end of month 3: My balance is \$2000  
At the end of month 4: My balance is \$1000  
At the end of month 5: My balance is \$0



**PRO TIP:** Use “-=” to subtract a number from a variable instead (subtraction assignment)



# INFINITE LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A while loop that *always* meets its logical condition is known as an **infinite loop**

- These can be caused by incorrect logic or uncertainty in the task being solved

```
stock_portfolio = 800000
year_counter = 0

while stock_portfolio < 1000000:
    investment_income = stock_portfolio * 0 # 0% interest rate
    stock_portfolio += investment_income
    year_counter += 1
    print(f'At the end of year {year_counter}: '
          + f'My balance is ${round(stock_portfolio, 2)}')
```

```
At the end of year 51461: My balance is $800000
At the end of year 51462: My balance is $800000
At the end of year 51463: My balance is $800000
```

-----  
KeyboardInterrupt

This indicates a manually stopped execution

The while loop here will run while stock\_portfolio is less than 1m, which **will always be the case**, as it's not growing due to 0% interest



If your program is stuck in an infinite loop, you will need to **manually stop it** and modify your logic



# ASSIGNMENT: WHILE LOOPS

 **NEW MESSAGE**  
February 9, 2022

**From:** Lucy Lift (CEO)  
**Subject:** Inventory Projections

Good morning,

Our investors are considering loaning us money to purchase more skis.

Can you advise how many months of stock we have left at the current monthly sales volume?

I would like to view the stock level at the end of each month as well.

Thanks!

 [inventory\\_logic.ipynb](#)

 [Reply](#)    [Forward](#)

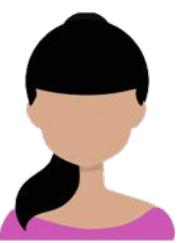
## Results Preview

```
current_inventory = 686
monthly_sales = 84
month = 0
```

```
for month in range(1, 10):
    current_inventory -= monthly_sales
    print(f'At the end of month {month}, we have {current_inventory} pairs of skis')
```

```
At the end of month 1, we have 602 pairs of skis
At the end of month 2, we have 518 pairs of skis
At the end of month 3, we have 434 pairs of skis
At the end of month 4, we have 350 pairs of skis
At the end of month 5, we have 266 pairs of skis
At the end of month 6, we have 182 pairs of skis
At the end of month 7, we have 98 pairs of skis
At the end of month 8, we have 14 pairs of skis
At the end of month 9, we have -70 pairs of skis
```

# ASSIGNMENT: WHILE LOOPS



## NEW MESSAGE

February 9, 2022

From: Lucy Lift (CEO)

Subject: Inventory Projections

Good morning,

Our investors are considering loaning us money to purchase more skis.

Can you advise how many months of stock we have left at the current monthly sales volume?

I would like to view the stock level at the end of each month as well.

Thanks!

inventory\_logic.ipynb

Reply

Forward

## Solution Code

```
current_inventory = 686
monthly_sales = 84
month = 0

while current_inventory > 0:
    current_inventory -= monthly_sales
    month += 1
    print(f'At the end of month {month}, we have {current_inventory} pairs of skis')
```

At the end of month 1, we have 602 pairs of skis  
At the end of month 2, we have 518 pairs of skis  
At the end of month 3, we have 434 pairs of skis  
At the end of month 4, we have 350 pairs of skis  
At the end of month 5, we have 266 pairs of skis  
At the end of month 6, we have 182 pairs of skis  
At the end of month 7, we have 98 pairs of skis  
At the end of month 8, we have 14 pairs of skis  
At the end of month 9, we have -70 pairs of skis



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

Loops can be **nested** within another loop

- The nested loop is referred to as an *inner* loop (the other is an *outer* loop)
- These are often used for navigating nested lists and similar data structures

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small. ← item\_list[0], size\_list[0]  
Snowboard is available in medium. ← item\_list[0], size\_list[1]  
Snowboard is available in large. ← item\_list[0], size\_list[2]  
Boots is available in small. ← item\_list[1], size\_list[0]  
Boots is available in medium. ← item\_list[1], size\_list[1]  
Boots is available in large. ← item\_list[1], size\_list[2]



How does this code work?

- The inner loop (size\_list) will run completely for each iteration of the outer loop (item\_list)
- In this case, the inner loop iterated three times for each of the two iterations of the outer loop, for a total of six print statements
- **NOTE:** There is no limit to how many layers of nested loops can occur



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.



item\_list (outer loop)

	0	Snowboard
	1	Boots

size\_list (inner loop)

	0	small
	1	medium
	2	large



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.  
Snowboard is available in medium.

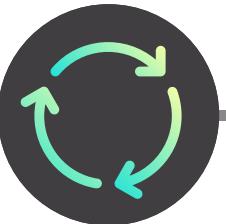


item\_list (outer loop)

0	Snowboard
1	Boots

size\_list (inner loop)

0	small
1	medium
2	large



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.  
Snowboard is available in medium.  
Snowboard is available in large.



item\_list (outer loop)

0	Snowboard
1	Boots

size\_list (inner loop)

0	small
1	medium
2	large



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.  
Snowboard is available in medium.  
Snowboard is available in large.  
Boots is available in small.



item\_list (outer loop)

0	Snowboard
1	Boots

size\_list (inner loop)

0	small
1	medium
2	large



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.  
Snowboard is available in medium.  
Snowboard is available in large.  
Boots is available in small.  
Boots is available in medium.

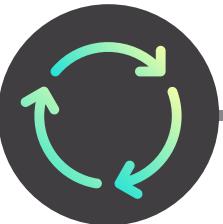


item\_list (outer loop)

0	Snowboard
1	Boots

size\_list (inner loop)

0	small
1	medium
2	large



# NESTED LOOPS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

## EXAMPLE

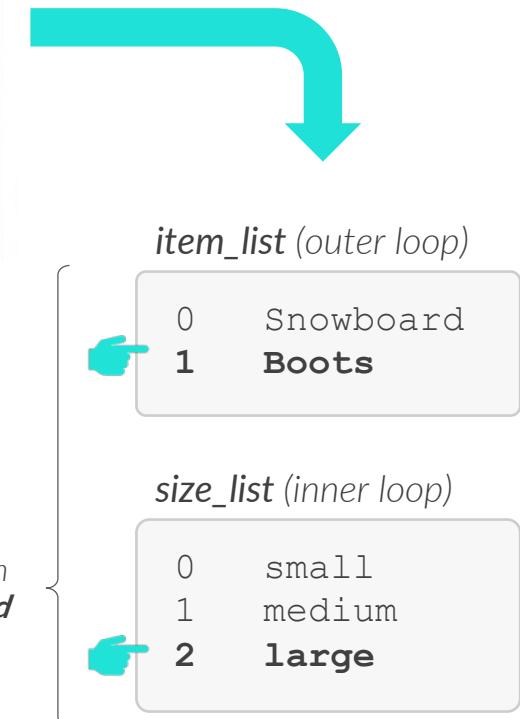
Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.  
Snowboard is available in medium.  
Snowboard is available in large.  
Boots is available in small.  
Boots is available in medium.  
Boots is available in large.

You exit a nested loop when  
*all its loops are completed*



# ASSIGNMENT: NESTED LOOPS

 **NEW MESSAGE**  
February 9, 2022

**From:** Alfie Alpine (Marketing Manager)  
**Subject:** Multi-Order Discount

Good morning,

I've attached a nested list containing the subtotals for customers that have made multiple orders.

We need to apply a 10% discount for each of their transactions and return the new totals in a list with the same structure as the original.

Thanks!

 multi\_order\_discounts.ipynb     Reply     Forward

## Results Preview

```
multi_order_customers
```

```
[[1799.94, 29.98, 99.99],  
 [15.98, 119.99],  
 [24.99, 24.99],  
 [649.99, 99.99],  
 [599.99, 399.97]]
```

```
discounted_prices = []
```

```
[[1619.95, 26.98, 89.99],  
 [14.38, 107.99],  
 [22.49, 22.49],  
 [584.99, 89.99],  
 [539.99, 359.97]]
```

# ASSIGNMENT: NESTED LOOPS

 **NEW MESSAGE**  
February 9, 2022

**From:** [Alfie Alpine \(Marketing Manager\)](#)  
**Subject:** Multi-Order Discount

Good morning,

I've attached a nested list containing the subtotals for customers that have made multiple orders.

We need to apply a 10% discount for each of their transactions and return the new totals in a list with the same structure as the original.

Thanks!

 [multi\\_order\\_discounts.ipynb](#)

 Reply    Forward

## Solution Code

```
multi_order_customers
```

```
[[1799.94, 29.98, 99.99],  
 [15.98, 119.99],  
 [24.99, 24.99],  
 [649.99, 99.99],  
 [599.99, 399.97]]
```

```
discounted_prices = []
```

```
for customer in multi_order_customers:  
    customer_discounts = []  
    for transaction in customer:  
        customer_discounts.append(round(transaction * 0.9, 2))  
    discounted_prices.append(customer_discounts)
```

```
discounted_prices
```

```
[[1619.95, 26.98, 89.99],  
 [14.38, 107.99],  
 [22.49, 22.49],  
 [584.99, 89.99],  
 [539.99, 359.97]]
```



# LOOP CONTROL

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

**Loop control** statements help refine loop behavior and handle potential errors

- These are used to change the flow of loop execution based on certain conditions

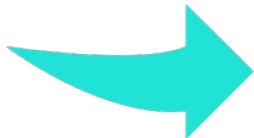
## Break



Stops the loop before completion

*Good for avoiding infinite loops & exiting loops early*

## Continue



Skips to the next iteration in the loop

*Good for excluding values that you don't want to process in a loop*

## Pass



Serves as a placeholder for future code

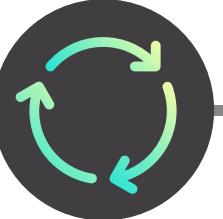
*Good for avoiding run errors with incomplete code logic*

## Try, Except



Help with error and exception handling

*Good for resolving errors in a loop without stopping its execution midway*



# BREAK

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

Triggering a **break** statement will exit the loop that it lives in

- This helps exit potential infinite loops when they can't be avoided by refining our logic
- It also helps set logical conditions to exit for loops early

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99,
             599.99, 24.99, 1799.94, 99.99]
revenue = 0

for subtotal in subtotals:
    revenue += subtotal
    print(round(revenue, 2))
    if revenue > 2000:
        break
```

```
15.98
915.95
1715.92
1833.88
1839.87
2439.86
```

The for loop here would normally run the length of the entire subtotals list (9 iterations), but the **break** statement triggers once the revenue is greater than 2,000 after the 6<sup>th</sup> transaction



# BREAK

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

Triggering a **break** statement will exit the loop that it lives in

- This helps exit potential infinite loops when they can't be avoided by refining our logic
- It also helps set logical conditions to exit for loops early

```
stock_portfolio = 100
year_counter = 0

while stock_portfolio < 1000000:
    investment_income = stock_portfolio * .05
    stock_portfolio += investment_income
    year_counter += 1
    print(f'My balance is ${round(stock_portfolio, 2)} in year {year_counter}')
    # break if i can't retire in 30 years
    if year_counter >= 30:
        print('Guess I need to save more.')
        break
```

```
My balance is $105.0 in year 1
My balance is $110.25 in year 2
My balance is $115.76 in year 3
My balance is $121.55 in year 4
```

```
:
.
.

My balance is $411.61 in year 29
My balance is $432.19 in year 30
Guess I need to save more.
```

The while loop here will run while stock\_portfolio is less than 1,000,000 (this would take 190 iterations/years)

A **break** statement is used inside an IF function here to exit the code in case the year\_counter is greater than 30



**PRO TIP:** Use a counter and a combination of IF and break to set a max number of iterations



# CONTINUE

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

Triggering a **continue** statement will move on to the next iteration of the loop

- No other lines in that iteration of the loop will run
- This is often combined with logical criteria to exclude values you don't want to process

```
item_list = ['ski-extreme', 'snowboard-basic', 'snowboard-extreme',
             'snowboard-comfort', 'ski-comfort', 'ski-backcountry']

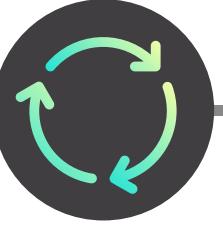
snowboards = []

for item in item_list:
    if 'ski' in item:
        continue
    snowboards.append(item)

print(snowboards)

['snowboard-basic', 'snowboard-extreme', 'snowboard-comfort']
```

A **continue** statement is used inside an IF statement here to avoid appending "ski" items to the snowboards list



# PASS

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

A **pass** statement serves as a placeholder for future code

- Nothing happens and the loop continues to the next line of code

```
item_list = ['ski-extreme', 'snowboard-basic', 'snowboard-extreme',
             'snowboard-comfort', 'ski-comfort', 'ski-backcountry']

snowboards = []

for item in item_list:
    if 'ski' in item:
        pass # need to write complicated logic later!
    snowboards.append(item)

snowboards

['ski-extreme',
 'snowboard-basic',
 'snowboard-extreme',
 'snowboard-comfort',
 'ski-comfort',
 'ski-backcountry']
```

The **pass** statement is used in place of the eventual logic that will live there, avoiding an error in the meantime



# TRY, EXCEPT

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# loop to calculate how many of each item I can buy
for price in price_list:
    affordable_quantity = 50//price # My budget is 50 dollars
    print(f"I can buy {affordable_quantity} of these.")
```

`TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'`

This for loop was stopped by a  
**TypeError** in the second iteration



# TRY, EXCEPT

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# loop to calculate how many of each item I can buy
for price in price_list:
    try:
        affordable_quantity = 50//price # My budget is 50 dollars
        print(f"I can buy {affordable_quantity} of these.")
    except:
        print("The price seems to be missing.")
```

```
I can buy 8 of these.
The price seems to be missing.
I can buy 2 of these.
I can buy 2 of these.
The price seems to be missing. ←
The price seems to be missing. ←
I can buy 0 of these.
```

Placing the code in a **try** statement handles the errors via the **except** statement without stopping the loop

Are 0 and '74.99' missing prices, or do we need to treat these exceptions differently?



# TRY, EXCEPT

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

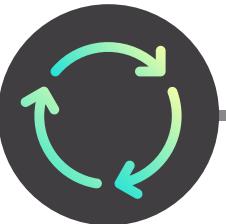
```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

for price in price_list:
    try:
        affordable_quantity = 50//price
```

The 0 price in the price\_list  
returns a **ZeroDivisionError**

```
50//0
```

**ZeroDivisionError:** integer division or modulo by zero



# TRY, EXCEPT

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# loop to calculate how many of each item I can buy
for price in price_list:
    try:
        affordable_quantity = 50//price # My budget is 50 dollars
        print(f"I can buy {affordable_quantity} of these.")
    except ZeroDivisionError:
        print("This product is free, I can take as many as I like.")
    except:
        print("That's not a number")
```

```
I can buy 8.0 of these.
That's not a number
I can buy 2.0 of these.
I can buy 2.0 of these.
This product is free, I can take as many as I like.
That's not a number
I can buy 0.0 of these.
```

If anything in the **try** block returns a `ZeroDivisionError`, the first **except** statement will run

The second **except** statement will run on any other error types



**PRO TIP:** Add multiple `except` statements for different error types to handle each scenario differently

# ASSIGNMENT: LOOP CONTROL

 NEW MESSAGE  
February 9, 2022

From: **Jerry Slush** (IT Manager)  
Subject: **Affordability Calculator**

Good morning,

We've made good progress on the item affordability calculator, which really helps our customers buy with confidence. We just need these tweaks to the logic:

- Skip processing for any 'None' elements
- Make sure prices stored as strings get processed (we'll fix the database later)
- Change 50 to a variable 'budget' so we can change it based on customer input
- Clean up general 'except' statement

 Improved\_affordability\_calculator.ipynb     Reply     Forward

## Results Preview

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]
```

```
# Create Budget Variable
```

```
I can buy 8.0 of these.  
I can buy 2.0 of these.  
I can buy 2.0 of these.  
This product is free, I can take as many as I like.  
string data detected  
I can buy 0.0 of these.  
I can buy 0.0 of these.
```

# ASSIGNMENT: LOOP CONTROL

 **NEW MESSAGE**  
February 9, 2022

From: **Jerry Slush (IT Manager)**  
Subject: **Affordability Calculator**

Good morning,

We've made good progress on the item affordability calculator, which really helps our customers buy with confidence. We just need these tweaks to the logic:

- Skip processing for any 'None' elements
- Make sure prices stored as strings get processed (we'll fix the database later)
- Change 50 to a variable 'budget' so we can change it based on customer input
- Clean up general 'except' statement

 Improved\_affordability\_calculator.ipynb     Reply     Forward

## Solution Code

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# Create Budget Variable
budget = 50

for price in price_list:
    if price is None: # is keyword to test for none
        continue
    try:
        affordable_quantity = budget // price
        print(f"I can buy {affordable_quantity} of these.")
    # If we get a ZeroDivisionError, print a special message
    except ZeroDivisionError:
        print("This product is free, I can take as many as I like.")
    # If TypeError, try converting to float
    except TypeError:
        affordable_quantity = budget // float(price)
        print('string data detected') ## let user know improper data detected
        print(f"I can buy {affordable_quantity} of these.")
```

```
I can buy 8.0 of these.
I can buy 2.0 of these.
I can buy 2.0 of these.
This product is free, I can take as many as I like.
string data detected
I can buy 0.0 of these.
I can buy 0.0 of these.
```

# KEY TAKEAWAYS

---



For loops run a **predetermined** number of times

- Analysts use for loops most often, as we usually work with datasets that have a known length



While loops run until a given **logical condition** is no longer met

- The number of iterations is often unknown beforehand, which means they carry a risk of entering an infinite loop – always double check your logic or create a backup stopping condition!



Loop control statements help **refine logic** and **handle potential errors**

- These are key in “fool proofing” loops to avoid infinite loops, set placeholders for future logic, skip iterations, and resolve errors

# DICTIONARIES & SETS

# DICTIONARIES & SETS



In this section we'll cover **dictionaries** and **sets**, two iterable data types with helpful use cases that allow for quick information retrieval and store unique values

## TOPICS WE'LL COVER:

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

## GOALS FOR THIS SECTION:

- Learn to store, append, and look up data in dictionaries and some helpful dictionary methods
- Build dictionaries using the `zip()` function
- Use set operations to find relationships in the values across multiple datasets



# LIST LIMITATIONS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

Lists can be inefficient when you need to look up specific values, as they can only be accessed via indexing

**EXAMPLE**    *Looking up the inventory status for goggles*

```
inventory_status = [['skis', 'in stock'], ['snowboard', 'sold out'],
                     ['goggles', 'sold out'], ['boots', 'in stock']]

item_looking_for = 'goggles'

for item in inventory_status:
    if item[0] == item_looking_for:
        print(item[1])

sold out
```

The nested list is used to pair unique items with their inventory status ('in stock' or 'sold out')

To find the status for a specific item, you need to iterate through the list to find the item of interest, then grab its corresponding inventory status



# DICTIONARY BASICS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

**Dictionaries** store key-value pairs, where keys are used to look up values

- **Keys** must be unique & immutable (*simple data types like strings are immutable*)
- **Values** do not need to be unique and can be any data type

**EXAMPLE**    Looking up the inventory status for goggles

```
inventory_status = {'skis': 'in stock',
                     'snowboard': 'sold out',
                     'goggles': 'sold out',
                     'boots': 'in stock'}
```

inventory\_status['goggles'] ←

'sold out'

Dictionaries are created with curly braces {}  
Keys and values are separated by colons :  
Key-value pairs are separated by commas ,

To retrieve dictionary values, simply enter the associated key

- NOTE: You cannot look up dictionary values or indices

The **KeyError** will be returned if a given key is not in the dictionary

```
inventory_status['in stock']  
KeyError: 'in stock'  
  
inventory_status[2]  
KeyError: 2
```



# DICTIONARY BASICS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

Dictionary values can be lists, so to access individual attributes:

1. Retrieve the **list** by looking up its **key**
2. Retrieve the **list element** by using its **index**

**EXAMPLE**    Looking up the stock quantity for skis

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

```
item_details['skis']
```

```
[249.99, 10, 'in stock']
```

```
item_details['skis'][1]
```

```
10
```

The key is the item name, and the value is a list storing item price, inventory, and inventory status

This returns the second element (index of 1) in the list stored as the value for the key 'skis'



# DICTIONARY BASICS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

You can conduct **membership tests** on dictionary keys

```
item_details = {'skis': [249.99, 10, 'in stock'],
                 'snowboard': [219.99, 0, 'sold out'],
                 'goggles': [99.99, 0, 'sold out'],
                 'boots': [79.99, 7, 'in stock']}

print('skis' in item_details)
print('bindings' in item_details)
```

True  
False

And you can **loop through\*** them

```
for item in item_details:
    print(item, item_details[item])

skis [249.99, 10, 'in stock']
snowboard [219.99, 0, 'sold out']
goggles [99.99, 0, 'sold out']
boots [79.99, 7, 'in stock']
```

Note that the items that are being looped over are the dictionary keys



# MODIFYING DICTIONARIES

List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

Referencing a new key and assigning it a value will **add a new key-value pair**, while referencing an existing key will **overwrite the existing pair**

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

Notice that 'bindings' is not in the dictionary keys

```
item_details['bindings'] = [149.99, 4, 'in stock']
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [149.99, 4, 'in stock']}
```

Therefore, assigning a value to a key of 'bindings' adds the key-value pair to the dictionary

```
item_details['bindings'] = [139.99, 0, 'out of stock']
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [139.99, 0, 'out of stock']}
```

Now that 'bindings' is a key in the dictionary, assigning a value to a key of 'bindings' overwrites the value in the dictionary for that key



# MODIFYING DICTIONARIES

List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

Use the **del** keyword to delete key-value pairs

`item_details`

```
{'skis': [249.99, 10, 'in stock'],
'snowboard': [219.99, 0, 'sold out'],
'goggles': [99.99, 0, 'sold out'],
'boots': [79.99, 7, 'in stock'],
'bindings': [149.99, 4, 'in stock']}
```

`del item_details['boots']`

`item_details`

```
{'skis': [249.99, 10, 'in stock'],
'snowboard': [219.99, 0, 'sold out'],
'goggles': [99.99, 0, 'sold out'],
'bindings': [139.99, 0, 'out of stock']}
```

This has deleted the key-value pair with a key of 'boots' from the dictionary

# ASSIGNMENT: DICTIONARY BASICS

 NEW MESSAGE  
February 10, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: Birthday Snacks

Good morning,

I'm glad we brought you on – I'm passing on the responsibility of managing birthday snacks to you.

Here are some of the changes needed:

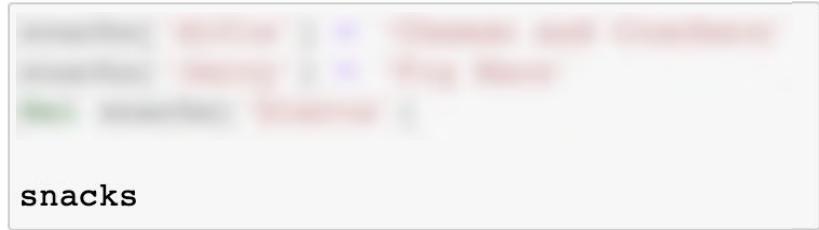
- Can you let me know what Stuart's snack is?
- Add me – I like 'Cheese and Crackers'
- Nobody else wants raisins, can you change Jerry's snack to 'Fig Bars'?
- Remove Sierra – she doesn't work here

 [snack\\_dictionary.ipynb](#)  

**Results Preview**

```
snacks = {  
    'Sally': 'Popcorn',  
    'Ricard': 'Chocolate Ice Cream',  
    'Stuart': 'Apple Pie',  
    'Jerry': 'Raisins',  
    'Sierra': 'Peanut Butter Cookies'  
}
```

  
`'Apple Pie'`

  
`snacks`

```
{'Sally': 'Popcorn',  
 'Ricard': 'Chocolate Ice Cream',  
 'Stuart': 'Apple Pie',  
 'Jerry': 'Fig Bars',  
 'Alfie': 'Cheese and Crackers'}
```

# ASSIGNMENT: DICTIONARY BASICS



NEW MESSAGE

February 10, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: Birthday Snacks

Good morning,

I'm glad we brought you on – I'm passing on the responsibility of managing birthday snacks to you.

Here are some of the changes needed:

- Can you let me know what Stuart's snack is?
- Add me – I like 'Cheese and Crackers'
- Nobody else wants raisins, can you change Jerry's snack to 'Fig Bars'?
- Remove Sierra – she doesn't work here

[snack\\_dictionary.ipynb](#)

Reply

Forward

## Solution Code

```
snacks = {  
    'Sally': 'Popcorn',  
    'Ricard': 'Chocolate Ice Cream',  
    'Stuart': 'Apple Pie',  
    'Jerry': 'Raisins',  
    'Sierra': 'Peanut Butter Cookies'  
}
```

```
snacks['Stuart']  
'Apple Pie'
```

```
snacks['Alfie'] = 'Cheese and Crackers'  
snacks['Jerry'] = 'Fig Bars'  
del snacks['Sierra']
```

```
snacks
```

```
{'Sally': 'Popcorn',  
 'Ricard': 'Chocolate Ice Cream',  
 'Stuart': 'Apple Pie',  
 'Jerry': 'Fig Bars',  
 'Alfie': 'Cheese and Crackers'}
```

# ASSIGNMENT: DICTIONARY CREATION

 **NEW MESSAGE**  
February 10, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Inventory Status**

Hello!

We need a quick way for our sales staff to look up the inventory of our items (I've attached two lists).

Can you create a dictionary with each item as a key, with 'in stock' if the corresponding inventory value is greater than 0, and 'sold out' if the inventory value equals zero?

These lists are a small sample, so make sure you aren't manually coding the dictionary.

 *inventory\_status.ipynb*

## Results Preview

```
items = ['skis', 'snowboard', 'goggles', 'boots']  
inventory = [10, 0, 0, 7]
```

```
inventory_status
```

```
{'skis': 'in stock',  
'snowboard': 'sold out',  
'goggles': 'sold out',  
'boots': 'in stock'}
```

# ASSIGNMENT: DICTIONARY CREATION

 **NEW MESSAGE**  
February 10, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Inventory Status**

Hello!

We need a quick way for our sales staff to look up the inventory of our items (I've attached two lists).

Can you create a dictionary with each item as a key, with 'in stock' if the corresponding inventory value is greater than 0, and 'sold out' if the inventory value equals zero?

These lists are a small sample, so make sure you aren't manually coding the dictionary.

 *inventory\_status.ipynb*

## Results Preview

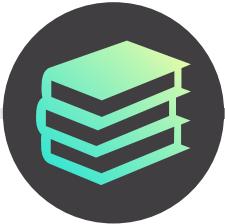
```
items = ['skis', 'snowboard', 'goggles', 'boots']
inventory = [10, 0, 0, 7]

inventory_status = {}

for i, item in enumerate(items):
    if inventory[i] == 0:
        inventory_status[item] = 'sold out'
    else:
        inventory_status[item] = 'in stock'

inventory_status

{'skis': 'in stock',
 'snowboard': 'sold out',
 'goggles': 'sold out',
 'boots': 'in stock'}
```



# DICTIONARY METHODS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

**keys**

Returns the keys from a dictionary

**.keys()**

**values**

Returns the values from a dictionary

**.values()**

**items**

Returns key value pairs from a dictionary as a list of tuples

**.items()**

**get**

Returns a value for a given key, or an optional value if the key isn't found

**.get(key, value if key not found)**

**update**

Appends specified key-value pairs, including entire dictionaries

**.update (key:value pairs)**



# KEYS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The `.keys()` method returns the keys from a dictionary

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

```
item_details.keys()
```

```
dict_keys(['skis', 'snowboard', 'goggles', 'boots'])
```

```
for item in item_details.keys():
    print(item)
```

```
skis
snowboard
goggles
boots
```

```
key_list = list(item_details.keys())
```

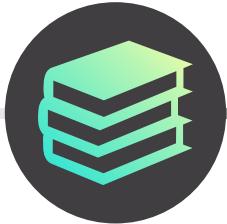
```
print(key_list)
```

```
['skis', 'snowboard', 'goggles', 'boots']
```

`.keys()` returns a **view object** that represents the keys as a list  
(this is more memory efficient than creating a list)

This view object can be iterated through, which has the same behavior as looping through the dictionary keys directly

This view object can be converted into a list or a tuple if needed



# VALUES

List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

The `.values()` method returns the values from a dictionary

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.values()  
  
dict_values([[249.99, 10], [219.99, 0], [99.99, 0], [79.99, 7]])
```

`.values()` returns a **view object** that represents the values as a list (this is more memory efficient than creating a list)

```
price_list = []  
for attribute in item_details.values():  
    price_list.append(attribute[0])  
  
price_list  
  
[249.99, 219.99, 99.99, 79.99]
```

This view object can be looped through as well. Here we're grabbing the first element from each of the lists returned by `.values()` and appending them to a new list



# ITEMS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The `.items()` method returns key-value pairs from a dictionary as a list of tuples

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.items()  
  
dict_items([('skis', [249.99, 10]), ('snowboard', [219.99, 0]),  
           ('goggles', [99.99, 0]), ('boots', [79.99, 7]))]
```

`.items()` returns a **view object** that represents the key-value pairs as a list of tuples

```
for key, value in item_details.items():  
    print(f'The {key} costs {value[0]}.)'
```

```
The skis costs 249.99.  
The snowboard costs 219.99.  
The goggles costs 99.99.  
The boots costs 79.99.
```

You can **unpack** the tuple to retrieve individual keys and values

In this case, the variable 'key' is assigned to the key in the tuple, and 'value' is assigned to the dictionary value



# ITEMS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The `.items()` method returns key-value pairs from a dictionary as a list of tuples

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.items()  
  
dict_items([('skis', [249.99, 10]), ('snowboard', [219.99, 0]),  
           ('goggles', [99.99, 0]), ('boots', [79.99, 7]))]
```

`.items()` returns a **view object** that represents the key-value pairs as a list of tuples

```
for item, item_attributes in item_details.items():  
    print(f'The {item} costs {item_attributes[0]}.' )
```

```
The skis costs 249.99.  
The snowboard costs 219.99.  
The goggles costs 99.99.  
The boots costs 79.99.
```

You can give these variables intuitive names, although k, v is common to represent keys and values



# GET

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The `.get()` method returns the values associated with a dictionary key

- It won't return a `KeyError` if the key isn't found
- You can specify an optional value to return if the key is not found

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

```
item_details.get('boots')
```

```
[79.99, 7, 'in stock']
```

`.get()` returns the value associated with the 'boots' key

```
item_details['bindings']
```

```
KeyError: 'bindings'
```

```
item_details.get('bindings')
```

```
item_details.get('bindings', "Sorry we don't carry that item.")
```

```
"Sorry we don't carry that item."
```

The difference between using `.get()` and simply entering the key directly is that `.get()` will not return an error if the key is not found

And you can specify an optional value to return if the key is not found

- `.get(key, value if key not found)`



# UPDATE

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The **.update()** method appends key-value pairs to a dictionary

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}

item_details.update({'bindings': [139.99, 0, 'out of stock']})
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [139.99, 0, 'out of stock']}
```

`.update()` appends new key-value pairs to a dictionary, in this case a single pair for a key of 'bindings'

- `.update(key:value pairs)`

```
new_items = {'scarf': [19.99, 100, 'in stock'], 'snowpants': 'N/A'}

item_details.update(new_items)
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'scarf': [19.99, 100, 'in stock'],
 'snowpants': 'N/A'}
```

This is the preferred way to **combine dictionaries**

As a reminder, dictionary values do not need to be the same type; note that the value for 'snowpants' is 'N/A', while the values for the rest of the keys are lists

# ASSIGNMENT: DICTIONARY METHODS



## NEW MESSAGE

February 10, 2022

From: **Stuart Slope** (Business Analyst)

Subject: European Planning

We're exploring the launch of our first store in Torino, Italy and I need help building out some of the data.

First, I need a dictionary containing the item numbers as keys, the number of sizes for each item as values.

Then, I need you to add items 10010 and 10011 with size counts of 4 and 7, respectively.

Finally, pull the prices out of the product dictionary and return a list with converted Euro pricing.

european\_planning.ipynb

Reply

Forward

## Results Preview

```
size_counts = {}
```

```
print(size_counts)
```

```
{10001: 1, 10002: 2, ... 10008: 3, 10009: 7}
```

```
print(size_counts)
```

```
{10001: 1, 10002: 2, ... 10009: 7, 10010: 4, 10011: 7}
```

```
euro_prices = []
```

```
euro_prices
```

```
[5.27, 8.79, 17.59, 21.99, 87.99, 70.39, 105.59, 87.99, 175.99]
```

# ASSIGNMENT: DICTIONARY METHODS



## NEW MESSAGE

February 10, 2022

From: **Stuart Slope** (Business Analyst)  
Subject: European Planning

We're exploring the launch of our first store in Torino, Italy and I need help building out some of the data.

First, I need a dictionary containing the item numbers as keys, the number of sizes for each item as values.

Then, I need you to add items 10010 and 10011 with size counts of 4 and 7, respectively.

Finally, pull the prices out of the product dictionary and return a list with converted Euro pricing.

european\_planning.ipynb

Reply

Forward

## Solution Code

```
size_counts = {}
for key, value in item_dict.items():
    size_counts[key] = len(value[3])

print(size_counts)

{10001: 1, 10002: 2, ... 10008: 3, 10009: 7}
```

```
new_size_counts = {10010: 4, 10011: 7}
size_counts.update(new_size_counts)

print(size_counts)
```

```
euro_prices = []
exchange_rate = .88
for value in item_dict.values():
    euro_prices.append(round(value[1] * exchange_rate, 2))

euro_prices
```

[5.27, 8.79, 17.59, 21.99, 87.99, 70.39, 105.59, 87.99, 175.99]



# ZIP

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The **zip()** function combines two iterables, like lists, into a single iterator of tuples

- `first_iterable[0]` gets paired with `second_iterable[0]`, and so on

```
item_list = ['skis', 'snowboard', 'goggles', 'boots']
price_list = [249.99, 219.99, 99.99, 79.99]
inventory = [10, 0, 0, 7]

zip(price_list, inventory)

<zip at 0x7fb0f012f580>
```

Here, we're zipping together two lists and returning a **zip object** that contains the instructions for pairing the  $i^{\text{th}}$  object from each iterable

```
item_attributes = list(zip(price_list, inventory))

item_attributes

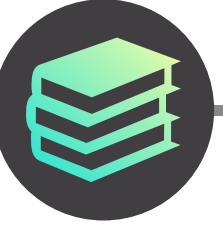
[(249.99, 10), (219, 0), (99, 0), (99.99, 7)]
```

When you create a list from the zip object, you get a list with the  $i^{\text{th}}$  element from each iterable paired together in a tuple

```
list(zip(item_list, price_list, inventory))

[('skis', 249.99, 10),
 ('snowboard', 219, 0),
 ('goggles', 99, 0),
 ('boots', 99.99, 7)]
```

Any number of iterables can be combined this way



# ZIP

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

The **zip()** function is commonly used to build dictionaries

```
item_details = [[249.99, 10, 'in stock'],
                [219.99, 0, 'sold out'],
                [99.99, 0, 'sold out'],
                [79.99, 7, 'in stock']]

item_names = ['skis', 'snowboard', 'goggles', 'boots']

item_dict = dict(zip(item_names, item_details))

item_dict
```

```
dict(zip(item_list, price_list, inventory))
```

**ValueError: dictionary update sequence element #0 has length 3; 2 is required**

```
item_dict = dict(zip(item_list, zip(price_list, inventory)))

item_dict
```

When creating a dictionary from the zip object, the elements of the first iterable become the **keys**, and the second become the **values**

Note that you can only create a dictionary from a zip object with two iterables

But you can zip iterables together within the second argument

# ASSIGNMENT: ZIP

 **NEW MESSAGE**  
February 10, 2022

**From:** Jerry Slush (IT Manager)  
**Subject:** European Dictionary

Hi there!

We're developing the architecture for our planned European store.

Can you create a dictionary by combining item\_ids as keys with the lists item\_names, euro\_prices, item\_category, and sizes as the values?

We'll upload this to our database shortly.

Exciting times!

 european\_dictionary.ipynb

Reply Forward

## Results Preview

euro\_items

```
{10001: ('Coffee', 5.27, 'beverage', ['250mL']),  
10002: ('Beanie', 8.79, 'clothing', ['Child', 'Adult']),  
10003: ('Gloves', 17.59, 'clothing', ['Child', 'Adult']),  
10004: ('Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),  
10005: ('Helmet', 87.99, 'safety', ['Child', 'Adult']),  
10006: ('Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),  
10007: ('Coat', 105.59, 'clothing', ['S', 'M', 'L']),  
10008: ('Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']),  
10009: ('Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11])}
```

# ASSIGNMENT: ZIP

 **NEW MESSAGE**

February 10, 2022

**From:** Jerry Slush (IT Manager)

**Subject:** European Dictionary

Hi there!

We're developing the architecture for our planned European store.

Can you create a dictionary by combining item\_ids as keys with the lists item\_names, euro\_prices, item\_category, and sizes as the values?

We'll upload this to our database shortly.

Exciting times!

 european\_dictionary.ipynb

Reply Forward

## Solution Code

```
euro_items = dict(zip(item_ids, zip(item_names, euro_prices, item_category, sizes)))
euro_items

{10001: ('Coffee', 5.27, 'beverage', ['250mL']),
 10002: ('Beanie', 8.79, 'clothing', ['Child', 'Adult']),
 10003: ('Gloves', 17.59, 'clothing', ['Child', 'Adult']),
 10004: ('Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),
 10005: ('Helmet', 87.99, 'safety', ['Child', 'Adult']),
 10006: ('Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),
 10007: ('Coat', 105.59, 'clothing', ['S', 'M', 'L']),
 10008: ('Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']),
 10009: ('Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11])}
```



# NESTED DICTIONARIES

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

You can **nest dictionaries** as values of another dictionary

- The nested dictionary is referred to as an *inner* dictionary (the other is an *outer* dictionary)

```
item_history = {  
    2019: {"skis": [249.99, 10, "in stock"], "snowboard": [219.99, 0, "sold out"]},  
    2020: {"skis": [259.99, 10, "in stock"], "snowboard": [229.99, 0, "sold out"]},  
    2021: {"skis": [269.99, 10, "in stock"], "snowboard": [239.99, 0, "sold out"]},  
}  
  
item_history  
  
{2019: {'skis': [249.99, 10, 'in stock'],  
        'snowboard': [219.99, 0, 'sold out']},  
  2020: {'skis': [259.99, 10, 'in stock'],  
        'snowboard': [229.99, 0, 'sold out']},  
  2021: {'skis': [269.99, 10, 'in stock'],  
        'snowboard': [239.99, 0, 'sold out']}}}
```

The outer dictionary here has years as keys, and inner dictionaries as values

The inner dictionaries have items as keys, and lists with item attributes as values

```
item_history[2020]  
  
{'skis': [259.99, 10, 'in stock'], 'snowboard': [229.99, 0, 'sold out']}
```

To access an inner dictionary, reference the outer dictionary key

```
item_history[2020]['skis']  
  
[259.99, 10, 'in stock']
```

To access the values of an inner dictionary, reference the outer dictionary key, then the inner dictionary key of interest

# ASSIGNMENT: NESTED DICTIONARIES

 NEW MESSAGE  
February 10, 2022

From: **Jerry Slush** (IT Manager)  
Subject: European Dictionary Updates

Hi again!

We decided to restructure the dictionary you created earlier into a nested dictionary with each attribute represented by a key for fast lookup. Can you:

1. Verify the price on item 10009
2. Update the sizes for item 10009 to European sizing (they are stored in a list)
3. Create a dictionary based on the new one that includes item name as keys and sizes as values

 modified\_european\_dictionary.ipynb     Reply     Forward

## Results Preview

175.99

euro\_data[10009]

```
{'name': 'Ski Boots',
 'price': 175.99,
 'category': 'hardware',
 'sizes': [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5]}
```

product\_sizes

```
{'Coffee': ['250mL'],
 'Beanie': ['Child', 'Adult'],
 'Gloves': ['Child', 'Adult'],
 'Sweatshirt': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Helmet': ['Child', 'Adult'],
 'Snow Pants': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Coat': ['S', 'M', 'L'],
 'Ski Poles': ['S', 'M', 'L'],
 'Ski Boots': [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5]}
```

# ASSIGNMENT: NESTED DICTIONARIES

 NEW MESSAGE  
February 10, 2022

From: **Jerry Slush** (IT Manager)  
Subject: European Dictionary Updates

Hi again!

We decided to restructure the dictionary you created earlier into a nested dictionary with each attribute represented by a key for fast lookup. Can you:

1. Verify the price on item 10009
2. Update the sizes for item 10009 to European sizing (they are stored in a list)
3. Create a dictionary based on the new one that includes item name as keys and sizes as values

 modified\_european\_dictionary.ipynb     Reply     Forward

## Solution Code

```
euro_data[10009]['price']
```

```
175.99
```

```
boot_sizes = [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5]
```

```
euro_data[10009]['sizes'] = boot_sizes
```

```
product_sizes = {}
for product_details in euro_data.values():
    product_sizes[product_details['name']] = product_details['sizes']

product_sizes
```

```
{'Coffee': ['250mL'],
 'Beanie': ['Child', 'Adult'],
 'Gloves': ['Child', 'Adult'],
 'Sweatshirt': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Helmet': ['Child', 'Adult'],
 'Snow Pants': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Coat': ['S', 'M', 'L'],
 'Ski Poles': ['S', 'M', 'L'],
 'Ski Boots': [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5]}
```



# SETS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

A **set** is a collection of unique values

- Sets are **unordered**, which means their values cannot be accessed via index or key
- Sets are mutable (values can be added/removed), but set *values* must be **unique** & **immutable**

```
my_set = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
my_set  
{'skis', 'sled', 'snowboard'}
```

Sets can be created with curly braces `{}`

The absence of colons differentiates them from dictionaries

```
my_set = set(['snowboard', 'snowboard', 'skis', 'snowboard', 'sled'])  
my_set  
{'skis', 'sled', 'snowboard'}
```

Sets can also be created via conversion using `set()`

Note that duplicate values are automatically removed when created



# WORKING WITH SETS

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

You can conduct **membership tests** on sets

```
my_set
```

```
{'skis', 'sled', 'snowboard'}
```

```
'snowboard' in my_set
```

```
True
```

You can **loop through** them

```
for value in my_set:  
    print(value)
```

```
snowboard  
sled  
skis
```

But you **can't index** them (*they are unordered*)

```
my_set[0]
```

```
TypeError: 'set' object is not subscriptable
```

# ASSIGNMENT: SETS

 NEW MESSAGE  
February 10, 2022

From: **Stuart Slope** (Business Analyst)  
Subject: Product Categories

Hey,  
I'm doing an analysis on product categories.  
Can you collect the unique product category values from our European dictionary??  
How many unique categories are there?  
Once you have that, can you check if 'outdoor' is in there yet?  
This will be really helpful, thanks!

 product\_categories.ipynb     Reply     Forward

**Results Preview**

```
print(unique_categories)
['clothing', 'hardware', 'beverage', 'safety']
```

4

False

# ASSIGNMENT: SETS

 NEW MESSAGE  
February 10, 2022

From: **Stuart Slope** (Business Analyst)  
Subject: Product Categories

Hey,  
I'm doing an analysis on product categories.  
Can you collect the unique product category values from our European dictionary??  
How many unique categories are there?  
Once you have that, can you check if 'outdoor' is in there yet?  
This will be really helpful, thanks!

 product\_categories.ipynb     Reply     Forward

## Solution Code

```
categories = []  
  
for value in euro_items.values():  
    categories.append(value[2])  
  
unique_categories = set(categories)  
  
print(unique_categories)  
  
{'clothing', 'hardware', 'beverage', 'safety'}
```

```
len(unique_categories)
```

4

```
'outdoor' in unique_categories
```

False



# SET OPERATIONS

List Limitations

Dictionary Basics

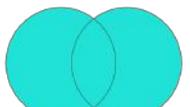
Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

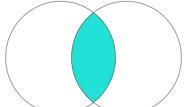
Python has useful **operations** that can be performed between sets



**union**

Returns all unique values in both sets

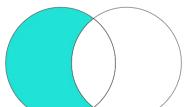
`set1.union(set2)`



**intersection**

Returns values present in both sets

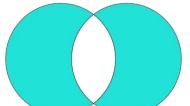
`set1.intersection(set2)`



**difference**

Returns values present in set 1, but not set 2

`set1.difference(set2)`



**symmetric difference**

Returns values not shared between sets  
(opposite of intersection)

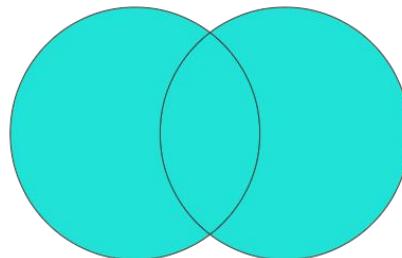
`set1.symmetric_difference(set2)`



**PRO TIP:** Chain set operations to capture the relationship between three or more sets, for example – `set1.union(set2).union(set3)`



# UNION



List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

**Union** returns all unique values in both sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.union(saturday_items)  
  
{'goggles', 'helmet', 'skis', 'sled', 'snowboard'}
```

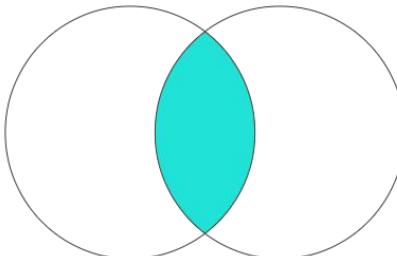
All values from both sets are returned, without duplicates

```
sunday_items = {'coffee'}  
  
friday_items.union(saturday_items).union(sunday_items)  
  
{'coffee', 'goggles', 'helmet', 'skis', 'sled', 'snowboard'}
```

All values from the three sets are returned by chaining two union operations



# INTERSECTION



List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.intersection(saturday_items)  
  
{'skis', 'snowboard'}
```

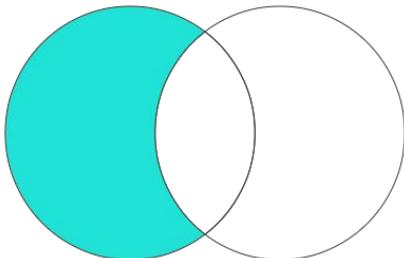
Only the values in both sets are returned, without duplicates

```
sunday_items = {'coffee'}  
  
friday_items.intersection(saturday_items).intersection(sunday_items)  
  
set()
```

Since no value is present in all three sets, an empty set is returned



# DIFFERENCE



List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.difference(saturday_items)  
  
{'sled'}
```

'sled' is the only value in friday\_items  
that is NOT in saturday\_items

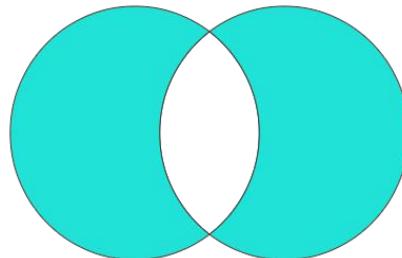
```
saturday_items - friday_items  
  
{'goggles', 'helmet'}
```

If you reverse the order, the output changes – 'goggles' and 'helmet' are  
in saturday\_items but NOT in friday\_items

Note that the subtraction sign can be used instead of difference



# SYMMETRICAL DIFFERENCE



List Limitations

Dictionary Basics

Modifying  
Dictionaries

Dictionary  
Methods

Nested  
Dictionaries

Sets

**Symmetrical difference** returns all values not shared between sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.symmetric_difference(saturday_items)  
  
{'goggles', 'helmet', 'sled'}
```

'sled' is only in set 1, and 'goggles' and 'helmet' are only in set 2



# SET USE CASES

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

1. Sets are more efficient than lists for performing **membership tests**

```
time_list = list(range(1000000))
time_set = set(range(1000000))
```

Using  
*Lists*

```
%%time
100000 in time_list
```

CPU times: user 5.07 ms, sys: 454 µs, total: 5.53 ms

Using  
*Sets*

```
%%time
100000 in time_set
```

CPU times: user 5 µs, sys: 1e+03 ns, total: 6 µs



Sets are implemented as **hash tables**, which makes looking up values extremely fast; the downside is that they cannot preserve order (lists rely on dynamic arrays that preserve order but have slower performance)



# SET USE CASES

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

2. Sets can **gather unique values** efficiently without looping

*Using Lists*

```
%%time
unique_items = []

for item in shipments_today:
    if item not in unique_items:
        unique_items.append(item)
```

`unique_items`

```
CPU times: user 27 µs, sys: 1 µs, total: 28 µs
['ski', 'snowboard', 'helmet', 'hat', 'goggles']
```

*Using Sets*

```
%%time
list(set(shipments_today))
```

```
CPU times: user 9 µs, sys: 0 ns, total: 9 µs
['snowboard', 'ski', 'helmet', 'hat', 'goggles']
```



# SET USE CASES

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

- Set operations can find the **data shared, or not shared, between items** without looping

Using  
*Lists*

```
shipment_today = ['ski', 'snowboard', 'ski', 'ski', 'helmet', 'hat', 'goggles']
shipment_yesterday = ['hat', 'goggles', 'snowboard', 'hat', 'bindings']

unique_today = []
for item_t in shipment_today:
    if item_t not in shipment_yesterday:
        if item_t not in unique_today:
            unique_today.append(item_t)

unique_today

['ski', 'helmet']
```

Using  
*Sets*

```
set(shipment_today).difference(set(shipment_yesterday))

{'helmet', 'ski'}
```

# ASSIGNMENT: SET OPERATIONS

 NEW MESSAGE  
February 10, 2022

From: **Stuart Slope** (Business Analyst)  
Subject: **Weekend Sale Analysis**

Hey,

There are three lists with the customers who made a purchase on Friday, Saturday, and Sunday.

Can you get me the set of unique customers who made purchases on Saturday or Sunday?

Then return the customers that made purchases on Friday AND during the weekend; we want to target them with additional promotions.

## Results Preview

`weekend_set`

```
{'C00002',
'C00004',
'C00006',
'C00008',
'C00010',
'C00016',
'C00017',
'C00018',
'C00019',
'C00021',
'C00022'}
```

```
{'C00004', 'C00006', 'C00008', 'C00010', 'C00016'}
```

# ASSIGNMENT: SET OPERATIONS

 NEW MESSAGE  
February 10, 2022

From: **Stuart Slope** (Business Analyst)  
Subject: **Weekend Sale Analysis**

Hey,

There are three lists with the customers who made a purchase on Friday, Saturday, and Sunday.

Can you get me the set of unique customers who made purchases on Saturday or Sunday?

Then return the customers that made purchases on Friday AND during the weekend; we want to target them with additional promotions.

## Solution Code

```
weekend_set = set(saturday_customers).union(set(sunday_customers))  
weekend_set
```

```
{'C00002',  
'C00004',  
'C00006',  
'C00008',  
'C00010',  
'C00016',  
'C00017',  
'C00018',  
'C00019',  
'C00021',  
'C00022'}
```

```
weekend_set.intersection(set(friday_customers))  
{'C00004', 'C00006', 'C00008', 'C00010', 'C00016'}
```

# KEY TAKEAWAYS

---



## Dictionaries

- Looking up values in dictionaries by their keys is more efficient than scanning through lists
- Dictionaries can mimic a structure like tabular data (e.g., Excel sheets).



## Dictionary methods

- The `.keys()`, `.values()`, and `.items()` methods are critical, but `.get()` and `.update()` are also very helpful



## Use the `zip()` function

- The `zip` function stitches together multiple iterables into a single iterable of tuples



## Use sets

- Converting lists to sets can help perform these operations more efficiently

# FUNCTIONS

# FUNCTIONS



In this section we'll learn to write custom **functions** to boost efficiency, import external functions stored in modules or packages, and write comprehensions

## TOPICS WE'LL COVER:

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

## GOALS FOR THIS SECTION:

- Identify the key components of Python functions
- Define custom functions and manage variable scope
- Practice importing packages, which are external collections of functions
- Learn how to write lambda functions and apply functions with map()
- Learn how to write comprehensions, powerful expressions used to create iterables



# FUNCTIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Functions** are reusable blocks of code that perform specific tasks when called

- They take in data, perform a set of actions, and return a result

**Input**

Inputs to a function are known as **arguments**

**Data**, is some form, is usually one of the arguments passed into a function

Other arguments might be options for processing the data or formatting the output

**Function**

Functions are blocks of code that perform a specific task

When you use a function, it is known as a **function call**, or **calling a function**

The function itself is stored somewhere else, and you are 'calling' it by using its name

**Output**

Outputs are **values** returned by a function

**Side Effects**

Side effects are changes or actions made by the function, other than the output



Functions help **boost efficiency** as programmers immensely!

# THE ANATOMY OF MAX

Function Components

Defining Functions

Variable Scope

Modules

Packages

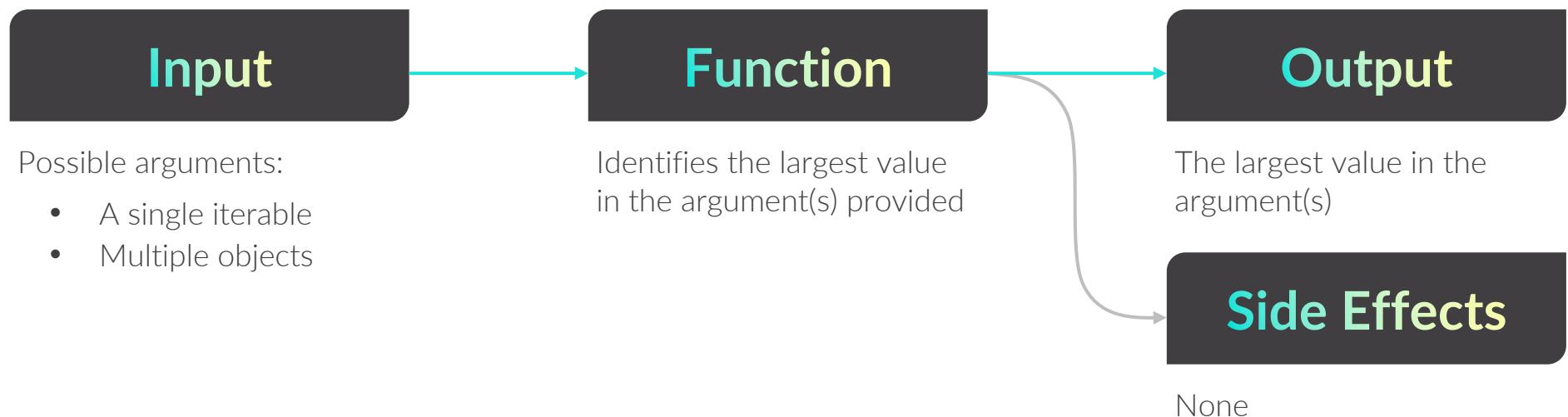
Lambda Functions

Comprehensions

The built-in **max()** function works the same way (as do others like `len()`, `sum()`, etc.)

```
In [5]: max?  
  
Docstring:  
max(iterable, *, default=obj, key=func) -> value  
max(arg1, arg2, *args, *, key=func) -> value  
  
With a single iterable argument, return its biggest item. The  
default keyword-only argument specifies an object to return if  
the provided iterable is empty.  
With two or more arguments, return the largest argument.
```

The documentation provides details on the input, function, and output for `max()`



# THE ANATOMY OF MAX

Function Components

Defining Functions

Variable Scope

Modules

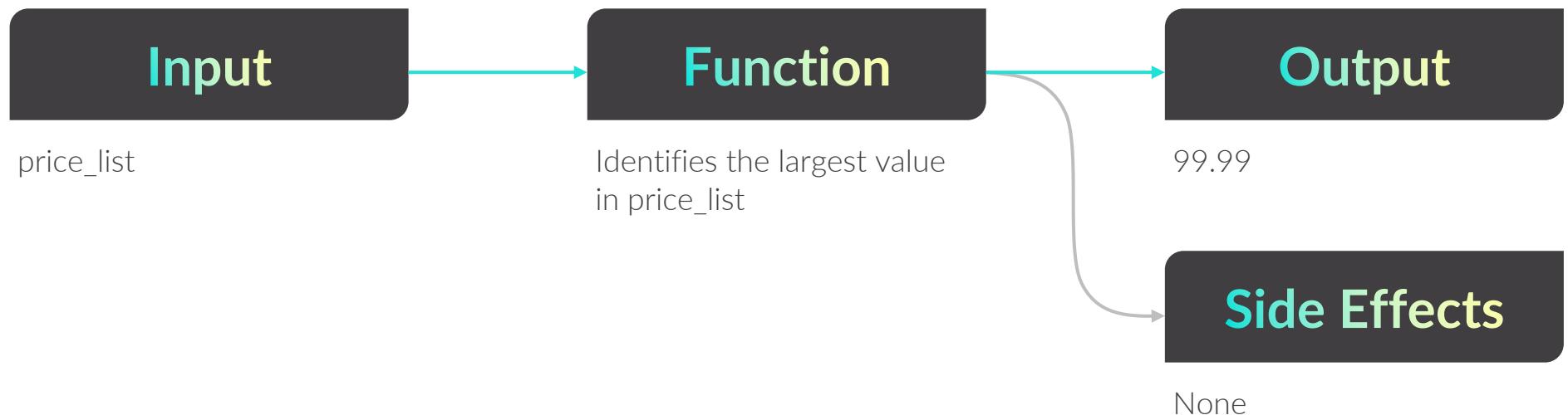
Packages

Lambda Functions

Comprehensions

The built-in **max()** function works the same way (as do others like `len()`, `sum()`, etc.)

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]  
max(price_list)
```



# DEFINING A FUNCTION

Function Components

Defining Functions

Variable Scope

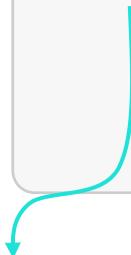
Modules

Packages

Lambda Functions

Comprehensions

```
def function_name(arguments):
```



Indicates you're defining a new function



An intuitive name to call your function by



Variable names for the function inputs, separated by commas

**Examples:**

- avg
- usd\_converter



Functions have the same **naming rules** and best practices as variables – use 'snake\_case'!



# DEFINING A FUNCTION

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

```
def function_name(arguments):  
    do this
```



*Code block for the function  
that uses the arguments to  
perform a specific task*



# DEFINING A FUNCTION

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

```
def function_name(arguments):  
    do this  
    return output
```

*Ends the function*  
(without it, the function  
returns None)

*Values to return*  
(usually variables)



# DEFINING A FUNCTION

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

## EXAMPLE

Defining a function that concatenates two words, separated by a space

```
def concatenator(string1, string2):
    combined_string = string1 + ' ' + string2
    return combined_string
```

Here we're defining a function called **concatenator**, which accepts two arguments, combines them with a space , and returns the result

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

When we call this function with two string arguments, the combined string is returned

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

Note that we don't need a code block before return, here we're combining strings in the return statement



# WHEN TO DEFINE A FUNCTION?

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

You should take time to **define a function** when:

- ✓ You find yourself copying & pasting a block of code repeatedly
- ✓ You know you will use a block of code again in the future
- ✓ You want to share a useful piece of logic with others
- ✓ You want to make a complex program more readable



**PRO TIP:** There are no hard and fast rules, but in data analysis you'll often have a similar workflow for different projects – by taking the time to package pieces of your data cleaning or analysis workflow into functions, you are saving your future self (and your colleagues') time

# THE DOCSTRING

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

You can create a **docstring** for your function

- This is used to embed text describing its arguments and the actions it performs

Use **triple quotes** inside the function to create its docstring

```
def concatenator(string1, string2):
    """combines two strings, separated with a space

    Args:
        string1 (str): string to put before space
        string2 (str): string to put after space

    Returns:
        str: string1 and string2 separated by a space
    """
    return string1 + " " + string2
```

Use `'?` to retrieve the docstring, just like with built-in functions

concatenator?

```
Signature: concatenator(string1, string2)
Docstring:
combines two strings, separated with a space

Args:
    string1 (str): string to put before space
    string2 (str): string to put after space

Returns:
    str: string1 and string2 separated by a space
File:      /var/folders/f8/075hbnj13wb0f9yzh9k4nyz00000
gn/T/ipykernel_21060/3833548733.py
Type:     function
```



**PRO TIP:** Take time to create a docstring for your function, especially if you plan to share it with others. What's the point in creating reusable code if you need to read all of it to understand how to use it?

# ASSIGNMENT: DEFINING A FUNCTION

 **NEW MESSAGE**  
February 11, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Sales Tax Calculator**

Good morning,

We want to expand the work you've done on sales tax to all our stores, which requires a function to account for changes to taxes or expansion into new states.

Write a function that takes a given subtotal and tax rate, and returns the total amount owed (subtotal plus tax).

Thanks in advance!

 [tax\\_calculator\\_function.ipynb](#)

 [Reply](#)    [Forward](#)

## Results Preview

```
tax_calculator(100, .09)
```

109.0

# ASSIGNMENT: DEFINING A FUNCTION

 **NEW MESSAGE**  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Sales Tax Calculator**

Good morning,

We want to expand the work you've done on sales tax to all our stores, which requires a function to account for changes to taxes or expansion into new states.

Write a function that takes a given subtotal and tax rate, and returns the total amount owed (subtotal plus tax).

Thanks in advance!

 [tax\\_calculator\\_function.ipynb](#)

 [Reply](#)    [Forward](#)

## *Solution Code*

```
def tax_calculator(subtotal, sales_tax):
    """takes in a subtotal and tax rate and returns total owed

    Args:
        subtotal(float): cost of items in transaction
        tax_rate (float): tax rate at store location

    Returns:
        float: total amount owed on transaction
    """
    total = subtotal + (subtotal * sales_tax)

    return total
```



# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

There are several **types of arguments** that can be passed on to a function:

- **Positional** arguments are passed in the order they were defined in the function
- **Keyword** arguments are passed in any order by using the argument's name
- **Default** arguments pass a preset value if nothing is passed in the function call
- **\*args** arguments pass any number of positional arguments as tuples
- **\*\*kwargs** arguments pass any number of keyword arguments as dictionaries



# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Positional** arguments are passed in the order they were defined in the function

```
def concatenator(string1, string2):  
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

```
concatenator('World!', 'Hello')
```

```
'World! Hello'
```

The first value passed in the function will be string1, and the second will be string2

Therefore, changing the order of the inputs changes the output



# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Keyword** arguments are passed in any order by using the argument's name

```
def concatenator(string1, string2):
    return string1 + ' ' + string2

concatenator('Hello', 'World!')
'Hello World!'

concatenator(string2='World!', string1='Hello')
'Hello World!'
```

By specifying the value to pass for each argument, the order no longer matters

```
concatenator(string2='World!', 'Hello')
SyntaxError: positional argument follows keyword argument
```

Keyword arguments **cannot** be followed by positional arguments

```
concatenator('Hello', string2='World!')
```

Positional arguments **can** be followed by keyword arguments  
(the first argument is typically reserved for primary input data)



# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Default** arguments pass a preset value if nothing is passed in the function call

```
def concatenator(string1, string2='World!'):
    return string1 + ' ' + string2
```

```
concatenator('Hola')
```

```
'Hola World!'
```

```
concatenator('Hola', 'Mundo!')
```

```
'Hola Mundo!'
```

Assign a default value by using '=' when defining the function

Since a single argument was passed, the second argument defaults to 'World!'

By specifying a second argument, the default value is no longer used

```
def concatenator(string1='Hello', string2):
    return string1 + ' ' + string2
```

```
SyntaxError: non-default argument follows default argument
```

Default arguments must come after arguments without default values



# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**\*args** arguments pass any number of positional arguments as tuples

```
def concatenator(*args):
    new_string = ''
    for arg in args:
        new_string += (arg + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
'Hello world! How are you?'
```

Using '\*' before the argument name allows users to enter any number of strings for the function to concatenate

Since the arguments are passed as a tuple, we can loop through them or unpack them

```
def concatenator(*words):
    new_string = ''
    for word in words:
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!')
```

```
'Hello world!'
```

It's not necessary to use 'args' as long as the asterisk is there

Here we're using 'words' as the argument name, and only passing through two words

# ARGUMENT TYPES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**\*\*kwargs** arguments pass any number of keyword arguments as dictionaries

```
def concatenator(**words):
    new_string = ''
    for word in words.values():
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator(a='Hello', b ='there!',  
             c="What's", d='up?')
```

"Hello there! What's up?"

Using '\*\*' before the argument name allows users to enter any number of keyword arguments for the function to concatenate

Note that since the arguments are passed as dictionaries, you need to use the .values() method to loop through them



**PRO TIP:** Use \*\*kwargs arguments to unpack dictionaries and pass them as keyword arguments

```
def exponentiator(constant, base, exponent):
    return constant * (base**exponent)
```

```
param_dict = {'constant': 2, 'base': 3, 'exponent': 2}
```

```
exponentiator(**param_dict)
```

The exponentiator function has three arguments: constant, base, and exponent

Note that the dictionary keys in 'param\_dict' match the argument names for the function

By using '\*\*' to pass the dictionary to the function, the dictionary is unpacked, and the value for each key is mapped to the corresponding argument



# RETURN VALUES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

## Functions can return multiple values

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
('Hello world! How are you?', 'you?')
```

The values to return must be separated by commas

This returns a tuple of the specified values

```
sentence, last_word = concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
print(last_word)
```

```
Hello world! How are you?  
you?
```

The variable 'sentence' is assigned to the first element returned in the tuple, so if the order was switched, it would store 'you?'

You can unpack the tuple into variables during the function call

# RETURN VALUES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Functions can return multiple values as **other types of iterables** as well

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return [sentence.rstrip(), last_word]
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
['Hello world! How are you?', 'you?']
```

Wrap the comma-separated return values in square brackets to return them inside a list

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return {sentence.rstrip(): last_word}
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
{'Hello world! How are you?': 'you?'}
```

Or use dictionary notation to create a dictionary  
(this could be useful as input for another function!)



# VARIABLE SCOPE

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

The **variable scope** is the region of the code where the variable was assigned

1. **Local scope** – variables created inside of a function

- *These cannot be referenced outside of the function*

2. **Global scope** – variables created outside of functions

- *These can be referenced both inside and outside of functions*

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

Hello world! How are you?

Since the variable 'sentence' is assigned inside of the concatenator function, it has local scope

Trying to print this variable outside of the function will then return a `NameError`



# CHANGING VARIABLE SCOPE

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

You can **change variable scope** by using the *global* keyword

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

Hello world! How are you?

By declaring the variable 'sentence' as global, it is now recognized outside of the function it was defined in

# CHANGING VARIABLE SCOPE

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

You can **change variable scope** by using the *global* keyword

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    global sentence
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

SyntaxError: name 'sentence' is assigned to before global declaration

Note that the variable must be declared as *global* **before it is assigned a value**, or you will receive a SyntaxError



**PRO TIP:** While it might be tempting, declaring global variables within a function is considered bad practice in most cases – imagine if you borrowed this code and it overwrote an important variable! Instead, use 'return' to deliver the values you want and assign them to local variables

# CREATING MODULES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

To save your functions, **create a module** in Jupyter by using the `%%writefile` magic command and the `.py` extension

```
%%writefile saved_functions.py
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

Writing `saved_functions.py`

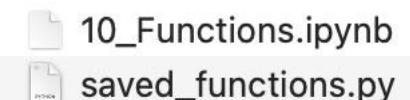
```
%%writefile saved_functions.py
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

def multiplier(num1, num2):
    return num1 * num2
```

Overwriting `saved_functions.py`

This creates a Python module that you can import functions from

- Follow `%%writefile` with the name of the file and the `.py` extension
- By default, the `.py` file is stored in the same folder as the notebook
- You can share functions easily by sending this file to a friend or colleague!



Multiple functions can be saved to the same module

# IMPORTING MODULES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

To **import saved functions**, you can either import the entire module or import specific functions from the module

```
import saved_functions
saved_functions.concatenate('Hello', 'world!')
('Hello world!', 'world!')
```

```
saved_functions.multiplier(5, 10)
```

50

**import module**

reads in external Python modules

If you import the entire module, you need to reference it when calling its functions, in the form of **module.function()**

```
from saved_functions import concatenate, multiplier
concatenate('Hello', 'world!')
```

```
('Hello world!', 'world!')
```

```
multiplier(5, 10)
```

50

**from module import function**

imports specific functions from modules

By importing specific functions, you don't need to reference the entire module name when calling a function



This method can lead to **naming conflicts** if another object has the same name

# ASSIGNMENT: CREATING A MODULE

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Modified Sales Tax Calculator**

Hey again,  
Can you make the following changes to our sales tax calculator?

1. Since we've created this calculator for Stowe employees, set the default sales tax to 6%
2. We want the function to return subtotal, tax and sales tax in a list
3. Save this function to tax\_calculator.py

Thanks!

 modified\_tax\_calculator.ipynb     Reply     Forward

**Results Preview**

```
tax_calculator(100)
[100, 6.0, 106.0]
```

# ASSIGNMENT: CREATING A MODULE

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Modified Sales Tax Calculator**

Hey again,  
Can you make the following changes to our sales tax calculator?

1. Since we've created this calculator for Stowe employees, set the default sales tax to 6%
2. We want the function to return subtotal, tax and sales tax in a list
3. Save this function to tax\_calculator.py

Thanks!

 modified\_tax\_calculator.ipynb     Reply     Forward

## *Solution Code*

```
%%writefile tax_calculator.py

def tax_calculator(subtotal, sales_tax=.06):
    """takes in a subtotal and tax rate and returns total owed

Args:
    subtotal(float): cost of items in transaction
    tax_rate (float, optional): tax rate at store location. Defaults to .06.

Returns:
    list: list containing subtotal, total, and tax
"""
    tax = round(subtotal * sales_tax, 2)
    total = round(subtotal + tax, 2)

    return [subtotal, tax, total]
```

# ASSIGNMENT: IMPORTING A MODULE

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **Applied Sales Tax Calculator**

Hey again,

Now that you've updated and saved the function, can you import it, pass a list of transactions through, and create a list containing the transaction information for each?

Once you've done that, I'd like you to create a dictionary with the supplied customer IDs as keys and the transaction information as values.

Thanks!

 applied\_tax\_calculator.ipynb     Reply     Forward

## Results Preview

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]
```

```
full_transactions
```

```
[[15.98, 0.96, 16.94],  
 [899.97, 54.0, 953.97],  
 [799.97, 48.0, 847.97],  
 [117.96, 7.08, 125.04],  
 [5.99, 0.36, 6.35],  
 [599.99, 36.0, 635.99]]
```

```
customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']
```

```
customer_dict
```

```
{'C00004': [15.98, 0.96, 16.94],  
 'C00007': [899.97, 54.0, 953.97],  
 'C00015': [799.97, 48.0, 847.97],  
 'C00016': [117.96, 7.08, 125.04],  
 'C00020': [5.99, 0.36, 6.35],  
 'C00010': [599.99, 36.0, 635.99]}
```

# ASSIGNMENT: IMPORTING A MODULE

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Applied Sales Tax Calculator

Hey again,

Now that you've updated and saved the function, can you import it, pass a list of transactions through, and create a list containing the transaction information for each?

Once you've done that, I'd like you to create a dictionary with the supplied customer IDs as keys and the transaction information as values.

Thanks!

 applied\_tax\_calculator.ipynb     Reply     Forward

## Solution Code

```
from tax_calculator import tax_calculator

subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]

full_transactions = []

for subtotal in subtotals:
    full_transactions.append(tax_calculator(subtotal))

full_transactions

[[15.98, 0.96, 16.94],
 [899.97, 54.0, 953.97],
 [799.97, 48.0, 847.97],
 [117.96, 7.08, 125.04],
 [5.99, 0.36, 6.35],
 [599.99, 36.0, 635.99]]

customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']

customer_dict = dict(zip(customer_ids, full_transactions))

customer_dict

{'C00004': [15.98, 0.96, 16.94],
 'C00007': [899.97, 54.0, 953.97],
 'C00015': [799.97, 48.0, 847.97],
 'C00016': [117.96, 7.08, 125.04],
 'C00020': [5.99, 0.36, 6.35],
 'C00010': [599.99, 36.0, 635.99]}
```



# IMPORTING EXTERNAL FUNCTIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

The same method used to import your own modules can be used to import external modules, packages, and libraries

- **Modules** are individual .py files
- **Packages** are collections of modules
- **Libraries** are collections of packages (*library and package are often used interchangeably*)

```
dir(saved_functions)
```

Use the dir() function to view the contents for any of the above

```
[ '__builtins__',  
  '__cached__',  
  '__doc__',  
  '__file__',  
  '__loader__',  
  '__name__',  
  '__package__',  
  '__spec__',  
  'concatenator',  
  'multiplier']
```

- This is showing the directory for the saved\_functions module
- Modules have many attributes associated with them that the functions will follow
- '\_\_file\_\_' is the name of the .py file

These are the two functions inside the module



# IMPORTING EXTERNAL FUNCTIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

## EXAMPLE

Importing the math package, which contains handy mathematical functions

```
import math  
  
dir(math)  
  
['__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 'acos',  
 'acosh',  
 'asin',  
 'asinh',  
 'atan',  
 'atan2',  
 'atanh',  
 'bitwise',  
 'ceil',  
 'comb',  
 'comb2旧',  
 'copysign',  
 'cosh',  
 'erf',  
 'erfc',  
 'exp',  
 'expm1',  
 'fabs',  
 'factorial',  
 'fdim',  
 'floor',  
 'fmod',  
 'fsum',  
 'gamma',  
 'gcd',  
 'hypot',  
 'isclose',  
 'iscopysign',  
 'isfinite',  
 'isinf',  
 'isnan',  
 'isneginf',  
 'isposinf',  
 'isreal',  
 'isub',  
 'j0',  
 'j1',  
 'jinc',  
 'jpi',  
 'jpiinc',  
 'lcm',  
 'log',  
 'log10',  
 'log1p',  
 'log2',  
 'modf',  
 'nextafter',  
 'perm',  
 'prod',  
 'radians',  
 'sin',  
 'sinh',  
 'sqrt',  
 'tan',  
 'tanh',  
 'trunc']
```

This is arc cosine

This package has a LOT of functions  
(too many for a single page!)

*What does math.sqrt return?*

```
math.sqrt(81)
```

9.0

It's helpful to look up the official documentation, which will usually give a helpful overview of the package and its contents

[docs.python.org/3/library/math.html](https://docs.python.org/3/library/math.html)



**PRO TIP:** Import packages with an alias using the `as` keyword – this saves keystrokes while still explicitly referencing the package to help avoid naming conflicts

```
import math as m
```

```
m.sqrt(81)
```

9.0



# PRO TIP: NAMING CONFLICTS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Importing external functions can lead to **naming conflicts**

```
def sqrt(number):
    return f'The square root of {number} is its square root.'
```

```
sqrt(10)
```

```
'The square root of 10 is its square root.'
```

```
from math import sqrt
```

```
sqrt(10)
```

```
3.1622776601683795
```

In this case we have a `sqrt` function we defined ourselves, and another that we imported

In scenarios like these, only the **most recently created** object with a given name will be recognized

```
def sqrt(number):
    return f'The square root of {number} is its square root.'
```

```
import math as m
```

```
print(m.sqrt(10))
```

```
print(sqrt(10))
```

```
3.1622776601683795
```

```
The square root of 10 is its square root.
```

To avoid conflicts, refer to the package name or an alias with imported functions



# INSTALLING PACKAGES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

While many come preinstalled, you may need to **install a package** yourself

'! sends the following code to your operating system's shell

(Yes, this means you can use the command line via Jupyter)

!conda install openpyxl

conda is a package manager for Python

install tells Python to install the following package

The name of the package to install



**DO NOT install packages like this**

It can lead to installing packages in the wrong instance of Python



```
import sys  
!conda install --yes --prefix {sys.prefix} openpyxl
```

Adding this extra code snippet ensures the package gets installed correctly

# PRO TIP: MANAGING PACKAGES

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Use Anaconda to **review all installed packages** and their versions

```
!conda list
```

notebook	0.4.8	pysyntexcascos_0
numpy	1.21.2	py39h4b4dc7a_0
numpy-base	1.21.2	py39he0bd621_0
openpyxl	3.0.9	pyhd3eb1b0_0

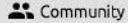
**conda list** returns a list of installed packages and versions



Environments



Learning



Community



base (root)



Name	Description	Version
alabaster	Configurable, python 2+3 compatible sphinx theme.	0.7.12
anaconda	Simplifies package management and deployment of anaconda	2021.11
anaconda-client	Anaconda cloud command line client library	1.9.0
anaconda-project	Tool for encapsulating, running, and reproducing data science projects	0.10.1

The **environments tab** in Anaconda Navigator also shows the installed packages and versions

```
import sys
```

```
!conda install --yes --prefix {sys.prefix} openpyxl
```

```
!conda install --yes --prefix {sys.prefix} package_name=1.2.3
```

**conda install** updates a package to its latest version; to install a specific version, use **package\_name=<version>**

You may need to install an older version of a package so it is compatible with another package you are working with



# ESSENTIAL PACKAGES FOR ANALYTICS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Python has incredible packages for **data analysis** beyond math and openpyxl  
Here are some to explore further:





# MAP

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

The **map()** function is an efficient way to apply a function to all items in an iterable

- `map(function, iterable)`

```
def currency_formatter(number):
    return '$' + str(number)
```

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
```

```
map(currency_formatter, price_list)
```

```
<map at 0x7fa430944d90>
```

```
list(map(currency_formatter, price_list))
```

```
['$5.99', '$19.99', '$24.99', '$0', '$74.99', '$99.99']
```

The `map` function returns a **map object** - which saves memory until we've told Python what to do with the object

You can convert the `map` object into a list or other iterable

# ASSIGNMENT: MAP

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Re: Applied Sales Tax Calculator

Hey again,  
We're really making some great progress on implementing Python – can you import the tax calculator and apply it to the newest list of transactions?  
Don't use a for loop!  
Thanks!

 mapping\_the\_calculator.ipynb     Reply     Forward

## Results Preview

```
subtotals = [1799.94, 99.99, 254.95, 29.98, 99.99]
```

```
[[1799.94, 108.0, 1907.94],  
[99.99, 6.0, 105.99],  
[254.95, 15.3, 270.25],  
[29.98, 1.8, 31.78],  
[99.99, 6.0, 105.99]]
```

# ASSIGNMENT: MAP

 NEW MESSAGE  
February 11, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: Re: Applied Sales Tax Calculator

Hey again,  
We're really making some great progress on implementing Python – can you import the tax calculator and apply it to the newest list of transactions?  
Don't use a for loop!  
Thanks!

 mapping\_the\_calculator.ipynb     Reply     Forward

## *Solution Code*

```
from tax_calculator import tax_calculator  
  
subtotals = [1799.94, 99.99, 254.95, 29.98, 99.99]  
  
list(map(tax_calculator, subtotals))  
  
[[1799.94, 108.0, 1907.94],  
 [99.99, 6.0, 105.99],  
 [254.95, 15.3, 270.25],  
 [29.98, 1.8, 31.78],  
 [99.99, 6.0, 105.99]]
```



# LAMBDA FUNCTIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Lambda functions** are single line, anonymous functions that are only used briefly

- **lambda** arguments: expression

```
(lambda x: x**2)(3)
```

```
9
```

Lambda functions can be called on a single value, but typically aren't used for this

```
(lambda x: x**2, x**3)(3)
```

```
NameError
```

They cannot have multiple outputs or expressions

```
(lambda x, y: x * y if y > 5 else x / y)(6, 5)
```

```
1.2
```

They can take multiple arguments and leverage conditional logic

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88

converted = map(lambda x: round(x * exchange_rate, 2), price_list)
list(converted)
```

```
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

They are usually leveraged in combination with a function like `map()` or in a comprehension.

# ASSIGNMENT: LAMBDA FUNCTIONS

 NEW MESSAGE  
February 11, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: One-time discount

Hey,

We need to apply a 10% discount for customers who purchased more than \$500 worth of product – they're starting to complain it hasn't been applied.

This is a one-time promotion, so can you code up something quick?

We won't reuse it.

Thanks!

 [discount\\_orders.ipynb](#)

 Reply    Forward

## Results Preview

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]
```

```
discounted_subtotals = list(
```

```
discounted_subtotals
```

```
[15.98, 809.97, 719.97, 117.96, 5.99, 539.99]
```

# ASSIGNMENT: LAMBDA FUNCTIONS



## NEW MESSAGE

February 11, 2022

From: **Alfie Alpine** (Marketing Manager)  
Subject: One-time discount

Hey,

We need to apply a 10% discount for customers who purchased more than \$500 worth of product – they're starting to complain it hasn't been applied.

This is a one-time promotion, so can you code up something quick?

We won't reuse it.

Thanks!

📎 [discount\\_orders.ipynb](#)

Reply

Forward

## Solution Code

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]

discounted_subtotals = list(
    map(
        lambda subtotal: round(subtotal * 0.9, 2) if subtotal > 500 else subtotal,
        subtotals
    )
)

discounted_subtotals
```

[15.98, 809.97, 719.97, 117.96, 5.99, 539.99]



# PRO TIP: COMPREHENSIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

**Comprehensions** can generate sequences from other sequences

Syntax: `new_list = [expression for member in other_iterable (if condition)]`

## EXAMPLE

*Creating a list of Euro prices from USD prices*

*Before*, you needed a for loop to create the new list

```
usd_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

euro_list
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

*Now*, you can use comprehensions to do this with a single line of code

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]

euro_list = [round(price * exchange_rate, 2) for price in usd_list]
euro_list
[5.27, 8.79, 17.59, 21.99, 87.99]
```

```
euro_list = [round(price * exchange_rate, 2) for price in usd_list if price < 10]
euro_list
[5.27, 0.0]
```

You can even leverage conditional logic to create very powerful expressions!

# ASSIGNMENT: COMPREHENSIONS

 NEW MESSAGE

February 11, 2022

**From:** Ricardo Nieva (Financial Planner)

**Subject:** Cost of European Items

Hi there,

I'm working on some numbers for our European expansion. Can you help me calculate the combined cost of all of our European items?

The data is stored in the euro\_data dictionary.

Thanks!

 [cost\\_of\\_european\\_items.ipynb](#)

 Reply     Forward

## Results Preview

euro\_data

```
{10001: ['Coffee', 5.27, 'beverage', ['250mL']],
 10002: ['Beanie', 8.79, 'clothing', ['Child', 'Adult']],
 10003: ['Gloves', 17.59, 'clothing', ['Child', 'Adult']],
 10004: ['Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10005: ['Helmet', 87.99, 'safety', ['Child', 'Adult']],
 10006: ['Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10007: ['Coat', 105.59, 'clothing', ['S', 'M', 'L']],
 10008: ['Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']],
 10009: ['Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11]]}
```

581.59

# ASSIGNMENT: COMPREHENSIONS

 NEW MESSAGE

February 11, 2022

From: Ricardo Nieva (Financial Planner)

Subject: Cost of European Items

Hi there,

I'm working on some numbers for our European expansion. Can you help me calculate the combined cost of all of our European items?

The data is stored in the euro\_data dictionary.

Thanks!

 cost\_of\_european\_items.ipynb

## Solution Code

```
euro_data
```

```
{10001: ['Coffee', 5.27, 'beverage', ['250mL']],
 10002: ['Beanie', 8.79, 'clothing', ['Child', 'Adult']],
 10003: ['Gloves', 17.59, 'clothing', ['Child', 'Adult']],
 10004: ['Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10005: ['Helmet', 87.99, 'safety', ['Child', 'Adult']],
 10006: ['Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10007: ['Coat', 105.59, 'clothing', ['S', 'M', 'L']],
 10008: ['Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']],
 10009: ['Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11]]}
```

```
sum([value[1] for value in euro_data.values()])
```

581.59



# PRO TIP: DICTIONARY COMPREHENSIONS

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Comprehensions can also **create dictionaries** from other iterables

Syntax: `new_dict = {key: value for key, value in other_iterable(if condition)}`

*These can be expressions (including function calls!)*

## EXAMPLE

*Creating a dictionary of inventory costs per item (stock quantity \* price)*

```
items = ['skis', 'snowboard', 'goggles', 'boots']
status = [[5, 249.99], [0, 219.99], [0, 99.99], [12, 79.99]]

inventory_costs = {k: round(v[0] * v[1], 2) for k, v in zip(items, status)}

inventory_costs
{'skis': 1249.95, 'snowboard': 0.0, 'goggles': 0.0, 'boots': 959.88}
```

*This is creating a dictionary by using `items` as keys, and the product of `status[0]` and `status[1]` as values*

*`zip()` is being used to stitch the two lists together into a single iterable*

```
inventory_costs = {
    k: round(v[0] * v[1], 2) for k, v in zip(items, status) if v[0] > 0
}

inventory_costs
{'skis': 1249.95, 'boots': 959.88}
```

*You can still use conditional logic!*

# ASSIGNMENT: DICTIONARY COMPREHENSIONS

 NEW MESSAGE  
February 11, 2022

From: **Jerry Slush** (IT Manager)  
Subject: Final Tax Calculator

Hey there,  
We're getting close to having a final ETL process for our European dictionary. Can you create a function that applies our tax calculator to a list of subtotals and matches the output up with customer\_IDs?  
Store them in a dictionary, with customer IDs as keys, and the transactions as values.  
Thanks – we're getting close to implementing this!

 [final\\_tax\\_calculator.ipynb](#)  

## Results Preview

```
from tax_calculator import tax_calculator

customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']

subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]

transaction_dict_creator(customer_ids, subtotals, .08)
```

```
{'C00004': [15.98, 1.28, 17.26],  
 'C00007': [899.97, 72.0, 971.97],  
 'C00015': [799.97, 64.0, 863.97],  
 'C00016': [117.96, 9.44, 127.4],  
 'C00020': [5.99, 0.48, 6.47],  
 'C00010': [599.99, 48.0, 647.99]}
```

# ASSIGNMENT: DICTIONARY COMPREHENSIONS

 NEW MESSAGE  
February 11, 2022

From: **Jerry Slush (IT Manager)**  
Subject: **Final Tax Calculator**

Hey there,

We're getting close to having a final ETL process for our European dictionary. Can you create a function that applies our tax calculator to a list of subtotals and matches the output up with customer\_IDs?

Store them in a dictionary, with customer IDs as keys, and the transactions as values.

Thanks – we're getting close to implementing this!

 [final\\_tax\\_calculator.ipynb](#)     Reply     Forward

## Solution Code

```
from tax_calculator import tax_calculator

customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']

subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]

def transaction_dict_creator(customer_ids, subtotals, tax_rate):
    # docstring omitted for screenshot
    customer_dict = {
        customer_id: tax_calculator(subtotal, tax_rate)
        for customer_id, subtotal in zip(customer_ids, subtotals)
    }
    return customer_dict

transaction_dict_creator(customer_ids, subtotals, .08)

{'C00004': [15.98, 1.28, 17.26],
 'C00007': [899.97, 72.0, 971.97],
 'C00015': [799.97, 64.0, 863.97],
 'C00016': [117.96, 9.44, 127.4],
 'C00020': [5.99, 0.48, 6.47],
 'C00010': [599.99, 48.0, 647.99]}
```



# PRO TIP: COMPREHENSIONS VS MAP

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

Note that exchange\_rate is a default argument equal to .88

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
[currency_converter(price) for price in price_list]
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

Here, in both the comprehension and map(), price\_list is being passed as a positional argument to the currency\_converter function, and the default exchange\_rate value is applied

```
list(map(currency_converter, price_list))
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

*But what if I want to change the exchange rate?*



# PRO TIP: COMPREHENSIONS VS MAP

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
[currency_converter(price, exchange_rate=.85) for price in price_list]
```

```
import functools
list(map(functools.partial(currency_converter, exchange_rate=.85), price_list))
[5.09, 16.99, 21.24, 0.0, 63.74, 84.99]
```

Note that exchange\_rate is a default argument equal to .88

In the comprehension, the exchange\_rate argument is easy to specify

To do so with map(), you need to create a partial function (outside the course scope)



In general, both methods provide an efficient way to apply a function to a list – Comprehensions are usually preferred, as they are more efficient and highly readable, but there will be instances when using map with lambda is preferred (*like manipulating a Pandas Column*)

You may prefer not to use comprehensions for particularly complex logic (*nested loops, multiple sequences, lots of conditions*), but for most use cases comprehensions are a best practice for creating new sequences based off others.

# KEY TAKEAWAYS

---



**Functions** are reusable blocks of code that make us more efficient analysts

- *If you find yourself writing the same code over and over again, consider making it a function*



External **packages & libraries** provide access to functions developed by others

- *Most data analysts work with features from packages like pandas and matplotlib regularly*



**Map()** applies a function to an iterable without the use of a for loop

- *This is often paired with lambda functions for handy one-off procedures*



**Comprehensions** can create sequences and dictionaries from other iterables

- *They are more efficient than map() at applying a function to a list, and allow you to specify different arguments*

# MANIPULATING EXCEL SHEETS

# MANIPULATING EXCEL SHEETS



In this section we'll import the openpyxl package and **manipulate data from Excel sheets** by using the Python skills learned throughout the course

## TOPICS WE'LL COVER:

The openpyxl Package

Navigating Workbooks

Looping Through Cells

Modifying Cells

## GOALS FOR THIS SECTION:

- Read and save Excel files in Python
- Navigate workbooks, worksheets, and cells
- Loop through cell ranges and modify cell values



# THE OPENPYXL PACKAGE

The openpyxl  
Package

Navigating  
Workbooks

Iterating Through  
Cells

Modifying Cells

The **openpyxl** package is designed for reading and writing Excel files

Without ever needing to open Excel itself, you can use openpyxl to:

- Create, modify, or delete workbooks, worksheets, rows, or columns
- Leverage custom Python functions or Excel formulas (yes, *really!*)
- Automate Excel chart creation
- ... or do almost anything else that can be done in Excel natively

## Example use cases:

- Cleaning or manipulating Excel sheets before uploading them into a database
- Automating database pulls (sqlalchemy library) and creating Excel sheets for end users
- Summarizing Excel data before sending it to another user



openpyxl **should be installed** in Anaconda already, but if it isn't you can install it like you would any other package

# THE MAVEN SKI SHOP DATA

## *Orders\_info*

	A	B	C	D	E	F	G	H
1	Order_ID	Customer_ID	Order_Date	Subtotal	Tax	Total	Location	Items_Ordered
2	1000000	C00004	11/26/2021	15.98			Sun Valley	10001, 10002
3	100001	C00007	11/26/2021	899.97			Stowe	10008, 10009, 10010
4	100002	C00015	11/26/2021	799.97			Mammoth	10011, 10012, 10013
5	100003	C00016	11/26/2021	117.96			Stowe	10002, 10003, 10004, 10006
6	100004	C00020	11/26/2021	5.99			Sun Valley	10001
7	100005	C00010	11/26/2021	599.99			Mammoth	10010
8	100006	C00006	11/26/2021	24.99			Mammoth	10004
9	100007	C00001	11/26/2021	1799.94			Mammoth	10008, 10008, 10009, 10009, 10009, 10010, 10010
10	100008	C00003	11/26/2021	99.99			Sun Valley	10005
11	100009	C00014	11/26/2021	254.95			Sun Valley	10002, 10003, 10004, 10006, 10007
12	100010	C00001	11/26/2021	29.98			Mammoth	10002, 10003
13	100011	C00001	11/26/2021	99.99			Mammoth	10005
14	100012	C00005	11/26/2021	25.98			Sun Valley	10001, 10003
15	100013	C00008	11/26/2021	649.98			Stowe	10012, 10013
16	100014	C00013	11/26/2021	89.99			Sun Valley	10014
17	100020	C00004	11/27/2021	119.99			Sun Valley	10007
18	100021	C00017	11/27/2021	599.99			Stowe	10010
19	100022	C00019	11/27/2021	649.98			Sun Valley	10012, 10013
20	100023	C00002	11/27/2021	24.99			Stowe	10004
21	100024	C00008	11/27/2021	99.99			Stowe	10005
22	100025	C00021	11/27/2021	99.99			Mammoth	10008
23	100026	C00022	11/27/2021	5.99			Sun Valley	10001
24	100027	C00006	11/28/2021	24.99			Mammoth	10002
25	100031	C00018	11/28/2021	999.96			Stowe	10005, 10008, 10009, 10010
26	100032	C00018	11/28/2021	99.99			Stowe	10006
27	100033	C00010	11/28/2021	399.97			Mammoth	10005, 10008, 10009
28	100034	C00016	11/28/2021	89.99			Stowe	10014



## *Item\_info*

Product_ID	Product_Name	Price	Cost	Available Sizes
10001	Coffee	5.99	0.99	250mL
10002	Beanie	9.99	4.29	Child, Adult
10003	Gloves	19.99	7.99	Child, Adult
10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL
10005	Helmet	99.99	49.99	Child, Adult
10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL
10007	Coat	119.99	54.55	S, M, L
10008	Ski Poles	99.99	69.99	S, M, L
10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
10010	Skis	599.99	249.99	S, M, L
10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
10012	Bindings	149.99	89.99	NA
10013	Snowboard	499.99	199.99	S, M, L, Powder



# NAVIGATING WORKBOOKS

The openpyxl  
Package

Navigating  
Workbooks

Iterating Through  
Cells

Modifying Cells

```
import openpyxl as xl

workbook = xl.load_workbook(filename='maven_ski_shop_data.xlsx')
```

Just supply a file name!\*

**workbook.sheetnames** returns the worksheet names in a workbook

```
workbook.sheetnames
```

```
['Item_Info', 'Inventory_Levels', 'Orders_Info']
```



Note that 'workbook' is simply a variable name that stores the workbook object

Using this intuitive name makes the code easier to read, but any name can be used instead (wb is common as well)



# NAVIGATING WORKSHEETS

The openpyxl  
Package

Navigating  
Workbooks

Iterating Through  
Cells

Modifying Cells

**workbook.active** returns the name of the worksheet openpyxl is pointed to

```
workbook.active
```

```
<Worksheet "Orders_Info">
```

```
workbook.active = 1  
workbook.active
```

```
<Worksheet "Inventory_Levels">
```

```
workbook.active.title
```

```
'Orders_Info'
```

The first sheet (value of 0) is active by default, but you can change it by assigning the sheet's index to **workbook.active**

Add .title to return the title as text

Sheets can also be **referenced by name** (like dictionary keys)

```
workbook['Item_Info']
```

```
<Worksheet "Item_Info">
```

```
items = workbook['Item_Info']
```

```
inventory = workbook['Inventory_Levels']
```

```
orders = workbook['Orders_Info']
```

Assign sheet names to variables to make the workbook easier to navigate



# NAVIGATING CELLS

The openpyxl  
Package

Navigating  
Workbooks

Iterating Through  
Cells

Modifying Cells

You can **navigate cells** by using 'A1' style coordinates, or Python-esque indices

`sheet['coordinate']` returns the specified cell object, and `cell.value` returns the cell's contents

```
items['B1']  
<Cell 'Item_Info'.B1>
```

Remember that 'items' is the name  
assigned to `workbook['Item_Info']`

```
print(items["B1"].value)  
print(items["B4"].value)
```

Product\_Name  
Gloves

`sheet.cell(row= , column= )` returns cell objects as well

```
items.cell(row=4, column=2).value
```

This is equivalent to 'B4'  
(0 indexing doesn't apply!)

	A	B	C	D	E
1	Product_ID	Product_Name	Price	Cost	Available Sizes
2	10001	Coffee	5.99	0.99	250mL
3	10002	Beanie	9.99	4.29	Child, Adult
4	10003	Gloves	19.99	7.99	Child, Adult
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L, XL, XXL
6	10005	Helmet	99.99	49.99	Child, Adult
7	10006	Snow Pants	79.99	32.49	XS, S, M, L, XL, XXL
8	10007	Coat	119.99	54.55	S, M, L
9	10008	Ski Poles	99.99	69.99	S, M, L
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
11	10010	Skis	599.99	249.99	S, M, L
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
13	10012	Bindings	149.99	89.99	NA
14	10013	Snowboard	499.99	199.99	S, M, L, Powder

# ASSIGNMENT: NAVIGATING WORKBOOKS

 **NEW MESSAGE**  
February 12, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Tax Calculation**

Hey there, we've just run our annual Black Friday Sale, and there have been issues with the data. Most of our data team is skiing this weekend, so we need your help.

A customer mentioned they weren't charged sales tax and have graciously reached out to pay it.

Can you calculate the sales tax (8%) and total for customer C00003? It should be in row 10.

Thanks!

 [excel\\_data\\_sales\\_tax.ipynb](#)

**Results Preview**

```
import openpyxl as xl
```

[REDACTED]

[REDACTED]

[REDACTED]

**Sales Tax: \$8.0**  
**Total: \$107.99**

# ASSIGNMENT: NAVIGATING WORKBOOKS

 **NEW MESSAGE**  
February 12, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Tax Calculation**

Hey there, we've just run our annual Black Friday Sale, and there have been issues with the data. Most of our data team is skiing this weekend, so we need your help.

A customer mentioned they weren't charged sales tax and have graciously reached out to pay it.

Can you calculate the sales tax (8%) and total for customer C00003? It should be in row 10.

Thanks!

 [excel\\_data\\_sales\\_tax.ipynb](#)

## *Solution Code*

```
import openpyxl as xl  
  
wb = xl.load_workbook(filename='maven_ski_shop_data.xlsx')  
  
orders = wb['Orders_Info']
```

```
from tax_calculator import tax_calculator  
  
transaction = tax_calculator(orders['D10'].value, .08)  
  
print('Sales Tax: $' + str(round(transaction[1], 2)))  
print('Total: $' + str(round(transaction[2], 2)))
```

Sales Tax: \$8.0  
Total: \$107.99



# DETERMINING SHEET RANGES

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

**sheet.max\_row** and **sheet.max\_column** help determine the number of rows and columns with data in a worksheet, to then use as stopping conditions for loops

`items.max_row`

14

This returns the index of  
the last row with data

`items.max_column`

5

This returns the index of the  
last column with data  
('E' is the fifth column)

	A	B	C	D	E
1	Product_ID	Product_Name	Price	Cost	Available Sizes
2	10001	Coffee	5.99	0.99	250mL
3	10002	Beanie	9.99	4.29	Child, Adult
4	10003	Gloves	19.99	7.99	Child, Adult
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L, XL, XXL
6	10005	Helmet	99.99	49.99	Child, Adult
7	10006	Snow Pants	79.99	32.49	XS, S, M, L, XL, XXL
8	10007	Coat	119.99	54.55	S, M, L
9	10008	Ski Poles	99.99	69.99	S, M, L
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
11	10010	Skis	599.99	249.99	S, M, L
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
13	10012	Bindings	149.99	89.99	NA
14	10013	Snowboard	499.99	199.99	S, M, L, Powder



# LOOPING THROUGH CELLS

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

Excel columns usually contain data fields, while rows contain individual records

To **loop through cells** in a column, you need to move row by row in that column

```
for row in range(1, items.max_row + 1):
    print(f'B{row}', items[f'B{row}'].value)
```

```
B1 Product_Name → print('B1', items['B1'].value)
B2 Coffee → print('B2', items['B2'].value)
B3 Beanie → print('B3', items['B3'].value)
B4 Gloves
B5 Sweatshirt
B6 Helmet
B7 Snow Pants
B8 Coat
B9 Ski Poles
B10 Ski Boots
B11 Skis
B12 Snowboard Boots
B13 Bindings
B14 Snowboard
```



How does this code work?

- The for loop is iterating through each **row** in a specified range
- Since the end of **range** is not inclusive, we need to stop at **items.max\_row + 1**
- The range goes from 1-14, so the loop will run 14 times



# WRITING DATA TO CELLS

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

```
items['F1'].value
```

```
items['F1'] = 'Euro Price'
```

```
items['F1'].value
```

```
'Euro Price'
```

```
items.max_column
```

```
6
```

Cell 'F1' starts with no data, and then a value of 'Euro Price' is assigned to it

Now that column 'F' has data, max\_column has increased to 6



# WRITING DATA TO A COLUMN

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

You can **write data to a column** by looping through its rows and assigning values

```
exchange_rate = .88
for row in range(2, items.max_row + 1):
    items[f'F{row}'] = round(items[f'C{row}'].value * exchange_rate, 2)
```

Each row in column 'F' is now equal to the matching value in column 'C' multiplied by .88

Note that the range is starting at 2, since the first row is reserved for the column header

```
for index, cell in enumerate(items['F'], start=1):
    print(f'F{index}', cell.value)
```

F1 Euro Price  
F2 5.27  
F3 8.79  
F4 17.59  
F5 21.99  
F6 87.99  
F7 70.39  
F8 105.59  
F9 87.99  
F10 175.99  
F11 527.99  
F12 114.39  
F13 131.99  
F14 439.99

You can reference all the cells in a column!



Use **enumerate's** start argument to start the index at 1, rather than the default of 0, to align with Excel row numbers

# ASSIGNMENT: WRITING DATA TO A COLUMN

 **NEW MESSAGE**  
February 12, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: **New Currency Prices**

Hi again!

In addition to a planned EU expansion this year, we're considering expanding into Japan and the UK next year.

Since we're going to do this a few times, can you create a currency converter function?

Once we have that, create a column for 'GBP Price' and 'JPY Price', to store Pound and Yen prices.

The notebook has conversion rates and more details.

 [pound\\_and\\_yen\\_pricing.ipynb](#) Reply Forward

## Results Preview

```
def currency_converter(price,
```

Product_ID	Product_Name	Price	Cost	Available Sizes	Euro Price	GBP Price	JPY Price
10001	Coffee	5.99	0.99	250mL	5.27	4.55	736.77
10002	Beanie	9.99	4.29	Child, Adult	8.79	7.59	1228.77
10003	Gloves	19.99	7.99	Child, Adult	17.59	15.19	2458.77
10004	Sweatshirt	24.99	10.59	XS, S, M, L, XL, XXL	21.99	18.99	3073.77
10005	Helmet	99.99	49.99	Child, Adult	87.99	75.99	12298.77
10006	Snow Pants	79.99	32.49	XS, S, M, L, XL, XXL	70.39	60.79	9838.77
10007	Coat	119.99	54.55	S, M, L	105.59	91.19	14758.77
10008	Ski Poles	99.99	69.99	S, M, L	87.99	75.99	12298.77
10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11	175.99	151.99	24598.77
10010	Skis	599.99	249.99	S, M, L	527.99	455.99	73798.77
10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11	114.39	98.79	15988.77
10012	Bindings	149.99	89.99	NA	131.99	113.99	18448.77
10013	Snowboard	499.99	199.99	S, M, L, Powder	439.99	379.99	61498.77

# ASSIGNMENT: WRITING DATA TO A COLUMN

 NEW MESSAGE  
February 12, 2022

From: **Sally Snow** (Ski Shop Manager)  
Subject: New Currency Prices

Hi again!

In addition to a planned EU expansion this year, we're considering expanding into Japan and the UK next year.

Since we're going to do this a few times, can you create a currency converter function?

Once we have that, create a column for 'GBP Price' and 'JPY Price', to store Pound and Yen prices.

The notebook has conversion rates and more details.

 pound\_and\_yen\_pricing.ipynb

Reply

Forward

## *Solution Code*

```
def currency_converter(price, ex_rate=0.88):
    return round(price * ex_rate, 2)
```

*GBP Price with range:*

```
pound_exchange_rate = 0.76

items["G1"] = "GBP Price"

for row in range(2, items.max_row + 1):
    items["G" + str(row)] = currency_converter(
        items["C" + str(row)].value, pound_exchange_rate
    )
```

*JPY Price with enumerate:*

```
yen_exchange_rate = 123

for index, cell in enumerate(items["C"], start=1):
    if index == 1:
        items[f'H{index}'] = "JPY Price"
    else:
        items[f'H{index}'] = currency_converter(
            cell.value, yen_exchange_rate
        )
```



# INSERTING COLUMNS

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

You can **insert columns** to a worksheet without overwriting existing data

`sheet.insert_cols(idx=index)` inserts a column in the specified sheet and index

```
items['E1'].value
```

```
'Available Sizes'
```

*'Available Sizes' is the current value for cell 'E1'*

```
items.insert_cols(idx=5)
```

```
items['E1'] = 'Euro Price'
```

```
print('Column E header: ' + items['E1'].value)
print('Column F header: ' + items['F1'].value)
```

```
Column E header: Euro Price
```

```
Column F header: Available Sizes
```

*After inserting a column at index 5 (column 'E'), and assigning 'E1' a value of 'Euro Price', note that 'Available Sizes' was shifted right to 'F1'*



# DELETING COLUMNS

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

You can **delete columns** from a worksheet as well

`sheet.delete_cols(idx=index)` deletes the column at the specified sheet and index

```
items['E1'].value  
'Euro_Price'
```

'Euro\_Price' is now the current value for cell 'E1'

```
items.delete_cols(idx=5)  
  
print('Column E header: ' + items['E1'].value)
```

After deleting the column at index 5 (column 'E'),  
'Available Sizes' was moved back to 'E1' from 'F1'



# SAVING YOUR WORKBOOK

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

```
wb.save('maven_data_new_pricing.xlsx')
```



	A	B	C	D	E	F
1	Product_ID	Product_Name	Price	Cost	Available Sizes	Euro Price
2	10001	Coffee	5.99	0.99	250mL	5.27
3	10002	Beanie	9.99	4.29	Child, Adult	8.79
4	10003	Gloves	19.99	7.99	Child, Adult	17.59
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L, XL, XXL	21.99
6	10005	Helmet	99.99	49.99	Child, Adult	87.99
7	10006	Snow Pants	79.99	32.49	XS, S, M, L, XL, XXL	70.39
8	10007	Coat	119.99	54.55	S, M, L	105.59
9	10008	Ski Poles	99.99	69.99	S, M, L	87.99
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11	175.99
11	10010	Skis	599.99	249.99	S, M, L	527.99
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11	114.39
13	10012	Bindings	149.99	89.99	NA	131.99
14	10013	Snowboard	499.99	199.99	S, M, L, Powder	439.99

maven\_data\_new\_pricing.xlsx  
maven\_ski\_shop\_data.xlsx



## WARNING

This will replace any existing files with the same name in your folder

If you overwrite your input excel sheet, you could lose raw data that could be impossible to recover

Be careful!



# BRINGING IT ALL TOGETHER

The openpyxl  
Package

Navigating  
Workbooks

Looping Through  
Cells

Modifying Cells

You now have a workflow that reads in an Excel workbook, creates a 'Euro Price' column based on a given exchange rate, and saves it back out

```
import openpyxl as xl
```

Import openpyxl

```
workbook = xl.load_workbook(filename='maven_ski_shop_data.xlsx')
```

Read in the Excel workbook

```
items = workbook['Item_Info']
```

Assign the sheet to a variable

```
items['F1'] = 'Euro Price'
```

Write the column header

```
exchange_rate = .88
```

Set the exchange rate

```
for row in range(2, items.max_row + 1):
    items[f'F{row}'] = round(items[f'C{row}'].value * exchange_rate, 2)
```

Loop through the column,  
convert, and write the values

```
workbook.save('maven_data_new_pricing.xlsx')
```

Save the Excel workbook

# KEY TAKEAWAYS

---



The **openpyxl** package can manipulate Excel data using Python

- *You can read, modify, and save Excel workbooks without ever needing to open Excel*



Most of the **other Excel functionalities** are possible as well

- *You can create charts, apply conditional formatting, and write Excel formulas (just write them as strings in the cells where you want to place them!)*



This is just a sneak peak of Python's **capabilities to automate** external tasks

- *Beyond Excel, Python can manipulate flat files, read and write to databases, scrape web data, and more*

# FINAL PROJECT

# PROJECT DATA: BLACK FRIDAY ORDERS

## *Orders\_Info*

	A	B	C	D	E	F	G	H
1	Order_ID	Customer_ID	Order_Date	Subtotal	Tax	Total	Location	Items_Ordered
2	100000	C00004	11/26/2021	15.98			Sun Valley	10001, 10002
3	100001	C00007	11/26/2021	899.97			Stowe	10008, 10009, 10010
4	100002	C00015	11/26/2021	799.97			Mammoth	10011, 10012, 10013
5	100003	C00016	11/26/2021	117.96			Stowe	10002, 10003, 10004, 10006
6	100004	C00020	11/26/2021	5.99			Sun Valley	10001
7	100005	C00010	11/26/2021	599.99			Mammoth	10010
8	100006	C00006	11/26/2021	24.99			Mammoth	10004
9	100007	C00001	11/26/2021	1799.94			Mammoth	10008, 10008, 10009, 10009, 10009, 10010, 10010
10	100008	C00003	11/26/2021	99.99			Sun Valley	10005
11	100009	C00014	11/26/2021	254.95			Sun Valley	10002, 10003, 10004, 10006, 10007
12	100010	C00001	11/26/2021	29.98			Mammoth	10002, 10003
13	100011	C00001	11/26/2021	99.99			Mammoth	10005
14	100012	C00005	11/26/2021	25.98			Sun Valley	10001, 10003
15	100013	C00008	11/26/2021	649.98			Stowe	10012, 10013
16	100014	C00013	11/26/2021	89.99			Sun Valley	10014
17	100020	C00004	11/27/2021	119.99			Sun Valley	10007
18	100021	C00017	11/27/2021	599.99			Stowe	10010
19	100022	C00019	11/27/2021	649.98			Sun Valley	10012, 10013
20	100023	C00002	11/27/2021	24.99			Stowe	10004
21	100024	C00008	11/27/2021	99.99			Stowe	10005
22	100025	C00021	11/27/2021	99.99			Mammoth	10008
23	100026	C00022	11/27/2021	5.99			Sun Valley	10001
24	100027	C00006	11/28/2021	24.99			Mammoth	10002
25	100031	C00018	11/28/2021	999.96			Stowe	10005, 10008, 10009, 10010
26	100032	C00018	11/28/2021	99.99			Stowe	10006
27	100033	C00010	11/28/2021	399.97			Mammoth	10005, 10008, 10009
28	100034	C00016	11/28/2021	89.99			Stowe	10014



# PART 1: DATA PREP

---

 **NEW MESSAGE**  
February 13, 2022

**From:** **Sally Snow** (Ski Shop Manager)  
**Subject:** **Black Friday Data**

Hi again, I have a BIG final ask for you!  
Our Excel expert has been skiing for 3 months now, and we need really to analyze our Black Friday sales data!  
In the Excel workbook attached you'll see that we're missing data for taxes and totals. Can you fill in those rows of data using Python?  
You'll find more detailed instructions in the notebook attached, and I'll follow up shortly with more steps for the analysis.

 [maven\\_ski\\_shop\\_data.xlsx](#)  
[maven\\_ski\\_shop\\_analysis.ipynb](#)

---

## *Key Objectives*

---

1. Read in data from an Excel workbook
2. Define a function that prints cell contents
3. Create a dictionary using a comprehension and string methods
4. Use for loops to manipulate Excel data
5. Import and call a previously saved function
6. Write data into Excel cells
7. Save an Excel workbook

# PART 2: DATA ANALYSIS

 **NEW MESSAGE**  
February 13, 2022

**From:** **Sally Snow** (Ski Shop Manager)

**Subject:** **RE: Black Friday Data**

Great work on fixing the data, thank you!

Now the fun part – it's time for analysis.

We need to calculate some key metrics by aggregating our data, which will be super helpful for determining how well we performed during Black Friday.

You think you can pull it off? The detailed instructions should be in the notebook I sent earlier.

Can't wait to see the results!

 [maven\\_ski\\_shop\\_analysis.ipynb](#)

 [Reply](#)    [Forward](#)

## *Key Objectives*

1. Define a function that sums Excel columns, leveraging a list comprehension
2. Apply numerical functions to calculate KPIs
3. Use set operations to find unique items
4. Create a dictionary using nested loops
5. Challenge: Write a function that calculates the sum of an Excel column, grouped by the unique values in another column