



Introduction Python

Code Kickstart (Python Programming)

Link Google Colab :

- [Python Basic](#)



Profile

Hello I'm Iqbal Ridhalta Putra

I'm currently working as Data Analyst

Table of Contents

-
- 1 Introduction to Python & Tools
 - 2 Data Types : Numerical & Categorical
 - 3 Conditional Statement & Looping
 - 4 Data Handling
-





Introduction to Python & Tools

WHY PYTHON?

MEET PYTHON

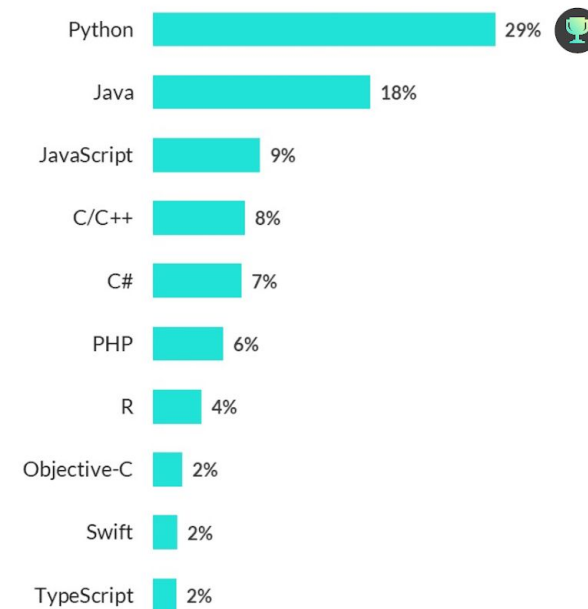


Python is a free, open-source programming language that is both powerful and easy to learn.

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):



Share of language tutorial searches on Google

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

MACHINE LEARNING, AI & IoT ECOSYSTEM

Machine Learning Frameworks

SK

Scikit-Learn

Comprehensive machine learning library with algorithms for classification, regression, clustering, and dimensionality reduction

XG

XGBoost

Optimized gradient boosting framework designed for speed and performance in machine learning competitions

LG

LightGBM

Fast, distributed, high-performance gradient boosting framework with lower memory usage and faster training speed

Deep Learning & Neural Networks

TF

TensorFlow

Open-source platform for machine learning with comprehensive ecosystem for developing and deploying ML models

PT

PyTorch

Dynamic neural network framework with strong GPU acceleration and intuitive Python-first development experience

KR

Keras

High-level neural networks API that provides simple and fast prototyping for deep learning models

AI Development & NLP

HF

Hugging Face

Platform providing pre-trained models and tools for natural language processing, computer vision, and audio tasks

SP

spaCy

Industrial-strength NLP library for advanced natural language processing in Python with pre-trained models

OA

OpenAI API

Access to powerful AI models including GPT for text generation, DALL-E for images, and Whisper for speech

IoT Development & Edge Computing

RP

Raspberry Pi

Single-board computer platform perfect for IoT projects, edge computing, and prototyping connected devices

AR

Arduino

Open-source electronics platform for building interactive projects with sensors, actuators, and microcontrollers

MP

MicroPython

Lightweight Python implementation optimized for microcontrollers and embedded systems in IoT applications

JUPYTER NOTEBOOKS





INSTALLING ANACONDA (MAC)

1) Go to **anaconda.com/products/distribution** and click

ANACONDA DISTRIBUTION

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

Download



2) Click **X** on the Anaconda Nucleus pop-up

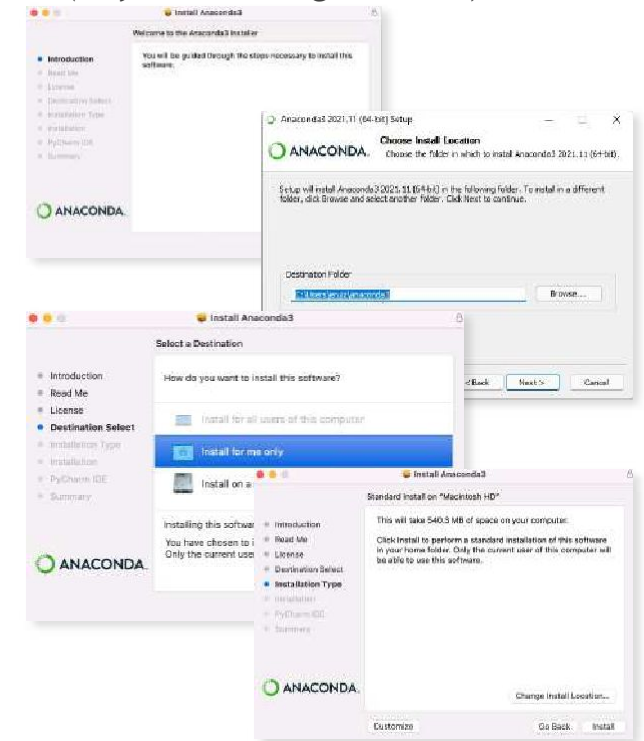
(no need to launch)



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

Anaconda3-202....pkg

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





INSTALLING ANACONDA (PC)

1) Go to **anaconda.com/products/distribution** and click

ANACONDA DISTRIBUTION

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

Download



2) Click **X** on the Anaconda Nucleus pop-up

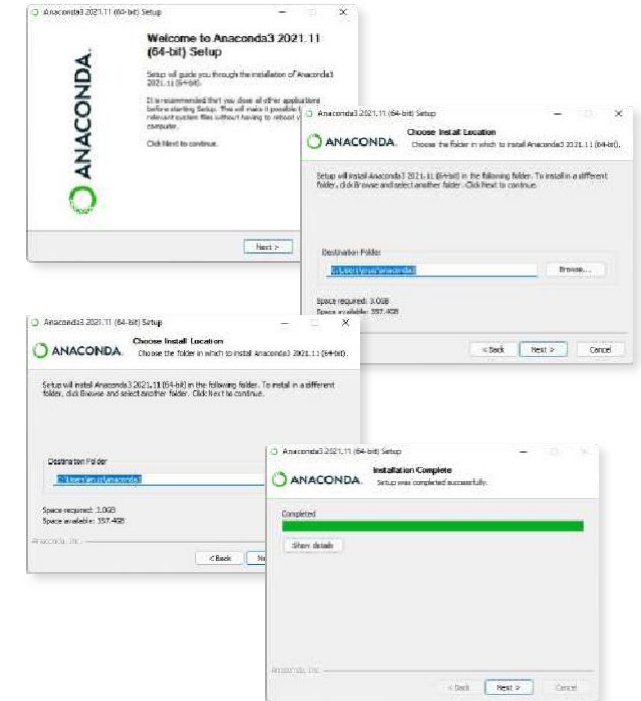
(no need to launch)



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

Anaconda3-2021.11....exe

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



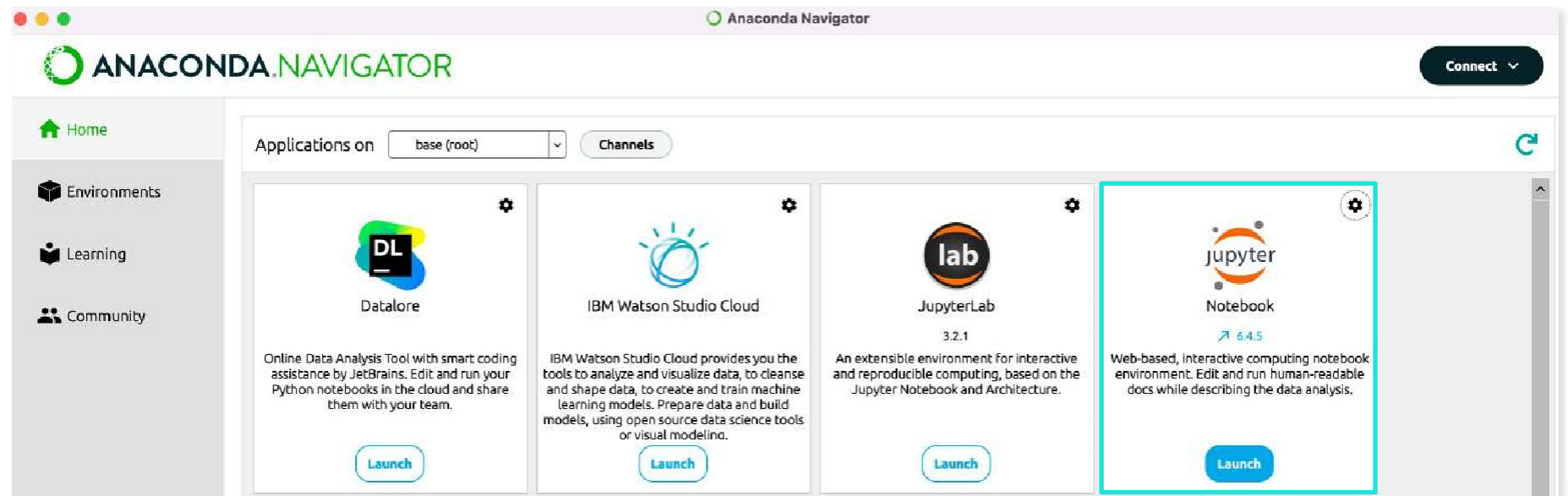


LAUNCHING JUPYTER

1) Launch **Anaconda Navigator**

2) Find **Jupyter Notebook** and click

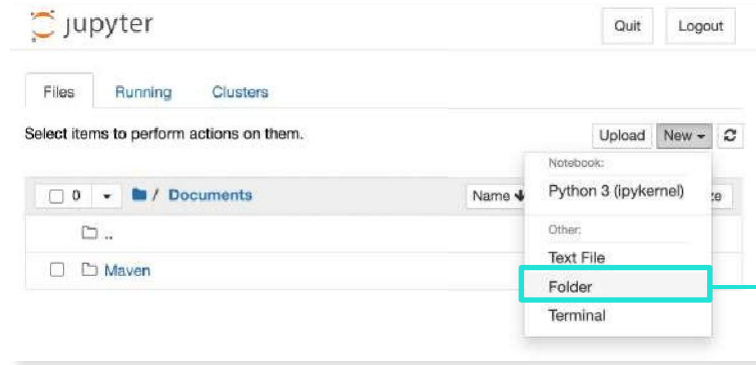
Launch





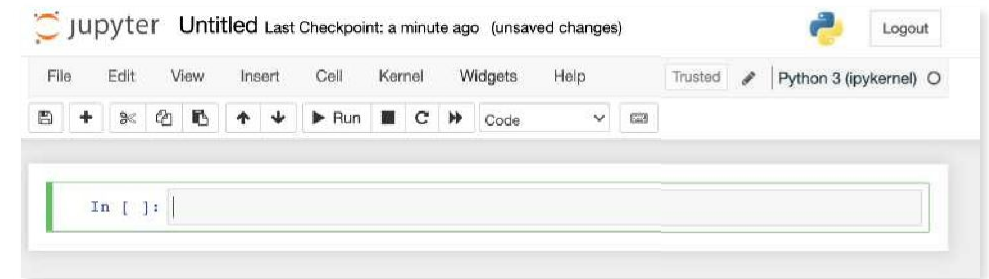
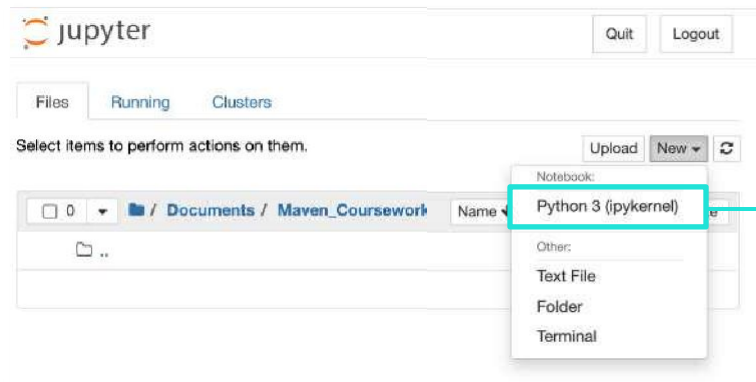
YOUR FIRST JUPYTER NOTEBOOK

1) 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

2) 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

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

```
python
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.ip
```



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

Menu Bar

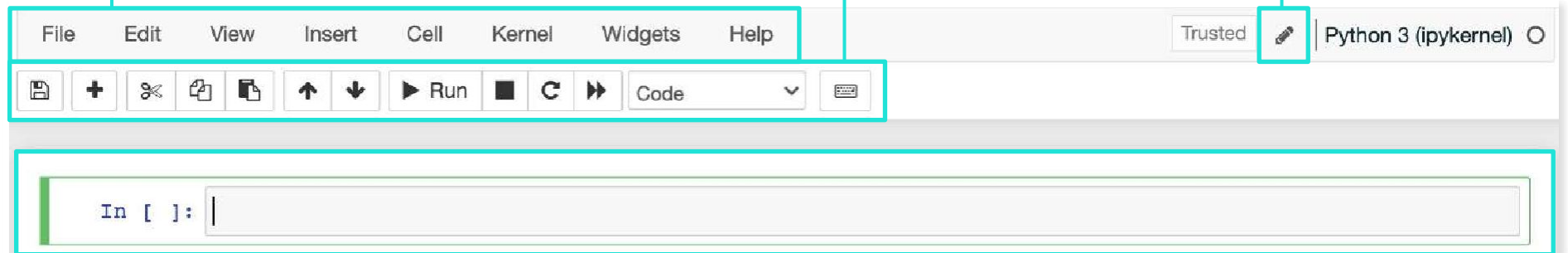
Options to manipulate the way the notebook functions

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 ~~edit new~~ code to be executed

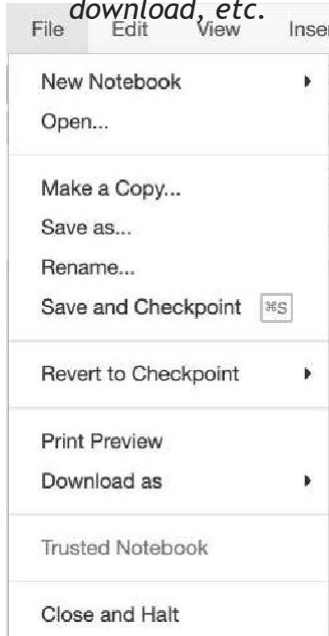




MENU OPTIONS

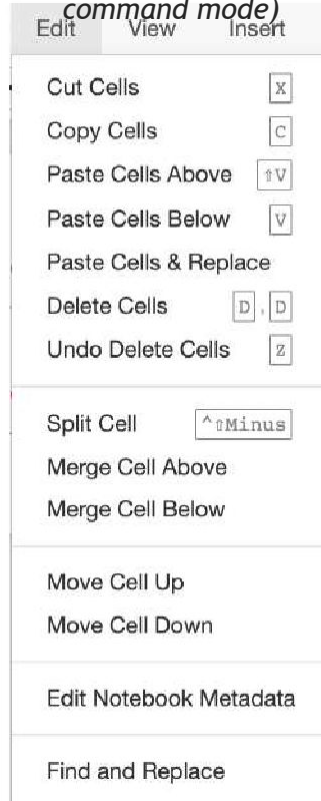
File

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



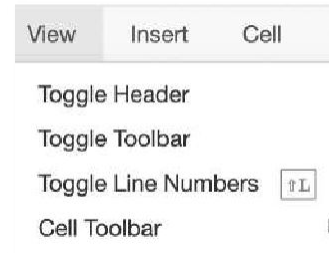
Edit

Edit cells within
your notebook
(while in
command mode)



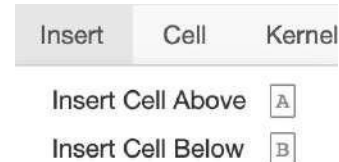
View

Edit cosmetic options
for your
notebook.



Insert

Insert new cells into
your
notebook





MENU OPTIONS

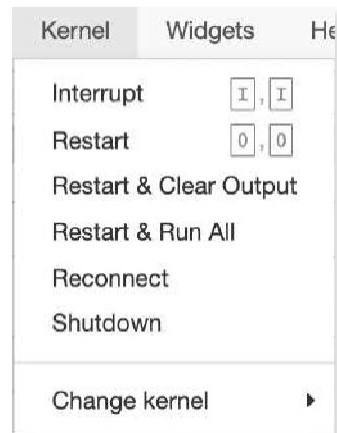
Cell

Access options for running the cells in your notebook



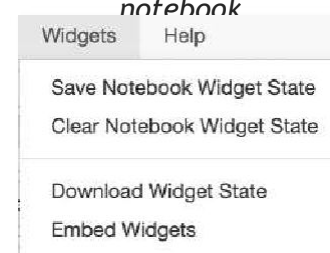
Kernel

Interact with the instance of Python that runs your code



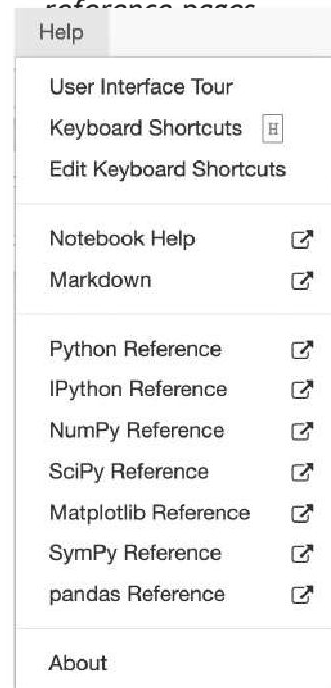
Widgets

Manage interactive elements, or 'widgets' in your notebook



Help

View or edit keyboard shortcuts and access Python reference pages

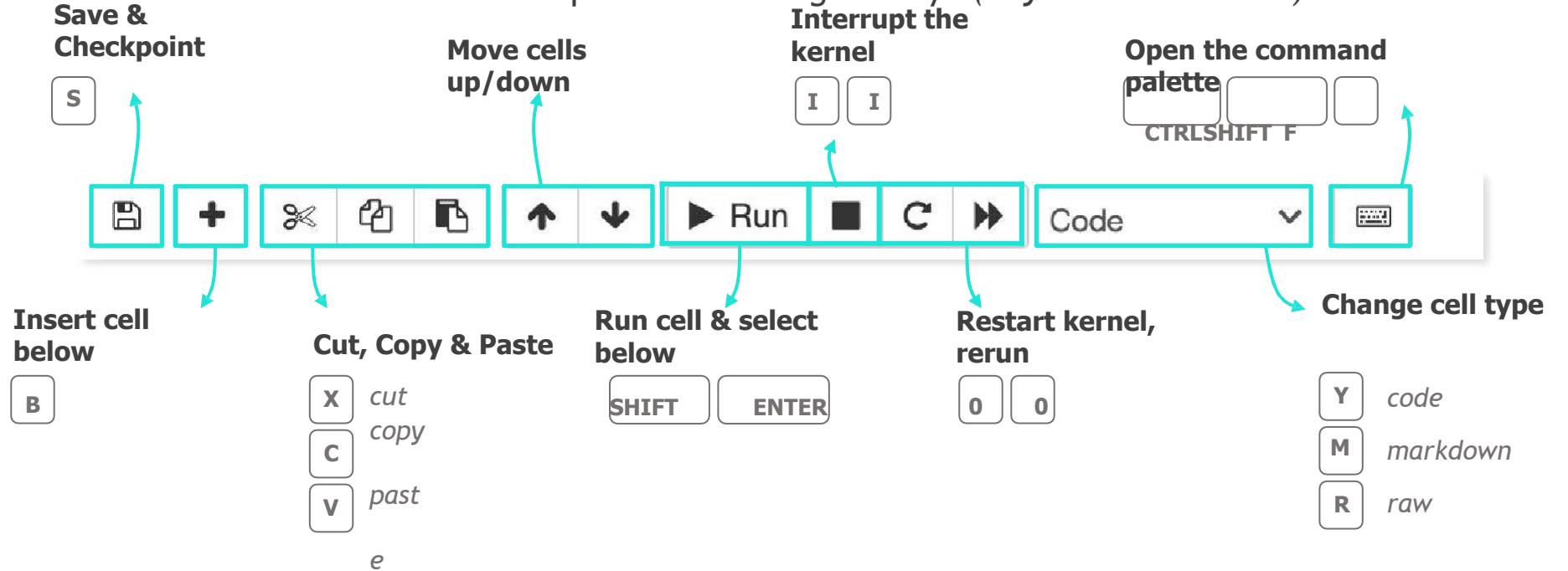




THE TOOLBAR

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

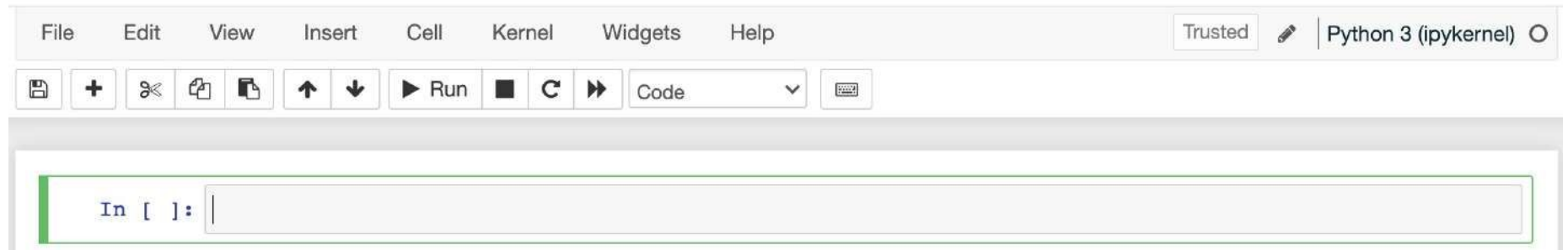
- These actions can also be performed using hotkeys (*keyboard shortcuts*)



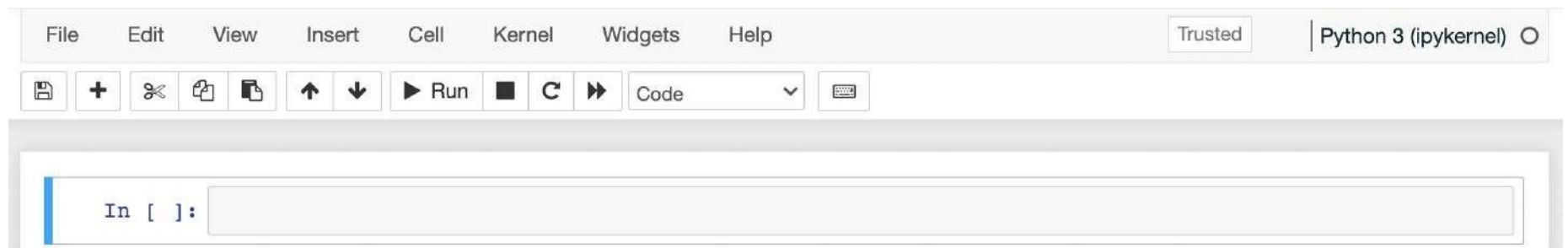


EDIT & COMMAND MODES

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



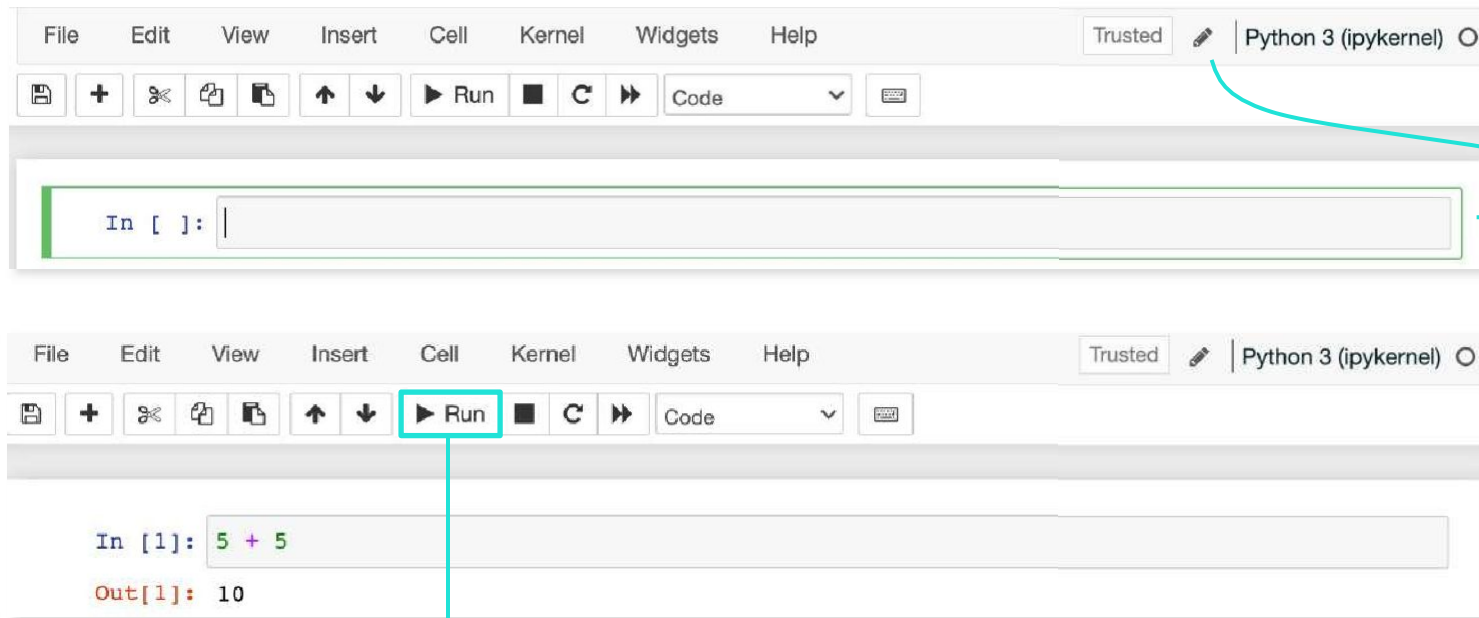
COMMAND MODE is for editing **the notebook**, and is indicated by **blue** highlights and no icon





THE CODE CELL

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



In **edit mode**, the cell will be highlighted **green** and a pencil icon will appear

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!*





THE CODE CELL

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
```

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

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[*]`

```
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

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
```

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

```
# 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
```

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



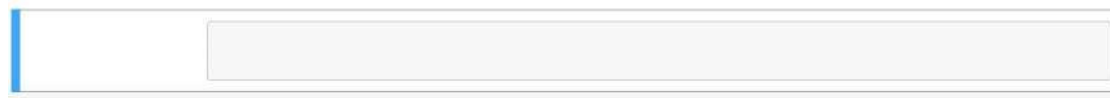
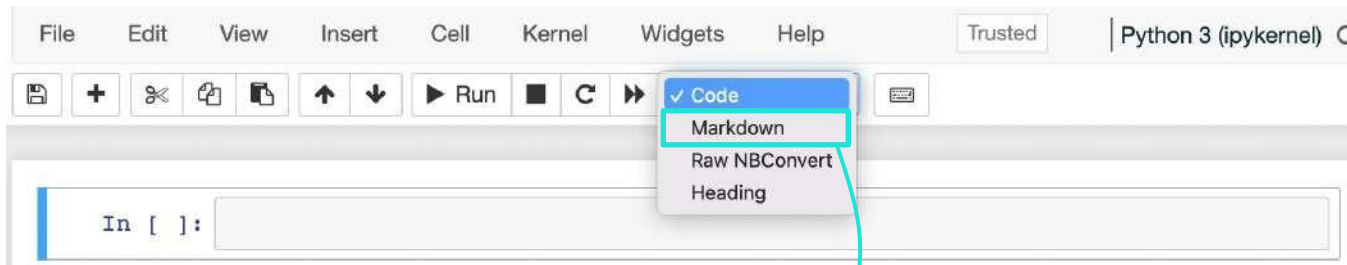


MARKDOWN CELLS

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

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

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

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



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

```
In [18]: print?
```

Docstring:
print(value, ..., sep=' ', end='\n')

Prints the values to a stream, or
Optional keyword arguments:
file: a file-like object (stream
sep: string inserted between va
end: string appended after the
flush: whether to forcibly flush
Type: builtin_function_or_me





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



ALTERNATIVE: GOOGLE COLAB

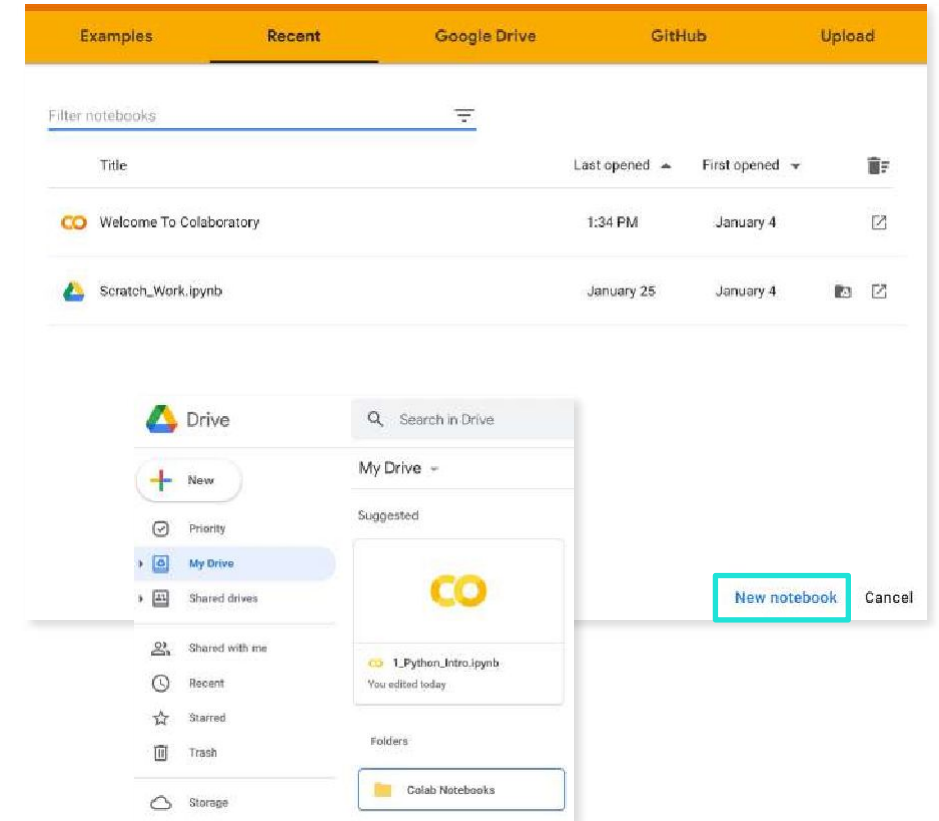
Google Colab is Google's cloud-based version of Jupyter Notebooks

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

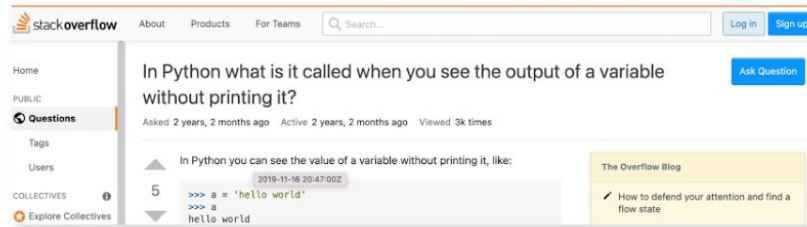




HELPFUL RESOURCES

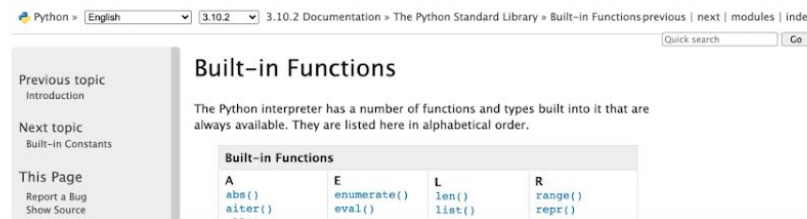


Google your questions – odds are someone else has asked the same thing and it has been answered (include *Python* in the query!)



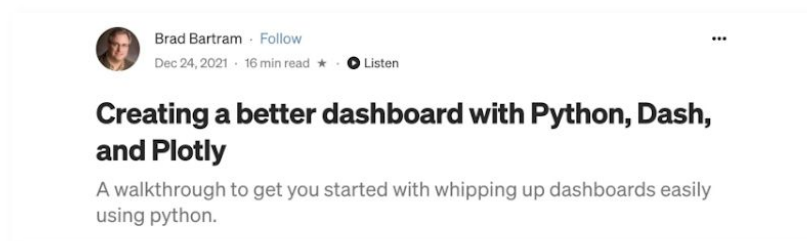
Stack Overflow is a public coding forum that will most likely have the answers to most of the questions you'll search for on Google

<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/>



Data Types



DATA TYPES



PYTHON DATA TYPES

Python has a series of built-in **data types** with different properties and use cases. These can be grouped into the following categories:

Numeric



Text



Boolean



None



Sequence



Mapping



Set





PYTHON DATA TYPES

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

The **type()** function will return the data type of the object passed to it

```
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

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

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



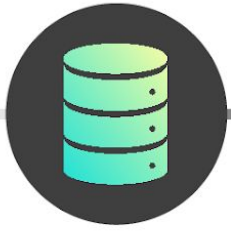
Set



Text



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



MUTABILITY

A data type is **mutable** if it can be modified after its creation

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

VARIABLES



VARIABLE ASSIGNMENT

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

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

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**



OVERWRITING 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

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

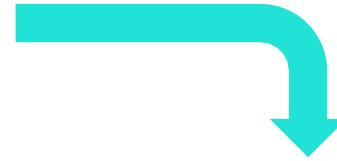
Then price stores the value 6, and 5 gets removed from memory

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

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

5 6

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



Memory

price 7

6 5



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



OVERWRITING 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

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

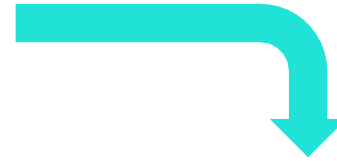
Then price stores the value 6, and 5 gets removed from memory

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

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

5 6

Or create a new variable for the old value



Memory

y
price 7

6 5



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



OVERWRITING 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

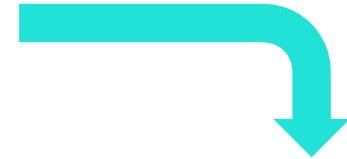
*Python stores the value of 5 in memory when it is assigned to **price***

*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
new_price	7

5



NAMING RULES

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

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

NUMERIC DATA



NUMERIC DATA TYPES

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

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

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)

Integers into floats:

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

(1.0, 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

+	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

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

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

Without Parentheses

$3*6 + 5 - 2**3 - 1$ Exponentiation first

$3*6 + 5 - 8 - 1$ Then multiplication

$18 + 5 - 8 - 1$ Finally, addition & subtraction

14

With Parentheses (to control execution)

The diagram illustrates the evaluation of the expression $3 * ((6 + 5) - 2 ** (3 - 1))$ using the order of operations (PEMDAS/BODMAS). The steps are shown in four horizontal panels, with arrows indicating the sequence of operations:

- Step 1:** The original expression is shown: $3 * ((6 + 5) - 2 ** (3 - 1))$. The innermost parentheses are highlighted: $(6 + 5)$ and $(3 - 1)$.
- Step 2:** The expression is updated to $3 * (11 - 2 ** 2)$. The result of the first addition, 11, is shown. The exponentiation $2 ** 2$ is now the next operation to be performed.
- Step 3:** The expression is updated to $3 * (11 - 4)$. The result of the exponentiation, 4, is shown. The subtraction $11 - 4$ is now the next operation to be performed.
- Step 4:** The expression is updated to $3 * 7$. The result of the subtraction, 7, is shown. The final multiplication $3 * 7$ is the last operation.

Arrows indicate the flow from the original expression to the final result, showing the sequence of operations: Addition & subtraction in inner parentheses first, Then exponentiation in outer parenthesis, Then subtraction in outer parenthesis, and Finally, multiplication.



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

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

*This rounds the number to
2 decimal places*

```
abs(-3)
```

3

*This will always return a
positive number*

```
round(3.1415926535)
```

3

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

```
round(3.51)
```

4

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



SUM, MIN & MAX

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

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

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

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

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*



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

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]
```

```
'!'
```

String[index] returns a specified character:

- *String[-1] returns the last character*



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

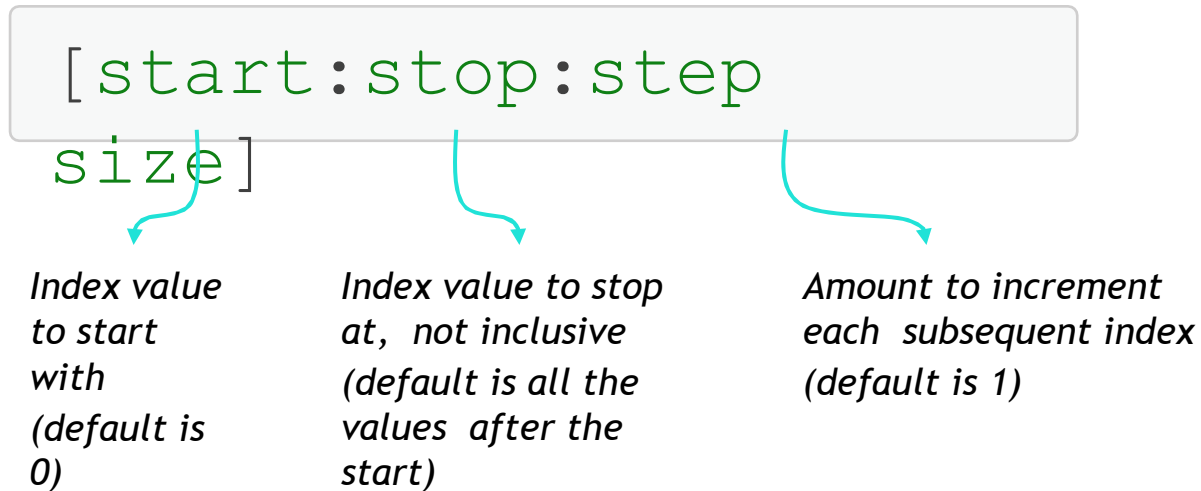
```
message[52]
```

IndexError: string index out of range



STRING SLICING

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



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

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

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

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

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

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

EXAMPLE

Grabbing components of a text string

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

```
'!tgnitn 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



THE LENGTH FUNCTION

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

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

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

9

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

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

3



METHODS

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

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*)**



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)
```

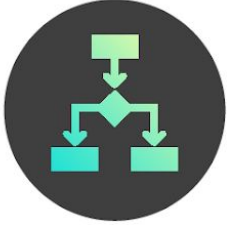
```
'I hope it snows tonight!'
```

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>

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:

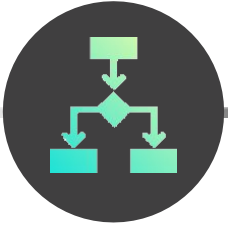
The Boolean Data Type

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** has two possible values: **True** & **False**

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

```
True * 1
```

1

```
False * 1
```

0

The **not** keyword inverts Boolean values:

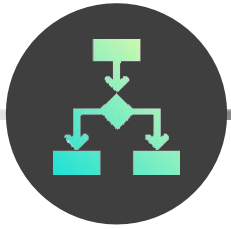
- **not True** is equivalent to **False**
- **not False** is equivalent to **True**

```
(not True) * 1
```

0

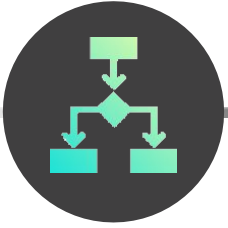
```
(not False) * 1
```

1



COMPARISON OPERATORS

	TRUE	FALSE
== Equal	<code>5 == 5</code>	<code>5 == 3</code>
!= Not Equal	<code>'Hello' != 'world!'</code>	<code>10 != 10</code>
< Less Than	<code>5 < 6 < 7</code>	<code>21 < 12</code>
> Greater Than	<code>10 > 5</code>	<code>10 > 5 > 7</code>
<= Less Than or Equal	<code>5 <= 5</code>	<code>(5 + 5) <= 8</code>
>= Greater Than or Equal	<code>11 >= 9 >= 9</code>	<code>len('I') >= len('am')</code>



MEMBERSHIP TESTS

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

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

```
'rain' in message
```

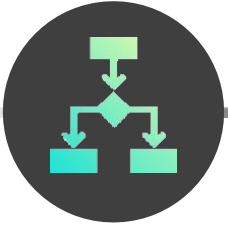
False

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

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

False

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



BOOLEAN OPERATORS

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

*You can chain together any number of Boolean
expressions*

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

True

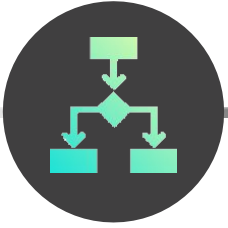
```
5 > 4 or 7 > 8
```

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

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')
```

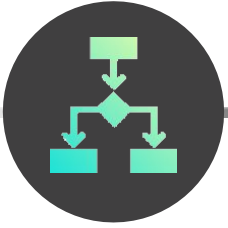
True

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



THE IF STATEMENT

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

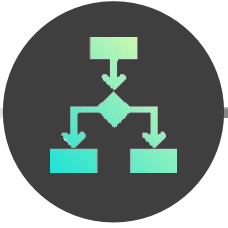
```
if condition:
```

*Indicates
an if
statement*

*A logical expression
that evaluates to True
or False*

Examples:

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

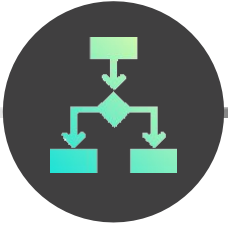


THE IF STATEMENT

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

EXAMPLE

Determining experience level for a snowboard

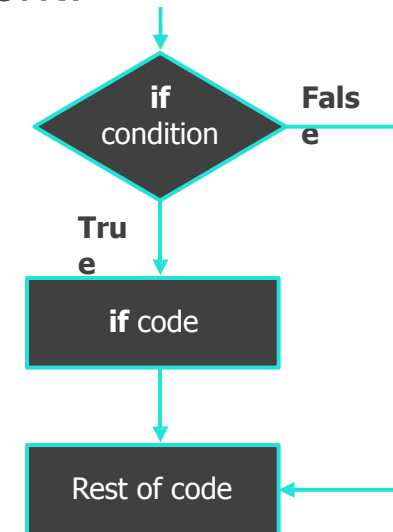
```
price = 999.99  
  
if price > 500:  
    print('This snowboard is designed for experienced users.')
```

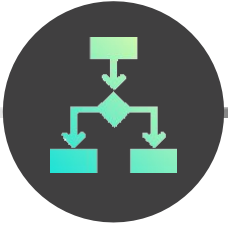
`print('This code will run whether or not the if statement is True.')`

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

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*

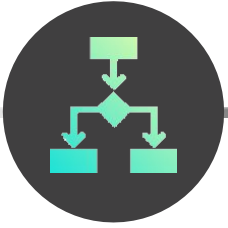
```
price = 499.99  
  
if price > 500:  
    print('This product is for experienced users.')  
print('This code will run whether or not the if statement is True.')
```

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

EXAMPLE

Determining experience level for a snowboard

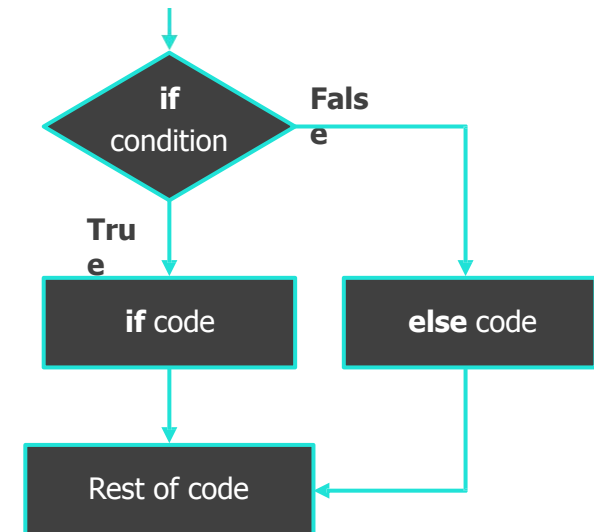
```
price = 499.99
price_threshold = 500

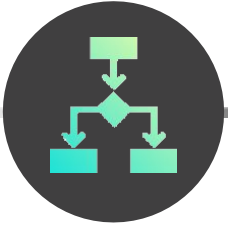
if price > price_threshold:
    print('This snowboard is for experienced users.')
else:
    print('This board is suitable for a beginner.')
```

This board is suitable for a beginner.



How does this code work?





THE ELIF STATEMENT

The **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

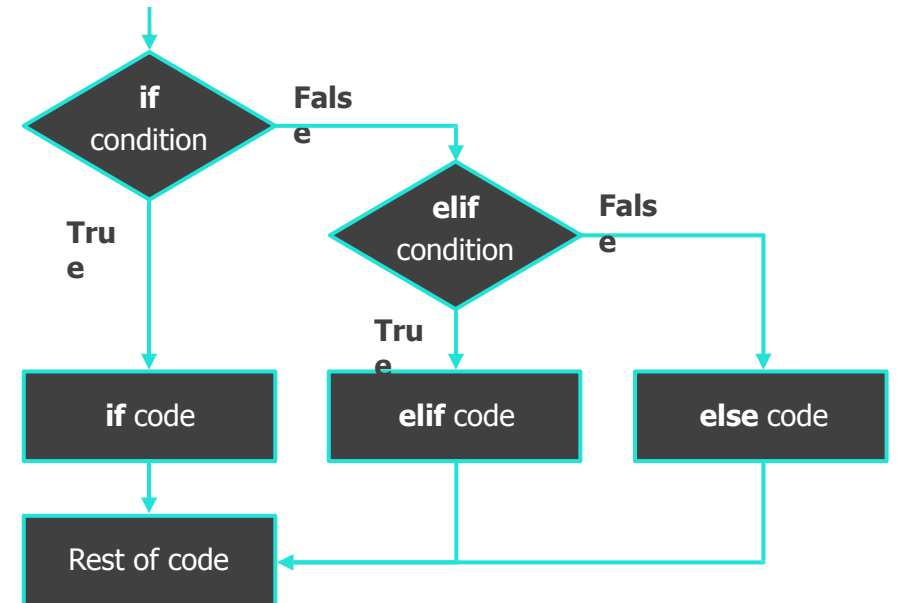
```
price = 499.99
expert_threshold = 500
intermediate_threshold = 300

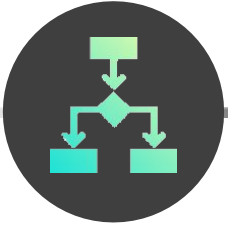
if price > expert_threshold:
    print('This board is for experienced users.')
elif price > intermediate_threshold:
    print('This board is good for intermediate_users.')
else:
    print('This should be suitable for a beginner.')
```

This board is good for intermediate_users.



How does this code work?





THE ELIF STATEMENT

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

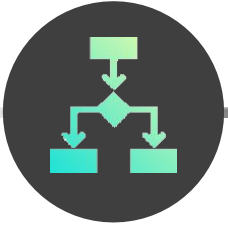
```
price = 1499.99
luxury_threshold = 1000
expert_threshold = 500
intermediate_threshold = 300

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



NESTED IF STATEMENTS

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

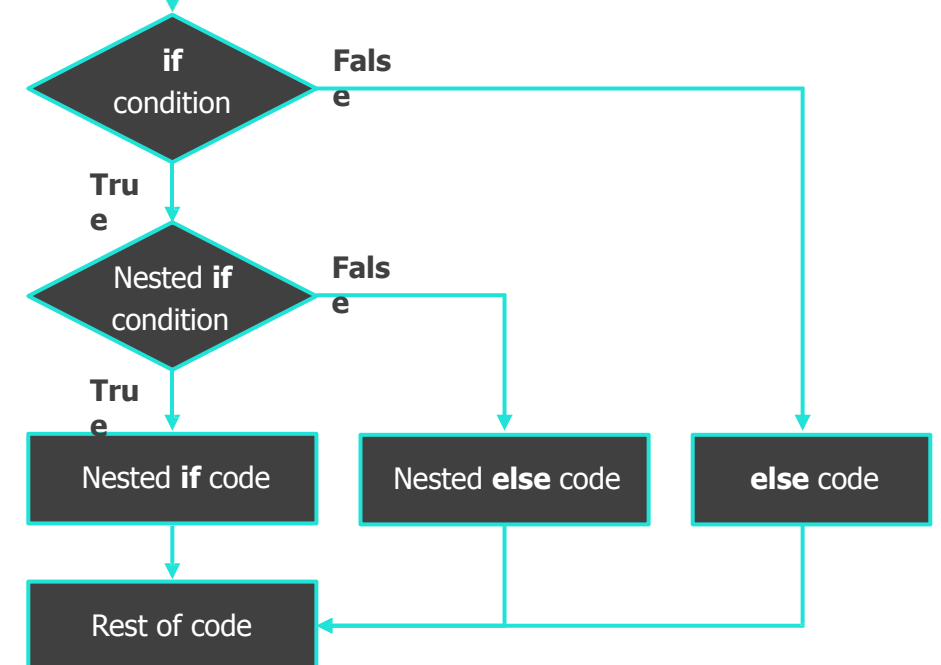
```
price = 499.99
budget = 500
inventory = 0

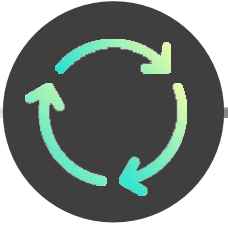
if budget > price:
    if inventory > 0:
        print('You can afford this and we have it in stock!')
    else: # equivalent to if inventory <= 0
        print("You can afford this but it's out of stock.")
else:
    print(f'Unfortunately, this board costs more than ${budget}.')
```

You can afford this but it's out of stock.



How does this code work?





LOOP BASICS

A **loop** is a block of code that will repeat until a given condition is met

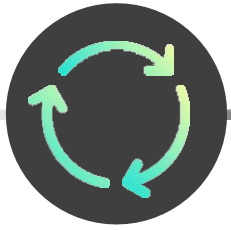
There are **two types** of loops:

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



FOR LOOPS

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*

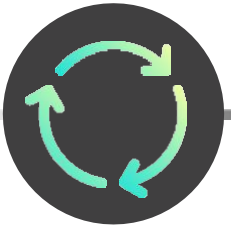
*A variable name for
each item in the
iterable*

Examples:

- Price
- Product
- Customer

*Iterable object
to loop
through*

- Examples:**
- List
 - Tuple
 - String
 - Dictionary
 - Set



FOR LOOPS

A **for loop** will run a specified number of times

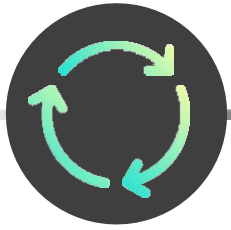
- This often corresponds with the length of a list, tuple, or other iterable data type

```
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

EXAMPLE

Printing individual letters in a string

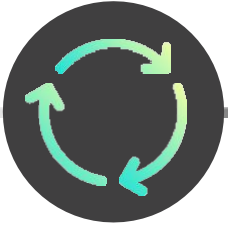
```
iterable = 'Maven'  
for item in iterable:  
    print(item)
```

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**



LOOP BASICS

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)
```

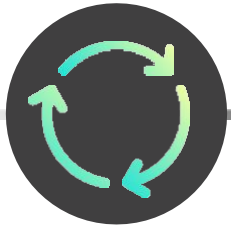
```
[5.27, 8.79, 17.59, 21.99, 87.99]
```

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

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]
```

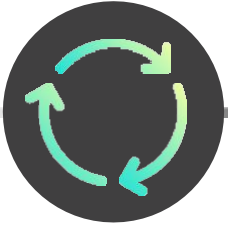


How does this code work?

- 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

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)

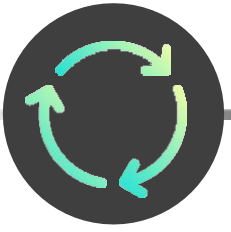
[5.27, 8.79, 17.59, 21.99, 87.99]
```

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

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)
```

The index is selecting
the elements in
the list

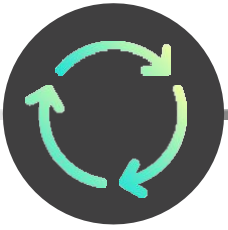
The for loop here is looping over indices in a range the size of the **euro_list**, meaning that the code will run 5 times (length of the list):

1. $i = 0$
2. $i = 1$
3. $i = 2$
4. $i = 3$
5. $i = 4$

```
[5.27, 8.79, 17.59, 21.99, 87.99]
```



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

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']

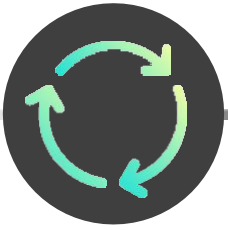
for i in range(len(euro_list)):
    print(f"The {item_list[i].lower()} costs {euro_list[i]} euros.")
```

```
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] = Snowboard`
- `euro_list[i] = 5.27`



PRO TIP: ENUMERATE

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)
```

*Use the index
for multiple
lists!*

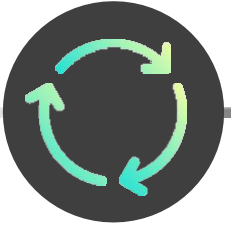
```
euro_list = [5.27, 8.79, 17.59, 21.99, 87.99]  
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']  
  
for index, element in enumerate(euro_list):  
    print(item_list[index], element)
```

```
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***



WHILE LOOPS

While loops run until a given logical expression becomes FALSE

- In other words, the loop runs *while* the expression is TRUE

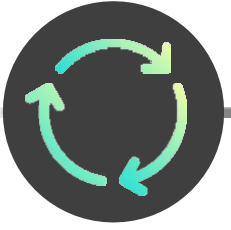
```
while logical expression:
```

*Indicates
a While
Loop*

*A logical expression that
evaluates to TRUE or
FALSE*

Examples:

- `counter < 10`
- `inventory > 0`
- `revenue > cost`



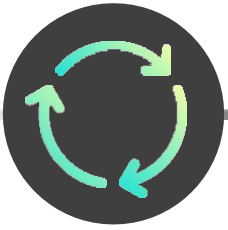
WHILE LOOPS

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

While loops often include **counters** that grow with each iteration

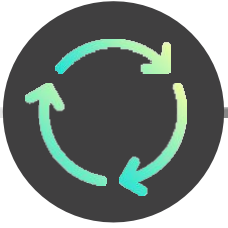
- Counters help us track how many times our loop has run
- They can also serve as a backup condition to exit a loop early (*more on this later!*)

```
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
- 2 ← iteration 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

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

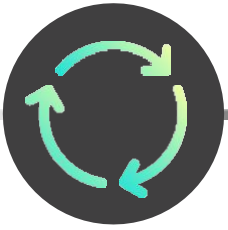
    print(f'At the end of year {year_counter}: '
          + f'My balance is ${round(stock_portfolio, 2)}')
```

```
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

EXAMPLE

Calculating bank balance until we're out of money.

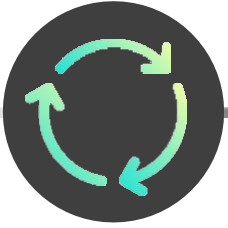
```
bank_balance = 5000
month_counter = 0

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

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



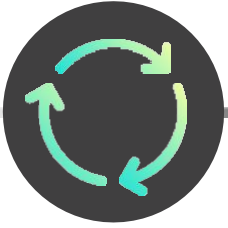
If your program is stuck in an infinite loop, you will need to **manually stop it** and modify your logic

▶ Run



Code





NESTED LOOPS

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']

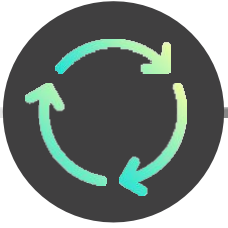
for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

```
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

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.
```



Data Gathering



DATA GATHERING PROCESS

Data Gathering melibatkan mencari data, membacanya ke dalam Python, melakukan transformasi jika diperlukan, dan menyimpan data dalam Pandas DataFrame.

1) Find the data

```
{
  "Name": ["Jim", "Dwight",
          "Angela", "Tobi"],
  "Age":  [26, 28, 27, 33],
  "Department": ["Sales", "Sales",
                 "Accounting",
                 "Human Resources"]
}
```

*Raw data can come
in many shapes and
forms*

2) Read in the data



*Apply transformations
using Python, if
necessary*

3) Store the data

	Name	Age	Department
0	Jim	26	Sales
1	Dwight	28	Sales
2	Angela	27	Accounting
3	Tobi	33	Human Resources

*Tables in Python are
stored as **Pandas
DataFrames**
(more on this later!)*

DATA SOURCES

Macam-macam Data Sources :



Local files

Data yang tersimpan dalam komputer lokal

Common examples:

- Flat files (.csv, .txt, etc.)
- Spreadsheets (.xlsx, etc.)



Databases

Data yang tersimpan dalam database

Common examples:

- SQL databases
- NoSQL databases



Web access

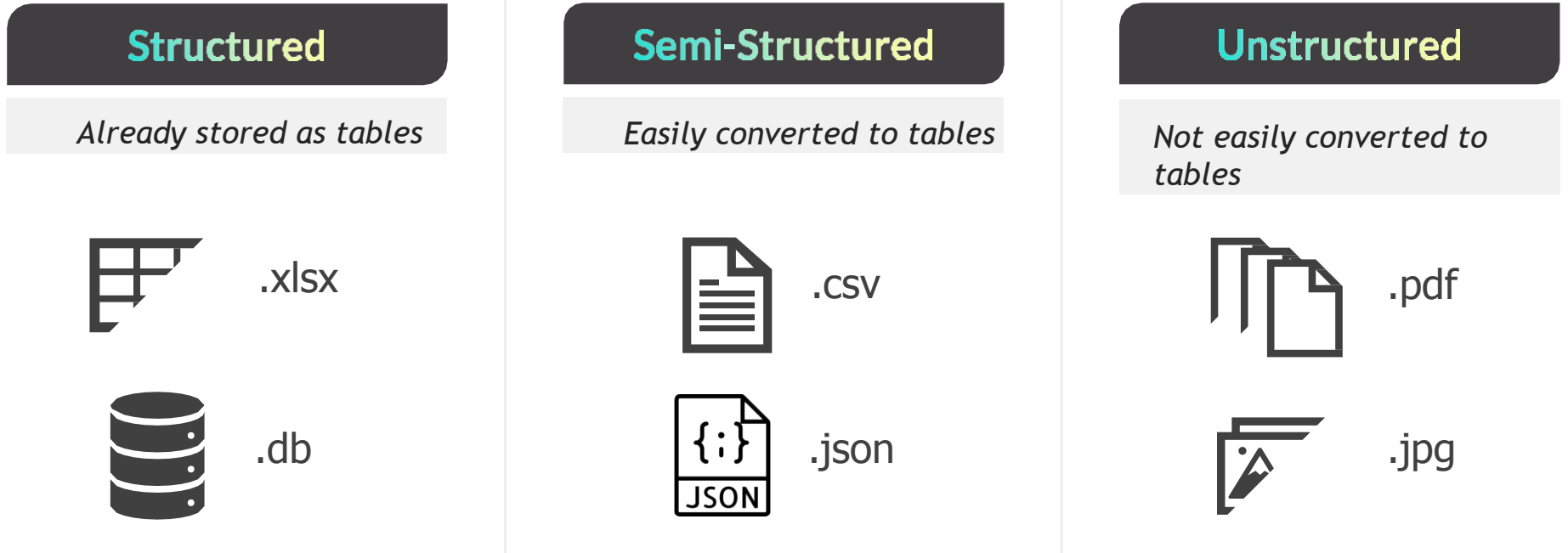
Data dari web

Common examples:

- Web scraping (.html, etc.)
- API (.json, .xml, etc.)



STRUCTURED VS UNSTRUCTURED DATA



Untuk melakukan analisa kedalam data, data yang dibutuhkan harus kedalam bentuk **structured data**.

- Sebagian besar sumber data sudah diorganisir dalam cara ini dan siap untuk di analisis.
- Sumber data yang tidak terstruktur memerlukan transformasi tambahan sebelum dilakukan analisis.

THE PANDAS DATAFRAME

Pandas adalah library dasar yang digunakan untuk melakukan manipulasi pada data frame. Data Frame adalah tabel/data tabular dengan array dua dimensi yaitu baris dan kolom.

Column Label/ Header

01234

Index Label

Column Index

01234

Row Index

Column

Element/ Value/ Entry

Row

	0	1	2	3	4	
	Name	Age	Marks	Grade	Hobby	
0	S1	Joe	20	85.10	A	Swimming
1	S2	Nat	21	77.80	B	Reading
2	S3	Harry	19	91.54	A	Music
3	S4	Sam	20	88.78	A	Painting
4	S5	Monica	22	60.55	B	Dancing

THE PANDAS DATAFRAME

Pandas memiliki fungsi untuk membaca berbagai jenis data. Function yang umum digunakan untuk mengimport data yaitu

Function	Deskripsi
<code>pd.read_csv</code>	mengimport data berformat csv file
<code>pd.read_excel</code>	mengimport data berformat excel atau xlsx
<code>pd.read_table</code>	mengimport data berformat dat
<code>pd.read_txt</code>	mengimport data berformat txt file
<code>pd.read_json</code>	mengimport data berformat json (Java Script Object Notation)



Data Manipulation



Data Manipulation

Manipulasi data adalah proses mengubah, mengorganisir, menggabungkan, atau mengolah data dalam cara tertentu agar lebih cocok atau lebih sesuai untuk analisis atau keperluan lainnya. Tujuan utama dari manipulasi data adalah untuk mendapatkan pemahaman yang lebih baik tentang data, menghapus atau memperbaiki data yang tidak valid, serta menyusun data dalam format yang lebih berguna dan mudah diolah

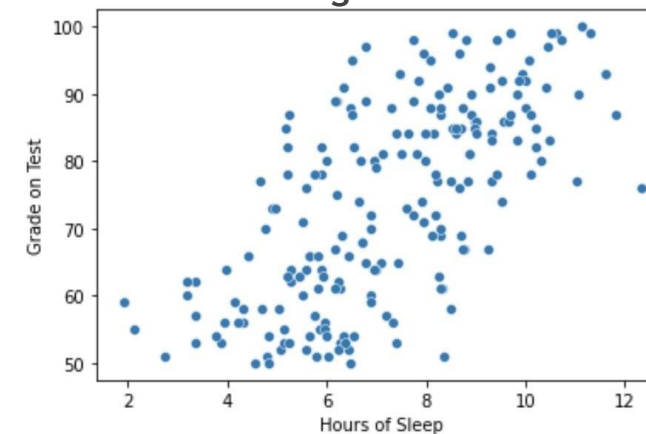
- Mendapatkan pemahaman yang lebih baik tentang data dengan melihatnya dari berbagai sudut
- Mengungkap pola, hubungan, dan wawasan dari data

*From
data...*

	Hours of Sleep	Hours Studied	Grade on Test
0	10.602667	4.586892	99
1	8.172997	4.405120	72
2	6.430132	-0.519630	52
3	7.963793	5.004348	80
4	8.279421	2.984489	87

EDA

*... to
insights*



COMMON Data Manipulation TECHNIQUES

Berikut adalah beberapa teknik Data Manipulation umum yang digunakan oleh Data Scientist:

Exploring data

*Viewing & summarizing
data from multiple angles*

Examples:

- Filtering
- Sorting
- Grouping / Agregasi
- Merging
- Encoding



Tidak ada urutan khusus yang harus diikuti dalam menyelesaikan tugas-tugas ini;
Anda bebas untuk menggabungkan dan memadukan mereka sesuai dengan kumpulan data Anda.

FILTERING

Anda dapat menyaring DataFrame dengan melewati uji logika ke aksesori **loc[]**

- Terapkan beberapa filter dengan menggunakan operator "&" dan "|" (AND/OR)

tea

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190



```
tea.loc[tea.type == 'herbal']
```

	type	name	temp
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175

This returns rows where "type" is equal to "herbal"

```
mask = (tea.type == 'herbal') | (tea.temp >= 200)  
tea.loc[mask]
```

	type	name	temp
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175

This returns rows where "type" is equal to "herbal" OR "temp" is greater than or equal to 200

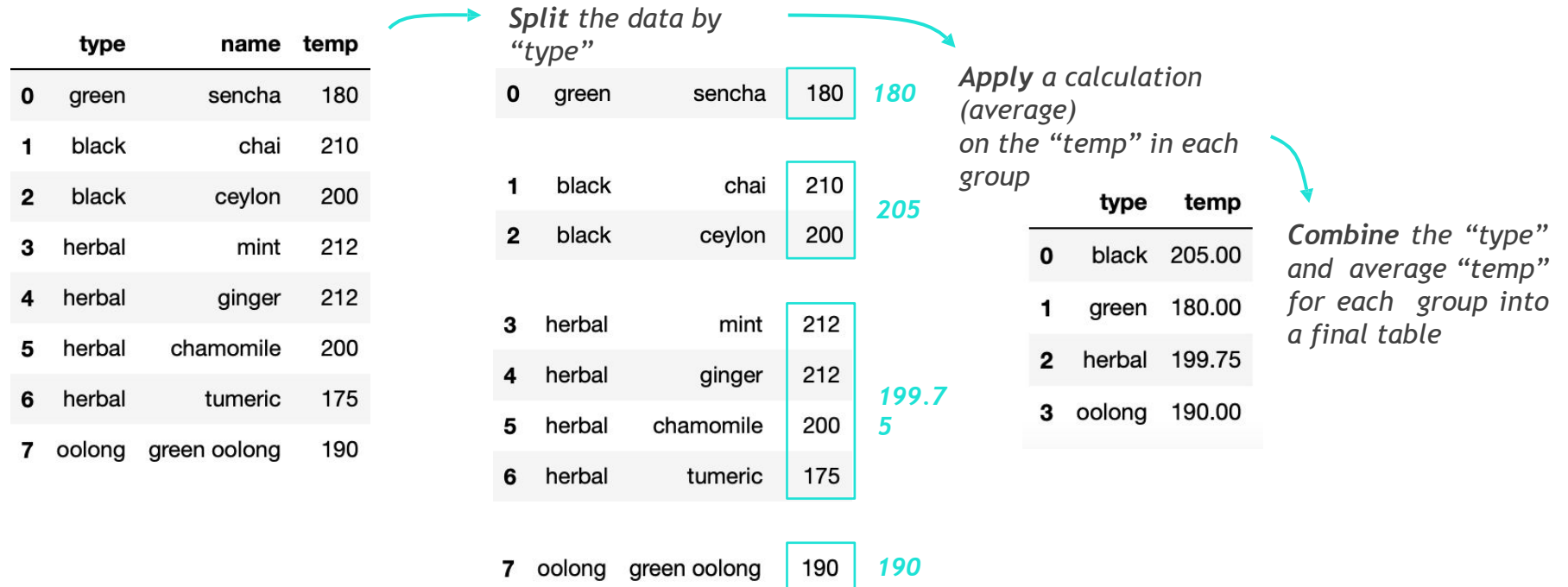
← Use a **Boolean mask** to apply multiple filters with complex logic

GROUPING

Pendekatan "split-apply-combine" digunakan untuk mengelompokkan data dalam DataFrame dan menerapkan perhitungan pada setiap kelompok

EXAMPLE

Mencari rata-rata suhu untuk setiap jenis teh



GROUPING

Anda dapat mengelompokkan DataFrame dengan menggunakan metode **.groupby()**

```
df.groupby(col)[col].aggregation()
```

The
DataFrame
to group

The column(s) to group
by
(unique values
determine the **rows**
in the output)

The column(s) to
apply the
calculation to
(these become the
new
columns in the
output)

The calculation(s)
to apply for each
group
(these become the
new
values in the
output)

Examples:

- `mean()`
- `sum()`
- `min()`
- `max()`
- `count()`
- `nunique()`

GROUPING

tea

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190



```
tea.groupby('type')['temp'].mean()
```

```
type
black    205.00
green    180.00
herbal    199.75
oolong    190.00
Name: temp, dtype: float64
```

This returns the
average
“temp” by “type”

```
tea.groupby('type')['temp'].mean().reset_index()
```

	type	temp
0	black	205.00
1	green	180.00
2	herbal	199.75
3	oolong	190.00

Use
reset_index() to
return a
DataFrame



MULTIPLE AGGREGATIONS

Dapat menggabungkan metode `.agg()` dengan `.groupby()` untuk menerapkan beberapa agregasi pada setiap kelompok

```
tea.groupby('type')['temp'].agg(['min', 'max', 'count', 'nunique']).reset_index()
```

	type	min	max	count	nunique
0	black	200	210	2	2
1	green	180	180	1	1
2	herbal	175	212	4	3
3	oolong	190	190	1	1

This returns the minimum & maximum temperatures, the number of teas, and the unique temperatures for each tea type



You can also write the code this way!

```
(tea.groupby('type')['temp']  
  .agg(['min', 'max', 'count', 'nunique'])  
  .reset_index())
```



PRO TIP: Ketika menggabungkan beberapa metode bersama-sama, bungkus kode dalam tanda kurung sehingga Anda dapat meletakkan setiap metode pada baris terpisah (ini membuat membaca kode menjadi lebih mudah!)

APPENDING

Gunakan **pd.concat()** untuk menggabungkan, atau menumpuk secara vertikal, beberapa DataFrame

- Kolom-kolom pada DataFrame harus identik
- **pd.concat([df_1, df_2])** akan menumpuk baris-baris dari "df_2" di bagian bawah "df_1"

sales_may

	date	store	sales
0	2022-05-01	1	341
1	2022-05-01	2	291
2	2022-05-01	3	493

sales_june

	date	store	sales
0	2022-06-01	1	67
1	2022-06-01	2	144
2	2022-06-01	3	226



pd.concat([sales_may, sales_june])

	date	store	sales
0	2022-05-01	1	341
1	2022-05-01	2	291
2	2022-05-01	3	493
0	2022-06-01	1	67
1	2022-06-01	2	144
2	2022-06-01	3	226



Chain .reset_index() to the code to make the index go from 0 to 5

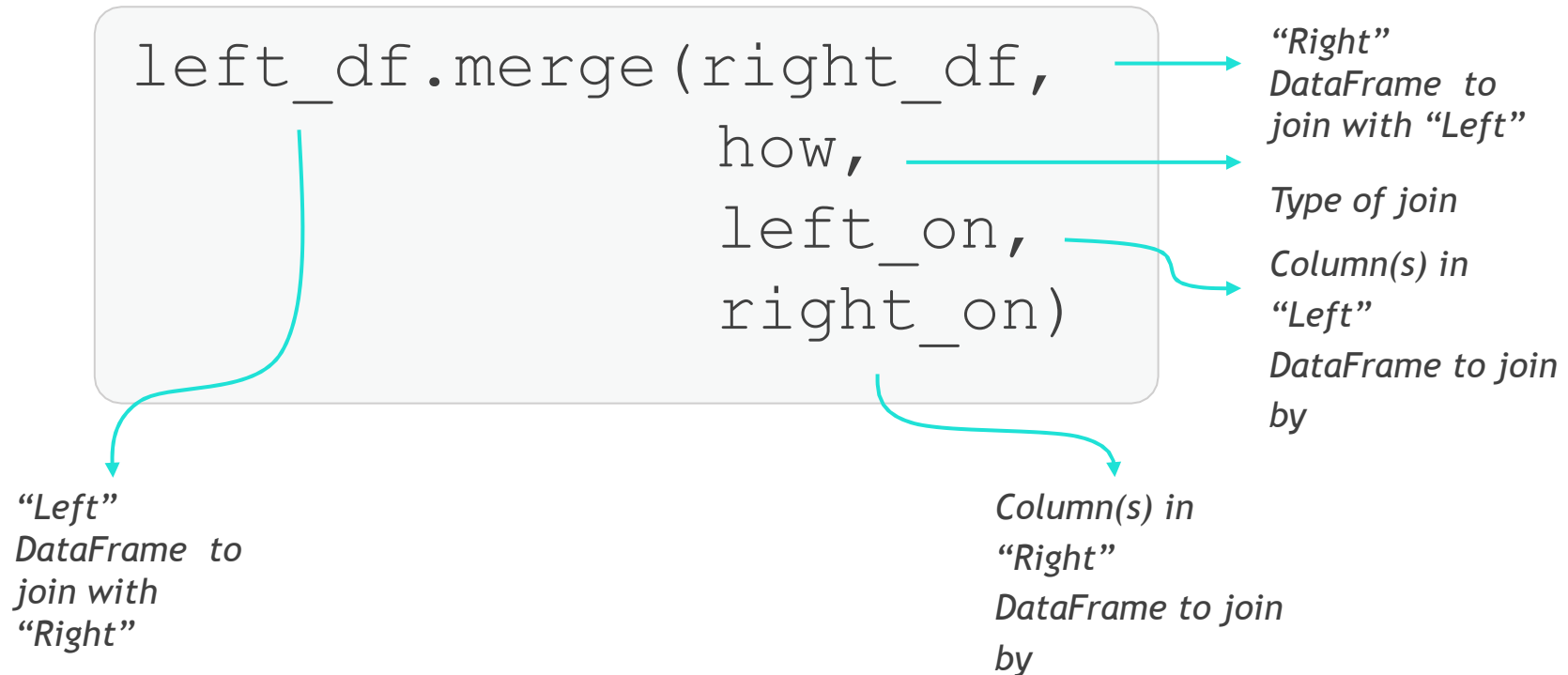


PRO TIP: Anda juga dapat menggunakan `.concat()` untuk menggabungkan DataFrame secara horizontal dengan mengatur `axis = 1`

JOINING

Gunakan **.merge()** untuk menggabungkan dua DataFrame berdasarkan nilai yang sama dalam satu atau beberapa kolom

- DataFrame harus memiliki setidaknya satu kolom dengan nilai yang cocok
- Ini berbeda dari metode `.join()` pada Pandas, yang menggabungkan DataFrames berdasarkan indeks mereka



JOINING

sales_may

	date	store	sales
0	2022-05-01	1	341
1	2022-05-01	2	291
2	2022-05-01	3	493
3	2022-05-01	4	428
4	2022-05-01	5	152

regions

	store	region
0	2	North
1	3	East
2	4	West



```
sales_may.merge(regions,  
                 how='left',  
                 left_on='store',  
                 right_on='store')
```

	date	store	sales	region
0	2022-05-01	1	341	NaN
1	2022-05-01	2	291	North
2	2022-05-01	3	493	East
3	2022-05-01	4	428	West
4	2022-05-01	5	152	NaN

← This added the region from the "regions" table to the "sales" table based on the "store" field



Data Cleaning



DATA CLEANING OVERVIEW

Tujuan utama dari Data cleaning adalah mengubah raw data menjadi data yang siap dianalisis.

Hal-hal umum pada data cleaning :

- Mengubah kolom menjadi tipe data yang tepat untuk analisis
- Menangani masalah data yang dapat mempengaruhi hasil analisis Anda
- Membuat kolom baru dari kolom yang sudah ada yang bermanfaat untuk analisis

Note : tidak ada urutan yang pasti dalam melakukan data cleaning karena proses data cleaning akan dipengaruhi dari jenis dataset.

DATA TYPES

Ketika menggunakan Pandas untuk membaca data, kolom secara otomatis diberikan tipe data.

Gunakan atribut **.dtypes** untuk melihat tipe data untuk setiap kolom dalam DataFrame. *Perlu diingat bahwa terkadang kolom numerik (int, float) dan kolom tanggal & waktu (datetime) tidak dikenali dengan benar oleh Pandas dan dibaca sebagai kolom teks (object).*

```
df = pd.read_csv('customers.csv')  
df.head()
```

	Name	City	State	Income	Birthdate
0	Susan Rodriguez	Maplewood	NJ	\$45,000	1970-09-12
1	Joseph Martinez	Portland	OR	\$120,000	1960-12-10
2	Joseph Martinez	Portland	OR	\$120,000	1960-12-10
3	Wayne Nielson	Dahlgren	VA	\$75,000	1968-12-05
4	Beverly Nixon	Long Island	NY	\$45,000,000	1962-03-05

Default data
types

df.dtypes

```
Name      object  
City       object  
State      object  
Income     object  
Birthdate  object  
dtype: object
```

These need to be converted to be analyzed!

Data Type	Description
bool	Boolean, True/False
int64	Integers
float64	Floating point, decimals
object	Text or mixed values
datetime64	Date and time values



PRO TIP: Gunakan metode `.info()` sebagai alternatif untuk menampilkan informasi tambahan bersama dengan tipe data.

CONVERTING TO DATETIME

Gunakan `pd.to_datetime()` untuk mengubah kolom objek menjadi kolom datetime.

df.Birthdate

```
0    1970-09-12
1    1960-12-10
2    1960-12-10
3    1968-12-05
4    1962-03-05
5    1993-09-23
6    1996-03-12
7    1994-01-13
8    1967-08-27
9             NaN
10            NaN
11            NaN
12    1968-05-19
```

Name: Birthdate, dtype: object



```
# convert to a datetime column
df.Birthdate = pd.to_datetime(df.Birthdate)
df.Birthdate
```

```
0    1970-09-12
1    1960-12-10
2    1960-12-10
3    1968-12-05
4    1962-03-05
5    1993-09-23
6    1996-03-12
7    1994-01-13
8    1967-08-27
9             NaT
10            NaT
11            NaT
12    1968-05-19
```

Name: Birthdate, dtype: datetime64[ns]

Perhatikan bahwa nilai yang hilang sekarang ditampilkan sebagai NaT (Not a Timestamp) bukan NaN (Not a Number).



PRO TIP: Pandas cukup baik dalam mendeteksi nilai datetime dari objek jika teksnya dalam format standar (seperti "YYYY-MM-DD"), tetapi Anda juga dapat secara manual menentukan format tersebut menggunakan argumen format di dalam fungsi `pd.to_datetime()`: **`pd.to_datetime(dt_col, format='%Y-%M-%D')`**

CONVERTING TO NUMERIC

Gunakan **pd.to_numeric()** untuk mengubah kolom objek menjadi kolom numerik.

- Untuk menghapus karakter non-numerik (\$, %, dll.), gunakan **str.replace()**.

df.Income

```
0      $45,000
1     $120,000
2     $120,000
3      $75,000
4   $45,000,000
5      $50,000
6     $105,000
7      $62,000
8      $60,000
9          NaN
10     $80,000
11          NaN
12    $110,000
Name: Income, dtype: object
```

Data mata uang dibaca sebagai objek oleh Python karena adanya simbol dolar (\$) dan pemisah koma (,)

```
# remove all dollar signs and commas
clean_income = df.Income.str.replace('$', '').str.replace(',', '')
clean_income
```

```
0      45000
1     120000
2     120000
3      75000
4   45000000
5      50000
6     105000
7      62000
8      60000
9          NaN
10     80000
11          NaN
12     110000
Name: Income, dtype: object
```

Name: Income, dtype: object

```
# convert to a numeric column
df.Income = pd.to_numeric(clean_income)
```

df.dtypes

```
Name      object
City      object
State     object
Income    float64
Birthdate datetime64[ns]
dtype: object
```



Alternatifnya adalah menggunakan `Series.astype()` untuk mengonversi menjadi tipe data yang lebih spesifik seperti 'int', 'float', 'object', dan 'bool', tetapi `pd.to_numeric()` dapat mengatasi nilai yang hilang (NaN), sedangkan `Series.astype()` tidak bisa.

DATA ISSUES OVERVIEW

Data Issues perlu diidentifikasi dan diperbaiki sebelumnya agar tidak mempengaruhi atau memutarbalikkan hasil analisis. Masalah umum dalam data yang "berantakan" meliputi:

	Name	City	State	Income	Birthdate	
	Susan Rodriguez	Maplewood	NJ	\$45,000	1970-09-12	
Duplicates	Joseph Martinez	Portland	OR	\$120,000	1960-12-10	
	Joseph Martinez	Portland	OR	\$120,000	1960-12-10	
	Wayne Nielson	Dahlgren	VA	\$75,000	1968-12-05	
	Beverly Nixon	Long Island	NY	\$45,000,000	1962-03-05	Outliers
	Veronica Comerford	Tuskegee	AL	\$50,000	1993-09-23	
Inconsistent text & typos	Ivan Layton	Albany	New York	\$105,000	1996-03-12	
	Laura Hailey	Ephraim	Utah	\$62,000	1994-01-13	
	John Depaul	Baton Rouge	LA	\$60,000	1967-08-27	
	James Weiss	Houston	TX			
	Addie Stevenson	Jacksonville		\$80,000		Missing data
	Daniel Long					
	Queen Watson	Los Angeles	CA	\$110,000	1968-05-19	

MISSING DATA

Terdapat berbagai cara untuk merepresentasikan data yang hilang dalam Python.

- **np.NaN** - NaN dari NumPy adalah representasi yang paling umum (nilai disimpan sebagai float)
- **pd.NA** - NA dari Pandas adalah tipe data yang lebih baru untuk data yang hilang (nilai dapat disimpan sebagai integer)
- **None** - Default tipe data yang hilang dari Python dasar (tidak mendukung perhitungan numerik)

```
df = pd.read_csv("customers.csv")  
df.tail()
```

	Name	City	State	Income	Age
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0

← Missing values are treated as *np.NaN* when data is read into Pandas

IDENTIFYING MISSING DATA

Cara paling mudah untuk mengidentifikasi data yang hilang adalah dengan menggunakan metode `.isna()`.

- Juga dapat menggunakan `.info()` atau `.value_counts(dropna=False)`.

```
df.isna()
```

	Name	City	State	Income	Age
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
5	False	False	False	False	False
6	False	False	False	False	False
7	False	False	False	False	False
8	False	False	False	False	False
9	False	False	False	True	True
10	False	False	True	False	True
11	False	True	True	True	True
12	False	False	False	False	False



```
df.isna().sum()
```

```
Name      0
City       1
State      2
Income     2
Age        3
dtype: int64
```

Use `sum()` to return the missing values by column



```
df[df.isna().any(axis=1)]
```

	Name	City	State	Income	Age
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN

Or use `any(axis=1)` to select the rows with missing values

- *This returns `True` for any missing values*

HANDLING MISSING DATA

Terdapat beberapa cara untuk handle **Missing Values**:

- Pertahankan data yang hilang seperti apa adanya
- Hapus seluruh baris atau kolom yang memiliki data yang hilang.
- Isi data numerik yang hilang dengan 0 atau dengan nilai pengganti seperti rata-rata, modus, dll.
- Menyelesaikan data yang hilang berdasarkan keahlian domain Anda.

```
df[df.isna().any(axis=1)]
```

	Name	City	State	Income	Age	
9	James Weiss	Houston	TX	NaN	NaN	← Missing age values are likely ok to keep
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN	
11	Daniel Long	NaN	NaN	NaN	NaN	

We have no data on this customer →

← This is likely Jacksonville, FL

← Can we calculate an approximate income?



Tidak ada cara yang benar atau salah dalam menangani data yang hilang, oleh karena itu penting untuk bersikap bijaksana dalam mengatasi masalah ini.

IMPUTING MISSING DATA

Metode **.fillna()** mengisi data yang hilang dengan nilai yang sesuai.

- Terdapat banyak cara untuk mengisi data dalam suatu kolom (nol, rata-rata, modus, dll.), jadi luangkan waktu sejenak untuk memutuskan yang terbaik - jika Anda tidak yakin, mintalah pendapat dari seorang ahli dalam bidang tersebut.

```
df[["Income"]]
```

	Income
0	45000.0
1	120000.0
2	120000.0
3	75000.0
4	45000000.0
5	50000.0
6	105000.0
7	62000.0
8	60000.0
9	NaN
10	80000.0
11	110000.0

← What value
can
be used here?



```
df["Income"] = df["Income"].fillna(df["Income"].median())  
df
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0

Using the median
removes the impact
of the outlier

DUPLICATE DATA

Data duplikat mewakili adanya satu atau lebih baris yang redundan yang mengandung informasi yang sama dengan yang lain, dan oleh karena itu dapat dihapus.

df

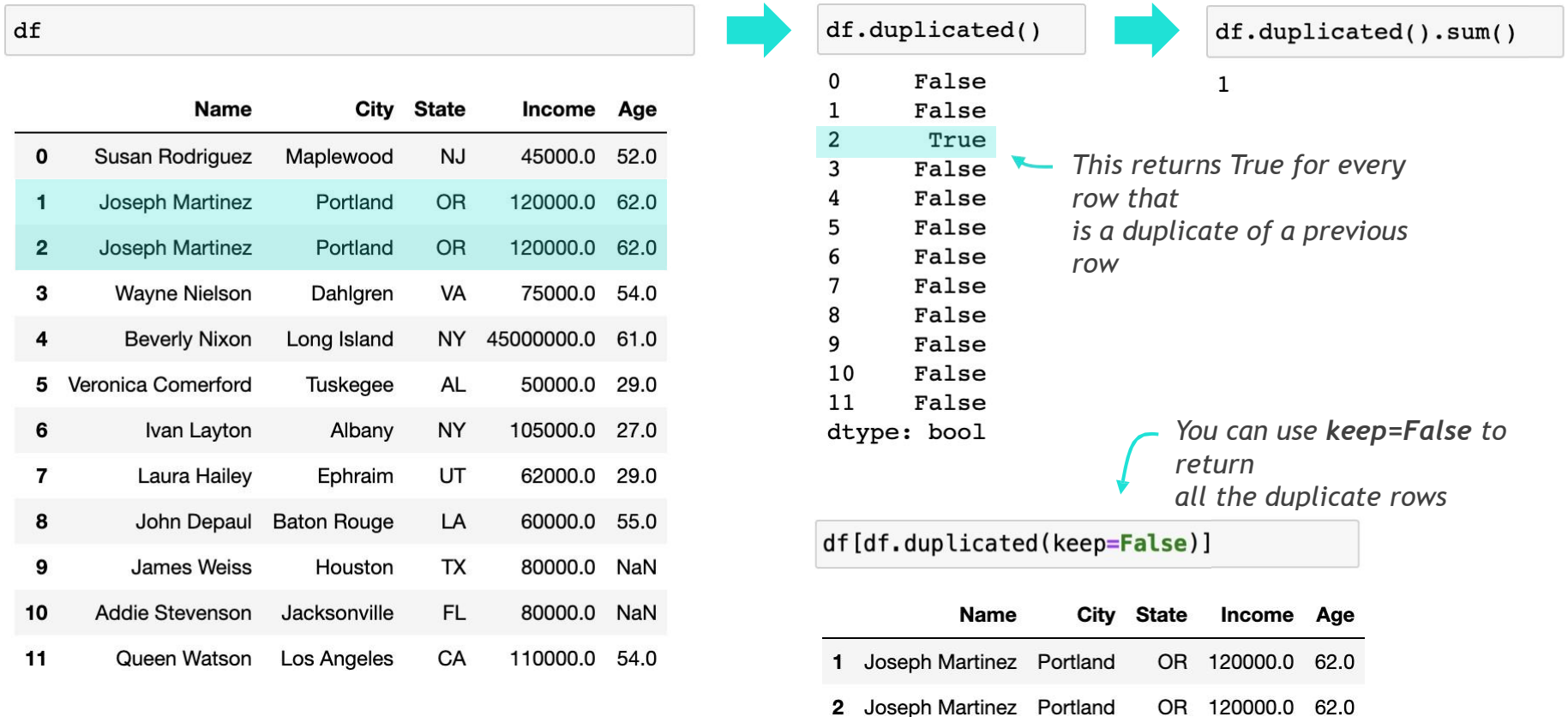
	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	NY	105000.0	27.0
7	Laura Hailey	Ephraim	UT	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0

This is duplicate data on the same customer

These are the same values, but they're not considered duplicate data

IDENTIFYING DUPLICATE DATA

Gunakan **.duplicated()** untuk mengidentifikasi data duplicate



REMOVING DUPLICATE DATA

Gunakan **.drop_duplicates()** untuk menghilangkan data yang duplicate

```
df.drop_duplicates()
```

*You may need
to reset the
index*

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	NY	105000.0	27.0
7	Laura Hailey	Ephraim	UT	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0

OUTLIERS

Outlier adalah nilai dalam kumpulan data yang jauh lebih besar atau lebih kecil daripada nilai lainnya.

df

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Wayne Nielson	Dahlgren	VA	75000.0	54.0
3	Beverly Nixon	Long Island	NY	45000000.0	61.0
4	Veronica Comerford	Tuskegee	AL	50000.0	29.0
5	Ivan Layton	Albany	NY	105000.0	27.0
6	Laura Hailey	Ephraim	UT	62000.0	29.0
7	John Depaul	Baton Rouge	LA	60000.0	55.0
8	James Weiss	Houston	TX	80000.0	NaN
9	Addie Stevenson	Jacksonville	FL	80000.0	NaN
10	Queen Watson	Los Angeles	CA	110000.0	54.0

Average income (including outlier) = \$4.1M

Average income (excluding outlier) = \$82K



Jika outlier tidak diidentifikasi dan ditangani, mereka dapat memiliki dampak yang signifikan pada perhitungan dan model-model.

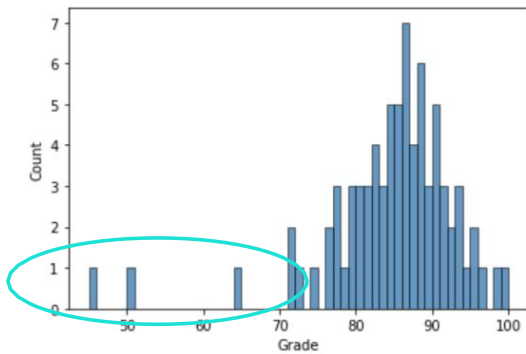
IDENTIFYING OUTLIERS

Mengidentifikasi outlier dengan berbagai cara menggunakan plot dan statistik.

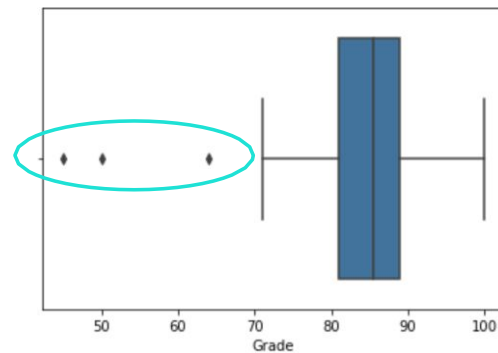
EXAMPLE

Identifying outliers in student grades from a college class

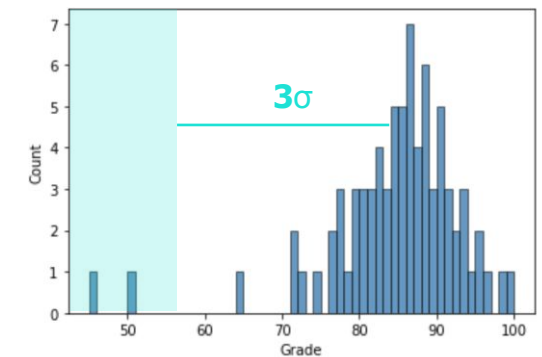
Histogram



Box Plot



Standard Deviation



Sangat penting juga untuk mendefinisikan apa yang akan dianggap sebagai outlier dalam setiap skenario.



Should we flag 3 or 2 outliers?

Use your domain expertise!

HANDLING OUTLIERS

Sama seperti dengan data yang hilang, terdapat beberapa cara untuk menangani outlier:

- Keep outliers
- Menghapus seluruh baris atau kolom yang memiliki outlier.
- Mengganti outlier dengan NaN atau nilai pengganti seperti rata-rata, modus, nilai maksimum, dll.
- Menyelesaikan outlier berdasarkan keahlian domain Anda.

```
df.head()
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Wayne Nielson	Dahlgren	VA	75000.0	54.0
3	Beverly Nixon	Long Island	NY	45000000.0	61.0
4	Veronica Comerford	Tuskegee	AL	50000.0	29.0



How would you handle this?



Thanks

