Basics of Python Programming Language

Outline:

- Introduction to Python
- Introduction to Google Colab & Jupyter Notebook for Writing and Executing Python Codes.
- Getting used to the User Interface (UI) of Jupyter Notebook/Google Colab.
- Python Data Types: Numeric, Strings, Boolean, List, Dictionary, Tuple, Set, None.
- Iterables & Mutability; List vs. Tuples.

1. Introduction to Python:

What is Python?

- Python is a high-level, interpreted, and general-purpose programming language.
- Known for its simplicity and readability, making it ideal for beginners.
- Widely used in Data Science, Machine Learning, Web Development, Automation, and more.

Why Python for Data Science and Machine Learning?

- Rich ecosystem of libraries (e.g., NumPy, Pandas, scipy, matplotlib, plotly, statsmodels, Scikit-learn, TensorFlow).
- Easy to learn and use, with a large community for support.
- Great for prototyping and production-level code.

2. Introduction to Google Colab & Jupyter Notebook

What are Google Colab and Jupyter Notebook?

Google Colab: A free, cloud-based platform for writing and executing Python code in a notebook environment.

Jupyter Notebook: An open-source web application for creating and sharing documents with live code, equations, and visualizations.

Why use them?

- Interactive Coding Environment.
- Easy to Share and Collaboration.
- No Setup Required (especially for Google Colab).

Getting Started with Google Colab:

- Open Google Colab.
- Create a new notebook.
- Familiarize yourself with the interface:
- Cells for code or text.
- Run cells using Shift + Enter.
- Add new cells using the + button.

3. Python Data Types

Python has several built-in data types. Let's explore them with examples:

1. Numeric Types:

- Integers (int): Whole numbers (e.g., 5, -3).
- Floats (float): Decimal numbers (e.g., 3.14, -0.001).

In [ ]:
```
# Example: Numeric Types
age = 25          # int
height = 5.9      # float
print("Age:", age, "Type:", type(age))
print("Height:", height, "Type:", type(height))
```
```
Age: 25 Type: <class 'int'>
Height: 5.9 Type: <class 'float'>
```

Real-life Use Case:

Storing age, height, temperature, or any measurable quantity.

2. Strings (str):

- A sequence of characters enclosed in single or double quotes.
- Strings are immutable (cannot be changed after creation).

In [3]:
```
# Example: Strings

name = "Rossi"
greeting = 'Hello, ' + name
print(greeting)
print("Length of name:", len(name))
```
```
Hello, Rossi
Length of name: 5
```

Real-life Use Case:

Storing names, addresses, or text data.

3. Boolean (bool):

- Represents True or False.
- Used in conditions and logical operations.

In [4]:
```
# Example: Boolean

is_student = True
is_working = False
print("Is student?", is_student)
print("Is working?", is_working)
```
```
Is student? True
Is working? False
```

In [9]:
```
age = 18
if age > 60: #If Evaluated True then the first part will be printed.
    print("You are a senior citizen")
elif age > 18: #If Evaluated True then the second part will be printed.
    print("You are an adult")
else:
    print("You are not an adult")
```
```
You are not an adult
```

Real-life Use Case:

Checking conditions (e.g., "Is the user logged in?").

4. Lists (list):

- Ordered, mutable collection of items.
- Can contain mixed data types.

In [11]:
```
# Example: Lists

fruits = ["apple", "banana", "cherry"]
fruits.append("orange")  # Add an item using the "append" Method
print("Fruits:", fruits)
print("First fruit:", fruits[1])  # Access by index
```
```
Fruits: ['apple', 'banana', 'cherry', 'orange']
First fruit: banana
```

NB: Python is Zero Indexed; Starts from O

Real-life Use Case:

Storing a list of items, such as shopping lists or to-do tasks.

5. Tuples (tuple):

- Ordered, immutable collection of items.
- Faster than lists for fixed data.

In [ ]:
```
# Example: Tuples

coordinates = (10.0, 20.0)
print("Coordinates:", coordinates)
print("X-coordinate:", coordinates[0])
```
```
Coordinates: (10.0, 20.0)
X-coordinate: 10.0
```

Real-life Use Case:

Storing fixed data, such as coordinates or RGB values.

6. Dictionaries (dict):

- Unordered collection of key-value pairs.
- Keys must be unique and immutable.

In [12]:
```
# prompt: Create a dictionary with Keys: Customer Name, Number of Items Purchased, Age
# Add Multiple values under each key

customer_data = {
    'Customer Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Number of Items Purchased': [5, 2, 8, 3],
    'Age': [25, 30, 22, 40]
}
```

In [4]:
```
# prompt: Convert the customer_data list as dataframe

import pandas as pd

customer_data = {
    'Customer Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Number of Items Purchased': [5, 2, 8, 3],
    'Age': [25, 30, 22, 40]
}

customer_df = pd.DataFrame(customer_data)
customer_df
```

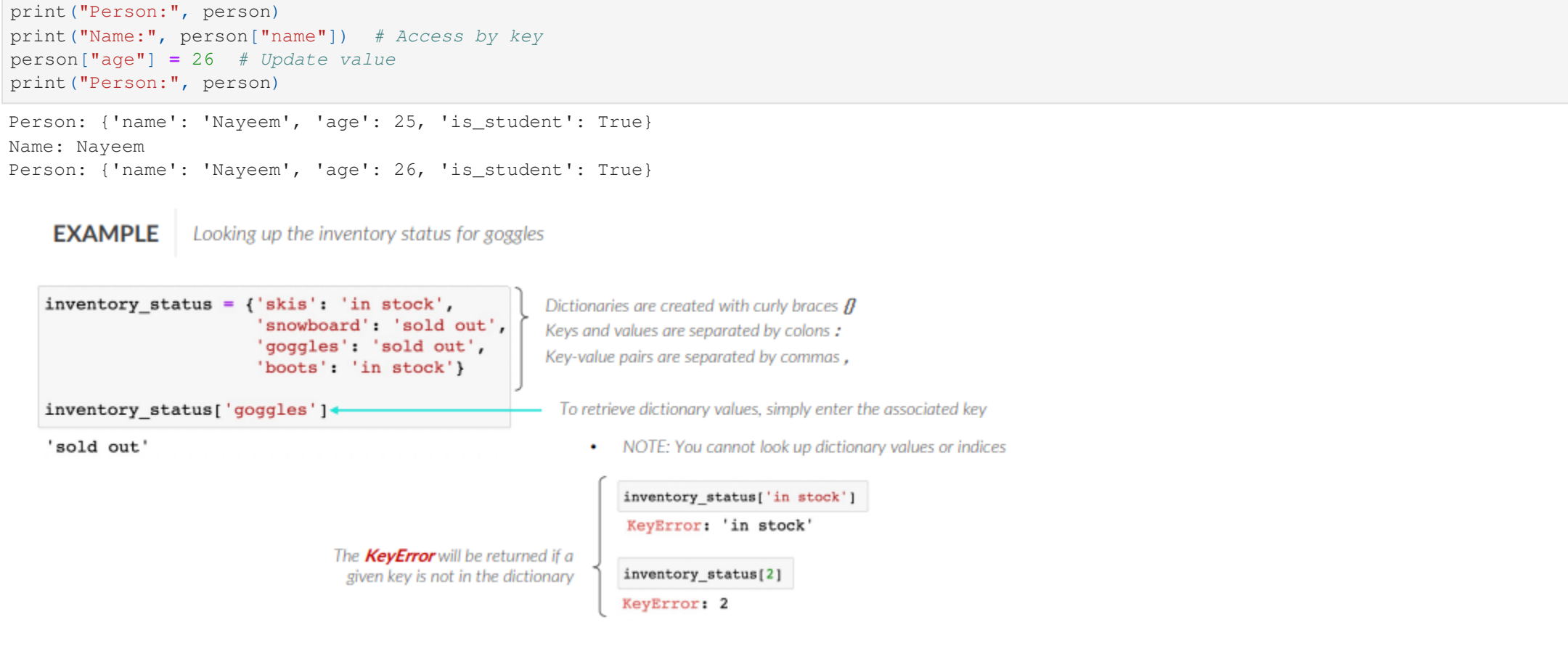
Out[4]:

	Customer Name	Number of Items Purchased	Age
0	Alice	5	25
1	Bob	2	30
2	Charlie	8	22
3	David	3	40

In [5]:
```
customer_df['Age']
```
Out[5]:
```
0    25
1    30
2    22
3    40
Name: Age, dtype: int64
```

In [14]:
```
# Example: Dictionaries

person = {"name": "Nayeem",
          "age": 25,
          "is_student": True}
print("Person:", person)
print("Name:", person["name"])  # Access by key
person["age"] = 26  # Update value
print("Person:", person)
```
```
Person: {'name': 'Nayeem', 'age': 25, 'is_student': True}
Name: Nayeem
Person: {'name': 'Nayeem', 'age': 26, 'is_student': True}
```


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
inventory_status['in stock']
KeyError: 'in stock'
inventory_status[2]
KeyError: 2
The KeyError will be returned if a given key is not in the dictionary

Real-life Use Case:

Storing structured data, such as user profiles or configurations.

In [ ]:
```
#Converting the Dictionary into a Dataframe
import pandas as pd

pd.DataFrame([person])
```

Out[ ]:

	name	age	is_student
0	Alice	26	True

Dictionary values can be lists; so, to access individual attributes:

1. Retriev the **list** by looking up its **key**.
2. Retrieve the **list** element by using its **index**.

In [16]:
```
# Looking up the stock quantity for skis

item_details = {'skis': [250, 12, 'in stock'],
                'snowboard': [220, 0, 'sold out'],
                'goggles': [100, 0, 'sold out'],
                'boots': [80, 5, 'in stock']}
item_details
```
Out[16]:
```
{'skis': [250, 12, 'in stock'],
 'snowboard': [220, 0, 'sold out'],
 'goggles': [100, 0, 'sold out'],
 'boots': [80, 5, 'in stock']}
```

In [ ]:
```
item_details['skis']
```
Out[ ]:
```
[250, 12, 'in stock']
```

In [17]:
```
item_details['skis'][1]
```
Out[17]:
```
12
```

In [ ]:
```
pd.DataFrame(item_details)
```

Out[ ]:

	skis	snowboard	goggles	boots
0	250	220	100	80
1	12	0	0	5
2	in stock	sold out	sold out	in stock

In [ ]:
```
# Membership Tests on Dictionary Keys

print('skis' in item_details)
print('stick' in item_details)
```
```
True
False
```

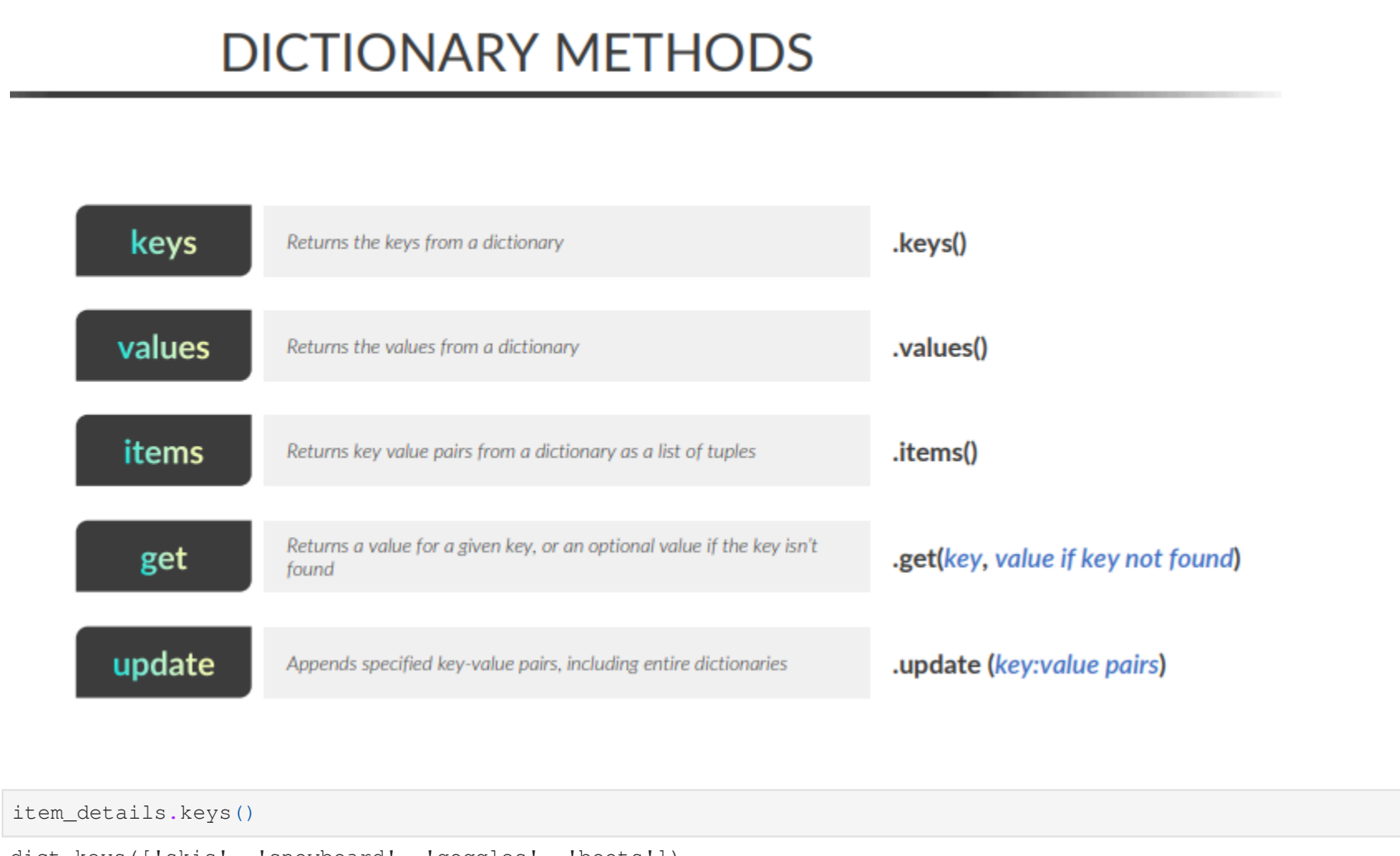
In [ ]:
```
# Modifying Dictionary

item_details['stick'] = [120, 8, 'in stock']
item_details
```
Out[ ]:
```
{'skis': [250, 12, 'in stock'],
 'snowboard': [220, 0, 'sold out'],
 'goggles': [100, 0, 'sold out'],
 'boots': [80, 5, 'in stock'],
 'stick': [120, 8, 'in stock']}
```

In [18]:
```
item_details['stick'] = [120, 0, 'sold out']
item_details
```
Out[18]:
```
{'skis': [250, 12, 'in stock'],
 'snowboard': [220, 0, 'sold out'],
 'goggles': [100, 0, 'sold out'],
 'boots': [80, 5, 'in stock'],
 'stick': [120, 0, 'sold out']}
```

In [19]:
```
# Deleting Keys from Dictionary

del item_details['stick']
item_details
```
Out[19]:
```
{'skis': [250, 12, 'in stock'],
 'snowboard': [220, 0, 'sold out'],
 'goggles': [100, 0, 'sold out'],
 'boots': [80, 5, 'in stock']}
```


DICTIONARY METHODS
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)

In [ ]:
```
item_details.keys()
```
Out[ ]:
```
dict_keys(['skis', 'snowboard', 'goggles', 'boots'])
```

In [ ]:
```
item_details.values()
```
Out[ ]:
```
dict_values([[250, 12, 'in stock'], [220, 0, 'sold out'], [100, 0, 'sold out'], [80, 5, 'in stock']])
```

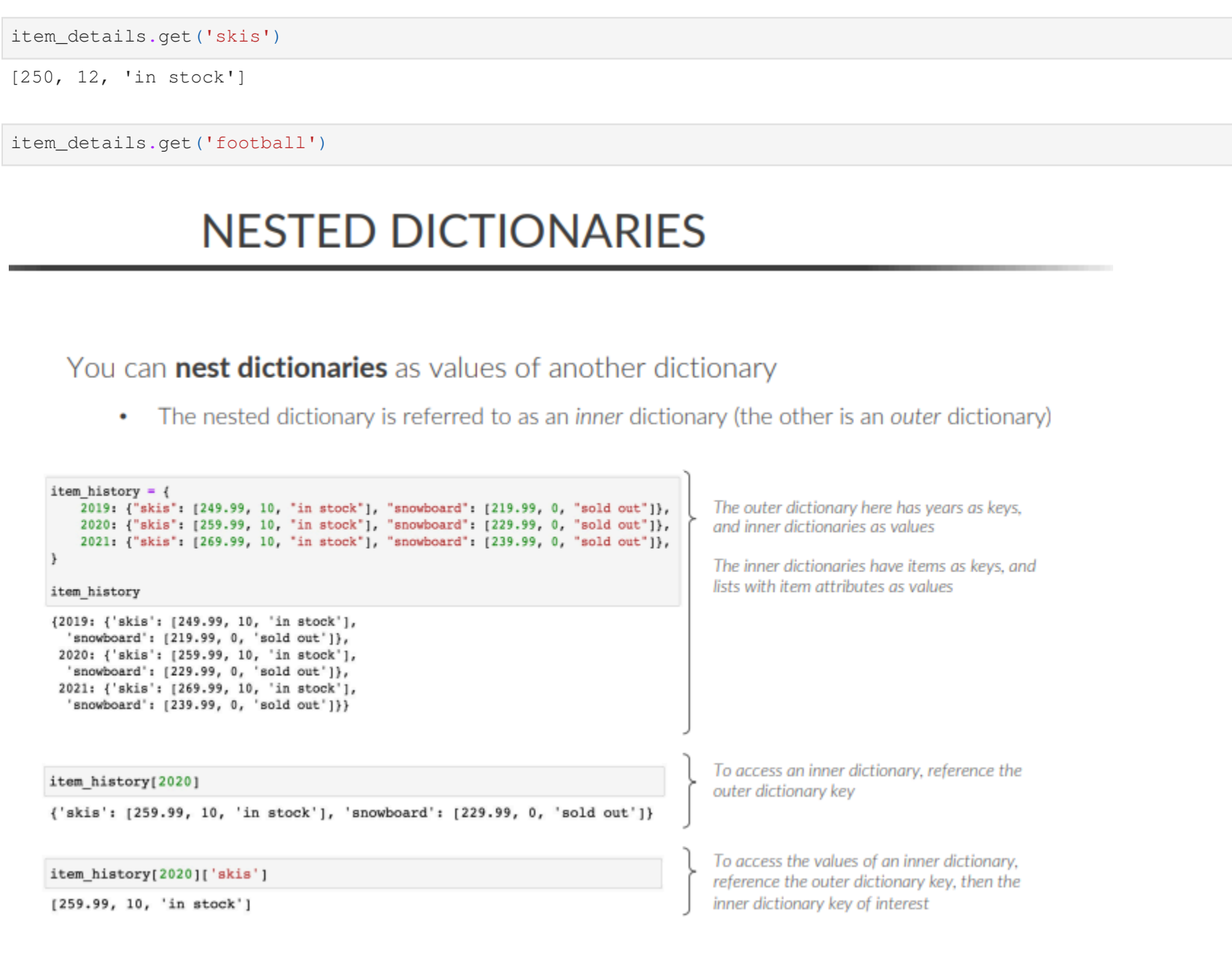
In [ ]:
```
# The .items() method returns key-value pairs from a dictionary as a list of tuples
item_details.items()
```
Out[ ]:
```
dict_items([('skis', [250, 12, 'in stock']), ('snowboard', [220, 0, 'sold out']), ('goggles', [100, 0, 'sold out']), ('boots', [80, 5, 'in stoc
k'])])
```

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.

In [ ]:
```
item_details.get('skis')
```
Out[ ]:
```
[250, 12, 'in stock']
```

In [20]:
```
item_details.get('football')
```


NESTED DICTIONARIES
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)
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

7. Sets (set):

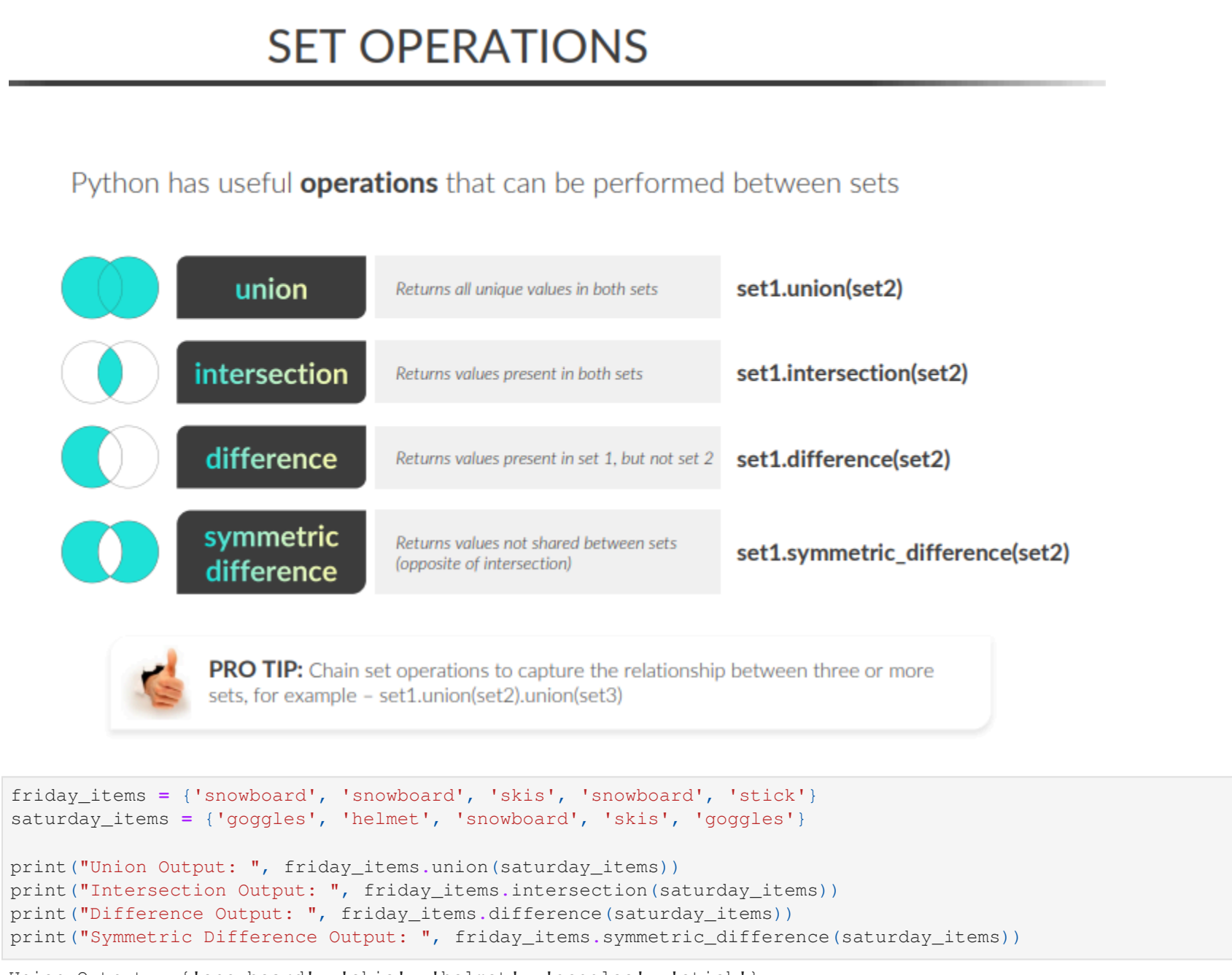
- Unordered collection of **unique items**, which means their values cannot be accessed via index or key.
- Sets are mutable (values can be added/removed), but set values must be unique & immutable
- Useful for operations like union, intersection, and difference.

In [21]:
```
# Example: Sets

unique_numbers = {1, 2, 3, 3, 4}  # Duplicates are removed
print("Unique Numbers:", unique_numbers)
```
```
Unique Numbers: {1, 2, 3, 4}
```

Real-life Use Case:

Removing duplicates from a list or checking membership.


SET OPERATIONS
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)

In [ ]:
```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'stick'}
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}

print("Union Output: ", friday_items.union(saturday_items))
print("Intersection Output: ", friday_items.intersection(saturday_items))
print("Difference Output: ", friday_items.difference(saturday_items))
print("Symmetric Difference Output: ", friday_items.symmetric_difference(saturday_items))
```
```
Union Output:  {'snowboard', 'skis', 'helmet', 'goggles', 'stick'}
Intersection Output:  {'snowboard', 'skis'}
Difference Output:  {'stick'}
Symmetric Difference Output:  {'helmet', 'goggles', 'stick'}
```

8. None (NoneType):

- Represents the absence of a value.
- Often used as a placeholder.

In [ ]:
```
# Example: None

result = None
print("Result:", result)
```
```
Result: None
```

Real-life Use Case:

Initializing a variable before assigning a value.

4. Iterables and 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: Whether an object can be changed after creation.

- **Mutable:** Lists, dictionaries, sets.
- **Immutable:** Strings, tuples, integers, floats.

In [22]:
```
# Example: Mutability

# List (Mutable)
colors = ["red", "green", "blue"]
colors[0] = "yellow"
print("Updated Colors:", colors)
```
```
Updated Colors: ['yellow', 'green', 'blue']
```

In [24]:
```
# Tuple (Immutable)
rgb = ("red", "green", "blue")
rgb[0] = "yellow"  # This will raise an error
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-8ecafad6bed9> in <cell line: 0>()
      1 # Tuple (Immutable)
      2 rgb = ("red", "green", "blue")
----> 3 rgb[0] = "yellow"  # This will raise an error

TypeError: 'tuple' object does not support item assignment
```

Real-life Use Case:

Use lists for dynamic data and tuples for fixed data.

5. List vs. Tuples:

Feature	List	Tuple
Mutability	Mutable	Immutable
Performance	Slower	Faster
Use Case	Dynamic data	Fixed data
Syntax	`[1, 2, 3]`	`(1, 2, 3)`

6. Exercise:

- Create a list of their favorite foods and print the second item.
- Create a dictionary to store their name, age, and favorite color.
- Convert a list of numbers into a set to remove duplicates.