Nicole Angel B. Lopena

CS3C

Dictionaries  
  
● Creation of New Dictionary

>To create a new dictionary in Python within a Jupyter Notebook, all you need to do is use curly brackets '{}’ and key-value pairs.

**Example code:**

# Creating a new dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Displaying the dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'age': 27, 'city': 'Tokyo'}

● Accessing Items in the Dictionary

> Python gives you the ability to get things from dictionaries by utilizing the item's key and square brackets '[]'.

**Example Code:**

# Dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Accessing items in the dictionary

name = my\_dict["name"]

age = my\_dict["age"]

city = my\_dict["city"]

# Displaying the accessed items

print("Name:", name)

print("Age:", age)

print("City:", city)

# Output:

Name: Nick

Age: 27

City: Tokyo

●It is also possible to obtain entries from a dictionary using the 'get()' method. In the event that the key is absent, this function yields 'None.' Alternatively, you can provide a default value, which will be substituted and yield the identical outcome as the other approach.:

# Accessing items using get() method

name = my\_dict.get("name")

age = my\_dict.get("age")

city = my\_dict.get("city")

# Displaying the accessed items

print("Name:", name)

print("Age:", age)

print("City:", city)

● Change Values in the Dictionary

> To change an item's value in Python, simply assign a new value to the relevant key.

**Example Code:**

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Changing values in the dictionary

my\_dict["age"] = 31

my\_dict["city"] = "Kyoto"

# Displaying the updated dictionary

print(my\_dict)

# Output

{'name': 'Nick', 'age': 31, 'city': 'Kyoto'}

● You may also use the 'update()' function to change many variables at once and provide another dictionary with the updated values (which will provide the same results as the first technique):

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Dictionary with updated values

update\_dict = {

"age": 31,

"city": "Kyoto"

}

# Updating values in the dictionary

my\_dict.update(update\_dict)

# Displaying the updated dictionary

print(my\_dict)

● Loop Through a Dictionary Values

> A for loop in Python can be used to cycle through the values in a dictionary. Here's how to cycle through the values in a dictionary in a Jupyter Notebook:

**Example Code:**

# Dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Looping through dictionary values

for value in my\_dict.values():

print(value)

# Output

Nick

27

Tokyo

This code generates a view object called "my\_dict.values()" that displays a list of all the values in the dictionary. "print(value)" outputs each value that the view object's for loop iterates over.

● You can also cycle through both keys and values simultaneously with the 'items()' function:

# Looping through dictionary keys and values

for key, value in my\_dict.items():

print(key, ":", value)

# Output:

name : Nick

age : 27

city : Tokyo

● Check if Key Exists in the Dictionary

> In Python, you can use the in keyword or the 'get()' method to find out if a key is in a dictionary.

**Example Code:**

Applying the keyword 'in':

# Dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Checking if a key exists using the 'in' keyword

key\_to\_check = "age"

if key\_to\_check in my\_dict:

print(f"The key '{key\_to\_check}' exists in the dictionary.")

else:

print(f"The key '{key\_to\_check}' does not exist in the dictionary.")

# Output:

The key 'age' exists in the dictionary.

Applying the ‘get’ method:

# Dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Checking if a key exists using the 'get()' method

key\_to\_check = "age"

if my\_dict.get(key\_to\_check) is not None:

print(f"The key '{key\_to\_check}' exists in the dictionary.")

else:

print(f"The key '{key\_to\_check}' does not exist in the dictionary.")

# Output:

The key 'age' exists in the dictionary.

● Checking for Dictionary Length

> Python's 'len()' function can be used to find the length of a dictionary.

**Example Code:**

# Dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Checking the length of the dictionary

dict\_length = len(my\_dict)

# Displaying the length of the dictionary

print("Length of the dictionary:", dict\_length)

# Output:

Length of the dictionary: 3

'len(my\_dict)' in our example returns the number of key-value pairs in the dictionary'my\_dict', which is 3.

● Adding Items in the Dictionary

> In Python, adding items to a dictionary is as easy as assigning a value to a new key or an already-existing key.

**Example Code:**

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Adding a new key-value pair

my\_dict["email"] = "john@example.com"

# Displaying the updated dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'age': 27, 'city': 'Tokyo', 'email': 'nick@example.com'}

The dictionary 'my\_dict' now has a new key-value pair ("email", "nick@example.com") added to it.

● Additionally, you can use the 'update()' method to add multiple key-value pairs from a different dictionary:

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Dictionary with additional key-value pairs

additional\_info = {

"email": "nick@example.com",

"phone": "476-951-8320"

}

# Adding multiple key-value pairs to the dictionary using update()

my\_dict.update(additional\_info)

# Displaying the updated dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'age': 27, 'city': 'Tokyo', 'email': 'nick@example.com'}

● Removing Items in the Dictionary

> In In Python, you can remove entries from a dictionary by using the 'pop()' function or the 'del' keyword.

**Example Code:**

Applying the 'pop()' technique:

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Removing an item using the 'pop()' method

removed\_value = my\_dict.pop("age")

# Displaying the updated dictionary and the removed value

print("Updated dictionary:", my\_dict)

print("Removed value:", removed\_value)

# Output:

Updated dictionary: {'name': 'Nick', 'city': 'Tokyo'}

Removed value: 27

The value '27' linked to the key 'age' was obtained in this instance and assigned to the variable'removed\_value'. The 'pop()' method was utilized to extract the key from the dictionary'my\_dict' in order to achieve this.

● Remove an Item Using del Statement

> In Python, you may use the 'pop()' function or the ‘del’ keyword to delete items from a dictionary.

**Example Code:**

Applying the keyword 'del':

# Original dictionary

my\_dict = {

"name": "Nick",

"age": 27,

"city": "Tokyo"

}

# Removing an item using the 'del' keyword

del my\_dict["age"]

# Displaying the updated dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'city': 'Tokyo'}

You can employ any strategy, based on your own needs. The 'pop()' method is useful when all you want to do is remove a key-value pair; however, the 'del' keyword is useful if you also want to retrieve the value that corresponds to the removed key.

● The dict() Constructor

> To create new dictionaries in Python, utilize the 'dict()' constructor that comes with the language. Dictionaries can be produced in many circumstances using various types of arguments. This is how to use the 'dict()' constructor in a Jupyter notebook:

**Example Code:**

1.Creating a dictionary from keyword arguments:

# Using keyword arguments

my\_dict = dict(name="Nick", age=27, city="Tokyo")

# Displaying the dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'age': 27, 'city': 'Tokyo'}

2.Making a dictionary out of a tuple list:

# Using a list of tuples

my\_list = [("name", "Nick"), ("age", 27), ("city", "Tokyo")]

my\_dict = dict(my\_list)

# Displaying the dictionary

print(my\_dict)

#Output

{'name': 'Nick', 'age': 27, 'city': 'Tokyo'}

3.combining two parallel lists to create a dictionary:

# Using two parallel lists

keys = ["name", "age", "city"]

values = ["Nick", 27, "Tokyo"]

my\_dict = dict(zip(keys, values))

# Displaying the dictionary

print(my\_dict)

# Output:

{'name': 'Nick', 'age': 27, 'city': 'Tokyo'}

4.Making a dictionary that is empty:

# Creating an empty dictionary

empty\_dict = dict()

# Displaying the empty dictionary

print(empty\_dict)

# Output:

{}

● Dictionary Methods

> Python dictionaries are versatile data structures with a number of built-in functions for performing various tasks. Here are several well-liked dictionary strategies:

**Code Example:**

# Sample dictionary

my\_dict = {'name': 'Bonnie', 'age': 30, 'city': 'New York'}

# Example usage of dictionary methods

print("Original dictionary:", my\_dict)

#Output

Original dictionary: {'name': ' Bonnie', 'age': 30, 'city': 'New York'}

1.'clear()': Empties the dictionary of all its contents.

# Using clear()

my\_dict.clear()

print("After clear():", my\_dict)

#Output

After clear(): {}

2.'copy()': Provides a cursory duplicate of the dictionary.

# Using copy()

my\_dict = {'name': ' Bonnie', 'age': 30, 'city': 'New York'}

copy\_dict = my\_dict.copy()

print("Copy of the dictionary:", copy\_dict)

# Output:

Copy of the dictionary: {'name': ' Bonnie', 'age': 30, 'city': 'New York'}

3."get(key[, default])": Provides the value associated with the given key. It returns None if the key is not supplied, or the default value if the key cannot be found.

# Using get(key[, default])

print("Value for key 'name':", my\_dict.get('name'))

print("Value for key 'gender':", my\_dict.get('gender', 'Not found'))

#Output

Value for key 'name': Bonnie

Value for key 'gender': Not found

4.'items()': Provides a view object with a list of tuples with a key-value pair for each tuple.

# Using items()

print("Items in the dictionary:", my\_dict.items())

#Output

Items in the dictionary: dict\_items([('name', ' Bonnie'), ('age', 30), ('city', 'New York')])

5.'keys()': Provides a view object with a list of all the dictionary's keys shown.

# Using keys()

print("Keys in the dictionary:", my\_dict.keys())

#Output

Keys in the dictionary: dict\_keys(['name', 'age', 'city'])

6.‘pop(key[, default])’: Returns the value of the item that was removed using the given key. In the event that the key cannot be located, it either issues a Key Error or returns the default value.

# Using pop(key[, default])

print("Removed item:", my\_dict.pop('age'))

print("Dictionary after pop('age'):", my\_dict)

#Output

Removed item: 30

7.‘popitem()’ removes and gives back a tuple containing any given key-value pair. In case the dictionary is empty, raise a KeyError.

# Using popitem()

print("Removed item:", my\_dict.popitem())

print("Dictionary after popitem():", my\_dict)

#Output

Removed item: ('city', 'New York')

Dictionary after pop('age'): {'name': ' Bonnie', 'city': 'New York'}

Dictionary after popitem(): {'name': ' Bonnie'}

8.‘update(iterable)’: Imparts key-value pairs from an iterable object or another dictionary into the dictionary.

# Using update(iterable)

my\_dict.update({'country': 'USA'})

print("Updated dictionary:", my\_dict)

#Output

Updated dictionary: {'name': ' Bonnie', 'country': 'USA'}

9.‘values()’: Provides a view object with a list of all the dictionary's values shown.

# Using values

print("Values in the dictionary:", my\_dict.values())

#Output

Values in the dictionary: dict\_values(['Bonnie', 'USA'])

**Jupyter notebook**

● Adding Folder

> Typically, adding a folder to Jupyter Notebook is done through your operating system's file management system. You can find it by creating a new folder in the location of your choice, which you can access from within the Jupyter Notebook. The steps are as follows:

1. Launch Jupyter Notebook: To launch your Jupyter Notebook server, use your terminal or command line to run the appropriate command. Once the server is up and running, open it in your web browser.

2. Navigate to the appropriate directory: Use the Jupyter Notebook file browser interface to navigate to the directory in order to create the new folder.

3. Make the Folder: After you are in the right directory, look for a button or option to create a new folder. This option is typically indicated by the word "New" or a plus sign next to a folder icon.

4. Give the Folder a Name: Upon asking, give the new folder a name. Give the folder a name that appropriately conveys its purpose or contents.

5. Check Creation: After the folder name, check the process of creation. The directory listing in Jupyter Notebook should now display the new folder.

6. Check the Folder: To make sure the folder was created correctly, navigate to its location within the directory structure. Along with any other existing files or folders, the newly created folder should be visible.

7. Use the Folder: Using the previously created folder, you may now organize your files and data inside the Jupyter Notebook. You can add files, make subdirectories, and perform other file management tasks as needed.

● Adding Text file

> Similarly, you can add a text file using your operating system's file management system. Once the text file is in the proper location, you can read its contents or perform any operations on it within the Jupyter Notebook.

1. Start Jupyter Notebook: To access Jupyter Notebook, launch the Jupyter Notebook server and open it in your web browser.

2. Go to the Desired Directory: Use the file browser interface in Jupyter Notebook to find the directory where you want to create a new text file.

3. Create the Text File: Locate a button or option to start a new file. Typically, a file symbol with a plus sign or the term "New" are used to indicate this choice.

4. Click "Create New Text File" after selecting "Text File." The name of this may be "Text File," "Notebook," or something else entirely, depending on how your Jupyter Notebook is laid up.

5. Give the Text File a Name: Upon request, provide a name for the newly created text file.

6. Open the Text File: After the text file has been generated, click on its name in the file browser interface to open it. This will open the text file in a new tab within your Jupyter Notebook environment.

7. Edit the Text File: You can now directly edit the contents of a text file in Jupyter Notebook. Use the provided text editor interface to type or paste text.

8.Save the Text File: When you're done altering the text file, don't forget to save your changes. To save the file, you may need to click the "Save" button on your Jupyter Notebook interface or use a keyboard shortcut (usually Ctrl + S or Command + S on Mac).

9. Check the Text File: To make sure the text file was created and saved appropriately, navigate to its position within the directory structure. Along with any existing files or directories, the newly created text file should be displayed.

10. Use the Text File: The Jupyter Notebook environment now allows you to store notes, code snippets, and documentation in a text file.

● CSV file for data analysis and visualization  
> CSV (Comma Separated Values) files are widely used for data processing and display. To read and work with the data in CSV files, one can utilize Python libraries like Pandas. For visualization, libraries like Seaborn and Matplotlib can be utilized.

1. Obtain a CSV File: You can create your own or download CSV files from a number of sources, such as online archives and government databases. Ensure the CSV file has the structured data you want to examine and display.

2. Upload CSV File to Jupyter Notebook: Use the file browser interface in Jupyter Notebook to upload the CSV file to your current working directory. Jupyter Notebook usually offers a "Upload" button or file import option.

3. Import Libraries: You must import the necessary Python libraries in order to analyze and visualize data. Pandas is a popular library for manipulating data; Matplotlib or Seaborn are good choices for visualization.

4. Download the CSV Data: Use Pandas to import the CSV data into a DataFrame, a tabular data structure. You can use the'read\_csv()' function to accomplish this.

import pandas as pd

# Load CSV data into a DataFrame

df = pd.read\_csv('your\_file.csv')

5. Examine the Data: Before starting any analysis or visualization, it's critical to understand the structure and contents of the data. Make use of the "info()," "describe()," and "head()" functions to explore the DataFrame.

# Display the first few rows of the DataFrame

print(df.head())

# Get information about the DataFrame

print(df.info())

# Generate descriptive statistics

print(df.describe())

6. Analyze the Data: After you have a firm understanding of the information, you can start to make inferences from it. Use Pandas techniques to filter, combine, or alter the data to meet your analytical goals.

# Example: Calculate mean of a numerical column

mean\_value = df['column\_name'].mean()

7. Visualize the Data: Create plots and charts using Matplotlib, Seaborn, or other visualization tools to communicate insights and display the data in an efficient manner.

import matplotlib.pyplot as plt

import seaborn as sns

# Example: Create a histogram

plt.hist(df['numerical\_column'])

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.title('Histogram of Numerical Column')

plt.show()

8. Iterate and refine: Modify your approach in light of your results and the problems you intend to solve, making the required changes to your analysis and visualization process.

● Import libraries

>Libraries can be imported into Jupyter Notebook using the 'import' statement in Python. The following Jupyter Notebook code snippets can be used to import libraries that are frequently used for Python data analysis and visualization.:

# Importing Pandas for data manipulation

import pandas as pd

# Importing Matplotlib for basic plotting

import matplotlib.pyplot as plt

# Importing Seaborn for statistical data visualization

import seaborn as sns

# Importing NumPy for numerical computations

import numpy as np

# Importing SciPy for scientific computing and statistics

import scipy.stats as stats

● Finding data

> In Jupyter Notebook, you may find and import data directly into your notebook environment using a number of different methods. These are a few common techniques.:

●Using Online Repositories: Among other online repositories, you can obtain datasets from GitHub, Kaggle, and the UCI Machine Learning Repository. Use the file upload function to upload the files to your Jupyter Notebook environment after they have been downloaded.

● Making use of APIs: APIs, or application programming interfaces, offer access to certain datasets. Use libraries such as'requests' to make HTTP requests and receive data directly into your notebook from these APIs.

● Using Built-in Datasets: Some Python packages come with built-in datasets that can be used for experimentation and learning. Pre-built datasets from apps like "scikit-learn" and "seaborn" can be loaded directly into your notebook.

● Using Web Scraping: If the data is accessible online, you may use web scraping methods with libraries like BeautifulSoup or Scrapy to extract and import it into your notebook.

● Accessing Local Files: From your notebook environment, you can immediately access data that is stored locally on your computer by using file paths.

import a dataset from the preset scikit-learn datasets:

from sklearn.datasets import load\_iris

# Load the Iris dataset

iris = load\_iris()

# Access data and target

X = iris.data # Features

y = iris.target # Target variable

import a dataset from the preset scikit-learn datasets:

import seaborn as sns

# Load the 'iris' dataset from seaborn

iris\_df = sns.load\_dataset('iris')

# Display the first few rows of the dataset

print(iris\_df.head())

● Importing data

> Depending on the structure and data source, a Jupyter Notebook can be filled in in a number of ways. These are a few common techniques:

● From a Local File: Pandas may read information stored locally on your computer in formats such as CSV, Excel, JSON, and more.

import pandas as pd

# Load data from a CSV file

df = pd.read\_csv('file.csv')

# Load data from an Excel file

df = pd.read\_excel('file.xlsx')

# Load data from a JSON file

df = pd.read\_json('file.json')

●From a URL: If the data is hosted online, Pandas can be used to read it directly from a URL.

import pandas as pd

# Load data from a URL

url = 'https://example.com/data.csv'

df = pd.read\_csv(url)

● Using APIs: If the data is accessible through an API, you may use libraries like "requests" to retrieve and parse it into a Pandas DataFrame.

import requests

import pandas as pd

# Fetch data from an API

response = requests.get('https://api.example.com/data')

data = response.json()

# Convert data to DataFrame

df = pd.DataFrame(data)

● Using Pre-Made Datasets: Some Python programs allow you to import pre-made datasets directly into your notebook.

from sklearn.datasets import load\_iris

# Load the Iris dataset

iris = load\_iris()

# Access data and target

X = iris.data # Features

y = iris.target # Target variable

● Using Web Scraping: If the data is accessible online, you may use web scraping methods with libraries like BeautifulSoup or Scrapy to extract and import it into your notebook.

import pandas as pd

import requests

from bs4 import BeautifulSoup

# Fetch HTML content

url = 'https://example.com/data'

response = requests.get(url)

soup = BeautifulSoup(response.content, 'html.parser')

# Convert data to DataFrame

df = pd.DataFrame(...)

● Data attributes

> Data attributes are the characteristics or features of a dataset. These characteristics provide information about the data's contents, arrangement, and metadata. The following are common attributes of data in a typical dataset:

● Dimensions: The number of rows and columns in the dataset indicates its size. Common ways to show this data are as a tuple (rows, columns) or as discrete properties indicating the number of rows (instances) and columns (features).

● Column Names: Also known as feature names, column names are the labels assigned to each column in the dataset. These labels provide context and indicate the type of data that is present in each column. They are often stored as a list of strings or an array.

● Missing values are the absence of data in certain cells or entries in a dataset. Accurately identifying and handling missing values during data preparation is crucial to avoid biased analysis and issues with model performance.

● Summary Statistics: These provide descriptive information about the dataset, including the mean, median, standard deviation, minimum and maximum values for numerical columns, and other information. The distribution and central tendency of the data are revealed by these statistics.

● Unique Values: The dataset's distinct values, which may be found in each column, are denoted by unique values. Identifying the unique values can help with data quality assessment, variable classification, and anomaly detection.

● Data Range: The data range describes the values found in each column, including the minimum and maximum values. It makes determining the data's range and locating anomalies or outliers easier.

● The metadata contains additional information about the dataset, such as the date and source of the data, the methods used to acquire the data, and any relevant annotations or comments. Metadata can help make sense of the data and put it in context.

Python users can access and explore these capabilities by using a variety of functions and methods provided by data manipulation libraries such as Pandas. For example:

import pandas as pd

# Assuming 'df' is your DataFrame

# Dimensions

print("Dimensions:", df.shape)

# Column names

print("Column Names:", df.columns)

# Data types

print("Data Types:", df.dtypes)

# Missing values

print("Missing Values:", df.isnull().sum())

# Summary statistics

print("Summary Statistics:", df.describe())

# Unique values

print("Unique Values:", df.nunique())

# Data range

print("Data Range:", df.min(), df.max())