CS PROFESSIONAL ELECTIVE 2 | FINALS: ASSIGNMENT #4

MATILLA, CARL ANDRIE D. | BSCS 3C

DICTIONARIES

CREATION OF NEW DICTIONARIES

Using curly brackets '{}' and key-value pairs is all it takes to construct a new dictionary in Python inside of a Jupyter Notebook.

Example:

```
# Creating a new dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}
# Displaying the dictionary
print(my_dict)

# Output:
{'name': 'John', 'age': 30, 'city': 'New York'}
```

ACCESSING ITEMS IN DICTIONARIES

Python allows you to get objects from dictionaries by using square brackets '[]' and the item's key.

Example:

```
# Dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# 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: John
Age: 30
City: New York
```

- The '**get()**' function may also be used to retrieve entries from a dictionary. If the key is missing, this function returns 'None.' Alternatively, you may supply a default value that will be returned in its place, giving you the same result as the other method:

```
# 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 DICTIONARIES

In Python, all it takes to modify an item's value is to assign a new value to the appropriate key.

```
# Original dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}
```

```
# Changing values in the dictionary
my_dict["age"] = 35
my_dict["city"] = "San Francisco"

# Displaying the updated dictionary
print(my_dict)

# Output
{'name': 'John', 'age': 35, 'city': 'San Francisco'}
```

- By giving another dictionary with the modified values, you can also use the **'update()'** function to alter many values at once (also give the same output as the first method):

```
# Original dictionary
my_dict = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# Dictionary with updated values
update_dict = {
    "age": 35,
    "city": "San Francisco"
}

# Updating values in the dictionary
my_dict.update(update_dict)

# Displaying the updated dictionary
print(my_dict)
```

LOOP THROUGH A DICTIONARY VALUES

In Python, a for loop may be used to iterate through a dictionary's values. The following describes how to loop over a dictionary's values in a Jupyter Notebook:

```
# Dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# Looping through dictionary values
for value in my_dict.values():
   print(value)

# Output
John
30
New York
```

'my_dict.values()' in this code yields a view object that shows a list of every value in the dictionary. **'print(value)'** prints each value that the for loop iterates over in the view object.

- The **'items()'** function also allows you to cycle through both keys and values at the same time:

```
# Looping through dictionary keys and values
for key, value in my_dict.items():
   print(key, ":", value)

# Output:
name : John
age : 30
city : New York
```

CHECK IF KEY EXISTS IN DICTIONARIES

The **'get()'** function or the in keyword can be used in Python to determine if a key is present in a dictionary.

Applying the keyword 'in':

```
# Dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# 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": "John",
   "age": 30,
   "city": "New York"
}

# 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

The **'len()'** method in Python may be used to determine a dictionary's length.

```
# Dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# 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
```

The number of key-value pairs in the dictionary **'my_dict'** is returned by **'len(my_dict)'** in our example, which is 3.

ADDING ITEMS IN DICTIONARY

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

```
# Original dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# Adding a new key-value pair
my_dict["email"] = "john@example.com"
```

```
# Displaying the updated dictionary
print(my_dict)

# Output:
{'name': 'John', 'age': 30, 'city': 'New York', 'email': 'john@example.com'}
```

The dictionary **'my_dict'** now has a new key-value pair ("email", "john@example.com") added to it.

- Additionally, you may add several key-value pairs from a different dictionary by using the **'update()'** method:

```
# Original dictionary
my_dict = {
   "name": "John",
   "age": 30,
   "city": "New York"
}

# Dictionary with additional key-value pairs
additional_info = {
   "email": "john@example.com",
   "phone": "123-456-7890"
}

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

# Displaying the updated dictionary
print(my_dict)

# Output:
{'name': 'John', 'age': 30, 'city': 'New York', 'email': 'john@example.com'}
```

REMOVING ITEMS IN THE DICTIONARY

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

Applying the 'pop()' technique:

```
# Original dictionary
my_dict = {
    "name": "John",
  "age": 30,
  "city": "New York"
}

# 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': 'John', 'city': 'New York'}
Removed value: 30
```

In this example, the value **'30'** associated with the key **'age'** was returned and placed in the variable **'removed_value'**. This was accomplished by using the **'pop()'** function to remove the key from the dictionary **'my_dict'**.

REMOVE AN ITEM USING A DEL STATEMENT

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

Applying the keyword 'del':

```
# Original dictionary
my_dict = {
  "name": "John",
   "age": 30,
   "city": "New York"
```

```
}

# Removing an item using the 'del' keyword
del my_dict["age"]

# Displaying the updated dictionary
print(my_dict)

# Output:
{'name': 'John', 'city': 'New York'}
```

Either approach can be used, depending on your particular requirements. When all you want to do is delete a key-value combination, the **'pop()'** method works well; however, if you also want to get the value that goes with the removed key, the **'del'** keyword works well.

THE DIC() CONSTRUCTOR

In Python, the built-in 'dict()' constructor is used to generate new dictionaries. Different kinds of arguments can be used to generate dictionaries in different contexts. Using the 'dict()' constructor in a Jupyter Notebook looks like this:

1. **Creating a dictionary from keyword arguments:**

```
# Using keyword arguments
my_dict = dict(name="John", age=30, city="New York")

# Displaying the dictionary
print(my_dict)

# Output:
{'name': 'John', 'age': 30, 'city': 'New York'}
```

2. **Making a dictionary out of a tuple list:**

```
# Using a list of tuples
my_list = [("name", "John"), ("age", 30), ("city", "New York")]
my_dict = dict(my_list)
```

```
# Displaying the dictionary
print(my_dict)

#Output
{'name': 'John', 'age': 30, 'city': 'New York'}
```

3. **combining two parallel lists to create a dictionary:**

```
# Using two parallel lists
keys = ["name", "age", "city"]
values = ["John", 30, "New York"]
my_dict = dict(zip(keys, values))

# Displaying the dictionary
print(my_dict)

# Output:
{'name': 'John', 'age': 30, 'city': 'New York'}
```

4. **Making a dictionary that is empty:**

```
# Creating an empty dictionary
empty_dict = dict()

# Displaying the empty dictionary
print(empty_dict)

# Output:
{}
```

DICTIONARY METHODS

Dictionaries are flexible data structures in Python that include several built-in methods for carrying out different tasks. Here are a few popular dictionary techniques:

```
# Sample dictionary
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Example usage of dictionary methods
print("Original dictionary:", my_dict)
```

```
#Output
Original dictionary: {'name': 'Alice', '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': 'Alice', 'age': 30, 'city': 'New York'}
copy_dict = my_dict.copy()
print("Copy of the dictionary:", copy_dict)
# Output:
Copy of the dictionary: {'name': 'Alice', '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': Alice
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', 'Alice'), ('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 KeyError 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': 'Alice', 'city': 'New York'}
Dictionary after popitem(): {'name': 'Alice'}
```

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': 'Alice', '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(['Alice', 'USA'])
```

JUPYTER NOTEBOOK

ADDING A FOLDER

Usually, you use the file management system of your operating system to add a folder in Jupyter Notebook. To locate it, you may make a new folder in the desired location, which you can then access within the Jupyter Notebook. Here are the steps:

1. **Open Jupyter Notebook:** Use your terminal or command prompt to execute the relevant command to start your Jupyter Notebook server. Open the server in your web browser after it has started operating.
2. **Go to the desired directory by navigating:** To create the new folder, navigate to the directory using Jupyter Notebook's file browser interface.
3. **Make the Folder:** Look for a button or option to create a new folder once you are in the appropriate directory. Usually, this option is denoted by the word "New" or a folder symbol with a plus sign.
4. **Give the Folder a Name:** Give the new folder a name when requested. Select a name for the folder that accurately describes its contents or objective.
5. **Verify Creation:** Following folder name, verify the creation procedure. Now, the new folder ought to show up in Jupyter Notebook's directory listing.
6. **Verify the Folder:** Navigate to the folder's position inside the directory structure to confirm that it was created correctly. The newly formed folder ought to be shown with any other files or folders that are already there.
7. **Use the Folder:** You may now arrange your files and data inside of Jupyter Notebook by using the recently generated folder. As needed, you can add files, create subfolders, and carry out other file management operations.

ADDING A TEXT FILE

In a similar vein, the file management mechanism of your operating system allows you to add a text file. You may read the text file's contents or carry out any actions on it within the Jupyter Notebook once you've placed it in the correct position.

1. **Open Jupyter Notebook:** Launch the Jupyter Notebook server and open it in your web browser to access Jupyter Notebook.
2. **Go to the Desired Directory:** To locate the directory where you wish to create the new text file, use Jupyter Notebook's file browser interface.

3. **Create the Text File:** To create a new file, look for an option or button. Usually, this option is denoted by the word "New" or a file symbol with a plus sign.
4. **Select Text File:** Click on the "Create New Text File" option. Depending on the layout of your Jupyter Notebook, this may be named "Text File," "Notebook," or something else entirely.
5. **Give the Text File a Name:** Give the new text file a name when requested.
6. **Open the Text FIle:** Click on the name of the text file in the file browser interface to open it after it has been generated. This will launch the text file in your Jupyter Notebook environment in a new tab.
7. **Modify the Text File:** Jupyter Notebook now allows you to directly alter the text file's contents. To type or paste text, use the given text editor interface.
8. **Save the Text File:** Remember to save your edits once you've finished editing the text file. You might need to use a keyboard shortcut (typically Ctrl + S or Command + S on Mac) or click a "Save" button on your Jupyter Notebook interface in order to save the file.
9. **Verify the Text File:** Navigate to the text file's location inside the directory structure to confirm that it was produced and saved correctly. The newly formed text file ought to be shown with any other files or directories that are already there.
10. **Use the Text File:** You may now save notes, code snippets, and documentation in the text file within the Jupyter Notebook environment.

CSV FILE FOR DATA ANALYSIS & VISUALIZATION

Files using CSV (Comma Separated Values) are frequently used for data display and analysis. Python packages such as Pandas may be used to read CSV files and manipulate the data they contain. Libraries such as Matplotlib and Seaborn can be used for visualization.

1. **Obtain a CSV File:** You may make your own or acquire CSV files from a variety of sources, including government databases and internet archives. Make sure the structured data you wish to analyze and visualize is included in the CSV file.
2. **Upload the CSV File to Jupyter Notebook:** Upload the CSV file to your current working directory using Jupyter Notebook's file browser interface. Typically, Jupyter Notebook has an "Upload" button or option for importing files.
3. **Import Libraries:** To analyze and visualize data, import the Python libraries that are required. For data manipulation, popular libraries include Pandas; for visualization, consider Matplotlib or Seaborn.

4. **Load the CSV Data:** To load the CSV data into a DataFrame, a tabular data structure, utilize Pandas. To do this, you can utilize the'read_csv()' method.

```
import pandas as pd

# Load CSV data into a DataFrame
df = pd.read_csv('your_file.csv')
```

5. **Examine the Data:** It's important to comprehend the structure and contents of the data before beginning any analysis or visualization. Utilize functions such as "info()," "describe()," and "head()" to investigate 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 gaining a grasp of the information, you may begin to analyze it in order to draw conclusions. To filter, combine, or modify the data in accordance with your analytical objectives, use Pandas techniques.

```
# Example: Calculate mean of a numerical column
mean_value = df['column_name'].mean()
```

7. **Visualize the Data:** To effectively share insights and show the data, construct plots and charts using Matplotlib, Seaborn, or other visualization tools.

```
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:** Make necessary adjustments to your analysis and visualization process, adjusting your strategy in light of your findings and the issues you hope to address.

IMPORT LIBRARIES

Python's 'import' statement may be used to import libraries in Jupyter Notebook. We can use the following code snippets in your Jupyter Notebook to import libraries that are often used for data analysis and visualization in Python:

```
# 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

You may use a variety of techniques in Jupyter Notebook to locate and import data straight into your notebook environment. Here are some typical methods:

- **Utilizing Online Repositories:** You may get datasets from GitHub, Kaggle, and the UCI Machine Learning Repository, among other online repositories. Once downloaded, use the file upload feature to upload them to your Jupyter Notebook environment.

- **Utilizing APIs:** Application Programming Interfaces, or APIs, provide access to certain datasets. To make HTTP requests and get data straight into your notebook from these APIs, you may utilize libraries like as'requests'.
- **Using Built-in Datasets:** For learning and experimentation, certain Python packages include built-in datasets. You can load pre-built datasets from programs like "scikit-learn" and "seaborn" straight into your notebook.
- **Using Web Scraping:** If the data is available online, you may extract it and import it into your notebook by using web scraping techniques with libraries such as BeautifulSoup or Scrapy.
- **Accessing Local Files:** You may use file paths to directly access data that is stored locally on your computer from your notebook environment.

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

There are several ways to import data into a Jupyter Notebook, depending on the format and data source. Here are a few typical methods:

- **From a Local File:** Pandas can read data from files like CSV, Excel, JSON, and more if it is kept locally on your computer.

```
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:** Pandas may be used to read data straight from a URL if it is hosted online.

```
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 available via an API, you may retrieve it and parse it into a Pandas DataFrame by using libraries such as "requests."

```
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:** You can import pre-made datasets from certain Python packages straight 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 available online, you may extract it and import it into your notebook by using web scraping techniques with libraries such as BeautifulSoup or Scrapy.

```
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

The qualities or attributes of a dataset are referred to as data attributes. These properties give details on the metadata, contents, and organization of the data. Common data properties in a typical dataset are as follows:

- **Dimensions:** The dataset's dimensions show how big it is in terms of rows and columns. A tuple (rows, columns) or discrete attributes designating the number of rows (instances) and columns (features) are common ways to display this data.
- **Column Names:** The labels given to each column in the dataset are called column names or feature names. These labels specify the kind of data that is contained in each column and offer context. Usually, they are kept as an array or list of strings.
- **Missing Values:** In a dataset, missing values are the lack of data in certain cells or entries. In order to prevent biased analysis or problems with model performance, it is essential to recognize and manage missing values correctly during data preparation.
- **Summary Statistics:** The dataset's mean, median, standard deviation, minimum and maximum values for numerical columns, and other descriptive information

are provided by summary statistics. These statistics provide information on the data's central tendency and distribution.

- **Unique Values:** Distinct values found in every column of the dataset are represented by unique values. Recognizing the distinct values can aid in classifying variables, spotting anomalies, and evaluating the quality of the data.
- **Data Range:** The values observed in each column, including minimum and maximum, are specified by the data range. It facilitates figuring out the range of data and identifying outliers or abnormalities.
- Additional details about the dataset, including as the date and source of the data, the techniques used for data collection, and any pertinent annotations or comments, are included in the metadata. The data may be understood and placed in context with the aid of metadata.

Python users may utilize a range of methods and functions offered by data manipulation libraries like Pandas to access and investigate these features. 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())
```