Engineering and Applied Science Programs for Professionals
Whiting School of Engineering
Johns Hopkins University
685.621 Algorithms for Data Science

Introduction to Python refresher and Data Structures refresher

This document offers a Python refresher that covers syntax and fundamentals, including a review of basic Python concepts through quick tutorials. To regain proficiency in Python coding, participants can engage in practice coding, which involves solving problems on platforms of the students' choice. The section on Data Science libraries focuses on packages like NumPy, pandas, scikit-learn, and Matplotlib, covering topics such as data manipulation and visualization basics. The Python Basics section covers syntax, variables, basic data types, and control structures (if, for, while). In addition, it includes a review of functions and modules, covering topics such as writing and using functions, as well as understanding and utilizing modules and packages.

The refresher on Data Structures provides a recap of fundamental structures such as lists, stacks, queues, linked lists, and their operations. It also explores more advanced structures like trees, heaps, graphs, and hash tables. These structures find application in data science, for instance, decision trees and network analysis. To enhance understanding, it is recommended to implement these data structures in Python by writing code.

This document is an extension of the research and lecture notes completed at Johns Hopkins University, Whiting School of Engineering, Engineering for Professionals, Artificial Intelligence Master's Program, Computer Science Master's Program and Data Science Master's Program.

Contents

1 Python Refresher using e NumPy, pandas, scikit-learn, and Matplotlib

A review of the major Python libraries employed in data science and machine learning, accompanied by a straightforward code illustration for each. These illustrations can be executed in a Jupyter Notebook to gain practical familiarity.

1.1 NumPy Refresher

NumPy is a robust library that serves as the foundation for the majority of Python-based data science tasks. Its effectiveness lies in its ability to perform array operations efficiently, facilitate broadcasting, and offer an extensive range of mathematical tools. It offers support for arrays, mathematical functions, random number generation, and additional functionalities. To gain a deeper understanding, you can refer to the following comprehensive explanation accompanied by coding examples that can be executed in a Jupyter Notebook.

1.1.1 NumPy Arrays

NumPy arrays are the core concept of the library. They are similar to Python lists but can be multi-dimensional and are more efficient for numerical operations.

Creating a NumPy Array:

```
import numpy as np

# Create a simple one-dimensional array
one_d_array = np.array([1, 2, 3, 4, 5])
print("One-dimensional array:\n", one_d_array)

# Create a two-dimensional array
two_d_array = np.array([[1, 2, 3], [4, 5, 6]])
print("\nTwo-dimensional array:\n", two_d_array)
```

1.1.2 Array Operations

Vectorized operations, which are executed element-wise, are fully supported by NumPy arrays and are known for their exceptional speed.

Array Operations Example:

```
# Basic arithmetic operations
addition = one_d_array + 10
print("Addition:\n", addition)

# Element-wise multiplication
multiplication = one_d_array * 2
print("\nMultiplication:\n", multiplication)

# More complex operations like square root
sqrt_array = np.sqrt(one_d_array)
print("\nSquare root:\n", sqrt_array)
```

1.1.3 Broadcasting

Broadcasting is a feature of NumPy that allows for performing arithmetic operations on arrays with different shapes.

Broadcasting Example:

```
# Adding a one-dimensional array to a two-dimensional array
result = two_d_array + one_d_array
print("Broadcasting result:\n", result)
```

1.1.4 Mathematical Functions

NumPy offers a wide range of mathematical functions that can be used to perform operations on these arrays.

Mathematical Functions Example:

Mathematical Functions Example:

```
# Compute the exponential of all elements in the input array
exponential_array = np.exp(one_d_array)
print("Exponential:\n", exponential_array)

# Calculate the natural logarithm
log_array = np.log(one_d_array)
print("\nNatural logarithm:\n", log_array)
```

1.1.5 Random Number Generation

In addition, NumPy possesses robust functionality for generating random numbers.

Random Number Generation Example:

```
# Generate a 2x3 array of random numbers from the normal distribution
random_array = np.random.randn(2, 3)
print("Random array from normal distribution:\n", random_array)

# Generate random integers from 0 to 10
random_integers = np.random.randint(0, 10, size=(5,))
print("\nRandom integers:\n", random_integers)
```

1.1.6 Array Indexing and Slicing

NumPy arrays can be indexed and sliced using a syntax similar to that of Python lists.

Array Indexing and Slicing Example:

```
# Accessing the second element of one_d_array
second_element = one_d_array[1]
print("Second element:", second_element)

# Slicing the first two elements of one_d_array
slice_array = one_d_array[:2]
print("\nFirst two elements:", slice_array)

# Slicing a two-dimensional array
```

```
slice_two_d_array = two_d_array[:, 1:3]
print("\nSliced two-dimensional array:\n", slice_two_d_array)
```

1.1.7 Reshaping Arrays

The data in an array can be modified without altering its shape.

Reshaping Array Example:

```
# Reshape a one-dimensional array to a 2x5 two-dimensional array
reshaped_array = np.reshape(one_d_array, (2, -1))
print("Reshaped array:\n", reshaped_array)
```

The provided code snippets can be executed in a Jupyter Notebook cell by cell. To view the output, each cell needs to be executed individually. NumPy offers a comprehensive documentation with numerous additional functionalities and examples that can be explored to enhance your comprehension.

1.2 pandas Refresher

pandas is a fast, powerful, flexible, and user-friendly open-source tool for manipulating and analyzing data. It is built on top of the Python programming language. pandas introduces two new data structures to Python – `DataFrame`, which can be compared to a relational data table, and `Series`, which represents a single column.

1.2.1 Example - Basic Data Manipulation with pandas:

```
import pandas as pd

# Creating a simple DataFrame
data = {'Name': ['Anna', 'Bob', 'Charles'],
        'Age': [28, 34, 29],
        'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)

# Accessing data
print(df)

# Adding a column
df['Employed'] = [True, False, True]
print("\nDataFrame with an added column:\n", df)
```

1.2.2 pandas Series

A Series is an array that is labeled and can hold data of any type in one dimension.

Creating a pandas Series:

```
import pandas as pd

# Creating a series from a list
s = pd.Series([1, 3, 5, 7, 9])
print("Series from a list:\n", s)

# Creating a series with an index
s_indexed = pd.Series([1, 3, 5, 7, 9], index=['a', 'b', 'c', 'd', 'e'])
print("\nSeries with an index:\n", s_indexed)
```

1.2.3 pandas DataFrame

A DataFrame is a tabular data structure with labeled axes (rows and columns) that is potentially heterogeneous and can be resized.

Creating a pandas DataFrame:

```
# Creating a DataFrame from a dictionary
data = {'Product': ['Apples', 'Bananas', 'Cherries', 'Dates'],
        'Price': [1.50, 2.00, 3.00, 4.00]}
df = pd.DataFrame(data)
print("DataFrame:\n", df)
```

1.2.4 Data Selection and Indexing

The process of choosing and organizing data is of utmost importance in pandas for data manipulation.

Data Selection Example:

```
# Selecting a single column, which yields a Series
print("Price column as a Series:\n", df['Price'])

# Selecting via the DataFrame's `loc` method
print("\nRows with loc:\n", df.loc[1:2])

# Selecting via the DataFrame's `iloc` method
print("\nRows with iloc:\n", df.iloc[1:3])
```

1.2.5 Data Filtering

Performing data filtration based on specified conditions is a frequently encountered task in the pandas library.

Data Filtering Example:

```
# Filtering rows where price is greater than 2.00
filtered_df = df[df['Price'] > 2.00]
print("Filtered DataFrame:\n", filtered_df)
```

1.2.6 Data Manipulation

Manipulating data in pandas is a simple process that involves adding, removing, or modifying information.

Data Manipulation Example:

```
# Adding a new column
df['In Stock'] = [True, True, False, True]
print("DataFrame with new column:\n", df)

# Removing a column
df.drop('In Stock', axis=1, inplace=True)
print("\nDataFrame after removing column:\n", df)
```

1.2.7 Handling Missing Data

pandas provides strong capabilities for managing data that is missing.

Handling Missing Data Example:

```
# Creating a DataFrame with missing values
data_with_missing = {'Product': ['Apples', None, 'Cherries', 'Dates'],
                     'Price': [1.50, 2.00, None, 4.00]}
df_missing = pd.DataFrame(data_with_missing)
print("DataFrame with missing values:\n", df_missing)

# Filling missing values
df_missing_filled = df_missing.fillna('Unknown')
print("\nDataFrame after filling missing values:\n", df_missing_filled)
```

1.2.8 Data Aggregation and Grouping

The act of grouping data and executing aggregate functions is a frequently encountered analytical pattern.

Data Aggregation Example:

```
# Grouping and aggregating data
data = {'Product': ['Apples', 'Bananas', 'Apples', 'Bananas', 'Cherries'],
        'Sales': [10, 20, 15, 30, 7]}
df_sales = pd.DataFrame(data)

# Sum sales by product
sales_by_product = df_sales.groupby('Product').sum()
print("Sum of sales by product:\n", sales_by_product)
```

1.2.9 Input/Output Operations

pandas has the ability to read and write data in a variety of formats.

I/O Example:

```
# Reading from a CSV file
df_from_csv = pd.read_csv('path_to_csv_file.csv')

# Writing to a CSV file
df.to_csv('path_to_output_csv_file.csv')
```

These examples offer a sneak peek into the capabilities of pandas and can be executed in a Jupyter Notebook. The process of working with pandas typically entails data cleaning, data transformation, data visualization, and data analysis.

1.3 scikit-learn Refresher

scikit-learn is a powerful and user-friendly tool that facilitates the process of predictive data analysis. It is built on the NumPy, SciPy, and Matplotlib libraries. scikit-learn allows for efficient implementation of machine learning algorithms. The estimators serve as the foundation objects that utilize a fit method to learn from data. The predictors, which are a type of estimator, can forecast a target based on input data by employing a predict method. The transformers, also estimators, can modify a dataset by employing a transform method. The term "models" is often used interchangeably with predictors, but

it can also encompass statistical models and transformers. scikit-learn is accessible to all and can be reused in a variety of contexts.

1.3.1 Example - Supervised Learning with KNN

In supervised learning, classification is a task that involves predicting the categorical class labels of new instances using historical observations.

Using the Iris Dataset:

```
# Import the necessary functions
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Load the Iris dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the model (k-Nearest Neighbors)
knn = KNeighborsClassifier(n_neighbors=5)

# Fit the model on the training data
knn.fit(X_train_scaled, y_train)

# Make predictions on the test data
y_pred = knn.predict(X_test_scaled)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Confusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)
```

> Output:
>
> ```
> Confusion Matrix:
>  [[19  0  0]
>  [ 0 13  0]
>  [ 0  0 13]]
>
> Classification Report:
>               precision    recall  f1-score   support
> ```

```
              0       1.00      1.00      1.00        19
              1       1.00      1.00      1.00        13
              2       1.00      1.00      1.00        13

    accuracy                            1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

1.3.2 Example - Regression with Linear Regression

Regression is an alternative form of supervised learning task in which the objective is to forecast a continuous value rather than a category.

Using the Diabetes Dataset:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load the diabetes dataset
diabetes = datasets.load_diabetes()
X, y = diabetes.data, diabetes.target

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create linear regression object
regr = LinearRegression()

# Train the model using the training sets
regr.fit(X_train, y_train)

# Make predictions using the testing set
y_pred = regr.predict(X_test)

# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(y_test, y_pred))
```

Output:

```
Mean squared error: 2924.46
```

1.3.3 Example - Unsupervised Learning with K-Means

Clustering refers to an unsupervised learning task that aims to classify similar instances into clusters.

Using the Iris Dataset:

```
from sklearn import datasets
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Load the iris dataset
iris = datasets.load_iris()
X = iris.data
```

```
# KMeans clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X)

# Plot the clustered data
plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.title("K-Means Iris Dataset with 3 Clusters")
plt.xlabel("Petal Length")
plt.ylabel("Petal Width")
plt.savefig('KMeansIrisClustering.svg')
plt.show()
```

> Output:

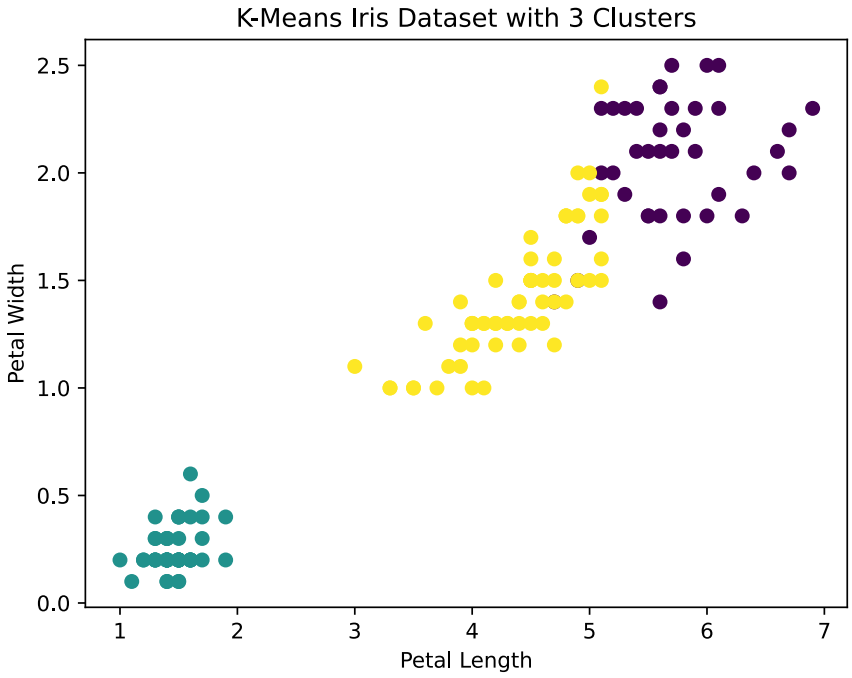

Figure 1: K-Means Iris Dataset with 3 Clusters, x-axis Petal Length, y-axis Petal Width

You can run these examples in a Jupyter Notebook environment, which provides instant outputs and visualizations. Scikit-learn's API is consistent, making it simple to try out various models and techniques. The library includes numerous utilities for tasks such as data splitting, preprocessing, model selection, and evaluation, making it a valuable tool for data scientists.

1.4 Matplotlib Refresher

Matplotlib is a Python library that offers a wide range of options for creating visualizations, including static, animated, and interactive ones. It enables users to embed plots into applications using popular GUI toolkits such as Tkinter, wxPython, Qt, or GTK, thanks to its object-oriented API.

1.4.1 Example - Histogram

Histograms are employed to depict the dispersion of a dataset.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate Random Data
data = np.random.randn(1000)

# Plot
plt.hist(data, bins=30)
plt.title('Histogram')
plt.xlabel('Data')
plt.ylabel('Frequency')
plt.savefig('Histogram.svg')
plt.show()
```

> Output:

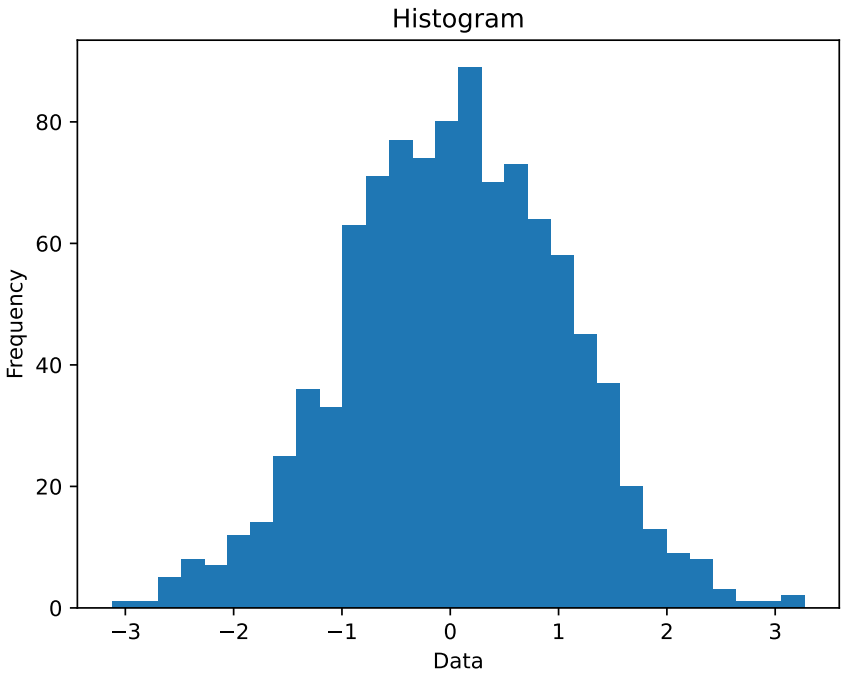

Figure 2: Example histogram figure.

1.4.2 Example - Multiple Figures and Axes

With Matplotlib, it is possible to create multiple subplots within a single figure.

```
import matplotlib.pyplot as plt
```

```
# The 1st subplot
plt.subplot(1, 2, 1) # The parameters are (rows, columns, panel number)
plt.plot([1, 2, 3], [4, 5, 6])
plt.title('First Subplot')

# The 2nd subplot
plt.subplot(1, 2, 2)
plt.plot([4, 5, 6], [1, 2, 3])
plt.title('Second Subplot')

# Use tight_layout to adjust plot layout
plt.tight_layout()
plt.show()
```

> Output:

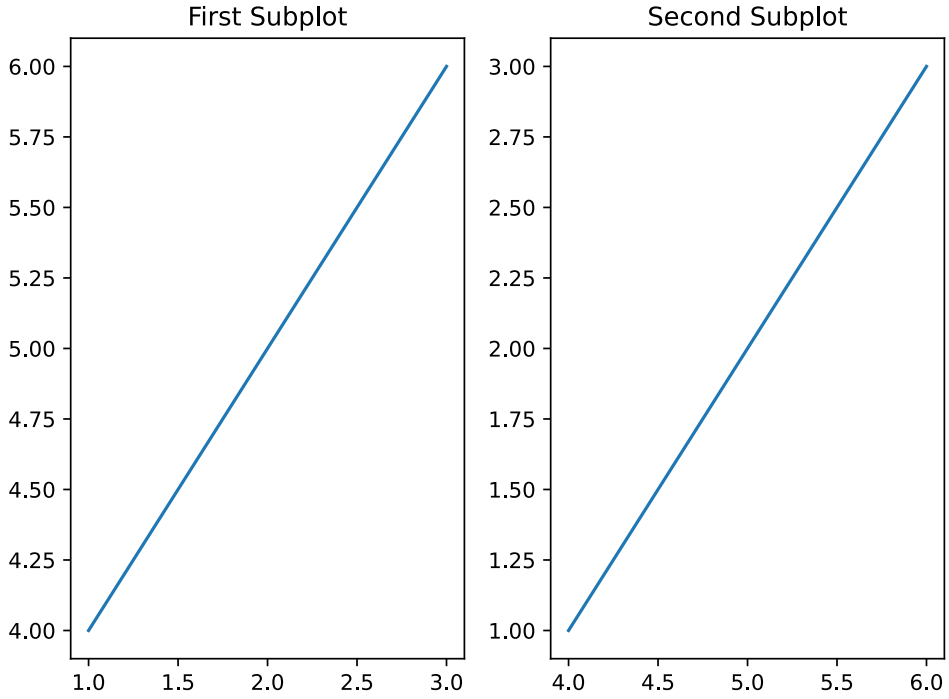

Figure 3: Example Multiple Figures and Axes.

1.4.3 Example - Bar Chart

Bar charts are effective for the purpose of comparing quantities.

```
import matplotlib.pyplot as plt

# Generate Data
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 30]
```

```
# Plot
plt.bar(categories, values)
plt.title('Bar Chart')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

> Output:

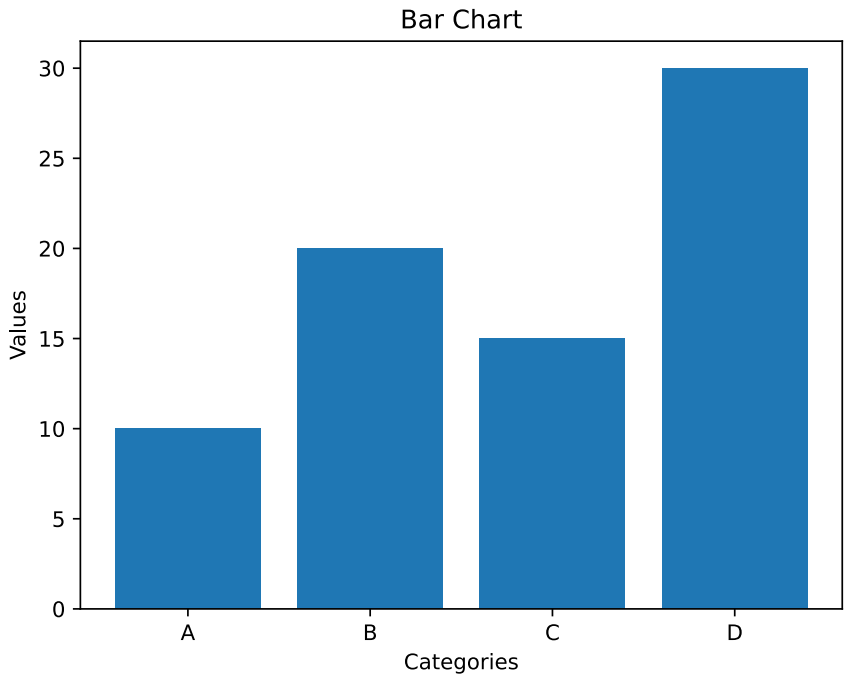

Figure 4: Example Bar Charts.

1.4.4 Example - Pie Chart

Pie charts are visual representations of data in a circular format that are useful for illustrating proportions.

```
import matplotlib.pyplot as plt

# Data
sizes = [25, 35, 20, 20]
labels = ['Dogs', 'Cats', 'Lizards', 'Other']

# Plot
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Pie Chart')
plt.show()
```

1.4.5 Example - 3D Plotting

Matplotlib also has the capability to plot in three dimensions.

```
# Generate Data for the 3D Example
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)
z = np.sin(np.sqrt(x**2 + y**2))

# Plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')

plt.title('3D Surface Plot')
plt.show()
```

> Output:

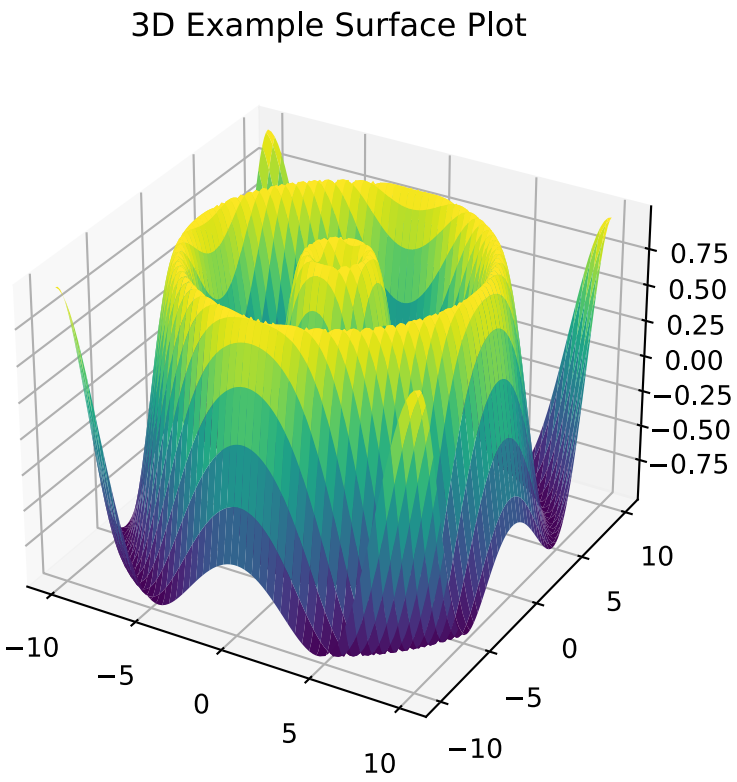

Figure 5: Example 3D Surface Plot.

To execute these code examples, make sure you have a Jupyter Notebook open. Input each code sample into separate cells and run them individually. By executing each cell, you will be able to see the output immediately below it. This interactive coding approach is highly beneficial for applying and exploring the data science concepts taught in this course.

2 Data Structures Refresher using Python

This section presents various topics related to data structures for effective data work and analysis. Lists and Arrays are important for storing and manipulating ordered collections of data. Concepts covered include indexing, slicing, and operations like append, insert, and delete. Understanding multidimensional arrays is crucial for working with matrices and numerical data. Dictionaries (Hash Maps) are crucial for storing key-value pairs, enabling quick lookups and data organization. Stacks and Queues are useful for managing data in a last-in-first-out (LIFO) or first-in-first-out (FIFO) manner, respectively. Knowledge of linked lists, whether singly or doubly linked, can be beneficial for various data manipulation tasks. Trees and Graphs are essential for understanding tree and graph data structures, which are used in tasks like decision tree-based algorithms and network analysis. Heaps are commonly used for priority queues and can be advantageous in optimization problems. Sets are helpful for storing unique elements and performing set operations such as union, intersection, and difference. Tuples are similar to lists but are immutable, making them suitable for storing fixed data structures. Data frames, commonly used in libraries like Pandas in Python, are tabular data structures that are central to data manipulation and analysis. Understanding how hashing functions work and their applications in data retrieval and data integrity is important for Hashing. Tries are tree-like structures used in applications such as autocomplete and spell checking. These data structure topics form the foundation for various data processing, analysis, and machine learning tasks. To gain a deeper understanding, it is recommended to research these topics in more detail using Python-based data structure frameworks and datasets from online resources such as Kaggle, Data.gov, and open-source repositories.

2.1 Lists and Arrays

Lists and arrays are fundamental data structures in Python used extensively in data science projects.

Lists are ordered collections of items. They can contain elements of different data types and are mutable, which means you can modify their contents.

Creating a List

```
# Creating a list
my_list = [1, 2, 3, 4, 5]

# Accessing elements
print(my_list[0])  # Access the first element (index 0)

# Modifying elements
my_list[2] = 10  # Modify the third element (index 2)

# Adding elements
my_list.append(6)  # Add an element to the end of the list

# Removing elements
my_list.remove(4)  # Remove an element by value
```

> Output:
>
> ```
> 1
> ```

NumPy as described in the Python refresher is a good library for numerical operations, and it provides arrays that are more efficient for numerical computations compared to lists.

Creating a NumPy Array

```python
# Importing NumPy
import numpy as np

# Creating a NumPy array
my_array = np.array([1, 2, 3, 4, 5])

# Accessing elements
print(my_array[0])  # Access the first element (index 0)

# Modifying elements
my_array[2] = 10  # Modify the third element (index 2)

# Basic array operations
sum_result = np.sum(my_array)  # Sum of all elements
mean_result = np.mean(my_array)  # Mean of all elements

# Generating arrays
zeros_array = np.zeros(5)  # Array of zeros
ones_array = np.ones(5)    # Array of ones

# Array operations
result = my_array * 2  # Multiplying all elements by 2
```

> Output:
>
> ```
> 1
> ```

Creating a Multidimensional Array

```python
# Importing NumPy
import numpy as np

# Creating a 2D NumPy array
matrix_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing elements
print(matrix_2d[0, 1])  # Access element at row 0, column 1

# Slicing
row_1 = matrix_2d[1, :]  # Get the entire second row
column_2 = matrix_2d[:, 2]  # Get the entire third column

# Matrix operations
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

# Element-wise addition
result_addition = matrix_a + matrix_b

# Matrix multiplication
result_multiplication = np.dot(matrix_a, matrix_b)

# Transpose
matrix_transpose = np.transpose(matrix_a)
```

```
# Shape and dimensions
shape = matrix_2d.shape  # Get the shape of the 2D array
dimensions = matrix_2d.ndim  # Get the number of dimensions

# Generating multidimensional arrays
identity_matrix = np.eye(3)  # 3x3 identity matrix
random_matrix = np.random.rand(2, 2)  # 2x2 random matrix

# Reshaping arrays
flat_array = np.array([1, 2, 3, 4, 5, 6])
reshaped_array = flat_array.reshape(2, 3)  # Reshape to a 2x3 array
```

> Output:
>
> ```
> 2
> ```

2.2 Dictionaries

Dictionaries consist of pairs of keys and values, with each key being distinct and linked to a specific value. They are highly adaptable and extensively utilized in the field of data science for a range of applications, including the storage of metadata or the conversion of categorical data into numerical representations.

Creating a Dictionary

```
# Creating a dictionary
student_info = {
    "name": "John Doe",
    "age": 26,
    "major": "Data Science",
    "GPA": 3.85
}

# Accessing values using keys
name = student_info["name"]
age = student_info["age"]
major = student_info["major"]
GPA = student_info["GPA"]

# Modifying values
student_info["GPA"] = 3.9  # Update GPA

# Adding new key-value pairs
student_info["university"] = "Johns Hopkins University"

# Checking if a key exists
if "GPA" in student_info:
    print("GPA is available in the dictionary.")

# Removing key-value pairs
del student_info["major"]

# Iterating through keys and values
for key, value in student_info.items():
    print(f"{key}: {value}")
```

```python
# Dictionary comprehension
grades = {"Algorithms": 90.5, "Data Science": 85.9, "Data Engineering": 79.9}
passed_subjects = {subject: score for subject, score in grades.items() if score >= 80}

# Nested dictionaries
students = {
    "student1": {"name": "Alice", "age": 25},
    "student2": {"name": "Bob", "age": 28}
}

# Handling missing keys
# Get value or default if key is missing
subject_score = grades.get("Data Patterns and Representations", "N/A")

# Length of a dictionary
num_keys = len(student_info)

# Keys and values as lists
keys = list(student_info.keys())
values = list(student_info.values())
```

> Output:
>
> ```
> GPA is available in the dictionary.
> name: John Doe
> age: 26
> GPA: 3.9
> university: Johns Hopkins University
> ```

Dictionaries play a crucial role in data science as they are highly beneficial for various tasks such as data preprocessing, feature engineering, and data transformation. These tasks often require mapping and manipulating data based on specific criteria or metadata.

2.3 Stacks and Queues

Stacks and queues play a crucial role as data structures in Python and find applications in a wide range of data science assignments and projects.

Creating a Stack A stack is a type of linear data structure that adheres to the Last-In-First-Out (LIFO) principle. In a stack, elements are inserted and deleted from the same end, referred to as the "top" of the stack.

```python
# Importing the deque class for implementing a stack
from collections import deque

# Creating a stack using a deque
stack = deque()

# Pushing elements onto the stack
stack.append(1)
stack.append(2)
stack.append(3)
```

```python
# Popping elements from the stack
top_element = stack.pop()  # Removes and returns the top element

# Checking if the stack is empty
is_empty = len(stack) == 0

# Example: Evaluating a mathematical expression with a stack
def evaluate_expression(expression):
    stack = deque()
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if char == '+':
                result = operand1 + operand2
            elif char == '-':
                result = operand1 - operand2
            elif char == '*':
                result = operand1 * operand2
            elif char == '/':
                result = operand1 / operand2
            stack.append(result)
    return stack.pop()

result = evaluate_expression("3+4*2-6/2")  # Should evaluate to 7
```

Creating a Queue A queue is a different type of linear data structure that adheres to the principle of First-In-First-Out (FIFO). In a queue, elements are inserted at the rear end and removed from the front end.

```python
# Importing the deque class for implementing a queue
from collections import deque

# Creating a queue using a deque
queue = deque()

# Enqueuing elements
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeuing elements
front_element = queue.popleft()  # Removes and returns the front element

# Checking if the queue is empty
is_empty = len(queue) == 0

# Example: Simulating a print queue
def print_queue(queue):
    while queue:
        document = queue.popleft()
        print(f"Printing document: {document}")
```

```
# Enqueue documents to the print queue
print_queue = deque()
print_queue.append("Document 1")
print_queue.append("Document 2")

# Start printing
print_queue(print_queue)
```

Data science utilizes stacks and queues for a range of purposes, including executing graph traversal algorithms, organizing data in a particular sequence, and resolving problems that necessitate a Last-In-First-Out (LIFO) or First-In-First-Out (FIFO) methodology.

2.4 Linked Lists

A linked list is a type of data structure that is composed of nodes. Each node contains two components: data and a reference (or pointer) to the next node in the sequence. There are various types of linked lists, including singly linked lists where each node points to the next node, and doubly linked lists where each node points to both the next and previous nodes.

Creating a Linked List

```
# Define a Node class to represent individual elements in the linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Define a LinkedList class to manage the linked list
class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def search(self, target):
        current = self.head
        while current:
            if current.data == target:
```

```
                return True
            current = current.next
        return False

    def delete(self, target):
        if not self.head:
            return

        if self.head.data == target:
            self.head = self.head.next
            return

        current = self.head
        while current.next:
            if current.next.data == target:
                current.next = current.next.next
                return
            current = current.next

# Creating a singly linked list
my_list = LinkedList()
my_list.append(1)
my_list.append(2)
my_list.append(3)

# Displaying the linked list
my_list.display()

# Searching for an element
result = my_list.search(2)

# Deleting an element
my_list.delete(2)
```

Linked lists are valuable in the field of data science for tasks that require the storage of dynamic data or the manipulation of sequential data.

2.5 Trees and Graphs

Data science relies heavily on trees and graphs as essential data structures for a variety of tasks, such as decision tree algorithms, network analysis, and data modeling.

Creating a Tree A tree refers to a data structure that is organized in a hierarchical manner, where nodes are linked together by edges. Nodes can have multiple child nodes or none at all, but they always have a single parent node, with the exception of the root node. Trees have various applications, including decision trees and hierarchical data storage.

```
# Define a TreeNode class to represent individual nodes in the tree
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []

    def add_child(self, child_node):
```

```
        self.children.append(child_node)

# Create a tree
root = TreeNode("Root")
child1 = TreeNode("Child 1")
child2 = TreeNode("Child 2")
root.add_child(child1)
root.add_child(child2)

# Display the tree using a recursive function
def display_tree(node, depth=0):
    print("  " * depth + node.data)
    for child in node.children:
        display_tree(child, depth + 1)

display_tree(root)
```

Creating a Graph A graph is a structure for storing data that includes nodes (also known as vertices) and edges that link these nodes together. Graphs are utilized for a range of data analysis purposes, such as network analysis and modeling social networks.

```
# Define a Graph class to represent a graph
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph:
            self.graph[vertex1].append(vertex2)
        if vertex2 in self.graph:
            self.graph[vertex2].append(vertex1)

    def display_graph(self):
        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {', '.join(neighbors)}")

# Create a graph
my_graph = Graph()
my_graph.add_vertex("A")
my_graph.add_vertex("B")
my_graph.add_vertex("C")
my_graph.add_edge("A", "B")
my_graph.add_edge("B", "C")

# Display the graph
my_graph.display_graph()
```

The utilization of trees and graphs is crucial in data science for the depiction and examination of diverse data structures and connections.

2.6 Heaps

A heap is a data structure based on a binary tree that has a specific property called the heap property. In a min-heap, for instance, the value of the parent node is either smaller or equal to the values of its children, guaranteeing that the smallest element is always located at the root.

Creating a Min-Heap

```
# Importing the heapq module for heap operations
import heapq

# Create an empty list to represent a min-heap
min_heap = []

# Insert elements into the min-heap
heapq.heappush(min_heap, 3)
heapq.heappush(min_heap, 1)
heapq.heappush(min_heap, 4)
heapq.heappush(min_heap, 2)

# Get the minimum element without removing it
min_element = min_heap[0]

# Pop the minimum element from the min-heap
min_value = heapq.heappop(min_heap)

# Display the min-heap
print(min_heap)  # The remaining elements will be reordered to maintain the heap property
```

Creating a Max-Heap

```
# Create an empty list to represent a max-heap
max_heap = []

# Insert elements into the max-heap (using negation to simulate a max-heap)
heapq.heappush(max_heap, -3)
heapq.heappush(max_heap, -1)
heapq.heappush(max_heap, -4)
heapq.heappush(max_heap, -2)

# Get the maximum element without removing it (negate the result)
max_element = -max_heap[0]

# Pop the maximum element from the max-heap (negate the result)
max_value = -heapq.heappop(max_heap)

# Display the max-heap
print(max_heap)  # The remaining elements will be reordered to maintain the heap property
```

In the field of data science, heaps are frequently employed in algorithms such as Dijkstra's shortest path algorithm. They are also utilized in a range of optimization problems where the objective is to efficiently locate the minimum or maximum values.

2.7 Sets

A set is a collection of elements that are not arranged in any particular order and each element is unique. Sets are beneficial for carrying out set operations such as union, intersection, and difference. They are frequently employed in data science to manage distinct values and execute data manipulation tasks.

Creating a Set

```
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Adding elements to a set
my_set.add(6)
my_set.add(7)

# Removing elements from a set
my_set.remove(4)

# Checking if an element is in the set
element_exists = 3 in my_set

# Iterating through a set
for item in my_set:
    print(item)

# Set operations
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Union of sets
union_result = set1.union(set2)

# Intersection of sets
intersection_result = set1.intersection(set2)

# Difference of sets
difference_result = set1.difference(set2)

# Subset check
is_subset = set1.issubset(set2)

# Superset check
is_superset = set1.issuperset(set2)
```

Sets play a crucial role in data science, especially when it comes to tasks such as eliminating duplicates from datasets, carrying out set operations on categorical data, and identifying unique elements within data collections.

2.8 Tuples

A tuple is a sequence of elements that are arranged in a specific order and cannot be modified after creation. In the field of data science, tuples are employed when there is a requirement to store and retrieve a predetermined set of values.

Creating a Tuple

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Accessing elements by index
element = my_tuple[2]  # Access the third element (index 2)

# Iterating through a tuple
for item in my_tuple:
    print(item)

# Tuple packing and unpacking
name = "John"
age = 30
location = "New York"

# Packing values into a tuple
person_info = (name, age, location)

# Unpacking values from a tuple
name, age, location = person_info

# Length of a tuple
tuple_length = len(my_tuple)

# Concatenating tuples
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined_tuple = tuple1 + tuple2

# Slicing a tuple
sliced_tuple = my_tuple[1:4]  # Get elements from index 1 to 3

# Checking if an element exists in a tuple
element_exists = 3 in my_tuple

# Counting occurrences of an element in a tuple
count = my_tuple.count(3)

# Finding the index of an element in a tuple
index = my_tuple.index(4)

# Tuple with a single element (note the comma)
single_element_tuple = (1,)

# Immutability of tuples
# my_tuple[0] = 10  # This will result in an error since tuples are immutable
```

Tuples are frequently employed in the field of data science to guarantee the preservation of a set of values throughout the entire data processing pipeline. They are also utilized to return multiple values from functions and to organize data in a predetermined format.

2.9 Data Frames

A data frame is a structured data representation with labeled axes (rows and columns), presented in a tabular format. It provides an efficient way to store and manipulate structured data. The Pandas library is commonly utilized in the field of data science for working with data frames.

Creating a Data Frame

```
# Importing the Pandas library
import pandas as pd

# Creating a data frame from a dictionary
data = {
    'Name': ['John', 'Lamar', 'Odell', 'Zay', 'Gus'],
    'Age': [61, 27, 31, 23, 35],
    'College': ['Miami (OH)', 'Louisville', 'LSU', 'Boston College', 'Rutgers']
}

df = pd.DataFrame(data)

# Displaying the data frame
print(df)

# Accessing columns
names = df['Name']
ages = df['Age']

# Filtering rows
young_people = df[df['Age'] < 30]

# Adding a new column
df['Gender'] = ['Female', 'Male', 'Male', 'Male', 'Female']

# Deleting a column
df.drop('College', axis=1, inplace=True)

# Summary statistics
summary_stats = df.describe()

# Grouping and aggregation
grouped_data = df.groupby('Gender')['Age'].mean()

# Sorting the data frame
sorted_df = df.sort_values(by='Age', ascending=False)

# Writing data to a CSV file
df.to_csv('output_data.csv', index=False)

# Reading data from a CSV file
csv_df = pd.read_csv('output_data.csv')
```

Data frames are an effective instrument for investigating, sanitizing, analyzing, and visualizing data in data science undertakings. They enable effortless manipulation of data, execution of diverse operations, and efficient handling of structured datasets.

2.10 Hashing

Hashing is a procedure that transforms data, such as text or numbers, into a string of characters with a fixed size. This resulting string is usually a distinct representation of the original input data. In Python, hash functions can be utilized for performing hashing operations.

Creating a Hash

```
# Importing the hashlib library for hash functions
import hashlib

# Creating a simple hash
data = "Hello, world!"

# Creating a hash object using SHA-256 hash function
sha256_hash = hashlib.sha256()

# Updating the hash object with data
sha256_hash.update(data.encode())

# Getting the hexadecimal representation of the hash
hashed_data = sha256_hash.hexdigest()

# Displaying the hashed data
print("Original Data:", data)
print("Hashed Data:", hashed_data)

# Using hash() built-in function
value = "Hello"
hashed_value = hash(value)

# Displaying the hashed value
print("Hashed Value:", hashed_value)
```

Hashing is extensively utilized in the field of data science for a multitude of applications, including but not limited to, data indexing in databases, data integrity verification, and efficient data retrieval in hash tables. The primary objective of hash functions is to produce distinct hash values for different inputs, ensuring data consistency and enhancing security.

2.11 Trie

A trie is a type of tree structure that uses nodes to represent individual characters within a string. The nodes are linked together in a manner that enables efficient retrieval and searching of strings.

Creating a Trie

```
# Define a TrieNode class
class TrieNode:
    def __init__(self):
        self.children = {}  # Dictionary to store child nodes
        self.is_end_of_word = False  # Marks the end of a word

# Define a Trie class
class Trie:
    def __init__(self):
```

```
        self.root = TrieNode()  # The root node of the trie

    # Insert a word into the trie
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    # Search for a word in the trie
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

# Create a trie
trie = Trie()

# Insert words into the trie
trie.insert("apple")
trie.insert("app")
trie.insert("banana")
trie.insert("bat")

# Search for words in the trie
search_result_apple = trie.search("apple")
search_result_banana = trie.search("banana")
search_result_orange = trie.search("orange")

print("Search 'apple':", search_result_apple)
print("Search 'banana':", search_result_banana)
print("Search 'orange':", search_result_orange)
```

Tries are extremely beneficial in the field of data science when it comes to tasks such as autocomplete, spell-checking, and the implementation of efficient search engines. They enable quick and effective operations involving strings.