knowledgehut upGrad

Introduction to NumPy

About Me

Hi, I am Akash, an experienced Data Analytics professional with over 8 years of industry experience, skilled in developing:

- Machine learning models
- Statistical analysis and
- Visualizations

- Proficient in utilizing tools such as Python, SAS, PySpark, SQL, SparkSQL, Hive, Tableau, and Excel.

- Has a proven track record of delivering innovative solutions to real-world problems in various domains such as healthcare, finance, and power.

Pre-requisites

Hope you have gone through the self-learning content for this session on the PRISM portal.

By the End of this Session, you will:

- Construct, manipulate, and identify elements within NumPy arrays to gain a foundational understanding of this core data structure.
- Execute various mathematical and logical operations on NumPy arrays, enhancing skills in numerical data manipulation.
- Master the extraction of specific elements or subsets of an array using indexing and slicing, allowing precise and flexible data access.
- Apply indexing and slicing techniques effectively and perform mathematical operations on arrays.
- Employ built-in functions in NumPy to perform various operations on arrays, increasing proficiency in leveraging available tools for data processing and analysis.

Recap

Poll Time

Q. Which of the following tasks do you think NumPy is commonly used for? (Select all that apply)

a. Mathematical computations

b. Data visualization

c. Linear algebra operations

d. Modifying string

Poll Time

Q. Which of the following tasks do you think NumPy is commonly used for? (Select all that apply)

a. [✓ **Mathematical computations**]

b. Data visulaization

c. [✓ **Linear algebra operations**]

d. Modifying strings

Introduction to NumPy

Why NumPy?

NumPy is a Python library for numerical computing.

Provides a powerful array object that allows efficient storage and manipulation of large multidimensional arrays.

NumPy offers a wide range of mathematical functions and operations to work with these arrays.

Widely used in scientific computing, data analysis, and machine learning tasks.

Need for NumPy in Data Science

NumPy provides high-performance multidimensional arrays, allowing efficient storage and manipulation of large datasets, which is crucial in data science applications.

Offers a comprehensive set of mathematical functions and operations, enabling complex numerical computations and data transformations.

NumPy's array operations are optimized and executed at the C level, resulting in faster execution times compared to traditional Python lists or loops.

Provides powerful indexing and slicing capabilities, allowing for easy extraction and manipulation of subsets of data.

Need for NumPy in Data Science

NumPy integrates seamlessly with other data science libraries like pandas, Matplotlib, and scikit-learn, forming a powerful ecosystem for data analysis and machine learning.

Many data science algorithms and frameworks, including machine learning models, rely on NumPy arrays as their fundamental data structure, making it a crucial component in data science workflows.

Demo – Import NumPy

Pop Quiz

Q. What is the purpose of NumPy in Data Science?

a. To provide a high-performance array object for efficient storage and manipulation of large datasets

b. To offer a comprehensive set of mathematical functions and operations for complex numerical computations and data transformations

c. To integrate seamlessly with other data science libraries like pandas and scikit-learn

d. All of the listed

Pop Quiz

Q. What is the purpose of NumPy in Data Science?

a. To provide a high-performance array object for efficient storage and manipulation of large datasets

b. To offer a comprehensive set of mathematical functions and operations for complex numerical computations and data transformations

c. To integrate seamlessly with other data science libraries like pandas and scikit-learn

[✓ **d. All of the listed**]

Introduction to NumPy Arrays

What are NumPy Arrays?

NumPy arrays are data structures that store homogeneous, multi-dimensional data.

They provide efficient storage and manipulation of large datasets, allowing for faster computations compared to traditional Python lists.

NumPy arrays have a fixed size and shape, enabling efficient memory allocation and optimized mathematical operations on the data.

Creating and using NumPy Arrays

Import the NumPy library:

```
import numpy as np
```

Create a NumPy array using the np.array() function:

```
arr = np.array([1, 2, 3, 4, 5])
```

This creates a 1-dimensional array with the values [1, 2, 3, 4, 5].

Access and manipulate elements in the array:

```
print(arr[0])
# Access the first element (1)
arr[2] = 6
# Modify the third element to 6
```

Perform mathematical or logical operations on arrays:

```
sum_arr = arr + 2 # Add 2 to each element
product_arr = arr * 3 # Multiply each element by 3
```

Demo – Arrays Indexing and Slicing

Perform slicing to extract subsets of the array:

```
print(arr[1:4])
# Extract elements from index 1 to 3 (inclusive)
print(arr[:3])
# Extract elements from the beginning up to index 2
print(arr[2:]) #
Extract elements from index 2 to the end
```

Create and manipulate multi-dimensional arrays:

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr_2d[1][2]) # Access the element at row 1, column 2
```

Pop Quiz

Q. Given the NumPy array arr = np.array([1, 2, 3, 4, 5, 6, 7]), what will be the result of the following slicing operation?
arr[2:5]

a. [3, 4, 5]

b. [2, 3, 4]

c. [4, 5, 6]

d. [2, 3, 4, 5]

Pop Quiz

Q. Given the NumPy array arr = np.array([1, 2, 3, 4, 5, 6, 7]), what will be the result of the following slicing operation?
arr[2:5]

a. **[3, 4, 5]** [✓]

b. [2, 3, 4]

c. [4, 5, 6]

d. [2, 3, 4, 5]

Introduction to Array Shapes, Dimensions, and Data Types

Arrays in NumPy have three important properties: Shape, dimensions, and data types.

Shape

- The shape of an array refers to the size of each dimension. It tells you the number of elements along each axis of the array.
- For example, a 1-dimensional array with 5 elements has a shape of (5). Whereas a 2-dimensional array with 3 rows and 4 columns has a shape of (3, 4).
- You can access the shape of an array using the shape attribute.

Dimensions

- The dimensions of an array represent the number of axes or levels it has.
- A 1-dimensional array has a single dimension, a 2-dimensional array has two dimensions, and so on. The number of dimensions of an array is known as its rank.
- You can access the number of dimensions using the ndim attribute.

Data Types

- Arrays in NumPy have a data type, which determines the type of elements they can store.
- NumPy supports various data types, including integers, floating-point numbers, booleans, and more.
- The data type of an array is determined during its creation and can be accessed using the dtype attribute.

Q&A

Array Mathematical Operations

Basic Mathematical Operations

Aggregation Functions: NumPy provides functions to compute various aggregations over an array, such as np.sum(), np.mean(), np.max(), np.min(), and more. These functions allow you to calculate aggregate values across the entire array or along specific axes.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
result = np.sum(arr) # Output: 21 # Sum of all elements

# Mean along axis 0 (column-wise)
result = np.mean(arr, axis=0) # Output: [2.5, 3.5, 4.5]
```

Basic Mathematical Operations

Mathematical Functions: NumPy provides a comprehensive set of mathematical functions that can be applied to arrays. These include functions like np.sin(), np.cos(), np.exp(), np.log(), and more. These functions operate element-wise on arrays, applying the mathematical operation to each element.

Example:

```
import numpy as np
arr = np.array([0, np.pi/2, np.pi])

# Exponential function
result = np.exp(arr) # Output: [1.0, 4.81047738, 23.14069263]

# Natural logarithm function
result = np.log(arr) # Output: [-inf, 0.4515827, 1.14472989]
```

Dot Product

The dot product, also known as the inner product or scalar product, is a mathematical operation between two arrays that results in a scalar value. In NumPy, you can calculate the dot product using the np.dot() function or the @ operator.

The dot product is calculated as the sum of the element-wise products of the corresponding elements from the two arrays. However, for the dot product to be valid, the arrays must have compatible shapes.

Dot Product

Here's an example of how to calculate the dot product using NumPy:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
dot_product = np.dot(arr1, arr2)
# Output: 32
```

In this example, the dot product of arr1 and arr2 is calculated as (1 * 4) + (2 * 5) + (3 * 6) = 32.

The dot product is useful in various applications, such as calculating similarity measures, projection, regression analysis, and solving systems of linear equations. It provides a way to measure the relationship or alignment between two arrays.

Pop Quiz

Q. Given two NumPy arrays, arr1 = np.array([2, 3, 4]) and arr2 = np.array([5, 6, 7]), what will be the result of the dot product of these arrays?

a. 47

b. 56

c. 32

d. [10, 18, 28]

Pop Quiz

Q. Given two NumPy arrays, arr1 = np.array([2, 3, 4]) and arr2 = np.array([5, 6, 7]), what will be the result of the dot product of these arrays?

a. 47

b. 56 ✓

c. 32

d. [10, 18, 28]

Exponentiation and Logarithms

NumPy provides functions to perform exponentiation and logarithmic operations on arrays efficiently.

Here's an overview of exponentiation and logarithmic functions available in NumPy:

Exponentiation:
Exponential Function: The exponential function np.exp() calculates the exponential value of each element in the array.
Example:

```
import numpy as np
arr = np.array([1, 2, 3])
# Exponential function
result = np.exp(arr)
# Output: [2.71828183, 7.3890561, 20.08553692]
```

Power Function: The power function np.power() raises each element in the array to a specified power.
Example:

```
import numpy as np
arr = np.array([2, 3, 4])
# Power function
result = np.power(arr, 2)
# Output: [4, 9, 16]
```

Exponentiation and Logarithms

Natural Logarithm: The natural logarithm function np.log() calculates the natural logarithm (base e) of each element in the array.
Example:

```
import numpy as np
arr = np.array([1, 2, 3])
# Natural logarithm
result = np.log(arr)
# Output: [0.0, 0.69314718, 1.09861229]
```

These are some of the exponentiation and logarithmic functions available in NumPy. They allow you to perform element-wise calculations on arrays, providing convenient tools for various mathematical and scientific computations.

Trigonometric Functions

NumPy provides a variety of trigonometric functions that can be applied element-wise to arrays.

Here's a brief overview of some commonly used trigonometric functions in NumPy:

Sine Function: The sine function np.sin() calculates the sine of each element in the array

Example:

```
import numpy as np
 arr = np.array([0, np.pi/2, np.pi])
# Sine function
result = np.sin(arr)
# Output: [0.0, 1.0, 1.2246468e-16]
```

Trigonometric Functions

Here's a brief overview of some commonly used trigonometric functions in NumPy:

Tangent Function: The tangent function np.tan() calculates the tangent of each element in the array.

Example:

```
import numpy as np
arr = np.array([0, np.pi/4, np.pi/2])
# Tangent function
result = np.tan(arr)
# Output: [0.0, 1.0, 1.63312394e+16]
```

Pop Quiz

Q. Given the NumPy array arr = np.array([2, 3, 4]), what will be the result of raising each element in the array to the power of 3 using the power function np.power()?

a. [8, 9, 16]

b. [2, 6, 12]

c. [6, 9, 12]

d. [8, 27, 64]

Pop Quiz

Q. Given the NumPy array arr = np.array([2, 3, 4]), what will be the result of raising each element in the array to the power of 3 using the power function np.power()?

a. [8, 9, 16]

b. [2, 6, 12]

c. [6, 9, 12]

d. [8, 27, 64]

Reshaping and Resizing an Array

Reshaping and Resizing

Here's a brief overview of reshaping and resizing arrays in NumPy:

Resizing:
Resize: The np.resize() function changes the shape of an array and fills in any new elements with repeated or truncated values from the original array.
Example:

```
import numpy as np
arr = np.array([1, 2, 3])
# Resize the array to length 5
resized_arr = np.resize(arr, 5)
 print(resized_arr) - # Output: [1, 2, 3, 1, 2]
```

Reshape: The arr.reshape() method allows you to change the shape of an array similar to np.reshape(). However, it can also modify the original array if possible without creating a new copy.
Example:

```
import numpy as np
 arr = np.array([1, 2, 3, 4, 5, 6])
# Reshape to a 2x3 matrix
arr.reshape((2, 3))
print(arr) - # Output: # [[1, 2, 3], # [4, 5, 6]]
```

Poll Time

Q. Which NumPy function can be used to change the shape of an array?

a. np.resize()

b. np.reshape()

c. np.flatten()

d. np.resize() and np.reshape()

Poll Time

Q. Which NumPy function can be used to change the shape of an array?

a. np.resize()

[✓ **b. np.reshape()**]

c. np.flatten()

d. np.resize() and np.reshape()

Exercise

Exercise: Reshape and Resize

1. Create a NumPy array called arr with the values [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

2. Reshape arr into a 2x5 matrix using the np.reshape() function and assign it to a new variable called reshaped_arr.

3. Print the reshaped_arr.

4. Use the np.resize() function to resize reshaped_arr to have a length of 15 elements. Assign the resized array to a new variable called resized_arr.

5. Print the resized_arr.

Q&A

Summary

- Explored NumPy, a foundational Python library essential for efficient numerical computing in various scientific and data analysis tasks.
- Discussed the core feature of NumPy, the homogeneous, multidimensional arrays that provide efficient handling of large datasets.
- Practiced accessing and manipulating these arrays through techniques such as indexing for individual elements and slicing for subsets.
- Applied a wide range of mathematical functions and operations, including basic arithmetic, exponentiation, logarithms, and trigonometric functions, all element-wise to arrays.
- Utilized the capability to reshape or resize arrays, demonstrating a flexible and efficient approach to data computation.
- Investigated inherent properties of NumPy arrays, such as shape, size, dimensions, and data types, and learned how to access and specify them.
- Executed matrix multiplication or dot product operations, understanding the outcomes as either a scalar value or a new array.

Next Session:
Working with Pandas and Regular Expressions

THANK YOU

Please complete your assessments and review the self-learning content for this session on the **PRISM** portal.

knowledgehut
upGrad

knowledgehut upGrad

Introduction to pandas and Regular Expressions

Pre-requisites

Hope you have gone through the self-learning content for this session on the PRISM portal.

By the End of this Session, you will:

- Understand the fundamental concepts and characteristics of DataFrames, Series, and regular expressions.
- Know how to create, manipulate, and analyze DataFrames and Series using Python and pandas.
- Gain insights into the applications and use cases of DataFrames, Series, and regular expressions in data analysis and text processing tasks.
- Develop practical skills in using regular expressions to search, extract, and manipulate text patterns and apply them to real-world scenarios.

What Have You Learned So Far?

- ✓ Explored NumPy, a foundational Python library for efficient numerical computing in scientific and data analysis tasks.
- ✓ Learned about homogeneous, multidimensional arrays in NumPy, which enable efficient handling of large datasets.
- ✓ Practiced array manipulation techniques such as indexing for accessing individual elements and slicing for extracting subsets of data.
- ✓ Applied various mathematical functions and operations to arrays, including arithmetic, exponentiation, logarithms, and trigonometric functions, all performed element-wise.
- ✓ Utilized the flexibility of NumPy arrays to reshape or resize them, providing efficient data computation capabilities.

Poll Time

Q. Which of the following is a core data structure in pandas?

a. Array

b. List

c. DataFrame

d. Tuple

Poll Time

Q. Which of the following is a core data structure in pandas?

a. Array

b. List

[✓ c. **DataFrame**]

d. Tuple

Introduction to pandas

What are pandas?

- pandas is a Python library that brings data manipulation and analysis to a whole new level.
- Think of pandas as your reliable sidekick in the world of programming and data analysis.
- It provides handy data structures, like DataFrames, that make handling and analyzing data a breeze.
- With pandas, you can conquer data chaos and transform it into a well-organized, insightful adventure.
- It's like having a data superhero by your side, ready to tackle any data-related challenge.

Need for pandas

- pandas, a Python library, is essential for efficient data manipulation and analysis. It simplifies complex tasks like data cleansing and enables seamless integration of datasets.

- pandas provides a wide range of statistical tools for extracting insights and supports data visualization for effective communication.

- It enhances productivity and empowers data professionals with efficient data management and confident decision-making.

Pop Quiz

Q. What is pandas?

a. A Python library for data manipulation and analysis

b. An adorable bear species that loves bamboo

c. A tool for building websites and applications

d. A fictional superhero from a comic book series

Pop Quiz

Q. What is pandas?

a. **A Python library for data manipulation and analysis**

b. An adorable bear species that loves bamboo

c. A tool for building websites and applications

d. A fictional superhero from a comic book series

Reading Various File Formats in pandas

Reading Various File Formats in pandas:

CSV (Comma-Separated Values): CSV files are a popular format for storing tabular data. Pandas provides a simple way to read CSV files using the read_csv() function.

It intelligently detects the delimiter (usually a comma) and loads the data into a pandas DataFrame, allowing easy manipulation and analysis.

Example code:

```
import pandas as pd
# Read a CSV file into a DataFrame
df = pd.read_csv('data.csv')
```

Reading Various File Formats in pandas

Excel: Excel files (.xlsx or .xls) are widely used for data storage. pandas enable reading Excel files using the read_excel() function. It supports reading data from specific sheets, skipping rows, and more.

The data is loaded into a DataFrame, providing a flexible and efficient way to work with Excel data. Example code:

```
import pandas as pd
# Read an Excel file into a DataFrame
df = pd.read_excel('data.xlsx')
```

pandas' ability to effortlessly handle CSV and Excel files simplifies data ingestion, enabling data analysts and scientists to quickly access and analyze data without manual data entry or complex file parsing.

Demo: Reading Various File Formats

Pop Quiz

Q. Which of the following file formats can be read using pandas?

a. CSV

b. Excel

c. JSON

d. All of the above

Pop Quiz

Q. Which of the following file formats can be read using pandas?

a. CSV

b. Excel

c. JSON

[✓ **d. All of the above**]

Creating pandas Series and DataFrames

What are Series?

In pandas, a Series is a one-dimensional labeled data structure that represents a column of data. It is similar to a NumPy array but has an additional index, which provides labels for each element in the Series.

From a List or Array:

You can create a Series by passing a list or an array to the `pd.Series()` function. pandas will automatically assign a numeric index to each element.

```
import pandas as pd data = [10, 20, 30, 40, 50]
series = pd.Series(data)
```

From a Scalar Value: You can create a Series from a single scalar value, specifying the index label. pandas will replicate the scalar value across the Series.

```
import pandas as pd
value = 5
series = pd.Series(value, index=['a', 'b', 'c'])
```

What is a DataFrame?

In pandas, a DataFrame is a two-dimensional labelled data structure that resembles a table or a spreadsheet.

It consists of rows and columns, where each column can contain different data types. The DataFrame provides a powerful and flexible way to analyse and manipulate structured data.

How to Create a DataFrame?

From a Dictionary: You can create a DataFrame from a dictionary, where keys represent column names and values are lists or arrays containing the data for each column.

```
import pandas as pd
data = {'Name': ['John', 'Emma', 'Mike'],
        'Age': [25, 30, 35],
        'City': ['London', 'New York', 'Paris']}

df = pd.DataFrame(data)labeled
```

What is a DataFrame?

From a NumPy Array: You can create a DataFrame from a NumPy array, providing the data and specifying column names.

```
import pandas as pd
import numpy as np

data = np.array([[1, 2], [3, 4], [5, 6]])
columns = ['A', 'B']

df = pd.DataFrame(data, columns=columns)
```

These examples demonstrate the simplicity of creating a DataFrame in pandas. DataFrames offer a versatile and efficient structure for working with tabular data, making them a fundamental tool in data analysis and manipulation.

Demo - Creating pandas Series and DataFrames

Common Operations on Series and DataFrames

- Performing operations on Series and DataFrames is a fundamental aspect of data analysis in pandas. Here are some common operations:

- **Accessing Data:** You can access data in a Series using the index labels or numerical positions. For DataFrames, you can access specific columns or rows sing column names or row indices.

- **Data Manipulation:** pandas provide powerful tools for data manipulation. You can filter data based on specific conditions, perform arithmetic operations between columns or Series, apply functions element-wise, and more.

- **Handling Missing Data:** pandas offers methods to handle missing data, such as dropna() to remove missing values and fillna() to fill missing values with specific values or strategies.

Common Operations on Series and DataFrames

- Aggregation and Summary Statistics: You can calculate various summary statistics on Series and DataFrames, such as mean, median, sum, min, max, and count. pandas provide convenient methods like mean(), sum(), min(), max(), and count() for these calculations.

- **Data Visualization:** pandas integrates with popular visualization libraries like Matplotlib and Seaborn to create insightful plots and charts. You can visualize data distributions, trends, relationships, and more directly from Series and DataFrames.

- These are just a few examples of the common operations you can perform on Series and DataFrames in pandas. pandas' extensive functionality allows for efficient data manipulation, exploration, and analysis, making it a go-to tool for data professionals.

.loc and .iloc Functions

loc:

- The **loc** function in pandas is used to access and modify data in a DataFrame using labels.
- It provides a way to select rows and columns based on their label values.
- The syntax for using **loc** is **df.loc[row_label, column_label]**, where **df** is the DataFrame object.
- You can specify single labels, lists of labels, or label ranges to select the desired data.
- The labels can be either row labels, column labels, or a combination of both.
- Example: **df.loc[2, 'column_name']** selects the value at row 2 and column 'column_name' in the DataFrame.

.loc and .iloc Functions

iloc:

- The **iloc** function in pandas is used to access and modify data in a DataFrame using integer-based indexing.
- It provides a way to select rows and columns based on their integer positions.
- The syntax for using **iloc** is **df.iloc[row_index, column_index]**, where **df** is the DataFrame object.
- You can specify single indices, lists of indices, or index ranges to select the desired data.
- The indices start from 0 for the first row or column.
- Example: **df.iloc[0, 2]** selects the value at the first row and third column in the DataFrame.

Reset Index Function

reset_index:

- The **reset_index** method in pandas is used to reset the index of a DataFrame or Series to the default integer index.
- It generates a new index with consecutive integers starting from 0 and assigns it to the DataFrame or Series.
- The original index becomes a new column in the DataFrame or Series.
- By default, the **reset_index** method modifies the DataFrame in place and returns a new DataFrame with the reset index.
- The syntax for using **reset_index** is **df.reset_index()**, where **df** is the DataFrame object.
- Example: **df.reset_index()** resets the index of the DataFrame **df** to the default integer index.

Parameters of **reset_index**:

- **drop**: It is a Boolean parameter that specifies whether to drop the current index column or not. By default, it is set to **False**. If set to **True**, the current index column is dropped, and the DataFrame or Series is reset with the default integer index.

Reset Index Function

Parameters of reset_index:

- **drop:** It is a Boolean parameter that specifies whether to drop the current index column or not. By default, it is set to **False**. If set to **True**, the current index column is dropped, and the DataFrame or Series is reset with the default integer index.

- **inplace**: It is a Boolean parameter that determines whether to modify the DataFrame in place or return a new DataFrame with the reset index. By default, it is set to **False**, which means a new DataFrame with the reset index is returned. If set to **True**, import pandas as pd.

Reset Index Function

```
# Create a sample DataFrame
data = {
    'Name': ['John', 'Alice', 'Bob'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}
df = pd.DataFrame(data)

# Set 'Name' column as the index
df.set_index('Name', inplace=True)

# Reset the index
df_reset = df.reset_index()

# Display the original DataFrame and the reset index DataFrame
print("Original DataFrame:")
print(df)
print("\nReset Index DataFrame:")
print(df_reset)
 the DataFrame is modified in place, and None is returned.
```

Groupby

groupby:

- The groupby method in pandas is used to group data in a DataFrame based on one or more columns.
- It allows you to split the data into groups based on a criterion defined by the column(s) specified.
- The syntax for using groupby is df.groupby('column_name'), where df is the DataFrame object.
- You can also group by multiple columns by passing a list of column names to the groupby method.
- Example: df.groupby('column_name') groups the data in the DataFrame df based on the unique values in the 'column_name' column.

Aggregation

Aggregation:

- After applying groupby, you can perform various operations on the grouped data, such as aggregation, transformation, or filtering.
- Common operations include calculating summary statistics like mean, sum, count, etc., within each group.
- You can use methods like sum(), mean(), count(), min(), max(), etc., on the grouped DataFrame to perform aggregations.

Group by

```
import pandas as pd

# Create a sample DataFrame
data = {
    'Category': ['A', 'A', 'B', 'B', 'A', 'B'],
    'Value': [10, 15, 20, 25, 30, 35]
}
df = pd.DataFrame(data)

# Group the data by the 'Category' column
grouped = df.groupby('Category')

# Calculate the sum of values within each group
sum_by_category = grouped['Value'].sum()

# Display the grouped and summed data
print(sum_by_category)
```

Pop Quiz

Q. How do you access the first few rows of a DataFrame?

a. `head()`

b. `tail()`

c. `first()`

d. `top()`

Pop Quiz

Q. How do you access the first few rows of a DataFrame?

[✓ **a. head()**]

b. tail()

c. first()

d. top()

Q&A

Pop Quiz

Q. Which method is used to calculate the descriptive statistics of DataFrame?

a. sum()

b. describe()

c. unique()

d. sort_values()

Pop Quiz

Q. Which method is used to calculate the descriptive statistics of DataFrame?

a. `sum()`

[✓ **b. `describe()`**]

c. `unique()`

d. `sort_values()`

Introduction to Regular Expression

Introduction to Regular Expression

- Regular expressions, often abbreviated as regex or regexp, are powerful tools used for pattern matching and manipulating text. They are widely used in various programming languages, including Python, to search, extract, and modify text based on specific patterns.

  Key Concepts:
  Pattern Matching:Regular expressions enable us to search for specific patterns within text.Patterns can be simple, such as matching a specific word or character, or complex, involving multiple conditions and constraints.

Use Cases of Regular Expression

- Regular expressions (regex) have a wide range of use cases in various fields. Here are some common applications of regular expressions:

- **Text Search and Extraction:**
  Pattern matching: Regular expressions are commonly used to search for specific patterns within text. For example, finding all email addresses or phone numbers in a document.

- **Data extraction:**
  Regular expressions can be used to extract specific data from text, such as retrieving dates, URLs, or values from a structured document.

Introduction to Python’s re Module

- The re module in Python provides support for working with regular expressions.
- Regular expressions (regex) are powerful tools for pattern matching and manipulation of text data.
- The re module allows you to search, extract, and manipulate specific patterns within strings.
- It provides functions like re.match(), re.search(), and re.findall() for pattern matching operations.

Introduction to Python’s re Module

- Regular expressions use a combination of ordinary characters and meta-characters to define patterns.
- Meta-characters include symbols like . (dot), * (asterisk), + (plus), ? (question mark), and [] (square brackets), among others.
- These meta-characters enable you to match specific characters, repetitions, alternatives, or character classes in the text.
- The re module also supports capturing groups, which allow you to extract specific parts of a matched pattern.

Re.match and re.search

re.match(pattern, string):

- The **re.match()** function searches for the specified pattern at the beginning of the string.
- It returns a match object if the pattern is found, or **None** otherwise.
- It checks for a match only at the beginning of the string.
- Example: **re.match('abc', 'abcdef')** matches 'abc' at the beginning of the string 'abcdef'.

Re.match and re.search

re.search(pattern, string):

- The **re.search()** function searches for the specified pattern anywhere within the string.
- It returns a match object if the pattern is found, or **None** otherwise.
- It scans the entire string to find the first occurrence of the pattern.
- Example: **re.search('123', 'a123b456')** matches '123' within the string 'a123b456'.

Re.sub and re.findall

re.sub(pattern, replacement, string):

- The **re.sub()** function replaces occurrences of the pattern in the string with the specified replacement.
- It returns a new string with the replacements made.
- It replaces all occurrences of the pattern by default, unless the **count** parameter is specified.
- Example: **re.sub('\d+', 'NUM', 'apple 123 orange 456')** replaces all digits with 'NUM' in the string.

Re.sub and re.findall

re.findall(pattern, string):

- The **re.findall()** function returns all non-overlapping matches of the pattern in the string as a list.
- It searches the entire string and returns all matches.
- Example: **re.findall('\d+', 'apple 123 orange 456')** returns a list with ['123', '456'].

Pop Quiz

Q. Why are regular expressions required?

a. To perform mathematical calculations

b. To manipulate images and graphics

c. To search and manipulate text patterns

d. To create user interfaces

Pop Quiz

Q. Why are regular expressions required?

a. To perform mathematical calculations

b. To manipulate images and graphics

[✓ **c. To search and manipulate text patterns**]

d. To create user interfaces

Meta Characters

Role of Meta Characters in Regular Expression

Meta characters are special characters in regular expressions that have a specific role and meaning. They provide enhanced functionality and flexibility when working with pattern matching in regular expressions. Examples :

- **. (dot):**

Matches any single character except a newline. Example: "The cat sat on the mat."
In this example, the dot character '.' can be used in a regular expression pattern to match any single character. If we use the pattern "c.t", it will match "cat" because the dot acts as a placeholder for any character in that position.

- ***** (asterisk):**

Matches zero or more occurrences of the preceding character or group. Example: "I have read the book multiple times." - The asterisk matches zero or more occurrences, so the pattern "mul*tiple" would match both "multiple" and "muptiple" (zero occurrences of 'l').

Role of Meta Characters in Regular Expression

- **+ (plus):**

Matches one or more occurrences of the preceding character or group. Example: "She laughed and laughed and laughed." - The plus matches one or more occurrences, so the pattern "laugh+ed" would match "laughed", "laugheded", "laughededed", and so on.

- **? (question mark):**

Matches zero or one occurrence of the preceding character or group. Example: "Is the color blue or green?" - The question mark matches zero or one occurrence, so the pattern "colo?r" would match both "color" (zero occurrences of 'o') and "colour" (one occurrence of 'o').

Demo - Meta Characters

Introduction to Special Sequences

[redacted]

Introduction to Special Sequences

Special sequences are predefined patterns in regular expressions that represent common types of characters or character classes.

They provide a convenient way to match specific types of characters without explicitly listing all possible characters.

Here are some commonly used special sequences in regular expressions:

\d:
Matches any digit (0-9).
Equivalent to the character class [0-9].
\D:
Matches any non-digit character.
Equivalent to the character class [^0-9].
\w:
Matches any alphanumeric character (letter, digit, or underscore).
Equivalent to the character class [a-zA-Z0-9_].
\W:
Matches any non-alphanumeric character.
Equivalent to the character class [^a-zA-Z0-9_].
\s:
Matches any whitespace character (space, tab, newline).
Equivalent to the character class [\t\n\r\f\v].

Introduction to Special Sequences

```
import re

# Example text
text = "The phone number is 123-456-7890."

# Pattern to match a phone number
pattern = r"\d{3}-\d{3}-\d{4}"

# Use re.findall() to find all occurrences of the pattern in the text
matches = re.findall(pattern, text)

# Print the matched phone numbers
print(matches)
```

Pop Quiz

Q. Which special sequence in regular expressions matches any non-digit character?

a. \d

b. \D

c. \w

d. \W

Pop Quiz

Q. Which special sequence in regular expressions matches any non-digit character?

a. \d

[✓ **b. \D**]

c. \w

d. \W

Q&A

Activity 1

Create a Python program that simulates a basic inventory management system. Write a function named "updateInventory" that takes two parameters: an existing inventory dictionary and a new inventory dictionary. The function should update the existing inventory with the quantities from the new inventory and return the updated inventory dictionary.

Here are the requirements for the function:

- The inventory dictionaries have the following structure: {'item_name': quantity}.
- If an item exists in both the existing inventory and the new inventory, the function should add the quantities together.
- If an item exists only in the new inventory, the function should add it to the existing inventory.
- If an item exists only in the existing inventory, it should remain unchanged in the updated inventory.
- The function should handle both integer and float quantities.

Example:

```
existing_inventory = {'item1': 10, 'item2': 5, 'item3': 3} new_inventory = {'item2': 2, 'item4': 7}
updated_inventory = updateInventory(existing_inventory, new_inventory) print(updated_inventory)
# Output: {'item1': 10, 'item2': 7, 'item3': 3, 'item4': 7}
```

Activity 2

You are building an e-commerce application, and you need to calculate the total price of a customer's shopping cart. Write a Python function named "calculateCartTotal" that takes a list of items in the customer's cart as input and returns the total price of the cart. Each item in the cart is represented as a dictionary with the following keys: 'name' (string), 'price' (float), and 'quantity' (integer). The function should calculate the total price by multiplying the price of each item by its quantity and summing up the results.

```
cart = [ {'name': 'Product A', 'price': 10.99, 'quantity': 2}, {'name':
'Product B', 'price': 5.49, 'quantity': 3}, {'name': 'Product C', 'price':
2.99, 'quantity': 1} ]
```

Activity 3

Create a Python program that uses regular expressions to validate email addresses. Ask the student to write a function called "validate_email" that takes an email address as input and returns True if the email address is valid and False otherwise.

Here are the requirements for a valid email address:

- The email address should contain a username, followed by the "@" symbol, followed by a domain name, and finally, a top-level domain (TLD).

- The username can contain alphanumeric characters, as well as dots (.), underscores (_), and dashes (-). It should have at least one character before the "@" symbol.

- The domain name can contain alphanumeric characters, as well as dots (.) and dashes (-). It should have at least one character after the "@" symbol and before the TLD.

- The TLD can contain alphabetic characters only and should have at least two characters.

Activity 3

- **For example, the following email addresses should be considered valid:**
  - john.doe@example.com
  - jane_doe123@example.co.uk

- **On the other hand, the following email addresses should be considered invalid:**
  - john.doe@example (missing TLD)
  - jane_doe123@ (missing domain name)
  - @example.com (missing username)

Summary

- Introduced the pandas library and provided powerful tools for data analysis and manipulation in Python.
- Implemented two essential data structures, DataFrame and Series in pandas to represent tabular data and one-dimensional arrays with labeled indexes, respectively.
- Facilitated various data manipulation tasks such as data cleaning, transformation, aggregation, and visualization, enabling efficient data analysis workflows.
- Associated each element in the Series with an index, enabling label-based indexing and retrieval of data.
- Worked on series-provided functionalities for performing operations, calculations, and transformations on data, including filtering, sorting, and aggregating.
- Utilized regular expressions (regex) as a powerful tool in Python for pattern matching and manipulation of text data, allowing for tasks such as searching, extracting, and manipulating specific patterns within strings.

Session Feedback

Next Session:
Visualization in Python

THANK YOU

Please complete your assessments and review the self-learning content for this session on the **PRISM** portal.

knowledgehut
upGrad