Data Engineering Day 02

The credit for this course goes to Coursera. Click More

Link for Azure data Engineer.

Python For AI, Data Science and Development

Pandas:

Data Structures: Pandas offers two primary data structures - Data Frame and Series.

1. A Data Frame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).
2. A Series is a one-dimensional labeled array, essentially a single column or row of data.

Data Import and Export: Pandas makes it easy to read data from various sources, including CSV files, Excel spreadsheets, SQL databases, and more. It can also export data to these formats, enabling seamless data exchange.

Data Merging and Joining: You can combine multiple Data Frames using methods like merging and join, like SQL operations, to create more complex datasets from different sources.

Efficient Indexing: Pandas provides efficient indexing and selection methods, allowing you to access specific rows and columns of data quickly.

Custom Data Structures: You can create custom data structures and manipulate data in ways that suit your specific needs, extending Pandas' capabilities.

Importing Pandas:

Import Pandas using the import command, followed by the library's name.
Commonly, Pandas is imported as pd for brevity in code.

```
1  import pandas as pd
```

Data Loading:

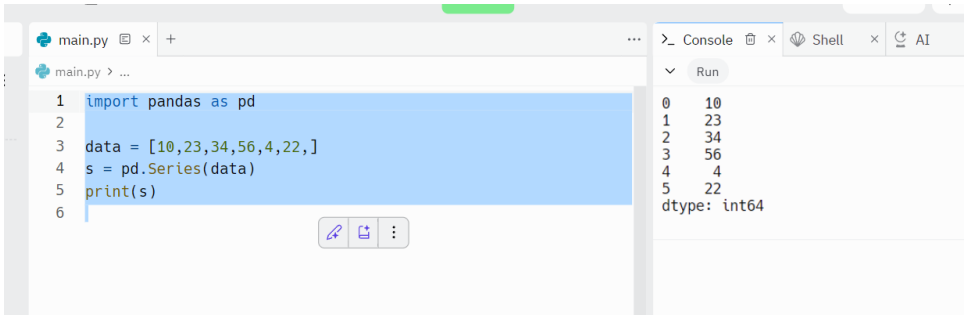
- Pandas can be used to load data from various sources, such as CSV and Excel files.
- The read_csv function is used to load data from a CSV file into a Pandas DataFrame.

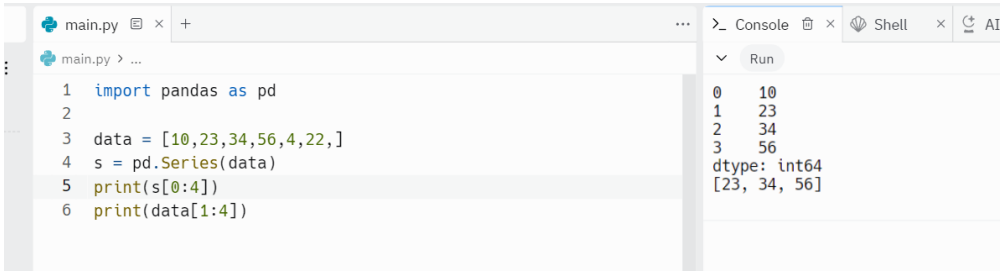
To read a CSV (Comma-Separated Values) file in Python using the Pandas library, you can use the pd.read_csv() function. Here's the syntax to read a CSV file:

```
1  import pandas as pd
2
3  # Read the CSV file into a DataFrame
4  df = pd.read_csv('your_file.csv')
```

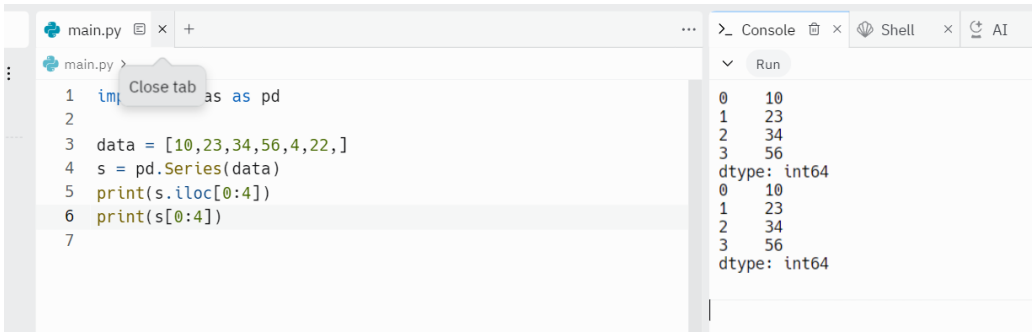
Here's a basic example of creating a Series in Pandas:

```python
import pandas as pd

# Create a Series from a list
data = [10, 20, 30, 40, 50]
s = pd.Series(data)

print(s)
```





What is the difference between writing s.iloc[0:4] and s[0:4]?



Series Attributes and Methods

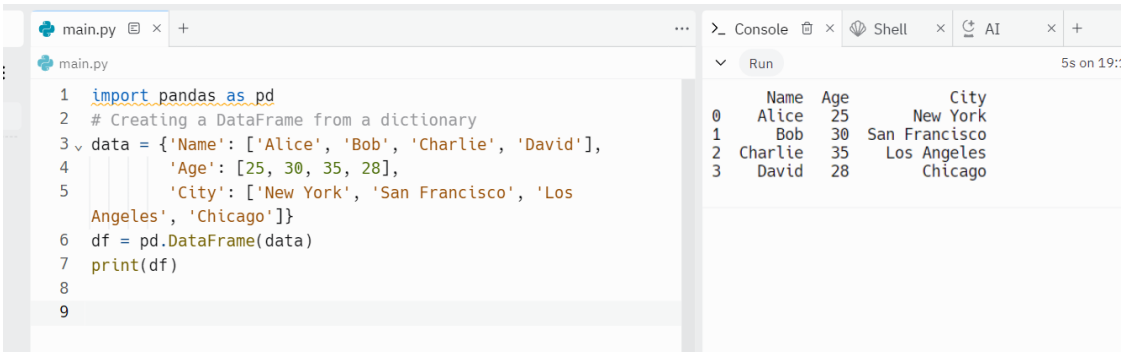
Pandas Series come with various attributes and methods to help you manipulate and analyze data effectively. Here are a few essential ones:

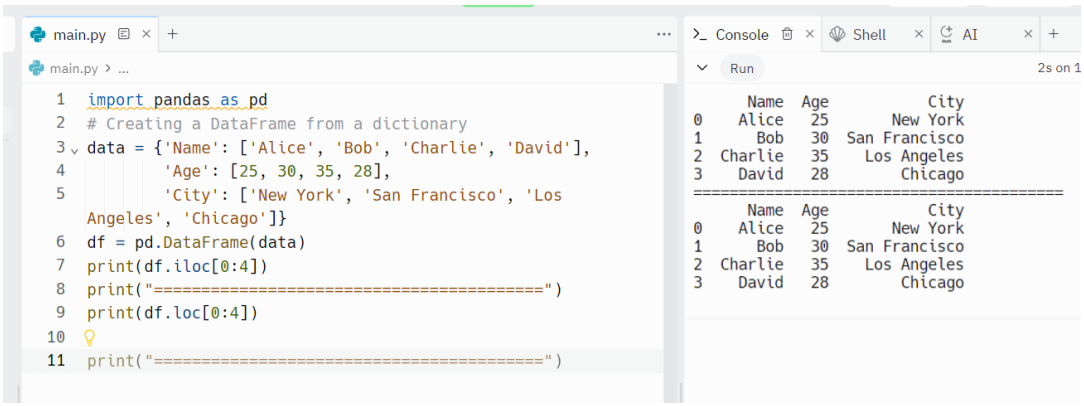
- **values**: Returns the Series data as a NumPy array.
- **index**: Returns the index (labels) of the Series.
- **shape**: Returns a tuple representing the dimensions of the Series.
- **size**: Returns the number of elements in the Series.
- **mean(), sum(), min(), max()**: Calculate summary statistics of the data.
- **unique(), nunique()**: Get unique values or the number of unique values.
- **sort_values(), sort_index()**: Sort the Series by values or index labels.
- **isnull(), notnull()**: Check for missing (NaN) or non-missing values.
- **apply()**: Apply a custom function to each element of the Series.

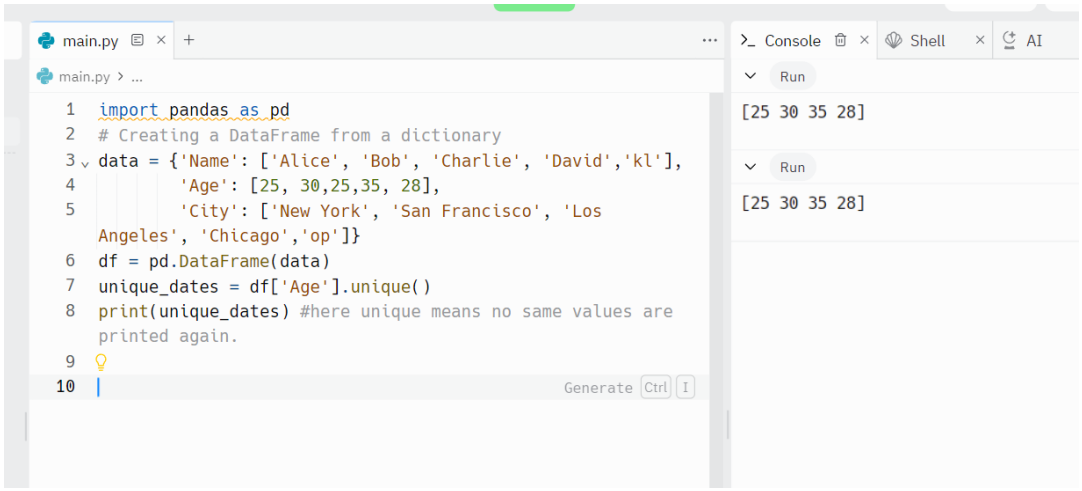
Creating DataFrames from Dictionaries:

DataFrames can be created from dictionaries, with keys as column labels and values as lists representing rows.

```python
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 28],
        'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}

df = pd.DataFrame(data)

print(df)
```







For further information, click me

NumPy:

For more information, click me

What is Numpy?

NumPy is a Python library used for working with arrays, linear algebra, fourier transform, and matrices. NumPy stands for Numerical Python and it is an open source project. The array object in NumPy is called **ndarray**, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

NumPy is usually imported under the np alias.

It's usually fixed in size and each element is of the same type. We can cast a list to a numpy array by first importing `numpy` :

[1]:
```
# import numpy library

import numpy as np
```

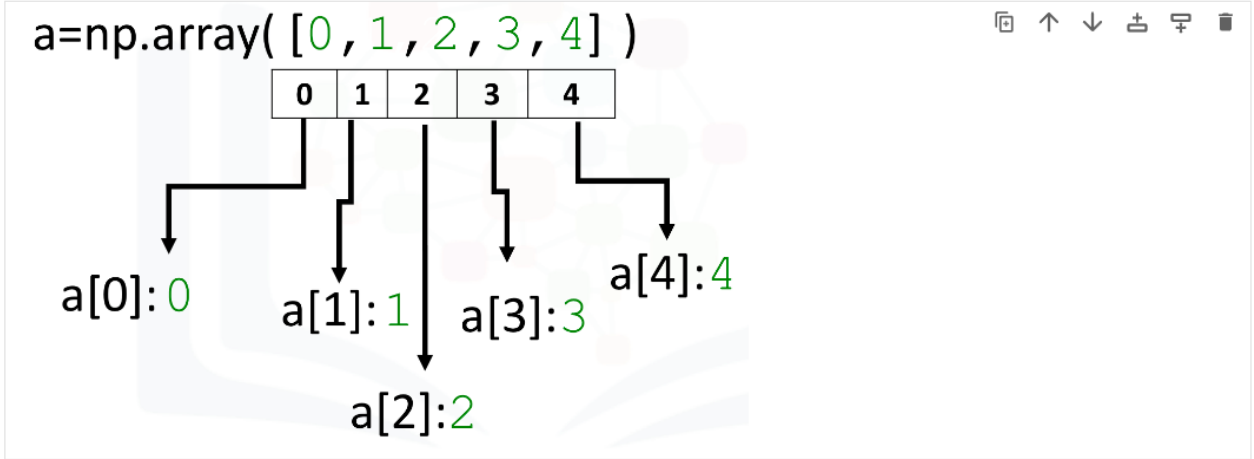
We then cast the list as follows:

[5]:
```
# Create a numpy array

a = np.array([0, 1, 2, 3, 4])
print(a)
```

```
[0 1 2 3 4]
```

Each element is of the same type, in this case integers:

Each element is of the same type, in this case integers:



```
b.dtype
```

If we examine the attribute dtype we see float 64, as the elements are not integers:

Assign value

We can change the value of the array. Consider the array `c`:

[12]:
```
# Create numpy array

c = np.array([20, 1, 2, 3, 4])
c
```

[12]:
```
array([20,  1,  2,  3,  4])
```

We can change the first element of the array to 100 as follows:

[13]:
```
# Assign the first element to 100

c[0] = 100
c
```

[13]:
```
array([100,   1,   2,   3,   4])
```

We can change the 5th element of the array to 0 as follows:

[ ]:
```
# Assign the 5th element to 0

c[4] = 0
c
```

Slicing

Like lists, we can slice the numpy array. Slicing in python means taking the elements from the given index to another given index.

We pass slice like this: [start:end].The element at end index is not being included in the output.

We can select the elements from 1 to 3 and assign it to a new numpy array `d` as follows:

[ ]:
```
# Slicing the numpy array

d = c[1:4]
d
```

We can assign the corresponding indexes to new values as follows:

[ ]:
```
# Set the fourth element and fifth element to 300 and 400

c[3:5] = 300, 400
c
```

We can also define the steps in slicing, like this: [start:end:step].

[19]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

```
[2 4]
```

If we don't pass start its considered 0

[20]:
```
print(arr[:4])
```

```
[1 2 3 4]
```

If we don't pass end it considers till the length of array.

[21]:
```
print(arr[4:])
```

```
[5 6 7]
```

If we don't pass step its considered 1

[ ]:
```
print(arr[1:5:])
```

Numpy Array Operations

You could use arithmetic operators directly between NumPy arrays

Array Addition

Consider the numpy array `u`:

[46]:
```
u = np.array([1, 0])
u
```

[46]:
```
array([1, 0])
```

Consider the numpy array `v`:

[47]:
```
v = np.array([0, 1])
v
```

[47]:
```
array([0, 1])
```

We can add the two arrays and assign it to z:

[48]:
```
# Numpy Array Addition

z = np.add(u, v)
z
```

[48]:
```
array([1, 1])
```

The operation is equivalent to vector addition:

Array Subtraction

Consider the numpy array a:

[56]:
```
a = np.array([10, 20, 30])
a
```

[56]:
```
array([10, 20, 30])
```

Consider the numpy array b:

[57]:
```
b = np.array([5, 10, 15])
b
```

[57]:
```
array([ 5, 10, 15])
```

We can subtract the two arrays and assign it to c:

[ ]:
```
c = np.subtract(a, b)

print(c)
```

Array Multiplication

Consider the vector numpy array `y`:

[59]:
```
# Create a numpy array

x = np.array([1, 2])
x
```

[59]:
```
array([1, 2])
```

[60]:
```
# Create a numpy array

y = np.array([2, 1])
y
```

[60]:
```
array([2, 1])
```

We can multiply every element in the array by 2:

[61]:
```
# Numpy Array Multiplication

z = np.multiply(x, y)
z
```

[61]:
```
array([2, 2])
```

This is equivalent to multiplying a vector by a scaler:

Array Division

Consider the vector numpy array a:

```
[ ]: a = np.array([10, 20, 30])
     a
```

Consider the vector numpy array b:

```
[ ]: b = np.array([2, 10, 5])
     b
```

We can divide the two arrays and assign it to c:

```
[ ]: c = np.divide(a, b)
     c
```

Try it yourself

Perform division operation on the given numpy array arr1 and arr2:

```
[ ]: arr1 = np.array([10, 20, 30, 40, 50, 60])
     arr2 = np.array([3, 5, 10, 8, 2, 33])

     # Enter your code here
```

▶ Click here for the solution

```
[67]: #Elements of Y
      print(Y[0])
      print(Y[1])
```

```
3
2
```

We are performing the dot product which is shown as below