Explorando datos con Python

Una parte importante del rol de un científico de datos es explorar, analizar y visualizar datos. Hay muchas herramientas y lenguajes de programación que pueden usar para hacer esto. Uno de los enfoques más populares es usar cuadernos Jupyter (como este) y Python.

Python es un lenguaje de programación flexible que se usa en una amplia gama de escenarios, desde aplicaciones web hasta programación de dispositivos. Es extremadamente popular en la comunidad de ciencia de datos y aprendizaje automático debido a los muchos paquetes que admite para el análisis y la visualización de datos.

En este cuaderno, exploraremos algunos de estos paquetes y aplicaremos técnicas básicas para analizar datos. Esto no pretende ser un ejercicio completo de programación de Python o incluso una inmersión profunda en el análisis de datos. Más bien, pretende ser un curso intensivo sobre algunas de las formas comunes en que los científicos de datos pueden usar Python para trabajar con datos.

Explorando arreglos de datos con NumPy. Comencemos mirando algunos datos simples.
Supongamos que un profesor universitario toma una muestra de las calificaciones de los estudiantes de una clase para analizarlas.
Ejecute el código en la celda de abajo haciendo clic en el botón ▶ Ejecutar para ver los datos.

```
data = [50,50,47,97,49,3,53,42,26,74,82,62,37,15,70,27,36,35,48,52,63,64]
print(data)
```

Los datos se cargaron en una estructura de lista de Python, que es un buen tipo de datos para la manipulación general de datos, pero no está optimizado para el análisis numérico. Para eso, vamos a utilizar el paquete NumPy, que incluye funciones y tipos de datos específicos para trabajar con Numbers en Python.

Ejecute la celda a continuación para cargar los datos en una matriz NumPy.

```
import numpy as np

grades = np.array(data)
print(grades)
```

En caso de que se esté preguntando acerca de las diferencias entre una lista y una matriz NumPy, comparemos cómo se comportan estos tipos de datos cuando los usamos en una expresión que los multiplica por 2.

```
print (type(data),'x 2:', data * 2)
print('---')
print (type(grades),'x 2:', grades * 2)
```

Analicemos los resultados:

Tenga en cuenta que multiplicar una lista por 2 crea una nueva lista del doble de longitud con la secuencia original de elementos de la lista repetida. Multiplicar una matriz NumPy, por otro lado, realiza un cálculo por elementos en el que la matriz se comporta como un vector, por lo que terminamos con una matriz del mismo tamaño en la que cada elemento se ha multiplicado por 2.

La conclusión clave de esto es que las matrices NumPy están diseñadas específicamente para admitir operaciones matemáticas en datos numéricos, lo que las hace más útiles para el análisis de datos que una lista genérica.

Es posible que haya notado que el tipo de clase para la matriz NumPy anterior es numpy.ndarray. El nd indica que se trata de una estructura que puede constar de múltiples dimensiones. (Puede tener n dimensiones). Nuestra instancia específica tiene una sola dimensión de calificaciones de los estudiantes.

Ejecute la celda de abajo para ver la forma de la matriz.

```
grades.shape
```

La forma confirma que esta matriz tiene solo una dimensión, que contiene 22 elementos. (Hay 22 grados en la lista original). Puede acceder a los elementos individuales de la matriz por su posición ordinal de base cero. Obtengamos el primer elemento (el que está en la posición 0).

```
grades[0]
```

Ahora que conoce la matriz NumPy, es hora de realizar un análisis de los datos de las calificaciones.

Puede aplicar agregaciones a los elementos de la matriz, así que encontremos la calificación promedio simple (en otras palabras, el valor de la calificación media).

```
grades.mean()
```

Entonces, la calificación media es de alrededor de 50, más o menos en el medio del rango posible de 0 a 100.

Agreguemos un segundo conjunto de datos para los mismos estudiantes. Esta vez, registraremos la cantidad típica de horas por semana que dedicaron a estudiar.

```
# Define an array of study hours
study_hours = [10.0,11.5,9.0,16.0,9.25,1.0,11.5,9.0,8.5,14.5,15.5,
               13.75,9.0,8.0,15.5,8.0,9.0,6.0,10.0,12.0,12.5,12.0]

# Create a 2D array (an array of arrays)
student_data = np.array([study_hours, grades])

# display the array
student_data
```

Ahora los datos consisten en una matriz bidimensional: una matriz de matrices. Veamos su forma.

```
# Show shape of 2D array
student_data.shape
```

La matriz student_data contiene dos elementos, cada uno de los cuales es una matriz que contiene 22 elementos.

Para navegar por esta estructura, debe especificar la posición de cada elemento en la jerarquía. Entonces, para encontrar el primer valor en la primera matriz (que contiene los datos de las horas de estudio), puede usar el siguiente código.

```
# Show the first element of the first element
student_data[0][0]
```

Ahora tiene una matriz multidimensional que contiene tanto el tiempo de estudio del estudiante como la información de calificaciones, que puede usar para comparar el tiempo de estudio con la calificación de un estudiante.

```
# Get the mean value of each sub-array
avg_study = student_data[0].mean()
avg_grade = student_data[1].mean()

print('Average study hours: {:.2f}\nAverage grade: {:.2f}'.format(avg_study, avg_grade))
```

Explorando datos tabulares con Pandas

NumPy proporciona muchas de las funciones y herramientas que necesita para trabajar con números, como matrices de valores numéricos. Sin embargo, cuando comienza a manejar tablas de datos bidimensionales, el paquete Pandas ofrece una estructura más conveniente para trabajar: el DataFrame.

Ejecute la siguiente celda para importar la biblioteca de Pandas y cree un DataFrame con tres columnas. La primera columna es una lista de nombres de estudiantes, y la segunda y tercera columnas son las matrices NumPy que contienen el tiempo de estudio y los datos de calificación.

```
import pandas as pd

df_students = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky', 'Frederic', 'Jimmie', 'Rhonda',
'Giovanni', 'Francesca', 'Rajab', 'Naiyana', 'Kian', 'Jenny', 'Jakeem','Helena','Ismat','Anila','Skye','Daniel','Aisha'],

                                'StudyHours':student_data[0],

                                'Grade':student_data[1]})

df_students
```

Tenga en cuenta que, además de las columnas que especificó, el DataFrame incluye un índice para identificar de forma única cada fila. Podríamos haber especificado el índice explícitamente y asignado cualquier tipo de valor apropiado (por ejemplo, una dirección de correo electrónico). Sin embargo, debido a que no especificamos un índice, se creó uno con un valor entero único para cada fila.

Encontrar y filtrar datos en un DataFrame

Puede usar el método loc de DataFrame para recuperar datos para un valor de índice específico, como este.

```
# Get the data for index value 5
df_students.loc[5]
```

También puede obtener los datos en un rango de valores de índice, como este:

```
# Get the rows with index values from 0 to 5
df_students.loc[0:5]
```

Además de poder usar el método loc para buscar filas según el índice, puede usar el método iloc para buscar filas según su posición ordinal en el DataFrame (independientemente del índice):

```
# Get data in the first five rows
df_students.iloc[0:5]
```

Mire cuidadosamente los resultados de iloc[0:5] y compárelos con los resultados de loc[0:5] que obtuvo anteriormente. *¿Puedes ver la diferencia?*

El método loc devolvió filas con etiqueta de índice en la lista de valores de 0 a 5, que incluye 0, 1, 2, 3, 4 y 5 (seis filas). Sin embargo, el método iloc devuelve las filas en las posiciones incluidas en el rango de 0 a 5. Dado que los rangos de enteros no incluyen el valor del límite superior, esto incluye las posiciones 0, 1, 2, 3 y 4 (cinco filas).

iloc identifica valores de datos en un DataFrame por posición, que se extiende más allá de las filas a las columnas. Entonces, por ejemplo, puede usarlo para encontrar los valores de las columnas en las posiciones 1 y 2 en la fila 0, así:

```
df_students.iloc[0,[1,2]]
```

Volvamos al método loc y veamos cómo funciona con las columnas. Recuerde que loc se usa para ubicar elementos de datos en función de valores de índice en lugar de posiciones. En ausencia de una columna de índice explícita, las filas en nuestro DataFrame se indexan como valores enteros, pero las columnas se identifican por nombre:

```
df_students.loc[0,'Grade']
```

Aquí hay otro truco útil. Puede usar el método loc para encontrar filas indexadas en función de una expresión de filtrado que hace referencia a columnas nombradas distintas del índice, como esta:

```
df_students.loc[df_students['Name']=='Aisha']
```

En realidad, no necesita usar explícitamente el método loc para hacer esto. Simplemente puede aplicar una expresión de filtrado de DataFrame, como esta:

```
df_students[df_students['Name']=='Aisha']
```

Y en buena medida, puede lograr los mismos resultados utilizando el método de consulta de DataFrame, como este:

```
df_students.query('Name=="Aisha"')
```

Los tres ejemplos anteriores subrayan una verdad confusa sobre trabajar con Pandas. A menudo, hay varias formas de lograr los mismos resultados. Otro ejemplo de esto es la forma en que se refiere a un nombre de columna de DataFrame. Puede especificar el nombre de la columna como un valor de índice con nombre (como en los ejemplos de df_students['Name'] que hemos visto hasta ahora), o puede usar la columna como una propiedad del DataFrame, así:

```
df_students[df_students.Name == 'Aisha']
```

Manejo de valores faltantes

Bibliografía

https://learn.microsoft.com/es-es/training/modules/explore-analyze-data-with-python/3-exercise-explore-data