Python - pandas: La biblioteca que marcó un hito en análisis de datos

Wes McKinney (Autor original)

2011 (Año de publicación en inglés)

En este documento hablaremos de **pandas**, una biblioteca de Python de estructuras y herramientas de datos enriquecidos para trabajar con conjuntos de datos estructurados comunes a estadísticas, finanzas, ciencias sociales y muchos otros campos. La biblioteca proporciona rutinas integradas e intuitivas para realizar manipulaciones y análisis de datos comunes en dichos conjuntos de datos. Su objetivo es ser la capa fundamental para el futuro de la informática estadística en Python. Sirve como un fuerte complemento para la pila científica existente de Python, mientras implementa y mejora los tipos de herramientas de manipulación de datos que se encuentran en otros lenguajes de programación estadística como **R**. Además de detallar su diseño y características **pandas**, discutiremos futuras oportunidades de trabajo y crecimiento para aplicaciones de análisis de datos y estadísticas en lenguaje Python.

Introducción

Python se usa cada vez más en aplicaciones científicas tradicionalmente dominadas por **[R]**, **[MATLAB**], **[Stata]**, **[SAS]**, entre otros entornos de investigación comerciales o de código abierto. La madurez y la estabilidad de las bibliotecas numéricas fundamentales (**[NumPy]**, **[SciPy]** y otras), la calidad de la documentación y la disponibilidad de distribuciones de "fregadero de cocina" (**[EPD]**, **[Pythonxy]**) han recorrido un largo camino haciendo que Python sea accesible y conveniente para una amplia audiencia. Además, **[matplotlib]** integrado con **[IPython]** proporciona un entorno interactivo de investigación y desarrollo con visualización de datos adecuada para la mayoría de los usuarios. Sin embargo, la adopción de Python para el modelado estadístico aplicado ha sido relativamente lenta en comparación con otras áreas de la ciencia computacional.

Una cuestión importante para los posibles programadores estadísticos de Python en el pasado ha sido la falta de bibliotecas que implementen modelos estándar y un marco coherente para especificar modelos. Sin embargo, en los últimos años ha habido nuevos desarrollos significativos en econometría ([StaM]), estadísticas bayesianas ([PyMC]) y aprendizaje automático ([SciL]), entre otros campos. Sin embargo, todavía es difícil para muchos estadísticos elegir Python sobre R dada la naturaleza específica del dominio del lenguaje R y la amplitud de las bibliotecas de código abierto bien examinadas disponibles para los usuarios de R ([CRAN]). A pesar de este obstáculo, creemos que el lenguaje Python y las bibliotecas y herramientas actualmente disponibles pueden aprovecharse para hacer de Python un entorno superior para el análisis de datos y la computación estadística.

Otro problema que impidió a muchos usar Python en el pasado para aplicaciones de análisis de datos ha sido la falta de estructuras de datos enriquecidas con manejo integrado de metadatos. Por metadatos entendemos etiquetar información sobre puntos de datos. Por ejemplo, una tabla u hoja de cálculo de datos probablemente tendrá etiquetas para las columnas y posiblemente también para las filas. Alternativamente, algunas columnas de una tabla pueden usarse para agrupar y agregar datos en una tabla dinámica o de contingencia. En el caso de un conjunto de datos de series

de tiempo, las etiquetas de fila podrían ser marcas de tiempo. A menudo es necesario tener la información de etiquetado disponible para permitir que muchos tipos de manipulaciones de datos, como fusionar conjuntos de datos o realizar una operación de agregación o "agrupar por", se expresen de manera intuitiva y concisa. Los lenguajes de bases de datos específicos de dominio como SQL y los lenguajes estadísticos como R y SAS tienen una gran cantidad de tales herramientas. Hasta hace relativamente poco, Python tenía pocas herramientas que proporcionaran el mismo nivel de riqueza y expresividad para trabajar con conjuntos de datos etiquetados.

La biblioteca de pandas, en desarrollo desde 2008, está destinada a cerrar la brecha en la riqueza de las herramientas de análisis de datos disponibles entre Python, un lenguaje de sistemas científicos y sistemas de propósito general, y las numerosas plataformas de computación estadística y lenguajes de bases de datos específicos. Nuestro objetivo no solo es proporcionar una funcionalidad equivalente, sino también implementar muchas características, como la alineación automática de datos y la indexación jerárquica, que no están disponibles de manera tan estrechamente integrada en otras bibliotecas o entornos informáticos que sepamos. Aunque inicialmente se desarrolló para aplicaciones de análisis de datos financieros, esperamos que pandas permita a Python científico ser un entorno de computación estadística más atractivo y práctico para profesionales académicos y de la industria por igual. El nombre de la biblioteca deriva de los datos del panel, si bien ofrecemos una viñeta de algunas de las principales características de interés en pandas, este documento no es de ninguna manera exhaustivo. Para obtener más información, remitimos al lector interesado a la documentación en línea en https://pandas.pydata.org/docs/ ([pandas]).

Conjuntos de datos estructurados

Los conjuntos de datos estructurados suelen llegar en formato tabular, es decir como una lista bidimensional de observaciones y nombres para los campos de cada observación. Por lo general, una observación puede identificarse de manera única mediante uno o más valores o etiquetas. Mostra-

mos un conjunto de datos de ejemplo para un par de acciones en el transcurso de varios días. La matriz de datos **NumPy** con **dtype** estructurado se puede utilizar para contener estos datos:

```
>>> data

array([('GOOG', '2009 - 12 - 28', 622.87, 1697900.0),
       ('GOOG', '2009 - 12 - 29', 619.40, 1424800.0),
       ('GOOG', '2009 - 12 - 30', 622.73, 1465600.0),
       ('GOOG', '2009 - 12 - 31', 619.98, 1219800.0),
       ('AAPL', '2009 - 12 - 28', 211.61, 23003100.0),
       ('AAPL', '2009 - 12 - 29', 209.10, 15868400.0),
       ('AAPL', '2009 - 12 - 30', 211.64, 14696800.0),
       ('AAPL', '2009 - 12 - 31', 210.73, 12571000.0)],
      dtype=[('item', '| S4'), ('date', '| S10'),
             ('price', '< f8'), ('volume', '< f8')])

>>> data['price']

array([622.87, 619.4, 622.73, 619.98, 211.61, 209.1,
       211.64, 210.73])
```

Las matrices estructuradas (o de registro) NumPy como esta pueden ser efectivas en muchas aplicaciones, pero en nuestra experiencia no proporcionan el mismo nivel de flexibilidad y facilidad de uso que otros entornos estadísticos. Un problema importante es que no se integran bien con el resto de NumPy, que está diseñado principalmente para trabajar con matrices de tipo homogéneo.

R proporciona la clase **data.frame** que almacena datos de tipos mixtos como una colección de columnas independientes. El lenguaje R central y sus bibliotecas de terceros se crearon teniendo en cuenta el objeto `data.frame`, por lo que la mayoría de las operaciones en un conjunto de datos de este tipo son muy naturales. Un `data.frame` también es flexible en tamaño, una característica importante al ensamblar una colección de datos. El siguiente

fragmento de código carga los datos almacenados en los datos del archivo **CSV** en la variable **df** y agrega una nueva columna de valores booleanos:

```
> df <- read.csv('data')

  item       date  price   volume
1 GOOG 2009-12-28 622.87  1697900
2 GOOG 2009-12-29 619.40  1424800
3 GOOG 2009-12-30 622.73  1465600
4 GOOG 2009-12-31 619.98  1219800
5 AAPL 2009-12-28 211.61 23003100
6 AAPL 2009-12-29 209.10 15868400
7 AAPL 2009-12-30 211.64 14696800
8 AAPL 2009-12-31 210.73 12571000

> df$ind <- df$item == "GOOG"

> df

  item       date  price  volume    ind
1 GOOG 2009-12-28 622.87  1697900  TRUE
2 GOOG 2009-12-29 619.40  1424800  TRUE
3 GOOG 2009-12-30 622.73  1465600  TRUE
4 GOOG 2009-12-31 619.98  1219800  TRUE
5 AAPL 2009-12-28 211.61 23003100 FALSE
6 AAPL 2009-12-29 209.10 15868400 FALSE
7 AAPL 2009-12-30 211.64 14696800 FALSE
8 AAPL 2009-12-31 210.73 12571000 FALSE
```

pandas proporciona una clase **DataFrame** con un nombre similar que implementa gran parte de la funcionalidad de su contraparte R, aunque con algunas mejoras importantes que discutiremos. Aquí convertimos la matriz estructurada anterior en un objeto **Pandas DataFrame** y, de manera similar, agregamos la misma columna:

```
>>> from pandas import DataFrame

>>> data = DataFrame(data)

>>> data

  item       date price    volume
0 GOOG 2009-12-28 622.9 1.698e+06
1 GOOG 2009-12-29 619.4 1.425e+06
2 GOOG 2009-12-30 622.7 1.466e+06
3 GOOG 2009-12-31 620    1.22e+06
4 AAPL 2009-12-28 211.6   2.3e+07
5 AAPL 2009-12-29 209.1 1.587e+07
6 AAPL 2009-12-30 211.6  1.47e+07
7 AAPL 2009-12-31 210.7 1.257e+07

>>> data['ind'] = data['item'] == 'GOOG'

>>> data

  item       date price    volume   ind
0 GOOG 2009-12-28 622.9 1.698e+06  True
1 GOOG 2009-12-29 619.4 1.425e+06  True
2 GOOG 2009-12-30 622.7 1.466e+06  True
3 GOOG 2009-12-31   620  1.22e+06  True
4 AAPL 2009-12-28 211.6   2.3e+07 False
5 AAPL 2009-12-29 209.1 1.587e+07 False
6 AAPL 2009-12-30 211.6  1.47e+07 False
7 AAPL 2009-12-31 210.7 1.257e+07 False
```

Estos datos se pueden reformar o "**pivotar**" en las columnas de fecha y elemento en una forma diferente para futuros ejemplos mediante el pivote del método **DataFrame**:

```
>>> del data['ind']  # delete ind column
```

```
>>> data.pivot('date', 'item')

            price       volume
   item     AAPL  GOOG  AAPL      GOOG
date
2009-12-28 211.6 622.9 2.3e+07   1.698e+06
2009-12-29 209.1 619.4 1.587e+07 1.425e+06
2009-12-30 211.6 622.7 1.47e+07  1.466e+06
2009-12-31 210.7 620 1.257e+07   1.22e+06
```

El resultado de la operación de pivote tiene un índice jerárquico para las columnas. Como mostraremos en una sección posterior, esta es una forma poderosa y flexible de representar y manipular datos multidimensionales. Actualmente, el método de pivote de `DataFrame` solo admite pivotar en dos columnas para cambiar la forma de los datos, pero podría aumentarse para considerar más que solo dos columnas. Mediante el uso de índices jerárquicos, podemos garantizar que el resultado siempre será bidimensional. Más adelante en el documento demostraremos la función **`pivot_table`** que puede producir resúmenes de datos de tablas dinámicas de hoja de cálculo como objetos `DataFrame` con filas y columnas jerárquicas.

Más allá de los datos de observación, uno también encontrará con frecuencia datos categóricos, que se pueden usar para dividir los identificadores en agrupaciones más amplias. Por ejemplo, los tickers de acciones pueden clasificarse según su industria o país de incorporación.

Aquí hemos creado un objeto `DataFrame` para **`cats`** que almacena clasificaciones de país e industria para un grupo de acciones:

```
>>> cats

       country industry
AAPL   US   TECH
IBM    US    TECH
SAP    DE   TECH
```

```
GOOG   US   TECH
C      US   FIN
SCGLY  FR   FIN
BAR    UK   FIN
DB     DE   FIN
VW     DE   AUTO
RNO    FR   AUTO
F      US   AUTO
TM     JP   AUTO
```

modelo de datos de pandas

Cada eje de una estructura de datos de pandas tiene un objeto **`Index`** que almacena información de etiquetado sobre cada cosa a lo largo de ese eje. El índice más general es simplemente un vector unidimensional de etiquetas (almacenado en una matriz de datos `NumPy`). Es conveniente pensar en el Índice como una implementación de un conjunto ordenado. En los datos de stock anteriores, el índice de la fila contiene simplemente números de observación secuenciales, mientras que el índice de la columna contiene los nombres de las columnas. No es necesario ordenar las etiquetas, aunque se podría implementar una subclase de índice para requerir la clasificación y proporcionar operaciones optimizadas para los datos ordenados (por ejemplo, datos de series de tiempo).

El objeto **`Index`** se usa para muchos propósitos:

- Realizar búsquedas para seleccionar subconjuntos de sectores de un objeto
- Proporcionar rutinas rápidas de alineación de datos para alinear un objeto con otro
- Habilitar el corte/selección intuitivo para formar nuevos objetos `Index`
- Formar uniones e intersecciones de objetos `Index`

Estos son algunos ejemplos de cómo se usa internamente el índice:

```
>>> index = Index(['a', 'b', 'c', 'd', 'e'])

>>> 'c' in index

True

>>> index.get_loc('d')

3

>>> index.slice_locs('b', 'd')

(1, 4)

# for aligning data
>>> index.get_indexer(['c', 'e', 'f'])

array([2, 4, -1], dtype=int32)
```

El índice básico utiliza un **dict** Python internamente para asignar etiquetas a sus respectivas ubicaciones e implementar estas características, aunque las subclases podrían adoptar un enfoque de rendimiento más especializado y potencialmente más alto. Los objetos multidimensionales como `DataFrame` no son subclases adecuadas del `ndarray` de `NumPy` ni utilizan matrices con `dtype` estructurado. En lanzamientos recientes de pandas hay una nueva estructura de datos interna conocida como **BlockManager** que manipula una colección de objetos `ndarray` n-dimensionales a los que nos referimos como bloques. Dado que `DataFrame` necesita poder almacenar datos de tipo mixto en las columnas, cada uno de estos objetos internos de Bloque contiene los datos de un conjunto de columnas que tienen el mismo tipo. En el ejemplo anterior, podemos examinar el `BlockManager`, aunque la mayoría de los usuarios nunca necesitarían hacer esto:

```
>>> data._data

BlockManager
Items:
    [item date price volume ind]
Axis 1:
    [0 1 2 3 4 5 6 7]
FloatBlock:
    [price volume], 2 x 8, dtype float64
ObjectBlock:
    [item date], 2 x 8, dtype object
BoolBlock:
    [ind], 1 x 8, dtype bool
```

La importancia clave de `BlockManager` es que muchas operaciones, por ejemplo, cualquier cosa orientada a filas (en lugar de orientada a columnas), especialmente en objetos `DataFrame` homogéneos, son significativamente más rápidas cuando los datos se almacenan en un solo `ndarray`. Sin embargo, como es común insertar y eliminar columnas, sería un desperdicio tener un paso de reasignación en cada paso de inserción o eliminación de columna. Como resultado, `BlockManager` proporciona efectivamente un esquema de evaluación diferida en el que las columnas recién insertadas se almacenan en nuevos objetos `Block`. Más tarde, ya sea explícitamente o cuando se invocan ciertos métodos en `DataFrame`, los bloques que tienen el mismo tipo se consolidarán, es decir, se combinarán para formar un único bloque homogéneo:

```
>>> data['newcol'] = 1.

>>> data._data

BlockManager
Items:
    [item date price volume ind newcol]
Axis 1:
```

```
    [0 1 2 3 4 5 6 7]
FloatBlock:
    [price volume], 2 x 8
ObjectBlock:
    [item date], 2 x 8
BoolBlock:
    [ind], 1 x 8
FloatBlock:
    [newcol], 1 x 8

>>> data.consolidate()._data

BlockManager
Items:
    [item date price volume ind newcol]
Axis 1:
    [0 1 2 3 4 5 6 7]
BoolBlock:
    [ind], 1 x 8
FloatBlock:
    [price volume newcol], 3 x 8
ObjectBlock:
    [item date], 2 x 8
```

La separación entre el objeto interno `BlockManager` y el `DataFrame` externo, orientado al usuario, brinda a los desarrolladores de pandas una gran cantidad de libertad para modificar la estructura interna para lograr un mejor rendimiento y uso de memoria.

Acceso a datos basado en etiquetas

Si bien la indexación basada en **`[]`** estándar (usando **`__getitem__`** y **`__setitem__`**) está reservada para el acceso a columnas en `DataFrame`, es útil poder indexar ambos ejes de un `DataFrame` de forma matricial usando

etiquetas. Nos gustaría poder obtener o establecer datos en cualquier eje usando uno de los siguientes:

- Una lista o conjunto de etiquetas o enteros.
- Un segmento, ya sea con enteros (p. Ej. `1: 5`) o etiquetas (p. Ej. `Lab1: lab2`)
- Un vector booleano
- Una sola etiqueta

Para evitar sobrecargar excesivamente los métodos relacionados con `[]`, lo que lleva a una semántica de indexación ambigua en algunos casos, hemos implementado un atributo especial de indexación de etiquetas **`ix`** en todas las estructuras de datos de pandas. Por lo tanto, podemos pasar una tupla de cualquiera de los objetos de indexación anteriores para obtener o establecer valores.

```
>>> df

              A       B       C        D
2000-01-03  -0.2047  1.007  -0.5397  -0.7135
2000-01-04   0.4789 -1.296   0.477   -0.8312
2000-01-05  -0.5194  0.275   3.249   -2.37
2000-01-06  -0.5557  0.2289 -1.021   -1.861
2000-01-07   1.966   1.353  -0.5771  -0.8608

>>> df.ix[:2, ['D', 'C', 'A']]

            D       C       A
2000-01-03 -0.7135 -0.5397 -0.2047
2000-01-04 -0.8312  0.477   0.4789

>>> df.ix[-2:, 'B':]

           B       C       D
```

```
2000-01-06 0.2289 -1.021  -1.861
2000-01-07 1.353  -0.5771 -0.8608

Setting values also works as expected.

>>> date1, date2 = df.index[[1, 3]]

>>> df.ix[date1:date2, ['A', 'C']] = 0

>>> df

            A           B       C       D
2000-01-03 -0.6856      0.1362  0.3996  1.585
2000-01-04  0          0.8863  0       1.907
2000-01-05  0         -1.351   0       0.104
2000-01-06  0         -0.8863  0       0.1741
2000-01-07 -0.05927    -1.013   0.9923 -0.4395
```

Alineación de datos

Las operaciones entre conjuntos de datos relacionados, pero de diferente tamaño, pueden plantear un problema, ya que el usuario debe primero asegurarse de que los puntos de datos estén correctamente alineados. Como ejemplo, considere series de tiempo en diferentes rangos de fechas o series de datos económicos en diferentes conjuntos de entidades:

```
>>> s1     >>> s2
AAPL   0.044   AAPL  0.025
IBM    0.050   BAR   0.158
SAP    0.101   C     0.028
GOOG   0.113   DB    0.087
C      0.138   F     0.004
SCGLY  0.037   GOOG  0.154
```

```
BAR    0.200   IBM   0.034
DB     0.281
VW     0.040
```

Uno podría elegir alinear explícitamente (o reindexar) uno de estos objetos de la Serie con el otro antes de agregarlos, utilizando el método **reindex**:

```
>>> s1.reindex(s2.index)

 AAPL  0.0440877763224
 BAR    0.199741007422
 C      0.137747485628
 DB     0.281070058049
 F                 NaN
 GOOG   0.112861123629
 IBM   0.0496445829129
```

Sin embargo, a menudo nos parece preferible simplemente ignorar el estado de la alineación de datos:

```
>>> s1 + s2

 AAPL   0.0686791008184
 BAR     0.358165479807
 C        0.16586702944
 DB      0.367679872693
 F                  NaN
 GOOG     0.26666583847
 IBM    0.0833057542385
 SAP                NaN
 SCGLY              NaN
 VW                 NaN
```

Aquí, los datos se alinearon automáticamente en función de sus etiquetas y se agregaron. El objeto resultante contiene la unión de las etiquetas entre

los dos objetos para que no se pierda información. Discutiremos el uso de **NaN** (No es un número) para representar los datos faltantes en la siguiente sección.

Claramente, el usuario paga una sobrecarga lineal siempre que se produce una alineación automática de datos y buscamos minimizar esa sobrecarga en la medida de lo posible. Se puede evitar la reindexación cuando se comparten objetos Index, lo que puede ser una estrategia efectiva en aplicaciones sensibles al rendimiento. [Cython], una herramienta ampliamente utilizada para crear extensiones Python C e interactuar con el código C/C ++, se ha utilizado para acelerar estos algoritmos centrales.

La alineación de datos con `DataFrame` se produce automáticamente en las etiquetas de columna y fila. Esta alineación de datos profundamente integrada difiere de cualquier otra herramienta fuera de Python que conozcamos. Similar a lo anterior, si las columnas en sí son diferentes, el objeto resultante contendrá la unión de las columnas:

```
>>> df                    >>> df2

              AAPL  GOOG                     AAPL
2009-12-28  211.6  622.9  2009-12-28    2.3e+07
2009-12-29  209.1  619.4  2009-12-29  1.587e+07
2009-12-30  211.6  622.7  2009-12-30   1.47e+07
2009-12-31  210.7    620


>>> df / df2

                AAPL GOOG
2009-12-28  9.199e-06  NaN
2009-12-29  1.318e-05  NaN
2009-12-30   1.44e-05  NaN
2009-12-31        NaN  NaN
```

Esto puede parecer una característica simple, pero en la práctica otorga

una libertad inmensa ya que ya no es necesario desinfectar los datos de una fuente no confiable. Por ejemplo, si cargó dos conjuntos de datos de una base de datos y las columnas y filas, se pueden agregar juntos, por ejemplo, sin tener que verificar si las etiquetas están alineadas. Por supuesto, después de realizar una operación entre dos conjuntos de datos, puede realizar una limpieza *ad hoc* de los resultados utilizando funciones como **`fillna`** y **`dropna`**:

```
>>> (df / df2).fillna(0)

                AAPL GOOG
 2009-12-28  9.199e-06    0
 2009-12-29  1.318e-05    0
 2009-12-30   1.44e-05    0
 2009-12-31          0    0

>>> (df / df2).dropna(axis=1, how='all')

                AAPL
 2009-12-28  9.199e-06
 2009-12-29  1.318e-05
 2009-12-30   1.44e-05
 2009-12-31        NaN
```

Manejo de datos faltantes

Es común que un conjunto de datos tenga observaciones faltantes. Por ejemplo, un grupo de series temporales económicas relacionadas almacenadas en un `DataFrame` puede comenzar en diferentes fechas. La realización de cálculos en presencia de datos faltantes puede conducir a un código complicado y una pérdida de rendimiento considerable. Elegimos usar `NaN` en lugar de usar `NumPy` con el objeto `MaskedArray` por razones de rendimiento (que están más allá del alcance de este documento), ya que

NaN se propaga en operaciones de coma flotante de forma natural y puede detectarse fácilmente en algoritmos. Si bien esto conduce a un buen rendimiento, viene con inconvenientes: a saber, que NaN no se puede usar en matrices de tipo entero, y no es un valor "*nulo*" intuitivo en matrices de objetos o cadenas (aunque se usa en estas matrices independientemente).

Consideramos el uso de NaN como un detalle de implementación e intentamos proporcionar al usuario las funciones API apropiadas para realizar operaciones comunes en puntos de datos faltantes. En el ejemplo anterior, podemos usar el método **dropna** para descartar datos faltantes, o podríamos usar **fillna** para reemplazar los datos faltantes con un valor específico:

```
>>> (s1 + s2).dropna()

 AAPL  0.0686791008184
 BAR    0.358165479807
 C       0.16586702944
 DB     0.367679872693
 GOOG    0.26666583847
 IBM   0.0833057542385


>>> (s1 + s2).fillna(0)

 AAPL   0.0686791008184
 BAR     0.358165479807
 C        0.16586702944
 DB      0.367679872693
 F                  0.0
 GOOG     0.26666583847
 IBM    0.0833057542385
 SAP                0.0
 SCGLY              0.0
 VW                 0.0
```

Los métodos **reindex** y **fillna** están equipados con un par de opciones de interpolación simples para propagar valores hacia adelante y hacia atrás, lo que es especialmente útil para datos de series de tiempo:

```
>>> ts                    >>> ts2

2000-01-03  0.03825       2000-01-03  0.03825
2000-01-04  -1.9884       2000-01-06  -0.0588
2000-01-05  0.73255       2000-01-11  0.04410
2000-01-06  -0.0588       2000-01-14  -0.1786
2000-01-07  -0.4767
2000-01-10  1.98008
2000-01-11  0.04410


>>> ts3 = ts + ts2

>>> ts3                   >>> ts3.fillna(method='ffill')

2000-01-03  0.07649       2000-01-03  0.07649
2000-01-04  NaN           2000-01-04  0.07649
2000-01-05  NaN           2000-01-05  0.07649
2000-01-06  -0.1177       2000-01-06  -0.1177
2000-01-07  NaN           2000-01-07  -0.1177
2000-01-10  NaN           2000-01-10  -0.1177
2000-01-11  0.08821       2000-01-11  0.08821
2000-01-14  NaN           2000-01-14  0.08821
```

Series y DataFrame también tienen métodos aritméticos explícitos con los que se puede usar un **fill_value** para especificar un tratamiento de datos faltantes en el cálculo. Una opción ocasional es tratar los valores faltantes como 0 al agregar dos Objetos de la serie:

```
>>> ts.add(ts2, fill_value=0)
```

```
2000-01-03  0.0764931953608
2000-01-04   -1.98842046359
2000-01-05   0.732553684194
2000-01-06  -0.117727627078
2000-01-07  -0.476754320696
2000-01-10     1.9800873096
2000-01-11  0.0882102892097
2000-01-14  -0.178640361674
```

Los métodos comunes de ndarray se han reescrito para excluir automáticamente los datos faltantes de los cálculos:

```
>>> (s1 + s2).sum()

1.3103630754662747

>>> (s1 + s2).count()

6
```

Similar a la función **is.na** de **R**, que detecta los valores de NA (no disponible), pandas tiene funciones API especiales **isnull** y **notnull** para determinar la validez de un punto de datos. Estos contrastan con **numpy.isnan** en el sentido de que pueden usarse con tipos distintos de flotante y también detectan otros marcadores de "falta" que se producen en la naturaleza, como el valor Python None.

```
>>> isnull(s1 + s2)

 AAPL   False
 BAR    False
 C      False
 DB     False
 F      True
 GOOG   False
```

```
IBM    False
SAP    True
SCGLY  True
VW     True
```

Tenga en cuenta que el valor de NA de R es distinto de NaN. Los desarrolladores principales de NumPy están trabajando actualmente en una implementación de valor de NA que esperamos satisfaga las necesidades de bibliotecas como pandas en el futuro.

Indexación jerárquica

Una adición relativamente reciente a pandas es la capacidad de un eje de tener un índice jerárquico, conocido en la biblioteca como **MultiIndex**. Semánticamente, esto significa que cada ubicación en un solo eje puede tener múltiples etiquetas asociadas.

```
>>> hdf

                A             B         C
foo one   -0.9884       0.09406     1.263
    two      1.29       0.08242  -0.05576
    three  0.5366       -0.4897    0.3694
bar one  -0.03457        -2.484   -0.2815
   two   0.03071       0.1091     1.126
baz two   -0.9773         1.474  -0.06403
   three  -1.283       0.7818    -1.071
qux one    0.4412         2.354    0.5838
   two    0.2215       -0.7445    0.7585
   three    1.73        -0.965   -0.8457
```

La indexación jerárquica se puede ver como una forma de representar datos de dimensiones superiores en una estructura de datos de dimensiones

inferiores (aquí, un Marco de datos 2D). Por ejemplo, podemos seleccionar filas del `DataFrame` anterior especificando solo una etiqueta del nivel más a la izquierda del índice:

```
>>> hdf.ix['foo']

              A        B         C
 one    -0.9884  0.09406     1.263
 two       1.29  0.08242  -0.05576
 three   0.5366  -0.4897    0.3694
```

Por supuesto, si se especifican todos los niveles, podemos seleccionar una fila o columna al igual que con un índice normal.

```
>>> hdf.ix['foo', 'three']

 A   0.5366
 B  -0.4897
 C   0.3694


# same result
>>> hdf.ix['foo'].ix['three']
```

El índice jerárquico se puede usar con cualquier eje. Del ejemplo de pivote anterior obtuvimos:

```
>>> pivoted = data.pivot('date', 'item')

>>> pivoted

             price            volume
              AAPL   GOOG       AAPL       GOOG
 2009-12-28  211.6  622.9    2.3e+07  1.698e+06
 2009-12-29  209.1  619.4  1.587e+07  1.425e+06
```

```
2009-12-30  211.6  622.7   1.47e+07  1.466e+06
2009-12-31  210.7    620  1.257e+07   1.22e+06
```

```
>>> pivoted['volume']

                AAPL       GOOG
2009-12-28    2.3e+07  1.698e+06
2009-12-29  1.587e+07  1.425e+06
2009-12-30   1.47e+07  1.466e+06
2009-12-31  1.257e+07   1.22e+06
```

Existen varios métodos de utilidad para manipular un `MultiIndex`, como **`swaplevel`** y **`sortlevel`**:

```
>>> swapped = pivoted.swaplevel(0, 1, axis=1)

>>> swapped

             AAPL    GOOG       AAPL       GOOG
            price   price     volume     volume
2009-12-28  211.6   622.9    2.3e+07  1.698e+06
2009-12-29  209.1   619.4  1.587e+07  1.425e+06
2009-12-30  211.6   622.7   1.47e+07  1.466e+06
2009-12-31  210.7     620  1.257e+07   1.22e+06
```

```
>>> swapped['AAPL']

            price     volume
2009-12-28  211.6    2.3e+07
2009-12-29  209.1  1.587e+07
2009-12-30  211.6   1.47e+07
2009-12-31  210.7  1.257e+07
```

Aquí hay un ejemplo para **`sortlevel`**:

```
>>> pivoted.sortlevel(1, axis=1)

             price     volume  price     volume
              AAPL       AAPL   GOOG       GOOG
 2009-12-28  211.6    2.3e+07  622.9  1.698e+06
 2009-12-29  209.1  1.587e+07  619.4  1.425e+06
 2009-12-30  211.6   1.47e+07  622.7  1.466e+06
 2009-12-31  210.7  1.257e+07    620   1.22e+06
```

Pivote avanzado y remodelación

Estrechamente relacionado con la indexación jerárquica y el ejemplo pivote anterior, ilustramos una remodelación más avanzada de datos utilizando los métodos de *apilar* y *desapilar*. La pila cambia de forma eliminando un nivel de las columnas de un objeto `DataFrame` y moviendo ese nivel a las etiquetas de fila, produciendo un ID `Series` u otro `DataFrame` (si las columnas eran un `MultiIndex`).

```
>>> df

              AAPL   GOOG
 2009-12-28  211.6  622.9
 2009-12-29  209.1  619.4
 2009-12-30  211.6  622.7
 2009-12-31  210.7    620


>>> df.stack()

 2009-12-28  AAPL  211.61
             GOOG  622.87
 2009-12-29  AAPL   209.1
```

```
             GOOG   619.4
 2009-12-30  AAPL  211.64
             GOOG  622.73
 2009-12-31  AAPL  210.73
             GOOG  619.98


>>> pivoted

             price            volume
              AAPL   GOOG       AAPL       GOOG
 2009-12-28  211.6  622.9    2.3e+07  1.698e+06
 2009-12-29  209.1  619.4  1.587e+07  1.425e+06
 2009-12-30  211.6  622.7   1.47e+07  1.466e+06
 2009-12-31  210.7    620  1.257e+07   1.22e+06


>>> pivoted.stack()

                   price     volume
 2009-12-28  AAPL  211.6    2.3e+07
             GOOG  622.9  1.698e+06
 2009-12-29  AAPL  209.1  1.587e+07
             GOOG  619.4  1.425e+06
 2009-12-30  AAPL  211.6   1.47e+07
             GOOG  622.7  1.466e+06
 2009-12-31  AAPL  210.7  1.257e+07
             GOOG    620   1.22e+06
```

Por defecto, el nivel más interno está apilado. El nivel para apilar se puede especificar explícitamente:

```
>>> pivoted.stack(0)

                        AAPL       GOOG
```

```
2009-12-28  price       211.6      622.9
            volume    2.3e+07  1.698e+06
2009-12-29  price       209.1      619.4
            volume  1.587e+07  1.425e+06
2009-12-30  price       211.6      622.7
            volume   1.47e+07  1.466e+06
2009-12-31  price       210.7        620
            volume  1.257e+07   1.22e+06
```

El método de *desapilar* es el inverso de **stack**:

```
>>> df.stack()
2009-12-28  AAPL  211.61
            GOOG  622.87
2009-12-29  AAPL   209.1
            GOOG   619.4
2009-12-30  AAPL  211.64
            GOOG  622.73
2009-12-31  AAPL  210.73
            GOOG  619.98
```

```
>>> df.stack().unstack()
                AAPL   GOOG
2009-12-28  211.6  622.9
2009-12-29  209.1  619.4
2009-12-30  211.6  622.7
2009-12-31  210.7    620
```

Estos métodos de remodelación se pueden combinar con el método integrado `DataFrame` y `Series` para seleccionar o agregar datos a un nivel. Aquí tomamos el máximo entre **AAPL** y **GOOG** para cada par de fecha/campo:

```
>>> pivoted.stack(0)
                        AAPL       GOOG
2009-12-28  price       211.6      622.9
            volume    2.3e+07  1.698e+06
2009-12-29  price       209.1      619.4
            volume  1.587e+07  1.425e+06
2009-12-30  price       211.6      622.7
            volume   1.47e+07  1.466e+06
```

```
 2009-12-31  price       210.7        620
             volume  1.257e+07   1.22e+06


>>> pivoted.stack(0).max(1).unstack()

             price     volume
 2009-12-28  622.9    2.3e+07
 2009-12-29  619.4  1.587e+07
 2009-12-30  622.7   1.47e+07
 2009-12-31    620  1.257e+07
```

Este tipo de agregaciones están estrechamente relacionadas con las operaciones "*agrupar por*" que discutiremos en la siguiente sección.

Agrupar por: agrupar y agregar datos

Una operación muy común en lenguajes similares a SQL y generalmente en análisis de datos estadísticos es agrupar datos por algunos identificadores y realizar una agregación o transformación de los datos. Por ejemplo, supongamos que tenemos un conjunto de datos simple como este:

```
>>> df

    A    B            C        D
 0  foo  one     -1.834    1.903
 1  bar  one      1.772  -0.7472
 2  foo  two      -0.67   -0.309
 3  bar  three  0.04931   0.3939
 4  foo  two    -0.5215    1.861
 5  bar  two     -3.202   0.9365
 6  foo  one     0.7927    1.256
 7  foo  three   0.1461   -2.655
```

Podríamos agrupar usando la columna **A** así:

```
>>> df.groupby('A').mean()

            C       D
bar  -0.4602  0.1944
foo  -0.4173  0.4112
```

El objeto devuelto por **groupby** es un objeto intermedio especial con muchas características agradables. Por ejemplo, puede usarlo para recorrer las partes del conjunto de datos correspondientes a cada grupo:

```
>>> for key, group in df.groupby('A'):
...     print key
...     print group

bar

    A    B            C        D
 1  bar  one      1.772  -0.7472
 3  bar  three  0.04931   0.3939
 5  bar  two     -3.202   0.9365


foo

    A    B            C       D
 0  foo  one     -1.834   1.903
 2  foo  two      -0.67  -0.309
 4  foo  two    -0.5215   1.861
 6  foo  one     0.7927   1.256
 7  foo  three   0.1461   -2.65
```

También es posible agrupar por múltiples columnas:

```
df.groupby(['A', 'B']).mean()

                  C        D
 bar one      1.772  -0.7472
    three 0.04931   0.3939
    two    -3.202   0.9365
 foo one   -0.5205    1.579
    three  0.1461   -2.655
    two   -0.5958   0.7762
```

El resultado predeterminado de una agrupación multi-clave por agregación es un índice jerárquico. Esto se puede deshabilitar cuando se llama `groupby`, lo que puede ser útil en algunas configuraciones:

```
df.groupby(['A', 'B'], as_index=False).mean()

    A    B            C        D
 0  bar  one      1.772  -0.7472
 1  bar  three  0.04931   0.3939
 2  bar  two     -3.202   0.9365
 3  foo  one    -0.5205    1.579
 4  foo  three   0.1461   -2.655
 5  foo  two    -0.5958   0.7762
```

En una configuración completamente general, las operaciones grupales se refieren a asignar etiquetas de eje a cubos. En los ejemplos anteriores, cuando pasamos nombres de columna, simplemente estamos estableciendo una correspondencia entre las etiquetas de fila y los identificadores de grupo. Hay otras formas de hacer esto; lo más general es pasar una función de Python (para una sola clave) o una lista de funciones (para multiclave) que se invocará en cada etiqueta, produciendo una especificación de grupo:

```
>>> dat

                    A        B        C        D
```

```
 2000-01-03   0.6371    0.672   0.9173    1.674
 2000-01-04  -0.8178   -1.865    -0.23   0.5411
 2000-01-05    0.314   0.2931  -0.6444  -0.9973
 2000-01-06    1.913  -0.5867    0.273   0.4631
 2000-01-07    1.308    0.426   -1.306  0.04358


>>> mapping

{'A': 'Group 1', 'B': 'Group 2',
'C': 'Group 1', 'D': 'Group 2'}

>>> for name, group in dat.groupby(mapping.get,
... axis=1):
... print name
print group

Group 1
                   A        C
 2000-01-03   0.6371   0.9173
 2000-01-04  -0.8178    -0.23
 2000-01-05    0.314  -0.6444
 2000-01-06    1.913    0.273
 2000-01-07    1.308   -1.306

Group 2
                   B        D
 2000-01-03    0.672    1.674
 2000-01-04   -1.865   0.5411
 2000-01-05   0.2931  -0.9973
 2000-01-06  -0.5867   0.4631
 2000-01-07    0.426  0.04358
```

Cierta creatividad con funciones de agrupación permitirá al usuario realizar operaciones bastante sofisticadas. El objeto devuelto por **groupby** puede

iterar, agregar (con una función arbitraria), transformar (calcular una versión modificada de cada grupo de datos) o hacer un grupo de aplicación general. Si bien no tenemos espacio para entrar en gran detalle con ejemplos de cada uno de estos, la función de aplicación es interesante ya que intenta combinar los resultados de la agregación en un objeto pandas. Por ejemplo, podríamos agrupar el objeto **df** anterior por la columna **A**, seleccionar solo la columna **C** y aplicar la función de descripción a cada subgrupo de la siguiente manera:

```
>>> df.groupby('A')['C'].describe().T

            bar      foo
count         3        5
mean    -0.4602  -0.4173
std       2.526   0.9827
min      -3.202   -1.834
10%      -2.552   -1.368
50%     0.04931  -0.5215
90%       1.427   0.5341
max       1.772   0.7927
```

Tenga en cuenta que, bajo el capó, llamar a **describe** genera y pasa una función dinámica para aplicar que invoca **describe** en cada grupo y pega los resultados juntos. Transpusimos el resultado con **.T** para hacerlo más legible.

Pivoteo de tablas

Una aplicación obvia que combina **groupby** y operaciones de remodelación es crear tablas dinámicas, una forma común de resumir datos en aplicaciones de hoja de cálculo como *Microsoft Excel*. Analizaremos brevemente un conjunto de datos de propinas recopilados de un restaurante ([Bryant]):

```
>>> tips.head()

    sex     smoker  time    day  size  tip_pct
 1  Female  No      Dinner  Sun     2  0.05945
 2  Male    No      Dinner  Sun     3   0.1605
 3  Male    No      Dinner  Sun     3   0.1666
 4  Male    No      Dinner  Sun     2   0.1398
 5  Female  No      Dinner  Sun     4   0.1468
```

La función **pivot_table** en pandas toma un conjunto de nombres de columna para agrupar en las filas de la tabla dinámica, otro conjunto para agrupar en las columnas y, opcionalmente, una función de agregación para cada grupo (que por defecto es mean)

```
>>> import numpy as np

>>> from pandas import pivot_table

>>> pivot_table(tips, 'tip_pct', rows=['time', 'sex'],
cols='smoker')

 smoker          No      Yes
 time    sex
 Dinner  Female  0.1568  0.1851
         Male    0.1594  0.1489
 Lunch   Female  0.1571  0.1753
         Male    0.1657  0.1667
```

Convenientemente, el objeto devuelto es un DataFrame, por lo que puede ser remodelado y manipulado por el usuario:

```
>>> table = pivot_table(tips, 'tip_pct',
rows=['sex', 'day'],
    cols='smoker', aggfunc=len)
```

```
>>> table

 smoker        No  Yes
 sex     day
 Female  Fri    2    7
         Sat   13   15
         Sun   14    4
         Thur  25    7
 Male    Fri    2    8
         Sat   32   27
         Sun   43   15
         Thur  20   10


>>> table.unstack('sex')

 smoker      No           Yes
 sex     Female  Male  Female  Male
 day
 Fri          2     2       7     8
 Sat         13    32      15    27
 Sun         14    43       4    15
 Thur        25    20       7    10
```

Para muchos usuarios, esta será una alternativa atractiva al volcado de un conjunto de datos en una hoja de cálculo con el único propósito de crear una tabla dinámica.

```
>>> pivot_table(tips, 'size',
rows=['time', 'sex', 'smoker'],
    cols='day', aggfunc=np.sum,
    fill_value=0)

            day          Fri   Sat  Sun Thur
 time    sex     smoker
```

```
Dinner  Female  No        2    30   43   2
                Yes       8    33   10   0
Dinner  Male    No        4    85  124   0
                Yes      12    71   39   0
Lunch   Female  No        3     0    0  60
                Yes       6     0    0  17
Lunch   Male    No        0     0    0  50
                Yes       5     0    0  23
```

Combinando o uniendo conjuntos de datos

Combinar, unir o fusionar conjuntos de datos relacionados es una operación bastante común. Al hacerlo, estamos interesados en asociar observaciones de un conjunto de datos con otro a través de una clave de combinación de algún tipo. Para datos 2D indexados de manera similar, las etiquetas de fila sirven como una clave natural para la función de unión:

```
>>> df1                      >>> df2

              AAPL   GOOG                MSFT   YHOO
2009-12-24    209  618.5  2009-12-24     31  16.72
2009-12-28  211.6  622.9  2009-12-28  31.17  16.88
2009-12-29  209.1  619.4  2009-12-29  31.39  16.92
2009-12-30  211.6  622.7  2009-12-30  30.96  16.98
2009-12-31  210.7    620


>>> df1.join(df2)

              AAPL   GOOG   MSFT   YHOO
2009-12-24    209  618.5     31  16.72
2009-12-28  211.6  622.9  31.17  16.88
2009-12-29  209.1  619.4  31.39  16.92
```

```
2009-12-30  211.6  622.7  30.96  16.98
2009-12-31  210.7    620    NaN    NaN
```

Uno podría estar interesado en unirse también en algo que no sea el índice, como los datos categóricos que presentamos en una sección anterior:

```
>>> data.join(cats, on='item')

   country        date  industry  item  value
0  US       2009-12-28  TECH      GOOG  622.9
1  US       2009-12-29  TECH      GOOG  619.4
2  US       2009-12-30  TECH      GOOG  622.7
3  US       2009-12-31  TECH      GOOG    620
4  US       2009-12-28  TECH      AAPL  211.6
5  US       2009-12-29  TECH      AAPL  209.1
6  US       2009-12-30  TECH      AAPL  211.6
7  US       2009-12-31  TECH      AAPL  210.7
```

Esto es similar a una operación de unión SQL entre dos tablas o una operación **BUSCARV** en una hoja de cálculo como *Excel*. Es posible unirse en varias claves, en cuyo caso la tabla que se está uniendo actualmente debe tener un índice jerárquico correspondiente a esas claves. Estaremos trabajando en más métodos de unión y fusión en una versión futura de pandas.

Rendimiento y uso para grandes conjuntos de datos

El uso de objetos `DataFrame` sobre matrices `NumPy` homogéneas para el cálculo incurre en una sobrecarga de varios factores:

- Las funciones computacionales como suma, media y estándar se han anulado para omitir datos faltantes

- La mayoría de las estructuras de datos del índice del eje dependen de Python `dict` para realizar búsquedas y alineación de datos. Esto también da como resultado una huella de memoria ligeramente mayor, ya que el `dict` que contiene el mapeo de etiquetas se crea una vez y luego se almacena.

- La estructura de datos interna de `BlockManager` consolida los datos de cada tipo (punto flotante, entero, booleano, objeto) en matrices bidimensionales. Sin embargo, este es un costo inicial que acelera los cálculos orientados a filas y la alineación de datos más adelante.

- Realizar búsquedas repetidas de valores por etiqueta pasa a través de mucho más código Python que búsquedas simples basadas en enteros en objetos `ndarray`.

El usuario inteligente aprenderá qué operaciones no son muy eficientes en `DataFrame` y `Series` y recurrirá a trabajar directamente con los objetos `ndarray` subyacentes (accesibles a través del atributo de valores) en tales casos. Lo que sacrifica `DataFrame` en rendimiento lo compensa en flexibilidad y expresividad.

Con enteros de 64 bits que representan marcas de tiempo, pandas de hecho proporcionan algunas de las rutinas de alineación de datos más rápidas para series de tiempo indexadas de manera diferente que se encuentran en el software de código abierto. Como trabajar con series de tiempo grandes e irregulares requiere tener un índice de marca de tiempo, pandas está bien posicionado para convertirse en el estándar de oro para el procesamiento de series de tiempo de código abierto de alto rendimiento.

Con respecto al uso de memoria y grandes conjuntos de datos, pandas actualmente solo está diseñado para su uso con conjuntos de datos en memoria. Nos gustaría ampliar su capacidad para trabajar con conjuntos de datos que no caben en la memoria, tal vez de manera transparente utilizando el módulo de multiprocesamiento o un backend de cómputo paralelo para organizar cálculos a gran escala.

pandas para usuarios de R

Dado que el nombre y la característica de "*DataFrame*" se superponen con el proyecto [R] y sus paquetes de terceros, pandas establecerá comparaciones inevitables con R. pandas aporta un conjunto de herramientas de análisis de datos robusto, con todas las funciones e integrado a Python, manteniendo un sistema simple y una API fácil de usar. Como casi todas las manipulaciones de datos que involucran objetos `data.frame` en R se pueden expresar fácilmente utilizando el `DataFrame` de pandas, en la mayoría de los casos es relativamente sencillo portar las funciones de R a `Python`. Sería útil proporcionar una guía de migración para los usuarios de R, ya que no hemos copiado las convenciones de nombres o la sintaxis de R en la mayoría de los lugares, sino más bien los nombres basados en el sentido común y haciendo que la sintaxis y la API sean lo más "*Pythonic*" posible.

R no proporciona funcionalidad de indexación de una manera tan profundamente integrada como lo hace pandas. Por ejemplo, las operaciones entre objetos `data.frame` procederán en R sin importar si las etiquetas coinciden siempre que tengan el mismo largo y ancho. Algunos paquetes R, como `zoo` y `xts`, proporcionan estructuras de datos indexados con alineación de datos, pero están muy especializados en datos de series de tiempo ordenadas.

La indexación jerárquica con selección de subconjunto de tiempo constante es otra característica importante que falta en las estructuras de datos de R. Fuera del alcance de este documento hay una comparación rigurosa de rendimiento de R y pandas. En casi todos los puntos de referencia que hemos realizado comparando R y pandas, pandas supera significativamente a R.

Otras características notables

Hay muchas otras características en pandas que vale la pena explorar para los usuarios interesados:

- Funcionalidad de series de tiempo: generación de rango de fechas, desplazamiento y retraso, conversión de frecuencia y llenado hacia adelante/atrás
- Integración con [matplotlib] para generar de forma concisa trazados con metadatos
- Estadísticas de ventanas móviles (p. Ej., Desviación estándar móvil, promedio móvil ponderado exponencialmente) y regresión lineal y de panel de ventanas móviles.
- Estructura de datos de panel tridimensional para manipular colecciones de objetos DataFrame
- Versiones dispersas de las estructuras de datos.
- Robustas herramientas de E/S para leer y escribir objetos pandas en archivos planos (texto delimitado, CSV, Excel) y formato HDF5

Paquetes relacionados

Varios otros paquetes de Python tienen cierto grado de superposición de características con `pandas`. Dado que `ndarray` solo es aplicable a muchos problemas en su forma homogénea (`dtype` no estructurado), en pandas nos hemos distanciado de `ndarray` para proporcionar una estructura de datos más flexible, (potencialmente) heterogénea y de tamaño variable. Las referencias incluyen algunos otros paquetes de interés.

pandas pronto se convertirán en una dependencia de **`statsmodels`** ([StaM]), la principal biblioteca estadística y econométrica en Python, para hacer

que las herramientas de modelado estadístico y análisis de datos en Python sean más coherentes e integradas. Planeamos combinar pandas con un marco de fórmulas para hacer que la especificación de modelos estadísticos sea fácil e intuitiva cuando se trabaja con un `DataFrame` de datos, por ejemplo.

Conclusiones

Creemos que en los próximos años habrá una gran oportunidad para atraer a los usuarios que necesitan herramientas de análisis de datos estadísticos a Python que podrían haber elegido previamente R, `MATLAB` u otro entorno de investigación. Al diseñar estructuras de datos robustas y fáciles de usar que sean coherentes con el resto de la pila científica de Python, podemos hacer de Python una opción convincente para las aplicaciones de análisis de datos. En nuestra opinión, pandas proporciona una base sólida sobre la cual se puede establecer un ecosistema de análisis de datos muy poderoso.

Referencias

[pandas] W. McKinney, pandas: a python data analysis library, https://pandas.pydata.org/

[scipy2010] W. McKinney, Data Structures for Statistical Computing in Python Proceedings of the 9th Python in Science Conference, http://conference.scipy.org/scipy2010/. 2010

[Larry] K. Goodman. la/larry: ndarray with labeled axes, http://larry.sourceforge.net/

[SciTS] M. Knox, P. Gerard-Marchant, scikits.timeseries: python time series analysis, http://pytseries.sourceforge.net/

[StaM] S. Seabold, J. Perktold, J. Taylor, statsmodels: statistical modeling in Python, http://statsmodels.sourceforge.net

[SciL] D. Cournapeau, et al., scikit-learn: machine learning in Python, https://scikit-learn.org/stable/

[PyMC] C. Fonnesbeck, A. Patil, D. Huard, PyMC: Markov Chain Monte Carlo for Python, http://code.google.com/p/pymc/

[Tab] D. Yamins, E. Angelino, tabular: tabarray data structure for 2D data, http://parsemydata.com/tabular/

[NumPy] T. Oliphant, https://numpy.org/

[SciPy] E. Jones, T. Oliphant, P. Peterson, https://scipy.org/

[matplotlib] J. Hunter, et al., matplotlib: Python plotting, https://matplotlib.org/

[EPD] Enthought, Inc., EPD: Enthought Python Distribution, https://www.enthought.com/

[Pythonxy] P. Raybaut, Python(x,y): Scientific-oriented Python distribution, http://www.pythonxy.com/

[CRAN] The R Project for Statistical Computing, https://cran.r-project.org/

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn, et al., The Cython compiler, https://cython.org

[IPython] Fernando Pérez, Brian E. Granger, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. http://ipython.org

[Grun] Batalgi, Grunfeld data set, https://www.wiley.com/legacy/wileychi/baltagi/

[nipy] J. Taylor, F. Perez, et al., nipy: Neuroimaging in Python, https://nipy.org/

[pydataframe] A. Straw, F. Finkernagel, pydataframe, https://code.google.com/archive/p/pydataframe/

[R] R Development Core Team. 2010, R: A Language and Environment for Statistical Computing, https://www.R-project.org

[MATLAB] The MathWorks Inc. 2010, MATLAB, https://www.mathworks.com

[Stata] StatCorp. 2010, Stata Statistical Software: Release 11 https://www.stata.com

[SAS] SAS Institute Inc., SAS System, https://www.sas.com