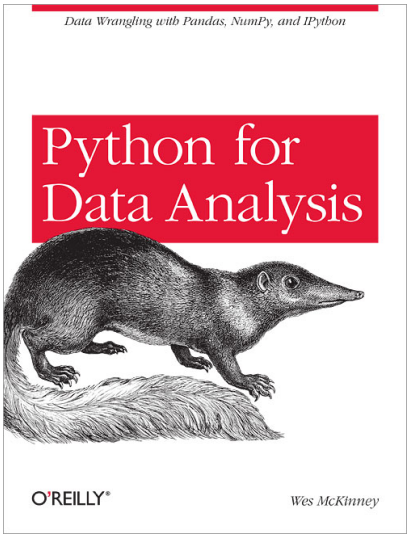
computacional 1, Reporte 1

Juan José López Rodríguez

November 2018

1 Introduction

El segundo capítulo del libro "Python for Data Analysis" escrito por Wes McKinney cubre conceptos básicos de la estructura de **Python**, por lo que en la primera actividad del curso, se nos encargó leerlo, a continuación se presenta el resumen



2 Resumen

CHAPTER 2

Python Language Basics, IPython, and Jupyter Notebooks

Uno de los principales componentes del proyecto Jupyter es el cuaderno, un tipo de documento interactivo para código, texto (con o sin marcado), visualizaciones de datos y otros resultados. El cuaderno Jupyter interactúa con kernels, que son implementaciones del protocolo de computación interactiva Jupyter en cualquier cantidad de lenguajes de programación. El kernel Jupyter de Python usa el sistema IPython para su comportamiento subyacente
Si estamos usando el Jupyter notebook, podemos copiar y pegar el código de cualquier celda y ejecutarlo. También es posible el correr código desde el portapapeles en el Ipython Shell.

Cuando guarde el Jupyter notebook (consulte "Guardar y punto de control" en el menú Archivo del notebook), creará un archivo con la extensión .ipynb. Este es un formato de archivo independiente que contiene todo el contenido (incluido cualquier resultado de código evaluado) actualmente en el bloc de notas. Estos pueden ser cargados y editados por otros usuarios de Jupyter. Para cargar una computadora portátil existente, coloque el archivo en el mismo directorio donde comenzó el proceso de la computadora portátil (o en una subcarpeta dentro de ella), luego haga doble clic en el nombre de la página de destino. Puedes probarlo con los cuadernos de mi repositorio de wesm / pydata-book en GitHub.

Finalización de tabulación large

Es un proceso que ofrece Python que te permite empezar a escribir una variable (objeto, función, etc.) y busca los caracteres que ya habias escrito para ver a cual se le parece. Ej:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and         an_example  any
```

Introspección

Usar un signo de interrogación (?) Antes o después de una variable mostrará información general sobre el objeto:

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:       list
String Form:[1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [10]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

Atajos de teclado de terminal

IPython tiene muchos atajos de teclado para navegar por el prompt (que será familiar para los usuarios del editor de texto Emacs o el shell bash de Unix) e interactuar con el histórico de comandos del Shell

Keyboard shortcut	Description
Ctrl-P or up-arrow	Search backward in command history for commands starting with currently entered text
Ctrl-N or down-arrow	Search forward in command history for commands starting with currently entered text
Ctrl-R	Readline-style reverse history search (partial matching)
Ctrl-Shift-V	Paste text from clipboard
Ctrl-C	Interrupt currently executing code
Ctrl-A	Move cursor to beginning of line
Ctrl-E	Move cursor to end of line
Ctrl-K	Delete text from cursor until end of line
Ctrl-U	Discard all text on current line
Ctrl-F	Move cursor forward one character
Ctrl-B	Move cursor back one character
Ctrl-L	Clear screen

Los comandos especiales

Los comandos especiales de IPython (que no están integrados en Python) se conocen como comandos "mágicos". Estos están diseñados para facilitar tareas comunes y le permiten controlar fácilmente el comportamiento del sistema IPython. Un comando mágico es cualquier comando precedido por el símbolo de porcentaje. Por ejemplo, puede verificar el tiempo de ejecución de cualquier instrucción de Python, como una multiplicación de matrices, utilizando la función de magia **timeit**.

Los comandos mágicos se pueden ver como programas de línea de comandos para ejecutar dentro del sistema IPython. Muchos de ellos tienen opciones adicionales de "línea de comando", que se pueden ver todas (como era de esperar)

con ?.
A continuación, los comandos especiales más comunes:

Command	Description
`%quickref`	Display the IPython Quick Reference Card
`%magic`	Display detailed documentation for all of the available magic commands
`%debug`	Enter the interactive debugger at the bottom of the last exception traceback
`%hist`	Print command input (and optionally output) history
`%pdb`	Automatically enter debugger after any exception
`%paste`	Execute preformatted Python code from clipboard
`%cpaste`	Open a special prompt for manually pasting Python code to be executed
`%reset`	Delete all variables/names defined in interactive namespace
`%page` *OBJECT*	Pretty-print the object and display it through a pager
`%run` *scrtpt.py*	Run a Python script inside IPython
`%prun` *statement*	Execute *statement* with `cProfile` and report the profiler output
`%time` *statement*	Report the execution time of a single statement
`%timeit` *statement*	Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time
`%who, %who_ls, %whos`	Display variables defined in interactive namespace, with varying levels of information/ verbosity
`%xdel` *variable*	Delete a variable and attempt to clear any references to the object in the IPython internals

Lenguaje básico de Paython

Semántica del lenguaje

El diseño del lenguaje Python se distingue por su énfasis en la legibilidad, la simplicidad y la claridad. Algunas personas llegan incluso a compararlo con un "pseudocódigo ejecutable"

Todo es un objeto

Una característica importante del lenguaje Python es la consistencia de su modelo de objetos. Cada número, cadena, estructura de datos, función, clase, módulo, etc. existe en el intérprete de Python en su propia "caja", que se conoce como objeto de Python. Cada objeto tiene un tipo asociado (por ejemplo, cadena o función) y datos internos. En la práctica, esto hace que el lenguaje sea muy flexible, ya que incluso las funciones se pueden tratar como cualquier otro objeto.

Variables y paso de argumento

Al asignar una variable (o nombre) en Python, está creando una referencia al objeto en el lado derecho del signo de igual. En términos prácticos, considere una lista de enteros:

```
In [8]: a = [1, 2, 3]
```

Supongamos que asignamos a una nueva variable b:

```
In [9]: b = a
```

En algunos idiomas, esta asignación causaría la copia de los datos [1, 2, 3]. En Python, a y b en realidad ahora se refieren al mismo objeto, la lista original [1, 2, 3]

Cuando pasa objetos como argumentos a una función, se crean nuevas variables locales que hacen referencia a los objetos originales sin ninguna copia. Si vincula un objeto nuevo a una variable dentro de una función, ese cambio no se reflejará en el alcance principal. Por lo tanto, es posible alterar las partes internas de un argumento mutable. Supongamos que tenemos la siguiente función:

```
def append_element(some_list, element):
    some_list.append(element)
```

Entonces tenemos:

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

c

Variables y paso de argumento

atributos y metodos

Los objetos en Python suelen tener tanto atributos (otros objetos Python almacenados "dentro" del objeto) como métodos (funciones asociadas con un objeto que puede tener acceso a los datos internos del objeto). A ambos se accede a través de la sintaxis obj.attribute$_name$:

```
In [1]: a = 'foo'
```

A los atributos y métodos también se puede acceder por nombre a través de la función getattr:

```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

```
In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

Operadores binarios y comparaciones

Operation	Description
`a + b`	Add a and b
`a - b`	Subtract b from a
`a * b`	Multiply a by b
`a / b`	Divide a by b
`a // b`	Floor-divide a by b, dropping any fractional remainder
`a ** b`	Raise a to the b power
`a & b`	`True` if both a and b are `True`; for integers, take the bitwise AND
`a | b`	`True` if either a or b is `True`; for integers, take the bitwise OR
`a ^ b`	For booleans, `True` if a or b is `True`, but not both; for integers, take the bitwise `EXCLUSIVE-OR`

Operation	Description
`a == b`	`True` if a equals b
`a != b`	`True` if a is not equal to b
`a <= b, a < b`	`True` if a is less than (less than or equal) to b
`a > b, a >= b`	`True` if a is greater than (greater than or equal) to b
`a is b`	`True` if a and b reference the same Python object
`a is not b`	`True` if a and b reference different Python objects

Tipos de escalares

Python junto con su biblioteca estándar tiene un pequeño conjunto de tipos

incorporados para manejar datos numéricos, cadenas, valores booleanos (Verdadero o Falso), y fechas y horas.

Table 2-4. Standard Python scalar types

Type	Description
`None`	The Python "null" value (only one instance of the None object exists)
`str`	String type; holds Unicode (UTF-8 encoded) strings
`bytes`	Raw ASCII bytes (or Unicode encoded as bytes)
`float`	Double-precision (64-bit) floating-point number (note there is no separate `double` type)
`bool`	A `True` or `False` value
`int`	Arbitrary precision signed integer

Booleanos

Los dos valores booleanos en Python se escriben como Verdadero y Falso. Comparaciones y otras expresiones condicionales evalúan como Verdadero o Falso. Los valores booleanos son combinados con las palabras clave **and** o **or**:

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

Fecha y horas

El módulo **datetime** de Python incorporado proporciona tipos de fecha y hora, fecha y hora. El tipo **datetime**, como puede imaginar, combina la información almacenada en fecha y hora y es el más utilizado:

```
In [102]: from datetime import datetime, date, time

In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

Especificación de formato de datetime

Type	Description
%Y	Four-digit year
%y	Two-digit year

Type	Description
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	Two-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]
%U	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
%W	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
%z	UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
%F	Shortcut for %Y-%m-%d (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

Flujo de control

Python tiene varias palabras clave integradas para lógica condicional, bucles y otros conceptos de flujo de control estándar que se encuentran en otros lenguajes de programación.

if,elif y else

La instrucción **if** es uno de los tipos de instrucciones de flujo de control más conocidos. Verifica una condición que, si es Verdadero, evalúa el código en el bloque que sigue:

```
if x < 0:
    print('It's negative')
```

Una instrucción **if** puede ir seguida opcionalmente por uno o más bloques elif y un bloque else si todas las condiciones son falsas:

```
if x < 0:
    print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

for loops son para iterar sobre una colección (como una lista o tupla) o un iterater. La sintaxis estándar para un bucle for es:

```
for value in collection:
    # do something with value
```

Puede avanzar un bucle **for** a la siguiente iteración, omitiendo el resto del bloque, usando la palabra clave **continue**. Considere este código, que resume los enteros en una lista y omite los valores de **None**:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

Un bucle for se puede salir completamente con la palabra clave break. Este código suma elementos de la lista hasta que se alcanza un 5:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

Un **while loop** especifica una condición y un bloque de código que se ejecutará hasta que la condición se evalúe como Falso o el bucle finalice explícitamente con una ruptura:

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

Pass

Es la declaración "no-op" en Python. Se puede usar en bloques donde no se debe realizar ninguna acción (o como un marcador de posición para el código aún no implementado); solo es necesario porque Python usa espacios en blanco para delimitar bloques:

```
if x < 0:
    print('negative!')
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print('positive!')
```

Range
: La función devuelve un iterador que produce una secuencia de enteros espaciados uniformemente:

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se puede dar un inicio, un final y un paso (que pueden ser negativos):

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

Como puede ver, el rango produce enteros hasta pero sin incluir el punto final. Un uso común de rango es para iterar a través de secuencias por índice:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

3 Conclusion

El libro de Wes McKinney me resulto una buena introduccion al entorno de Python. Ve las cosas superficiales y no te atiborra de informacion, cosa que para la introduccion y tener una idea general me parece excelente.

References

[1] McKinney *Python for Data Analysis Data Wrangling with Pandas, NumPy, and IPython*, segunda edicion, O'Reill, Estados Unidos, 2017.