Breve introducción a Python y SymPy

Fernando Mazzone

19 de marzo de 2016

1. Descripción

Python es un lenguaje de programación interpretado, abierto, facil de aprender, potente y portátil. Es utilizado en proyectos de todo tipo, no sólo aplicaciones científicas.

SciPy, Python científico, es un conjunto de módulos de python para distintos tipos de cálculos. Está integrado por los módulos, SymPy (para cálculos simbólicos), numpy (cálculos numéricos), matplotlib (gráficos) entre otros. En este curso sólo usaremos SymPy.

SymPy es una biblioteca de Python para matemática simbólica. Su objetivo es convertirse en un sistema de álgebra computacional (SAC) completo, manteniendo el código lo más simple posible para que sea comprensible y fácilmente extensible. SymPy está escrito enteramente en Python y no requiere de ninguna biblioteca externa.

SageMath es un sistema de software de matemáticas, libre, de código abierto bajo la licencia GPL. Es construído sobre muchos paquetes de código abierto existentes: NumPy, SciPy, matplotlib, SymPy, Maxima, GAP, FLINT, R y muchos más. Se acceda a su poder combinado a través de un lenguaje común, basado en Python.

2. Local y online

Se pueden usar todos los recursos anteriores de dos formas

1. Instalando el software necesario en una computadora. Nos referiremos a este modo como de acceso local.

2. A traves de trasacciones en línea que permiten usar una computadora remota que ejecuta las instrucciones y programas que se tipean en una página web con la que se interactúa usualmente por medio de un navegador. Hay varios sitios que ofrecen este servicio. Sugerimos la SageMathCloud. El usuario debe registrase.

3. Instalación local

Hay mucho software dedicado a gestionar el uso de python, recomendamos las siguientes por la sencillez de la instalación.

3.1. Windows

La distribución python(x,y) instala el interprete de python y todos los módulos de scipy. Además el entorno de desarrollo integrado (IDE) spyder.

Lamentablemente no es posible instalar SageMath en windows, sólo se instala bajo linux.

3.2. linux

Aquí todo es más sencillo, el interprete de python suele venir con la distribución del SO y se puede instalar los módulos, SymPy, NumPy, etc, recurriendo al administrador de paquetes o tipeando la sentencia adecuada en la línea de comandos. Para instalar SAGE se lo descarga de la página oficial y se descomprime.

3.3. Android

Qpython es una aplicación que permite ejecutar código python y una versión básica de sympy desde tablets y smartphones. Se descarga desde la plataforma google play.

3.4. Otros recursos de utilidad:

Tanto para linux o windows ipython, Anaconda, emacs.

4. Forma de trabajo: por medio de scripts e interactiva

Se puede trabajar de dos formas

1. Interactivamente, ingresando sentencias, de a una por vez, en la línea de comandos y obteniendo respuestas.
2. Haciendo un script (programa) donde se guardan todas las sentencias que se desea ejecutar. Posteriormente este script se puede ejecutar, ya sea desde la línea de comandos o en desde un IDE (spyder) oprimiendo un botón de ejecución.

5. Características del Lenguaje

Seguiremos en esta exposición a [1] de manera cercana. Las principales características del lenguaje son:

- Interpretado. Es necesario un conjutno de programas, el interprete, que entienda el código python y ejecute las acciones contenidas en él.
- implementa tipos dinámicos
- Multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

- Multiplataforma.
- Es comprendido con facilidad. Usa palabras donde otros lenguajes utilizarían símbolos. Por ejemplo, los operadores lógicos `!, || y \&\&` en Python se escriben not, or y and, respectivamente.
- El contenido de los bloques de código (bucles, funciones, clases, etc.) es delimitado mediante espacios o tabuladores.
- Empieza a contar desde cero (elementos en listas, vectores, etc).

6. Elementos del Lenguaje

6.1. Comentarios

Hay dos formas de producir comentarios, texto que el interprete no ejecuta y que sirve para entender un programa.

La primera, para comentarios largos es utilizanda la notación `''' comentario '''`.

La segunda notación utiliza el símbolo #, no necesita símbolo de finalización pues se extienden hasta el final de la línea.

```
'''
Comentario largo en un script de Python
'''
print "Hola mundo" # Comentario corto
```

El intérprete no tiene en cuenta los comentarios, lo cual es útil si deseamos poner información adicional en nuestro código como, por ejemplo, una explicación sobre el comportamiento de una sección del programa.

```
x = 1
x = "texto" # Esto es posible porque los tipos son asignados \
dinamicamente
```

6.2. Variables

Las variables se definen de forma dinámica, lo que significa que no se tiene que especificar cuál es su tipo de antemano y que una variable puede tomar distintos valores en distintos momentos de un programa, incluso puede tomar un tipo diferente al que tenía previamente. Se usa el símbolo = para asignar valores a variables. Es importante distinguir este = (de asignación) con el igual que es utilizado para definir igualdades en sympy, para ecuaciones por ejemplo.

6.3. Tipo de datos

Tipo	Clase	Notas	Ejemplo
`str`	Cadena	Inmutable	`'Cadena'`
`list`	Secuencia	Mutable, puede contener objetos de diversos tipos	`[4.0, 'Cadena', True]`
`tuple`	Secuencia	Inmutable, puede contener objetos de diversos tipos	`(4.0, 'Cadena', True)`
`dict`	Mapping	Grupo de pares clave:valor	`{'key1': 1.0, 'key2': False}`
`int`	Número entero	Precisión fija, convertido en *long* en caso de overflow.	`42`
`long`	Número entero	Precisión arbitraria	`42L` ó `456966786151987643L`
`float`	Número decimal	Coma flotante de doble precisión	`3.1415927`
`complex`	Número complejo	Parte real y parte imaginaria *j*.	`(4.5 + 3j)`
`bool`	Booleano	Valor booleano verdadero o falso	`True o False`

Se clasifican en:

Mutable si su contenido puede cambiarse.

Inmutable si su contenido no puede cambiarse.

Se usa el comando `\type` para averiguar que tipo de dato contiene una variable

```
>>> x=1
>>> type(x)
<type 'int'>
>>> x='Ecuaciones'
>>> type(x)
<type 'str'>
```

6.4. Listas y tuples

- Es una estructura de dato, que contiene, como su nombre lo indica, listas de otros datos en cierto orden. Listas y tuplas son muy similares.
- Para declarar una lista se usan los corchetes [], en cambio, para declarar una tupla se usan los paréntesis (). En ambos casos los elementos se separan por comas, y en el caso de las tuplas es necesario que tengan como mínimo una coma.
- Tanto las listas como las tuplas pueden contener elementos de diferentes tipos. No obstante las listas suelen usarse para elementos del mismo tipo en cantidad variable mientras que las tuplas se reservan para elementos distintos en cantidad fija.
- Para acceder a los elementos de una lista o tupla se utiliza un índice entero (empezando por ”0”, no por ”1”). Se pueden utilizar índices negativos para acceder elementos a partir del final.
- Las listas se caracterizan por ser mutables, mientras que las tuplas son inmutables.

```
>>> lista = ["abc", 42, 3.1415]
>>> lista[0] # Acceder a un elemento por su indice
'abc'
>>> lista[-1] # Acceder a un elemento usando un indice negativo
3.1415
>>> lista.append(True) # Agregar un elemento al final de la lista
>>> lista
['abc', 42, 3.1415, True]
>>> del lista[3] # Borra un elemento de la lista usando un indice
>>> lista[0] = "xyz" # Re-asignar el valor del primer elemento
>>> lista[0:2] # elementos del indice "0" al "2" (sin incluir ultimo)
['xyz', 42]
>>> lista_anidada = [lista, [True, 42L]] # Es posible anidar listas
>>> lista_anidada
[['xyz', 42, 3.1415], [True, 42L]]
>>> lista_anidada[1][0] # Acceder a un elemento de una lista dentro de
    otra lista
True
```

```
>>> tupla = ("abc", 42, 3.1415)
>>> tupla[0] # Acceder a un elemento por su indice
'abc'
>>> del tupla[0] # No es posible borrar ni agregar
( Excepcion )
>>> tupla[0] = "xyz" # Tampoco es posible re-asignar
( Excepcion )
>>> tupla[0:2] # elementos del indice "0" al "2" sin incluir
('abc', 42)
>>> tupla_anidada = (tupla, (True, 3.1415)) # es posible anidar
>>> 1, 2, 3, "abc" # Esto tambien es una tupla
(1, 2, 3, 'abc')
>>> (1) #  no es una tupla, ya que no posee al menos una coma
1
>>> (1,) # si es una tupla
(1,)
>>> (1, 2) # Con mas de un elemento no es necesaria la coma final
(1, 2)
>>> (1, 2,) # Aunque agregarla no modifica el resultado
(1, 2)
```

6.5. Diccionarios

- Para declarar un diccionario se usan las llaves \{\}. Contienen elementos separados por comas, donde cada elemento está formado por un par clave:valor (el símbolo : separa la clave de su valor correspondiente).
- Los diccionarios son mutables, es decir, se puede cambiar el contenido de un valor en tiempo de ejecución.

- En cambio, las claves de un diccionario deben ser inmutables. Esto quiere decir, por ejemplo, que no podremos usar ni listas ni diccionarios como claves.
- El valor asociado a una clave puede ser de cualquier tipo de dato, incluso un diccionario.

```
>>> dicci = {"cadena": "abc", "numero": 42, "lista": [True, 42L]}
>>> dicci["cadena"] # Usando una clave, se accede a su valor
'abc'
>>> dicci["lista"][0]
True
>>> dicci["cadena"] = "xyz" # Re-asignar el valor de una clave
>>> dicci["cadena"]
'xyz'
>>> dicci["decimal"] = 3.1415927 # nuevo elemento clave:valor
>>> dicci["decimal"]
3.1415927
>>> dicci_mixto = {"tupla": (True, 3.1415), "diccionario": dicci}
>>> dicci_mixto["diccionario"]["lista"][1]
42L
>>> dicci = {("abc",): 42} # tupla puede ser clave pues es inmutable
>>> dicci = {["abc"]: 42} # No es posible que una clave sea una lista
( Excepcion )
```

6.6. Listas por comprensión

Una lista por comprensión es una expresión compacta para definir listas. Al igual que el operador lambda, aparece en lenguajes funcionales. Ejemplos:

```
>>> range(5) #  "range" devuelve una lista, empezando en 0 \
y terminando con el numero indicado menos uno
[0, 1, 2, 3, 4]
>>> [i*i for i in range(5)]
[0, 1, 4, 9, 16]
>>> lista = [(i, i + 2) for i in range(5)]
>>> lista
[(0, 2), (1, 3), (2, 4), (3, 5), (4, 6)]
```

6.7. Funciones

- Las funciones se definen con la palabra clave `def`, seguida del nombre de la función y sus parámetros. Otra forma de escribir funciones, aunque menos utilizada, es con la palabra clave `lambda` (que aparece en lenguajes funcionales como Lisp). Generalemente esta forma es apropiada para funciones que es posible definir en una sola línea.
- El valor devuelto en las funciones con `def` será el dado con la instrucción `return`.

```
>>> def suma(x, y = 2): # el argumento y tiene un valor por defecto
...     return x + y # Retornar la suma
...
>>> suma(4) # La variable "y" no se modifica, siendo su valor: 2
6
>>> suma(4, 10) # La variable "y" si se modifica
14
```

```
>>> suma = lambda x, y = 2: x + y
>>> suma(4) # La variable "y" no se modifica
6
>>> suma(4, 10) # La variable "y" si se modifica
14
```

6.8. Condicionales

Una sentencia condicional (`if condicion`) ejecuta su bloque de código interno sólo si `condicion` tiene el valor bolleano `True`. Condiciones adicionales, si las hay, se introducen usando `elif` seguida de la condición y su bloque de código. Todas las condiciones se evalúan secuencialmente hasta encontrar la primera que sea verdadera, y su bloque de código asociado es el único que se ejecuta. Opcionalmente, puede haber un bloque final (la palabra clave `else` seguida de un bloque de código) que se ejecuta sólo cuando todas las condiciones fueron falsas.

```
>>> verdadero = True
>>> if verdadero: # No es necesario poner "verdadero == True"
...     print "Verdadero"
... else:
...     print "Falso"
...
Verdadero
>>> lenguaje = "Python"
>>> if lenguaje == "C":
...     print "Lenguaje de programacion: C"
... elif lenguaje == "Python": # Se pueden agregar "elif" como se
    quiera
...     print "Lenguaje de programacion: Python"
... else:
...     print "Lenguaje de programacion: indefinido"
...
Lenguaje de programacion: Python
>>> if verdadero and lenguaje == "Python":
...     print "Verdadero y Lenguaje de programacion: Python"
...
Verdadero y Lenguaje de programacion: Python
```

6.9. Bucles

El bucle `for` es similar a otros lenguajes. Recorre un objeto iterable, esto es una lista o una tupla, y por cada elemento del iterable ejecuta el bloque de código interno. Se define con la palabra clave `for` seguida de un nombre de variable, seguido de `in`, seguido del iterable, y finalmente el bloque de código interno. En cada iteración, el elemento siguiente del iterable se asigna al nombre de variable especificado:

```
>>> lista = ["a", "b", "c"]
>>> for i in lista: # Iteramos sobre una lista, que es iterable
...     print i
...
a
b
c
>>> cadena = "abcdef"
>>> for i in cadena: # Iteramos sobre una cadena, que es iterable
...     print i, # una coma al final evita un salto de linea
...
a b c d e f
```

```
>>> numero = 0
>>> while numero < 10:
...     print numero
...     numero += 1,  #un buen programador modificara las variables de
    control al finalizar el ciclo while
...
0 1 2 3 4 5 6 7 8 9
```

Referencias

[1] Wikipedia. Python, 2016.