Introducción a Python para todos.

Fernando Quintero A. quintero@mx1.ibm.com

IBM Systems IT Security Services
IBM de Mexico

Contents

- Introducción
 - El lenguaje Python

Índice

- Introducción
 - El lenguaje Python

El Lenguaje Python

- Lenguaje de propósito general Open Source.
- Orientado a Objetos, Procedural y Funcional.
- Facil de interactuar con C/ObjC/Java/Fortran.
- Facil de interactuar con C++ (via SWIG).
- Gran ambiente interactivo.
- Descargas: http://www.python.org
- Documentacion: http://www.python.org/doc/
- Libro Gratis: http://www.diveintopython.org

Versiones de Python en la actualidad.

2.7, 3.x ...

- version Actual es 2.7.x
- La nueva versión es la 3.x aún en adopción.
- Soporte de la versión 2.7 termina en 2020

Instalando y ejecutando Python.

- Python viene pre instalado en MacOSX y Linux.
- Los binarios de Windows se pueden descargar en http://python.org/
- ¡Fácil!

El intérprete de Python

La interfaz interactiva de python

```
% python
Python 2.7 (r25:51908, May 25 2012, 16:14:04)
[GCC 4.1.2 20061115 (prerelease) (SUSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El intérprete de python evalúa:

- El interprete siempre esta en : >>>
- Para salir de Python:
 - CTRL-Z ENTER

Ejecutando programas en UNIX.

% python archivo.py

 Se puede hacer un archivo ejecutable y agregar lo siguiente al principio del archivo : #!/usr/bin/env python para poder ejecutarlo.

Baterias incluidas

- Gran colección de módulos incluidos en la distribución estándar.
 - http://docs.python.org/modindex.html

numpy

- Ofrece capacidades de Matlab en Python
- Operaciónes rápidas de arreglos
- Arreglos 2D, Arreglos multi-D arrays, álgebra lineal etc.
- Decargass: http://numpy.scipy.org/
- Tutorial: http://www.scipy.org/Tentative_NumPy_Tutorial

matplotlib

Biblioteca de Plotting de gran calidad

```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green',
alpha=0.75)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
1 = plt.plot(bins, y, 'r-', linewidth=1)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Descargas: http://matplotlib.sourceforge.net/

PyFITS (astronomía)

• FITS I/O made simple:

```
>>> import pyfits
>>> hdulist = pyfits.open('nput.fits'
>>> hdulist.info()
Filename: test1.fits
No. Name Type Cards Dimensions Format
O PRIMARY PrimaryHDU 220 () Int16
1 SCI ImageHDU 61 (800, 800) Float32
2 SCI ImageHDU 61 (800, 800) Float32
3 SCI ImageHDU 61 (800, 800) Float32
4 SCI ImageHDU 61 (800, 800) Float32
>>> hdulist[0].header['argname'
'NGC121'
>>> scidata = hdulist[1].data
>>> scidata.shape
(800, 800)
>>> scidata.dtype.name 'loat32'
>>> scidata[30:40,10:20] = scidata[1,4] = 999
```

• Descargas:

http://www.stsci.edu/resources/software_hardware/pyfits

Lo Básico

Un código de ejemplo.

```
x = 34 - 23  # Un comentario.
y = "Hola"  # Otro.
z = 3.45
if z == 3.45 or y == "Hola"
    x = x + 1
    y = y + "Mundo"
print x
print y
```

Entendiendo el código anterior

- Asignación usa = y comparación usa ==.
- Para números + * / % .
 - Uso especial de + para la concatenación de cadenas.
 - Uso especial de % para el formateo de cadenas (como printf en C)
- Los operadores lógicos son palabras (and, or, not) no símbolos.
- El comando básico de impresión es print.
- La primera asignación a la variable la crea.
 - Los tipos de variables no necesitan ser declaradas.
 - Python determina el tipo de variable por su cuenta.

Tipos Básicos (Datatypes)

• Entero (por defecto para números)

```
z = 5 / 2 # La respuesta es 2, división de enteros.
```

Flotantes

```
x = 3.456
```

- Cadenas
- Se puede usar "" o " para especificar

```
"abc" 'abc' (Lo mismo.)
```

• Se pueden especificar apóstrofes sin pareja dentro de las cadenas.

```
"matt's"
```

• Use triple dobles comillas para cadenas multi-línea o cadenas que contengan ambos ' y ' dentro de ellos:

```
"""a'b"c'""
```

Espacios en Blanco

- Los espacios en blanco son significativos en Python: especialmente en la indentación y líneas nuevas.
 - Use nueva linea para terminar una línea de código.
 - Se usa \ cuando se deba ir a la siguiente línea de forma prematura.
- No se usan corchetes { } para marcar bloques de código en Python ... Se usa indentación consistente en su lugar.
 - La primera línea con menos indentación esta fuera del bloque.
 - La primera línea con mas indentación inicia un bloque anidado.
- Los dos puntos aparecen al principio de un bloque nuevo.
 - (Ej.para definiciones de clases o funciones.)

Comentarios

- Los comentarios inician con # El resto de la línea se ignora.
- Se puede incluir una cadena de documentación como la primera línea de cualquier función o clase que se defina.
- El ambiente de desarrollo, depuración, y otras herramientas lo usan: siempre es un buen estilo incluir uno..

```
def mi_funcion(x, y):
    """Esta es una cadena de documentación. Esta
función hace .... '""
# El código va aquí....
```

Asignaciones

- Asignar una variable en Python significa establecer un nombre que contenga una referencia a algún objeto.
 - La asignación crea referencias, no copias.
- Los nombres en Python no tienen un tipo intrínseco. Los Objetos tienen tipos.
 - Python determina el tipo de la referencia automáticamente basado en objeto de datos asignado a él.
- Se crea un nombre la primera vez que aparece en el lado izquierdo de una expresión de asignación:

$$x = 3$$

• Una referencia se borra a través de garbage collection después que cualquier nombre ligado a él esta fuera del alcance.

Accesando nombres no existentes

• Si se trata de accesar un nombre antes de que sea creado apropiadamente (al ponerlo en el lado izquierdo de una asignación) , se obtendrá un error.

```
>>> y
Traceback (most recent call last):
    File "<pyshell#16>", line 1, in -toplevel
          y
NameError: name " is not defined
>>> y = 3
>>> y
3
```

Asignación múltiple

• Se pueden hacer asignaciones a múltiples nombres al mismo tiempo.

Reglas de nomenclatura

 Los nombres son sensibles a mayúsculas o minúsculas y no pueden iniciar con un número. Pueden contener letras, números y guion bajo.

```
bob Bob _bob _2_bob_ bob_2 BoB
```

• Existen algunas palabras reservadas:

```
and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while
```

Tipos de secuencia

- Tupla
 - Una secuenca ordenada inmutable de articulos
 - los artículos pueden ser de tipo mixto, incluyendo colecciones de tipos.
- Cadenas
 - Inmutable
 - Conceptualmente similar a las tuplas.
- Lista
 - Mutable secuencia ordenada de articulos de tipo mixto.

Sintáxis similar

- Los tres tipos de secuencias (tuplas, cadenas, and listas) comparten mucho de la misma sintaxis y funcionalidad.
- Diferencias principales:
 - Tuplas and cadenas son inmutables
 - Las listas son mutables
 - Las operaciones que se muestran en esta sección pueden ser aplicadas a todas los tipos de secuencia.
- La mayoría de ejemplos muestran la operacion sobre uno de ellos.

Tipos de secuencias 1

• Las tuplas se definen usando parentesis (y comas).

>>>
$$tu = (23, 'abc', 4.56, (2,3), 'def')$$

• Las listas se definen usando corchetes (y comas).

• Las cadenas se definen usando comillas (", ', o """).

```
>>> st = "Hola Mundo"
```

>>> st = """esta es una cadena multilínea
que usa tres comillas."""

Tipos de secuencia 2

- Se pueden accesar miembros individuales de una tupla, lista o cadena usando un notación de corchetes.
- Nótese que todas utilizan como base el índice 0.

Indices positivos y negativos.

>>>
$$t = (23, 'abc' 4.56, (2,3), 'def')$$

Índice positivo: se cuenta desde la izquierda, empezando con 0.

Búsqueda negativa: Se cuenta de la derecha, comenzando con -1.

Secciones: Regresar copia de un subconjunto 1

```
>>> t = (23, 'abc' 4.56, (2,3), 'def')
```

Regresar una copia del contenedor con un subset de los miembros originales. Se inicia copiando en el primer índice y se detiene <u>antes</u> del segundo índice.

```
>>> t[1:4]
('abc' 4.56, (2,3))
```

También se pueden usar índices negativos en subconjuntos

```
>>> t[1:-1]
('abc' 4.56, (2,3))
```

Secciones: Regresando copia de un subconjunto 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Se omite el primer índice para a hacer una copia desde el principio del contenedor.

```
>>> t[:2]
(23, 'abc')
```

Se omite el segundo índice para hacer una copia iniciando desde el primer índice hasta el final del contenedor.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copiando la secuencia entera.

Para hacer una copia de una secuencia entera, se puede utilizar [:]

```
>>> t[:]
(23, 'abc' 4.56, (2,3), 'def')
```

Nótese la diferencia entre esas dos líneas para secuencias mutables:

El operador 'in'

 Prueba boleana para saber si un valor esta dentro de un contenedor.

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

Para cadenas, pruebas para subcadenas

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

• Precaución: el operador in también se usa en la sintaxis de bucles for y listas .

El operador +

• El operador + produce una nueva tupla, lista, o cadena cuyo valor es la concatenación de sus argumentos.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> "Hola"+ " "+ "Mundo"
'Hola Mundo'
```

El operador *

• El operador * produce una nueva tupla, lista o cadena que "repite" el contenido original.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> "HolaHolaHola"
```

Tuplas: Inmutables

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
Traceback (most recent call last):
File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

No se puede cambiar una tupla.

Se puede hacer una nueva tupla y asignar su referencia a un nombre usado previamente

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Listas: Mutables

```
>>> li = ['abc' 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc' 45, 4.34, 23]
```

- Se pueden cambiar las listas.
- El nombre li aún apunta a la misma referencia de memoria cuando terminamos.
- La mutabilidad de las listas significa que no son tan rápidas como las tuplas.

Operaciones sólo en listas 1

El método extend vs el operador +

- + Crea una lista nueva (con una nueva referencia de memoria)
- extend opera en la lista li en la misma referencia.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, "3, 4, 5, "9, 8, 7]
```

Confuso:

- Extend toma una lista como un argumento.
- Append toma un singleton como un argumento.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, "3, 4, 5, "9, 8, 7, [10, 11, 12]]
```

Operaciones sólamente en listas 3

Operaciones sólo en listas 4

Tuplas vs Listas

- Las listas son mas lentas pero mas poderosas que las tuplas
 - Las listas pueden ser modificadas, y tiene muchas operaciones prácticas que podemos ejecutar en ellas.
 - Las tuplas son inmutables pero tienen menos características.
- Para convertir entre listas y tuplas utilice las funciones list() y tuple()

```
li = list(tu)
tu = tuple(li)
```

Diccionarios: Un tipo de mapeo.

- Los diccionarios almacenan un mapeo entre una serie de llaves y una serie de valores.
 - Las llaves pueden ser de cualquier tipo inmutable.
 - Los valores pueden ser de cualquier tipo.
 - Un sólo diccionario puede almacenar valores de diferentes tipos.
- Se puede definir, modificar, ver, buscar y borrar los pares llave-valor en el diccionario.

Usando diccionarios

```
>>> d = {'user': 'bozo', 'pswd':1234}
>>> d['user']
'hozo'
>>> d['pswd']
1234
>>> d['bozo']
Traceback (innermost last):
File '<interactive input>' line 1, in ?
KeyError: bozo
>>> d = {'user': 'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'user': 'clown', 'pswd':1234}
>>> d['id'] = 45
>>> d
{'user': 'clown', 'id':45, 'pswd':1234}
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user'] # Remueve uno.
>>> d
{'p':1234, 'i':34}
>>> d.clear()
                 # Remueve todos.
>>> d
1
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()
                    # Lista de llaves.
['user','p','i']
>>> d.values()
                      # Lista de valores.
['bozo' 1234, 34]
>>> d.items()
                      # Lista de tuplas.
[('user', 'bozo'), ('p', 1234), ('i', 34)]
```

Funciones

- def crea una función y le asigna un nombre.
- return regresa un resultado.
- Los argumentos son pasados por asignación.
- Los argumentos y tipos de regreso no se declaran.

```
def <nombre>(arg1, arg2, ..., argN):
    return <valor>
def veces(x,y):
    return x*y
```

Pasando argumentos a funciones.

- Los argumentos se pasan por asignación.
- Los argumentos que se pasan se asignan a nombres locales.
- Los nombres de los argumentos asignados no afectan a quien lo llama.
- Cambiando un argumento mutable puede afectar a quien lo llama.

Argumentos opcionales.

• Se pueden definir valores por defecto para argumentos que no necesitan ser especificados.

```
def func(a, b, c=10, d=100):
    print a, b, c, d
>>> func(1,2)
    1 2 10 100
>>> func(1,2,3,4)
    1,2,3,4
```

Gotchas

- Todas las funciones en Python tienen un valor de retorno
 - aun si no hay un return dentro del código.
- Las funciones sin ningún valor de retorno regresan el valor especial de None.
- No existe "sobrecarga" de funciones en Python.
 - Dos funciones diferentes no pueden tener el mismo nombre, aún cuando tengan diferentes argumentos.
- Las funciones pueden ser usadas como cualquier otro tipo de datos estas pueden ser:
 - Argumentos a función.
 - Regresar valores de funciones.
 - Asignarse a variables.
 - Partes de tuplas, listas, etc.

Control de flujo

Ejemplos:

```
1.)
if x == 3:
    print "X igual a 3."

elif x == 2:
    print "X igual a 2."

else:
    print "X igual a otra cosa."

print "esto esta fuera del 'if'"
```

```
2.)
x = 3
while x < 10:
       if x > 7:
        x += 2
       continue
      x = x + 1
      print "Aun en el bucle."
       if x == 8:
        break
print "Fuera del bucle."
3.)
for x in range(10):
       if x > 7:
        x += 2
       continue
       x = x + 1
        print "Aun en el bucle"
       if x == 8:
         break
print "Fuera del bucle."
```

¿Porqué usar módulos?

- Reuso de código.
 - Las rutinas pueden ser llamadas múltiples veces dentro de un programa.
 - Las rutinas pueden ser utilizadas por múltiples programas.
- Particionamiento de Namespace.
 - Agrupar datos junto con funciones utilizadas para esos datos.
- Implementar servicios compartidos o datos.
 - Puede proveer estructuras globales de datos que se accesan por múltiples subprogramas.

Módulos

- Los módulos son funciones y variables definidos en archivos separados.
- Los artículos son importados utilizando from o import.

```
from module import function
function()
import module
module.function()
```

- Los módulos son Namespaces.
 - Se pueden utilizar para organizar nombres de variables, p.e.
 atomo.posicion = atomo.posicion molecula.posicion

Clases y Objetos

¿Qué es un objeto?

- Una pieza de software que contiene variables y métodos.
- El diseño orientado a objetos se enfoca en:
 - Encapsulación:
 - Dividir el código en una interfaz pública y una implementación privada de dicho interfaz.
 - Polimorfismo:
 - La habilidad de sobrecargar operadores estándar de tal manera que tengan el comportamiento apropiado basado en su contexto.
 - Herencia:
 - La habilidad de crear subclases que contengan especializaciones de sus padres.

Clase de Ejemplo

```
class atomo(object):
      def init (self.noat.x.v.z):
             self.noat = noat
             self.posicion = (x,y,z)
      def simbolo(self):
             NoAt_a_Simbolo = { 1: 'H', 2:'He', 3:'Li', 4:'Be',5:'B', 6:'C' }
               return NoAt_a_Simbolo[self.noat]
      def __repr__(self): # sobrecarga impresion
             return '%d %10.4f %10.4f %10.4f'%(self.noat, self.posicion[0],
               self.posicion[1],self.posicion[2])
      def str (self): # sobrecarga cadena
             return '%d %10.4f %10.4f %10.4f'%(self.noat, self.posicion[0].
               self.posicion[1],self.posicion[2])
>>> at = atomo(6,0,0,1,0,2,0)
>>> print at
      6 0.0000 1.0000 2.0000
>>> at.simbolo()
       , C;
```

Clase Átomo

- Sobrecarga el constructor por omisión.
- Define las variables de la clase (noat,posicion) que son persistentes y locales al objeto atomo.
- La mejor manera de manejar memoria compartida:
 - En lugar de pasar una lista larga de argumentos, encapsular algunos de estos datos en un objeto y pasar el objeto.
 - Resultados mucho mas limpios.
- Sobrecarga el operador print
- Ahora queremos usar la clase atomo para construir moléculas.

Clase Molécula

```
class molecula:
     def init (self.nombre='Generico'):
          self.nombre = nombre
          self.lista atomo = []
     def agrega_atomo(self,atomo):
          self.lista_atomo.append(atomo)
     def __repr__(self):
          cadena = 'Esta es una molecula llamada %s\n' % self.nombre
          cadena = cadena +'Tiene %d atomos\n' % len(self.lista_atomo)
          for atomo in self.lista atomo:
           cadena = cadena + str(atomo) + '\n'
          return cadena
```

Usando la Clase Molécula

```
>>> mol = molecula('Agua')
>>> at = atomo(8,0.,0.,0.)
>>> mol.agrega_atomo(at)
>>> mol.agrega_atomo(atomo(1,0.,0.,1.))
>>> mol.agrega_atomo(atomo(1,0.,1.,0.))
>>> print mol
Esta es una molecula llamada Agua
Tiene 3 atomos
8 0.000 0.000 0.000
1 0.000 0.000 1.000
1 0.000 1.000 0.000
```

 Nótese que la funcion print llama la función print de atomos.

• Reuso de código: sólo se tiene que escribir el código que imprime un átomo una sóla vez; esto significa que si se cambia la especificación del átomo, solo se tendrá que modificar en un solo lugar.

Herencia

```
class qm_molecula(molecula):
    def agrega_base(self):
        self.base = []
        for atomo in self.lista_atomo:
        self.base = agrega_bf(atomo,self.base)
```

- __init___, __repr___, y __agrega_atomo___ se toman de la clase padre (molecula)
- agrega una nueva funcion agrega_base() para agregar un set base
- Otro ejemplo de reuso de código.
 - Las funciones básicas no tienen que ser reescritas, sólo heredadas.
 - Menos que reescribir cuando las especificaciones cambian.

Sobrecarga

```
class qm_molecula(molecula):
    def __repr__(self):
        cadena = 'QM Rifa!\n'
        for atomo in self.lista_atomo:
        cadena = cadena + 'atomo' + '\n'
        return cadena
```

- Ahora sólo heredamos __init__ y agrega_atomo del padre.
- Definimos una nueva versión de __repr__ especialmente para QM.

Agregamos a la función padre

 Algunas veces se quiere extender, mas que reemplazar las funciones padre.

```
class qm_molecula(molecula):
    def __init__(self,nombre="Generico",base="6-31G**"):
        self.base = base
        molecula.__init__(nombre)
```

Datos públicos y privados

 En Python cualquier cosa con dos guiones bajos es privado.

 Cualquier cosa con un guion bajo es semi-privado, y te deberías sentir culpable por accesar estos datos directamente.

 Algunas veces es útil como un paso intermedio para convertir los datos en privados.

E/S Archivos, Cadenas, Excepciones

```
>>> try:
 ... 1 / 0
 ... except:
         print('Eso fue tonto!')
    finally:
         print('Esto se ejecuta de cualquier manera')
 Eso fue tonto!
Esto se ejecuta de cualquier manera
fileptr = open('archivo')
somestring = fileptr.read()
for line in fileptr:
print line
fileptr.close()
>>> a = 1
>>> b = 2.4
>>> c = 'Juan'
>>> '%s tiene %d monedas por un total de $%.02f' % (c, a, b)
 'Juan tiene 1 monedas por un total de $2.40'
```

¡Gracias!

¿Dudas?

http://github.com/fquinteroa/Introduccion_a_Python Fernando Quintero IBM de Mexico quintero@mx1.ibm.com