Python 2.*

Ondiz Zarraga

6 de octubre de 2015

Resumen

Índice

1. Sobre el lenguaje

- Interpretado
- Indentación obligatoria
- Distingue mayúsculas - minúsculas
- No hay declaración de variables (*dynamic typing*)
- Orientado a objetos
- Garbage colector: quita los objetos a los que no haga referencia nada
- Comentarios → #
- Imprimir en pantalla → print a,b (si no se quiere salto de línea coma al final). Para darle formato (igual que C):
  - Float: %dígitos.dígitos f variable
  - String: % s variable
  - Notación científica: %dígitos.dígitos E variable
  - print *'%s letra %s' (a,b)* escribe *a letra b*. Identifica los % con los elementos del tuple.
- Nombres de variables:
  - Solo pueden empezar por _ o letra, luego cualquier carácter
  - Diferencia mayúsculas y minúsculas (*case sensitive*)
  - No puede ser igual que palabra reservada: *and, del, fo, is, raise, asser, elif, from, lambda, return, break, else, global, not, try, class, except, if, or while, continue, exec, import, pass, yield, def, finally, in, print*
- En línea de comandos, la variable _ guarda el valor de la última operación

2. Tipos de variables

2.1. Números

2.1.1. Enteros

- Int. Trata cualquier nº sin .0 como entero (cuidado con la división)

2.1.2. Reales

- Float, coma flotante
- Hay números complejos: a + bj (ver 7)

2.1.3. Booleans

- Se definen los enteros 0 y 1 como True y False pero SIGUEN SIENDO enteros
- Cualquier objeto diferente de 0 o que no esté vacío es True. Cualquier objeto igual a 0 o vacío es False
- Operadores booleanos: < / > devuelven 0/1; *and* y *or* devuelven objeto[1] (el primero que sea True), *short circuit*

2.2. Strings

- Se definen con comillas simples o dobles. Así pueden incluirse comillas simples/dobles en el propio string: "Knight's" // 'Knight"s' [2]
- Se pueden usar comillas triples (' ' ' o " " ") para textos de más de una línea. Coge todo lo que haya entre las comillas, incluidos Enter o Tab. También para documentación (ver 6.4). Son útiles para desactivar trozos de código en lugar de comentar línea a línea con #.

2.3. Listas

- Colección de datos de cualquier tipo
- Mutable
- Definición → lista =[...,...,...]
- Los índices empiezan en cero: elemento nº 1 → lista[0]

[1] Es lo mismo porque pueden usarse objetos en *if*s o *while*s. Además, como *or* devuelve objeto puede usarse para elegir un objeto no vacío/nulo de una lista: *X = A or B or C or None*

[2] 'Knight\'s' es equivalente, \funciona como tecla de escape. Para tabulaciones: \t. Saltos de línea: \n. Para ignorar las teclas de escape en las direcciones: *open(r 'C:...')*, (la *r* ignora las teclas de escape) o *'C:\\..'* (doble barra)

- Se pueden anidar listas para definir matrices: M=[[...,...],[...,...]] [3] *TRUCO:* Para coger *columnas: col = [row[nº columna] for row in M]*
- Se pueden convertir strings en listas con *lista = list(string)*
- Métodos útiles: *lista.append*(elemento a añadir) añade elemento[4] (más en 4) (!) Añade UN elemento: en el caso lista.append([1,2,3]) el elemento n+1 de lista será la lista [1,2,3]. Para que añada los elementos de una lista a otra lista: *lista.extend([lista])* o lista1 + lista2 (maravillas del overload :D); *extend* es más rápido porque sólo modifica una lista, no crea una nueva.
- Memoria: si se hace B = A, ambos hacen referencia al mismo objeto, si se cambia uno cambia los dos [5]. Para que esto no ocurra B = A[:] (hacen referencia a distinto lugar de la memoria)
- Para prealquilar memoria: L = [None] * 100 # lista de 100 variables
- Filtrado de listas: *lista_filtrada= [elem for elem in lista_inicial if condición]* Si la condición es verdadera para un elemento, se incluye dicho elemento en la lista filtrada.

2.4. Tuples

- Lista no mutable. Útil como etiqueta en diccionario(no se pueden usar listas)
- No tienen métodos
- Tuple de un único elemento: (elem,) Hace falta la coma (en las listas no)
- Tuple assigment: a,b = c,d

2.5. Diccionarios

- Colección de datos - etiquetas
- Definición → *D ='etiqueta1': valor1, ...*
- Las etiquetas pueden ser strings o tuples
- Están desordenados, así que se busca por etiqueta: *D[etiqueta]*
- Se puede hacer uno vacío e ir rellenando. Se rellena igual que se busca.
- Diccionarios para representar sparse matrix: D = (tuple de coordenadas): valor (lo demás 0). Ejemplo: (2,3,4):88, (7,8,9):99
- Métodos útiles: *D.keys()* devuelve las etiquetas; *D.haskey(etiqueta)* devuelve True si el diccionario tiene esa etiqueta; *D.pop(elemento)* borra este elemento (también válido en listas)

[3] Para definir matrices eficientemente usar Numpy
[4] CUIDADO: devuelve None, no asignar
[5] Equivalente a *static* en Java y C

3. Sintaxis

- Después de *if / for / def / ...* → :
- Final de línea = final de expresión. Para dividir una sentencia en varias líneas: \. Las siguientes líneas se pueden indentar de ∀ manera. (Las expresiones entre corchetes/llaves/paréntesis también pueden ocupar varias líneas sin \)
- Final de indentación [6] = final de bloque

3.1. Condiciones: *if - elif - else*

Sintaxis:

if condición:
    expresión1
    expresión2

elif condición:
    expresión alternativa1
else:
    expresión alternativa2

- Si sólo tiene una condición puede hacerse en una única línea: *if* condición: expresión
- No hay *case*, se usan diccionarios
- Sintaxis alternativa: *A = Y if X else Z*[7] equivale a:

    if X:
        A = Y
    else:
        A = Z

3.2. Bucles

Para parar un bucle → Ctrl + C

3.2.1. Bucles *for*

Sintaxis:

for contador *in* objeto :
    expresiones
else:
    expresiones

- Se puede iterar en strings, listas, tuples ...
- Se le pueden poner *break*s y *continue*s (ver 3.2.2)

[6] Cualquier indentación es válida mientras sea coherente en todo el bloque
[7] Y ? X : Z de C y Java

- Se puede vectorizar en lugar de usar bucles *for* para ganar velocidad. Ejemplo:
  1. *cuadrados = [x**2 for x in [1, 2, 3, 4]]*(puede anidarse un *if*)
  2. *cuadrados = [] # lista vacía*

     for x *in* [1,2,3,4]:
         *cuadrados.append(x**2)*

3.2.2. Bucles *while*

Sintaxis:

while condición:
    expresión1
    expresión2

try: (como un *try* - *catch* de Java)
except:
    maneja error

Excepciones dentro del while:

while condición1:
    expresiones
if condición2 : *break # sale del while, no ejecuta else*
if condición3 : *continue # vuelve al while*
else:
    expresiones *# ejecuta si no ha ejecuta un break*

3.2.3. Bucles *do while*

- No hay bucles *do while*
- Para hacer un equivalente a un *do while*

  while True:
      expresiones
      if exitTest() : break

3.3. Operadores

- **Suma** → + , funciona como suma para el caso de los números y como concatenación para los strings y listas (L = [1,2] → L = L +[3] → L = [1,2,3])
- **Multiplicación** → * , multiplicación entre números y repetición para listas y strings. Ej: 'Spam'*3 → 'SpamSpamSpam'
- **División** → /, división entera entre int (truncando [8]) y división entre cualquier combinación float - int.

[8] Para que divida normal entre dos enteros .0 o hacer un cast

- **Módulo** → %, el resto en una división entera
- **Exponente** → **
- **Slice** → objeto[*desde donde:paso: hasta donde*]. No coge el último elemento. Si el último número es negativo va hacia atrás. [9]
- **Cast**: convertir un tipo de objetos en otro. En entero: *int(dato)*. En real: *float(dato)*. En string: str(dato). En lista: list(tuple). En tuple: tuple(lista).
- Operador **in**: *if* x *in* y. Devuelve True si x pertenece a y. Ejemplos:
  - *'m' in 'Spam'* → True
  - Para leer un fichero: *for* line *in open('file.txt')*:
  - Para iterar en un diccionario: *for* key *in D.keys()*: o *for* key *in* D :

[9] A[:] coge todos los elementos

4. Métodos interesantes

- range(inicio, fin, paso) → devuelve una lista. Puede usarse en bucles *for*: *for* i *in* range(3): *# para repetir 3 veces*
- len(lista) → devuelve la longitud de una lista
- sorted(lista) → ordena y devuelve lista ordenda
- sum(lista) → suma los elementos
- any(lista) → True si hay algún elemento no vacío
- all(lista) → True si TODOS no vacíos
- zip(lista1, lista2) → junta el primer elemento de lista1 y el de lista2 en un tuple, el segundo con el segundo etc. Por ejemplo:

  lista1 = [1,2,3,4]
  lista2 = [5,6,7,8]
  zip(lista1, lista2) # queda [(1,5),(2,6),(3,7),(4,8)]

  - Funciona con listas y strings
  - Vale para hacer diccionarios
  - Para acción paralela: *for*(x,y) *in* zip(lista1, lista2)
- enumerate(lista)→ [(nº,valor),(nº,valor),...], enumera todos los elementos de una lista
- filter(secuencia)→ aplica una condición a una secuencia y devuelve los objetos para los que es True
- map(función, secuencia)→ aplica una función a una secuencia de objetos. Ej:

  def inc(x): *return* x + 10
  map(inc,counters) # suma 10 a todos los elementos de counters
- is → a *is* b, para ver si dos objetos son el mismo. Si dos objetos son el mismo y son mutables, cambios en uno afectan a ambos.
- execfile('script.py')→ ejecuta el script (si está en el Current Directory)

5. Ficheros

- Para escribir:
  f = open('nombre.txt','w') # modo escribir
  f.write('texto')
  f.close() # se crea el fichero
- Para leer: (se lee como string)
  f = open('nombre.txt')
  string = f.read()

  Dos opciones:

  string # muestra el contenido
  print string # muestra el contenido con formato
- Para convertir cosas de un fichero en objetos Python, ejemplos:
  1. **Strings**:
     F = open('file.txt')
     line = F.readline() # lee línea
     line.rstrip() # quita \n
  2. **Números**: '44,44,45 \n'
     F = open('file.txt')
     line = F.readline() # lee línea
     parts = line.split[10]*(',') # parte por la coma, quedaría ['43','44','45\n']*
     numbers = [int(P)] for P in parts # convierte a número
  3. **Código**: '[1,2,3]${'a':1, 'b':2 }\n' *F = open('file.txt')*
     line = F.readline() # lee línea
     parts = line.split('$') # parte por $
     eval(parts[0]) # convierte a cualquier tipo de objeto (trata como código) #
     objects = [eval(P) for P in parts] para coger todo. Queda [[1,2,3],'a':1,'b':2]
  4. **Pickle**: Para el caso que *eval()* no sea fiable, se puede usar el módulo *pickle*:
     F = open('file.txt','w')
     import picle
     pickle.dump(D,F) # coge cualquier objeto de la fila
     F.close()
- Hay que poner \n para saltar línea
- Si no se cambia el path guarda los ficheros en C:\Temp

[10] string.split('carácter')→ convierte string en lista de palabras cortando por el carácter indicado. Para convertir lista en string: delimiter.join(lista) (delimiter es el carácter que une)

6. Funciones

Definición de funciones:

def nombre(arg1, arg2,...) # *definición*
    expresiones
    return valor1, valor2 # *devuelve el valor*

- Las funciones son código ejecutable, pueden definirse dentro un *if/for/while...* (son objetos)
- Las variables dentro de una función sólo son visibles dentro de la propia función.
- Los argumentos inmutables se pasan por valor y los objetos con puntero
- Se pueden asignar valores a los argumentos en la propia llamada: *func(a=3)*
- Memoization: guardar valores conocidos de una función en un diccionario para ahorrar tiempo de cálculo

6.1. Argumentos por defecto

Se pueden incluir valores por defecto en la definición de una función, por ejemplo: *def* func(a,b=2,c=3). El primero requerido, si no se da el valor de los demás se utilizan los por defecto. Si se mete sólo $a \rightarrow b$ y c por defecto; con a y $b \rightarrow c$ por defecto; con a,b y $c \rightarrow$ ninguno por defecto.

6.2. Recepción de argumentos

Hay diferentes maneras de recibir los argumentos de una función:

- La típica: *def* func(a,b=2,c=3)
- *def* func(*args)→ coge todos los argumentos y los mete en un tuple
- *def* func(**args)→ coge los argumentos del tipo a=3 y los mete en un diccionario (vale para desempaquetar argumentos si se usa en la llamada a la función)
- Se pueden combinar diferentes modos de recibir argumentos en el siguiente orden: a, a=3, *args, **args2

6.3. Funciones anónimas o lambdas

Sintaxis:

lambda arg1, arg2,...,argN
    expresión que usa los argumentos

Se pueden asignar a un objeto: $f = lambda\ x,y,z : x + y + z$
f(2,3,4)

6.4. Ayuda de las funciones (docstring)

Línea entre comillas triples después de la definición. Es un atributo de la función. Se pide con el método *función.__doc__*

7. Módulos

- Dos modos para importar:
  1. *import módulo*
  2. *from módulo import** (para importar todo el módulo) o *from módulo import función*.

  La diferencia entre las dos es que para (2) se llama a las funciones por su nombre, sin especificar el módulo al que pertenecen; no distingue entre funciones con el mismo nombre, llama a la última importada (los métodos y atributos del método importado se sitúan en el espacio de nombres local). para el caso (1), en cambio, se llama a las funciones como: *módulo.función()* [11]
- Los módulos se crean igual que los scripts, sólo que en lugar de llamarlos hay que importarlos (según (1) o (2), sin .py)
- Si se cambia un módulo hay que volverlo a cargar: *reload(módulo)*
- Módulos útiles:
  - **Math**: funciones matemáticas ($\pi \rightarrow$ math.pi())
  - **Cmath**: módulo para funciones complejas (a.conjugate(), a.real(), a.imag() ...)
  - **Random**: números aleatorios reales o enteros, elegir de lista ...(nº random $\rightarrow$ random.random())
  - **Debugger**: pdb
  - **Numpy**: módulo para tratar matrices eficientemente: *array([[1,2],[3,4]])*(aquí la coma entre las dos listas representa salto de fila). Hay que instalarlo
  - **Scipy**: para funciones científicas. Hay que instalarlo.
  - **Pychecker**: para ver errores. Importarlo antes del módulo que se quiere verificar:
    from pychecker *import* checker
    import módulo
  - **Timeit**: para medir tiempos: *timeit.timeit('función')*
  - **cProfile**: para medir cuellos de botella. Ejecutarlo en el *main* después de importarlo: *cProfile.run('main()')*
  - **Matplotlib**: paquete para dibujar gráficos. Hay que descargarlo.

Si un módulo tiene funciones y código y sólo se quieren importar las funciones y que no ejecute el código: *if* __name__ = '__main__' después de la definición de clase/métodos. Así sólo se ejecuta lo del programa principal.

[11] Se aconseja sólo importar un módulo con *from modulo import** en cada sesión para no tener problemas con funciones con el mismo nombre

8. OOP

- Ver tipo de objeto: *type(objeto)*
- Ver los atributos de un objeto: *dir(objeto)*. Con *dir(__builtins__)* se ven las funciones y atributos que hay antes de importar ningún módulo.
- Ver los métodos aplicables a un objeto *objeto.__methods__*
- Ver lo que hace un método: *help(objeto.método)*

8.1. Clases

- Definición de clases: *class* Nombre(tipo):
- Atributos: objeto.atributo = valor
- Para crear una instancia de una clase: a = Nombre()
- Métodos: funciones definidas dentro de una clase. Para llamarlos:
  1. clase.método(objeto al que se le aplica[12])
  2. objeto al que se le aplica.método()
- Hay overloading y polimorfismo

8.2. Herencia

- *class* Hijo(Padre)
- La subclase coge los métodos de la superclase
- Si se vuelve a escribir un método en la subclase, overload
- Admite herencia múltiple: Hijo(Padre1,Padre2,...)
- Relaciones posibles entre clases:
  - HAS-A: referencia a otros objetos
  - IS-A: herencia
  - DEPENDS-ON: cambios en una clase provocan cambios en otras

8.3. Métodos de sobrecarga de operadores

Los más comunes:

- Método __init__, para inicializar, valores por defecto. CONSTRUCTOR (∼ Java)
- Método __del__, para reclamar un objeto. DESTRUCTOR
- Método __add__, +

[12] Al objeto al que se le aplica un método (lo que va antes del punto) se le llama parámetro. Al primer parámetro de un método se le llama *self* por convenio

- Método __or__, — / OR
- Método __str__, para imprimir (∼ Java)
- Método __call__, llamada a funciones, X()
- Método __getattr__, x.atributo
- Método __setattr__, x.atributo = valor
- Método __getitem__, para indexar, x[key]
- Método __setitem__, asignar valor a una posición, x[key] = valor
- Método __len__, longitud
- Método __cmp__, comparación, == / <>
- Método __lt__, <
- Método __eq__, ==
- Método __radd__, suma por la derecha
- Método __iadd__, x+
- Método __iter__, iteraciones (bucles *for*, vectorización con *in*, *map*)

9. Manejo de errores