Introducción a Big Data con Python

Programación en Python. Tipos básicos y colecciones

Jesús García López de Lacalle

27 de septiembre de 2016

Tipos básicos

- Enteros (inmutables):
  - *int* de Python = *long* de C (depende de la plataforma)
    n = 23
  - *long* de Python = números arbitrariamente largos
    n = 23L
- Reales (inmutables):
  - *float* de Python = *double* de C
    x = 0.2703
    y = 0.1e-3
  - *decimal* de Python = permite ajustar la precisión
- Complejos (inmutables):
  - *complex* de Python (la parte real e imaginaria son *float*)
    c = 2.1+7.8j

Tipos básicos

- Operadores aritméticos:

Operador	Descripción	Ejemplo
+	Suma	`r = 3 + 2  # r es 5`
-	Resta	`r = 4 - 7  # r es -3`
-	Negación	`r = -7  # r es -7`
*	Multiplicación	`r = 2 * 6  # r es 12`
**	Exponente	`r = 2 ** 6  # r es 64`
/	División	`r = 3.5 / 2  # r es 1.75`
//	División entera	`r = 3.5 // 2 # r es 1.0`
%	Módulo	`r = 7 % 2  # r es 1`

Tipos básicos

- Operadores a nivel bit:

Operador	Descripción	Ejemplo
&	and	`r = 3 & 2  # r es 2`
|	or	`r = 3 | 2  # r es 3`
^	xor	`r = 3 ^ 2  # r es 1`
~	not	`r = ~3  # r es -4`
<<	Desplazamiento izq.	`r = 3 << 1 # r es 6`
>>	Desplazamiento der.	`r = 3 >> 1 # r es 1`

Tipos básicos - cadenas

- Cadenas (inmutables):
  - *str* de Python (se marcan con "..." o con '...')
  - Pueden ser *Unicode* o *raw*:
    c = u'äóè'
    c = r'\n'
  - Pueden extenderse en varias líneas:
    """primera línea
    esto se vera en otra línea"""

- Operadores:
  - Suma:
    'uno' + 'dos' → 'unodos'
  - Producto:
    'uno' * 3 → 'unounouno'

- Funciones básicas:
  - type:
    type('uno') → <type 'str'>
  - len:
    len('uno') → 3

Tipos básicos - cadenas

- count: c.count(sub,start,end)

  devuelve el número de veces que aparece *sub* en *c[start,end]*

  c = 'hola mundo'
  c.count('o',0,5) → 1

- find: l.find(sub,start,end)

  devuelve la primera posición de *sub* en *c[start,end]*

  c = 'hola mundo'
  c.find('o',0,5) → 1

Tipos básicos - cadenas

- join: c.join(seq)

  devuelve una cadena resultante de concatenar las cadenas de seq separadas por la cadena que llama al método

  c = '#'
  seq = ['hola', 'gran', 'mundo']
  c.join(seq) → 'hola#gran#mundo'

Tipos básicos - cadenas

- partition: c.partition(sep)

  busca *sep* en *c* y devuelve la subcadena anterior, *sep* y la subcadena posterior.

  Si no se encuentra *sep* devuelve *c* y dos cadenas vacías

  c = 'hola mundo'
  sep = ' '
  c.partition(sep) → ('hola', ' ', 'mundo')

Tipos básicos - cadenas

- replace: c.replace(old,new,count)

  devuelve una cadena en la que se han reemplazado las apariciones de *old* por *new*. Si se especifica *count*, este indica el número máximo de reemplazamientos.

  c = 'hola hola hola'
  old = 'h'
  new = 'carac'
  c.replace(old,new,2) → 'caracola caracola hola'

Tipos básicos - cadenas

- split: c.split(sep,maxsplit)

  devuelve una lista con las subcadenas en las que se divide *c* al suprimir el delimitador *sep*.

  Si no se especifica *sep* se usan espacios y si se especifica *maxsplit*, este indica el número máximo de particiones a realizar

  c = 'hola#gran#mundo'
  sep = '#'
  c.split(sep,1) → ['hola', 'gran#mundo']

Tipos básicos

- Booleanos: *True* y *False*
- Operadores:

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	`r = True and False # r es False`
or	¿se cumple a o b?	`r = True or False # r es True`
not	No a	`r = not True # r es False`

Tipos básicos

- Operadores relacionales:

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`r = 5 == 3 # r es False`
!=	¿son distintos a y b?	`r = 5 != 3 # r es True`
<	¿es a menor que b?	`r = 5 < 3  # r es False`
>	¿es a mayor que b?	`r = 5 > 3  # r es True`
<=	¿es a menor o igual que b?	`r = 5 <= 5 # r es True`
>=	¿es a mayor o igual que b?	`r = 5 >= 3 # r es True`

Colecciones - listas

- *list* de Python (mutables): son equivalentes a *arrays*
- Pueden contener cualquier tipo de objeto:
  l = [22, True, 'una lista', [], [1, 2]]
- Extraer y modificar elementos:
  elemento = l[0] → elemento = 22
  elemento = l[-2] → elemento = []
  l[1] = False → l = [22, False, 'una lista', [], [1, 2]]
- Operadores:
  - Suma:
    [0,1] + [2,3] → [0,1,2,3]
  - Producto:
    [0,1] * 3 → [0,1,0,1,0,1]
- Funciones básicas:
  - type:
    type([0,1]) → <type 'list'>
  - len:
    len([0,1]) → 2

Colecciones - listas

- Extraer partes:

  l = [22, True, 'una lista', [], [1, 2]]

  l[1:4] → [True, 'una lista', []]

  l[0:4:2] → [22, 'una lista']

  l[:2] → [22, True]

  l[2:] → ['una lista', [], [1, 2]]

  l[::2] → [22, 'una lista', [1, 2]]

- Modificar partes:

  l = [22, True, 'una lista', [], [1, 2]]

  l[:2] = [False] → l = [False, 'una lista', [], [1, 2]]

Colecciones - listas

- append: l.append(object)

  añade *object* al final de la lista

  l = [0,1,2,3]
  x = 4
  l.append(x) → l = [0,1,2,3,4]
  x = -1 → l = [0,1,2,3,4]
  y = [5]
  l.append(y) → l = [0,1,2,3,4,[5]]
  y[0] = [-1] → l = [0,1,2,3,4,[-1]]
  y = [5] → l = [0,1,2,3,4,[-1]]

Colecciones - listas

- count: l.count(value)

  cuenta el número de veces que aparece *value* en la lista

  l = [0,1,0,2]
  l.count(0) → 2

- extend: l.extend(iterable)

  añade los elemento de *iterable* a la lista

  l = [0,1,0,2]
  x = [-1, -2]
  l.extend(x) → l = [0,1,0,2,-1,-2]

Colecciones - listas

- index: l.index(value,start,stop)

  encuentra la primera aparición de *value* en *l[start,stop]*

  l = [2,1,2,3]
  l.index(2,1,3) → 2 (da error si no lo encuentra)

- insert: l.insert(index,object)

  inserta *object* en la posición *index*

  l = [0,1,0,2]
  l.insert(0,'a') → l = ['a',0,1,0,2] (si index está fuera de rango inserta en el extremo)

Colecciones - listas

- pop: l.pop(index)

  devuelve el valor en la posición *index* y lo borra de la lista

  l = [0,1,2,3]
  l.pop(2) → l = [0,1,3]

- remove: l.remove(value)

  elimina la primera aparición de *value* en la lista

  l = [0,1,0,2]
  l.remove(0) → l = [1,0,2]

Colecciones - listas

- reverse: l.reverse()

  invierte la lista

  l = [0,1,2,3]
  l.reverse() → l = [3,2,1,0]

- sort: l.sort(cmp=None,key=None,reverse=False)

  ordena la lista

  l = [3,1,0,2]
  l.sort(cmp=orden) → l = [1,0,2,3]

```
def orden(param1,param2):
    if param1 < param2:
        return(-1)
    elif param1 == param2:
        return(0)
    else:
        return(1)
```

Colecciones - tuplas

- *tuple* de Python (inmutables): estructura similar a *list*

  Se definen usando ( ) en lugar de [ ]. Realmente los paréntesis no son necesarios, aunque se recomienda su uso tal como hace Python

  t = (1, 2, True, 'Python')
  t = (1,)
  t = (1) → t es de tipo *int*

- Operadores:
  - Suma:
    (0,1) + (2,3) → (0,1,2,3)
  - Producto:
    (0,1) * 3 → (0,1,0,1,0,1)

- Funciones básicas:
  - type:
    type((0,1)) → <type 'tuple'>
  - len:
    len((0,1)) → 2

Colecciones - tuplas

- Pueden contener cualquier tipo de objeto:
  t = (22, True, 'una lista', (), [1, 2])

- Extraer elementos:
  elemento = t[0] → elemento = 22
  elemento = t[-2] → elemento = ()
  t[1] = False → error

- Extraer partes:
  t[1:4] → (True, 'una lista', ())
  t[0:4:2] → (22, 'una lista')
  t[:2] → (22, True)
  t[2:] → ('una lista', (), [1, 2])
  t[::2] → (22, 'una lista', [1, 2])

Colecciones - diccionarios

- *dict* de Python (mutables):

  Se definen usando { } e incluyen un conjunto de pares *key: value*. La mutabilidad afecta al campo *value* mientras el campo *key* es inmutable.

  d = {'miercoles': 'Programacion', 'jueves': 'Bases de Datos'}
  print d →
  {'jueves': 'Bases de Datos', 'miercoles': 'Programacion'}

- Están implementados mediante tablas Hash

- Funciones básicas:
  - type:
    type(d) → <type 'dict'>
  - len:
    len(d) → 2

Colecciones - diccionarios

- Extraer y modificar elementos:
  d = {'miercoles': 'Programacion', 'jueves': 'Bases de Datos'}
  elemento = d['miercoles'] → elemento = 'Programacion'
  d['jueves'] = 'Big Data' →
  {'jueves': 'Big Data', 'miercoles': 'Programacion'}

- *del*: del d[key]
  borra la entrada conclave *key*
  d = {'miercoles': 'Programacion', 'jueves': 'Big Data'}
  del d['miercoles'] → {'jueves': 'Big Data'}

- *del*: del d
  borra el diccionario d completamente

Colecciones - diccionarios

- *clear*: d.clear( )
  borra todas las entradas de d
  d = {'miercoles': 'Programacion', 'jueves': 'Big Data'}
  d.clear( ) → { }

- *copy*: d.copy( )
  hace una copia de d

- *get*: d.get(key,default=None)
  obtiene el *value* asociado a *key* y el segundo argumento si no exite la clave *key* en el diccionario

- *has_key*: d.has_key(key)
  devuelve *True* si existe la clave *key* y *False* en caso contrario

Colecciones - diccionarios

- *items*: d.items( )
  devuelve una lista con todos los pares *(key, value)*
  d = {'miercoles': 'Programacion', 'jueves': 'Big Data'}
  d.items( ) →
  [('jueves', 'Big Data'), ('miercoles', 'Programacion')]

- *keys*: d.keys( )
  devuelve una lista con todos los *keys*
  d = {'miercoles': 'Programacion', 'jueves': 'Big Data'}
  d.keys( ) → ['jueves', 'miercoles']

- *values*: d.values( )
  devuelve una lista con todos los *values*
  d = {'miercoles': 'Programacion', 'jueves': 'Big Data'}
  d.values( ) → ['Big Data', 'Programacion']

Colecciones - diccionarios

- *setdefault*: d.setdefault(key,default=None)
  devuelve el *value* de la clave *key* y, si no existe, añade la pareja *key: default* devolviendo *default*
  d = {'jueves': 'Big Data'}
  d.setdefault('lunes',-1) → -1
  d = {'jueves': 'Big Data', 'lunes': -1]

- *update*: d1.update(d2)
  incorpora en d1 los pares *key: value* de d2. Si una clave de d2 ya existe en d1 actualiza el campo *value*
  d1 = {1: 'Programacion', 2: 'Estructura de datos'}
  d2 = {3: 'Programacion', 2: 'Big Data'}
  d1.update(d2) →
  {1: 'Programacion', 2: 'Big Data', 3: 'Programacion'}