Funciones sobre iteradores

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Diez

Contenido

Introducción

- Módulo *itertools* .
- Funciones para iteradores.
- Funciones inspiradas por las de lenguajes de programación funcional como Clojure o Haskell.
- Eficientes en tiempo y memoria.
- Se pueden combinar para expresar algoritmos basados en iteraciones más complicados.
- Más eficiente que usar listas, no se producen datos hasta que se necesitan.

[Hellman, 2011, pág 141-142]

Unión de iteradores (I)

- Dados varios iteradores obtener uno con la concatenación: *chain*()

```
1 >>>from itertools import chain
2 >>>
3 >>>for i in chain([1, 2, 3], ['a', 'b', 'c']):
4         print(i, end="␣")
5
6 1 2 3 a b c
```

[Hellman, 2011, pág 142]

Unión de iteradores (II)

```
1 # Equivalente a  itertools .chain:
2
3 def chain(* iterables ):
4     # chain('ABC', 'DEF') --> A B C D E F
5     for  it  in  iterables :
6         for  element in  it :
7             yield  element
```

https://docs.python.org/3.3/library/itertools.html#itertools.chain

Unión de iteradores (III)

- Función predefinida **zip**() (no está en *itertootls*).
- Iterador que agrega elementos de varios iterables.
- Devuelve un iterador de tuplas.
- La tupla *i*-ésima contiene el elemento *i*-ésimo de cada uno de los argumentos.
- El iterador se para cuando el argumento (iterable) más corto se para.
- Con un solo argumento devuelve un iterador con tuplas de 1 elemento.

`https://docs.python.org/3/library/functions.html#zip`

Unión de iteradores (IV)

```
>>>list(zip(range(5)))
[(0,), (1,), (2,), (3,), (4,)]

>>>x = [1, 2, 3]
>>>y = [4, 5, 6]
>>>zipped = zip(x, y)
>>>list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>>x2, y2 = zip(*zip(x, y))
>>>x2, y2
((1, 2, 3), (4, 5, 6))
>>>(x, y) == (list(x2), list (y2))
True
```

```
>>>args = [1, 30, 6]
>>>print(args)
[1, 30, 6]
>>>print(*args)
1 30 6
>>>list(range(*args))
[1, 7, 13, 19, 25]
```

https://docs.python.org/3/library/functions.html#zip

Selección de iteradores

- Iterador que dado otro devuelve elementos seleccionados por índice.

[Hellman, 2011, pág 143]

```
1  print('Stop at 5:')
2  for i in islice(count(), 5):
3      print(i, end=" ")
4  print('\n')
5
6  print('Start at 5, Stop at 10:')
7  for i in islice(count(), 5, 10):
8      print(i, end=" ")
9  print('\n')
10
11 print('By tens to 100:')
12 for i in islice(count(), 0, 100, 10):
13     print(i, end=" ")
14 print('\n')
```

Duplicado de iteradores (I)

- Obtener varios iteradores independientes a partir de una entrada.
  - Por defecto 2

```
1 >>>r = islice(count(), 5)
2 >>>i1, i2 = tee(r)
3 >>>
4 >>>print('i1:', list(i1))
5 i1: [0, 1, 2, 3, 4]
6 >>>print('i2:', list(i2))
7 i2: [0, 1, 2, 3, 4]
```

[Hellman, 2011, pág 144]

Duplicado de iteradores (II)

- Posible utilidad: mismos datos a distintos algoritmos que se procesan en paralelo.
- Los iteradores obtenidos comparten la entrada, no debería usarse el iterador original una vez que se crean los nuevos.

```
r: 0 1 2
i1: [3, 4]
i2: [3, 4]
```

```
r = islice(count(), 5)
i1, i2 = tee(r)

print('r:', end=" ")
for i in r:
    print(i, end=" ")
    if i > 1:
        break
print()

print('i1:', list(i1))
print('i2:', list(i2))
```

[Hellman, 2011, pág 144-145]

Duplicado de iteradores (III)

```
1  # Implementación equivalente a itertools.tee
2  def tee(iterable, n=2):
3      it = iter(iterable)
4      deques = [collections.deque() for i in range(n)]
5      def gen(mydeque):
6          while True:
7              if not mydeque:    # when the local deque is empty
8                  newval = next(it)      # fetch a new value and
9                  for d in deques:    # load it to all the deques
10                     d.append(newval)
11             yield mydeque.popleft()
12     return tuple(gen(d) for d in deques)
```

https://docs.python.org/3.3/library/itertools.html#itertools.tee

Conversión de iteradores (I)

- Función predefinida **map**(): llama a una función con los valores de los iteradores de entradas.
- Se para con el primer iterador de entrada que se pare.

```
1 print('Doubles:')
2 for i in map(lambda x:2*x, range(5)):
3     print(i)
4
5 print('Multiples:')
6 for i in map(lambda x,y:(x, y, x*y), range(5), range(5,10)):
7     print('%d * %d = %d' % i)
```

[Hellman, 2011, pág 145]

Conversión de iteradores (II)

- *starmap*(): reparte los elementos de un solo iterador, los usa como argumentos de la función con el operador $*$.
- $f(*i)$ en vez de $f(i1, i2)$.

```
1 values = [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
2 for i in starmap(lambda x,y:(x, y, x*y), values):
3     print('%d * %d = %d' % i)
```

[Hellman, 2011, pág 146]

Conversión de iteradores (III)

```
1 # Equivalente a  itertools .starmap
2
3 def starmap(function,  iterable ):
4     # starmap(pow, [(2,5),  (3,2),  (10,3)])  -->  32 9 1000
5     for  args  in  iterable :
6         yield  function(*args)
```

https://docs.python.org/3.3/library/itertools.html#itertools.starmap

Generación de valores (I)

- Función *count*(): produce valores numéricos desde un valor inicial (por defecto 0) con un determinado incremento (por defecto 1).
- Sin límite superior.
- Se suele usar con **map**() o **zip**().

```
1 for i in zip(count(1), ['a', 'b', 'c']):
2     print(i)
```

```
1 (1, 'a')
2 (2, 'b')
3 (3, 'c')
```

[Hellman, 2011, pág 146]

Generación de valores (II)

- Función *cycle*(): iterador que repite indefinidamente los contenidos de los argumentos que recibe.
- Necesita almacenar los contenidos del iterador de entrada.

```
1 for i, item in zip(range(7), cycle(['a', 'b', 'c'])):
2     print(i, item)
```

[Hellman, 2011, pág 147]

Generación de valores (III)

```
1  # implementación equivalente a  itertools . cycle
2
3  def cycle( iterable ):
4      # cycle('ABCD') --> A B C D A B C D A B C D ...
5      saved = []
6      for element in iterable :
7          yield element
8          saved.append(element)
9      while saved:
10         for element in saved:
11             yield element
```

https://docs.python.org/3.3/library/itertools.html#itertools.cycle

Generación de valores (IV)

- Función *repeat*(): genera el mismo un número de veces o indefinidamente si no hay argumento adicional.

```
1 for i in repeat('over-and-over', 5):
2     print(i)
```

[Hellman, 2011, pág 147]

```
1 >>>list(map(pow, range(10), repeat(2)))
2 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

https://docs.python.org/3.3/library/itertools.html#itertools.repeat

Filtrado (I)

- Función *dropwhile*() devuelve un iterador a partir de otro, genera sus valores después que una condición sea falsa.

```
1 def should_drop(x):
2     print('Testing:', x)
3     return (x<1)
4
5 for i in dropwhile(should_drop,
6                    [ -1, 0, 1, 2, -2 ]):
7     print('Yielding:', i)
```

```
1 Testing: -1
2 Testing: 0
3 Testing: 1
4 Yielding: 1
5 Yielding: 2
6 Yielding: -2
```

[Hellman, 2011, pág 148-149]

Filtrado (II)

- Función *takewhile*(), opuesta a *dropwhile*().

```
1 def should_take(x):
2     print('Testing:', x)
3     return (x<2)
4
5 for i in takewhile(should_take,
6                    [ -1, 0, 1, 2, -2 ]):
7     print('Yielding:', i)
```

```
1 Testing: -1
2 Yielding: -1
3 Testing: 0
4 Yielding: 0
5 Testing: 1
6 Yielding: 1
7 Testing: 2
```

[Hellman, 2011, pág 149-150]

Filtrado (III)

- Función predefinida **filter** ().

```
1 def check_item(x):
2     print('Testing:', x)
3     return (x<1)
4
5 for i in filter(check_item,
6                 [ -1, 0, 1, 2, -2 ]):
7     print('Yielding:', i)
```

```
1 Testing: -1
2 Yielding: -1
3 Testing: 0
4 Yielding: 0
5 Testing: 1
6 Testing: 2
7 Testing: -2
8 Yielding: -2
```

[Hellman, 2011, pág 150]

Filtrado (IV)

- Función *filterfalse* ().

```
1 def check_item(x):
2     print('Testing:', x)
3     return (x<1)
4
5 for i in filterfalse(check_item,
6                       [-1, 0, 1, 2, -2]):
7     print('Yielding:', i)
```

```
1 Testing: -1
2 Testing: 0
3 Testing: 1
4 Yielding: 1
5 Testing: 2
6 Yielding: 2
7 Testing: -2
```

[Hellman, 2011, pág 150-151]

Agrupaciones (I)

- Función *groupby*(): iterador que genera conjuntos de valores organizados por una clave común.
- Además del iterable tiene un argumento opcional *key*.
- Nuevo grupo cada vez que cambia la clave.

```
>>>[k for k, g in groupby('AAAABBBCCDAABBB')]
['A', 'B', 'C', 'D', 'A', 'B']
>>>[list(g) for k, g in groupby('AAAABBBCCD')]
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D']]
```

[Hellman, 2011, pág 151]

https://docs.python.org/3.3/library/itertools.html#itertools.groupby

Agrupaciones (II)

- Devuelve un iterador, comparte el iterable subyacente.
- Cuando se avanza, el grupo previo deja de ser visible.
- Podría almacenarse en una lista.

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))   # Store group iterator as a list
    uniquekeys.append(k)
```

https://docs.python.org/3.3/library/itertools.html#itertools.groupby

Agrupaciones (III)

```
1 from operator import itemgetter
2
3 d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
4 di = sorted(d.items(), key=itemgetter(1))
5 for k, g in groupby(di, key=itemgetter(1)):
6     g1, g2 = tee(g)
7     print(k, list (g1), list (map(itemgetter(0), g2)))
```

```
1 1 [('a', 1), ('c', 1), ('e', 1)] ['a', 'c', 'e']
2 2 [('b', 2), ('d', 2), ('f', 2)] ['b', 'd', 'f']
3 3 [('g', 3)] ['g']
```

[Hellman, 2011, pág 151-152]

product (I)

- *itertools* .*product*(∗ *iterables* , *repeat*=1)
- Producto cartesiano de iterables.
- Equivalente a bucles anidados en una expresión generadora:
  - *product*(*A*, *B*)
  - ((*x*, *y*) **for** *x* **in** *A* **for** *y* **in** *B*)
- En cada iteración avanza el último iterable.
- Si los iterables están ordenados, las tuplas generadas están ordenadas.
- *product*(*A*, *repeat*=3) es equivalente a *product*(*A*, *A*, *A*)

`https://docs.python.org/3.3/library/itertools.html#itertools.product`

product (II)

```
>>>list(product('ABCD', 'xy'))
[('A', 'x'), ('A', 'y'), ('B', 'x'), ('B', 'y'), ('C', 'x'),
 ('C', 'y'), ('D', 'x'), ('D', 'y')]

>>>list(product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

https://docs.python.org/3.3/library/itertools.html#itertools.product

product (III)

```
1  # Equivalente a itertools.product, con la excepción de que en
2  # está versión se almacenan los valores a generar
3
4  def product(*args, repeat=1):
5      pools = [tuple(pool) for pool in args] * repeat
6      result = [[]]
7      for pool in pools:
8          result = [x+[y] for x in result for y in pool]
9      for prod in result:
10         yield tuple(prod)
```

https://docs.python.org/3.3/library/itertools.html#itertools.product

permutations (I)

- *itertools* . *permutations*(*iterable* , *r*=*None*)
- Permutaciones de longitud *r*.
- Por defecto la longitud es la del iterable.
- Se generan en orden lexicográfico.
- Si el iterador de entrada está ordenado, se generan tuplas ordenadas.
- Los elementos vienen dados por las posiciones, no por los valores.

`https://docs.python.org/3.3/library/itertools.html#itertools.permutations`

permutations (II)

```
>>> list(permutations("abc"))
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
 ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')]

>>> list(permutations("abcd", 2))
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'a'), ('b', 'c'),
 ('b', 'd'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('d', 'a'),
 ('d', 'b'), ('d', 'c')]

>>> list(permutations("aba"))
[('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
 ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')]
```

permutations (III)

```
1  # Implementación alternativa (e ineficiente) de permutations.
2  # Dados los elementos generados con product, filtra los que
3  # tienen repetidos
4
5  def permutations(iterable, r=None):
6      pool = tuple(iterable)
7      n = len(pool)
8      r = n if r is None else r
9      for indices in product(range(n), repeat=r):
10         if len(set(indices)) == r:
11             yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.permutations

permutations (IV)

```
1  # Equivalente a  itertools .permutations
2
3  def permutations( iterable ,  r=None):
4      pool = tuple( iterable )
5      n = len(pool)
6      r = n if r is None else r
7      if r > n:
8          return
9      indices = list (range(n))
10     cycles = list (range(n, n-r, -1))
11     yield tuple(pool[ i ] for i in indices [: r ])
```

https://docs.python.org/3.3/library/itertools.html#itertools.permutations

permutations (V)

```
1       while n:
2           for i in reversed(range(r)):
3               cycles[i] -= 1
4               if cycles[i] == 0:
5                   indices[i:] = indices[i+1:] + indices[i:i+1]
6                   cycles[i] = n - i
7               else:
8                   j = cycles[i]
9                   indices[i], indices[-j] = indices[-j], indices[i]
10                  yield tuple(pool[i] for i in indices[:r])
11                  break
12          else:
13              return
```

https://docs.python.org/3.3/library/itertools.html#itertools.permutations

combinations (I)

- *itertools* . *combinations*(*iterable* , *r*).
- Subsecuencias de longitud *r* de elementos del *iterable* .
- En orden lexicográfico de posiciones.
- Si el *iterable* de entrada está ordenado, las tuplas generadas estarán ordenadas.
- Por cada posición considera que hay un elemento único, aunque haya valores repetidos.

`https://docs.python.org/3.3/library/itertools.html#itertools.combinations`

combinations (II)

```
>>>list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'),
 ('C', 'D')]

>>>list(combinations(range(4), 3))
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]

>>>list(combinations('ABCA', 2))
[('A', 'B'), ('A', 'C'), ('A', 'A'), ('B', 'C'), ('B', 'A'),
 ('C', 'A')]
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations

combinations (III)

```
1  # Implementación alternativa (e ineficiente) de
2  # itertools.combinations.
3  # Filtra de permutations los que no tienen los índices ordenados
4
5  def combinations(iterable, r):
6      pool = tuple(iterable)
7      n = len(pool)
8      for indices in permutations(range(n), r):
9          if sorted(indices) == list(indices):
10             yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations

combinations (IV)

```
1 # Implementación alternativa de itertools.combinations.
2
3 def combinations( iterable , r ):
4     pool = tuple( iterable )
5     n = len(pool)
6     if r > n:
7         return
8     indices = list(range(r))
9     yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations

combinations (V)

```
10      while True:
11          for i in reversed(range(r)):
12              if indices[i] != i + n - r:
13                  break
14          else:
15              return
16          indices[i] += 1
17          for j in range(i+1, r):
18              indices[j] = indices[j-1] + 1
19          yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations

combinations_with_replacement (I)

- *itertools* . *combinations_with_replacement*(*iterable* , *r*)
- Secuencias de longitud *r* de elementos del *iterable* , con repeticiones.
- Orden lexicográfico.
- Si los elementos del *iterable* están ordenados, se generan tuplas ordenadas.
- Por cada posición considera que hay un elemento único, aunque haya valores repetidos.

`https://docs.python.org/3.3/library/itertools.html#itertools.combinations_with_replacement`

combinations_with_replacement (II)

```
1  >>>list(combinations_with_replacement('ABC', 2))
2  [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'),
3   ('C', 'C')]
4
5  >>>list(combinations_with_replacement('AB', 3))
6  [('A', 'A', 'A'), ('A', 'A', 'B'), ('A', 'B', 'B'),
7   ('B', 'B', 'B')]
8
9  >>>list(combinations_with_replacement('AAB', 2))
10 [('A', 'A'), ('A', 'A'), ('A', 'B'), ('A', 'A'), ('A', 'B'),
11  ('B', 'B')]
```

combinations_with_replacement (III)

```python
# Implementación alternativa (e ineficiente) de
# itertools.combinations_with_replacement.
# Filtra de product los que no tienen los índices ordenados

def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations_with_replacement

combinations_with_replacement (IV)

```
1  # Implementación alternativa de
2  # itertools.combinations_with_replacement.
3
4  def combinations_with_replacement( iterable , r ):
5      pool = tuple( iterable )
6      n = len(pool)
7      if not n and r:
8          return
9      indices = [0] * r
10     yield tuple(pool[ i ] for i in indices )
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations_with_replacement

combinations_with_replacement (V)

```
11      while True:
12          for i in reversed(range(r)):
13              if indices[i] != n - 1:
14                  break
15          else:
16              return
17          indices[i:] = [indices[i] + 1] * (r - i)
18          yield tuple(pool[i] for i in indices)
```

https://docs.python.org/3.3/library/itertools.html#itertools.combinations_with_replacement

Referencias (I)

[Hellman, 2011] El capítulo 3 está dedicado a algoritmos. La sección 3.2 está dedicada a *itertools* .


Hellman, D. (2011).
The Python Standard Library by Example.
Addison-Wesley.
http://doughellmann.com/pages/python-standard-library-by-example.html.