Colecciones en Python

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Diez

Contenido

Introducción (I)

- Algunas estructuras de datos como tipos incorporados: **list** , **tuple**, **dict**, **set**.
- Otras estructuras en la biblioteca estándar.
- Módulo *collections* :
  - *deque*: bicolas.
  - *defaultdict* : diccionarios con valor por defecto para claves que no están.
  - *OrderedDict*: diccionario que recuerda el orden en el que se insertaron los elementos.
  - *namedtuple*: extensión de tupla dónde los miembros tienen nombres además de posiciones numéricas.

[Hellman, 2011, pág 69]

Introducción (II)

- *array*: uso más eficiente de la memoria que **list**. Elementos de un solo tipo.
- *heapq*: montículos.
- *bisect*: listas ordenadas.
- *Queue*: uso seguro con *threads*.
- *struct*: decodificar datos de otra aplicación, por ejemplo de un fichero binario.
- *weakref*: referencias que pueden ser reclamadas por el recolector de basura una vez que no se necesitan.
- *pprint*: representaciones legibles.

[Hellman, 2011, pág 69–70]

Counter (I)

- *Counter*: cuantas veces se han añadido valores equivalentes.
- "bag", "multiset".
- Inicialización:
  - Sin argumentos.
  - Secuencia de elementos.
  - Diccionario con claves y apariciones.
  - Argumentos con nombre, se asocian las cadenas de los nombres sus apariciones.

[Hellman, 2011, pág 70]

Counter (II)

```
>>>import collections
>>>print(collections.Counter(['a', 'b', 'c', 'a', 'b', 'b']))
Counter({'b': 3, 'a': 2, 'c': 1})
>>>print(collections.Counter({'a':2, 'b':3, 'c':1}))
Counter({'b': 3, 'a': 2, 'c': 1})
>>>print(collections.Counter(a=2, b=3, c=1))
Counter({'b': 3, 'a': 2, 'c': 1})
```

[Hellman, 2011, pág 70–71]

Counter (III)

- Método *update* para añadir.

```
1 >>>c = collections.Counter(); c
2 Counter()
3 >>>c.update('abcdaab'); c
4 Counter({'a': 3, 'b': 2, 'c': 1, 'd': 1})
5 >>>c.update({'a':1, 'd':5}); c
6 Counter({'d': 6, 'a': 4, 'b': 2, 'c': 1})
7 >>>c.update(c=3, b=1); c
8 Counter({'d': 6, 'a': 4, 'c': 4, 'b': 3})
```

[Hellman, 2011, pág 71]

Counter (IV)

- Acceso a los elementos como en diccionarios.
- Si un elemento no está, no hay *KeyError*. Aparece 0 veces.

```
>>>c = collections.Counter('abcdaab')
>>>for letra in c:
        print('%s : %d' % (letra, c[letra]))
b : 2
a : 3
c : 1
d : 1
>>>c['e']
0
```

[Hellman, 2011, pág 72]

Counter (V)

- Método *elements*() devuelve un iterador que produce los elementos del contador.

```
>>>c = collections.Counter('extremadamente')
>>>c['z'] = 0
>>>print(c)
Counter({'e': 4, 'a': 2, 'm': 2, 't': 2, 'n': 1, 'x': 1,
         'r': 1, 'd': 1, 'z': 0})
>>>print(list(c.elements()))
['a', 'a', 'm', 'm', 'n', 'x', 'e', 'e', 'e', 'e', 'r', 'd',
 't', 't']
```

[Hellman, 2011, pág 72]

Counter (VI)

- Método *most_common*(*n*) produce una secuencia con los *n* elementos y sus apariciones más frecuentes.

```
>>>c = collections.Counter('extremadamente' * 100); c
Counter({'e': 400, 'a': 200, 'm': 200, 't': 200, 'n': 100,
         'x': 100, 'r': 100, 'd': 100})
>>>for letra, apariciones in c.most_common(3):
        print('%s : %d' % (letra, apariciones))

e : 400
a : 200
m : 200
```

[Hellman, 2011, pág 72–73]

Counter (VII)

- Operadores aritméticos.

```
>>> c1 = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
>>> c2 = collections.Counter('alfabeto')
>>> c1 + c2
Counter({'a': 4, 'b': 4, 'f': 1, 'l': 1, 'e': 1, 'o': 1, 'c': 1, 't': 1})
>>> c1 - c2
Counter({'b': 2, 'c': 1})
>>> c1 & c2 # intersección (mínimos positivos)
Counter({'a': 2, 'b': 1})
>>> c1 | c2 # unión (máximos)
Counter({'b': 3, 'a': 2, 'f': 1, 'l': 1, 'e': 1, 'o': 1, 'c': 1, 't': 1})
```

[Hellman, 2011, pág 73]

defaultdict (I)

- Diccionarios con valor por defecto.

```
>>>def default_factory():
        return 'default value'
>>>d = collections.defaultdict( default_factory , foo='bar')
>>>print(d)
defaultdict(<function default_factory at 0x7f7fa4c36488>,
            {'foo': 'bar'})
>>>print(d['foo'])
bar
>>>print(d['bar'])
default value
```

[Hellman, 2011, pág 74–75]

defaultdict (II)

```
1  >>>d = collections. defaultdict ( list )
2  >>>d
3   defaultdict (<class ' list '>, {})
4  >>>s = [('yellow', 1), ('blue', 2), ('yellow', 3),
5          ('blue', 4), ('red', 1)]
6  >>>for k, v in s:
7          d[k]. append(v)
8
9  >>>d
10  defaultdict (<class ' list '>, {'blue': [2, 4], 'yellow': [1, 3],
11                                  'red': [1]})
```

https://docs.python.org/3.3/library/collections.html?highlight=counter#defaultdict-examples

defaultdict (III)

```
>>>s = 'mississippi'
>>>d = collections.defaultdict(int)
>>>for k in s:
        d[k] += 1

>>>d
defaultdict(<class 'int'>, {'p': 2, 'm': 1, 's': 4, 'i': 4})
```

https://docs.python.org/3.3/library/collections.html?highlight=counter#defaultdict-examples

defaultdict (IV)

```
>>>def constant_factory(value):
        return lambda: value

>>>d = collections. defaultdict ( constant_factory ('<missing>'))
>>>d.update(name='John', action='ran')
>>>'%(name)s␣%(action)s␣to␣%(object)s' % d
'John ran to <missing>'
```

https://docs.python.org/3.3/library/collections.html?highlight=counter#defaultdict-examples

defaultdict (V)

```
1 s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4),
2       ('red', 1), ('blue', 4)]
3 d = collections.defaultdict(set)
4 >>>for k, v in s:
5         d[k].add(v)
6
7 >>>d
8 defaultdict(<class 'set'>, {'blue': {2, 4}, 'red': {1, 3}})
```

https://docs.python.org/3.3/library/collections.html?highlight=counter#defaultdict-examples

Deque (I)

- Bicolas (colas de doble extremo). Un tipo de secuencia, soporta las operaciones de **list** .

```
>>>d = collections.deque('abcdefg')
>>>print('Deque:', d, '. Length:', len(d))
Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g']) . Length: 7
>>>print('Left end:', d[0], '. Right end:', d[-1])
Left end: a . Right end: g
>>>d.remove('c')
>>>print('remove(c):', d)
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

[Hellman, 2011, pág 75–76]

Deque (II)

- Insertar por ambos extremos.

```
1 >>>d1 = collections.deque()
2 >>>d1.extend('abcdefg')
3 >>>print('extend     :', d1)
4 extend     : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
5 >>>d1.append('h')
6 >>>print('append     :', d1)
7 append     : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

[Hellman, 2011, pág 76]

Deque (III)

```
1 >>>d2 = collections.deque()
2 >>>d2.extendleft(range(6))
3 >>>print('extendleft:', d2)
4 extendleft: deque([5, 4, 3, 2, 1, 0])
5 >>>d2.appendleft(6)
6 >>>print('appendleft:', d2)
7 appendleft: deque([6, 5, 4, 3, 2, 1, 0])
```

[Hellman, 2011, pág 76]

Deque (IV)

- Eliminar por ambos extremos.

```
1 print('From␣the␣right:')
2 d = collections.deque('abcdefg')
3 while True:
4     try:
5         print(d.pop(), end='␣')
6     except IndexError:
7         break
8 print()
```

```
1 From the right:
2 g f e d c b a
```

[Hellman, 2011, pág 77]

Deque (V)

```
1 print('\nFrom␣the␣left:')
2 d = collections.deque(range(6))
3 while True:
4     try:
5         print(d.popleft(), end='␣')
6     except IndexError:
7         break
8 print()
```

```
1 From the left :
2 0 1 2 3 4 5
```

[Hellman, 2011, pág 77]

Deque (VI)

```
1  import threading
2  import time
3
4  def burn( direction , nextSource):
5      while True:
6          try: next = nextSource()
7          except IndexError: break
8          else:
9              print(' %8s: %s' % (direction, next), flush=True)
10             time.sleep(0.1)
11     print(' %8s done' % direction, flush=True)
12     return
```

[Hellman, 2011, pág 77–78]

Deque (VII)

```
13 candle = collections.deque(range(5))
14
15 left = threading.Thread(target=burn,
16                         args=('Left', candle.popleft))
17 right = threading.Thread(target=burn,
18                          args=('Right', candle.pop))
```

```
19 left.start()
20 right.start()
21
22 left.join()
23 right.join()
```

```
1   Left: 0
2   Right: 4
3   Left: 1
4   Right: 3
5   Left: 2
6   Right done
7   Left done
```

[Hellman, 2011, pág 78]

Deque (VIII)

- Rotar en ambas direcciones.

```
>>>d = collections.deque(range(10))
>>>d.rotate(2)
>>>print('Right rotation:', d)
Right rotation: deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])

>>>d = collections.deque(range(10))
>>>d.rotate(-2)
>>>print('Left rotation :', d)
Left rotation : deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

[Hellman, 2011, pág 78–79]

namedtuple (I)

- En las tuplas normales (**tuple**) se accede a los elementos mediante índices.

```
1 bob = ('Bob', 30, 'male')
2 print('Representation:', bob)
3
4 jane = ('Jane', 29, 'female')
5 print('\nField by index:', jane[0])
6
7 print('\nFields by index:')
8 for p in [ bob, jane ]:
9     print(' %s is a %d year old %s' % p)
```

[Hellman, 2011, pág 79]

namedtuple (II)

- Recordar en qué posición está cada valor.
- *namedtuple* asigna nombres a cada miembro.
- Eficientes en memoria, no hay un diccionario por instancia.
- Cada tipo de *namedtuple* está representada por su propia clase.
- Función "factoría" *namedtuple*().
- Argumentos: nombre de la nueva clase y una cadena con los nombres de los elementos.

[Hellman, 2011, pág 80]

namedtuple (III)

```
1  Person = collections.namedtuple('Person', 'name age gender')
2  print('Type of Person:', type(Person))
3
4  bob = Person(name='Bob', age=30, gender='male')
5  print('\nRepresentation:', bob)
6
7  jane = Person(name='Jane', age=29, gender='female')
8  print('\nField by name:', jane.name)
9
10 print('\nFields by index:')
11 for p in [ bob, jane ]:
12     print(' %s is a %d year old %s' % p)
```

[Hellman, 2011, pág 80-81]

namedtuple (IV)

- Nombres inválidos: repetidos o palabras reservadas.

```
1 try:
2       collections.namedtuple('Person', 'name class age gender')
3 except ValueError as err:
4       print(err)
```

```
1 Type names and field names cannot be a keyword: 'class'
```

[Hellman, 2011, pág 81]

namedtuple (V)

```
1 try:
2       collections.namedtuple('Person', 'name age gender age')
3 except ValueError as err:
4       print(err)
```

```
1 Encountered duplicate field name: 'age'
```

[Hellman, 2011, pág 81]

namedtuple (VI)

- Los nombres podrían estar fuera de control del programa (introducidos por el usuario, consulta a una BD,...)
- Argumento booleano *rename*.

[Hellman, 2011, pág 81-82]

```
1 with_class = collections.namedtuple(
2     'Person', 'name class age gender',
3     rename=True)
4 print( with_class._fields )
5
6 two_ages = collections.namedtuple(
7     'Person', 'name age gender age',
8     rename=True)
9 print(two_ages._fields )
```

```
1 ('name', '_1', 'age', 'gender')
2 ('name', 'age', 'gender', '_3')
```

OrderedDict (I)

- Subclase de diccionario que recuerda en que orden se insertaron los elementos.

```
1 print('Regular dictionary:')
2 d = {}
3 d['a'] = 'A'
4 d['b'] = 'B'
5 d['c'] = 'C'
6
7 for k, v in d.items():
8     print(k, v)
```

```
1 Regular dictionary:
2 b B
3 c C
4 a A
```

[Hellman, 2011, pág 82-83]

OrderedDict (II)

```
1 print('\nOrderedDict:')
2 d = collections.OrderedDict()
3 d['a'] = 'A'
4 d['b'] = 'B'
5 d['c'] = 'C'
6
7 for k, v in d.items():
8     print(k, v)
```

```
1 OrderedDict:
2 a A
3 b B
4 c C
```

[Hellman, 2011, pág 82-83]

OrderedDict (III)

- Igualdad: tienen que tener los mismos elementos en el mismo orden.

```
1 d1 = {}
2 d1['a'] = 'A'; d1['b'] = 'B'; d1['c'] = 'C'
3
4 d2 = {}
5 d2['c'] = 'C'; d2['b'] = 'B'; d2['a'] = 'A'
6
7 print(d1 == d2)
```

```
1 True
```

[Hellman, 2011, pág 83]

OrderedDict (IV)

```
1 d1 = collections.OrderedDict()
2 d1['a'] = 'A'; d1['b'] = 'B'; d1['c'] = 'C'
3
4 d2 = collections.OrderedDict()
5 d2['c'] = 'C'; d2['b'] = 'B'; d2['a'] = 'A'
6
7 print(d1 == d2)
```

```
1 False
```

[Hellman, 2011, pág 83-84]

OrderedDict (V)

- *popitem*(*last*=*True*): devuelve y elimina un par (*clave*, *valor*). El primero o el último.
- *move_to_end*(*key*, *last*=*True*): mueve a uno de los extremos.
- **reversed**() función (no método) para iterar en sentido inverso.

```
1 >>>d = OrderedDict.fromkeys('abcde'); ''.join(d.keys())
2 'abcde'
3 >>>d.move_to_end('b'); ''.join(d.keys())
4 'acdeb'
5 >>>d.move_to_end('b', last=False); ''.join(d.keys())
6 'bacde'
```

https://docs.python.org/3.3/library/collections.html

OrderedDict (VI)

```
>>># regular unsorted dictionary
>>>d = {'banana': 3, 'apple' :4,  'pear' :  1,  'orange' :  2}
>>>
>>># dictionary sorted by key
>>>OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple',  4),  ('banana',  3),  ('orange',  2),  ('pear',  1)])
>>>
>>># dictionary sorted by value
>>>OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear',  1),  ('orange',  2),  ('banana',  3),  ('apple',  4)])
>>>
>>># dictionary sorted by length of the key string
>>>OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear',  1),  ('apple',  4),  ('banana',  3),  ('orange',  2)])
```

https://docs.python.org/3.3/library/collections.html

array (I)

- Módulo *array*.
- Secuencia similar a *lista* , pero todos los elementos tienen que ser del mismo tipo primitivo.
- Inicializado con un argumento que describe el tipo de los datos.
  - Y opcionalmente, una secuencia inicial de datos.
- Operaciones soportadas: "slicing", iteraciones, añadir al final. . .

[Hellman, 2011, pág 84-85]

array (II)

```
1  import array
2
3  a = array.array('i', range(3))
4  print('  Initial :', a)
5
6  a.extend(range(3))
7  print('Extended:', a)
8
9  print(' Slice   :', a[2:5])
10
11 print(' Iterator :')
12 print( list (enumerate(a)))
```

```
1  Initial  : array('i', [0, 1,
2   2])
3 Extended: array('i', [0, 1,
4   2, 0, 1, 2])
5  Slice    : array('i', [2,
6 0, 1])
7  Iterator :
8 [(0, 0), (1, 1), (2, 2),
9 (3, 0), (4, 1), (5, 2)]
```

[Hellman, 2011, pág 85]

array (III)

Código	C	Python	Tamaño mínimo
'b'	signed char	int	1
'B'	unsigned char	int	1
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

https://docs.python.org/3.3/library/array.html

heapq (I)

- Módulo *heapq*.
- Montículo de mínimos.
- Sobre listas de Python.

[Hellman, 2011, pág 87-89]

```
>>>import heapq
>>>h = []
>>>for x in range(9, -1, -1):
        heapq.heappush(h, x)

>>>print(h)
[0, 1, 4, 3, 2, 8, 5, 9, 6, 7]
```

heapq (II)

- Para reorganizar los elementos de una lista existente: *heapify*

[Hellman, 2011, pág 90]

```
1 >>>h = list( range(9, -1, -1))
2 >>>h
3 [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
4 >>>heapq.heapify(h)
5 >>>h
6 [0, 1, 3, 2, 5, 4, 7, 9, 6, 8]
```

heapq (III)

- Para eliminar el elemento con menor valor (el más prioritario, la raíz): *heappop*

[Hellman, 2011, pág 90]

```
1 >>>h
2 [0, 1, 3, 2, 5, 4, 7, 9, 6, 8]
3 >>>heapq.heappop(h)
4 0
5 >>>h
6 [1, 2, 3, 6, 5, 4, 7, 9, 8]
```

heapq (IV)

- Para eliminar el menor y reemplazarlo por otro: *heapreplace*().
  - Más eficiente que eliminar e insertar.

[Hellman, 2011, pág 91]

```
1 >>>h = list( range(9, -1, -1))
2 >>>heapq.heapify(h)
3 >>>h
4 [0, 1, 3, 2, 5, 4, 7, 9, 6, 8]
5 >>>heapq.heapreplace(h, 10)
6 0
7 >>>h
8 [1, 2, 3, 6, 5, 4, 7, 9, 10, 8]
```

heapq (V)

- Funciones par obtener de un iterable los elementos más grandes o más pequeños: *nlargest*(), *nsmallest*().
- Cuando el número a obtener es relativamente pequeño.
  - Si solo un elemento es mejor usar **min** o **max**, si muchos elementos es mejor ordenar.
- Argumento opcional *key*.

[Hellman, 2011, pág 92-93]

- Para mezclar varios iterables ordenados: *merge*.
  - Devuelve un iterador sobre los valores ordenados.

heapq (VI)

```
>>>l = random.sample(range(10),10)
>>>l
[8, 6, 4, 2, 1, 3, 0, 7, 9, 5]
>>>heapq.nlargest(4, l)
[9, 8, 7, 6]
>>>heapq.nsmallest(4, l)
[0, 1, 2, 3]
>>>list(heapq.merge(range(0, 10, 3), range(1, 10, 4),
            range(0, 20, 5)))
[0, 0, 1, 3, 5, 5, 6, 9, 9, 10, 15]
```

bisect (I)

- Modulo *bisect*.
- Insertar elementos en una lista manteniendo el orden.

```
1 l = []
2 for i in range(1, 15):
3     r = random.randint(1, 100)
4     position = bisect.bisect(l, r)
5     bisect.insort(l, r)
6     print('%3d  %3d' % (r, position), l)
```

[Hellman, 2011, pág 93-94]

bisect (II)

```
1   18    0 [18]
2   73    1 [18, 73]
3   98    2 [18, 73, 98]
4    9    0 [9, 18, 73, 98]
5   33    2 [9, 18, 33, 73, 98]
6   16    1 [9, 16, 18, 33, 73, 98]
7   64    4 [9, 16, 18, 33, 64, 73, 98]
8   98    7 [9, 16, 18, 33, 64, 73, 98, 98]
9   58    4 [9, 16, 18, 33, 58, 64, 73, 98, 98]
10  61    5 [9, 16, 18, 33, 58, 61, 64, 73, 98, 98]
11  84    8 [9, 16, 18, 33, 58, 61, 64, 73, 84, 98, 98]
12  49    4 [9, 16, 18, 33, 49, 58, 61, 64, 73, 84, 98, 98]
13  27    3 [9, 16, 18, 27, 33, 49, 58, 61, 64, 73, 84, 98, 98]
14  13    1 [9, 13, 16, 18, 27, 33, 49, 58, 61, 64, 73, 84, 98, 98]
```

bisect (III)

- Duplicados: insertar a la izquierda o a la derecha.
- *insort* () es un alias de *insort_right* ().
- *bisect* () es un alias de *bisect_right* ().
- *bisect_left* (), *insort_left* ()

[Hellman, 2011, pág 95-96]

```
1 >>>l = sorted([random.randint(0, 9) for x in range(15)])
2 >>>l
3 [0, 0, 3, 3, 3, 3, 3, 5, 6, 6, 7, 7, 8, 8, 8]
4 >>>bisect.bisect(l, 3)
5 7
6 >>>bisect.bisect_left(l, 3)
7 2
```

bisect (IV)

```
>>>def grade(score, breakpoints=[60, 70, 80, 90],
                    grades='FDCBA'):
      i = bisect.bisect(breakpoints, score)
      return grades[i]

>>>[grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

`https://docs.python.org/3.3/library/bisect.html`

- No tiene argumentos *key* ni *reversal*.
- Usar claves precalculadas.

bisect (V)

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8),
            ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]; keys
[0, 1, 5, 8]
>>> data[bisect.bisect_left(keys, 0)]
('black', 0)
>>> data[bisect.bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect.bisect_left(keys, 5)]
('red', 5)
>>> data[bisect.bisect_left(keys, 8)]
('yellow', 8)
```

https://docs.python.org/3.3/library/bisect.html

queue (I)

- Módulo *Queue*.
- Estructura FIFO adecuada para *threads*.
- Añadir con *put*() en uno de los extremos.
- Eliminar del otro extremo con *get*().

[Hellman, 2011, pág 96-97]

```
1  >>>import queue
2
3  >>>q = queue.Queue()
4
5  >>>for i in range(5):
6          q.put(i)
7
8  >>>while not q.empty():
9          print(q.get(),
10                end = "␣")
11
12 0 1 2 3 4
```

queue (II)

- Colas LIFO (¿?): *LifoQueue*

```
1 >>>q = queue.LifoQueue()
2
3 >>>for i in range(5):
4         q.put(i)
5
6 >>>while not q.empty():
7         print(q.get(), end = "␣")
8
9 4 3 2 1 0
```

[Hellman, 2011, pág 97]

queue (III)

- Colas de prioridad: *PriorityQueue*

```
1  import queue
2  import threading
3
4  class Job:
5      def __init__(self, priority, description):
6          self.priority = priority
7          self.description = description
8          print('New job:', description)
9          return
10     def __lt__(self, other):
11         return self.priority < other.priority
```

[Hellman, 2011, pág 98]

queue (IV)

```
12 q = queue.PriorityQueue()
13
14 q.put( Job(3, 'Mid-level job') )
15 q.put( Job(10, 'Low-level job') )
16 q.put( Job(1, 'Important job') )
17
18 def process_job(q):
19     while True:
20         next_job = q.get()
21         print('Processing job:', next_job.description)
22         q.task_done()
```

[Hellman, 2011, pág 98]

queue (V)

```
23 workers = [ threading.Thread(target=process_job, args=(q,)),
24              threading.Thread(target=process_job, args=(q,)),
25              ]
26 for w in workers:
27     w.setDaemon(True)
28     w.start()
29
30 q.join()
```

[Hellman, 2011, pág 98]

queue (VI)

```
1 New job: Mid-level job
2 New job: Low-level job
3 New job: Important job
4 Processing job: Important job
5 Processing job: Mid-level job
6 Processing job: Low-level job
```

[Hellman, 2011, pág 99]

pprint (I)

- Módulo *pprint*: *"pretty printer"*.
- Representaciones de estructuras que pueden ser interpretadas correctamente por el intérprete.
- Y fáciles de leer.
- Si es posible, en una línea.
- Si varias líneas, indentadas.
- *pformat*() devuelve la cadena en vez de imprimir.

[Hellman, 2011, pág 123]

pprint (II)

```
1 data = [ (1, { 'a':'A', 'b':'B', 'c':'C', 'd':'D' }),
2          (2, { 'e':'E', 'f':'F', 'g':'G', 'h':'H',
3                'i':'I', 'j':'J', 'k':'K', 'l':'L',
4                }),
5          ]
```

```
1 from pprint_data import data
2
3 print 'PRINT:'
4 print data
5 print
6 print 'PPRINT:'
7 pprint(data)
```

[Hellman, 2011, pág 123]

pprint (III)

```
PRINT:
[(1, {'b': 'B', 'd': 'D', 'a': 'A', 'c': 'C'}), (2, {'j': 'J', 'l': '

PPRINT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'})]
```

pprint (IV)

```
1  class node(object):
2
3      def __init__ ( self , name, contents=[]):
4          self.name = name
5          self.contents = contents[:]
6
7      def __repr__ ( self ):
8          return ( 'node(' + repr( self.name) + ',␣' +
9                   repr( self.contents) + ')'
10                  )
```

[Hellman, 2011, pág 125]

pprint (V)

```
1 trees = [ node('node-1'),
2           node('node-2', [ node('node-2-1')]),
3           node('node-3', [ node('node-3-1')]),
4           node('node-4', [ node('node-4-1', [node('node-4-1-1')])])
5           ]
6
7 print( trees , "\n")
8
9 pprint( trees )
```

[Hellman, 2011, pág 125]

pprint (VI)

```
1 [node('node-1', []), node('node-2', [node('node-2-1', [])]), node('no
2
3 [node('node-1', []),
4  node('node-2', [node('node-2-1', [])]),
5  node('node-3', [node('node-3-1', [])]),
6  node('node-4', [node('node-4-1', [node('node-4-1-1', [])])])]
```

[Hellman, 2011, pág 125]

pprint (VII)

- Estructuras recursivas: referencia a la fuente original.

```
>>>local_data = ['a', 'b', 1, 2]
>>>local_data.append(local_data)
>>>print(local_data)
['a', 'b', 1, 2, [...]]
>>>print(id(local_data))
139908780033928
>>>pprint(local_data)
['a', 'b', 1, 2, <Recursion on list with id=139908780033928>]
```

[Hellman, 2011, pág 125-126]

pprint (VIII)

- Limite de anidamiento: parámetro opcional *depth*

```
>>> l = [1,[2,[3,[4,[5]]]]]
>>> pprint(l)
[1, [2, [3, [4, [5]]]]]
>>> pprint(l, depth=3)
[1, [2, [3, [...]]]]
>>> pprint(l, depth=1)
[1, [...]]
```

[Hellman, 2011, pág 125-126]

pprint (IX)

- Ajuste del ancho: argumento *width*.
- Si el ancho es demasiado pequeño, no se recortan las líneas si la sintaxis resultante fuera inválida.

[Hellman, 2011, pág 126-127]

```
>>>pprint(data, width=10)
[(1,
  {'a': 'A',
   'b': 'B',
   'c': 'C',
   'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'})]
```

Referencias (I)

[Hellman, 2011] El capítulo 2 está dedicado a estructuras de datos.


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