09 - Conjunts de dades

October 26, 2015

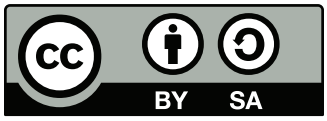

Figure 1: BY-SA

Authors : *Sonia Estradé*
José M. Gómez
Ricardo Graciani
Frank Güell
Manuel López
Xavier Luri
Josep Sabater

1 Tipus de dades estructurades

En capítols anteriors hem estudiat els tipus de dades bàsics, numèrics (`int, float, complex` i `bool`) i cadenes de caracters (strings), inclosos a l'especificació de Python. En aquest capítol ens centrarem en altres tipus de dades que reben el nom de col·leccions de dades estructurades. Així com els tipus bàsics només emmagatzemen a un únic valor o cadena, totes les col·leccions de dades es caracteritzen per poder emmagatzemar més d'un valor i per ser també iterables. Com ja veurem, les col.leccions són molt flexibles i el seu ús és molt freqüent en els programes.

Els tipus de dades estructurades que estudiarem a continuació són:

1. Llistes.
2. Tuples i tuples amb nom
3. Diccionaris
4. Conjunts (sets)

1.1 Llistes

Una llista (**list** en Python), de forma similar a un string, és una seqüència ordenada de valors. Cada dada (o valor) individual s'anomena **element** de la llista i **es pot accedir a ells mitjançant índexs**. Un índex és un valor enter que indica la posició de l'element en la seqüència ordenada al qual es vol accedir. Aquest índex especifica un desplaçament, com ja veurem, des de l'origen (esquerra) o el final (dreta) de la llista.

Una llista es diferencia d'un string (que només pot contenir caràcters) en:

- Una llista pot contenir una seqüència composada per qualsevol tipus de valors. Poden conviure diferents tipus de dades bàsics dins d'una mateixa llista.
- Una llista és **mutable**. Això vol dir que, a diferència d'un string, podem modificar el seu contingut després de la seva inicialització.

1.1.1 Creació de llistes

Com els tipus bàsics de dades que ja hem vist una llista es pot associar a un nom de variable, el qual permetrà després usar-la i accedir als seus diferents elements.

Les llistes poden ser creades mitjançant el seu constructor (**list**):

```
In [1]: # Creem una llista de strings mitjançant el constructor
        l_week_days = list( ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday') )

        # Mostrem el contingut
        print('l_week_days =', l_week_days)
```

```
l_week_days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

O amb claudàtors [ ]:

```
In [2]: l_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

        # Mostrem el contingut
        print('l_numbers =', l_numbers)
```

```
l_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Per tal de facilitar la creació de llistes numèriques d'enters python proporciona la funció **range()**. Podem reescriure l'exemple anterior usant aquesta funció de la següent forma:

```
In [3]: l_numbers_01 = list ( range(1, 10) )

        # Mostrem el contingut
        print('l_numbers_01 =', l_numbers_01)
```

```
l_numbers_01 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La sintaxi d'aquesta funció és: `range(inici, final[, pas])`

- Els arguments de la funció **range** han de ser nombres enters. Si no s'especifica el pas el seu valor serà 1. Si no s'especifica l'inici el primer valor serà el 0
- Si el pas és positiu, el contingut ve determinat per la fòrmula:

$$r[i] = start + step * i \quad on \quad i >= 0 \; and \; r[i] < stop$$

- Si el pas és negatiu, el contingut ve determinat per la fòrmula:

$$r[i] = start + step * i \quad on \quad i >= 0 \; and \; r[i] > stop$$

Veiem un exemple on s'especifica el pas:

```
In [4]: l_numbers_02 = list ( range(10, 110, 10) )

        # Mostrem el contingut
        print('l_numbers =', l_numbers_02)
```

```
l_numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

De forma similar podem construir una llista amb els mateixos valors que l'exemple anterior, però en ordre decreixent:

```
In [5]: l_numbers_03 = list ( range(100, 0, -10) )

        # Mostrem el contingut
        print('l_numbers =', l_numbers_03)
```

```
l_numbers = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Es poden crear llistes buides, a les que posteriorment es poden afegir elements:

```
In [6]: empty_list = []
        empty_plus = list()

        print (empty_list)
        print (empty_plus)
```

```
[]
[]
```

Podem també crear llistes que continguin valors de diferent tipus:

```
In [7]: #Llista mixta
        l_mixed = [134, 2, 'a', 3.14159, True, 'Julius', 3.0+6j]

        # Mostrem el contingut
        print('l_mixed =', l_mixed)
```

```
l_mixed = [134, 2, 'a', 3.14159, True, 'Julius', (3+6j)]
```

1.1.2 Operacions bàsiques

Donat que les llistes són seqüencies com els **string**, aquestes suporten moltes de les operacions que podem realitzar amb les cadenes de text:

Operació	Descripció
<list> + <list>	unió o concatenació de llistes
<list> * <int>	repetició dels elements de la llista
len(<list>)	nombre d'elements que conté la llista
<value> in <list>	indica si el valor indicat està a la llista

Nombre d'elements o mida d'una llista (len):

```
In [8]: l_00 = ['Hi', 'Bye', 'See You', 'Hello']
        l_01 = list ( range(8) )

        # Càlcul longitud
        n_l_00 = len( l_00 )
        n_l_01 = len( l_01 )

        text = 'La llista {} té {} elements.'
        print( text.format(l_00, n_l_00) )
        print( text.format(l_01, n_l_01) )
```

```
La llista ['Hi', 'Bye', 'See You', 'Hello'] té 4 elements.
La llista [0, 1, 2, 3, 4, 5, 6, 7] té 8 elements.
```

Unió o concatenació (+):

```
In [9]: l_02 = list ( range(5) )
        l_03 = list ( range(100, 105) )
        l_04 = ['Ernie', 'Paul', 'Lucy']

        # Concatenació de llistes
        print ('Concatenant', l_02, 'i', l_03, '::', l_02 + l_03)
        print ('Concatenant', l_04, 'i', l_02, '::', l_04 + l_02)
```

```
Concatenant [0, 1, 2, 3, 4] i [100, 101, 102, 103, 104] :: [0, 1, 2, 3, 4, 100, 101, 102, 103, 104]
Concatenant ['Ernie', 'Paul', 'Lucy'] i [0, 1, 2, 3, 4] :: ['Ernie', 'Paul', 'Lucy', 0, 1, 2, 3, 4]
```

Repetició (*):

```
In [10]: print ( ['Hallelujah', 'Brother!'] * 3 )
```

```
['Hallelujah', 'Brother!', 'Hallelujah', 'Brother!', 'Hallelujah', 'Brother!']
```

in: Retorna `True` o `False` en funció de si l'element especificat és troba o no contingut en la llista donada:

```
In [11]: transport = ['truck', 'car', 'bike', 'cicle', 'bus', 'train', 'rocket']

         print ( 'train' in transport )
         print( 'jet' in transport )
```

```
True
False
```

1.1.3 Accedint al contingut d'una llista:

Tal i com ja vàrem veure amb el tipus **string**, podrem també accedir als diferents elements d'una llista mitjançant l'operador claudàtor []. Podem establir la següent classificació en funció del nombre d'elements als quals accedirem:

1. **Indexació**: Accedim únicament a un element de la llista. Especifiquem un desplaçament o índex, el qual ha de ser un valor enter.

   - Mitjançant valors **positius** d'índex s'accedeix des de l'inici (esquerra) de la llista.
   - Mitjançant valors **negatius** d'índex s'accedeix des del final (dreta) de la llista.

   Sintaxi:

   `<nom_llista>[<valor_índex>]`

2. **Slicing o subconjunts**: En Python un segment (conjunt de valors) d'una llista s'anomena `slice`. Per tant, emprarem slicing si volem accedir a la vegada a més d'un element d'una llista. Com en el cas del la indexació, també admet l'ús d'enters positius i negatius en funció de si hi accedim des del principi o des del final de la llista.

   Sintaxi:

   `<nom_llista>[<índex_inicial>:<índex_final>:<pas>]`

   - On:
     - Sí s'omet el valor d'`índex_inicial`, aquest pren el valor de **0** (inici llista).
     - Sí s'omet el valor d'`índex_final`, aquest pren el valor de `len(<nom_llista>)`.

  - Sí s'omet el valor del **pas**, aquest pren el valor de **1**.
  - A l'igual que el **range()**, l'element especificat per l'**índex_final** mai està inclós.

Important:

- L'índex del primer element d'una llista accedint desde l'esquerra, a l'igual que en un string, és el *0*.
- Noteu però que en cap cas es poden usar índexs més enllà de la mida de la llista (genera un error).

Indexació Accedim a cadascun dels elements d'una llista especificant la seva posició des de l'inici:

```
In [12]: # Creem la llista
         latin_list = ['Lorem', 'ipsum', 'dolor sit amet']
         print ("Contingut: ", latin_list)

         # S'accedeix al valor usant el seu índex, començant per zero
         print ('Valor índex 0: ', latin_list[0])
         print ('Valor index 1: ', latin_list[1])
         print ('Valor index 2: ', latin_list[2])
```

```
Contingut:  ['Lorem', 'ipsum', 'dolor sit amet']
Valor índex 0:  Lorem
Valor index 1:  ipsum
Valor index 2:  dolor sit amet
```

Podem accedir al darrer element de dues maneres diferents, desde l'inici (desplaçament positiu) i des del final (desplaçament negatiu):

```
In [13]: print ('[0] Darrer element: ', latin_list[ len(latin_list)-1 ])
         print ('[1] Darrer element: ', latin_list[-1])
```

```
[0] Darrer element:  dolor sit amet
[1] Darrer element:  dolor sit amet
```

I el penúltim el podem obtenir així:

```
In [14]: print ('[0] Penúltim element: ', latin_list[ len(latin_list)-2 ])
         print ('[1] Penúltim element: ', latin_list[-2])
```

```
[0] Penúltim element:  ipsum
[1] Penúltim element:  ipsum
```

A més, en el cas que els elements d'una llista siguin strings, és possible accedir a una posició determinada d'un string determinat. Veiem-ne un exemple:

```
In [15]: fruits = ['Cherry', 'Banana', 'Grape', 'Kiwi', 'Lemon']

         print ("La darrera lletra de '{}' és una '{}'.".format( fruits[2], fruits[2][-1]) )
```

```
La darrera lletra de 'Grape' és una 'e'.
```

Slicing: Utilitzant slicing podem accedir a tots els elements d'una llista (començant pel seu inici, esquerra) que es trobin en posicions senars:

```
In [16]: #Llista de ciutats
         cities = ['Vladivostok', 'Lyon', 'Glasgow', 'Rome', 'Munich', 'Miami', 'Kyoto', 'Zagreb']

         print ( cities[0:len( cities):2] )
```

```
['Vladivostok', 'Glasgow', 'Munich', 'Kyoto']
```

Noteu que el resultat és també una llista, en aquest cas de 4 elements.

Podem obtenir el mateix resultat ometent els índex inicials i finals (els quals prendran el seu valor per defecte):

```
In [17]: print ( cities[::2] )
```

```
['Vladivostok', 'Glasgow', 'Munich', 'Kyoto']
```

Podem obtenir la mateixa subllista de ciutats però en ordre invers tot accedint a la llista de ciutats des del final i mitjançant un pas negatiu:

```
In [18]: print ( cities [len(cities)-2::-2] )
```

```
['Kyoto', 'Munich', 'Glasgow', 'Vladivostok']
```

Podem obtenir tota la llista en ordre invers si la recorrem de dreta a esquerra de la següent manera:

```
In [19]: print ( cities [::-1] )
```

```
['Zagreb', 'Kyoto', 'Miami', 'Munich', 'Rome', 'Glasgow', 'Lyon', 'Vladivostok']
```

Podem obtenir subllistes de valors consecutius (pas 1, valor per defecte si no especifiquem res):

```
In [20]: print ( cities[:3] )
         print ( cities[3:] )
         print ( cities[2:4] )
```

```
['Vladivostok', 'Lyon', 'Glasgow']
['Rome', 'Munich', 'Miami', 'Kyoto', 'Zagreb']
['Glasgow', 'Rome']
```

1.1.4 Ús dels elements d'una llista

Els elements d'una llista poden tractar-se com variables normals, per fer qualsevol operació permesa:

```
In [21]: integers = list( range(2,20,3) )

         # Sumem alguns elements de la llista
         add_value = integers[0]-integers[3]+integers[4]
         print ("Integers: ", integers)
         print ("Suma índexs 0,3,4: ", add_value)
```

```
Integers:  [2, 5, 8, 11, 14, 17]
Suma índexs 0,3,4:  5
```

1.1.5 Afegir i eliminar elements d'una llista

Python proporciona diversos mètodes per manipular el contigut d'una llista:

Mètode	Descripció
`extend(values)`	amplia la llista afegint tots elements de `value` al final de la llista
`append(value)`	afegeix un element al final de la llista
`pop(index)`	elimina i **retorna** l'element amb l'índex donat
`insert(index, value)`	afegeix un valor en una posició determinada
`remove(value)`	elimina el primer element que conté el valor donat

En el cas del mètode **pop()**, si no especifiquem índex es pren el darrer element.
Tots aquests mètodes s'apliquen sobre una llista usant la sintaxi <nom_llista>**.mètode()**

- **Important**: les modificacions s'efectuen directament sobre la llista indicada. Excepte `pop`, els altres mètodes retornen `None`. Cal evitar construccions de la forma: **list_test = list_test.append(X)**

Veiem en acció tots els mètodes anteriorment descrits:

```
In [22]: # Creem la llista original
         sports = ["cycling", "basketball"]
         print ("Llista original: ", sports)

         # Afegim més d'un element al final de la llista
         sports.extend( ["taekwondo", "rugby"] )
         print ("Extending: ", sports)

         # Afegim un únic element al final de la llista
         sports.append( ["running", "gymnastics"] )
         print ("Appending: ", sports)

         # Treiem un element donant l'índex i guardem el valor esborrar a 'el'
         el = sports.pop(3);
         print ("Popping '{}': {}".format(el, sports))

         # Afegim un element al mig de la llista
         sports.insert(2, "handball") # Cal indicar l'index on s'inserta el valor
         print ("Inserting index 2 ", sports)

         # Eliminem un element a partir del seu contingut
         sports.remove("basketball") # Elimina el primer element que contingui "X"
         print ("Removing 'basketball': ", sports)

         # Treiem el darrer element i guardem el valor esborrar a 'el'
         el = sports.pop();
         print ("Popping {}: {}".format(el, sports))
```

```
Llista original:  ['cycling', 'basketball']
Extending:  ['cycling', 'basketball', 'taekwondo', 'rugby']
Appending:  ['cycling', 'basketball', 'taekwondo', 'rugby', ['running', 'gymnastics']]
Popping 'rugby': ['cycling', 'basketball', 'taekwondo', ['running', 'gymnastics']]
Inserting index 2  ['cycling', 'basketball', 'handball', 'taekwondo', ['running', 'gymnastics']]
Removing 'basketball':  ['cycling', 'handball', 'taekwondo', ['running', 'gymnastics']]
Popping ['running', 'gymnastics']: ['cycling', 'handball', 'taekwondo']
```

Noteu les diferències entre el mètode `extend` i l'`append`. El primer permet inserir en una llista una sèrie de valors que són en una altra llista. En canvi el segon insereix únicament un element nou al final, la segona llista de valors (com a llista).

1.1.6 Sentència `del`

També és possible esborrar elements d'una llista directament mitjançant la sentència **del**, ja sigui indicant-li un índex o bé un conjunt amb `slicing`:

```
In [23]: test_list = list ( range(5, 55, 5) )
         print('Llista original: ', test_list)
```

```
        del test_list[3]
        print('Esborrem elem. 3: ', test_list)

        del test_list[::2]
        print('Esborrem elem. senars: ', test_list)
```

```
Llista original:  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
Esborrem elem. 3:  [5, 10, 15, 25, 30, 35, 40, 45, 50]
Esborrem elem. senars:  [10, 25, 35, 45]
```

1.1.7 Modificar elements d'una llista

Al tractar-se d'un tipus de dada mutable podem modificar el seu contingut directament especificant `indexació` o `slicing`:

```
In [24]: cereals = ['maize', 'rice', 'wheat', 'barley', 'millet', 'quinoa']
         print('Llista original cereals: ', cereals)

         #Modificant contingut d'un únic element
         cereals[1] = 'sorghum'
         print('Llista cereals: ', cereals)

         #Modificant contingut d'un subconjunt d'elements
         cereals[2:5:2] = ['oats', 'fonio']
         print('Llista cereals: ', cereals)
```

```
Llista original cereals:  ['maize', 'rice', 'wheat', 'barley', 'millet', 'quinoa']
Llista cereals:  ['maize', 'sorghum', 'wheat', 'barley', 'millet', 'quinoa']
Llista cereals:  ['maize', 'sorghum', 'oats', 'barley', 'fonio', 'quinoa']
```

1.1.8 Funcions avançades

Presentem aquí algunes funcions avançades per treballar amb llistes, però us remetem a la documentació de Python per a un llistat complet de les eines disponibles.

Cerca d'índex La funció `index()` permet buscar l'índex d'un element que contingui un patró donat.

```
In [25]: llista = ["A","B","C","D","E","A"]

         # Es pot buscar l'índex del primer element de la llista amb la funció "index"
         print("Index de 'A': ",llista.index("A")) # Només dóna el primer!
         print("Un altre index de 'A': ",llista[1:6].index("A")) # Ara dóna l'últim
         print("Index de 'B': ",llista.index("B"))

         # Atenció: si l'element no existeix dóna un error
         #print("Index de 'p': ",llista.index("p"))
```

```
Index de 'A':  0
Un altre index de 'A':  4
Index de 'B':  1
```

Ordenació Una llista es pot ordenar, de forma ascendent o descendent, amb la funció `sort()`:

```
In [26]: llista = [5,6,2,9,6,1,0,4]
         print ("Abans d'ordenar: ", llista)
```

```
        llista.sort() # Nota: el contingut de la llista és reemplaçat
        print ("Després d'ordenar: ", llista)

        # També es pot fer amb llistes de cadenes de text
        animals = ['lion', 'elefant', 'rat', 'jackal', 'gecko', 'lemur']
        print ("Abans d'ordenar: ", animals)

        animals.sort()
        print ("Després d'ordenar (ordre creixent): ", animals)

        animals.sort(reverse=True)
        print ("Després d'ordenar (ordre decreixent): ", animals)
```

```
Abans d'ordenar:  [5, 6, 2, 9, 6, 1, 0, 4]
Després d'ordenar:  [0, 1, 2, 4, 5, 6, 6, 9]
Abans d'ordenar:  ['lion', 'elefant', 'rat', 'jackal', 'gecko', 'lemur']
Després d'ordenar (ordre creixent):  ['elefant', 'gecko', 'jackal', 'lemur', 'lion', 'rat']
Després d'ordenar (ordre decreixent):  ['rat', 'lion', 'lemur', 'jackal', 'gecko', 'elefant']
```

Capgirar una llista Una llista es pot capgirar amb la funció `reverse()`:

```
In [27]: data_00 = list( range(100, 200, 10) )
         print ("Llista original: ", data_00)
         data_00.reverse()
         print ("Llista capgirada: ", data_00)
```

```
Llista original:  [100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
Llista capgirada:  [190, 180, 170, 160, 150, 140, 130, 120, 110, 100]
```

1.1.9 Llistes multidimensionals

Fins ara hem vist llistes "unidimensionals", on cada element queda identificat per un únic índex. Es poden definir llistes multidimensionals senzillament creant llistes els elements de les quals siguin altres llistes.

Per exemple

```
list_2D = [ [1,2] , [3,4] ]
```

Aquesta és una llista amb dos elements: `[1,2]` i `[3,4]`, però cadascun d'aquests elements és una altra llista de dos elements. El resultat és de fet una matriu 2x2 i es poden accedir als seus elements mitjançant dos indexs:

```
print(list_2D[0][1])
```

Veiem-ho a la pràctica usant una llista per a guardar una matriu 3x3:

```
In [28]: # Creem una llista contenint una matriu 3x3
         matrix_3x3 = [ [1,2,3], [4,5,6], [7,8,9] ]

         sep = '------------\n'
         print("La meva matriu:", matrix_3x3)
         print(sep)

         print ( "Element (0): {} - {}".format(matrix_3x3[0], type(matrix_3x3[0])) )
         print ( "Element (1): {} - {}".format(matrix_3x3[1], type(matrix_3x3[1])) )
         print ( "Element (2): {} - {}".format(matrix_3x3[2], type(matrix_3x3[2])) )
         print(sep)
```

```
        # Imprimim alguns elements
        print ( "Element (0,0): {} - {}".format(matrix_3x3[0][0], type(matrix_3x3[0][0])) )
        print ( "Element (1,1): {} - {}".format(matrix_3x3[1][1], type(matrix_3x3[1][1])) )
        print ( "Element (2,2): {} - {}".format(matrix_3x3[2][2], type(matrix_3x3[2][2])) )
        print(sep)
```

```
La meva matriu: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
------------

Element (0): [1, 2, 3] - <class 'list'>
Element (1): [4, 5, 6] - <class 'list'>
Element (2): [7, 8, 9] - <class 'list'>
------------

Element (0,0): 1 - <class 'int'>
Element (1,1): 5 - <class 'int'>
Element (2,2): 9 - <class 'int'>
------------
```

Noteu que les llistes multidimensionals, al contrari que les matrius algebràiques, no tenen per què ser regulars, com es veu en l'exemple següent.

```
In [29]: # Creem una llista 2D irregular
         list_2D = [ [1,2,3], [4,5], [7,8,9,0], [1] ]

         print ( "Element (0): {} - {}".format(list_2D[0], type(list_2D[0])) )
         print ( "Element (1): {} - {}".format(list_2D[1], type(list_2D[1])) )
         print ( "Element (2): {} - {}".format(list_2D[2], type(list_2D[2])) )
         print ( "Element (3): {} - {}".format(list_2D[3], type(list_2D[3])) )
```

```
Element (0): [1, 2, 3] - <class 'list'>
Element (1): [4, 5] - <class 'list'>
Element (2): [7, 8, 9, 0] - <class 'list'>
Element (3): [1] - <class 'list'>
```

```
In [30]: # Llista 1D
         list_1D = [1,2,3,4,5]
         print ("Llista: ", list_1D)
         print ("Mida de la llista: ", len(list_1D))
         print('------------\n')

         # Llista 2D
         list_2D = [[1,2], [3,4,5], [5]]
         print ("Llista: ", list_1D)
         print ("Nombre files: ", len(list_2D))  # 3
         print ("Num. columnes de la fila 0: ", len(list_2D[0]))  # 2
         print ("Num. columnes de la fila 1: ", len(list_2D[1]))  # 3
         print ("Num. columnes de la fila 2: ", len(list_2D[2]))  # 1
```

```
Llista:  [1, 2, 3, 4, 5]
Mida de la llista:  5
------------

Llista:  [1, 2, 3, 4, 5]
```

```
Nombre files:  3
Num. columnes de la fila 0:  2
Num. columnes de la fila 1:  3
Num. columnes de la fila 2:  1
```

Llista de llistes heterogenies: Podem també treballar amb llistes que a la vegada cada element sigui una altra llista de valors heterogenis. S'accedeix als seus elements de la mateixa manera que hem vist en l'apartat anterior:

```
In [31]: # Definim una llista on cada element conté ['nom', 'edat', 'pes']
         person_list = [ ['Patty', 22, 58], ['Ricky', 44 , 82.5], ['Johnny', 18, 75] ]

         row = person_list[1]
         print('Fila 1: ', row)

         column = row[2]
         print('Element[1][2]: ', column)
```

```
Fila 1:  ['Ricky', 44, 82.5]
Element[1][2]:  82.5
```

1.2 Tuples

Les tuples són una estructura de dades molt similar a les llistes. La diferència entre els dos tipus de dades i la decisió de quan usar una o altra pot ser una mica subtil, possiblement més enllà dels objectius d'aquest curs.

En qualsevol cas la principal diferència és:

- Les **tuples** a l'igual que els strings (i al contrari que les llistes i els diccionaris) són **immutables**: un cop definides no és possible alterar el seu contingut.

Una tupla (**tuple** en Python), a l'igual que un string i una llista, és una col.lecció de valors. Cada dada (o valor) individual s'anomena **part** de la tupla i **es pot accedir a elles mitjançant índexs**. Com ja hem vist per a les llistes un índex és un valor enter que especifica la posició de l'element de la seqüència ordenada al qual es vol tenir accedir. Aquest índex especifica un desplaçament, com ja veurem, des de l'origen (esquerra) o el final (dreta) de la llista.

1.2.1 Creació de tuples

Les llistes poden ser creades mitjançant el seu constructor (`tuple`). El contingut de la tupla ha d'anar entre parèntesi, i els diferents camps separats per comes:

```
In [32]: tuple_one = tuple( ("John", "Smith", 45, 1.70, ('name', 'last_name')) )
         print (tuple_one)
```

```
('John', 'Smith', 45, 1.7, ('name', 'last_name'))
```

Noteu que el símbol que caracteritza a una tupla són els **( )**, així com els [ ] caracteritzen a una llista.

Com podeu veure en l'exemple anterior qualsevol de les parts d'una llista pot ser a la seva vegada un tipus de dada estructurat.

De forma similar a les llistes, una tupla pot crear-se mitjançant l'operador parèntesi (**( )**):

```
In [33]: tuple_two = ('cookies', 10, 'ham', 45, 'eggs', 5, 'apples', 8)
         print(tuple_two)
```

```
('cookies', 10, 'ham', 45, 'eggs', 5, 'apples', 8)
```

Podem també crear tuples buides tot i que després, a diferència de les llistes, no podrem actualitzar el seu contingut:

```
In [34]: empty_tuple = ()
         empty_tuple_plus = tuple()

         print(empty_tuple)
         print(empty_tuple_plus)
```

```
()
()
```

1.2.2 Cas a tenir en compte

Si el que volem és una tupla que contingui un únic element, cal afegir al final i abans del parèntisi una coma:

```
In [35]: value = (56)
         tuple_one_value = (56,)

         print('Un valor:', value)
         print('Una tupla:', tuple_one_value)
```

```
Un valor: 56
Una tupla: (56,)
```

1.2.3 Operacions bàsiques

Igual que les llistes, les tuples suporten moltes de les operacions que podem realitzar amb les cadenes de text:

Operació	Descripció
<tuple> + <tuple>	unió o concatenació de tuple
<tuple> * <int>	repetició dels elements de la tuple
len(<tuple>)	nombre d'elements que conté la tuple
<value> in <tuple>	indica si el valor indicat està a la tuple

Nombre de parts o mida d'una tuple (len):

```
In [36]: print( "El nombre de parts de {} és {}".format(tuple_one, len(tuple_one)) )
         print( "El nombre d'elements de\n\t{}\n\tés {}".format(tuple_two, len(tuple_two)) )
```

```
El nombre de parts de ('John', 'Smith', 45, 1.7, ('name', 'last_name')) és 5
El nombre d'elements de
        ('cookies', 10, 'ham', 45, 'eggs', 5, 'apples', 8)
        és 8
```

Unió o concatenació (+):

```
In [37]: tuple_name = ('Ernie', 'Paul', 'Lucy')
         tuple_last_name = ('Williams', 'Howard', 'Johnson')

         new_tuple = tuple_name + tuple_last_name
         print(new_tuple)
```

```
('Ernie', 'Paul', 'Lucy', 'Williams', 'Howard', 'Johnson')
```

Repetició (*):

```
In [38]: tuple_ora = ('ora', 'pro', 'nobis', 123) * 3
         print(tuple_ora)
```

```
('ora', 'pro', 'nobis', 123, 'ora', 'pro', 'nobis', 123, 'ora', 'pro', 'nobis', 123)
```

in: Retorna `True` o `False` en funció de si l'element especificat és troba o no contingut en la tupla:

```
In [39]: print( "Tupla:\n\t", tuple_two )
         print( "\nL'element 'ham' existeix?", 'ham' in (tuple_two) )
```

```
Tupla:
         ('cookies', 10, 'ham', 45, 'eggs', 5, 'apples', 8)

L'element 'ham' existeix? True
```

Sempre podem comprovar si valor està o no contingut en una tupla, i en funció del resultat fer una acció o una altra:

```
In [40]: tuple_two = ('cookies', 10, 'ham', 45, 'eggs', 5, 'apples', 8)

         if 'eggs' not in tuple_two:
             print('No valor "eggs"')
         else:
             print('Existeix "eggs"')
```

```
Existeix "eggs"
```

1.2.4 Accedint al contingut d'una tupla

Com en cas de les llistes s'accedeix a un valor mitjançant l'operador claudator []:

```
In [41]: # Diccionari on el camp valor és de tipus simple
         tuple_three = ('qui', 'quae', 'quod', 'quorum', 'quarum')

         print ('Primer element: ', tuple_three[0])
         print ('Segon element: ', tuple_three[1])
         print ('Tercer element: ', tuple_three[2])
         print ('Quart element: ', tuple_three[3])
         print ('Cinqué element: ',tuple_three[4])
```

```
Primer element:  qui
Segon element:  quae
Tercer element:  quod
Quart element:  quorum
Cinqué element:  quarum
```

També és possible accedir a un cojunt de valors mitjançant *slicing*. Adrecem al lector a l'apartat corresponent de les llistes per tal de veure el funcionament de l'slicing.

1.2.5 Modificar elements d'una tupla

Recordem que una tupla és un tipus de dades **immutable**, per tant el seu contingut no es pot modificar una vegada aquesta hagi estat inicialitzada. Si s'intenta alterar part del seu contingut, obtindrem un error similar al següent:

```
In [42]: t_cheese = ('manchego', 'roncal', 'idiazabal', 'garrotxa', 'tupí', 'cabrales')
         t_cheese[1] = 'cuajada'


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-42-f53f9c420a54> in <module>()
           1 t_cheese = ('manchego', 'roncal', 'idiazabal', 'garrotxa', 'tupí', 'cabrales')
     ----> 2 t_cheese[1] = 'cuajada'


         TypeError: 'tuple' object does not support item assignment
```

1.2.6 Conversions

Una tupla es pot convertir en una llista i utilitzar qualsevol mètode d'aquest tipus:

```
In [43]: t_cheese = ('manchego', 'roncal', 'idiazabal', 'garrotxa', 'tupí', 'cabrales')

         l_cheese = list(t_cheese)
         l_cheese.append('cuajada')
         l_cheese.sort()
         print ( l_cheese )
```

```
['cabrales', 'cuajada', 'garrotxa', 'idiazabal', 'manchego', 'roncal', 'tupí']
```

1.2.7 Arguments d'un print

Una tupla es pot passar com un argument d'un print i imprimir les seves parts separadament. Els elements de la tupla usaran els especificadors de format de forma successiva, com es veu en el següent exemple:

```
In [44]: tuple_person_00 = ("Percy", "Strait", 45, 1.70)
         print ("Nom: %s  Cognom: %s  Edat: %d anys  Alçada: %f m." % tuple_person_00)

         tuple_person_01 = ("John", "Hopkins", 24, 1.98)
         print ("Nom: %s  Cognom: %s  Edat: %d anys  Alçada: %f m." % tuple_person_01)
```

```
Nom: Percy  Cognom: Strait  Edat: 45 anys  Alçada: 1.700000 m.
Nom: John  Cognom: Hopkins  Edat: 24 anys  Alçada: 1.980000 m.
```

1.2.8 Desempaquetat d'elements d'una tupla

Amb una única sentència podem assignar directament cadascun dels seus elements a variables diferents i després operar amb elles:

```
In [45]: point_3D_1 = (0.98, 0.45, 34.2)
         point_3D_2 = (15, 89.45, 2)

         x1, y1, z1 =  point_3D_1
         x2, y2, z2 =  point_3D_2

         print( 'Suma de components: ({}, {}, {})'.format(x1+x2, y1+y2, z1+z2) )
```

```
Suma de components: (15.98, 89.9, 36.2)
```

1.2.9 Retorn de múltiples valors des d'una funció

A vegades és necessari que una funció retorni més d'un valor. Això es pot aconseguir retornant una llista, però és més pràctic fer-ho com una tupla.

En aquest segon cas es pot rebre com a una tupla o com els seus valors separadament, com es veu en el següent exemple.

```
In [46]: import random

         def return_2_values():
             """ Aquesta funció és un exemple de retorn de dos valors en forma de tupla
             """
             x = random.randint(0, 100)
             y = random.randint(500, 1e3)

             # Retornem una tupla amb els dos valors
             return (x, y)

         # Podem cridar la funció i obtenir els dos valors a la
         # vegada, en variables separades
         val_x, val_y = return_2_values()
         print ( "[0] val_x = {}, val_y = {}".format(val_x, val_y) )

         # També es pot rebre com una tupla, si cal
         tuple_val = return_2_values()
         print ( "[1] val_x = {}, val_y = {}".format(tuple_val[0], tuple_val[1]) )
```

```
[0] val_x = 78, val_y = 909
[1] val_x = 29, val_y = 632
```

1.3 Tuples amb nom (*named tuples*)

Python permet etiquetar amb un nom els elements constituents d'una tupla mitjançant les **named tuples** . Això facilita l'ús d'aquests components ja que es poden referenciar amb els noms assignats.

1.3.1 Creació de *named tuples*

Les *named tuples* es creen mitjançant el constructor **namedtuple()**, present al mòdul **collections** (noteu que cal importar-lo per a usar-les). El primer argument és el nom de la `namedtuple` i el segon és un string que conté els noms de cadascun dels atributs, separats per una coma o un espai en blanc. Un cop definida una `namedtuple` es pot usar per crear instancies individuals amb la seva estructura, com es veu en l'exemple següent:

```
In [47]: # Cal fer aquesta importació per usar les tuples amb nom
         from collections import namedtuple

         # Definim una tupla amb nom per a representar punts en el pla
         # Les seves dues components són les coordenades x i y
         Point = namedtuple('point2D', 'x y')

         # Creem un parell de punts usant aquesta tupla amb nom
         p1 = Point(1.0, 5.0)
         p2 = Point(y=2.5, x=1.5)

         print (p1)
         print (p2)
```

```
point2D(x=1.0, y=5.0)
point2D(x=1.5, y=2.5)
```

Un altre exemple:

```
In [48]: from collections import namedtuple

         person = namedtuple('record', 'name, age, jobs')
         johnny = person(name='Johnny Guitar', age=25, jobs=['musician', 'painter'])

         print(johnny)
```

```
record(name='Johnny Guitar', age=25, jobs=['musician', 'painter'])
```

1.3.2 Accedint al contingut d'una *named tuple*

A l'igual que en una tupla convencional, es pot accedir a un valor mitjançant l'operador claudator [] però en aquest cas també es pot accedir a través de la seva etiqueta:

```
In [49]: # Usem la tupla Point definida anteriorment
         # Podem accedir a les coordenades a partir del seu nom o de la forma usual
         scalar_prod_1 = p1[0]*p2[0] + p1[1]*p2[1]  # Accés per índex
         scalar_prod_2 = p1.x*p2.x + p1.y*p2.y  # Accés per nom

         print (scalar_prod_1)
         print (scalar_prod_2)
```

```
14.0
14.0
```

Un altre exemple més:

```
In [50]: from math import sqrt

         length_1 = sqrt((p1.x-p2.x)**2 + (p1.y-p2.y)**2)
         length_2 = sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

         print (length_1)
         print (length_2)
```

```
2.5495097567963922
2.5495097567963922
```

També podem fer el desenpaquetat dels diferents elements, tal i com hem vist en les tuples convencionals:

```
In [51]: from collections import namedtuple

         person = namedtuple('record', 'name, age, jobs')
         johnny = person(name='Johnny Guitar', age=25, jobs=['musician', 'painter'])

         name, age, jobs = johnny

         print(name, age, jobs, sep=' - ')
```

```
Johnny Guitar - 25 - ['musician', 'painter']
```

1.4 Diccionaris

Així com les llistes són una col.lecció ordenada o seqüència de valors, els diccionaris contenen un conjunt de dades **no ordenat**.

Es tracta d'una especie de conjunt de parelles, on el primer element de la parella és la clau (**key**) i el segon es correspon al valor associat a aquesta clau. En els diccionaris no s'accedeix a les dades mitjançant un `índex` numèric o `slicing`, tal i com feiem en amb les llistes o les tuples, si no que s'accedeix mitjançant la clau. És a dir, quan accedim a un diccionari, donada una clau determinada obtenim el seu valor associat.

S'anomenen diccionaris precisament per la seva similitud amb els diccionaris de paper tradicionals. En aquests s'accedeix a un significat determinat a través del valor d'una clau, en aquest cas una paraula.

Important:

- Les claus han de ser úniques (no es poden repetir en un mateix diccionari).
- Les claus han de ser immutables, per tant, usarem strings, nombres o tuples.

1.4.1 Creació de diccionaris

Els diccionaris poden ser creats mitjançant el seu constructor (**dict**):

```
In [52]: telephones = dict( [('Steve', 99090), ('Nick', 1234), ('Wayne', 78666)] )
         print(telephones)
```

```
{'Wayne': 78666, 'Nick': 1234, 'Steve': 99090}
```

En l'exemple anterior el diccionari s'especifica mitjançant parelles **(clau, valor)** i és assignat a la variable `telephones`. Concretament, la clau del primer element del diccionari és la cadena de text **'Steve'** i el valor associat és **99090**, la segona clau és **'Nick'** i el valor **1234**, etc.

També podem crear-ne un de manera simplificada si les claus són strings simples:

```
In [53]: movies_dicc = dict(Life_of_Brian=1979, Forrest_Gump=1994, Casablanca=1942, Star_Trek=1966)
         print(movies_dicc)
```

```
{'Life_of_Brian': 1979, 'Forrest_Gump': 1994, 'Star_Trek': 1966, 'Casablanca': 1942}
```

Tal i com es pot apreciar al fer un print, l'ordre en que es mostren els elements continguts en un diccionari pot no correspondre amb l'ordre de creació.

De forma similar a les llistes, un diccionari pot crear-se mitjançant l'operador claus (**{ }**):

```
In [54]: inventory_dicc = {'cookies': 10, 'ham': 45, 'eggs': 5, 'apples': 8}
         print(inventory_dicc)
```

```
{'apples': 8, 'cookies': 10, 'eggs': 5, 'ham': 45}
```

El valor associat a una clau pot ser de qualsevol tipus, incloent tipus de dades estructurades com ara llistes o altres diccionaris:

```
In [55]: person_dicc={'Patty': [22, 58], 'Ricky': [44 , 82.5], 'Johnny': [18, 75]}
         print(person_dicc)

         heter_dicc={"hello": ["h","e","l","l","o"], "numbers": [5,4,3,2,1]}
         print(heter_dicc)

         basket_dicc={'Jordan': {'name': 'Michael', 'age': 40}, 'Bird': {'name': 'Larry', 'age': 55}}
         print(basket_dicc)
```

```
{'Johnny': [18, 75], 'Ricky': [44, 82.5], 'Patty': [22, 58]}
{'numbers': [5, 4, 3, 2, 1], 'hello': ['h', 'e', 'l', 'l', 'o']}
{'Jordan': {'name': 'Michael', 'age': 40}, 'Bird': {'name': 'Larry', 'age': 55}}
```

Les claus no han de ser necessàriament strings, també podem ser nombres enters:

```
In [56]: int_keys_dicc = {2: 'Firebird', 1: 'Mustang',  4: 'Camaro', 100: 'Corolla', -1: 'Patrol'}
         print( int_keys_dicc )
```

```
{1: 'Mustang', 2: 'Firebird', 4: 'Camaro', -1: 'Patrol', 100: 'Corolla'}
```

Podem també crear diccionaris buits:

```
In [57]: empty_dicc = {}
         empty_plus = dict()
         print(empty_dicc)
         print(empty_plus)
```

```
{}
{}
```

1.4.2 Operacions bàsiques

Tot seguit mostrem les operacions bàsiques comuns entre llistes i diccionaris:

Operació	Descripció
len(<dictionary>)	nombre d'elements que conté el diccionari
<key> in <dictionary>	indica si la clau està al diccionari

Nombre d'elements o mida d'un diccionari (len):

```
In [58]: print( "El nombre d'elements de {} és {}".format(inventory_dicc, len(inventory_dicc)) )
         print( "El nombre d'elements de\n\t{}\n\tés {}".format(basket_dicc, len(basket_dicc)) )
```

```
El nombre d'elements de {'apples': 8, 'cookies': 10, 'eggs': 5, 'ham': 45} és 4
El nombre d'elements de
        {'Jordan': {'name': 'Michael', 'age': 40}, 'Bird': {'name': 'Larry', 'age': 55}}
        és 2
```

in: Retorna cert o fals en funció de si l'element especificat és troba o no contingut en el diccionary:

```
In [59]: print( "Diccionari:\n\t", basket_dicc )
         print( "\nLa clau 'Bird' existeix?", 'Bird' in (basket_dicc) )
```

```
Diccionari:
         {'Jordan': {'name': 'Michael', 'age': 40}, 'Bird': {'name': 'Larry', 'age': 55}}

La clau 'Bird' existeix? True
```

Sempre podem comprovar si una clau existeix o no en un diccionari, i en funció del resultat fer una acció o una altra:

```
In [60]: inventory_dicc = {'eggs':1, 'ham':2, 'bacon':5}

         if 'eggs' not in inventory_dicc:
             print('No existeix la clau')
         else:
             print('Els valor associat a "eggs" és ', inventory_dicc['eggs'])
```

```
Els valor associat a "eggs" és  1
```

1.4.3 Accedint al contingut d'un diccionari

Com ja hem comentat, s'accedeix a un valor mitjançant la seva clau i l'operador claudator []:

```
In [61]: inventory_dicc = {'eggs':1, 'ham':2, 'bacon':5}
         person_dicc = {'Ricky':[44, 82.5], 'John':[22,70.6]}
         basket_dicc = {'Shaquile':{'age': 35, 'name': 'ONeal'}, 'Jordan':{'age': 40, 'name': 'Michael'}


         # Diccionari on el camp valor és de tipus simple
         print("El valor de la clau 'ham' és: ", inventory_dicc['ham'])
         print('---------------\n')

         # Diccionari on el camp valor és una llista
         print("El valor de la clau 'Ricky' és: ", person_dicc['Ricky'] )
         print("El valor de la clau 'Ricky'/segon valor llista és: ", person_dicc['Ricky'][1] )
         print('---------------\n')

         # Diccionari on el camp valor és un altre diccionari
         print("El valor de la clau 'Jordan' és: ", basket_dicc['Jordan'] )
         print("El valor de la clau 'Jordan'/'name' és: ", basket_dicc['Jordan']['name'] )
         print('---------------\n')
```

```
El valor de la clau 'ham' és:  2
---------------

El valor de la clau 'Ricky' és:  [44, 82.5]
El valor de la clau 'Ricky'/segon valor llista és:  82.5
---------------

El valor de la clau 'Jordan' és:  {'name': 'Michael', 'age': 40}
El valor de la clau 'Jordan'/'name' és:  Michael
---------------
```

1.4.4 Afegir i eliminar elements d'un diccionari

Es poden afegir nous elements al diccionari senzillament amb una assignació:

```
In [62]: # Inserint valors simples
         inventory_dicc = {'cookies': 10, 'ham': 45, 'eggs': 5, 'apples': 8}
         inventory_dicc['bananas'] = 16
         print( inventory_dicc )

         # Inserint valors estructurats (diccionaris)
         soccer_dicc = {'van Basten': {'name': 'Marco', 'age': 50}}
         soccer_dicc['Gullit'] = {'name': 'Ruud', 'age': 49}
         print( soccer_dicc )
```

```
{'apples': 8, 'bananas': 16, 'cookies': 10, 'eggs': 5, 'ham': 45}
{'van Basten': {'name': 'Marco', 'age': 50}, 'Gullit': {'name': 'Ruud', 'age': 49}}
```

I es poden esborrar amb la sentència **del**:

```
In [63]: del inventory_dicc['eggs']
         print( inventory_dicc )
```

```
{'apples': 8, 'bananas': 16, 'cookies': 10, 'ham': 45}
```

Els diccionaris també disposen de mètodes addicionals de manipulació, alguns similars als que hem vist previament en l'apartat de llistes:

Mètode	Descripció
`get(clau)`	retorna el valor associat a la clau donada
`update(clau_valor_parelles)`	afegeix les parelles corresponents al diccionari
`pop(clau)`	elimina i retorna el valor associat a la clau
`popitem()`	esborra/retorna una parella (clau, valor) arbitraria
`clear()`	esborra tots els elements

Tots aquests mètodes cal aplicar-los sobre un diccionari determinat: <nom_diccionari>**.mètode()**

```
In [64]: #Mostrem el contingut del diccionari
         inventory_dicc = {'cookies': 10, 'ham': 45, 'eggs': 5, 'apples': 8}
         print('Valors del dicc: ', inventory_dicc )

         #Valor associat a una clau existent
         print("El valor de 'apples' és {}".format(inventory_dicc.get('apples')) )

         #Valor associat a una clau inexistent
         print("El valor de 'lemon' és {}".format(inventory_dicc.get('lemon')) )

         #Treiem la parella determinada per la clau 'apples'
         ap_val = inventory_dicc.pop('apples')
         print("El valor de 'apples' és {} i el cont. del dicc és {}\n".format(ap_val, inventory_dicc))

         #Afegim noves parelles
         inventory_dicc.update( {'strawberry': 90, 'artichoke': 7} )
         print('Valors del dicc:', inventory_dicc )

         #Treiem qualsevol parella
         rand_val = inventory_dicc.popitem()
         text = "El valor de la parella és {}\n\t i el cont. del dicc és {}"
         print(text.format(rand_val, inventory_dicc))
         inventory_dicc.clear()
         print("Contingut després d'un clear: {}".format(inventory_dicc))

Valors del dicc:  {'apples': 8, 'cookies': 10, 'eggs': 5, 'ham': 45}
El valor de 'apples' és 8
El valor de 'lemon' és None
El valor de 'apples' és 8 i el cont. del dicc és {'cookies': 10, 'eggs': 5, 'ham': 45}

Valors del dicc: {'strawberry': 90, 'cookies': 10, 'artichoke': 7, 'eggs': 5, 'ham': 45}
El valor de la parella és ('strawberry', 90)
         i el cont. del dicc és {'cookies': 10, 'artichoke': 7, 'eggs': 5, 'ham': 45}
Contingut després d'un clear: {}
```

1.4.5 Modificar elements d'un diccionari

Al tractar-se d'un tipus de dada mutable podem modificar el seu contingut directament amb l'operador []:

```
In [65]: inventory_dicc = {'cookies': 10, 'ham': 45, 'eggs': 5, 'apples': 8}
         inventory_dicc['ham'] = 220
         print( inventory_dicc )

         basket_dicc['Bird'] = {'name': 'Larry', 'age': 66}
         print( basket_dicc )

         inventory_dicc['ham'] += 220
         print( inventory_dicc )
```

```
{'apples': 8, 'cookies': 10, 'eggs': 5, 'ham': 220}
{'Shaquile': {'name': 'ONeal', 'age': 35}, 'Jordan': {'name': 'Michael', 'age': 40}, 'Bird': {'name': 'I
{'apples': 8, 'cookies': 10, 'eggs': 5, 'ham': 440}
```

1.4.6 Separació de claus i valors

En cas que sigui necessari, les claus i els valors d'un diccionari es poden extreure com a llistes:

```
In [66]: # Obtenim les claus i valors separadament com a llistes
         print ( list(inventory_dicc.keys()) )
         print ( list(inventory_dicc.values()) )
```

```
['apples', 'cookies', 'eggs', 'ham']
[8, 10, 5, 440]
```

1.5 Set (conjunt)

Es similar a una llista, però només pot emmagatzemar elements únics (no hi pot haver repeticions). Els elements no tenen cap ordenació associada.

```
In [67]: animals_set = {'lion', 'elephant', 'tiger', 'lion'}

         print(animals_set)
```

```
{'elephant', 'tiger', 'lion'}
```

Els conjunts s'utilitzen de forma similar als altres conjunts de dades que hem estudiat en aquest tema.

1.5.1 Operacions amb conjunts

`add(element)`
Afegeix un element, que ha de ser inmutable, al conjunt.

```
In [68]: colours = {"red","green"}
         colours.add("yellow")
         colours
```

```
Out[68]: {'green', 'red', 'yellow'}
```

`clear()`
Esborra tots els elements d'un conjunt.

```
In [69]: colours.clear()
         colours
```

```
Out[69]: set()
```

`copy()`
Crea un copia complerat del conjunt i la retorna:

```
In [70]: cities = {"Stuttgart", "Konstanz", "Freiburg"}
         cities_backup = cities.copy()
         cities.clear()
         cities_backup

Out[70]: {'Freiburg', 'Konstanz', 'Stuttgart'}
```

Si la mateixa operació es fa només amb una assignació, les dades es perden.

```
In [71]: cities = {"Stuttgart", "Konstanz", "Freiburg"}
         cities_backup = cities  # Just an assigment
         cities.clear()
         cities_backup

Out[71]: set()
```

`difference(set)`
Retorna la diferencia entre dos o mes conjunts:

```
In [72]: x = {"a","b","c","d","e"}
         y = {"b","c"}
         z = {"c","d"}
         x.difference(y)

Out[72]: {'a', 'd', 'e'}

In [73]: x.difference(y).difference(z)

Out[73]: {'a', 'e'}
```

Also the minus operator (-) can be used:

```
In [74]: x - y

Out[74]: {'a', 'd', 'e'}

In [75]: x - y - z

Out[75]: {'a', 'e'}
```

`difference_update(set)`
Esborra tots els elements d'un altre conjunt d'aquest:

```
In [76]: x = {"a","b","c","d","e"}
         y = {"b","c"}
         x.difference_update(y)
         x

Out[76]: {'a', 'd', 'e'}
```

Es la mateixa operació que amb els operadors "x = x - y":

```
In [77]: x = {"a","b","c","d","e"}
         y = {"b","c"}
         x = x - y
         x
```

```
Out[77]: {'a', 'd', 'e'}
```

`discard(element)`
Esborra un element del conjunt. Si l'element no pertany al conjunt no es fa res.

```
In [78]: x = {"a","b","c","d","e"}
         x.discard("a")
         x

Out[78]: {'b', 'c', 'd', 'e'}
```

`remove(element)`
Fa el mateix que `discard`, però en aquest cas es llença un KeyError si l'element no hi es present:

```
In [79]: x = {"a","b","c","d","e"}
         x.remove("a")
         x

Out[79]: {'b', 'c', 'd', 'e'}

In [80]: x.remove("z")
         x


         ---------------------------------------------------------------------------

         KeyError                                  Traceback (most recent call last)

         <ipython-input-80-0bde869d614a> in <module>()
     ----> 1 x.remove("z")
           2 x


         KeyError: 'z'
```

`intersection(set)`
Retorn la intersecció de dos conjunts en un de nou, si els dos conjunts tenen elements en comú.

```
In [81]: x = {"a","b","c","d","e"}
         y = {"c","d","e","f","g"}
         x.intersection(y)

Out[81]: {'c', 'd', 'e'}
```

L'operador (&) també es pot fer servir:

```
In [82]: x = {"a","b","c","d","e"}
         y = {"c","d","e","f","g"}
         x & y

Out[82]: {'c', 'd', 'e'}
```

`pop()`
Es treu un element arbitrari del conjunt. Si el conjunt es buit, es llenças un error KeyError.

```
In [83]: x = {"a","b","c","d","e"}
         x.pop()
```

```
Out[83]: 'c'
```

`isdisjoint(set)`
Retorna cert quan la intersecció entre dos conjunts es nula:

```
In [84]: x = {"a","b","c","d","e"}
         y = {"c","d","e","f","g"}
         x & y

Out[84]: {'c', 'd', 'e'}

In [85]: x.isdisjoint(y)

Out[85]: False

In [86]: z = x - y
         z.isdisjoint(y)

Out[86]: True
```

`issubset(set)` and `issuperset(set)`
Comproven si els conjunts son un subconjunts (subset) o un superconjunts (superset) d'un altre. També es poden fer servir els operadors relacionals:

```
In [87]: x = {"a","b","c","d","e"}
         y = {"c","d"}
         print("x subset of y: ", x.issubset(y))
         print("y subset of x: ", y.issubset(x))
         print("x > y: ", x > y)
         print("x >= y: ", x >= y)
         print("x > x: ", x > x)
         print("x >= x: ", x >= x)
         print("----------------")
         print("x superset of y: ", x.issuperset(y))
         print("y superset of x: ", y.issuperset(x))
         print("x < y: ", x < y)
         print("x <= y: ", x <= y)
         print("x < x: ", x < x)
         print("x <= x: ", x <= x)
```

```
x subset of y:  False
y subset of x:  True
x > y:  True
x >= y:  True
x > x:  False
x >= x:  True
----------------
x superset of y:  True
y superset of x:  False
x < y:  False
x <= y:  False
x < x:  False
x <= x:  True
```

1.6 Python Collections

Com a resum del tema, tot seguit es mostra de forma tabulada les característiques principals dels diferents tipus de dades estructurades que suporta Python. S'ha inclós també el tipus **string** ja vist en temes anteriors:

Name	Type	Empty	Iterable	Mutable	Indexed	Unique	Homogeneous	Named
string	`str`	”	Yes	No	Yes	No	Yes	No
list	`list`	[]	Yes	Yes	Yes	No	No	No
tuple	`tuple`	()	Yes	No	Yes	No	No	No/Yes
dictionary	`dict`	**{}**	*Yes*	Yes	No	No	No	Yes
set	`set`	**set**()	Yes	Yes	No	Yes	No	No

Tot aquell lector que vulgui aprofundir en algún dels tipus anteriors, pot adreçar-se a la documentació oficial de Python 3.4.