Referencia rápida 2 (MatUAM)
Estructuras de datos

1. Una **lista** es una colección ordenada de datos no necesariamente del mismo tipo. Se puede construir
   - enumerando sus elementos separados por comas y entre corchetes cuadrados, [ ]:

```
L=[1,'a',4,3+2]; type(L)        <type 'list'>
L                          ⟼    [1, 'a', 4, 5]
L0=[]                           []
```

   - con el constructor **list**(), que toma como argumento una estructura de datos y construye la lista con los elementos de dicha estructura (ver, por ejemplo, cualquiera de las que aparecen en esta hoja)

```
L=list(); L
L1=list((1,2,4,8)); L1   ⟼   []
                             [1, 2, 4, 8]
```

2. **Listas predeterminadas**.

```
[1..7]          [1,2,3,4,5,6,7]
range(7)        [0,1,2,3,4,5,6]
range(3,8)  ⟼   [3,4,5,6,7]
range(1,11,3)   [1,4,7,10]
range(11,5,−1)  [11,10,9,8,7,6]
```

   y las versiones **srange**(), de enteros de Sagemath, y los *generadores* **xrange**() y **xsrange**() que nos van dando acceso a sus enteros uno a uno, pero sin guardarlos en memoria (cf. apartado 12).
3. Si la secuencia de elementos de la lista sigue cierto *patrón*, podemos generarla por **comprensión** (*comprehension list*)

```
cuadrados=[j**2 for j in xrange(11)]; cuadrados  ⟼  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

   En esta sintaxis vemos el uso de la estructura **for** elemento **in** contenedor a la que dedicaremos otra referencia rápida específica. De momento basta saber que realiza una asignación por cada elemento del contenedor, recorriendo este desde el primero al último. Además se puede *filtrar*, con expresiones **if**, para no tener que tomar todos los elementos sino solo aquellos que verifiquen cierta condición. Por ejemplo

```
cuadradosPrimos=[j**2 for j in srange(11) if j.is_prime()]; cuadradosPrimos  ⟼  [4, 9, 25, 49]
```

4. La partícula **in** nos permite averiguar si un elemento está en la lista,

```
1015 in [13*j+1 for j in xrange(80)]  ⟼  True
```

   y el comando **len**() devuelve el número de elementos de la lista,

```
len([j for j in srange(100) if (j^2+1).is_prime()])  ⟼  19
```

5. El *índice* de cada elemento en la lista indica su posición, con 0 el de la primera posición. Se pueden utilizar índices negativos, que indican las posiciones de derecha a izquierda, −1 es la última. Podemos extraer uno o una sublista de la lista con la notación *slice*:
   - [a] para el elemento de índice a;
   - [a:b] la sublista de b−a elementos consecutivos, desde el de índice a al del índice anterior a b. Se puede abreviar [0:b] por [:b]. De igual manera [a:] es una abreviatura si b coincide con la longitud completa de la lista, en particular [:] toma todos los elementos;
   - [a:b:c] para la sublista con los elementos de índices a hasta el del índice anterior a b, pero tomados de c en c.

```
L=range(2,100,3); L[0]      2
L[2:7]                ⟼    [8, 11, 14, 17, 20]
L[1:26:3]                   [5, 14, 23, 32, 41, 50, 59, 68, 77]
L[−1:−7:−1]                 [98, 95, 92, 89, 86, 83]
```

6. **Operaciones con listas y transformaciones:** Si L1, L2 son listas, dat un dato y k un entero:
   - L1+L2 (concatenación) devuelve la lista con los datos de ambas, en el orden dado;
   - k*L1 devuelve la lista resultado de concatenar k veces la lista L1;
   - L1.**count**(dat), contabiliza el número de apariciones de dat;
   - L1.**index**(dat), que devuelve el índice de la primera aparición de dat;
   - L1.**append**(dat) añade dat al final de la lista L1;
   - L1.**remove**(dat) elimina la primera aparición de dat en L1. Genera un error si dat no está en L1;
   - L1.**extend**(contenedor) amplía la lista L1 añadiendo, uno a uno, cada elemento del contenedor;
   - L1.**insert**(k,dat), cambia L[k] por dat, equivale a L[k]=dat;
   - L1.**pop**(k), devuelve y quita el elemento con índice k en la lista. Si no se especifica, se considera k=−1;
   - L.**reverse**() da la vuelta a la lista;
   - L.**sort**(), ordena los elementos de la lista, de menor a mayor. Con L.**sort**(**reverse**=True), lo hace de mayor a menor.

   Los métodos que transforman listas, salvo .**pop**(), no devuelven nada al evaluarse. En particular, una asignación como M=L.**reverse**() no asigna una lista a M, simplemente modifica L:

```
L=[1,2,4,6]; M=L.reverse(); L    [6, 4, 2, 1]
M
type(M)                     ⟼    <type 'NoneType'>
MM=L.pop(); MM                   1
L                                [6, 4, 2]
```

   Obsérvese que el efecto de la asignación L1=L es diferente a las asignaciones L2=**list**(L) o L3=**copy**(L). En el primer caso cualquier modificación sobre L modifica también L1 (y viceversa), pero no L2, ni L3. Si L=[1,2,3] :

```
L1, L2, L3= L, list(L), copy(L); L1, L2, L3   ⟼  ([1, 2, 3], [1, 2, 3], [1, 2, 3])
L[1]=5; L1, L2, L3                            ⟼  ([1, 5, 3], [1, 2, 3], [1, 2, 3])
L1[1]=10; L, L1                               ⟼  ([1, 10, 3], [1, 10, 3])
```

7. Otra estructura de datos es la **tupla**. La manera más sencilla de crear una tupla es poniendo entre paréntesis sus datos separados por comas. También se pueden generar con el constructor **tuple**(), al que se le indican los datos tomados de otra estructura, por ejemplo una lista u otra tupla.

   Con **in** se puede ver si un dato está en una tupla. Las tuplas no se pueden concatenar, aunque sí crear una tupla repitiendo k veces otra. Se accede a los elementos de la tupla con la notación *slice* de índices. Los métodos .**count**() e .**index**() antes comentados son aplicables a las tuplas.

   A diferencia de las listas los datos de una tupla no se pueden modificar. Así si t=(1,2,3) :

```
4 in t                                  False
t[0], t[−2], t[:2]                      (1, 1, (1, 2))
t1, t2, t3= t, tuple(t), copy(t); t, t1, t2, t3   ⟼  ((1, 2, 3), (1, 2, 3), (1, 2, 3))
t=(4,3,2,1); t, t1, t2, t3              ((4, 3, 2, 1), (1, 2, 3), (1, 2, 3), (1, 2, 3))
(3*t).count(2)                          3
t.index(3)                              1
```

8. Cualquier secuencia de caracteres entrecomillados, con comillas simples, dobles o triples, es una cadena de caracteres, <type 'str'>.

   La operación `+` concatena cadenas de caracteres, y con `*` y un natural `k`, se concatena `k` veces la misma cadena: `2*'x'+'v'+3*'i'` ⟼ `'xxviii'`

   El constructor `str()` devuelve una cadena de caracteres, una versión literal, del contenido entre paréntesis una vez evaluado o interpretado:

```
str(2+3*8)      26
'2'+3*'8'       2888
'2'+'3*8'   ⟼   23*8
str(2+3)*8      55555555
```

9. Aparte de los métodos .**index**() y .**count**(), las cadenas tienen otros métodos propios. Como ocurre con las tuplas, las cadenas de caracteres no se modifican. Enumeramos algunos métodos, y sus efectos tomando romanos=`'xxviii es 28'` :
   - .**capitalize**(), devuelve otra cadena resultado de cambiar el primer carácter de la cadena sobre la que se aplica por el mismo pero en mayúsculas si el carácter lo admite

```
romanos.capitalize()   ⟼   'Xxviii es 28'
romanos                    'xxviii es 28'
```

   - .**upper**(), produce otra cadena al cambiar todos los caracteres que lo admitan por el mismo en mayúsculas

```
romanos.upper()  ⟼  'XXVIII ES 28'
```

   - .**lower**(), da la cadena resultado de cambiar todos los caracteres que lo admitan por el mismo en minúsculas.

- **.replace**(cadA,cadB), sustituye en la cadena cada aparición de la subcadena cadA por cadB.

```
romanos.replace('i','j') ⟼ 'xxvjjj es 28'
```

- **.join**(cad), genera una cadena pegando, con la cadena sobre la que se aplica el método, las cadenas del contenedor de cadenas cad.

```
'-'.join(romanos)                  'x-x-v-i-i-i- -e-s- -2-8'
L=list(romanos); L.reverse()   ⟼
''.join(L)                         '82 se iiivxx'
```

- **.find**(cad), actúa como **.index**(cad), pero si cad no aparece en la cadena a la que se aplica el método, en lugar de dar un error como **.index**(), devuelve -1.

```
romanos.find('i')   ⟼  3
romanos.find('j')   ⟼  -1
```

- **.split**(sep), devuelve una lista, con las subcadenas que resultan de *trocear* la cadena original por cada aparición de la (posible) subcadena sep

```
romanos.split('')   ⟼  ['xxviii', 'es', '28']
romanos.split('j')  ⟼  ['xxviii es 28']
```

10. Cada elemento de los contenedores anteriores tiene asociado un índice que indica su posición en el mismo. Un **conjunto** (**set**), en sintonía con su nombre, es una estructura de datos sin esta cualidad, lo que ahorra espacio de memoria.

    Se construyen enumerando sus elementos entre llaves { } o con el constructor **set**(), tomando elementos de algún contenedor:

```
A={1,2,'a'}; type(A)  ⟼  <type 'set'>
A                        {1,2,'a'}
B=set(range(4)); B       {0, 1, 2, 3}
```

    Listas y conjuntos no pueden ser elementos de un conjunto. Existe, no obstante, el *generador* **powerset**() (o **subsets**()), que se puede recorrer con un **for**:

```
[S for S in powerset(set('ab'))]   ⟼  [[],  ['a'],  ['b'],  ['a',  'b']]
len([S for S in powerset({})])     ⟼  1
```

    De un conjunto podemos saber su cardinal (**len**(conjunto)), si un elemento pertenece a él (elemento **in** conjunto) y podemos realizar las operaciones habituales de conjuntos:

```
A, B={1,2,'a'}, set('abcd')
A.intersection(B)                  {'a'}
A.union(B)                         {'a', 'b', 'c', 'd', 1, 2}
A.difference(B)                    {1, 2}
A.symmetric_difference(B)      ⟼   {'b', 'c', 'd', 1, 2}
A.add('aa'); A                     {'a', 'aa', 1, 2}
A.update({3,4}); A                 {'a', 'aa', 1, 2, 3, 4}
A.isdisjoint(B)                    False
set('ac').issubset(B)              True
B.issuperset(set('ad'))            True
```

11. Un **diccionario** es una colección de pares `clave:valor`. Si una lista es una colección de objetos indexada por números enteros consecutivos, un diccionario permite como `clave` cualquier tipo de datos (que no se puedan modificar), y los `valores` pueden ser totalmente arbitrarios.

    Se utilizan llaves para delimitar la colección de pares del diccionario y dos puntos para separar cada clave de su valor:

```
variedades={'pera':['de agua', 'limonera', 'de San Juan'],'manzana':['Fuji',  'golden', 'reineta']}
```

    El constructor **dict**(), con una lista, tupla o conjunto de parejas, por ejemplo, genera un diccionario:

```
puntos=dict({(k,k**2) for k in [3..5]}); type(puntos)  ⟼  <type 'dict'>
puntos                                                    {3: 9, 4: 16, 5: 25}
```

    Si L1, L2 son dos listas, con **len**(L1)<=**len**(L1) ⟼ True, **dict**(**zip**(L1,L2)) genera un diccionario con claves los elementos de L1, y se asigna a cada clave L1[k] el valor L2[k]. Si k aparece repetido entre las claves, solo sobrevive la última aparición.

```
L1, L2=[1,1,2], list('abcd');   len(L1)<=len(L2)  ⟼  True
F=dict(zip(L1,L2)); F                             ⟼  {1: 'b', 2: 'c'}
```

    Una vez creado un diccionario:

- la particula **in** nos sirve para comprobar si cierto dato es una `clave` del diccionario:

```
'banana' in variedades  ⟼  False
```

- los métodos .**keys**(), .**values**() e .**items**() aplicados al diccionario, devuelven sendas listas de claves, valores y pares (clave,valor), respectivamente:

```
F.keys()                [1, 2]
F.values()      ⟼      ['b', 'c']
F.items()               [(1, 'b'), (2, 'c')]
```

- se accede al valor de cada clave utilizando esta entre corchetes [ ], a modo de índice:

```
variedades['pera']  ⟼  ['de agua', 'limonera', 'de San Juan']
```

- se añaden entradas al diccionario con asignación directa o con el método **.update**() usado con cualquier estructura con parejas (clave,valor):

```
puntos[7]=8
puntos.update([(k,k+1) for k in [9..11]])  ⟼  {3: 9, 4: 16, 5: 25, 7: 8, 9: 10, 10: 11, 11: 12}
puntos
```

- se eliminan entradas con **del**(), o simplemente **del**:

```
del(puntos[3]); del puntos[4]; puntos  ⟼  {5: 25, 7: 8, 9: 10, 10: 11, 11: 12}
```

- se sustituyen valores reasignando o modificando el valor:

```
puntos[7]=10; puntos                 {5: 25, 7: 10, 9: 10, 10: 11, 11: 12}
variedades['pera'].pop()       ⟼    'de San Juan'
variedades['pera']                   ['de agua', 'limonera']
```

---

12. Una estructura bastante particular es la de **generador**, de la que ya nos ha aparecido algún ejemplo (**xrange**, **xsrange** o **powerset**). Al construir un generador el código no se ejecuta completamente de manera que no se almacenan los potenciales elementos, lo que supone un ahorro de memoria. El cálculo de cada elemento queda pendiente, a la espera de la petición de uno nuevo (evaluación *perezosa*). Se puede solicitar un nuevo elemento con el método **.next**(), o recorrer al completo el generador con **for** elemento **in** generador .

    Una manera rápida de crear nuevos generadores es utilizando otro generador ya existente, y la notación por comprensión con paréntesis

    (expresion **for** elemento **in** generador **if** condicion)

    donde el filtro **if** es optativo:

```
G=(j**2 for j in xsrange(1,10^6) if j.is_prime()); type(G)  ⟼  <type 'generator'>
```

    Recién creado este generador:

```
G.next()
[G.next() for j in xrange(4)]  ⟼  4
                                  [9, 25, 49, 121]
```

    Obsérvese que no tiene sentido utilizar **len**(generador) para averiguar su cardinal, es decir cuántos elementos podría generar. Algunos generadores se construyen con un método para este cometido:

```
A=set('abxy'); A                                  {'a', 'b', 'x', 'y'}
Subsets(A).cardinality()                          64
{B for B in Subsets(A) if len(B)<=1}        ⟼    {{}, {'a'}, {'x'}, {'b'}, {'y'}}
len({X for X in Subsets(A) if len(X)==2})         6
```