.

Ampliació d'Informàtica
Apunts d'iteradors

Maria Dolors Ayala Vallespí
Dept. Ciències de la computació
E.T.S.E.I.B.
Universitat Politècnica de Catalunya

Tardor 2023

Contents

1 Iteradors

1.1 Introducció

1.1.1 Iterar

Iterar és repetir una acció més d'una vegada. Tant la sentència `for` com la `while` permeten iterar. Exemples:

```
def f1 (llista):
    suma = 0
    for e in llista:
        suma = suma + e
    return suma

def f2 (llista):
    suma = 0
    for i in range(len(llista)):
        suma = suma + llista[i]
    return suma

def f3 (llista):
    suma = 0
    i = 0
    while i < len(llista):
        suma = suma + llista[i]
        i = i + 1
    return suma
```

Les tres funcions anteriors realitzen la mateixa acció, que és equivalent a aplicar la funció `sum`. `f1` usa una sentència `for` que itera directament sobre els elements de la llista. `f2` usa una sentència `for` que itera sobre els enters `0, 1, ...len(llista)-1` que són els índexs dels elements de la llista. `f3` usa una sentència `while`.

1.1.2 Objecte iterable

Un objecte **contenidor** és un tipus d'objecte corresponent a una col·lecció d'objectes elementals. Els strings, llistes, tuples i diccionaris són objectes contenidors. Tots ells es poder recórrer amb la sentència `for`.

Un objecte **iterable** és un objecte contenidor capaç de retornar els seus elements d'un en un. Tots els objectes de tipus seqüencial són iterables: strings, llistes i tuples. També ho són alguns tipus no seqüencials com els diccionaris.

Un objecte iterable es pot iterar múltiples vegades mentre que un objecte iterador només es pot iterar una vegada: els iteradors s'esgoten. En realitat un objecte iterable produeix un iterador cada cop que algun mecanisme itera sobre els seus elements. Els objectes de tipus `file` són iteradors.

Les funcions següents estan predefinides sobre iterables: `all`, `any`, `enumerate`, `filter`, `map`, `max`, `min`, `sorted`, `sum`, `str.join` i `zip`.

Es disposa dels tipus predefinits iterables: `dict`, `list`, `set`, `str`, `tuple` i `range`.

1.2 Objecte iterador

Un **iterador** és un objecte que proporciona un mecanisme per recórrer la seqüència d'elements d'un objecte iterable.

Un objecte iterable es pot iterar múltiples vegades mentre que un objecte iterador només es pot iterar una vegada: els iteradors s'esgoten. En realitat un objecte iterable produeix un iterador cada cop que algun mecanisme itera sobre els seus elements.

Amb els iteradors cada element es processa quan es necessita i no cal tenir-los tots emmagatzemats en una estructura: és el concepte de seqüència.

Els tipus `str, list, tuple, dict, set` són iterables. Algunes funcions i mètodes retornen una classe que també és iterable: per exemple la funció `range` i els mètodes `keys`, `values` i `items` dels diccionaris.

Els iterables tenen el mètode `__iter__` que retorna un iterador. Els iteradors tenen el mètode `__next__` que retorna el següent element de l'iterador i aixeca l'excepció `StopIteration` quan aquest s'acaba.

Les funcions `iter` i `next` sobrecarreguen els dos mètodes indicats.

En general no s'usen els mètodes `__iter__` i `__next__` directament sinó indirectament a través del `for`. A continuació es mostra com és l'execució d'una sentència `for` com la següent:

```
>>> llista = [1, 2, 3, 4]
>>> for e in llista:
...     print(e)
```

1. crida la funció `iter` per l'objecte contenidor, `it = iter(llista)`, que retorna un objecte iterador, `it`, que té definida la funció `next` la qual permet accedir als elements del contenidor d'un en un.
2. a cada iteració, obté l'element següent aplicant la funció `next` a l'iterador obtingut: `e = next(it)`
3. a la iteració cinquena, l'acció `e = next(it)` produeix una excepció `StopIteration` la qual indica a la sentència `for` que s'han acabat els elements.

És a dir, seria equivalent al codi següent:

```
>>> llista = [1, 2, 3, 4]
>>> it = iter(llista)
>>> while True:
...     try:
...         e = next(it)
...         print(e)
...     except StopIteration:
...         break
```

Es pot alternar la sentència `for` i la funció `next`. Exemple:

```
>>> b = [4, 6, 3, 2, 9]
>>> it = iter(b)
>>> next(it)
4
>>> next(it)
6
>>> for a in it:
...     print(a)
...
3
2
9
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

El mateix es pot fer amb les funcions que retornen una classe iterable. Per exemple, la funció `range` crea un objecte de la classe `range` i la sentència `for e in range(n)` crea un iterador a partir d'aquest objecte.

Els iteradors només permeten fer un sol recorregut de les dades i en una direcció. Si hem de fer diversos processos amb un iterador, caldrà generar-lo tantes vegades com sigui necessari.

Un cop s'ha definit un objecte iterador, li podem aplicar totes les funcions de la llibreria predefinida que requereixen un iterable.

1.3 Generadors

Un generador (funció generadora) és una funció que crea i retorna un iterador, al qual també s'anomena generador. Els elements de la seqüència corresponent a l'iterador es generen i retornen amb la sentència `yield`.

En una funció generadora NO hi ha la sentència `return` com a les funcions que hem vist fins ara, sentència que s'executa una i només una vegada. En canvi, hi ha la sentència `yield` que es pot executar més d'una vegada, tal com es descriu al paràgraf anterior.

La sentència `yield`, com la sentència `return`, porta associada una expressió.

Les funcions generadores s'executen d'una forma especial. La funció generadora comença a executar-se quan hi ha una crida a la funció `next` aplicada a l'iterador generat per la funció generadora i s'atura quan troba una sentència `yield`. Aleshores, la funció `next` retorna el valor de l'expressió associada a la sentència `yield`. A continuació, s'executen les sentències que segueixen a la sentència `yield` fins a trobar la següent sentència `yield`, en la que s'atura fins que hi ha una altra crida a la funció `next`, i així successivament. Els valors de les variables de la funció generadora es conserven entre crides successives a la funció `next` (que van associades a successives execucions de la sentència `yield`.

Si cal aturar les iteracions abans d'arribar al final, es pot posar la sentència `return`. En aquest cas, la crida a `next` donarà `StopIteration`. Ara bé, un generador no pot retornar res usant la sentència `return`, només pot retornar valors amb la sentència `yield`.

Als objectes generadors els hi podem aplicar totes les funcions de la llibreria predefinida que

apliquen a un iterable: `sum`, `min`, `list`, `tuple`, `sorted`, etc.

Exemple 1: gen1 és una funció generadora

```
def gen1 ():
    yield 'joans'
    yield 'joseps'
    yield 'ases'
    yield "n'hi ha a totes les cases"

>>> a = gen1()  # a és un iterador
>>> next(a)
'joans'
>>> next(a)
'joseps'
>>> next(a)
'ases'
>>> next(a)
"n'hi ha a totes les cases"
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Exemple 2: genera els múltiples de `a` inferiors a `b`

```
def mult (a, b):
    x = a
    while x < b:
        yield x
        x = x + a

>>> it = mult(3, 15)
>>> next(it)
3
>>> next(it)
6
>>> next(it)
9
>>> next(it)
12
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(mult(3, 15)) # retorna el mateix perquè hem tornat a generar l'
    iterador
3

>>> for e in mult(3, 15): # tornem a generar l'iterador
...     print(e)
...
3
6
9
12
>>> sum(mult(3, 15))  # tornem a generar l'iterador
30
```

```
>>> it = mult(3, 15)
>>> max(it)
12
>>> min(it)   # l'iterador s'ha esgotat !!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: min() arg is an empty sequence
>>> it = mult(3, 15)  # tornem a generar l'iterador
>>> min(it)
3
```

Exemple 3: genera infinites potències de 2

```
def pot2():
    p = 1
    while True:    # bucle infinit !
        yield p
        p = p * 2

>>> it = pot2()
>>> next(a)
1
>>> next(a)
2
>>> next(a)
4
......          # podriem fer infinites crides a la funció next !
>>> max(po2())  # es queda en un bucle infinit !
```

1.4 Iteradors i seqüències

Les funcions generadores es relacionen amb el concepte de seqüència vista al curs anterior. Una funció generadora retorna un iterador i els elements d'aquest iterador són els de la corresponent seqüència, successió o sèrie.

La resolució dels problemes amb seqüències es poden dur a terme encadenant processos (funcions) a iteradors. Per això es disposa de:

- funcions creades explícitament i funcions anònimes (`lambda`)
- funcions de la llibreria predefinida o estàndard
- funcions del mòdul `itertools`
- funcions del mòdul `operator`
- funcions del mòdul `functools`

La metodologia alternativa de resolució de problemes encadenant processos es mostra a continuació resolent l'exemple Mostreig funció

Abans, però veurem què són les funcions lambda, ja que les usarem a la resta d'aquest capítol.

Funcions anònimes o lambda

Les funcions anònimes o `lambda` són funcions molt simples que avaluen una sola expressió i no tenen nom. La sintaxi de la seva definició requereix la paraula reservada `lambda` seguida dels paràmetres, després hi va el caràcter dos punts (`:`) seguit de l'expressió que retorna. Exemples:

```
>>> f1 = lambda a, b: a+b  # retorna la suma dels dos paràmetres
>>> f1(4, 8)
12
>>> f2 = lambda n: 2**n    # retorna l'enèsima potència de 2
>>> f2(4)
16
```

També es poden definir funcions lambda que incloguin la sentència condicional `if`. Aquesta ha de ser simple i amb una sola sentència a cada branca. La sintaxi que ens permet escriure una sentència condicional en una línia és la següent:

`lambda` paràmetres: sentència1 `if` exp_bool `else` sentència2

i equival a:

```
if exp_bool:
    sentència1
else:
    sentència2
```

Exemple:

```
>>> f3 = lambda op, x, y: x+y if op == 0 else x*y
>>> f3(0, 2, 3)
5
>>> f3(1, 2, 3)
6
```

L'anterior funció `lambda` és equivalent a la funció tradicional:

```
def f3(op, x, y):
    if op == 0:
        return x +y
    else:
        return x*y
```

Ara ja podem continuar amb l'exemple.

Enunciat

Per mostrejar una funció f dins l'interval $[a, b]$ i amb increments de d, hem de calcular valors successius de $f(x)$, començant a partir de $x = a$ amb increments uniformes de d fins a arribar a un valor superior a b. Els valors a, b i d poden ser reals. Podem suposar que $a < b$ i $d < b - a$

Dissenya la funció `positius(a, b, d, k)` que retorna la mitjana aritmètica dels valors mostrejats de la funció $f(x, k) = e^{x/k} - ksin(x)$ amb $k > 0$ que siguin positius. Si cap del valors mostrejats és positiu, la funció ha de retornar 0.

Solució clàssica

```
import math

def positius (a, b, d, k):
    x = a
    suma = 0
    n = 0
    while x <= b:
        y = math.exp (x/k)- k*math.sin(x)
        if y > 0:
            suma = suma + y
            n = n +1
        x = x + d
    if n != 0 :
        return suma / n
    else:
        return 0
```

En aquesta solució en un sol bucle es duen a terme diversos processos: mostrejar la funció, filtrar els punts que tenen l'ordenada positiva, comptar aquests punts i sumar aquests punts.

Solució basada en encadenar processos

Aquesta solució es basa en encadenar els esmentats processos usant les funcions `mostreja`, `filtra_positius`, `compta_elements` i `suma_elements`.

```
import math
import itertools

def positius_1(a, b, d, k):
    it_mostreig = mostreja(a, b, d, k)
    it_filtrat1 = filtra_positius (it_mostreig)
    it_mostreig = mostreja(a, b, d, k)
    it_filtrat2 = filtra_positius (it_mostreig)
    longitud = compta_elements(it_filtrat1)
    if longitud  == 0:
        return 0
    else:
        return suma_elements(it_filtrat2)/longitud

def positius_2(a, b, d, k):
    it_mostreig = mostreja(a, b, d, k)
    it_filtrat = filtra_positius (it_mostreig)
    it1, it2 = itertools.tee(it_filtrat, 2)
    longitud = compta_elements(it1)
    if longitud  == 0:
        return 0
    else:
        return suma_elements(it2)/longitud

def mostreja(a, b, d, k):
    x = a
    while x <= b:
        y = math.e ** (x/k) - k*math.sin(x)
        yield (x, y)
        x = x + d
```

```
def filtra_positius (it):
    for x, y in it:
        if y > 0:
            yield (x, y)

def compta_elements(it):
    n = 0
    for p in it:
        n = n + 1
    return n

def suma_elements(it):
    suma = 0
    for x, y in it:
        suma = suma + y
    return suma
```

La funció `mostreja` és una funció generadora que retorna un iterador format pels punts mostrejats (tuples) de la funció en l'interval i amb l'increment donats.

La funció `filtra_positius` és també una funció generadora que a partir d'un iterador en genera un altre. El procés que duu a terme és un filtrat dels punts que tenen l'ordenada positiva.

Les funcions `compta_elements` i `suma_elements` són funcions no generadores que a partir d'un paràmetre que és un iterador de punts retornen respectivament el nombre de punts que té i la suma de les ordenades d'aquests punts. Observem que totes dues funcions fan servir el mateix iterador que és el que ens ha donat la funció `filtra_positius` i recordem que els iteradors només es poden fer servir un sol cop, ja que es gasten. Com resolem aquest problema ?. Tenim dues opcions: o bé cridem les funcions `mostreja` i `filtra_positius` dos cops per tal de tenir dos iteradors iguals o bé fem servir una eina que ens aporta Python per repetir un iterador. Aquesta eina és la funció `tee` del mòdul `itertools`. Al codi hi ha les dues versions.

Si de tots els processos que necessitem aplicar per resoldre problemes n'haguéssim de fer el disseny de la corresponent funció, ja ens podríem quedar amb el disseny clàssic. Ara bé, els llenguatges de programació, i Python especialment, aporten moltes biblioteques amb moltes funcions que resolen processos bàsics i la metodologia proposada és usar aquestes funcions.

Solució basada en encadenar processos: usant eines existents

Aquesta solució s'ha fet després d'estudiar les funcions de les biblioteques que ens aporta Python, les quals es veuen amb més detall als apartats següents.

```
import math
import itertools

def positius(a, b, d, k):
    it_abcisses = mostreja_abs(a, b, d, k)
    it_ordenades = map (lambda x: math.e**(x/k) - k*math.sin(x), it_abcisses)
    it_filtrat = filter(lambda x: x > 0, it_ordenades)
    it1, it2 = itertools.tee(it_filtrat, 2)
    longitud = compta_elements(it1)
    if longitud  == 0:
        return 0
    else:
```

```
        return sum(it2)/longitud

def mostreja_abs(a, b, d, k):
    x = a
    while x <= b:
        yield x
        x = x + d

def compta_elements(it):
    n = 0
    for p in it:
        n = n + 1
    return n
```

En aquest disseny, el procés que duu a terme la funció `mostreja` s'ha partit en dos processos: el que obté les abcises i el que obté les ordenades.

La funció `mostreja_abs` és una funció generadora que retorna un iterador format per les abcises dels punts mostrejats de la funció en l'interval i amb l'increment donats.

Per obtenir les ordenades fem:

```
    it_ordenades = map (lambda x: math.e**(x/k) - k*math.sin(x), it_abcises)
```

En aquesta línia usem la funció `map` que és una funció de la llibreria predefinida. Aquesta funció també té dos paràmetres, `map(funció, iterable)`, i aplica la funció a tots els elements de l'iterable. Per tant retorna un iterable amb totes les ordenades de les abscisses que hi havia a `it_abscisses`, en el mateix ordre. La funció que aplica s'expressa com una funció `lambda`.

Ara hem de filtrar les ordenades positives:

```
    it_filtrat = filter(lambda x: x > 0, it_ordenades)
```

En aquesta línia usem la funció `filter` que és una funció de la llibreria predefinida. Aquesta funció també té dos paràmetres, `filter(funció, iterable)`, i filtra els elements de l'iterable `it_ordenades` de manera que l'iterable `it_filtrat` només conté les ordenades que són positives, en el mateix ordre.

Ara usem la funció `tee` del mòdul `itertools`, que permet repetir l'iterador dues vegades.

```
    it1, it2 = itertools.tee(it_ordenades_positives, 2)
```

Python no ofereix cap eina que permeti comptar els elements d'un iterador. Per tant hem de dissenyar la corresponent funció.

```
    longitud = compta_elements(it1)
```

En aquesta línia usem la funció `sum` que és una funció de la llibreria predefinida, que ja coneixem, i que podem usar pels iteradors.

```
    return sum(it2)/longitud
```

Ara que ja coneixem una mica totes aquestes eines, veiem que les podem combinar per comptar els elements d'un iterador. La següent expressió aplica un 1 per cada element de `it1` i després els suma: per tant, els compta:

```
     sum(map(lambda x: 1, it1))
```

1.5 Funcions de la llibreria predefinida

Nota prèvia: En aquest i altres apartats d'aquest capítol es mostra una introducció a funcions i classes de Python. És imprescindible consultar sempre la documentació de Python abans d'usar-les: allà hi trobarem l'especificació completa i actualitzada, així com altres elements que no s'introdueixen en aquest capítol.

Les funcions següents estan predefinides sobre iterables: `all`, `any`, `enumerate`, `filter`, `map`, `max`, `min`, `sorted`, `sum`, `str.join` i `zip`.

- `all`

  Retorna True si tots els elements de l'iterable són True

```
>>> a = [1, 2, 3, 4, 5]
>>> all(a)
True
>>> all(map(lambda x: x < 4, a))
False
>>> all(map(lambda x: x < 8, a))
True
```

- `any`

  Retorna True si algun dels elements de l'iterable és True

```
>>> it = iter([1, 2, 3, 4, 5])
>>> any(map(lambda x: x > 10, it))
False
>>> it = iter([1, 2, 3, 4, 5])
>>> any(map(lambda x: x < 2, it))
True
```

- `enumerate`

```
>>> vocals = ['a', 'e', 'i', 'o', 'u']
>>> it = enumerate(vocals)
>>> next(it)
(0, 'a')
>>> next(it)
(1, 'e')
>>> next(it)
(2, 'i')
>>> next(it)
(3, 'o')
>>> next(it)
(4, 'u')
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- `map(function, iterable, )` Retorna un iterador on cada element és el resultat d'aplicar la funció a l'iterable donat. Si hi ha més d'un iterador, la funció s'aplica a tots ells en paral.lel i s'atura quan s'acaba el més curt.

En aquest exemple i en els que segueixen, usem la funció list per mostrar els elements de l'iterador en una línia.

```
>>> list(map(lambda x: x*2, [24, 56, 76, 98]))
[48, 112, 152, 196]
>>> list(map(lambda x, y: x*y, 'PQRS', [2, 3, 4, 2]))
['PP', 'QQQ', 'RRRR', 'SS']
>>> list(map(lambda x, y: x+y, 'ABBBAA', 'AAABBB'))
['AA', 'BA', 'BA', 'BB', 'AB', 'AB']
```

En aquest exemple usem una funció completa (no lambda).

```
>>> def compara(x, y):
...     if x == y:
...        return x
...     else:
...        return '*'
...
>>> list(map(compara, 'ABBBAA', 'AAABBB')) # u
['A', '*', '*', 'B', '*', '*']
```

- `filter(function, iterable)`

  Retorna un iterador amb els elements de l'iterable que compleixen el predicat de la funció. La funció `itertools.filterfalse` és la complementària i retorna elements de l'iterable que no compleixen el predicat.

```
>>> list(filter(lambda x: x%2 == 0, [21, 56, 33, 76, 45, 98]))
[56, 76, 98]
>>> list(filter(lambda x: x !='*', 'ad***kjm**op'))
['a', 'd', 'k', 'j', 'm', 'o', 'p']
```

  En aquest exemple filtrem els strings que tenen tots dos caràcters iguals:

```
>>> la = ['AA', 'AB', 'BB', 'BC', 'XX', 'XY']
>>> list(filter(lambda x: x[0] == x[1], la))
['AA', 'BB', 'XX']
```

- `zip(iterables)`

  Retorna un iterador que agrega elements de cadascun dels iterables. S'atura quan s'acaba el més curt.

```
>>> list(zip('AEIOU', '123456789'))
[('A', '1'), ('E', '2'), ('I', '3'), ('O', '4'), ('U', '5')]
>>> lnoms = ['Mar', 'Iu', 'Pol', 'Ona']
>>> lc1 = ['Mill', 'Riu', 'Pons', 'Riba']
>>> lc2 = ['Pi', 'Pou', 'Bou', 'Mon']
>>> list(zip(lnoms, lc1, lc2))
[('Mar', 'Mill', 'Pi'), ('Iu', 'Riu', 'Pou'),
('Pol', 'Pons', 'Bou'), ('Ona', 'Riba', 'Mon')]
```

  En el següent exemple, després d'aplicar la funció `zip` que obté l'iterador de tuples d'strings anterior, a cadascun dels tuples s'aplica el mètode `join` amb un espai com a caràcter de juntura.

```
>>> it = map(' '.join, zip(lnoms, lc1, lc2))
>>> next(it)
'Mar Mill Pi'
>>> next(it)
'Iu Riu Pou'
>>> next(it)
'Pol Pons Bou'
>>> next(it)
'Ona Riba Mon'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

1.6 Funcions del mòdul itertools

El mòdul itertools implementa funcionalitats per crear iteradors.

1.6.1 Iteradors infinits

- `count(firstval[, step])`
  `count(10) --> 10 11 12 13 14 ...`
  `count(10, 3) --> 10 13 16 19 ...`
- `cycle('ABCD') --> A B C D A B C D ...`
- `repeat(object[, times])`
  `repeat(10) --> 10 10 10 10 10 ...`
  `repeat(10, 3) --> 10 10 10`

1.7 Iteradors finits

- `accumulate(iterable[, func])`,
  func: funció de 2 paràmetres, per operacions, usar mòdul operator
- `chain('ABC', 'DEF') --> A B C D E F`
  `chain('ABC', '123', 'JKL', '567') --> A B C 1 2 3 J K L 5 6 7`
- `compress(data, selector) --> (d[0] if s[0]), (d[1] if s[1]), ...`
  `compress('ABCDEF', [1, 0, 1, 0, 1, 1]) --> A C E F`
- `dropwhile(predicat,seq)` comença quan el predicat és False
  `dropwhile(lambda x:  x<5, [1,4,6,4,1]) --> 6 4 1`
- `filterfalse(predicat,seq)`
  elements en què predicat és False. Fa el contrari de la funció filter
  `filterfalse(lambda x:  x[0] == x[1], ['AA', 'AB', 'BB', 'BC', 'XX', 'XY']))`
  `--> 'AB', 'BC', 'XY'`

- `groupby(iterable, key = None)-->` (clau1, grup1), (clau2, grup2)
- `islice(seq, [start,] stop[, step]) -->` seq[start:stop:step]
  `islice('ABCDEFG', 2, None) --> C D E F G`
  `islice('ABCDEFGHIJK', 2, 9, 2) --> C E G I`
- `takewhile(predicat, iterable)` retorna elements mentre predicat és True
  `takewhile(lambda x:  x<5, [1,4,6,4,1]) --> 1 4`
- `tee(iterable[, n=2])` retorna n iteradors iguals i independents a partir d'un iterable.

1.7.1 Exemples de `accumulate`

```
>>> import itertools
>>> dades = [1, 2, 3, 4, 5]
>>> list(itertools.accumulate (dades))
[1, 3, 6, 10, 15]
>>> list(itertools.accumulate (['hola', 'que', 'tal']))
['hola', 'holaque', 'holaquetal']
>>> fm = lambda x, y: x//y
>>> list(itertools.accumulate ([1000, 4, 5, 10], fm))
[1000, 250, 50, 5]
\end{lstlisting
}
L'operació per defecte és la suma (concatenació si els elements són
strings). Per aplicar altres operacions, podem definir la corresponent
funció \texttt{lambda} o usar el mòdul \texttt{operator}, que es veurà
amb més detall al següent apartat.

\begin{lstlisting}
>>> import itertools
>>> import operator
>>> dades = [1, 2, 3, 4, 5]
>>> list(itertools.accumulate (dades, operator.mul))
[1, 2, 6, 24, 120]
>>> list(itertools.accumulate ([5, 4, 2, 3], operator.pow))
[5, 625, 390625, 59604644775390625]
>>> it = itertools.accumulate ([60, 23, 9, 3], operator.mod)
>>> next(it)
60          # 1r element
>>> next(it)
14          # resultat de 60%23
>>> next(it)
5           # resultat de 14%9
>>> next(it)
2           # resultat de 5%3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

1.7.2 Exemples de groupby

```
>>> a = groupby('AAAABBBCCD')
>>> for m, n in a:
...     m, list(n)
...
('A', ['A', 'A', 'A', 'A'])
('B', ['B', 'B', 'B'])
('C', ['C', 'C'])
('D', ['D'])

>>> a = groupby('AAAABBBCCD')
>>> for m, n in a:
...     m, len(list(n))
...
('A', 4)
('B', 3)
('C', 2)
('D', 1)
```

1.7.3 Exemples de tee

```
>>> import itertools
>>> a = iter(range(10))
>>> c, d=itertools.tee(a, 2)
>>> list(c)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(d)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(a)    # la funció tee esgota l'iterador
[]
```

1.8 Iteradors combinatoris

- `product(p, q, ... [repeat=1])` producte cartesià

```
>>> it = itertools.product('ABCD', [1, 2])
>>> for e in it:
...     print(e, end = ' ')
...
('A', 1) ('A', 2) ('B', 1) ('B', 2) ('C', 1) ('C', 2) ('D', 1) ('D', 2)

>>> it =itertools.product('ABCD', repeat =2)
>>> for e in it:
...     print(e, end = ' ')
...
('A', 'A') ('A', 'B') ('A', 'C') ('A', 'D') ('B', 'A')
('B', 'B') ('B', 'C') ('B', 'D') ('C', 'A') ('C', 'B')
('C', 'C') ('C', 'D') ('D', 'A') ('D', 'B') ('D', 'C')
('D', 'D')
```

- `permutations(p[, r])`. Variacions dels elements de `p` agafats de `r` en `r`. Són tuples de longitud `r`, en tots els ordres possibles, sense elements repetits.

```
>>> it =itertools.permutations('ABCD', 2)
>>> for e in it:
...     print(e, end = ' ')
...
('A', 'B') ('A', 'C') ('A', 'D') ('B', 'A') ('B', 'C')
('B', 'D') ('C', 'A') ('C', 'B') ('C', 'D') ('D', 'A')
('D', 'B') ('D', 'C')
```

- `combinations(p, r)`. Combinacions dels elements de `p` agafats de `r` en `r`. Són tuples de longitud `r`, ordenades, sense elements repetits.

```
>>> it =itertools.combinations('ABCD', 3)
>>> next(it3)
('A', 'B', 'C')
>>> next(it3)
('A', 'B', 'D')
>>> next(it3)
('A', 'C', 'D')
>>> next(it3)
('B', 'C', 'D')
>>> next(it3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

1.8.1 Funcions del mòdul operator

El mòdul `operator` proporciona un conjunt de funcions eficients corresponents als operadors bàsics de Python. Per exemple, `operator.add(x, y)` es equivalent a l'expressió `x+y`. Algunes de les operacions d'aquest mòdul són les següents:

operator	add	sub	mul	mod	truediv	floordiv	pow	lt	le	gt	ge
operació	+	-	*	%	/	//	**	<	<=	>	>=

operator	contains	getitem	setitem
operació	in	a[b]	a[b] = c

En exemples anteriors hem usat les operacions `mul`, `pow` i `mod`. En veurem d'altres en exemples successius.

1.8.2 Funcions del mòdul functools

D'aquest mòdul usarem la funció:

`functools.reduce(funció, iterable[, initial])`

La funció `reduce` aplica de forma acumulativa la funció donada com a primer paràmetre als elements de l'iterable, d'esquerra a dreta. La funció `reduce` fa una síntesi i retorna el resultat del procés acumulatiu; no retorna cap iterador. El tercer paràmetre, opcional, és el valor al qual inicialitzen aquest procés acumulatiu. Si no s'indica, es comença amb el primer element de l'iterable. Exemples:

```
>>> import functools
>>> import operator
>>> import math
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) # definim una funció lambda
15
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5]) # usem add (suma)
15
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5], 10) # valor inicial = 10
25
>>> functools.reduce(operator.sub, [100, 20, 30, 40, 7]) # fem diferències
3
>>> functools.reduce(operator.mul, [1, 2, 3, 4]) # equival a factorial(4)
24
>>> math.factorial(4)
24
>>> functools.reduce(operator.mul, [1, 2, 3, 4], 20) # és factorial(4)*20
480
```

La següent crida a la funció `reduce` realitza la seqüencia d'operacions: 1, 1+2, 3+3, 6+4 i 10+5. El resultat és 15.

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

La següent crida a la funció `reduce` realitza la seqüencia d'operacions: 10, 10+1, 11+2, 13+3, 16+4 i 20+5. El resultat és 25.

```
functools.reduce(operator.add, [1, 2, 3, 4, 5], 10)
```

Finalment, la següent crida a la funció `reduce` realitza la seqüencia d'operacions: 100, 100-20, 80-30, 50+40, 10-7. El resultat és 3.

```
functools.reduce(operator.sub, [100, 20, 30, 40, 7])
```

Comparem la funció `reduce` del mòdul `functools` amb la funció `accumulate` del mòdul `itertools`.

```
>>> import itertools
>>> import functools
>>> import operator
>>> functools.reduce(operator.sub, [100, 20, 30, 40, 7])
3

>>> it = itertools.accumulate([100, 20, 30, 40, 7], operator.sub)
>>> next(it)
100
>>> next(it)
80
>>> next(it)
50
>>> next(it)
10
```

```
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Bé, a part de què l'ordre dels paràmetres està capgirat, :), veiem que la funció `reduce` realitza un procés de síntesi acumulatiu i retorna el resultat final. En canvi la funció `accumulate` retorna un iterador els elements del qual són els successius valors que es van obtenint en l'anterior procés de síntesi acumulatiu.

2 Exercicis

En aquest apartat hi ha diversos exercicis del tema d'iteradors, explicats i resolts.

Alguns són adaptacions d'enunciats d'exercicis de Fonaments d'Informàtica i d'altres són nous. El darrer és un exercici d'examen.

Avís: Per a resoldre aquests exercicis no es poden fer servir llistes, tuples, diccionaris ni cap altra estructura de dades per a desar tots els elements d'un iterador. A més, en els exercicis en què no es demana el disseny d'una funció generadora sinó l'ús de funcions predeterminades o de les llibreries itertools o functools, no es poden usar ni la sentència `for` ni la sentència `while`.

Jocs de proves: Al fitxer testfiles-exercicis-iteradors.zip hi trobareu els jocs de proves dels exercicis.

2.1 Longitud de paraules

Enunciat

Donat un string amb paraules separades per espais en blanc i sense signes de puntuació i un enter, `n`,

1. dissenya la funció generadora `paraules1` que generi un iterador amb les paraules de l'string que tenen un nombre de lletres igual a `n`.
2. dissenya la funció `paraules2` que resolgui el mateix problema que l'anterior usant la funció `filter`.

Desa les dues funcions al fitxer `paraules.py`. Com que totes dues funcions fan el mateix, el següent joc de proves és valid per totes dues:

```
>>> it = paraules1 ('Si la casa es gran hi pots passejar', 4)
>>> it == iter(it)
True
>>> next(it)
'casa'
>>> next(it)
'gran'
>>> next(it)
'pots'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1 , in <module>
StopIteration
```

En aquest joc de proves, hi ha el test `it == iter(it)`. Aquest test comprova que el que retorna la funció, `it`, sigui un iterador (no una llista o algun altre iterable).

Resolució

La funció `paraules1` aplica primer el mètode `split` a l'string donat per tal d'obtenir una llista de paraules. Després cal recórrer aquesta llista i lliurar amb la sentència `yield` les paraules que tenen longitud igual a `n`:

```
def paraules1(s, n):
    for paraula in s.split():
        if len(paraula) == n:
            yield paraula
```

La funció `paraules2` ha d'aplicar la funció `filter` a la llista de paraules `s.split()`. La funció que cal posar com a paràmetre l'expressarem com una funció `lambda` que ha de ser una funció booleana que representa la condició de filtratge: `lambda paraula: len(paraula)==n`

```
def paraules2(s, n):
    return filter(lambda paraula: len(paraula) == n, s.split())
```

2.2 Aplicar prefix

Enunciat

Dissenyeu la funció `aplica_prefix` que, donada una llista de paraules i un prefix, retorni un iterador amb les mateixes paraules de la llista i en el mateix ordre però on cada paraula porti el prefix davant i separat per un guió. Deseu aquesta funció al fitxer `prefix.py`. En el disseny d'aquesta funció es recomana usar la funció `map`. Exemples:

```
>>> lpar = ['potent', 'disposat', 'judici']
>>> it = aplica_prefix(lpar, 'pre')
>>> it == iter(it)
True
>>> next(it)
'pre-potent'
>>> next(it)
'pre-disposat'
```

Resolució

Aquesta funció ha d'aplicar la funció `map` a les paraules de la llista donada, `lpar`. La funció que cal posar com a paràmetre l'expressarem com una funció `lambda` de forma que construeixi una nova paraula precedida pel prefix per cada paraula de la llista.

```
def aplica_prefix(lpar, prefix):
    return map(lambda paraula: prefix + '-' + paraula, lpar)
```

2.3 Dígits d'un nombre enter

1. Dissenya la funció generadora `xifres` que, donat un enter, genera un iterador amb els dígits de l'enter donat (les xifres, enters), de més a menys magnitud.

2. Dissenya la funció `calculs` que, donat un enter, retorna un tuple amb 3 enters: la xifra de valor màxim, la de valor mínim i la suma de les xifres de l'enter donat. Aquesta funció ha de cridar l'anterior. Es recomana usar la funció `itertools.tee`.

Deseu totes dues funcions al fitxer `xifres.py`. Exemples:

```
>>> it = xifres(7382)
>>> it == iter(it)
True
>>> for e in it:
...     print(e, end='/')
7/3/8/2/

>>> calculs(7382)
(8, 2, 20)
>>> calculs(9476345)
(9, 3, 38)
```

Resolució

La funció `xifres` converteix primer l'enter a string i després recorre els elements d'aquest string (xifres, strings), els converteix a enters i els lliura (sentència `yield`).

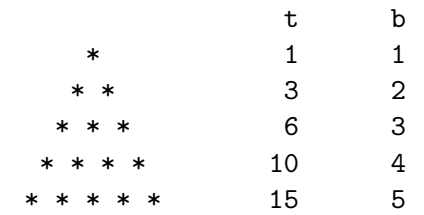
La funció `calculs` ha de cridar la funció `xifres` per obtenir l'iterador de les xifres i ha d'aplicar les funcions predefinides `max`, `min` i `sum` a l'iterador. Si apliquem una qualsevol d'aquestes funcions a l'iterador, aquest s'esgota. Per tant, necessitem tres rèpliques de l'iterador que podem obtenir amb la funció que ens recomanen usar `itertools.tee`. Caldrà, doncs, importar el mòdul `itertools`.

```
import itertools

def xifres(n):
    for x in str(n):
        yield int(x)

def calculs(n):
    a, b, c = itertools.tee(xifres(n), 3)
    return max(a), min(b), sum(c)
```

2.4 Nombres triangulars

Enunciat

Dissenya la funció generadora `triangulars` que generi un iterador infinit de tuples amb tots els nombres triangulars, començant per l'1. Cada tuple té dos elements: el nombre triangular i la seva base. Un nombre enter, `t`, és triangular si existeix una base, `b`, tal que $\sum_{i=1}^{b} i = t$. La següent figura mostra gràficament els primers 5 nombres triangulars i la seva base:



Deseu la funció al fitxer `triangulars.py`. La funció ha de passar el següent joc de proves:

```
>>> it = triangulars()
>>> it == iter(it)
True
>>> for i in range(6):
...     print(next(it), end='/')
(1, 1)/(3, 2)/(6, 3)/(10, 4)/(15, 5)/(21, 6)/
```

Resolució

Els nombres triangulars es generen a partir de la suma acumulativa dels naturals que són la seva base. Per tant, cal obtenir la seqüència dels nombres naturals i la de les successives sumes, que seran els nombres triangulars. En la caracterització d'aquesta seqüència no hi ha condició d'acabament ja que es demana un iterador infinit: la condició de permanència en la sentència `while` ha de ser sempre `True`. Per tant, l'expressió booleana que hi posarem serà `True`. Abans d'entrar al bucle obtenim el 1r element `i=1, n_triang = 1`. A dins el bucle, tractem l'element (lliurem el tuple) i obtenim el següent parell.

```
def triangulars():
    i = 1
    n_triang = 1
    while True:
        yield (n_triang, i)
        i = i + 1
        n_triang = n_triang + i
```

2.5 Construcció de frases 1

Enunciat

Donades tres llistes de paraules (strings), dissenya la funció `frases1` que retorni un iterador amb totes les possibles frases de 3 paraules que es poden formar amb el següent criteri: la frase i-èsima està formada per la paraula i-èsima de cada llista en l'ordre en què les llistes venen donades com a paràmetres. Cada frase es representa amb un string amb les tres paraules separades per un espai. El nombre de frases obtingudes és igual a la longitud de la llista més curta. Desa la funció al fitxer `frases1.py`. Exemple:

```
>>> l1 = ['la', 'alguna', 'una', 'aquesta']
>>> l2 = ['casa', 'poma', 'samarreta']
>>> l3 = ['petita', 'vermella']
>>> it = frases1(l1, l2, l3)
>>> next(it)
'la casa petita'
>>> next(it)
'alguna poma vermella'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Resolució

La funció `frases1` forma la 1a frase amb la 1a paraula de `l1`, de `l2` i de `l3`; la 2a frase amb la 2a paraula de `l1`, de `l2` i de `l3`; i així fins que s'acaben les paraules de la llista més curta. Aquest procés és el que fa la funció predeterminada `zip`. Ara bé aquesta funció retorna un iterador de tuples. Per l'exemple serien els tuples `('la', 'casa', 'petita')`, `('alguna', 'poma', 'vermella')`. Per convertir aquests tuples a strings hem d'usar el mètode `str.join` que hem d'aplicar a cada element de l'iterador amb la funció `map`:

```
    it1 = map(' '.join, it)
```

L'expressió `' '.join` també es pot escriure en forma de funció `lambda`:

```
  it1 = map(lambda x: ' '.join(x), it)
```

Però és més curta l'anterior sintaxi. Vegem el codi complet:

```
def frases1(l1, l2, l3):
    it = zip(l1, l2, l3)
    it1 = map(' '.join, it)
    return it1
```

2.6 Construcció de frases 2

Enunciat

Donades tres llistes de paraules (strings), dissenya la funció `frases2` que retorni un iterador amb totes les possibles frases de 3 paraules que es poden formar amb el següent criteri: la i-èsima paraula de cada frase és una paraula de la i-èsima llista donada. Cada frase es representa amb un string amb les tres paraules separades per un espai. Desa la funció al fitxer `frases2.py`. Exemple:

```
>>> l1 = ['la', 'alguna', 'una', 'aquesta']
>>> l2 = ['casa', 'poma', 'samarreta']
>>> l3 = ['petita', 'vermella']
>>> it = frases2 (l1, l2, l3)
>>> next(it)
'la casa petita'
>>> next(it)
'la casa vermella'
>>> next(it)
'la poma petita'
>>> next(it)
'la poma vermella'
>>> next(it)
'la samarreta petita'
>>> next(it)
'la samarreta vermella'
>>> next(it)
'alguna casa petita'
>>> next(it)
'alguna casa vermella'
```

Resolució

La funció `frases2` forma les frases realitzant un producte cartesià amb les paraules de les tres llistes. Per tant, es generaran `len(l1)*len(l2)*len(l3)` frases. En aquest cas hem d'usar la funció `itertools.product`. Aquesta funció també retorna un iterador de tuples. que convertirem a strings amb el mètode `str.join`. Hem d'importar el mòdul `itertools`:

```
import itertools
def frases2(l1, l2, l3):
    it = itertools.product(l1, l2, l3)
    it1 = map(' '.join, it)
    return it1
```

2.7 Successió matemàtica

Enunciat

Una successió matemàtica creixent es defineix com:

$x1 = 1$

$$x_i = \begin{cases} x_{i-1} + 5, & \text{si } i-1 \text{ és parell} \\ 2x_{i-1}, & \text{si } i-1 \text{ és senar} \end{cases}$$

1. Dissenya la funció generadora `successio` que genera un iterador infinit amb tots els elements d'aquesta successió, en ordre ascendent.

2. Dissenya la funció `llista_nums` que, donats dos enters n i m, retorna un iterador amb els elements de la successió creixent anterior, x, tals que $n <= x <= m$. Aquesta funció ha de cridar l'anterior i les funcions `itertools.dropwhile` i `itertools.takewhile`.

Deseu totes dues funcions al fitxer `successio.py`. Exemples:

```
>>> it = successio()
>>> it == iter(it)
True
>>> for i in range(16):
...     print(next(it), end = ' ')
1 2 7 14 19 38 43 86 91 182 187 374 379 758 763 1526

>>> list(llista_nums(100, 1000))
[182, 187, 374, 379, 758, 763]
```

Resolució

Per dissenyar la funció `successio` cal caracteritzar la seqüencia corresponent a la successió i també la dels nombres naturals que indica la posició dels elements i es necessita per obtenir el següent element de la successio. Per obtenir el següent element cal aplicar una sentència condicional seguint la definició matemàtica. Cada element que obtenim el lliurem amb la sentència `yield`. Com que ens demanen un iterador infinit, la condició de la sentència `while` és `True`.

```
def successio():
    i = 1
    x = 1
    while True:
        yield x
        if i%2 == 0:
            x = x + 5
        else:
            x = 2*x
        i = i +1
```

La funció `llista_nums` crida a la funció `successio` per obtenir l'iterador infinit, `it`.

```
    it = successio()
```

Després la funció `itertools.dropwhile` retorna un altre iterador infinit, `itn` amb tots els valors de l'iterador `it` més grans que `n`. Aquesta funció *deixa caure* (`drop`), és a dir, no lliura, els elements de `it` *mentre* (`while`) siguin més petits que `n`.

```
    itn = itertools.dropwhile(lambda x: x < n, it)
```

A continuació, la funció `itertools.takewhile` retorna un altre iterador infinit, `itnm` amb tots els valors de l'iterador `itn` més petits o iguals que `m`. Aquesta funció *pren* (`take`), és a dir, lliura, els elements de `itn` *mentre* (`while`) siguin més petits o iguals que `m`.

```
    itnm = itertools.takewhile(lambda x: x <= m, itn)
```

Hem d'importar el mòdul `itertools`. El codi complet és el següent:

```
import itertools
def llista_nums(n, m):
    it = successio()
    itn = itertools.dropwhile(lambda x: x < n, it)
    itnm = itertools.takewhile(lambda x: x <= m, itn)
    return itnm
```

2.8 Banderes

Enunciat

Un aficionat a la vexil·lologia (ciència que estudia les banderes) ens demana que l'ajudem a generar totes les banderes possibles amb 3 franges horitzontals de colors diferents, a partir d'una llista donada de colors, i tals que tinguin al mig un determinat color de la llista.

Per fer-lo content, nosaltres dissenyarem la funció `banderes` que, a partir d'una llista de colors (`string`) i d'un color dels de la llista, que representa el color del mig (`string`), retorna un iterador de banderes de les característiques sol.licitades. Cada bandera es representa amb un tuple de 3 `strings` on el 1r, 2n i 3r elements representen respectivament el color de la franja de sobre, del mig i de sota. Deseu la funció al fitxer `banderes.py`. Exemple:

```
>>> lcolors = ['vermell', 'blau', 'groc', 'blanc']
>>> it = banderes(lcolors, 'blanc')
>>> next(it)
('vermell', 'blanc', 'blau')
>>> next(it)
('vermell', 'blanc', 'groc')
>>> next(it)
('blau', 'blanc', 'vermell')
>>> next(it)
('blau', 'blanc', 'groc')
>>> next(it)
('groc', 'blanc', 'vermell')
>>> next(it)
('groc', 'blanc', 'blau')
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1 , in <module>
StopIteration
```

Aquest exemple genera les banderes de 3 colors diferents de la llista `lcolors = ['vermell', 'blau', 'groc', 'blanc']` i tals que tinguin el color blanc al mig (`colormig = 'blanc'`).

Resolució

La primera estratègia que seguim consisteix en generar un iterador, `itbanderes`, amb totes les possibles banderes de tres colors diferents, i després fer un filtre de les que tenen el color desitjat al mig. Pel primer pas, hem d'aplicar una funció combinatòria; com que no volem colors repetits i la bandera RGB és diferent de la BGR, és a dir, l'ordre dels colors importa, haurem d'usar la funció `itertools.permutations`, agafant els elements de la llista de 3 en 3.

```
    itbanderes = itertools.permutations(lcolors, 3)
```

A continuació hem d'aplicar la funció `filter` per filtrar les banderes, tuples `x` de l'iterador `itbanderes`, amb el color del mig demanat.

```
    itmig = filter(lambda x: x[1] == colormig, itbanderes)
```

Hem d'importar el mòdul `itertools` i aquí veiem el codi complet:

```
import itertools
def banderes(lcolors, colormig):
    itbanderes = itertools.permutations(lcolors, 3)
```

```
    itmig = filter(lambda x: x[1] == colormig, itbanderes)
    return itmig
```

Una altra estratègia consisteix en generar un iterador, `it2colors`, amb totes les possibles combinacions de dos colors per les franges de sobre i de sota, ja que el color de la franja del mig està prefixat. Per això cal aplicar la funció `itertools.permutations` agafant els elements de la llista de colors, eliminant el color que volem al mig, de 2 en 2.

Abans que res, hem d'eliminar el color que volem a la franja del mig de la llista de colors donada, amb el mètode `list.remove`, que és un mètode modificador:

```
    lcolors.remove(colormig)
```

Ara apliquem l'iterador combinatori:

```
    it2colors = itertools.permutations(lcolors, 2)
```

i finalment, amb la funció `map` apliquem una funció que converteixi els tuples de 2 elements de l'iterador `it2colors`, `x`, en tuples de 3 elements, amb el color desitjat al mig:

```
    itricolor = map(lambda x: (x[0], colormig, x[1]), it2colors)
```

Hem d'importar el mòdul `itertools` i aquí veiem el codi complet:

```
import itertools
def banderes(lcolors, colormig):
    lcolors.remove(colormig)
    it2colors = itertools.permutations(lcolors, 2)
    itricolor = map(lambda x: (x[0], colormig, x[1]), it2colors)
    return itricolor
```

2.9 Morse

Enunciat

Els codis Morse de les lletres s'obtenen construint grups de 1, 2, 3 i 4 caràcters amb els caràcters punt i guió que suposem que vénen en una llista, `['.', '-']`. Aquest mètode dóna 30 codis diferents: 2 codis amb un sol caràcter; i 4 codis de dos caràcters, 8 codis de tres caràcters i 16 codis de 4 caràcters que s'obtenen amb el producte cartesià entre la llista de dos caràcters i ella mateixa 2, 3 i 4 vegades respectivament.

Dissenya la funció `morse` que retorni un iterador amb aquests 30 codis. Tots els codis es representen amb un string i han s'estar en ordre lèxicogràfic a partir de la llista amb els dos caràcters. Exemple:

```
>>> it = morse()
>>> for i in range(2):                # 2 codis d'un caràcter
...  print(next(it), end = '   ')
.   -
>>> for i in range(4):                # 4 codis de dos caràcters
...  print(next(it), end = '   ')
..   .-   -.   --
>>> next(it)                          # 1r codi de tres caràcters
'...'
>>> for i in range(7):                # els altres 7 codis de tres caràcters
...  print(next(it), end = '   ')
..-   .-.   .--   -..   -.-   --.   ---
>>> next(it)                          # 1r codi de 4 caràcters
'....'
>>> next(it)                          # 2n codi de 4 caràcters
'...-'
>>> for i in range(12):               # 12 codis següents de 4 caràcters
...  print(next(it), end = '  ')
..-.  ..--  .-..  .-.-  .--.  .---  -...  -..-  -.-.  -.--  --..  --.-
>>> next(it)                          # penúltim codi de 4 caràcters
'---.'
>>> next(it)                          # últim codi de 4 caràcters
'----'
```

Desa la funció al fitxer `morse.py`.

Resolució

Hem de generar 4 grups de codis que obtindrem com 4 iteradors diferents. L'iterador, `it1` format pels dos codis d'un caràcter s'obté aplicant la funció `iter` a la llista formada per aquests dos caràcters:

```
    chars = ['.', '-']
    it1 = iter(chars)
```

Per obtenir els iteradors, `it2`, `it3` i `it4`, de 2, 3 i 4 caràcters respectivament, necessitem fer el producte cartesià de la llista `chars` per ella mateixa 2, 3 i 4 vegades respectivament. Per tant usarem la funció combinatòria `itertools.product`:

```
    it2 = itertools.product(chars, repeat =2)
    it3 = itertools.product(chars, repeat =3)
    it4 = itertools.product(chars, repeat =4)
```

Recordem que aquesta funció retorna tuples de caràcters i a l'enunciat demanen un string. Cal usar la funció `map` amb el mètode `''.join`. Podríem aplicar aquesta funció a cadascun dels iteradors anteriors o bé ajuntar-los i aplicar-la un sol cop. De fet, al final caldrà que ajuntem els quatre iteradors obtinguts. Doncs ara ajuntem els 3 iteradors esmentats amb la funció `itertools.chain`:

```
    it = itertools.chain(it2, it3, it4)
```

i a l'iterador obtingut li apliquem la funció `map` amb el mètode `''.join`:

```
    it = map(''.join, it)
```

Ara només falta afegir el primer de tots els iteradors al davant:

```
    itfinal = itertools.chain (it1, it)
```

Hem d'importar el mòdul `itertools` i aquí veiem el codi complet:

```
import itertools
def morse():
    chars = ['.', '-']
    it1 = iter(chars)
    it2 = itertools.product(chars, repeat =2)
    it3 = itertools.product(chars, repeat =3)
    it4 = itertools.product(chars, repeat =4)
    it = itertools.chain(it2, it3, it4)
    it = map(''.join, it)
    itfinal = itertools.chain (it1, it)
    return itfinal
```

2.10 Centre fisioteràpia 1

Enunciat

Un centre de fisioteràpia emmagatzema en una llista les dates que té concertades amb un determinat pacient, en ordre cronològic. Aquestes dates són instàncies de la classe `datetime.date`.

Dissenya la funció `cites_mes` que a partir d'una llista de dates com la descrita i un mes (enter), retorna un iterador sobre els dies (enters) de les dates corresponents al mes donat, en el mateix ordre que a la llista. Desa la funció al fitxer `citesmes.py`. Exemple:

```
>>> d1= datetime.date(2020, 2, 13)
>>> d2= datetime.date(2020, 2, 18)
>>> d3= datetime.date(2020, 3, 8)
>>> d4= datetime.date(2020, 3, 15)
>>> d5= datetime.date(2020, 3, 20)
>>> d6= datetime.date(2020, 3, 24)
>>> d7= datetime.date(2020, 4, 1)
>>> ld = [d1, d2, d3, d4, d5, d6, d7]

>>> it = cites_mes(ld, 3)
>>> for dia in it:
...     print(dia, end='/')
8/15/20/24/
```

Resolució

Pel disseny de la funció `cites_mes` aplicarem primer un filtre a la llista de dates (instàncies a la classe `datetime.date`) que filtri el mes donat. L'iterador `it1` és la seqüència de dates corresponents al mes donat:

```
    it1 = filter(lambda x: x.month == mes, ldates)
```

A continuació, la funció `map` permetrà extreure l'atribut `day` de cadascuna de les dates de l'iterador `it1`

```
    it2 = map (lambda x: x.day, it1)
```

Hem d'importar el mòdul `datetime` i aquí veiem el codi complet:

```
import datetime
def cites_mes(ldates, mes):
    it1 = filter(lambda x: x.month == mes, ldates)
    it2 = map (lambda x: x.day, it1)
    return it2
```

2.11 Centre fisioteràpia 2

El mateix centre ens demana que dissenyem la funció `total_cites` que a partir d'una llista de dates com la descrita, retorni un iterador de tuples de la forma `(mes, llista de dies)` on `mes` és un enter indicant el mes i `llista de dies` és una llista d'enters indicant els dies d'aquell mes en què el pacient té cites concertades, en el mateix ordre que la llista donada. Desa la funció al fitxer `totalcites.py`. Exemple:

```
>>> d1= datetime.date(2020, 2, 13)
>>> d2= datetime.date(2020, 2, 18)
>>> d3= datetime.date(2020, 3, 8)
>>> d4= datetime.date(2020, 3, 15)
>>> d5= datetime.date(2020, 3, 20)
>>> d6= datetime.date(2020, 3, 24)
>>> d7= datetime.date(2020, 4, 1)
>>> ld = [d1, d2, d3, d4, d5, d6, d7]

>>> it = total_cites(ld)
>>> next(it)
(2, [13, 18])
>>> next(it)
(3, [8, 15, 20, 24])
>>> next(it)
(4, [1])
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1 , in <module>
StopIteration
```

Resolució

Pel disseny de la funció `total_cites` necessitem agrupar les dates de la llista `ld` per mesos. La funció `itertools.groupby` ens permet fer aquest agrupament sempre que li indiquem la funció amb la qual fem l'agrupament com a paràmetre opcional, `key`. Com que volem agrupar per mesos, aquesta funció és: `lambda x: x.month`.

La funció `itertools.groupby` agrupa elements que siguin consecutius. Aleshores, si volem agrupar per mesos, cal que les dates de cada mes estiguin juntes. Com que la llista de dates `ld`, a la qual aplicarem aquesta funció, està ordenada cronològicament, totes les dates d'un mes estaran juntes i per tant es crearan grups de mesos complets i, a més, amb els dies en ordre cronològic.

```
    gb = itertools.groupby(ldates, key = lambda x: x.month)
```

El que retorna la funció `groupby`, `gb`, és un iterador de tuples on el 1r element és el mes i el 2n element un iterador de dates. Per entendre l'estructura d'aquest iterador, vegem el següent joc de proves:

```
>>> gb = itertools.groupby(ld, key = lambda x: x.month)
>>> (mes, it) = next(gb)    # 1r element de gb: (mes, iterador de dates)
>>> mes
2
>>> next(it)              # 1a data de l'iterador de dates del 1r elem. de gb
datetime.date(2020, 2, 13)
>>> next(it)              # 2a data de l'iterador de dates del 1r elem. de gb
datetime.date(2020, 2, 18)

>>> (mes, it) = next(gb) # 2n element de gb: (mes, iterador de dates)
>>> mes
3
>>> next(it)              # 1a data de l'iterador de dates del 2n elem. de gb
datetime.date(2020, 3, 8)
>>> next(it)              # 2a data de l'iterador de dates del 2n elem. de gb
datetime.date(2020, 3, 15)
>>> next(it)              # 3a data de l'iterador de dates del 2n elem. de gb
datetime.date(2020, 3, 20)
>>> next(it)              # 4a data de l'iterador de dates del 2n elem. de gb
datetime.date(2020, 3, 24)

>>> (mes, it) = next(gb) # 3r i últim element de gb
>>> mes
4
```

L'objecte `gb` ja conté la informació que ens demanen, però no en el format que ens el demanen. Per cada tuple de l'iterador `gb` hem de convertir el segon element que és un iterador de dates en una llista d'enters (dies). Per convertir una data (instància de la classe `datetime.date`) a un dia (enter) usarem la funció `fdia`:

```
    fdia = lambda data: data.day
```

Com veiem, podem usar un nom (`fdia`) per designar una funció `lambda`. Això facilitarà la comprensió de l'expressió que veurem tot seguit.

Si li diem `x` als tuples de `gb`, cal fer una aplicació a aquests tuples de forma que el 1r element `x[0]` quedi igual i el segon `x[1]` que és un iterador es converteixi en una llista. I, a més, els elements d'aquesta llista que són dates s'han de convertir en enters. Per això caldrà aplicar la funció `map` dues vegades.

Hem d'aplicar la funció `map` a l'iterador `x[1]` amb la funció `fdia` per tal que converteixi les dates en dies: `map(fdia, x[1])`:

I també hem d'aplicar la funció `map` als tuples `x` fent que `x[0]` quedi igual i `x[1]` es converteixi en una llista:

```
    it = map(lambda x: (x[0], list(map(fdia, x[1]))), gb)
```

Hem d'importar els mòduls `datetime` i `itertools` i aquí veiem el codi complet:

```
import datetime
import itertools
def total_cites(ldates):
    gb = itertools.groupby(ldates, key = lambda x: x.month)
    fdia = lambda data: data.day
    it = map(lambda x: (x[0], list(map(fdia, x[1]))), gb)
```

```
    return it
```

2.12 Dies feiners

El següent enunciat correspon a l'exercici 1 de l'examen final d'Informàtica-Torn 1, 20 de gener de 2020.

Enunciat

Avís: Per a resoldre aquest exercici no es poden fer servir llistes, tuples, diccionaris ni cap altra estructura de dades per a desar tots els elements d'un iterador.

Dissenyeu la funció `feiners(m, a)` i deseu-la al fitxer `iterdies.py`.

Donats dos enters `m` i `a` que representen un mes i un any, respectivament, retorna un iterador sobre la seqüència dels dies del més de l'any donat que són feiners, és a dir, que no són ni dissabte ni diumenge (els festius que no són dissabte ni diumenge també cal generar-los). Els element generats per l'iterador han de ser instàncies de la classe datetime.date i s'han de generar cronològicament. Per exemple,

```
>>> it = feiners(1, 2000)
>>> for d in it:
...       print(d, end=',')
2000-01-03,2000-01-04,2000-01-05,2000-01-06,2000-01-07,2000-01-10,2000-01-11,
2000-01-12,2000-01-13,2000-01-14,2000-01-17,2000-01-18,2000-01-19,2000-01-20,
2000-01-21,2000-01-24,2000-01-25,2000-01-26,2000-01-27,2000-01-28,2000-01-31,
```

En la resolució d'aquest problema us pot ser d'utilitat el mètode `datetime.date.weekday()` i la suma d'una instància de la classe `datetime.date` amb un interval de temps (`datetime.timedelta`). Alternativament, també podeu utilitzar la funció `calendar.monthrange()`.

Resolució

Abans de resoldre aquest exercici cal consultar el mòdul datetime. Recordem que el mètode weekday retorna un enter 0, 1, . . . ,6 si la data a la qual s'aplica és dilluns, dimarts, . . . , diumenge, respectivament. També recordem que si sumem una instància de la classe `datetime.date` amb un interval de temps (instància a la classe `datetime.timedelta`) el resultat és una altra instància de la classe `datetime.date`.

A més, si consultem el mòdul `calendar`, veiem que la funció `monthrange` té dos paràmetres que són l'any i el més i retorna un tuple amb el dia de la setmana del 1r dia d'aquest més i el nombre de dies d'aquest més.

Veurem tres possibles solucions. Veiem que a l'enunciat ens prohibeixen usar llistes, etc., però no ens prohibeixen fer servir for ni while. Les dues primeres solucions usaran for/while i la tercera no.

Solució 1

La funció `monthrange` retorna el nombre de dies d'un mes. Amb la funció range podem generar els dies del mes donat. Aleshores només hem de fer un recorregut pels dies del mes donat i filtrar els que són dies feiners. Vegem-ho pas a pas:

Importem els mòduls que necessitem:

```
import datetime
import calendar
```

Capçalera de la funció:

```
def feiners_1(mes, an):
```

Cridem `monthrange` per saber el nombre de dies d'aquest mes, `ndies`. El primer component del tuple que retorna aquesta funció, `diasetmana`, no l'usarem:

```
    diasetmana, ndies = calendar.monthrange(an, mes)
```

La funció `range(n)` genera una seqüència de `n` nombres enters començant per 0. Volem `ndies` nombres i començant per `1`. Per tant, la crida serà `range(1, ndies+1)`.

El recorregut per tots els dies del mes donat el farem amb la sentència `for` i sobre la seqüència que produeix la funció `range`:

```
    for dia in range(1, ndies+1):
```

Ara necessitem obtenir una instància de la classe `datetime.date` que guardarem a la variable `data` a partir de l'any i mes donats i del dia que va generant la sentència `for`.

També volem saber en quin dia de la setmana cau aquesta data, informació que guardem a la variable `wd` (weekday):

```
        data = datetime.date(an, mes, dia)
        wd = data.weekday()
```

Finalment, si el dia de la setmana és diferent de dissabte (`wd != 5`) i diferent de diumenge (`wd != 6`) és un dia feiner i cal lliurar-lo.

```
        if wd != 5 and wd != 6: # equival a wd <5
            yield data
```

L'expressió booleana de la sentència if és equivalent a: `wd < 5`

Veiem tot el disseny sencer:

```
import datetime
import calendar

def feiners_1(mes, an):
    diasetmana, ndies = calendar.monthrange(an, mes)
    for dia in range(1, ndies+1):
        data = datetime.date(an, mes, dia)
        wd = data.weekday()
        if wd != 5 and wd != 6: # equival a wd <5
            yield data
```

Solució 2

En aquesta solució seguim la pista de l'enunciat que ens recorda que podem sumar una instància de la classe `datetime.date` i una instància a la classe `datetime.timedelta` obtenint una altra instància de la classe `datetime.date`.

També generarem la seqüència dels dies del mes i any donats, però no els enters com fèiem a l'anterior solució sinó directament les instàncies de la classe `datetime.date`. Primer caracteritzarem la seqüència.

El 1r element de la seqüència és el dia 1 del mes i any donats, que s'obté com una instància a la classe `datetime.date`:

```
    data = datetime.date(an, mes, 1) # 1r element seqüència
```

Per obtenir el següent element, hem de sumar un dia a l'element anterior. Usarem la classe `datetime.timedelta`.

```
        data = data + datetime.timedelta(1)  # següent element seqüència
```

Arribarà un moment que, quan se sumi un dia, la data canviarà de mes i l'atribut `month` d'aquesta data serà diferent del mes donat com a paràmetre. La condició d'acabament serà `data.month != mes`, que és la contrària de la de permanència al `while`.

```
    while data.month == mes:          # condició permanència al while
```

Ja només hem d'incloure en el cos del `while` les mateixes dues sentències que hi ha al cos del `for`: obtenir el dia de la setmana i comprovar si és feiner:

```
        wd = data.weekday()
        if wd != 5 and wd != 6:
            yield data
```

Aquest disseny només requereix importar el mòdul `datetime` i tot seguit es mostra el disseny complet:

```
import datetime

def feiners_2(mes, an):
    data = datetime.date(an, mes, 1) # 1r element seqüència
    while data.month == mes:          # condició permanència al while
        wd = data.weekday()
        if wd != 5 and wd != 6:
            yield data
        data = data + datetime.timedelta(1)  # següent element seqüència
```

Solució 3

En aquesta solució no usarem `for` ni `while`. De les dues solucions anteriors hem après que necessitem obtenir els dies del mes donat i després filtrar els feiners.

Per obtenir els dies del mes, usarem una estratègia similar a la solució 1 i la funció `map`.

Obtenim primer el nombre de dies que té el mes:

```
    diasetmana, ndies = calendar.monthrange(an, mes)
```

Generarem els dies del mes (nombres enters) amb la funció `range` i li aplicarem (funció `map`) la següent funció anònima: `lambda dia: datetime.date(an, mes, dia)`, que per cada enter obté la corresponent instància a la classe `datetime.date`.

La comanda següent genera l'iterador dels dies del mes donat, `dies_mes`, que són instàncies a la classe `datetime.date`.

```
    dies_mes = map(lambda dia: datetime.date(an, mes, dia),range(1, ndies+1))
```

A l'iterador `dies_mes` ara cal aplicar-li un filtre (funció `filter`) que filtri les dates que són dies feiners. La funció que hem de posar com a paràmetre a `filter` l'expressem també com una funció anònima: `lambda data: data.weekday() < 5`. La variable `dies_feiners` és l'iterador de les dates (instàncies a la classe `datetime.date`) que corresponen a dies feiners i que cal retornar.

```
    dies_feiners = filter (lambda data: data.weekday() < 5, dies_mes)
```

Aquest disseny requereix importar els dos mòduls `datetime` i `calendar` i tot seguit es mostra el disseny complet:

```
import datetime
import calendar
```

```
def feiners(mes, an):
    # obté el nombre de dies que té el mes
    diasetmana, ndies = calendar.monthrange(an, mes)
    # geners els dies del mes
    dies_mes = map(lambda dia: datetime.date(an, mes, dia),range(1, ndies+1))
    # filtra els dies feiners del mes
    dies_feiners = filter (lambda data: data.weekday() < 5, dies_mes)
    return dies_feiners
```