# Apunts d'Informàtica

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

Tardor 2023

.

# Índex

# 1 Introducció

## 1.1 Programació i llenguatges de programació

Un computador pot fer bàsicament dues coses, però les fa de forma eficaç i eficient. Els computadors realitzen càlculs a una gran velocitat i són capaços de tractar grans quantitats d'informació.

D'altra banda, les persones aprofitem aquestes característiques i usem els computadors per a la resolució de problemes. Aquest és un ampli camp de recerca anomenat programació computacional.

Durant més de 70 anys s'ha avançat molt en aquesta àrea i s'han desenvolupat dos paradigmes de programació: el declaratiu i l'imperatiu.

El paradigma declaratiu es basa en la declaració de fets. Per exemple aquestes fórmules matemàtiques: $diametre = 2radi$ i $area\_cercle = \pi radi^2$ són declaracions de fet (fórmules), però no ens expliquen el que cal fer per calcular el diàmetre d'un cercle donada la seva àrea.

D'altra banda, el paradigma imperatiu dóna receptes que descriuen els passos a seguir per a resoldre un problema. A l'exemple anterior, la recepta per calcular el diàmetre seria:

1. calcula el radi amb l'expressió: $radi = \sqrt{\frac{area\_cercle}{\pi}}$
2. calcula el diàmetre com el doble del radi

Aquesta és una recepta molt senzilla però aquest paradigma es pot aplicar a mètodes més complexos.

En aquest curs seguirem el **paradigma imperatiu**.

Un mètode o algorisme és una llista finita d'instruccions o sentències que s'executen sobre un conjunt de dades de forma que es produeix un conjunt d'estats ben definits i al final s'obté un resultat.

Quan un algorisme està dirigit a un computador es denomina programa o funció. Així, en aquest curs, aprendrem a dissenyar programes o, més específicament, funcions.

**Llenguatges de programació**

Els primers computadors eren màquines que permetien executar un sol programa com, per exemple, la resolució de sistemes d'equacions lineals, però res més. Avui en dia encara hi ha computadors que utilitzen aquest enfoc com les calculadores de quatre funcions i molts dispositius a la indústria mecànica, per posar alguns exemples.

D'altra banda, els computadors actuals com els PC són computadors versàtils ja que poden emmagatzemar i executar molts programes diferents. Aquest enfoc va ser formulat per primer cop per Alan Turing al començament del segle passat.

Un llenguatge de programació és un llenguatge formal que permet escriure programes. Els llenguatges com Python, C, Java, etc. són llenguatges Turing-complets, que significa que es poden utilitzar per escriure programes que es puguin executar en computadors versàtils.

Un aspecte important sobre els llenguatges de programació és que les persones escriuen els programes i els computadors els executen. En realitat, un computador executarà exactament el que se li indiqui. Més vegades del que es voldria el que executa el computador no és el que pretenia el programador i aquest és el pitjor error de programació i més difícil de depurar.

Podem classificar els llenguatges de programació utilitzant diversos eixos:

1. Baix-nivell versus alt-nivell. Les instruccions poden estar al nivell de la màquina (llenguatge màquina) o bé són operacions més abstractes subministrades pel dissenyador del llenguatge que són més familiars a les persones.

2. General versus adreçat a un domini d'aplicació. Un llenguatge de propòsit general permet dissenyar programes que resolguin diversos tipus de problemes mentre que hi ha llenguatges dirigits a una aplicació concreta: per exemple, per dissenyar pàgines web (Adobe Flash) o per gestionar bases de dades (Excel).

3. Interpretat versus compilat. Un llenguatge interpretat executa directament les instruccions del programa mentre que un llenguatge compilat tradueix primer el codi font del programa a codi màquina (mitjançant un programa anomenat compilador) i després l'executa. Els llenguatges interpretats són generalment més fàcils de depurar i, per tant, són una bona opció per als principiants. Per altra banda, els llenguatges compilats són, en general, més eficients pel que fa a temps i espai d'execució.

Python és un llenguatge d'**alt-nivell**, de **propòsit general** i **interpretat**. És relativament fàcil d'aprendre i disposa d'un gran nombre de biblioteques de funcions (de programari lliure) que li proporcionen una extensa funcionalitat.

## 1.2 Elements bàsics de Python

Un llenguatge de programació ofereix un conjunt d'elements bàsics així com una sintaxi i una semàntica.

Un programa Python (script) és una seqüència de comandes, instruccions o sentències.

## 1.3 Objectes i tipus

### Objecte

Els objectes són els elements fonamentals que manipula un programa. També ens referirem als objectes com a valors.

### Tipus

Tot objecte és d'un determinat tipus. El tipus defineix el rang de valors dels objectes i les operacions que es poden fer amb els objectes.

Els tipus poden ser escalars or no escalars. Els objectes d'un tipus escalar són indivisibles mentre que els objectes d'un tipus no escalar presenten una estructura interna.

A Python hi ha tres tipus bàsics escalars: `int` (enter), `float` (real) i `bool` (booleà).

La funció `type` indica el tipus d'un objecte.

Exemples:

```
>>> type (3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type(True)
<class 'bool'>
```

## 1.4 El tipus enter (`int`)

Els nombres enters són un conjunt infinit i si els volem representar en un computador haurem de tenir en compte que només en podem representar un subconjunt. El tipus `int` permet representar un subconjunt dels nombres enters entre un nombre negatiu MIN i un nombre positiu MAX. Per exemple, un `int` de 4 bytes pot representar el rang de valors $[-2^{31}, 2^{31} - 1]$.

Els objectes del tipus `int` es representen amb una seqüència de dígits i el caràcter `-`. Exemples: `3,-24, 6789, 0` ....

Els operadors del tipus `int` són: `+, -, *, /, //, %` i `**`. Una operació es diu que és **binària** quan hi intervenen dos operands i es diu **unària** quan només hi intervé un operand. Totes les anteriors són binàries. L'operació canvi de signe és una operació unària i usa el mateix símbol (`-`) que l'operació diferència.

El resultat d'una suma, diferència i producte entre dos objectes de tipus `int` és un altre objecte de tipus `int`.

Hi ha dos operadors de divisió. El primer és l'operador **divisió real** i es simbolitza amb el símbol /. Tant si els operands són enters com reals sempre fa la divisió real i el resultat és de tipus **float** (real). Exemples:

```
>>> 6/4
1.5
>>> 6.0/4
1.5
>>> 6.0/4.0
1.5
```

L'altre operador de divisió és l'operador **divisió entera** i es representa amb el símbol //. Tant si els operands són enters com reals sempre fa la divisió entera. El resultat és de tipus **int** si els dos operands són enters i **float** altrament.

```
>>> 6//4
1
>>> 6.0//4
1.0
>>> 5//-2.0
-3.0
```

A part de l'operador per la divisió entera també hi ha l'operador que permet calcular el **mòdul o residu de la divisió entera**. Tots dos permeten aplicar l'operació de la divisió entera. Siguin $D$, $d$, $q$ i $r$, respectivament, el dividend, divisor, quocient i residu (o mòdul) d'una divisió entera, és a dir, es compleix:

$$D = qd + r$$

Aleshores, donats el dividend i divisor, el quocient i residu es calculen com:

```
q = D//d
r = D%d
```

Exemples:

```
>>> 6//4
1
>>> 6%4
2
>>> 17%5
3
>>> 18%5
2
```

La divisió entera aplicada a operands negatius dóna resultats que poden ser sorprenents. Analitzant els següents exemples es pot comprovar que el comportament és el mateix amb operands negatius o positius:

```
>>> 7//2
3
>>> 7%2
1
>>> 7//-2
-4
>>> 7%-2
-1
>>> -7//2
-4
>>> -7%2
1
>>> -7//-2
3
>>> -7%-2
-1
```

L'operador d'exponenciació es simbolitza amb un doble asterisc, `**`. Exemple: l'expressió matemàtica $2^4$ s'escriu en Python com: `2**4`

## 1.5 Tipus real (`float`)

El tipus `float` permet representar un subconjunt dels nombres reals. El tipus `float` en Python permet representar rangs de valors més amplis que el tipus `int`. Ara bé, en el

tipus `float` no només hi ha limitació en la magnitud del nombre sinó també en la precisió de la seva representació.

Els objectes de tipus `float` es representen amb una seqüència de dígits, el caràcter punt (obligatori) i el caràcter `-`. També es poden representar en notació científica. Exemples d'objectes o valors de tipus `float` en Python: `3.`, `-24.45`, `6789.3`, `0.0`, `1.0e2`, `2.3E-5`. En notació científica `2.3E-5` significa 2.3 $10^{-5}$.

El nom `float` prové de la forma com els valors d'aquest tipus estan representats a la memòria del computador que és una representació en base 2 similar a la notació científica o també dita de coma flotant (*floating point* en anglès). D'aquí ve el nom `float`.

Les operacions que suporta el tipus `float` són: `+`, `-`, `*`, `/`, `//`, `**`.

Es poden operar objectes de tipus `int` amb objectes de tipus `float`. Les regles que regeixen el tipus del resultat en aquestes operacions es mostren a la taula següent:

| op | int | float |
|---|---|---|
| int | int | float |
| float | float | float |

## 1.6 Tipus booleà (`bool`)

El tipus `bool` permet representar informació booleana. Els dos únics valors d'aquest tipus són `False` i `True`.

### 1.6.1 Operadors de comparació

Els operadors de comparació són operadors externs que permeten comparar objectes d'un mateix tipus (per exemple de tipus `int` o `float`) i donen un resultat de tipus booleà. Aquests operadors són: `==` (igual), `!=` (diferent), `<` (més petit), `<=` (més petit o igual), `>` (més gran), `>=` (més gran o igual). Noteu que per representar algun operador calen dos símbols i que l'ordre en què es posen és l'indicat.

Exemples:

```
>>> 3 > 4
False
>>> 2.5 <= 3
True
>>> 9 != 7
True
```

### 1.6.2 Operacions booleanes

Les operacions lògiques o booleanes són: `and`, `or` (binàries) i `not` (unària) i es defineixen tal com indiquen les anomenades taules de veritat següents:

| and | True | False |
|---|---|---|
| True | True | False |
| False | False | False |

| or | True | False |
|---|---|---|
| True | True | True |
| False | True | False |

| not | True | False |
|---|---|---|
| | False | True |

Usant conjuntament els operadors de comparació i les operacions booleanes es poden construir expressions booleanes més complexes. L'expressió `a <= b and b <= c` val `True` si `a` és més petit o igual que `b` i, a més, `b` és més petit o igual que `c`. A diferència d'altres llenguatges, en Python aquests operadors no són associatius i s'admet la notació matemàtica `a == b ==c` que és equivalent a l'expressió `a == b and b == c`. Dit d'una altra manera, els operadors de comparació es poden encadenar arbitràriament: `a <= b <= c < d > e` és equivalent a `a <= b and b <= c and c < d and d > e`.

Exemples:

```
>>> 1 < 6 < 10
True
>> 1 < 6 and 6 < 10
True
```

## 1.7 Tipus string o cadena de caràcters (`str`)

El tipus `str` (string o cadena de caràcters) permet representar seqüències de caràcters: noms, codis, números de telèfon, etc. Els objectes de tipus **str** es representen entre cometes simples o dobles. Exemples: `'abc'`, `'Demà plourà'`, `''L'amic i l'amat''`, `'aa@aaa.aa'`, `'44444444H'`. Hi poden haver caràcters alfabètics (lletres), numèrics (dígits) i altres.

El tipus `str` és un tipus no escalar, en realitat és un tipus seqüencial com altres tipus que es veuran més endavant. En aquest capítol es veuran les operacions amb strings considerats com un bloc (com si fossin un tipus escalar). Més endavant es veurà com accedir i operar amb cada caràcter de la seqüència, individualment. Les operacions per strings considerats com un bloc són la concatenació (`+`) i la repetició (`*`). A més, es pot obtenir la longitud d'un string (nombre de caràcters) amb la funció `len` de la biblioteca estàndard. Exemples:

```
>>> 'abc' + 'def'
'abcdef'
>>> len ('esmaperdut')
10
>>> 'abc'*3
'abcabcabc'
```

Els operadors `+` i `*` són operadors de sobrecàrrega: tenen diferents significats depenent dels tipus d'objectes als que s'apliquin. De totes maneres, quan s'apliquen al tipus `int` (o `float`) tenen un comportament equivalent a quan s'apliquen al tipus `str`. `2+3` es pot expressar com `1+1+1+1+1` i `'tu'+ 'jo'` com `'t'+'u'+'j'+'o'`. Respecte la repetició (*), l'expressió `2*3` és equivalent a l'expressió `3+3` i l'expressió `2*'tu'` és equivalent a l'expressió `'tu'+'tu'`.

La codificació més habitual de caràcters és la codificació ASCII (de l'anglès American Standard Code for Information Interchange). En aquesta codificació cada caràcter té un

codi assignat i es distingeix entre lletres majúscules i minúscules. En aquesta codificació no es consideren lletres accentuades ni amb altres símbols no propis de l'alfabet anglosaxó. L'assignació de codis és arbitrària excepte en els casos de 3 grups de caràcters: les lletres majúscules, les lletres minúscules i els dígits. Aquests grups estan posats de manera que les lletres estan en ordre alfabètic (primer hi ha el grup de les majúscules i després el de les minúscules) i els dígits en ordre numèric, i els codis corresponents són correlatius en cada grup. Per tant es pot considerar que hi ha definits els següents intervals:

- $['A','Z']$
- $['a','z']$
- $['0','9']$

La comparació entre strings es basa en l'ordre lexicogràfic, que és una generalització de l'ordre alfabètic. Exemples:

```
>>> 'a' < 'v'
True
>>>'7' > '9'
False
>>> 'ala' < 'pota'
False
>>> 'abracadabra' > 'ala'
False
>>>'100' < '200'
True
>>> '100' < '2'
True
```

Per tal de representar lletres accentuades i altres caràcters no propis de l'alfabet anglosaxó es pot usar la codificació UTF-8. De totes maneres, en aquest primer curs de programació, tot i que la versió de Python usada permet usar noms amb accents i altres caràcters no propis de la llengua anglesa, s'admet i aconsella que no es faci ús d'aquesta opció per evitar problemes, per exemple, en la comparació i ordenació d'strings.

## 1.8 Expressions

Una expressió és una combinació d'objectes, operadors i crides a funcions, que segueixen unes regles sintàctiques. Les expressions són **avaluades** per l'intèrpret de Python i donen lloc a un resultat. Els objectes que apareixen en una expressió s'anomenen **operands**. Les expressions en la majoria de llenguatges de programació, i Python en particular, s'escriuen en format de línia: no hi ha subíndexos ni superíndexos ni barres horitzontals de divisió. A continuació es mostren exemples d'expressions matemàtiques codificades en Python:

| exp. matemàtica | exp. Python |
|---|---|
| $2^4$ | `2**4` |
| $\frac{2+x}{3}$ | `(2+x)/3` |
| $\frac{6}{1+x}$ | `6/(1+x)` |
| $\frac{8}{2x}$ | `8/(2*x)` |
| $\frac{2x}{8}$ | `2*x/8` |

**Regles de precedència**:

Quan en una expressió hi ha més d'un operador, l'ordre en què s'avaluen depèn d'unes regles de precedència. Els operadors aritmètics tenen les regles de precedència habituals. La següent taula mostra els ordres de precedència dels operadors de Python:

| | |
|---|---|
| `()` | els parèntesis tenen la màxima precedència |
| `**` | associativitat per la dreta |
| `-` | canvi de signe (operador unari) |
| `*, /, //, %` | associativitat per l'esquerra |
| `+, -` | associativitat per l'esquerra |
| `==, !=, ...` | no són associatius |
| `not` | |
| `and` | |
| `or` | |

L'ordre de precedència per defecte es pot modificar usant parèntesis. Quan en una expressió hi apareixen operacions amb la mateixa precedència, s'avaluen d'esquerra a dreta (associatives per l'esquerra), excepte l'operació `**` que és associativa per la dreta. Exemple: `2**3**2` equival a $2^{(3^2)} = 2^9 = 512$ i `(2**3)**2` equival a $(2^3)^2 = 2^6 = 64$.

## 1.9 Variables

Una variable és un nom o identificador que fa referència (s'associa) a un objecte.

L'**identificador o nom** d'una variable pot ser arbitràriament llarg. Els noms de variables contenen lletres (majúscules o minúscules), dígits i també el caràcter guió baix (_). El primer caràcter ha de ser forçosament una lletra o bé el caràcter guió baix. Es distingeix entre majúscules i minúscules: `volum` i `Volum` són noms de variables diferents.

Els noms de variables no poden ser **paraules clau o reservades**. Les paraules reservades `int`, `and`, `False`, `def`, `if`, ... són les que s'usen en la sintaxi del llenguatge i, per tant, no es poden usar com noms de variables.

És molt recomanable triar noms que tinguin relació amb l'objecte que identifiquen, per tal que els programes siguin llegibles i auto-explicatius. Les variables poden tenir noms compostos. En aquest cas no es pot usar el caràcter espai en blanc per separar-ne les parts sinó que cal usar el caràcter guió baix: `volum esfera` és un nom de variable incorrecte; en canvi `volum_esfera` és correcte. Una altra notació usada és començant cada part del nom compost en majúscula sense separadors: `volumEsfera`.

## 1.10 Sentència d'assignació

Una sentència és una instrucció que pot ser interpretada i executada per l'intèrpret de Python. En aquest apartat s'introdueix la sentència d'assignació. Altes sentències com `if`, `for` o `while` s'introduiran més endavant.

La sentència d'assignació associa el resultat d'una expressió a una variable. La sintaxi és la següent:

`nom_variable = expressió`

Aquesta sintaxi i la corresponent semàntica es descriuen a continuació:

- L'operador d'assignació es representa mitjançant el símbol `=`. No s'ha de confondre amb l'operador de comparació d'igualtat `==`
- En llegir aquesta sentència hem de dir:
  **nom_variable pren per valor expressió**
  NO hem de dir *nom_variable és igual a expressió*
- L'operador d'assignació, `=`, no és commutatiu:
  - Al costat esquerra del símbol d'assignació hi ha sempre el nom d'una variable
  - Al costat dret del símbol d'assignació hi ha sempre una expressió
- El comportament de la sentència d'assignació és el següent:
  - primer s'avalua l'expressió del costat dret de la sentència i es produeix un resultat (valor)
  - aquest valor resultant s'assigna a la variable

Exemples:

```
>>> area=5       # area pren per valor 5
>>> area
5
>>> Area=area*2 # Area pren per valor area multiplicat per 2
>>> Area
10
>>> type(Area)
<class 'int'>
>>> Area=Area+15 # Area pren per valor Area més 15
>>> Area
25
>>> r= Area==30  #r pren per valor Area igual a 30
>>> r
False
>>> type (resultat)
<class 'bool'>
```

A costat d'algunes de les anteriors sentències d'assignació s'indica com s'ha de llegir: "area pren per valor 5" (i no *"area és igual a 5"*), "Area pren per valor Area més 15" (i no *"Area és igual a Area més 15"*), ...

Un objecte pot tenir associat cap, un o més d'un nom. En canvi un nom només té associat un únic objecte.

## 1.11 Biblioteca de funcions predefinides

Python disposa d'un conjunt de funcions estàndard o predefinides (biblioteca **builtins**). Vegeu la documentació Python per un llistat complet i la descripció detallada i actualitzada d'aquestes funcions. Per consultes puntuals es pot usar el shell de Python, que ofereix una documentació resumida, amb les funcions `dir` i `help`:

| | |
|---|---|
| llistar | `dir(__builtins__)` |
| documentar | `print(nomfuncio.__doc__)`<br>`help (nomfuncio)` (més senzill)<br>`q` per sortir |

Algunes funcions de la biblioteca estàndard són funcions de conversió entre tipus: `int`, `float`, `str`. Exemples:

```
>>> str (42.85)
'42.85'
>>> str (3*4)
'12'
>>> int (3*4.2)
12
>>> float (3*4)
12.0
>>> int('100')
100
>>> int('100 km')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '100 km'
```

El darrer exemple dóna error ja que l'string `'100 km'` no és un string que es pugui convertir a enter.

La funció `bool` retorna sempre `True`, excepte en els següent casos:

```
>>> bool(0.0)
False
>>> bool(0)
False
>>> bool('')
False
```

Per tant, el comportament de la funció `bool` aplicada als valors booleans no és l'esperada:

```
>>> bool('True')
True
>>> bool('False')
True
```

La funció `eval` de la biblioteca estàndard permet avaluar qualsevol expressió representada com un string i, per tant, es pot usar per convertir un booleà a string. Exemples:

```
>>> eval('True')
True
>>> eval('False')
False
>>> eval ('3*4')
12
>>> a = 5
>>> eval  ('a*2+6 < 20 and a < 10')
True
```

A continuació es mostren les funcions d'ús més comú de la biblioteca estàndard::

- `abs`: calcula el valor absolut
- `round`: arrodoneix un valor real
- `max, min`: màxim, mínim d'un conjunt de valors
- `type`: indica el tipus d'una variable o valor
- `len`: retorna el nombre d'elements d'una seqüència

```
>>> round(45.987623, 3)
45.988
>>> round(45.987623, 4)
45.9876
>>> max (4, 6, 5, 3)
6
>>> min (4, 6, 5, 3, 8)
3
```

## 1.12 Biblioteca de funcions matemàtiques

Python disposa d'una biblioteca amb les funcions matemàtiques més comuns: `log(x)`, `exp(x)` ($e^x$), `sqrt(x)` ($\sqrt(x)$), ...; funcions trigonomètriques: `sin(x), cos(x), tan(x)`, .... També hi ha funcions de conversió entre graus i radians: `degrees(x), radians(x)` i constants: `pi, e,` ... Per les funcions trigonomètriques l'angle s'ha d'expressar en radians.

Excepte per la biblioteca estàndard, per les altres biblioteques cal importar els mòduls corresponents. Hi ha bàsicament les dues formes següents de fer-ho:

| importar la biblioteca | crida a la funció |
|---|---|
| import math | math.sin(x) |
| from math import * | sin(x) |
| from math import sin | sin(x) |

Recomanem la primera, ja que posant el nom de la biblioteca davant del nom de la funció, ajuda a la legibilitat del codi.

Vegeu la documentació Python per un llistat complet i descripció detallada i actualitzada d'aquestes funcions. També es pot usar directament el shell de Python per obtenir una documentació resumida, com hem vist amb la biblioteca estàndard:

| | |
|---|---|
| llistar | `dir(math)` |
| documentar | `print(math.nomfuncio.__doc__)`<br>`help (math.nomfuncio)` |

Exemples:

```
>>> import math
>>> math.sqrt(25)
5.0
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.sin(30)
-0.9880316240928618
>>> math.sin(math.radians(30))
0.49999999999999994
>>> round(math.sin(math.radians(30)), 2)
0.5
```

## 1.13 Composició seqüencial

Un programa és una composició de sentències. La composició seqüencial és la més simple d'aquestes composicions. Es representa com:

```
sentència 1
sentència 2
........
sentència n
```

que vol dir que la sentència $i+1$ sempre s'executa després de la sentència $i$. Un programa amb aquesta composició executa una sentencia darrera l'altra en l'ordre en què apareixen i acaba quan ja no hi ha més sentències.

Exemple 1. Intercanvi de valor entre dues variables: versió 1

```
>>> a = 34      # línia 1
>>> b = 58      # línia 2
>>> aux = a     # línia 3
>>> a = b       # línia 4
>>> b = aux     # línia 5
>>> a, b
(58, 34)
```

Exemple 2. Intercanvi de valor entre dues variables: versió 2

```
>>> a = 34      # línia 1
>>> b = 58      # línia 2
>>> a = a+b     # línia 3
>>> b = a-b     # línia 4
>>> a = a-b     # línia 5
>>> a, b
(58, 34)
```

## 1.14 Comentaris

Els programes són executats pels computadors però també són llegits per persones. Per això, el codi ha de ser també entenedor per les persones. Un suggeriment per fer el codi més entenedor és usar noms de variables que tinguin significat i que reflecteixin el seu contingut. També cal que el programes continguin comentaris explicatius dirigits únicament a les persones i que no són executats pel computador. Aquests comentaris han d'anar precedits pel símbol `#` i no són processats per l'intèrpret de Python.

## 1.15 Traça

La traça d'un programa mostra la simulació de l'execució del programa. Concretament mostra el valor que van prenent les variables a l'inici i després de cada sentència. És una tècnica rudimentària però útil per tal d'entendre els resultats d'un programa.

Existeixen aplicacions que permeten fer una traça automàticament com Python tutor. Es pot fer una traça semi-automàticament, inserint sentències que usin la funció `print` que mostrarà el valor de les variables (**IMPORTANT:** cal esborrar sempre aquestes línies un cop el programa funciona). Finalment, també es pot fer una traça manual. En aquest cas, es representa amb una taula on les columnes corresponen a les variables i les files a les sentències.

En aquest exemple es mostren es traces manuals pels exemples 1 i 2 anteriors:

| Versió 1 | | | |
|---|---|---|---|
| línia | a | b | aux |
| 0 | ? | ? | ? |
| 1 | 4 | ? | ? |
| 2 | 4 | 5 | ? |
| 3 | 4 | 5 | 4 |
| 4 | 5 | 5 | 4 |
| 5 | 5 | 4 | 4 |

| Versió 2 | | |
|---|---|---|
| línia | a | b |
| 0 | ? | ? |
| 1 | 4 | ? |
| 2 | 4 | 5 |
| 3 | 9 | 5 |
| 4 | 9 | 4 |
| 5 | 5 | 4 |

## 1.16 Entrada/Sortida (Input/Output)

Actualment, les interfícies de les aplicacions són molt sofisticades, han de ser amigables i agradables. Han de guiar i ajudar als usuaris a usar-les de forma segura, eficaç i eficient.

De fet el codi necessari per una interfície pot ser de l'ordre d'un 80% del codi total. És a dir el codi del nucli de l'aplicació és d'un 20% del codi total.

La programació d'interfícies cau fora de l'àmbit d'aquest curs ja que aquest es concentra en el nucli de les aplicacions. Aquest curs es limita al disseny de funcions, que es defineixen indicant-ne les dades (paràmetres) i els resultats esperats (valors de retorn), com es veurà al capítol 2.

Per tant, en aquest apartat només es veuen succintament les funcions de la biblioteca estàndard `input` i `print`.

La funció `input` permet obtenir valors de l'usuari del programa i la funció `print` permet mostrar valors dels resultats. Aquestes dues funcions pràcticament no s'utilitzaran, tot i que la funció `print` ens podrà ajudar a corregir errors mostrant la traça d'una funció (vegeu l'apartat 1.15).

Exemples:

```
>>> x = input ('Entra el valor de x: ')
Entra el valor de x: 25
>>> x
'25'
>>> x = float(input ('Entra el valor de x: '))
Entra el valor de x: 25
>>> x
25.0
>>> n = 2
>>> print ('el doble de', n, 'és:', 2*n)
el doble de 2 és: 4
```

La funció `input` sempre retorna un string. Per obtenir un valor de tipus diferent, cal usar les corresponents funcions de conversió.

Al darrer exemple amb la funció `print`, es mostren 4 valors que s'indiquen separats per comes que corresponen a dos strings literals, el valor de la variable `n` i el resultat de l'expressió `2*n`.

## 1.17 Exercicis

1. Escriu al shell de Python el grup de sentències que duguin a terme els passos següents: assignar un valor enter a la variable `kgtot` que indica una quantitat expressada en kg i després obtenir les variables següents corresponents a la seva descomposició en tones mètriques (t), quintars mètrics (q), miriagrams (mag) i kilograms (kg).

```
>>> kgtot = 45391
>>> residu = kgtot % 1000
>>> q = residu //100
>>> residu = residu%100
>>> mag = residu //10
>>> kg = residu%10
>>> print(t,'tones,',q,'quintars,',mag,'miriagrams i',kg,'kg')
45 tones, 3 quintars, 9 miriagrams i 1 kg
```

2. Un cos es desplaça en moviment rectilini uniformement accelerat amb una acceleració $a$ expressada en $m/s^2$, una velocitat inicial $v0$ expressada en $m/s$ i una posició inicial $x0$ en $m$. Escriu al shell de Python el grup de sentències que duguin a terme els passos següents: assignar valors a les variables $a$, $v0$ i $x0$, i a la variable $x$ donada en $Km$ que indica la posició final (reals). Després obtenir la variable corresponent a la velocitat que tindrà quan es trobi a la posició $x$, expressada en $km/h$. Suggeriment: pots aplicar l'expressió següent:$v^2 - v_0^2 = 2a(x - x_0)$

```
>>> import math
>>> x0 = 134.0
>>> v0 = 11.2
>>> a = 2.3
>>> x = 4.4
>>> v = math.sqrt(v0**2 + 2*a*(x*1000 - x0))
>>> v = v/1000*3600
>>> print ('La velocitat actual és: ', round(v, 2), 'km/h')
La velocitat actual és:  505.91 km/h
```

# 2 Funcions

## 2.1 Definició

Una funció es una seqüència de sentències que realitzen una tasca concreta i que s'identifica amb un nom. El principal objectiu de les funcions és contribuir a l'organització dels programes en parts (grups de sentències) que resolen un determinat problema.

Cal tenir en compte que com més línies de codi hi ha en un programa, més fàcil és cometre errors i més difícil és mantenir-lo. Per tant, partir el codi en parts que resolen problemes específics i de mida petita redueix el temps de depuració i manteniment de les aplicacions informàtiques. A més, les funcions permeten aplicar generalització i reutilitzar codi.

La sintaxi per definir una funció és:

```
def nom_funció (paràmetres formals):          # capçalera
    sentències                                # cos
    return valors_de_retorn
```

En la definició d'una funció, Python usa dues paraules reservades `def` i `return`. La paraula `def` significa *defineix*. Els noms de les funcions segueixen les mateixes regles que els noms de variables. Després del nom de la funció hi ha una llista de paràmetres entre parèntesis. Aquesta llista pot ser buida o contenir un o més paràmetres separats per comes. Els parèntesis hi han de ser sempre.

La definició d'una funció és la primera **sentència composta** que veiem i totes elles segueixen un mateix patró:

- Una línia amb la **capçalera** que comença amb una paraula reservada i acaba amb el caràcter `:` (dos punts).
- Un **cos** amb una o més sentències **sagnades**. La guia d'estil Python recomana un sagnat de 4 espais. Atès que aquest és un error freqüent convé saber que en anglès sagnat és **indentation**.

Una funció rep informació a partir dels seus **paràmetres** i calcula i retorna resultats com a **valors de retorn**. Els paràmetres que apareixen a la capçalera d'una funció s'anomenen **paràmetres formals** perquè la funció es defineix formalment en funció d'ells. Aquests paràmetres són variables representades pels seus noms que segueixen les regles vistes per les variables generals. Els paràmetres formals no poden ser valors ni expressions. Serveixen per especificar les dades que la funció necessita per executar-se correctament.

La darrera sentència que s'executa en una funció és la sentència `return`. Aquesta sentència comença amb aquesta paraula reservada seguida d'una llista d'expressions de retorn. Aquesta llista pot ser buida o contenir un o més valors de retorn separats per comes. Els parèntesis no són necessaris. Python avalua les expressions de retorn i retorna els valors corresponents. Quan no hi ha cap valor de retorn la paraula `return` es pot ometre. En aquest cas, la funció retorna un valor especial anomenat `None`[1]. La sentència `return` és la darrera que s'executa en una funció i, per tant, si hi ha codi després d'aquesta sentència

[1] `None` és l'únic valor del tipus no escalar `NoneType`

no s'executarà mai (codi mort o inassolible). De totes maneres, aquesta sentència no té per què ser la que es trobi a la darrera línia.

Un fitxer que conté el codi d'una o més funcions s'anomena **mòdul**.

## 2.2 Crida a una funció

Una funció s'executa quan és cridada. La **sentència de crida** té la sintaxi següent:

```
variables = nom_funció (paràmetres actuals)
```

En el capítol anterior hem après a cridar funcions de la biblioteca estàndard o matemàtica. Els paràmetres que apareixen a la crida d'una funció s'anomenen **paràmetres actuals**. Els paràmetres actuals són els valors concrets als que s'aplica la funció. Quan es crida la funció, els paràmetres actuals de la sentència de crida s'assignen als paràmetres formals de la capçalera de la funció. Els paràmetres actuals poden ser qualsevol expressió.

Cada **paràmetre formal** es correspon amb un **paràmetre actual** i, per tant, n'hi ha d'haver el mateix nombre i aquesta correspondència ha de ser **posicional**, és a dir, al primer paràmetre formal se li assigna el primer paràmetre actual, al segon paràmetre formal se li assigna el segon paràmetre actual i així successivament. Aquesta mateixa regla posicional s'aplica als valors de retorn de la funció i les variables de la sentència de crida.

## 2.3 Variables locals i àmbit

Tota variable que apareix en el cos d'una funció i no és un paràmetre és una variable local.

Tota funció defineix un espai de noms o àmbit. Els paràmetres formals i les variables locals existeixen només dins l'àmbit de definició de la funció.

## 2.4 Exemples

Dissenya la funció `volum_esfera` que a partir del radi d'una esfera retorna el seu volum.

Exemples d'ús:

```
>>> volum_esfera (2.4) #volum d'unaesfera de radi = 2.4
57.90583579096705
>>> round(sphere_volume (2.4), 2) #arrodonit a 2 decimals
57.91

# volum de tres esferes de radi = 2.4
# crida a una funció dins d'una expressió
>>> tres_esferes = 3 * volum_esfera(2.4)
>>> round(tres_esferes, 2)
173.72
```

Aquesta funció té un paràmetre (`radi`) i un valor de retorn, el volum de l'esfera corresponent, obtingut a partir de l'expressió `4/3 * math.pi * radi ** 3`

```
import math

def volum_esfera (radi):
    return 4/3 * math.pi * radi ** 3
```

Dissenyem ara la funció `diventer` que a partir de dos enters corresponents al dividend i divisor, retorna el quocient i el residu.

```
def diventer(dividend, divisor):
   quocient, residu = dividend//divisor,dividend%divisor
   return quocient, residu
```

Desem aquesta funció en el fitxer (mòdul) de nom `funcdiventer.py`.

Exemple d'ús:

```
1  >>> dividend = 25
2  >>> divisor = 4
3  >>> quocient, residu = diventer(dividend, divisor)
4  >>> quocient, residu
5  (6, 1)
6  >>> diventer(67, 7)
7  (9, 4)
8  >>> a = 64
9  >>> b = 27
10 >>> diventer(a+5, b//3)
11 (7, 6)
12 >>> divmod(25,4)
13 (6, 1)
14 >>> divmod(67,7)
15 (9, 4)
16 >>> divmod(a+5, b//3)
17 (7, 6)
```

En aquest exemple, les línies 1 i 2 assignen valors a les variables `dividend` i `divisor`. La línia 3 crida a la funció `diventer` enviant aquestes variables com a paràmetres actuals i recull els resultats a les variables `quocient` i `residu` respectivament. En aquesta línia els paràmetres actuals són expressions on només hi ha el nom d'una variable. Aquesta variable ha d'estar inicialitzada per tal que la corresponent expressió es pugui avaluar. Cal notar que l'ordre entre paràmetres formals i actuals es respecta i també entre els valors de retorn i les variables que els recullen. A la línia 6 les expressions corresponents als paràmetres actuals són valors i a la línia 10 són expressions generals.

La biblioteca estàndard de Python disposa d'una funció que fa exactament aquesta tasca: la funció `divmod`. Les darreres línies mostren els resultats de cridar aquesta funció en lloc de la que hem dissenyat nosaltres. Com que és de la biblioteca estàndard no cal importar cap mòdul.

## 2.5 Flux de control

Les sentències d'un programa s'executen seqüencialment però a vegades l'execució pot saltar a una determinada sentència. Aquest fet s'anomena flux de control.

Una aplicació general consisteix en un gran nombre de funcions que poden cridar-se mútuament. Una funció pot cridar a altres funcions i pot ser cridada per altres funcions. Aquest fet es pot representar mitjançant un graf dirigit en el qual una aresta que va de la funció A a la funció B significa que la funció A crida la funció B.

Exemple: escriviu les tres funcions següents:

- funció `àrea_quadrat` que calcula l'àrea d'un quadrat donada la longitud del seu costat.
- funció `àrea_cercle` que calcula l'àrea d'un cercle donat el seu radi
- funció `dif_quadrat` que a partir d'un valor real corresponent al costat d'un quadrat, retorna la diferència entre l'àrea del quadrat i la del seu cercle inscrit. Aquesta funció ha de cridar les dues anteriors.

Codi de les funcions:

```
import math
def àrea_quadrat (costat):
    return costat **2
def àrea_cercle(radi):
    return math.pi * radi **2
def dif_quadrat(lcostat):
    aquadrat = àrea_quadrat (lcostat)
    acercle = àrea_cercle (lcostat/2)
    return aquadrat - acercle
```

Exemple d'ús:

```
>>> round(dif_quadrat(10.0), 2)
21.46
```

`dif_quadrat` avalua primer l'expressió de la primera sentència d'assignació i això implica un canvi del flux: el control es mou a la funció `àrea_quadrat` i al seu paràmetre formal `costat` se li assigna el valor del paràmetre actual `lcostat`. A continuació s'executa la sentència de la funció `àrea_quadrat`: s'avalua l'expressió `costat * 2` i es retorna el valor corresponent. Després el flux va cap enrere a la funció principal `dif_quadrat` i la seva primera sentència d'assignació s'executa completament: la variable `aquadrat` pren el valor retornat per la funció `àrea_quadrat`. Tot seguit, s'executa la següent sentència d'assignació de la funció `dif_quadrat`, però abans s'avalua l'expressió `lcostat/2` (paràmetre actual) i el seu valor s'assigna al paràmetre formal `radi` i el flux de control es desplaça a la funció `àrea_cercle`, etc.

En aquest punt, és molt clarificador executar aquest exemple amb l'aplicació Python tutor. Cal incloure la primera sentència que ha de ser executada: `aquad = dif_quadrat(10)` al final del codi. Primer, Python reconeix tots els elements que intervenen en aquest mòdul: la biblioteca matemàtica i les tres funcions. Observeu com el flux de control salta d'un punt a un altre quan s'executen les crides a funció i les sentències de retorn.

## 2.6 Exercicis

1. Dissenya la funció `unitats` que a partir d'una quantitat expressada en kg (`int`) retorni els valors corresponents a la seva descomposició en tones mètriques (t), quintars mètrics (q), miriagrams (mag) i kg. Desa aquesta funció al fitxer `unitatspes.py`.

```
def unitats(kgtot):
    t = kgtot //1000
    residu = kgtot%1000
    q = residu //100
    residu = residu%100
    mag = residu //10
    kg = residu%10
    return t, q, mag, kg
```

2. Un cos es desplaça en moviment rectilini uniformement accelerat amb una acceleració $a$ expressada en $m/s^2$, una velocitat inicial $v0$ expressada en $m/s$ i una posició inicial $x0$ en $m$ (reals). Dissenya la funció `mrua` que a partir d'aquestes dades i de la posició final $x$ donada en $Km$ (real), retorni la velocitat que tindrà quan es trobi a la posició $x$, expressada en $km/h$. Desa aquesta funció al fitxer `velmrua.py`. Suggeriment: pots aplicar l'expressió següent:$v^2 - v_0^2 = 2a(x - x_0)$

```
import math
def mrua(x0, v0, a, x):
    v = math.sqrt(v0**2 + 2*a*(x*1000 - x0))
    v = v/1000*3600
    return v
```

3. Dissenya la funció `és_múltiple(x, y)` que a partir de dos enters, $x$ i $y$, retorna `True` si $x$ és múltiple de $y$ o `False` altrament. Desa la funció en un fitxer de nom `funcesmultiple.py`.

```
def és_múltiple (x, y):
    resultat = x % y == 0
    return resultat
```

   Aquesta funció fa ús de l'operació mòdul de la divisió entera per determinar si `x` és múltiple de `y`: l¡expressió booleana `x%y==0` és equivalent a dir que `x` és múltiple de `y`. Notem que la funció es pot simplificar prescindint de la variable `resultat`.

```
def és_múltiple (x, y):
    return x % y == 0
```

   Alguns exemples d'ús:

```
>>> és_múltiple (4, 5)
False
>>> és_múltiple (10, 2)
True
>>> és_múltiple (10, 5)
True
```

4. Dissenya les funcions següents en el mòdul `functipuscaracter.py`

- La funció `majúscula` que donat un string que conté un sol caràcter retorna `True` si aquest caràcter és una lletra majúscula i `False` en cas contrari.
- La funció `minúscula` que donat un string que conté un sol caràcter retorna `True` si aquest caràcter és una lletra minúscula i `False` en cas contrari.
- La funció `dígit` que donat un string que conté un sol caràcter retorna `True` si aquest caràcter és un dígit i `False` en cas contrari.
- La funció `altres` que donat un string que conté un sol caràcter retorna `True` si aquest caràcter no és una lletra majúscula ni una lletra minúscula ni un dígit i `False` en cas contrari. Aquesta funció ha de cridar les 3 funcions anteriors.

```
def majúscula (c):
    return  'A' <= c <= 'Z'

def minúscula (c):
    return  'a' <= c <= 'z'

def dígit (c):
    return '0' <= c <= '9'

def altres (c):
    return not (majúscula (c) or minúscula(c) or dígit(c))
```

En aquestes funcions ens hem basat en la propietats de la codificació ASCII on les lletres majúscules constitueixen un interval ($['A','Z']$) així com les lletres minúscules ($['a','z']$) i els dígits ($['0','1']$).

Alguns exemples d'ús:

```
>>> majúscula ('A')
True
>>> majúscula ('e')
False
>>> minúscula ('B')
False
>>> minúscula ('t')
True
>>> dígit ('8')
True
>>> dígit ('p')
False
>>> altres ('k')
False
>>> altres (':')
True
```

5. Dissenya la funció `explica_suma` que donats dos enters d'un màxim de dues xifres, retorna un string amb les explicacions de com es duu a terme la suma. Desa aquesta funció al mòdul `funcexplicasuma.py`. Aquesta funció hauria de passar la prova següent:

```
>>> explica_suma(34, 88)
'La suma de les unitats, 4 i 8, és 2 i en portem 1. La suma
de les desenes, 3 i 8, més el que portem, 1, és 2 i en
portem 1. Les centenes del resultat són el que portem, 1.'
```

Nota: a l'string resultat no hi ha salts de línia. A l'exemple aquest string s'ha tallat en línies per tal que es pugui veure el resultat.

Pel disseny d'aquesta funció, dissenyarem primer dues funcions que donat un enter de dues xifres retornin respectivament la xifra corresponent a les unitats i la corresponent a les desenes que fan ús també de les operacions de la divisió i mòdul entre enters.

```
def unitats(a):
    return a%10

def desenes(a):
    return a//10

def explica_suma(a,b):
    su = unitats(a)+unitats(b)
    uu=unitats(su)
    du=desenes(su)
    sd=desenes(a)+desenes(b)+du
    ud=unitats(sd)
    dd=desenes(sd)
    s1='La suma de les unitats, '+str(unitats(a))+\
       ' i '+str(unitats(b))+', és '+str(uu)+\
       ' i en portem '+str(du)+'. '
    s2='La suma de les desenes, '+ str(desenes(a))+\
       ' i '+str(desenes(b))+', més el que portem, '\
      +str(du)+', és '+str(ud)+ ' i en portem '+\
       str(dd) + '. '
    s3='Les centenes del resultat són el '+ \
           'que portem, '+str(dd) + '.'
    return s1+s2+s3
```

# 3 Composició condicional

## 3.1 Definició

La composició condicional (sentència `if`) permet que el programa pugui avaluar una condició i modificar el flux de control segons el resultat.

Una condició es representa mitjançant una expressió booleana que ha de ser avaluada i, en funció del valor resultant, `True` or `False`, el programa executarà un grup de sentències o un altre.

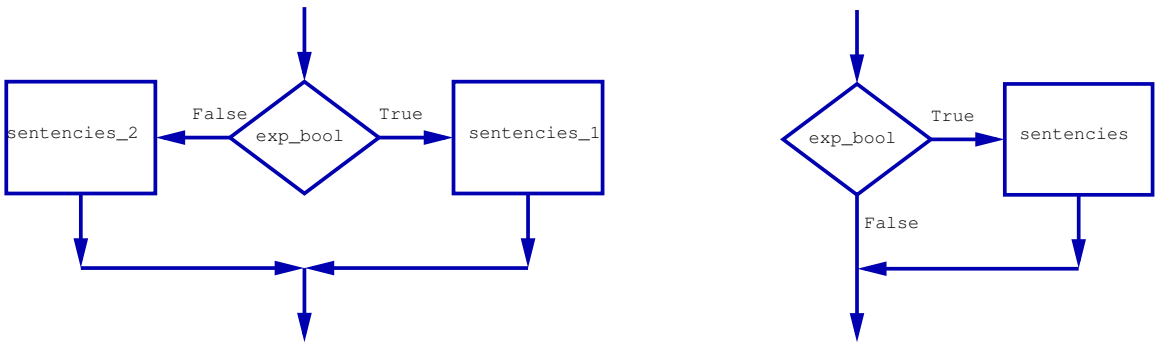
## 3.2 Sintaxi: casos simples

A continuació es mostren els casos amb només una condició:

```
if exp_bool:                          if exp_bool:
    sentències1                           sentències
else:
    sentències2
```

La composició condicional és una sentència composta que segueix el patró descrit per les funcions. La sintaxi de l'esquerra inclou dues línies que comencen amb dues paraules reservades, `if` i `else`, respectivament i acaben amb el caràcter 'dos punts'. Totes dues inclouen un bloc de sentències sagnades als que podem anomenar **branques**.

A les figures següents es mostren els diagrames de flux d'aquestes dues composicions seqüencials.




La composició de l'esquerra mostra el cas més general i s'executen els passos següents:

- s'avalua l'expressió booleana
- si el resultat és `True` s'executa el bloc `sentències1`
- si el resultat és `False` s'executa el bloc `sentències2`

En tots dos casos, després de la sentència condicional, l'execució del programa continua amb les sentències que hi ha després de la sentència condicional.

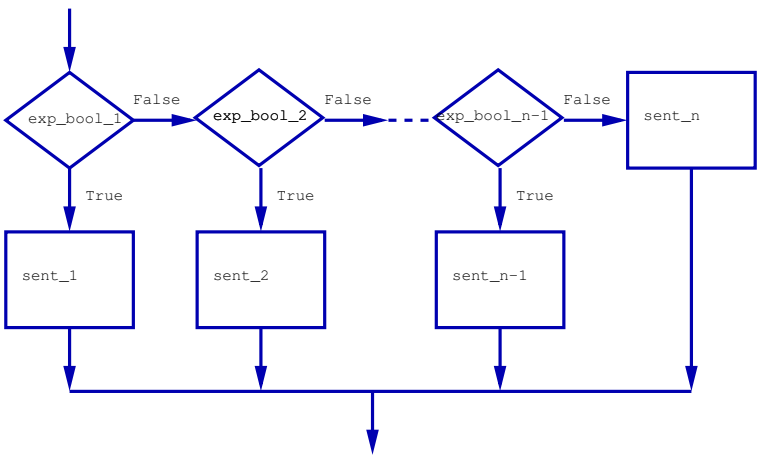
La composició condicional que es mostra a la dreta és un cas particular d'aquest en el que si l'expressió booleana és `False` no s'executa cap sentència. Alternativament es pot usar explícitament la sentència `pass`, que és la sentència per indicar que no s'executa res.

```
if exp_bool:
    sentències
else:
    pass
```

## 3.3 Sintaxi: cas general

A continuació es mostra el cas general amb diverses condicions encadenades:

```
if exp_bool_1:
   sent_1
elif exp_bool_2:
   sent_2
.........
elif exp_bool_n-1:
   sent_n-1
else:
   sent_n
```




Aquesta sintaxi indica que primer s'avalua l'expressió booleana `exp_bool_1`. Si el resultat és `True` s'executa el bloc de sentències `sent_1` i se surt de la composició condicional. Si `exp_bool_1` val `False` aleshores s'avalua l'expressió booleana `exp_bool_2`. Si aquesta és `True` s'executa el bloc de sentències `sent_2` i se surt de la composició condicional. I així successivament fins a l'expressió booleana `exp_bool_n-1`. Quan aquesta s'avalua a `True` s'executa el bloc de sentències `sent_n-1` i se surt de la composició condicional i si val `False` s'executa el bloc de sentències `sent_n` i se surt de la composició condicional.

A continuació es mostra el funcionament d'aquesta composició amb un exemple.

**Exercici 1**

Un centre esportiu ofereix entrenaments en natació. El centre només entrena nois i noies entre les edats de 11 i 19 anys (ambdós inclosos) i la categoria oficial per aquestes edats és la següent: edats de 11 i 12 anys: *aleví*, edats de 13 i 14 anys: *infantil*, edats de 15 i 16 anys: *cadet* i edats de 17, 18 i 19 anys *juvenil*. Dissenya una funció que donada l'edat (enter) d'un aspirant a matricular-se al centre esportiu, retorni un string indicant la categoria en majúscules, en cas que l'aspirant es pugui matricular o bé l'string `'SENSE SERVEI'` en cas contrari.

**Disseny**

Per tal de mostrar el funcionament de l'esquema general de composició condicional de Python, en aquest disseny es consideraran els intervals que defineixen cada categoria.

El següent disseny és correcte:

```
def natació1 (edat):
    if edat < 11:
        s = 'SENSE SERVEI'
    elif edat <= 12:
        s = 'ALEVÍ'
    elif edat <= 14:
        s = 'INFANTIL'
    elif edat <= 16:
        s = 'CADET'
    elif edat <= 19:
        s = 'JUVENIL'
    else:
        s = 'SENSE SERVEI'
    return s
```

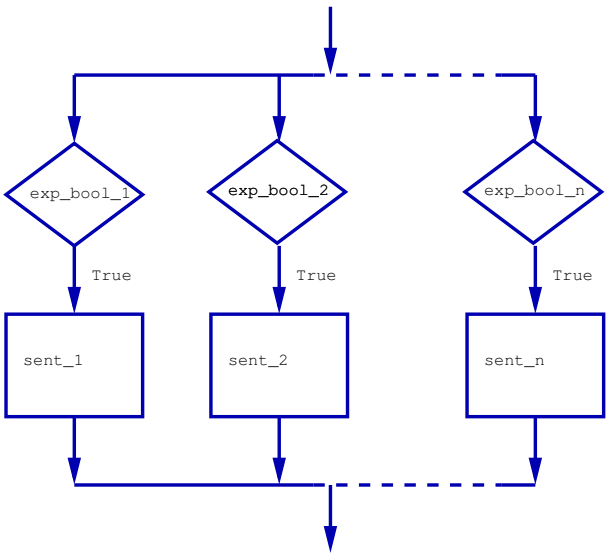
Aquest disseny és correcte degut a que hem tingut cura de posar les branques de la composició condicional en ordre creixent d'edat. Si no ho haguéssim fet aixi, el disseny seria incorrecte.

El següent disseny és **incorrecte**. Hem intercanviat l'ordre entre les categories *aleví* i *infantil* i aleshores classifica com a *infantil* les edats de 11, 12, 13 i 14 anys i no en classifica cap com a *aleví*. Per tant, si usem l'esquema general de Python cal anar en compte en l'ordre en que es posen les condicions.

```
def natació2 (edat):
    if edat < 11:
        s = 'SENSE SERVEI'
    elif edat <= 14:
        s = 'INFANTIL'
    elif edat <= 12:
        s = 'ALEVÍ'
    elif edat <= 16:
        s = 'CADET'
    elif edat <= 19:
        s = 'JUVENIL'
    else:
        s = 'SENSE SERVEI'
    return s
```

## 3.4 Composició condicional general: esquema teòric

El següent diagrama mostra l'esquema teòric de la composició condicional general:




En aquest cas, cal que el conjunt de les condicions constitueixin una partició de la unitat, és a dir, sempre s'ha de complir una i només una condició.

Veiem l'anterior exemple dissenyat usant l'esquema teòric:

```
def natació3 (edat):
    if edat < 11:
        s= 'SENSE SERVEI'
    elif 13 <= edat <= 14:
        s= 'INFANTIL'
    elif 11 <= edat <= 12:
        s= 'ALEVÍ'
    elif 15 <= edat <= 16:
        s = 'CADET'
    elif 17 <= edat <= 19:
        s = 'JUVENIL'
    else:
        s = 'SENSE SERVEI'
    return s
```

En aquest disseny no hem respectat l'ordre en les edats, però és un disseny correcte perquè segueix l'esquema teòric on les expressions booleanes de les branques són una partició de la unitat.

Per tant, es recomana aplicar sempre l'esquema teòric.

## 3.5 Composicions condicionals imbricades

Les sentències dins les branques d'una composició condicional, poden ser també composicions condicionals i, en aquest cas, se'ls anomena composicions condicionals **imbricades**.

El següent disseny és una altra possible solució correcta de l'anterior enunciat on hi apareixen composicions condicionals imbricades:

```
def natació4 (edat):
    if 11 <= edat <= 19:
```

```
        if 13 <= edat <= 14:
            s= 'INFANTIL'
        elif 11 <= edat <= 12:
            s= 'ALEVÍ'
        elif 15 <= edat <= 16:
            s = 'CADET'
        else:
            s = 'JUVENIL'
    else:
        s = 'SENSE SERVEI'
    return s
```

En aquest cas hi ha una composició condicional que distingeix entre les edats per les que el centre dóna servei i les que no. A la branca corresponent al cas en que es dóna servei hi ha una altra composició condicional que classifica segons edats.

Com exercici escriu les expressions booleanes que queden implícites en els dos `else` del disseny anterior.

## 3.6 Funcions amb més d'una sentència `return`

En tots els anteriors dissenys només hi ha una sentència `return`. El resultat a les diferents branques s'assigna a una variable que es retorna al final de la composició condicional.

Es pot prescindir d'aquesta variable i retornar els valors directament a cada branca de la composició condicional, tal com es mostra al disseny següent. Ara bé, cal recordar que una funció encara que en el disseny hi aparegui la sentència `return` més d'un cop, aquesta només s'ha d'executar una vegada.

```
def natació5 (edat):
    if 11 <= edat <= 19:
        if 13 <= edat <= 14:
            return 'INFANTIL'
        elif 11 <= edat <= 12:
            return 'ALEVÍ'
        elif 15 <= edat <= 16:
            return 'CADET'
        else:
            return 'JUVENIL'
    else:
        return 'SENSE SERVEI'
```

## 3.7 Funcions booleanes

Una **funció booleana** és una funció que retorna un valor booleà.

**Exercici 2**

Donat un interval tancat d'edats `[e1, e2]` i una altra edat `e3`, dissenya una funció que retorni un booleà que valgui `True` en el cas que `e3` estigui dins l'interval i `False` en cas

contrari.

**Disseny**

```
def interval1 (e1, e2, e3):
    if e1 <= e3 <= e2:
        resultat = True
    else:
        resultat = False
    return resultat
```

En aquest disseny hem traduït de manera gairebé literal el que diu l'enunciat a Python. Tot i que aquest disseny funciona correctament, cal veure que la composició condicional es pot simplificar. La variable resultat, que és una variable booleana, pot prendre per valor l'expressió booleana que hi ha a la composició condicional.

```
def interval2 (e1, e2, e3):
    resultat = e1 <= e3 <= e2
    return resultat
```

En tots dos casos podem prescindir de la variable `resultat`. En el darrer, la funció pot retornar l'expressió booleana directament:

```
def interval3 (e1, e2, e3):
    return e1 <= e3 <= e2
```

## 3.8 Exercicis

1. Dissenya una funció de nom `mínim` que rep dos enters `x` i `y` i retorna el mínim de tots dos.

   ```
   def mínim (x, y):
       if x < y:
           return x
       else:
           return y
   ```

2. Una botiga ofereix la següent promoció als seus clients: per la segona unitat de producte, aquesta es rebaixa un 30% i si es compren tres unitats o més de producte, tota la compra es rebaixa un 25%. A més, si el preu del producte és de 50 € o més hi ha una rebaixa addicional d'un 5% sobre la quantitat resultant en tots dos casos. Si només es compra una unitat no s'aplica cap tipus de descompte. Dissenya la funció `promo` que donat el preu i la quantitat d'unitats d'un producte retorni la quantitat a pagar sense descompte i amb descompte.

   ```
   def promo(preu, unitats):
       if unitats == 1:
           return preu, preu
       else:
           if unitats == 2:
               quant1 = preu * 1.7
           else:
   ```

```
            quant1 = preu *unitats * 0.75
        if preu >= 50:
            quant1 = quant1 * 0.95
        return preu*unitats, quant1
```

3. Un hotel ofereix una excursió que inclou un recorregut en vaixell i un en tren. El client pot triar entre començar pel vaixell i continuar en tren o al revés i també entre fer l'excursió al matí o a la tarda. L'excursió té un preu al que s'aplica un descompte general d'un tant per cent donat. Ara bé, si es comença l'excursió amb vaixell al descompte general se li afegeix un 5% i si es fa a la tarda un 10% addicional. Dissenya la funció `excursió` que donats dos strings, un indicant com es vol començar l'excursió, 'tren' o 'vaixell', i l'altre indicant si es vol fer 'matí' o 'tarda', i dos reals, el preu base i el descompte general en tant per cent, retorni el preu final de l'excursió.

```
def excursió(inici, hora, preu, d):
    # els descomptes són acumulatius
    if inici == 'vaixell':
        d = d + 5
    if hora == 'tarda':
        d =d + 10
    return preu * (1-d/100)
```

4. Dissenya primer dues funcions: la funció `cartesianes` que converteix de coordenades polars a cartesianes i la funció `polars` que fa la conversió inversa.

   A continuació dissenya la funció `vectors` que rep dos vectors en coordenades polars (l'angle en graus). Si els angles de tots dos vectors són inferiors a 45° o tots dos són iguals o superiors a 45°, la funció retorna suma dels vectors. Si un angle és inferior a 45° i l'altre és igual o superior a 45° la funció retorna el vector corresponent a l'angle més gran. Si els angles fossin iguals es retorna el vector amb el mòdul més gran. Aquesta funció ha de retornar el vector en coordenades polars i l'angle en graus. Suggeriment: usar les funcions `cartesianes` i `polars` i fer la suma amb coordenades cartesianes. Per simplificar poden considerar que tots els vectors són al primer quadrant.

```
import math

def cartesianes (mod, angle):
    angle = math.radians (angle)
    x = mod * math.cos(angle)
    y = mod * math.sin(angle)
    return x, y

def polars (x, y):
    mod = math.sqrt (x**2 + y **2)
    angle = math.atan(y/x)
    angle = math.degrees(angle)
    return mod, angle

def vectors (m1, a1, m2, a2):
```

```
    x1, y1 = cartesianes (m1, a1)
    x2, y2 = cartesianes (m2, a2)
    if (a1 < 45 and a2 < 45) or (a1 >= 45 and a2 >= 45):
        x3, y3 = x1+x2, y1+y2
        m3, a3 = polars (x3, y3)
        return m3, a3
    else:
        if a1 > a2:
            return m1, a1
        elif a1 < a2:
            return m2, a2
        else:
            if m1 > m2:
                return m1, a1
            else:
                return m2, a2
```

# 4 Strings

## 4.1 Definició

Els tipus de dades no escalars estan compostos de parts més petites i s'anomenen també **tipus de dades compostos**. Els strings, les llistes, els tuples i els diccionaris són tipus de dades compostos, que es veuran aquest curs.

Els strings (tipus `str`) són estructures de dades compostes per una agregació ordenada d'elements més simples (caràcters): són un tipus **seqüencial**. Els strings es poden operar en bloc com s'ha vist a la introducció, però també permeten l'accés als caràcters individuals tal com es veurà en aquest capítol.

Els strings es representen tancant els caràcters entre cometes simples o dobles: $'c_0 c_1 \dots c'_n$, on $c_i$ és un caràcter.

Hi ha alguns strings d'un caràcter que cal conèixer bé. Per un costat, el caràcter NUL es representa com `''` (entre les cometes no hi ha res) i permet representar un string buit. Cal no confondre'l visualment amb el caràcter ESPAI `' '` (entre les cometes hi ha un espai). Per altra banda hi ha els caràcters de control o seqüències d'escapada que són caràcters que representen accions i s'escriuen amb un caràcter precedit pel caràcter barra inversa (`'\'`). D'aquests caràcters el que més farem servir és el caràcter que representa el salt de línia `'\n'`. També hi ha els caràcters que representen el tabulador `'\t'` o el de retrocés `'\b'`, entre d'altres. Encara que en la seva representació gràfica hi ha dos caràcters, cal tenir present que són considerats com un sol caràcter.

A la codificació UTF-8 hi ha els caràcters corresponents a lletres accentuades (`à, é,` ...), lletres pròpies d'alfabets diferents de l'anglès (`ç, ñ,` ...) i altres caràcters com €.

Exemples:

```
>>> a = 'Fa un dia molt bonic'
>>> b = "L'aire és fresc"
>>> b
"L'aire és fresc"
>>> c = 'aquell dispositiu val 22.22€'
>>> c
'aquell dispositiu val 22.22€'
>>> el_no_res = ''
>>> el_no_res
''
>>> d = 'comanda: 200 €\niva(10%): 20 €\ntotal:   220 €'
>>> print(d)
comanda: 200 €
iva(10%): 20 €
total:   220 €
```

Observeu que en l'string `d` hi apareix el caràcter salt de línia i com queda representat en fer `print(d)`.

## 4.2 Operacions

Fins ara hem vist les operacions que operem amb strings com un sol bloc: concatenació (+) i repetició (*). També s'ha vist que els hi són aplicables els operadors de comparació: `==`, `!=`, `<`, `>`, `<=`, `>=` i que aquests es basen en l'ordre lexicogràfic.

La funció `len` de la biblioteca estàndard permet conèixer la longitud d'un string, és a dir, el nombre de caràcters que té.

Exemples

```
>>> a = "L'aire " + 'fresc'
>>> a
"L'aire fresc"
>>> len (a)
12   # compta tots els caràcters
>>> 'no '*3
'no no no ' # l'espai també es repeteix 3 cops
>>> len('') # caràcter NULL (la seva longitud és 0)
0
>>> len('\n') # salt de línia: es representa amb 2 caràcters
1             # però és un sol caràcter: la longitud és 1
>>> len('banc') < len('ballar') # comparació de les longituds
True
>>> 'banc' < 'ballar' # ordre lexicogràfic
False
>>> '23' < '123000'   # ordre lexicogràfic
False
```

## 4.3 Indexació

Es pot accedir a un caràcter individual d'un string mitjançant l'operador d'indexació. La sintaxi Python usa claudàtors, `[]`, i un índex enter. L'índex enter indica la posició del caràcter en l'string.

Si `a` és un string i `index` una expressió entera, l'expressió `a[index]` indica el caràcter de `a` a la posició `index`.

`index` pot ser una variable, valor o expressió (enter). Els índexs d'un string, `a` amb `n = len(a)`, compleixen la condició següent: $0 \leq index < len(a)$. Per tant el primer caràcter té índex `0` i el darrer caràcter té índex `n-1`, és a dir, `n` caràcters en total. Un error típic de principiant és escriure l'expressió `a[n]` que dóna l'error: `IndexError: string index out of range`, ja que el darrer element de `a` és `a[n-1]`.

Els índexs també poden ser negatius: $-len(a) \leq index < 0$. Compten cap enrere des del final fins a l'inici de l'string.

Vegem, per exemple, els índexs positius i negatius per l'string `'mandarina'`:

```
caràcters:            m  a  n  d  a  r  i  n  a
índexos positius:     0  1  2  3  4  5  6  7  8
índexos negatius:    -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Exemples:

```
>>> a = 'esmaperdut'
>>> len (a)
10
>>> a[0]
'e'
>>> a[-1]   # darrer caràcter: forma recomenada
't'
>>> a[-2]   # penúltim caràcter
'u'
>>> a[2]
'm'
>>> a[-9]
's'
list(enumerate(a))  # parelles (índex, element)
[(0, 'e'), (1, 's'), (2, 'm'), (3, 'a'), (4, 'p'), (5, 'e'),
(6, 'r'), (7, 'd'), (8, 'u'), (9, 't')]
```

## 4.4 Segmentació

Una llesca o segment (*slice*, en anglès) és un substring d'un string. Si `a` és un string, la sintaxi per accedir a un segment és: `a[pos1:pos2]`, $0 \leq pos1 < pos2 < len(a)$, pel cas d'índexs positius. S'interpreta com el substring format pels caràcters que van des del que està a la posició `pos1` fins just el d'abans del que està a la posició `pos2`. Es compleix que `len(a[pos1:pos2])` is `pos2-pos1`.

Si s'omet el primer índex (abans dels dos punts, el segment comença a l'inici de l'string (s'interpreta el valor com `0`). Si s'omet el segon índex, el segment s'estén fins al final de l'string (s'interpreta el valor com `len(a)`). A més, si el valor del segon índex és més gran que `len(a)`, el segment també s'estendrà fins al final (no donarà l'error "out of range" que donaria en l'operació d'indexació).

Aquesta notació s'estén amb un tercer element, `a[pos1:pos2:pas]`, $0 \leq pos1 < pos2 < len(a)$. S'interpreta com el substring format pels caràcters que van des de `pos1` fins just abans de `pos2` però amb el `pas` indicat. El valor per defecte de `pas` és 1.

`pos1`, `pos2` i `pas` són expressions enteres i poden ser positives o negatives.

Exemples:

```
>>> a = 'esmaperdut'
>>> a[0:4] # del caràcter 0 (primer) fins just abans del 4
'esma'
>>> a[:4]
'esma'
>>> a[4:len(a)]
'perdut'
>>> a[4:]
'perdut'
```

```
>>> a[4:200]  # pos2 pot ser més gran que len(a)
'perdut'
>>> a[-6:]
'perdut'
>>> a[-len(a):-6]
'esma'
>>> a[:6:2]
'emp'
>>> a[-6:-9:-1]
'pam'
>>> a[::-1]   # sintaxi per capgirar un string
'tudrepamse'
>>> b = 'lalalalalalala'
>>> b[::2]
'lllllll'
>>> b[1::2]
'aaaaaaa'
>>> c = 'enciams'
>>> c[:-1] # sintaxi per seleccionar-los tots menys el darrer
'enciam'
```

## 4.5 Operadors de pertinença

Els operadors de pertinença `in` i `not in` permeten determinar si un string és un substring d'un altre. Els dos operands d'aquests operadors són strings i el resultat és un booleà. Tot string és un substring d'ell mateix i l'string buit és un substring de qualsevol altre string. Notació:

```
string1 in string2 --> booleà
string1 not in string2 --> booleà
```

Exemples:

```
>>> a = 'mandarina'
>>> 'man' in a
True
>>> 'mani' in a
False
>>> 'ari' not in a
False
>>> 'farina' not in a
True
>>> 'mandarina' in a
True
```

Com ja s'ha vist a la introducció, a la taula ASCII hi ha un interval amb les lletres majúscules, un altre amb les minúscules i un altre amb els dígits. Per tant, l'expressió que indica si un caràcter `c` és una lletra majúscula pot fer ús d'aquest interval: `'A' <= c`

`<= 'Z'`. Ara bé, l'expressió que indica si un caràcter `c` és una vocal no pot seguir aquesta estratègia.

Les operacions de pertinença permeten usar una altra estratègia: definir el conjunt de caràcters adequat (vocals, nucleòtids ADN, etc.) i comprovar si un determinat caràcter pertany a aquest conjunt o no. Aquesta estratègia es pot aplicar a qualsevol conjunt de caràcters i es pot estendre als tipus llista i tuple que es veuran més endavant. Exemples:

```
>>> vocals = 'aeiouAEIOU'
>>> 'a' in vocals
True
>>> 'g' in vocals
False
>>> 'U' in vocals
True
>>> 'B' in vocals
False
>>> adn = 'ACGT'
>>> 'A' in adn
True
>>> 'E' in adn
False
>>> puntuació = '.,;:' # punt, coma, punt i coma, dos punts
>>> ':' in puntuació
True
>>> '9' in puntuació
False
>>> 'z' not in puntuació
```

## 4.6 Immutabilitat

Els strings són immutables, és a dir, no es poden modificar. Si intentem modificar un element d'un string, Python produeix un error d'execució:

```
>>> a ='correcte'
>>> a[-1] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Podem obtenir un nou string resultant d'aplicar canvis a un altre:

```
>>> a = "l'enunciat és correcte"
>>> a = 'la resposta' + a[10:-1] + 'a'
>>> a  # el nou string a s'obté a partir de l'antic string a
'la resposta és correcta'
```

## 4.7 Funcions de la biblioteca estàndard

Hi ha diverses funcions de la biblioteca estàndard que s'apliquen a strings: `len`, `min` i `max`. Exemples:

```
>>> max('Informatica')
't'
>>> min('Informatica')
'I'
>>> max ('vaixell', 'cotxe', 'moto', 'bicicleta')
'vaixell'
>>> min ('vaixell', 'cotxe', 'moto', 'bicicleta')
'bicicleta'
```

Recordem que els strings segueixen l'ordre lèxicografic (que és una extensió de l'ordre alfabètic) i que l'string màxim no és el que té més caràcters sinó el darrer considerant aquest ordre lèxicografic.

## 4.8 Tipus `str`: mètodes

Python ofereix diversos tipus de dades amb els corresponents mètodes. Un d'ells és el tipus `str` (string). Un mètode és una funció que s'aplica sobre un objecte d'un determinat tipus. Un mètode també es pot cridar (o invocar) com una funció però la sintaxi és lleugerament diferent: fa servir la **notació del punt** i el paràmetre al qual s'aplica el mètode (paràmetre principal) es posa davant del nom del mètode i separat per un punt. La resta de paràmetres (paràmetres secundaris) apareixen darrera el nom del mètode i entre parèntesis com en les funcions:

`nom_paràmetre_str.nom_mètode_str(paràmetres actuals)`

- `nom_paràmetre_str`: nom de la variable de tipus `str` a la que s'aplica el mètode
- `nom_mètode_str`: nom del mètode a aplicar
- `paràmetres actuals`: llista dels paràmetres secundaris

En aquest apartat es fa una breu introducció d'alguns mètodes i es recomana consultar la documentació Python per tenir informació completa i al dia. També es pot usar el shell de Python que mostra una breu documentació, usant les funcions `dir` i `help` de la biblioteca estàndard. A continuació es mostra la documentació del shell de Python per alguns mètodes del tipus `str` així com alguns exemples. La funció `dir` llista tots els mètodes del tipus `str`: `dir(str)`.

**Mètode** `count`

```
>>> help(str.count)
S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of
substring sub in string S[start:end].
Optional arguxments start and end are
interpreted as in slice notation.
```

Exemples:

```
>>> a = 'presseguer'
>>> a.count('e')
3
>>> b = 'cacahuet'
>>> b.count('ca')
2
>>> daus ='12666666543233326666'
>>> daus.count('66')
4
>>> daus1 = '166266366'
>>> daus1.count('66')
3
```

### Mètode `find`

El mètode `find` fa un procés invers al de l'operador d'indexació. Aquest a partir de l'índex dóna l'element corresponent i el mètode `find` retorna l'índex corresponent a un caràcter.

```
>>> help(str.find)
S.find(sub [,start [,end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within s[start:end].
Optional arguments start and end are interpreted as in
slice notation. Return -1 on failure.
```

Exemples:

```
>>> c = 'Hola que tal'
>>> c.find('que')
5
>>> c[5]
'q'
>>> c.find('adeu')
-1
>>> c.find('a') # index de la primera 'a'
3
```

### Mètode `replace`

```
>>> help(str.replace)
S.replace(old, new[, count]) -> string

Return a copy of string S with all occurrences of substring
old replaced by new. If the optional argument count is given,
only the first count occurrences are replaced.
```

Exemples:

```
>>> d = 'la porteria es quadrada'
>>> e = d.replace('la', 'el')
```

```
>>> f = e.replace('quadrada', 'redondo')
>>> f
'el porteria es redondo'
>>> d
'la porteria es quadrada'
```

Els strings són immutables i el mètode replace segueix aquesta propietat. A l'exemple anterior, la variable `d` no es modifica a l'expressió `d.replace('la', 'el')`. Tampoc es modifica la variable `e` a l'expressió `e.replace('quadrada', 'redondo')`.

**Mètodes per classificar caràcters**

Existeixen mètodes que permeten determinar el tipus d'un string: `isalnum`, `isalpha`, `isdigit`, `islower`, `isspace`, `isupper` i d'altres que permeten canviar de majúscules a minúscules i viceversa: `lower`, `upper`, `swapcase`.

A continuació es transcriu la petita ajuda que mostra el shell de Python per alguns d'aquests mètodes:

```
>>> help(str.isalnum)
S.isalnum() -> bool

Return True if all characters in S are alphanumeric and
there is at least one character in S, False otherwise.

>>> help(str.swapcase)
S.swapcase() -> string

Return a copy of the string S with uppercase characters
converted to lowercase and vice versa.
```

Exemples:

```
>>> a = 'Hola que tal'
>>> b = 'Informatica'
>>> c = 'USA'
>>> d = 'a'
>>> e = '4'
>>> f = '?'
>>> a.isalnum()
False
>>> b.isalnum()
True
>>> f.isalnum()
False
```

```
>>> f.isdigit()
False
>>> e.isdigit()
True
>>> a.swapcase()
'hOLA QUE TAL'
>>> c.lower()
```

```
'usa'
>>> b.upper()
'INFORMATICA'
```

### Més mètodes

Vegem altres mètodes d'strings (`ljust, center, rjust, endswith, startswith, zfill`) amb exemples:

```
>>> help(str.ljust)
  S.ljust(width[, fillchar]) -> str

  Return S left-justified in a Unicode string of length
  width. Padding is done using the specified fill character
  (default is a space).
```

```
>>> help(str.endswith)
    S.endswith(suffix[, start[, end]]) -> bool

    Return True if S ends with the specified suffix,
    False otherwise.
    With optional start, test S beginning at that position.
    With optional end, stop comparing S at that position.
    suffix can also be a tuple of strings to try.

>>> help(str.zfill)
    S.zfill(width) -> str

    Pad a numeric string S with zeros on the left, to fill
    a field of the specified width.
    The string S is never truncated.
```

Exemples:

```
>>> a = 'Hellow !!!'
>>> a.ljust(20, ' ')
'Hellow !!!          '
>>> a.center(20, ' ')
'     Hellow !!!     '
>>> a.rjust(20, ' ')
'          Hellow !!!'
>>> a.ljust(20, '-')
'Hellow !!!----------'
```

```
>>> b = 'telecomunicacions'
>>> prefix = 'tele'
>>> sufix = 'cions'
>>> b.startswith(prefix) # equivalent a l'expressió següent
True
>>> b[:len(prefix)] == prefix
True
>>> b.endswith(sufix) # equivalent a l'expressió següent
True
>>> b[-len(sufix):] == sufix
True
>>> '987'.zfill(6) # omple amb zeros per l'esquerra
'000987'
```

## 4.9 Exercicis

1. L'any 2000 s'implanta un nou sistema de matriculació de cotxes a Espanya que es basa en una codificació de 3 lletres i 4 dígits. Les lletres només poden ser les 20 següents: `B, C, D, F, G, H, J, K, L, M, N, P, R, S, T, V, W, X, Y, Z`. En aquesta codificació, el grup de tres lletres és en base 20 i té més pes que el grup de 4 dígits (en base 10).

   (a) Dissenyeu la funció `val_matrícula` que a partir d'una matrícula calcula la seva posició absoluta des de l'inici de la nova matriculació.

   (b) Dissenyeu la funció `matrícules` que a partir de dos strings que representen dues matrícules, retorni el nombre de cotxes que hi ha entre elles. Podem suposar que la primera matrícula donada és sempre anterior a la segona. La funció `matrícules` ha de cridar la funció `val_matrícula`.

   Exemples:

   ```
   >>> val_matrícula ('0000-BBB')
   0
   >>> val_matrícula ('6589-BBB')
   6589
   >>> val_matrícula ('9999-BBB')
   9999
   >>> val_matrícula ('0000-BBC')
   10000
   >>> val_matrícula ('9999-ZZZ')
   79999999
   >>> matrícules ('0000-BBB', '9999-ZZZ')
   79999999
   >>> matrícules ('0000-BBB', '9999-BBB')
   9999
   >>> matrícules ('0000-BBB', '0000-BBC')
   10000
   >>> matrícules ('9999-KLZ', '0000-KMB')
   1
   ```

Solució:

```
def val_matrícula (mat):
    lletres = 'BCDFGHJKLMNPRSTVWXYZ'
    p1 = lletres.find(mat[5])
    p2 = lletres.find(mat[6])
    p3 = lletres.find(mat[7])
    return (p1*20**2 + p2*20 + p3)*10000 + int(mat[:4])

def matrícules (mat1, mat2):
    return val_matrícula (mat2) - val_matrícula (mat1)
```

2. Dissenya la funció `travessa` que donat un string que conté el resultat d'un partit de futbol amb el format `'gel-gev'` (gols de l'equip local, un guió i gols de l'equip visitant), retorna el caràcter `'1'` si ha guanyat l'equip local, `'2'` si ha guanyat l'equip visitant i `'X'` si han empatat. Exemples:

```
>>> travessa('5-3')
'1'
>>> travessa('2-12')
'2'
>>> travessa('0-0')
'X'
>>> travessa('12-15')
'2'
>>> travessa('10-10')
'X'
>>> travessa('1-0')
'1'
```

Solució:

```
def travessa(resultat):
    posicio_guio = resultat.find('-')
    gols_local = int(resultat[:posicio_guio])
    gols_visitant = int(resultat[posicio_guio+1:])
    if gols_local > gols_visitant:
        return '1'
    elif gols_local < gols_visitant:
        return '2'
    elif gols_local == gols_visitant:
        return 'X'
```

# 5 Composició repetitiva

## 5.1 Sentència `for`

Els ordinadors poden realitzar tasques repetitives a gran velocitat i sense cometre errors. La **composició repetitiva o iteració** representa la repetició de l'execució d'un conjunt de sentències.

Les sentències `for` i `while` de Python permeten dissenyar composicions repetitives. En aquest capítol s'introdueix la sentència `for` pel recorregut d'strings.

Notació:

```
for caracter in cadena:
    tractar caracter
```

La sentència `for` és una altra sentència composta que segueix el patró ja descrit. A la primera línia hi ha dues paraules reservades noves, `for` i `in`), i acaba amb el caràcter `:` (dos punts). Segueix aquesta línia un bloc de sentències amb el corresponent sagnat.

La variable `caracter` pren successivament el valors de cadascun dels caràcters de `cadena` (string) des del primer fins al darrer. Aquest procés s'anomena **recorregut**. Si no es desitja tractar tots els caràcters de `cadena`, es pot posar el segment que interessi. Aquesta sentència repeteix el mateix proces (bloc de sentències) a cadascun dels caràcters de l'string. L'aplicació d'aquest procés a cada caràcter individual s'anomena **iteració** o **bucle**.

Molts dels problemes que impliquen un recorregut d'una estructura de dades es poden classificar segons una tipologia de processos. Els tres més comuns són: **sintetitzar** (reduce), **aplicar** (map) i **filtrar** (filter).

El procés de sintetitzar obté un resultat escalar a partir d'un tipus compost, com un string. A continuació s'enuncien exemples de problemes d'aquest tipus:

- comptar el nombre de vegades que l'string `'si'` apareix en un string donat
- comptar el nombre de lletres majúscules que hi ha en un string donat
- calcular el percentatge de vocals respecte del total de lletres que hi ha en un string donat

El mètode `count` vist al capítol anterior aplica un procés de síntesi. Ara bé, com tots els mètodes té limitacions i dels tres problemes enunciats només pot resoldre el primer. No permet comptar ni el nombre de lletres majúscules, ni el nombre de vocals.

Les funcions de la biblioteca estàndard `max` i `min` sobre strings també apliquen processos de síntesi.

En aquest capítol s'introdueix la tipologia de sintetitzar. Les tipologies d'aplicar i filtrar es veuran amb l'estructura de dades `list` en el capítol següent.

## 5.2 Recorregut d'strings: síntesi

En aquest apartat es tracten problemes en què a partir d'un string s'obté un resultat de tipus escalar.

Molts d'aquests processos són **comptadors** o **sumatoris**. En aquests processos cal una variable que fa la funció de **comptador** o **sumatori**, respectivament. Aquesta variable s'ha d'inicialitzar a `0` abans del recorregut de l'string i s'ha d'actualitzar adequadament a cada iteració.

**Enunciat 1**

Donat un string no buit, dissenya la funció `compta_majs` que retorni el nombre de lletres majúscules que conté. Exemple:

```
>>> s = 'En Joan Petit com balla !!!'
>>> compta_majs(s)
3
```

**Disseny**

```
def compta_majs (s):
    nombre_majs = 0
    for c in s:
        if c.isupper():
            nombre_majs = nombre_majs + 1
    return nombre_majs
```

La variable `nombre_majs` és un comptador. Cal inicialitzar aquest comptador a `0` abans de començar a comptar (abans de la sentència `for`) i actualitzar-lo a cada iteració. S'usa el mètode `isupper` del tipus `str isupper` per determinar si el caràcter és una lletra majúscula.

Per tal d'actualitzar un comptador o un sumatori es fan servir les operacions **incrementar** i **decrementar**. A continuació es mostra la corresponent sintaxi Python:

```
>>> c = c + 1 # el comptador s'incrementa en una unitat
>>> c = c - 1 # el comptador es decrementa en una unitat
>>> s = s + q # el sumatori s'incrementa en la quantitat q
>>> s = s - q # el sumatori es decrementa en la quantitat q
```

Python com altres llenguatges disposa d'una sintaxi específica per aquestes operacions:

```
>>> c += 1
>>> c -= 1
>>> s += q
>>> s -= q
```

**Enunciat 2**

Donat un string no buit, dissenya la funció `perc_vocals` que retorni el percentatge de vocals respecte del total de lletres que conté. Podem suposar que no hi ha vocals accentuades. Si a l'string no hi hagués cap lletra, la funció ha de retornar `0.0`. Exemple:

```
>>> round(perc_vocals('En Joan Petit com balla !!!'), 2)
42.11
```

**Disseny**

```
def perc_vocals (s):
    vocals = 'aeiouAEIOU'
    nombre_vocals = 0
    nombre_lletres = 0
    for c in s:
        if c.isalpha():
            nombre_lletres = nombre_lletres + 1
            if c in vocals:
                nombre_vocals = nombre_vocals + 1
    if nombre_lletres == 0:
        return 0.0
    else:
        return nombre_vocals/nombre_lletres*100
```

La funció `perc_vocals` aplica dos processos comptador per comptar el nombre total de lletres i el nombre de vocals. Observem que tots dos processos comptador es duen a terme amb una sola sentència `for`. Per determinar si un caràcter és una lletra s'usa el mètode `isalpha` i per determinar si és una vocal s'usa la variable`vocals` inicialitzada a un string amb les vocals en minúscula i majúscula juntament amb l'operador de pertinença `in`.

## 5.3 Recorregut a través dels índexs

Alguns processos requereixen poder accedir als elements dels strings a través dels seus índexs (posició). Per exemple, si cal analitzar grups de dos caràcters consecutius cal poder accedir a un caràcter i al següent i, per tant, cal accedir a aquests caràcters a través dels seus índexs.

La sentència `for` permet iterar sobre els índexs fent servir la funció de la biblioteca estàndard `range`.

La funció `range` genera una successió d'enters que es pot usar com una successió d'índexs. Aquesta funció es pot cridar de tres formes diferents:

1. `range(fi)`: genera la successió dels nombres enters des de `0` fins `fi-1` (ambdós inclosos).
2. `range(inici, fi)`: genera la successió dels nombres enters des de `inici` fins `fi-1` (ambdós inclosos).
3. `range(inici, fi, pas)`: genera la successió dels nombres enters: `inici`, `inici+pas`, `inici+2*pas`, ..., `final`, amb `final < fi`.

La tercera sintaxi és la general ja que els valors per defecte de `inici` i `pas` són `0` i `1` respectivament.

Observeu la semblança de la sintaxi de la funció `range` amb la de l'operador de segmentació.

Exemples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
```

Per recórrer un string a través dels seus índexs s'utilitza la sentència `for` amb la notació següent:

```
for i in range (len(cadena)):
    tractar cadena[i]
```

En aquesta notació la sentència `for` recorre els índexs de l'string `cadena` des del `0` fins `len(cadena)-1`, que són tots els possibles. Per accedir a l'element de l'string, usem l'operador d'indexació: `cadena[i]`.

Tots els exemples vistos abans es poden dissenyar recorrent els índexs en lloc dels elements directament. Ara bé, és una bona pràctica no usar índexs si no són estrictament necessaris.

A continuació es mostra el disseny de la funció `perc_vocals_index` que és la variant de l'anterior usant índexs:

```
def és_vocal (c):
    vocals = 'aeiouAEIOU'
    return c in vocals

def perc_vocals_index (s):
    nombre_vocals = 0
    nombre_lletres = 0
    for i in range(len(s)):
        if s[i].isalpha():
            nombre_lletres = nombre_lletres + 1
        if és_vocal(s[i]):
            nombre_vocals = nombre_vocals + 1
    return nombre_vocals/nombre_lletres*100
```

Els següents exercicis requereixen usar índexs:

**Enunciat 3**

En un joc es tiren dos daus diverses vegades i es vol conèixer el nombre de vegades que han sortit dos sisos. En un string hi ha una seqüència d'aquests parells de tirades (dígits entre l'`1` i el `6`, ambdós inclosos), és a dir, el primer i segon caràcter representen la primera tirada de dos daus, el tercer i el quart representen la segona tirada i així successivament. Dissenya la funció `sisos` que, a partir d'un string d'aquestes característiques, calcula i retorna el nombre de vegades que han sortit dos sisos en un joc. Exemple:

```
>>> sisos('12654323332666')
1
>>> sisos('3466526612')
2
```

**Disseny**

```
def sisos (daus):
    n = 0
    for i in range(0, len(daus), 2):
        if daus[i] == '6' and daus[i+1] == '6':
            n = n + 1
    return n
```

En aquest exercici la funció `range` genera els índexs parells (0, 2, 4, ...) i s'han de visitar els parells de caràcters $(c_0, c_1), (c_2, c_3), \ldots$. La sentència `for` recorre aquests índexs amb la variable `i`, accedint a l'element corresponent de l'string `daus[i]` i al següent, `daus[i+1]`.

Una forma alternativa d'escriure l'expressió booleana:
`daus[i] == '6' and daus[i+1] == '6'`
és mitjançant l'operador de segmentació:
`daus[i:i+2] == '66'`

Analitzeu si aquest exercici es pot resoldre amb el mètode `count` o no.

**Enunciat 4**

Un string conté un text on hi ha diverses frases. Cada frase acaba en un punt i la següent comença després d'aquest punt (no hi cap espai en blanc de separació), Dissenya la funció `frases` que, donat un string d'aquestes característiques, retorna el nombre de frases que comencen per vocal. Exemple:

```
>>> s1 = 'Demà sortim.Que vens?.Jo si.I en Joan ?.No ho sé.'
>>> s = s1 + 'On anem ?.Al cine.I a sopar?.Potser.'
>>> frases(s)
4
```

**Disseny**

```
def frases (s):
    vocals = 'aeiouAEIOU'
    if s[0] in vocals:
        n = 1
    else:
        n = 0
    for i in range(len(s)-1):
        if s[i] == '.' and s[i+1] in vocals:
            n = n + 1
    return n
```

Aquesta funció inicialitza el comptador a 1 o a 0, depenent de si la primera lletra és vocal o no. Després visita parells de caràcters que s'encavalquen:

$$(c_0, c_1), (c_1, c_2), (c_2, c_3) \ldots (c_{len(s)-2}, c_{len(s)-1})$$

per tal d'identificar els parells que segueixen el patró `('.', vocal)`.

## 5.4 Esquema de cerca

En tots els exemples vistos fins ara s'han recorregut de l'string ha estat complet. Hi ha una altra tipologia de processos en què no sempre és necessari recórrer tot l'string que

s'anomenen **processos de cerca**. En aquest tipus de processos es busca si algun element de l'string compleix una determinada condició i, quan es troba el primer element que la compleix el proces de cerca s'ha completat i el recorregut de l'string s'atura. Els únics casos en què un procés de cerca fa un recorregut complet són el cas en què cap element de l'string compleix la condició i el cas en què l'únic que la compleix és el darrer element.

El mètode `find` del tipus `str` aplica un proces de cerca. Exemples:

```
>>> 'No sé si es simplifica'.find('si')
6
>>> 'No sé si es simplifica'.find('no')
-1
```

En el primer cas, un cop trobat el primer `'si'` a la posició 6, l'string donat no cal que es recorri fins al seu darrer caràcter. En canvi en el segon cas l'string donat s'ha de recórrer fins al final per comprovar que l'string `'no'` no hi és.

Tots els mètodes del tipus `str` que permeten determinar el tipus d'un string (`isupper`, `islower`, etc.) també apliquen un proces de cerca. Exemples:

```
>>> '3 2 1 Jo votare SI !!!'.isupper()
False
>>> '3 2 1 JO VOTARE SI !!!'.isupper()
True
```

En el primer cas, un cop trobada la primera lletra minúscula, `'o'`, el mètode retorna `False` perquè ja es pot afirmar que l'string donat no té totes les lletres en majúscula. En canvi en el segon cas l'string donat s'ha de recórrer fins al final per poder afirmar que totes les lletres que hi ha són majúscules.

Per poder reflectir aquest comportament en la sintaxi Python vista fins ara, cal que hi hagi la possibilitat de sortir de la sentència `for` abans que s'hagi acabat. La sentència `break` és la que permet sortir del bucle `for` sense que s'hagin fet totes les iteracions.

L'esquema de cerca que es proposa és el següent:

```
trobat = False
for c in s:
    trobat = condició_de_cerca(c)
    if trobat:
       break
if trobat:
    tractament_trobat(c)
else:
    tractament_no_trobat (c)
```

En aquest esquema s'usa una variable d'estat `trobat`. Aquesta variable indica en cada instant si s'ha trobat algun element que compleix la condició o no. Al començament `trobat` s'inicialitza a `False`. A cada iteració del `for` s'avalua la condició de cerca per l'element en curs de l'string i la variable `trobat` passa a tenir aquest valor que és `True` o `False`. Quan la variable `trobat` val `True`, s'executa la sentència `break` i se surt del `for`. Un cop a fora cal diferenciar entre el cas que s'hagi trobat algun element que compleix la condició o el cas en què cap element compleix la condició. Exemples:

**Enunciat 5**

Considerem el mateix joc de l'enunciat 3 on es tiren dos daus diverses vegades però ara volem saber si algun cop han sortit dos valors iguals o no. Dissenya la funció `iguals` que a partir d'un string d'aquestes característiques retorna l'string `'afirmatiu'` si en alguna tirada de dos daus aquests tenen el mateix valor o bé l'string `'negatiu'` en cas contrari. Exemple:

```
>>> iguals('12654321332666')
'afirmatiu'
>>> iguals('34652112')
'negatiu'
```

En el primer exemple el recorregut acaba quan es troba el parell `'33'`. En el segon exemple es duu a terme un recorregut complet i el resultat és `'negatiu'`: hi ha el parell `'11'` però no correspon a una tirada.

**Disseny**

```
def iguals (daus):
    trobat = False
    for i in range(0, len(daus), 2):
        trobat =  daus[i] == daus[i+1]
        if trobat:
            break
    if trobat:
        return 'afirmatiu'
    else:
        return 'negatiu'
```

Ara dissenyarem una funció que calcula el mateix però en lloc de retornar els strings `'afirmatiu'` o `'negatiu'`, retorna un booleà `True` o `False`. A continuació es mostren tres dissenys equivalents:

```
def iguals1 (daus):
    trobat = False
    for i in range(0, len(daus), 2):
        trobat =  daus[i] == daus[i+1]
        if trobat:
            break
    if trobat:
        return True
    else:
        return False

def iguals2 (daus):
    trobat = False
    for i in range(0, len(daus), 2):
        trobat =  daus[i] == daus[i+1]
        if trobat:
            break
    return trobat

def iguals3 (daus):
```

```
    for i in range(0, len(daus), 2):
        if daus[i] == daus[i+1]:
            return True
    return False
```

La funció `igualS1` aplica l'esquema directament. La funció `igualS2` simplifica la composició condicional que hi ha després del `for` en una expressió booleana (on només hi ha la variable `trobat`).

La funció `igualS3` és correcta. Prescindeix de la variable d'estat `trobat` i quan troba l'element que compleix la condició surt no només del `for` sinó també de la funció usant la sentència `return` en lloc de `break`. Si es fan totes les iteracions del `for` quan aquest acaba la funció ha de retornar `False` ja que això significa que no s'ha trobat cap element que compleix la condició. Tot i que aquest disseny sembla més senzill, cal tenir present que aquesta simplificació només es pot fer en problemes molt simples com aquest i, per tant, es recomana usar sempre l'esquema complet inicial.

Es recomana parar atenció als nivells de sagnat dels anteriors exemples per entendre bé l'ús de les diverses composicions que hi apareixen. En Python el sagnat representa de forma gràfica el flux d'execució.

## 5.5 Exercicis

1. El Codi Morse és complicat de recordar, però existeixen regles mnemotècniques com la següent: recordar una paraula clau per cada lletra de forma que a partir de la paraula es pot obtenir el codi morse de la lletra seguint les regles següents:

   (a) la inicial de la paraula clau és la lletra corresponent,

   (b) el nombre de vocals que conté la paraula clau indica la longitud de la codificació en morse d'aquesta lletra,

   (c) si la vocal es una lletra O (minúscula, 'o', o bé majúscula, 'O') es substitueix per un guió ('-') i

   (d) si és qualsevol altra vocal es substitueix per un punt ('·')

   Dissenyeu la funció `morse` que a partir d'una paraula clau retorni la lletra que representa i el nombre de punts i ratlles (guions) de la seva codificació morse. Exemple:

```
>>> morse('Coca-cola')
('C', 2, 2)
>>> morse('Ventilador')
('V', 3, 1)
>>> morse('Motor')
('M', 0, 2)
```

   NOTA. Al capítol següent, es mostra una variant d'aquest exercici on es calcula el codi morse usant llistes.

```
def morse (paraula):
    vocals = 'aeiouAEIOU'
    lletra = paraula[0]
    nratlles = paraula.count('o') + paraula.count('O')
    nvoc = 0
    for c in paraula:
        if c in vocals:
            nvoc = nvoc + 1
    npunts = nvoc - nratlles
    return lletra, npunts, nratlles
```

   En aquest exercici el nombre de ratlles es calcula usant el mètode `count` per comptar el nombre de caràcters `'o'` i `'O'`. El nombre de punts serà el nombre total de vocals menys el nombre de ratlles.

2. El servei d'espionatge d'un determinat país ens demana que dissenyem la funció `subcadena` que a partir de dues cadenes de caràcters que contenen codis secrets, retorni la primera subcadena de caràcters iguals i en les mateixes posicions de les dues cadenes donades. Si no hi ha cap subcadena d'aquestes característiques, la funció ha de retornar l'string nul. Exemples:

```
>>> s1 = 'aeioubcde'
>>> s2 = 'dfgthbghtf'
>>> subcadena(s1, s2)
'b'
```

```
>>> s2 = 'aeiuoj'
>>> subcadena(s1, s2)
'aei'
>>> s2 = 'AE g-9cde'
>>> subcadena(s1, s2)
'cde'
>>> s2 = '..aeiou--!!!+hg'
>>> subcadena(s1, s2)
''
```

```
def subcadena (s1, s2):
    trobat = False
    lmin = min (len(s1), len(s2))
    for i in range(lmin):
        trobat = s1[i] == s2[i]
        if trobat:
            inici = i
            break
    if not trobat:
        return ''
    else:
        trobat = False
        for i in range(inici+1, lmin):
            trobat = s1[i] != s2[i]
            if trobat:
                fi = i
                break
        if trobat:
            return s1[inici:fi]
        else:
            return s1[inici:lmin]
```

En aquesta funció s'aplica l'esquema de cerca dues vegades. La primera per trobar la posició del primer caràcter coincident i la segona per trobar la posició del darrer caràcter coincident.

Com que hem de recórrer dues cadenes en paral·lel, el recorregut es farà través dels índexs i la seva extensió ens la donarà la longitud de l'string més curt.

# 6 Llistes

Les llistes, com els strings, són estructures de dades compostes de tipus seqüencial. A diferència dels strings, els elements de les llistes poden ser de qualsevol tipus. Les llistes poden ser homogènies (tots els elements del mateix tipus) o heterogènies (els elements poden ser de tipus diferent).

Una llista es representa tancant els seus elements entre claudàtors (`[]`) i separats per comes): $[e_0, e_1, \ldots, e_n]$. Una llista buida es representa com `[]`. Exemples:

```
#llista homogènia d'enters
>>> a1= [3, 8, 5, 30, 23]
#llista homogènia d'strings
>>> a2= ['cotxe', 'vaixell', 'moto']
#llista heterogènia
>>> a3= ['casa', True, 45.78, 32, 'hola']
>>> buida = []
```

## 6.1 Operacions

Les operacions de concatenació (+) i repetició (*) també són aplicables a les llistes, així com els operadors de comparació: `==, !=, <, >, <=, >=`, que es basen en l'ordre lexicogràfic aplicat a llistes. Quan es comparen dues llistes el que es compara són els seus elements en l'ordre en què apareixen a la llista. Cal tenir en compte que els elements que es comparen han de ser del mateix tipus.

La funció `len` de la biblioteca estàndard permet conèixer la longitud d'una llista, és a dir, el nombre d'elements que té.

Els operadors de pertinença `in` i `not in` també són aplicables a les llistes.

```
element in llista
element not in llista
```

Exemples:

```
>>> a1 = [3, 4, 5, 6]
>>> a2 = [10, 20, 30]
>>> a1 + a2
[3, 4, 5, 6, 10, 20, 30]
>>> a1 * 3
[3, 4, 5, 6, 3, 4, 5, 6, 3, 4, 5, 6]
>>> ['a', 'b'] * 3
['a', 'b', 'a', 'b', 'a', 'b']
>>> 5 in a1
True
>>> 10 in a1
False
>>> 20 in a1+a2
True
>>> a3 = ['casa', 23, True, 45.78, 32, 'hola']
```

```
>>> 'casa' in a3
True
>>> ['casa', 23] in a3
False
>>> a4 = ['casa', 'cotxe', True, 45.78, 32, 'hola']
>>> a3 == a4
False
>>> a3 < a4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

L'expressió `['casa', 23] in a3` dóna `False` perquè la llista `['casa', 23]` no és un element de la llista `a3`.

Per altra banda, les llistes `a3` i `a4` no tenen els elements del mateix tipus (el segon element de `a3` és un enter i el segon element de `a4` és un string i, per tant, només es poden comparar amb els operadors d'igualtat o no igualtat `==, !=`. L'expressió `a3 == a4` és avaluable i dóna `False`. Però l'expressió `a3 < a4` no és avaluable: primer es comparen els primers elements de `a3` i `a4` que es poden comparar perquè són strings tots dos. Com que són iguals (`'casa'`) es passa a comparar els segons elements `23` i `'cotxe'` i com que no es poden comparar perquè són de tipus diferents, Python emet un error.

## 6.2 Indexació i segmentació

Els mecanismes d'indexació i segmentació de les llistes són com els vistos pels strings.

Els elements d'una llista es poden accedir mitjançant un índex que ha de ser una expressió entera. Si `a` és una llista, la sintaxi per accedir a un element és: `a[index]`, $0 \leq index < len(a)$. L'índex també pot ser negatiu: $-len(a) \leq index < 0$

Una llesca o segment (slice) és un subconjunt dels elements d'una llista. Si `a` és una llista, la sintaxi per accedir a un segment és: `a[pos1:pos2:pas]`, $0 \leq pos1 < pos2 < len(a)$, on `pos1, pos2` i `pas` són expressions enteres.

Exemples:

```
>>> a1 = [10, 20, 30, 40, 50, 60]
>>> a1[0]
10
>>> a1[3]
40
>>> a1[2:4]
[30, 40]
>>> a1[::2]
[10, 30, 50]
```

## 6.3 Llistes imbricades

Els elements de les llistes poden ser de qualsevol tipus i, en particular, poden ser també llistes. Les llistes que tenen altres llistes com elements s'anomenen **llistes imbricades**. Per fer referència a la jerarquia en aquestes estructures es pot parlar de **llistes** i **subllistes**. Exemples:

```
>>> a=[1,2,3,[4,5]]
>>> a[3]
[4, 5]
>>> a[3][0]
4
```

El darrer element de la llista `a`, `a[3]`, es una altra llista, `[4, 5]`. Per accedir a un dels seus elements cal també indexar-la: `a[3][0]` accedeix al primer element de `a[3]`.

## 6.4 Mètodes del tipus `str` que usen llistes

Hi ha dos mètodes del tipus `str` (string) que operen amb strings i llistes: `split` i `join`. Recordem que per invocar mètodes cal usar la notació del punt.

A continuació es transcriu la petita ajuda que mostra el shell de Python per aquests mètodes:

```
S.split([sep [,maxsplit]]): list of strings

Return a list of the words in the string S, using sep as
the delimiter string. If maxsplit is given, at most
maxsplit splits are done. If sep is not specified or is
None, any whitespace string is a separator and empty
strings are removed from the result.
```

```
S.join(iterable): string

Return a string which is the concatenation of the strings
in the iterable. The separator between elements is S.
```

En la documentació del mètode `split` hi apareix el terme whitespace string. En l'àmbit de programació **whitespace** és qualsevol caràcter o seqüència de caràcters que representa espaiat horitzontal o vertical en tipografia: espai en blanc, tabulador i salt de línia.

El mètode `split` té dos tipus de comportament: general i especial. En el comportament general l'string es parteix en una llista d'strings usant el separador donat i aquest s'elimina. El comportament especial està dissenyat per un cas particular que es dóna molt sovint: el cas d'un text on les paraules estan separades per espais en blanc i on hi ha salts de línia. Quan a la crida al mètode `split` el paràmetre `sep` no s'especifica, tots els caràcters whitespace fan de separadors i, a més, s'eliminen tots els strings buits creats. Exemples:

```
>>> a = '00-11-22-33'
>>> a.split('-')       # comportament general
['00', '11', '22', '33']

>>> a = '00--11--22--33 ' # general: es creen strings buits
>>> a.split ('-')
['00', '', '11', '', '22', '', '33 ']

# específic: els strings buits s'eliminen
>>> b = 'Hola que    tal?\n\nCom   esteu ?'
>>> b.split()
['Hola', 'que', 'tal?', 'Com', 'esteu', '?']
```

En l'expressió `a.split ('-')` el mètode `split` té el comportament general. En el segon exemple, es creen strings buits entre cada parell de guions consecutius. A l'expressió `b.split()` no hi ha cap separador i el mètode `split` té el comportament específic: els caràcters whitespace fan de separador i els strings buits creats són eliminats.

El mètode `join` concatena tots els elements de la llista donada intercalant l'string donat que fa de juntura.

El mètode `join` té una certa relació amb la funció `list` de la biblioteca estàndard: l'un fa l'operació inversa de l'altre quan el caràcter de juntura és l'string buit. La funció `list` converteix qualsevol objecte de tipus seqüencial (com ara un string) en una llista.

A continuació es mostren exemples. Observeu el resultat d'aplicar la funció `str` a una llista:

```
>>> c = ['34', '93', '432', '7698']
>>> '-'.join(c)
'34-93-432-7698'
>>> d=['Ajuntem', 'paraules', 'i', 'fem', 'frases']
>>> ' '.join(d)
'Ajuntem paraules i fem frases'
>>> list('abcd')
['a', 'b', 'c', 'd']
>>> ''.join(['a', 'b', 'c', 'd'])
'abcd'
>>> str(['a', 'b', 'c', 'd'])  # conversió str a llista
"['a', 'b', 'c', 'd']"
```

Els següents exercicis fan ús dels mètodes vistos.

**Enunciat 1**

Donat un string amb un grup de consonants i un booleà, dissenya la funció `síl·laba` que retorni un string amb totes les síl·labes formades per aquests grup de consonants i cadascuna de les vocals en ordre alfabètic. Si el booleà val `True` el grup consonants s'ha de posar davant cada vocal i si val `False` a darrera. Per simplificar es considerarà que tot són lletres minúscules. Exemple:

```
>>> silaba('m', True)
'mamemimomu'
>>> silaba('rf', False)
```

```
'arferfirforfurf'
```

**Disseny**

```
def silaba(cons, davant):
    vocals = 'aeiou'
    if davant:
        return cons + cons.join(vocals)
    else:
        return cons.join(vocals) + cons
```

**Enunciat 2**

En un string hi ha diverses quantitats enteres separades per un guió. Dissenya la funció `suma_seq` que a partir d'un string com l'indicat, retorni la suma de les quantitats representades a l'string. Exemple:

```
>>> suma_seq('1-2-3-4-5')
15
>>> suma_seq('34-89-77-654-1')
855
```

**Disseny**

```
def suma_seq (s):
    suma = 0
    lvalors = s.split('-')
    for c in lvalors:
        suma = suma + int(c)
    return suma
```

La funció `suma_seq` és un exemple de síntesi. A partir de l'string donat obté un enter. El mètode `split` obté una llista amb els strings que contenen la informació numèrica. Després aquests strings s'han de passar a enters i sumar.

## 6.5 Recorregut de llistes

Tal com s'ha vist a l'anterior capítol, es poden aplicar esquemes de recorregut i cerca usant la sentència `for` directament sobre els elements de la llista i indirectament usant els índexs.

També és aplicable la tipologia de casos de síntesi (reduce), aplicació (map) i filtratge (filter). A continuació es mostren exemples de síntesi.

**Enunciat 3**

Donada una llista amb valors reals corresponents a les alçades d'una mostra de població femenina de 18 anys, dissenya la funció `mitjana` que retorni l'alçada mitjana. Exemple:

```
>>> la= [1.7, 1.73, 1.6, 1.64, 1.61, 1.78, 1.8, 1.67, 1.55]
>>> round(mitjana(la), 2)
1.68
```

**Disseny**

```
def mitjana_1 (alçades):
    suma = 0
    for a in alçades:
        suma = suma +a
    return suma/len(alçades)

def mitjana_2 (alçades):
    return sum(alçades)/len(alçades)
```

La primera versió de la funció `mitjana` calcula la suma de tots els elements de la llista amb una sentència `for` i una variable sumatori que s'ha d'inicialitzat a `0` abans de la sentència `for`. La segona versió usa la funció de la biblioteca estàndard `sum`.

**Un altre exercici**

En el següent exercici mostrem un altre exemple de síntesi: el procés de trobar el màxim (o el mínim) d'una estructura de dades. En aquest procés, fem servir una variable com a valor de referència i que complirà la condició d'invariant de ser el màxim (o el mínim) dels elements visitats. Aquesta variable s'ha d'inicialitzar en el cas d'obtenció d'un màxim a un valor **suficientment petit** (i en el cas d'un mínim a un valor **suficientment gran**). Si la tipologia de les dades no permet determinar aquests valors, es pot usar el primer element com a valor de referència.

**Enunciat 4**

Un interval es representa amb una llista de dos elements que representen els seus extrems inferior i superior (`float`). Donada una llista d'intervals, dissenya la funció `més_llarg` que retorna la longitud de l'interval més llarg (`float`). Exemple:

```
>>> lint = [[-3, 3], [1.2, 4.8], [-56, 24.6], [-20, -10]]
>>> més_llarg(lint)
80.6
```

**Disseny**

```
def més_llarg (lintervals):
    llarg = 0
    for interval in lintervals:
        longitud = interval[1] - interval[0]
        if longitud > llarg:
            llarg = longitud
    return llarg
```

Inicialitzem la variable de referència `llarg` a un valor **suficientment petit** (volem obtenir un valor màxim). A cada iteració comparem la variable `llarg` amb la longitud de l'interval en curs i l'actualitzem si cal. Observem que si hi ha dos intervals amb la mateixa longitud màxima, la funció retorna la primera que troba. Observem també que en aquest procés de çerca"del valor màxim, cal aplicar un esquema de recorregut, ja que cal visitar tots els elements per tal de obtenir el seu màxim.

**Exercici de cerca**

El següent exercici mostra un exemple d'esquema de cerca.

**Enunciat 5**

Dissenya primer la funció `pentavocalica` que donada una paraula (string) retorni un booleà que indiqui si a la paraula hi ha totes les vocals.

Donada una llista de paraules, dissenya la funció `pripenta` que retorni la primera paraula de la llista que tingui totes les vocals. Si no n'hi ha cap, ha de retornar l'string buit. Podem suposar que no hi ha vocals accentuades. Aquesta funció ha d'usar la funció `pentavocalica`. Exemple:

```
>>> pentavocalica('insuportable')
True
>>> pentavocalica('matematiques')
False
>>> pentavocalica('equacio')
True
>>> pentavocalica('tauro')
False
>>> llista =['matematiques', 'variable', 'equacio',
...   'formula', 'funcio']
>>> pripenta(llista)
'equacio'
>>> llista = ['jo', 'tu', 'nosaltres', 'vosaltres']
>>> pripenta(llista)
''
>>> llista = ['euforia', 'equacio', 'ouaire', 'continuarem',
...  'insuportable']
>>> pripenta(llista)
'euforia'
```

**Disseny**

```
def pentavocalica(paraula):
    vocals = 'aeiou'
    hi_són_totes = True
    for vocal in vocals:
        hi_són_totes = vocal in paraula
        if not hi_són_totes:
            break
    return hi_són_totes

def pripenta (lpar):
    trobat = False
    for paraula in lpar:
        trobat = pentavocalica(paraula)
        if trobat:
            break
    if trobat:
        return paraula
    else:
        return ''
```

En totes dues funcions s'ha aplicat l'esquema de cerca amb variable d'estat booleana.

Analitzeu i compareu l'ús de les variables booleanes `hi_són_totes` i `trobat`.

Totes dues funcions es poden dissenyar usant la sentència `return`:

```
def pentavocalica(paraula):
    vocals = 'aeiou'
    for vocal in vocals:
        if vocal not in paraula:
            return False
    return True

def pripenta (lpar):
    for paraula in lpar:
        if pentavocalica(paraula):
            return paraula
    return ''
```

## 6.6 Mutabilitat de les llistes

A diferència dels strings, les llistes són mutables, és a dir, es poden modificar. Exemples:

```
>>> a = ['hola', 2.0, 5, [10,20]]
>>> a[1]= 'adeu'     # substitució d'un element
>>> a
['hola', 'adeu', 5, [10, 20]]
>>> a = ['hola', 2.0, 5, [10,20]]
>>> a[1]= [1.0, 3.0]  # substitució d'un real per una llista
>>> a
['hola', [1.0, 3.0], 5, [10, 20]]
>>> a = ['hola', 2.0, 5, [10,20]]
>>> a[1:3]= [1,2,3,4] # substitució de diversos elements
>>> a
['hola', 1, 2, 3, 4, [10, 20]]
>>> s = [22, 33, 44, 55, 66, 77, 88]
>>> t = ['a', 'b', 'c', 'd']
>>> s[0:7:2]= t  # substitució alternada
>>> s
['a', 33, 'b', 55, 'c', 77, 'd']
```

En els exemples d'aquest apartat es comprova que la llista s'ha modificat mostrant-la abans i després de la modificació.

A part de modificar els seus components individuals, les llistes es poden modificar esborrant o afegint elements.

La sentència `del` permet esborrar un o més elements d'una llista. Exemples:

```
>>> aa = [3, 5, 8, 6, 2, 4, 1, 9]
>>> del aa[2]
>>> aa
[3, 5, 6, 2, 4, 1, 9]
>>> del aa[1:3]
```

```
>>> la
[3, 2, 4, 1, 9]
>>> lb = ['a', 'e', 'i', 'o', 'u']
>>> del lb[::2]
>>> lb
['e', 'o']
```

El mètode `append` permet afegir elements al final d'una llista. A l'apartat 6.9 es descriu aquest i altres mètodes sobre llistes.

## 6.7 Objectes, identificadors i efecte alies

Al primer capítol s'han introduït els conceptes de valor i variable. Els valors (objectes) són els elements fonamentals que manipula un programa. Una variable és un nom que fa referència a un valor. Ara s'introdueix el concepte d'**identitat** d'un objecte. És un enter que és únic i constant per a aquest objecte durant tota la seva vida útil i és la seva adreça de memòria d'aquest objecte. Dos objectes amb vides útils no superposades poden tenir la mateixa identitat. La funció `id` de la biblioteca estàndard retorna la identitat d'un objecte.

En els objectes immutables, com per exemple els strings, cada valor té una identitat única i, per tant, si dues o més variables tenen el mateix valor, tindran també la mateixa identitat. Aquest comportament no és cap problema perquè els strings són immutables. Exemples:

```
>>> x = 5
>>> y = 3+2
>>> id(x)
10105952
>>> id(y)
10105952        # x, y fan referència al mateix objecte
>>> a = 'aeiou'
>>> b = 'aeiou'
>>> c = a
>>> d = 'ae' + 'iou'
>>> id(a)
139889255176600
>>> id(b)
139889255176600
>>> id(c)
139889255176600
>>> id(d)      # a, b, c, d fan referència al mateix objecte
139889255176600
```

Per als tipus mutables com les llistes, cada vegada que es fa referència a un objecte de tipus llista, es crea un objecte nou. Per tant, pot haver-hi més d'un objecte compartint el mateix valor. Exemples:

```
>>> llistaa = [1, 2, 3, 4, 5, 6]
>>> llistab = [1, 2, 3, 4, 5, 6]
```

```
>>> id (llistaa)
139889255147336
>>> id (llistab)
139889255187464 #
>>> llistaa == llistab
True
>>> id(llistaa) == id(llistab)
False
>>> llistaa[2] = 34
>>> llistab[1] = 56
>>> llistaa
[1, 2, 34, 4, 5, 6]
>>> llistab
[1, 56, 3, 4, 5, 6]
```

Les llistes `llistaa` i `llistab` tenen el mateix valor, però no són el mateix objecte (les seves identitats són diferents). La mutabilitat de les llistes es basa en aquest comportament. Si ambdues llistes `llistaa` i `llistab` fessin referència al mateix objecte, és a dir, tinguessin la mateixa identitat, la modificació d'una llista afectaria l'altra.

Amb objectes mutables com les llistes, podem tenir les dues situacions següents. En la primera situació, que es mostra a l'exemple anterior, les llistes `llistaa` i `llistab` tenen identitats diferents. Per tant, el fet de modificar-ne una no té cap efecte en l'altre.

En l'altra situació, diverses llistes poden referir al mateix objecte, és a dir, poden tenir la mateixa identitat: són **àlies** i quan se'n modifica una les altres es modifiquen de la mateixa manera. La sintaxi per definir una llista de `llistax` com a àlies d'una altra llista existent `llistaa` és:

```
llistax = llistaa
```

Exemples:

```
>>> llistaa = [1, 2, 3, 4, 5, 6]
>>> llistab = [1, 2, 3, 4, 5, 6]
>>> llistac = llistaa[:3] + llistaa[3:]
>>> llistad = llistaa[:]
>>> id (llistaa)
139889255187144
>>> id (llistab)
139889255187656
>>> id (llistac)
139889255188168
>>> id (llistad)
139889255187528
>>> llistax = llistaa      # llistaa i llistax són àlies
>>> id (llistax)
139889255187144
>>> llistaa [2] = 456
>>> llistaa
[1, 2, 456, 4, 5, 6]
>>> llistab
```

```
[1, 2, 3, 4, 5, 6]
>>> llistac
[1, 2, 3, 4, 5, 6]
>>> llistad
[1, 2, 3, 4, 5, 6]
>>> llistax
[1, 2, 456, 4, 5, 6]
>>> llistax [3] = 1234
>>> llistaa
[1, 2, 456, 1234, 5, 6]
>>> llistax
[1, 2, 456, 1234, 5, 6]
```

Les llistes `llistaa, llistab, llistac` i `llistad` tenen identitats diferents. No es produeix efecte àlies.

En canvi les llistes `llistax` i `llistaa` tenen la mateixa identitat. Per tant, quan es modifica un element de `llistaa`, les llistes `llistab, llistac` i `llistad` no s'immuten i, en canvi, la llista `llistax` es modifica de la mateixa manera que `llistaa`. Si es modifica `llistax`, el canvi també afecta a `llistaa`.

Si es vol fer una còpia sense efecte àlies (**clonació**) s'ha de crear un nou objecte llista assignant-li el mateix valor. La sintaxi més adient per dur a terme aquest procés es:

```
llistad = llistaa[:]
```

La llista `llistad` és una copia de la llista `llistaa`, però no són el mateix objecte, és a dir, no tenen la mateixa identitat.

## 6.8 Funcions de la biblioteca estàndard

Les funcions de la biblioteca estàndard `max`, `min`, `sum` ja vistes i la funció `sorted` són aplicables a llistes homogènies. La funció `sorted` rep una llista i en retorna una altra amb els mateixos elements ordenats de petit a gran. Exemples:

```
>>> am = [4,6,5,1,2,3]
>>> max(am)
6
>>> min(am)
1
>>> sum(am)
21
>>> sorted(am)
[1, 2, 3, 4, 5, 6]
>>> am
[4, 6, 5, 1, 2, 3]
```

## 6.9 Tipus `list`: mètodes

En aquest apartat es veuen alguns dels mètodes del tipus `list`. Aquests mètodes poden modificar la llista o no. Hi ha mètodes que tenen valors de retorn i d'altres que no retornen res. De fet tota funció o mètode sempre té un valor de retorn: quan no retorna res en realitat retorna el valor `None`.

Les funcions i mètodes que modifiquen un paràmetre s'anomenen **modificadors** i els canvis que fan s'anomenen **efectes secundaris**. Aquests conceptes es descriuen en més detall a l'apartat 6.12.

La taula següent mostra alguns dels mètodes del tipus `list` classificats segons aquestes propietats:

| | NO modificador | modificador |
|---|---|---|
| amb valor de retorn | count, index | pop |
| retorna `None` | | append<br>sort, reverse<br>insert, remove |

Els mètodes s'invoquen usant la notació del punt:

`nom_variable_list.nom_mètode(paràmetres actuals)`

A continuació es transcriu la documentació resumida per alguns mètodes que aporta la funció `help`:

```
count(...)
    L.count(value) -> integer
    return number of occurrences of value

index(...)
    L.index(value, [start, [stop]]) -> integer
    return first index of value.
    Raises ValueError if the value is not present.

pop(...)
    L.pop([index]) -> item
    remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

append(...)
    L.append(object) -> None
    append object to end

sort(...)
    L.sort(key=None, reverse=False) -> None
    stable sort *IN PLACE*

reverse(...)
    L.reverse() -- reverse *IN PLACE*
```

```
insert(...)
    L.insert(index, object)
    insert object before index

remove(...)
    L.remove(value) -> None
    remove first occurrence of value.
    Raises ValueError if the value is not present.
```

Els mètodes `count` i `índex` són equivalents als mètodes `count` i `find` del tipus `str`. No són mètodes modificadors. El mètode `index` dóna error quan el valor no hi és.

El mètode `append` és un mètode modificador: afegeix un element al final d'una llista. Aquest mètode retorna `None`.

El mètode `sort` també és un mètode modificador: ordena la llista. Aquest mètode retorna `None`.

Exemples:

```
>>> lm = [4,6,5,1,2,3]
>>> lm.append(10)
>>> lm
[4, 6, 5, 1, 2, 3, 10]
>>> lp = ['t', 'e', 'g', 'h', 'a']
>>> lps = sorted(lp)
>>> lp
['t', 'e', 'g', 'h', 'a']
>>> lps
['a', 'e', 'g', 'h', 't']
>>> lq = [6, 7, 2, 1, 5, 4]
>>> lq
[6, 7, 2, 1, 5, 4]
>>> lq.sort()
>>> lq
[1, 2, 4, 5, 6, 7]
```

Observem la diferència entre la funció `sorted` i el mètode `sort`.

El mètode `reverse` modifica la llista capgirant els seus elements i retorna `None`.

La sintaxi `[::-1]` també permet capgirar una llista però no modifica la llista donada: en crea una altra.

Exemples:

```
>>> a1 = ['1', '2', '3', '4', '5']
>>> a2 = ['1', '2', '3', '4', '5']
>>> a1.reverse()
>>> a1
['5', '4', '3', '2', '1']
>>> rtm = a2[::-1]
>>> rtm
['5', '4', '3', '2', '1']
```

```
>>> a2
['1', '2', '3', '4', '5']
```

## 6.10 Recorregut de llistes: tipologies aplicació (map) i filtrat (filter)

Les tipologies `aplicació filtrat` impliquen la creació d'una nova llista a partir de la donada. El procés d'aplicació aplica un procediment a cadascun dels elements de la llista inicial per obtenir cadascun dels elements de la nova llista. En un procés de filtrat s'avalua una condició per cadascun dels elements de la llista inicial i només els que la compleixen s'afegeixen a la nova llista. Aquestes dues tipologies junt amb la de síntesi es poden combinar.

Així com en els processos comptador i sumatori cal inicialitzar la variable comptador a `0`, la creació d'una llista requereix una inicialització d'una llista buida `[]`. La incorporació de cada nou element al final d'aquesta nova llista es durà a terme amb el mètode `append`.

A continuació es mostren alguns exemples:

**Enunciat 6**

Un interval es representa amb una llista de dos elements (reals) que representen els seus extrems inferior i superior. Donada una llista d'intervals, dissenya la funció `centre` que retorna una altra llista de valors reals corresponents al valor del mig de cada interval, en el mateix ordre que en la llista inicial. Exemple:

```
>>> lint = [[-3, 3], [1.2, 4.8], [-56, 24.6], [-20, -10]]
>>> centre(lint)
[0.0, 3.0, -15.7, -15.0]
```

**Disseny**

```
def centre (lintervals):
    lcentres = []
    for interval in lintervals:
        c = (interval[0] + interval[1])/2
        lcentres.append(c)
    return lcentres
```

En aquest exemple s'ha usat una llista de llistes (llista imbricada), `lintervals`. Aquest és un cas d'aplicació (map): a cada interval de la llista se li aplica el procediment de calcular el seu valor del mig i aquests s'afegeixen a la llista `lcentres` usant el mètode `append`.

**Enunciat 7**

Dissenya la funció `noms` que donat un string amb un text, retorni una llista ordenada alfabèticament amb els substantius que hi ha en el text. Per detectar substantius aplicarem únicament la següent senzilla regla: un substantiu va sempre precedit dels articles 'la' o 'el' i aquestes paraules només precedeixen substantius. Podem suposar que al text no hi ha signes de puntuació i que totes les lletres són minúscules. Les paraules estan separades per un o més espais. Exemple:

```
>>> text = 'el llibre de ficció és sobre la taula i
...  la llibreta ha caigut sota el tamboret'
>>> noms(text)
['llibre', 'llibreta', 'tamboret', 'taula']
```

**Disseny**

```
def noms(text):
    lparaules = text.split()
    lsubs = []
    for i in range(len(lparaules)-1):
        if lparaules[i]=='el' or lparaules[i]=='la':
            lsubs.append(lparaules[i+1])
    lsubs.sort()
    return lsubs
```

En aquest exemple, primer s'obté una llista amb totes les paraules del text. Hem usat el mètode `split` en la modalitat especial en què no s'indica separador. Aleshores cal generar parells de paraules consecutives per tal de seleccionar els que segueixen el patró `('el', substantiu)` o `('la', substantiu)`). Per tal de generar parells de paraules consecutives, el recorregut de la llista es fa a través de l'índex i, a més, s'aplica un procés de filtratge. Al final s'ordena la nova llista amb el mètode `sort`

**Processos d'aplicar i filtrar sobre strings**

Els processos d'aplicar i filtrar també es poden fer sobre strings. Una primera estratègia es basa en construir primer una llista de caràcters (en lloc d'un string) per tal de poder usar el mètode `append` i després convertir la llista en un string amb el mètode `join`.

**Enunciat 8**

Donat un string, dissenya la funció `extreu_vocals` que retorni un altre string que contingui totes i només les vocals que hi ha a l'string donat i en el mateix ordre. Exemple:

```
>>> extreu\_vocals('Hola que tal')
'oauea'
>>> extreu\_vocals('Agosarat i esmaperdut')
'Aoaaieaeu'
```

**Disseny**

```
def és_vocal (c):
    vocals = 'aeiouAEIOU'
    return c in vocals

def extreu_vocals (s):
    novall = []
    for c in s:
        if és_vocal(c):
            novall.append(c)
    return ''.join(novall)
```

Aquest és un exemple de filtre: la condició que han de complir els caràcters per passar a formar part de l'string final és que siguin una vocal. Aquesta funció crida la funció

`és_vocal` dissenyada anteriorment.

**Enunciat 9**

Donat un string, dissenya la funció `aplica_vocal` que retorni un altre string on totes les vocals s'han substituït per la vocal donada. Exemple:

```
>>> extreu\_vocals('Hola que tal', 'a')
'Hala qua tal')
>>> extreu\_vocals('Agosarat i esmaperdut', 'i')
'igisirit i ismipirdit')
```

**Disseny**

```
def és_vocal (c):
    vocals = 'aeiouAEIOU'
    return c in vocals

def aplica_vocal (s, v):
    novall = []
    for c in s:
        if és_vocal(c):
            novall.append(v)
        else:
            novall.append(c)
    return ''.join(novall)
```

## 6.11 Append vs. concatenació

Python disposa d'un altre mecanisme per afegir elements que és la concatenació. Aplicat a strings, a cada iteració es concatena el nou caràcter a l'string que es construeix. A continuació és mostra un altre disseny de l'exercici 8 usant aquest altre mecanisme:

**Disseny**

```
def és_vocal (c):
    vocals = 'aeiouAEIOU'
    return c in vocals

def extreu_vocals (s):
    snou = ''
    for c in s:
        if és_vocal(c):
            snou = snou + c
    return snou
```

En aquest exemple no s'usa cap llista auxiliar. L'string nou, `snou`, s'inicialitza a l'string buit i cada nou caràcter es concatena amb `snou`. Tot i que aquesta versió funciona correctament, cal analitzar la sentència `snou = snou + c`. Observem que a cada iteració es crea un objecte nou (amb un caràcter més) ja que els strings no són mutables. Per tant, amb aquest disseny, la gestió que s'està fent de la memòria és bastant menys eficient.

## 6.12 Funcions modificadores

Una **funció modificadora** és aquella que modifica algun dels seus paràmetre i produeix un **efecte secundari**. Per tal de modificar un paràmetre aquest ha de ser d'un tipus mutable com les llistes i com els diccionaris (veure capítol 9). Per tant, les funcions amb tots els paràmetres immutables no poden ser modificadores per la pròpia naturalesa dels paràmetres. En canvi, si en una funció hi ha paràmetres mutables, la funció pot o no modificar-los. Designarem com **funció no modificadora** aquella funció que no modifica cap paràmetre.

Hem vist que quan assignem una llista a una altra, aquestes dues llistes esdevenen àlies. Les funcions modificadores es basen en aquest fet ja que quan es crida una funció amb un paràmetre mutable, els corresponents paràmetres formal i actual també esdevenen àlies. Per tant, si la funció modifica el paràmetre formal, aquesta modificació també es manifesta en el paràmetre actual.

**Enunciat 10**

Dissenya la funció modificadora `doblemodifica` que donada una llista la modifiqui de forma que els seus nous elements siguin els anteriors multiplicats per dos. Exemple:

```
>>> ln = [3, 5.2, 'hola', 25]
>>> rtn = double_modifier (ln)
>>> ln
[6, 10.4, 'holahola', 50]
>>> print (rtn)
None
```

**Disseny**

```
def doblemodifica(lm):
    for i in range(len(lm)):
        lm[i] = 2*lm[i]
```

Observacions:

- la llista pot tenir elements de qualsevol tipus al que se li pugui aplicar l'operador `*`
- aquesta funció ha de modificar els elements de la llista. Per això es necessita l'índex i, per tant, el recorregut amb `for` es fa a través dels índexs
- en aquesta funció no es crea cap nova llista sinó que es modifica la llista passada com a paràmetre. El doctest mostra els valors diferents de la llista `l1` abans i després de cridar la funció `doblemodifica`. Aquesta funció produeix un canvi en el valor del paràmetre formal i, com efecte secundari, també modifica el paràmetre actual, ja que són àlies.
- aquesta funció no retorna cap llista; el que retorna és el valor `None`.

Si no es vol modificar el paràmetre, podem dissenyar una funció no modificadora que crea i retorna una variable local. El següent exercici resol un problema similar a l'anterior amb una funció no modificadora.

**Enunciat 11**

Dissenya la funció `doblenomodifica` que donada una llista, en retorni una altra tal que els seus elements siguin iguals als de la llista donada multiplicats o replicats per dos. Exemple:

```
>>> l2 = [3, 5.2, 'hola', 25]
>>> rtn = doblenomodifica(l2)
>>> rtn
[6, 10.4, 'holahola', 50]
>>> l2
[3, 5.2, 'hola', 25]
```

**Disseny**

```
def doblenomodifica(lm):
    noval = []
    for e in lm:
        noval.append(2*e)
    return noval
```

Observacions:

- aquesta funció pot fer el recorregut amb `for` directament sense índexs
- en aquesta funció es crea una nova llista i no es modifica la llista passada com a paràmetre. Veiem als tests com la llista `l2` roman igual després de cridar la funció `doblenomodifica`
- aquesta funció retorna la nova llista creada. En el doctest la llista retornada s'assigna a la variable `rtn` i després es mostra el seu contingut.

Es recomana comparar detalladament els enunciats, els dissenys, els exemples de crida i les observacions dels dos exercicis anteriors per una millor comprensió dels conceptes de mutabilitat i funcions modificadores.

Als apartats anteriors hem vist diverses funcions de la biblioteca estàndard que s'apliquen a llistes així com a diversos mètodes de tipus `list`. Les funcions `max`, `min`, `sum` i `sorted` no són modificadores. Pel que fa als mètodes, a l'apartat 6.9 es mostra una taula indicant quins mètodes són modificadors i quins no.

En el següent exemple, es comparen la funció `sorted` (funció no modificadora) i el mètode `sort` (modificador).

```
>>> llistaa = [34, 25, 88, 76, 23, 14]
>>> llistab = sorted (llistaa)
>>> llistaa                  # llistaa no es modifica
[34, 25, 88, 76, 23, 14]
>>> llistab     # llista retornada per la funció sorted
[14, 23, 25, 34, 76, 88]
>>> rtn = llistaa.sort()
>>> llistaa                  # llistaa s'ha modificat
[14, 23, 25, 34, 76, 88]
```

```
>>> print(rtn)              # el mètode sort retorna None
None
```

La funció `sorted` és una funció no modificadora que no modifica el paràmetre de tipus `list` `llistaa` i retorna una llista diferent amb els mateixos elements que la llista donada en ordre ascendent. Al contrari, el mètode `sort` és un modificador ja que modifica la llista donada `llistaa` i retorna `None`.

Compareu aquestes observacions amb les que s'han fet per les dues funcions `doblemodifica` i `doblenomodifica`.

## 6.13 Paràmetres opcionals

Tant la funció `sorted` com el mètode `sort` tenen dos paràmetres opcionals: `reverse` i `key`.

El paràmetre `reverse` és un booleà. Per defecte val `False` i l'ordenació s'efectua en ordre creixent. Si val `True`, llavors l'ordenació es fa en ordre decreixent. Té el mateix efecte que ordenar primer i tot seguit capgirar (mètode `reverse`). Exemples:

```
#### exemples amb la funció sorted
>>> a = [3, 5, 1, 7, 8]
>>> sorted(a)
[1, 3, 5, 7, 8]
>>> sorted(a, reverse=True)
[8, 7, 5, 3, 1]

#### els mateixos exemples amb el mètode sort
>>> a = [3, 5, 1, 7, 8]
>>> a.sort()
>>> a
[1, 3, 5, 7, 8]
>>> a = [3, 5, 1, 7, 8]
>>> a.sort(reverse = True)
>>> a
[8, 7, 5, 3, 1]
```

El paràmetre `key` és una funció d'un paràmetre. L'ordre aplicat es pot personalitzar amb aquesta funció. El valor per defecte és `None`.

La funció `key` pot ser una funció o mètode ja disponible: vegeu els exemples a continuació amb la funció `len` i el mètode `lower`. També pot ser una funció dissenyada específicament: vegeu l'exemple a continuació amb la funció `nombre_vocals`, o una funció anònima.

Es poden crear funcions anònimes (`lambda`) amb la paraula reservada `lambda`. Les funcions lambda estan restringides sintàcticament a una sola expressió. Exemples:

| | |
|---|---|
| `lambda a, b:  a+b` | retorna la suma dels seus dos paràmetres |
| `lambda n:  2**n` | retorna l'enèsima potència de 2 |

A continuació es mostren exemples de la funció `sorted` i el mètode `sort` usant el paràmetre `key` de diverses maneres.

```
>>> from vocals import nombre_vocals

#### exemples amb la funció sorted: paràmetre key

>>> la = 'this is a test string from Andrew'.split()
>>> sorted(la, key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'this']

# ordenació per longitud
>>> sorted(la, key=len)
['a', 'is', 'this', 'test', 'from', 'string', 'Andrew']

# ordenació pel nombre de caràcters 't'
>>> sorted(la, key=lambda x: x.count('t'))
['is', 'a', 'from', 'Andrew', 'this', 'string', 'test']

# ordenació per nombre de vocals
>>> lb ='these are some test strings from Paul'.split()
>>> sorted(lb, key= nombre_vocals)
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
>>> sorted(lb, key= lambda x: nombre_vocals(x))
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']


#### els mateixos exemples amb el mètode sort: paràmetre key

# ordre alfabètic
# primer passem el text a minúscules
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=str.lower)
>>> la
['a', 'Andrew', 'from', 'is', 'string', 'test', 'this']

# ordenació per longitud
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=len)
>>> la
['a', 'is', 'this', 'test', 'from', 'string', 'Andrew']

# ordenació pel nombre de caràcters 't'
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=lambda x: x.count('t'))
>>> la
['is', 'a', 'from', 'Andrew', 'this', 'string', 'test']

# ordenació per nombre de vocals
lb = 'these are some test strings from Paul'.split()
>>> lb.sort(key= nombre_vocals)
>>> lb
```

```
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
>>> lb.sort(key= lambda x: nombre_vocals(x))
>>> lb
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
```

i la funció `nombre_vocals` és la següent:

```
def nombre_vocals(s):
    voc = 'aeiouAEIOU'
    nv = 0
    for c in s:
       if c in voc:
          nv = nv +1
    return nv
```

L'ordenació de llistes imbricades es fa per defecte en ordre lexicogràfic. El paràmetre `key` també es pot usar per definir una funció que modifiqui aquest ordre. Exemples:

```
#### ordenació de llistes imbricades: funció sorted
# exemple: llista de tuples del tipus (nom, curs, edat)
>>> estudiants = [('john', 'A', 15), ('jane', 'A', 12),
...               ('dave', 'B', 10), ('ed', 'B', 12)]

# ordena per nom (per defecte: primer element dels tuples)
>>> sorted(estudiants)
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# ordena per edat
>>> sorted (estudiants, key=lambda estudiant: estudiant[2])
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]

# ordena primer per edat i després per nom
>>> sorted (estudiants,
...     key=lambda estudiant: (estudiant[2], estudiant[0]))
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# ordena primer per edat i després per curs
>>> sorted (estudiants,
...     key=lambda estudiant: (estudiant[2], estudiant[1]))
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]


#### ordenació de llistes imbricades: mètode sort
# exemple: llista de tuples del tipus (nom, curs, edat)
>>> estudiants = [('john', 'A', 15), ('jane', 'A', 12),
...               ('dave', 'B', 10), ('ed', 'B', 12)]
```

```
# ordena per nom (per defecte: primer element dels tuples)
>>> estudiants.sort()
>>> estudiants
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# ordena per edat
>>> estudiants = [('john', 'A', 15), ('jane', 'A', 12),
...               ('dave', 'B', 10), ('ed', 'B', 12)]
>>> estudiants.sort(key=lambda estudiant: estudiant[2])
>>> estudiants
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]

# ordena primer per edat i després per nom
>>> estudiants = [('john', 'A', 15), ('jane', 'A', 12),
...               ('dave', 'B', 10), ('ed', 'B', 12)]
>>> estudiants.sort(key=lambda estud: (estud[2], estud[0]))
>>> estudiants
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# ordena primer per edat i després per curs
>>> estudiants = [('john', 'A', 15), ('jane', 'A', 12),
...               ('dave', 'B', 10), ('ed', 'B', 12)]
>>> estudiants.sort(key=lambda estud: (estud[2], estud[1]))
>>> estudiants
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]
```

Hi ha altres funcions que també es basen en l'ordre lexicogràfic que tenen el paràmetre opcional `key` com les funcions `max` i `min`.

# 7 Tuples

Els tuples són estructures de dades compostes per una agregació ordenada d'elements més simples: són un tipus seqüencial. Com les llistes, els tuples també poden ser homogenis o heterogenis.

Un tuple es representa amb els seus elements entre parèntesis i separats per comes: $(e_0, e_1, \ldots, e_n)$. En realitat els parèntesis no són obligatoris però, en general, es recomana posar-los i Python sempre representa els tuples tancats entre parèntesis.

```
>>> t1 = (3, 8, 5, 30, 23)  #tuple homogeni
#tuple heterogeni
>>> t2 = ('casa', 23, True, 45.78, 32, 'hola')
```

Un tuple buit es representa com `()`.

Un tuple amb un sol element `e` es representa com `(e,)`. La coma és necessària per distingir-lo d'un escalar. L'operador coma té menys prioritat que els altres i per tal que s'avaluï abans que els altres caldrà indicar-ho amb parèntesis.

Els tuples es poden operar en bloc amb les operacions de concatenació (+) i repetició (*) i es poden comparar amb els operadors de comparació `==, !=, <, >, <=, >=` que es basen en l'ordre lexicogràfic. La funció `len` de la biblioteca estàndard permet conèixer la longitud d'un tuple, és a dir, el nombre d'elements que té.

Els operadors de pertinença `in` i `not in` així com els mecanismes d'indexació i segmentació també són aplicables als tuples. L'índex també es posa entre claudàtors. Si `t` és un tuple, la sintaxi per accedir a un element és `a[index]`, $0 \leq index < len(t)$.

Exemples:

```
>>> (5)  # un enter
5
>>> (5,)  # un tuple amb un element
(5,)
>>> a = 3
>>> a + 4,  # l'operador + té més prioritat que l'operador ,
(7,)
>>> a = (5, 6, 7)
>>> a + 4, # l'operador + té més prioritat que l'operador ,
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
>>> b = a + (4,)  # concatenació de dos tuples
>>> b
(5, 6, 7, 4)
>>> b[1]
6
>>> b[1:]
(6, 7, 4)
>>> 7 in b
True
```

En l'expressió `(5, 6, 7) + 4`, s'aplica primer l'operador suma i, com que no es pot sumar un tuple i un enter, dóna error. Per concatenar un tuple amb 3 elements i un altre amb un element, cal representar ambdós tuples entre parèntesis.

La diferència fonamental entre els tuples i les llistes és que els tuples són immutables i les llistes són mutables.

## 7.1 Tuples i assignació múltiple

L'assignació de tuples es pot interpretar com una assignació múltiple vist a la introducció correspon realment a l'assignació de tuples. Exemples:

```
>>> m, n = 5, 6 # assignació de tuples
>>> m, n
(5, 6)
>>> m
5
>>> n
6
>>> a = 3, 4 # assignació de tuples: empaquetat
>>> a
(3, 4)
>>> x, y = a # assignació de tuples: desempaquetat
>>> x, y
(3, 4)
>>> x
3
>>> y
4
```

La primera línia és també una assignació de tuples equivalent a `a = (3, 4)`. Aquest tipus d'assignació entre tuples s'anomena **empaquetar** (pack) ja que s'assigna un tuple descrit pels seus elements a un tuple representat pel seu nom. El contrari d'empaquetar s'anomena **desempaquetar** (unpack) i s'assigna un tuple representat pel seu nom a un altre descrit pels seus elements tal com es mostra a la tercera línia. Les operacions d'empaquetar i desempaquetar també s'apliquen a strings i llistes:

```
>>> la = ['adeu', 'hola']
>>> s1, s2 = la
>>> s1
'adeu'
>>> s2
'hola'
>>> sa = 'xy'
>>> sa1, sa2 = sa
>>> sa1
'x'
>>> sa2
'y'
```

## 7.2 Tuples i funcions amb diversos valors de retorn

Les funcions Python sempre retornen un únic valor de retorn. Fins ara s'han dissenyat funcions que retornaven més d'un valor de retorn, però en realitat el que retornen és un únic valor: un tuple que Python mostra entre parèntesis.

Suposem que hem dissenyat la funció `exemple` que donats dos valors retorna la seva suma i el seu producte:

```
def exemple(a, b):
    return a+b, a*b
```

Aquesta funció en realitat retorna un únic objecte tuple amb dos elements.

```
# assignació de tuples: desempaquetat
>>> suma, prod = exemple(4, 5)
>>> suma, prod
(9, 20)
>>> t1 = exemple(3, 8) # assignació de tuples
>>> t1
(11, 24)
```

# 8 Exercicis de llistes i tuples

1. Un interval es representa amb un tuple de dos elements que representen els seus extrems inferior i superior. Donada una llista d'intervals, dissenya la funció `longitud` que retorna una altra llista amb les longituds d'aquells intervals que tinguin l'extrem inferior negatiu, en el mateix ordre que en la llista donada. Exemple:

```
>>> lintervals = [(-3, 3), (1.2, 4.8), (-56, 24.6),
...  (-20, -10)]
>>> longitud (lintervals)
[6, 80.6, 10]
```

```
def longitud (lintervals):
    llongituds = []
    for interval in lintervals:
        ei, es = interval
        if ei < 0:
            a = es - ei
            llongituds.append(a)
    return llongituds
```

   En aquest exercici es fa una assignació de tuples desempaquetant. També es duu a terme un procés d'aplicació en paral·lel amb un de filtratge.

2. Una matriu es representa com una llista de subllistes on cada subllista representa una de les files de la matriu. Dissenyeu la funció no modificadora `producte` que a partir d'una matriu i d'un valor numèric, retorni una altra matriu amb el resultat de multiplicar la matriu donada pel valor. Exemple:

```
>>> mat = [[2, 3, 4], [7, 5, 3], [8, 0, 9]]
>>> producte(mat, 2)
[[4, 6, 8], [14, 10, 6], [16, 0, 18]]
>>> mat
[[2, 3, 4], [7, 5, 3], [8, 0, 9]]
```

```
def producte(mat, esc):
    matnova = []
    for fila in mat:
        filanova = []
        for element in fila:
            filanova.append(element * esc)
        matnova.append(filanova)
    return matnova
```

3. Dissenyeu la funció modificadora `productemod` que a partir d'una matriu i d'un valor numèric, modifiqui la matriu donada de forma que sigui el resultat de multiplicar-la pel valor. Exemple:

```
>>> mat = [[2, 3, 4], [7, 5, 3], [8, 0, 9]]
>>> productemod(mat, 2)
>>> mat
[[4, 6, 8], [14, 10, 6], [16, 0, 18]]
```

```
def productemod(mat, esc):
    nfil = len(mat)
    ncol = len(mat[0])
    for i in range(nfil):
        for j in range (ncol):
            mat[i][j] = mat[i][j] * esc
```

Compareu els enunciats, els dissenys i els exemples dels dos exercicis anteriors per aprofundir en la comprensió dels conceptes de mutabilitat i funcions modificadores. Notem com la funció `producte` NO modifica la matriu i com SÍ que ho fa la funció modificadora `productemod`.

4. Una empresa representa el seu magatzem amb una llista de tuples on cada tuple representa un producte al magatzem amb el parell de dades: codi (string) i unitats (enter). Dissenya la funció `ocupació` que, a partir d'una llista de tuples com la indicada, retorni el tuple corresponent al producte del que hi ha més unitats al magatzem. Podem suposar que la llista no és buida.

```
>>> ll =[('AR35', 100), ('FT78',89), ('YH98', 34),
...  ('JH65', 120), ('UH56', 77), ('WE34', 76)]
>>> ocupacio(ll)
('JH65', 120)
```

```
def ocupacio (lmagatzem):
    maxim = -1
    for producte in lmagatzem:
        codi, unitats = producte
        if unitats > maxim:
            maxim = unitats
            codimax = codi
    return codimax, maxim

def ocupacio_1 (lmagatzem):
    return max(lmagatzem, key=lambda x: (x[1], x[0]))
```

Aquest és un exemple de síntesi corresponent al procés d'obtenció del màxim. La segona versió de la funció fa ús del paràmetre `key` de la funció `max`.

# 9 Diccionaris

## 9.1 Definició

Els tipus de dades compostos vistos fins ara, strings, llistes i tuples, són de tipus *seqüencial* i consisteixen en una agregació ordenada d'elements. El tipus diccionari és també un tipus de dades compost però és un tipus de dades *associatiu* i permet representar una aplicació entre dos conjunts.

Un diccionari és un conjunt de parells **clau:valor** que permeten representar l'aplicació del conjunt de les claus al conjunt dels valors. La clau és única i pot ser de qualsevol tipus immutable. Els valors poden ser de qualsevol tipus.

Un diccionari es representa tancant els seus elements (parells clau-valor) entre claus ({}) i separats per comes. A cada parell, el caràcter *dos punts* (`:`) separa la clau del valor.

```
{c1:v1, c2:v2,... , cN:vN}
```

El diccionari buit es representa com: {}. Exemples:

```
>>> d1 = {'pomes': 2.45, 'peres':2.45, 'cireres':3.5 }
>>> d2 = {(3, 4): 'blau', (3, 5): 'verd', (4, 5):'verd'}
>>> d3 = {'A': ['Alba', 'Anna'], 'B': ['Berta', 'Blanca'],
...       'C': ['Carme', 'Cèlia']}
```

El diccionari `d1` representa una aplicació de noms de productes (tipus string) a preus (tipus real), `d2` és una relació entre píxels donats per les seves coordenades (tipus tuple) i colors (tipus string) i `d3` és un diccionari on la clau és una lletra (string) i el valor una llista de noms propis femenins que comencen per aquesta lletra (llista d'strings).

Les claus han de ser úniques, però el valor no cal que ho sigui. Al diccionari `d1` el valor `2.45` està repetit, ja que el preu dels productes `'pomes'` i `'peres'` és el mateix, i al diccionari `d2` el valor `'verd'` també està repetit ja que el color dels píxels de coordenades `(3, 5)` i `(4, 5)` és el mateix: `'verd'`.

## 9.2 Indexació i operacions

L'accés als elements d'un diccionari, en no ser una estructura seqüencial, no es fa a través d'un índex enter. L'accés a un element d'un diccionari es fa a través de la clau. La sintaxi també usa els claudàtors:

`nom_diccionari[nom_clau]`

L'expressió `nom_diccionari[nom_clau]` és el valor corresponent a la clau `nom_clau`. Exemples:

```
>>> d1['pomes']      # accés o consulta a un element
2.45
>>> d2[(3, 5)]
'verd'
```

```
>>> d3['B']          # d3['B'] és una llista
['Berta', 'Blanca']
>>> d3['B'][1]       # 2n element de la llista d3['B']
'Blanca'
```

Els parells `clau:valor` es guarden en els diccionaris en un ordre arbitrari, determinat per Python i inaccessible pel programador.

Podem comparar diccionaris amb els operadors de relació, però l'únic que té sentit és comparar amb igualtat (`==`) o desigualtat (`!=`).

La funció `len` de la biblioteca estàndard dóna la longitud d'un diccionari, és a dir, el nombre de parells clau:valor que té.

Els operadors de pertinença `in` i `not in` són aplicables als diccionaris i usen la clau per determinar si un parell clau:valor pertany o no al diccionari. La sintaxi és:

```
nom_clau in nom_diccionari
```

Els diccionaris NO admeten les operacions de concatenació ni repetició ni el mecanisme de segmentació, propis de les estructures seqüencials.

Els diccionaris són un tipus de dades mutable i, per tant, podem inserir, modificar i esborrar elements d'un diccionari. Per inserir un nou element a un diccionari o per modificar el valor d'un element existent s'usa la mateixa sintaxi:

```
nom_diccionari[nom_clau] = valor
```

Si al diccionari no hi ha la clau `nom_clau`, aquesta operació insereix el parell `nom_clau-valor`. Si, en canvi, al diccionari ja hi un element amb aquesta clau, se'n modifica el seu valor.

Per esborrar un element d'un diccionari podem usar la sentència d'esborrat `del`:

```
del nom_diccionari[nom_clau]
```

Exemples:

```
>>> d1 = {'pomes': 2.45, 'peres':2.45, 'cireres':3.5 }
>>> d2 = {(3, 4): 'blau', (3, 5): 'verd', (4, 5):'verd'}
>>> d3 = {'A': ['Alba', 'Anna'], 'B': ['Berta', 'Blanca'],
...       'C': ['Carme', 'Cèlia']}
>>> d1 == d3
False
>>> len(d1)
3
>>> len(d3)
3
>>> 'A' in d3
True
>>> 'm' in d3
False
```

```
>>> (10, 10) in d2
False
>>> # inserció d'un nou element 'taronges':1.98
>>> d1 ['taronges'] = 1.98
>>> d1
{'pomes': 2.45, 'cireres': 3.5, 'peres': 2.45,
 'taronges': 1.98}
>>> d2[(4, 4)] = 'verd'      # inserció d'un nou element
>>> d2
{(4, 5): 'verd', (4, 4): 'verd', (3, 4): 'blau',
 (3, 5): 'verd'}
>>> d3['D'] = ['Dolors']    # inserció d'un nou element
>>> d3
{'D': ['Dolors'], 'A': ['Alba', 'Anna'],
 'C': ['Carme', 'Cèlia'], 'B': ['Berta', 'Blanca']}
>>> d3['A'] = []  # modificació del valor d'un element
>>> d3
{'D': ['Dolors'], 'A': [], 'C': ['Carme', 'Cèlia'],
'B': ['Berta', 'Blanca']}
>>> del d3['C']              # esborrat d'un element
>>> d3
{'D': ['Dolors'], 'A': [], 'B': ['Berta', 'Blanca']}
```

## 9.3 Recorregut i cerca en diccionaris

Podem aplicar esquemes de recorregut i cerca sobre els elements dels diccionari mitjançant la sentència `for` i usant les claus per accedir als elements. Python recorre els elements seguint l'ordre intern en que Python els guarda.

Notació:

```
for clau in diccionari:
    tractar clau, diccionari[clau]
```

## 9.4 Mètodes del tipus `dict`

El tipus diccionari (`dict`) disposa de diversos mètodes que es poden invocar usant la notació del punt:

`nom_variable_dict.nom_mètode(paràmetres actuals)`

En primer lloc hi ha els mètodes `keys`, `values` i `items` que retornen uns objectes anomenats *iteradors* que es poden convertir a llistes amb la funció `list`. Les llistes que s'obtenen són les de les claus, els valors i els tuples de dos elements (clau, valor), respectivament.

Exemples:

```
>>> d1 = {'pomes': 2.45, 'peres':2.45, 'cireres':3.5 }
>>> list(d1.keys())
['cireres', 'peres', 'pomes']
>>> list(d1.values())
[3.5, 2.45, 2.45]
>>> list(d1.items())
[('cireres', 3.5), ('peres', 2.45), ('pomes', 2.45)]
```

El mètode `get`, amb la següent especificació:

```
get(k[,d]):  D[k] if k in D, else d.  d defaults to None.
```

és equivalent a la funció següent:

```
def get_equivalent(D, k, d):
    if k in D:
        return D[k]
    else:
        return d
```

Els diccionaris són mutables i, de forma similar a les llistes, es pot produir l'efecte àlies. Quan s'assigna un diccionari a un altre aquests dos diccionaris esdevenen àlies l'un de l'altre. En el cas dels diccionaris si es vol evitar l'efecte àlies i obtenir una clonació (còpia sense efecte àlies) s'ha d'usar el mètode `copy` del tipus `dict`.

```
>>> d1 = {'pomes': 2.45, 'peres':3.25, 'cireres':3.5 }
>>> d2 = d1      # còpia amb efecte àlies: d1 i d2 són àlies
>>> d3 = d1.copy()  # clonació: d3 i d1 NO són àlies
>>> # modifiquem d1: d1 i d2 es modifiquen i d3 no
>>> d1['pomes'] = 1.98
>>> d1
{'cireres': 3.5, 'peres': 3.25, 'pomes': 1.98}
>>> d2
{'cireres': 3.5, 'peres': 3.25, 'pomes': 1.98}
>>> d3
{'pomes': 2.45, 'peres': 3.25, 'cireres': 3.5}
>>> del d2['peres']  # modifiquem d2: d1 i d2 es modifiquen
>>> d1
{'cireres': 3.5, 'pomes': 1.98}
>>> d2
{'cireres': 3.5, 'pomes': 1.98}
>>> d3
{'pomes': 2.45, 'peres': 3.25, 'cireres': 3.5}
>>> d3['peres'] = 2.15  # modifiquem d3: només es modifica d3
>>> d1
{'cireres': 3.5, 'pomes': 1.98}
>>> d2
{'cireres': 3.5, 'pomes': 1.98}
>>> d3
{'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}
```

## 9.5 Diccionaris i tests

En els exemples anteriors podem veure que el test consisteix en comparar si el diccionari obtingut és igual o no a l'esperat: s'usa una expressió booleana amb el signe d'igualtat (==) que dóna com a resultat `True` o `False`:

```
>>> d3 == {'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}
True
```

Si féssim el test tal com ho fem per altres tipus de dades:

```
>>> d3
{'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}
```

podria ser que obtinguéssim el diccionari `d3` com:

```
d3 = {'peres': 2.15, 'cireres': 3.5, 'pomes': 2.45}
```

Òbviament, el diccionari `d3` i l'esperat, encara que els seus elements no es mostren en el mateix ordre, són iguals i l'expressió:

```
d3 == {'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}
```

dóna `True`.

Ara bé, el mòdul doctest fa la comparació literal, és a dir, compara els strings:

```
''{'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}''
```

i

```
''{'peres': 2.15, 'cireres': 3.5, 'pomes': 2.45}''
```

i el resultat és `False`. Per tant, cal escriure els tests en la forma indicada, en què es compara si els diccionaris són iguals.

Podem dissenyar un test més explicatiu:

```
>>> if d3 != {'pomes': 2.45, 'peres': 2.15, 'cireres': 3.5}:
...     print(d3)
```

Si usem aquest test, quan el diccionari obtingut sigui diferent de l'esperat, el test mostrarà el diccionari obtingut i així la programadora el podrà comparar amb el diccionari esperat.

## 9.6 Exercicis

1. Es disposa d'un text on hi apareixen diverses quantitats numèriques i es vol modificar de forma que, si aquestes quantitats són inferiors a 10, hi apareguin amb la paraula corresponent en lloc del dígit. Podem suposar que les lletres del text són totes minúscules i no hi ha signes de puntuació. També es disposa d'un diccionari amb un parell per cada dígit, on la clau és un string amb un dígit i el valor un altre string amb la paraula corresponent.

   Dissenya la funció `canvinum` que a partir d'un text i un diccionari obtingui el text modificat tal com s'indica al paràgraf anterior. Exemple:

```
>>> dicc = {'0': 'zero', '1': 'un', '2': 'dos',
... '3': 'tres', '4': 'quatre', '5': 'cinc',
... '6': 'sis', '7': 'set', '8': 'vuit', '9': 'nou'}
>>> t1 = 'hi ha 5 taules i 3 cadires i només 1 armari'
>>> canvinum (t1, dicc)
'hi ha cinc taules i tres cadires i només un armari'
```

En aquest exercici s'usa un diccionari com una eina que permet aplicar modificacions a una altra estructura. El diccionari només es consulta.

```
def canvinum (text, dicc):
    lparaules = text.split()
    novall = []
    for paraula in lparaules:
        if paraula in dicc:
            novall.append(dicc[paraula])
        else:
            novall.append(paraula)
    return ' '.join(novall)
```

Notem que, en l'exemple donat, les paraules per expressar les quantitats de un i dos només hi són en masculí i, per tant, el text `han vingut 2 noies i 1 nena` sortiria modificat com `han vingut dos noies i un nena`. Analitzeu la dificultat de dissenyar una funció que tracti correctament aquests casos.

2. Es disposa d'una llista d'strings amb els números de loteria de Nadal que han rebut primers premis els anys anteriors. Dissenyeu una funció que retorni un diccionari on la clau sigui la xifra amb la que acaba un número (string) i el valor el nombre de vegades que hi ha hagut primers premis acabats amb aquella xifra. Es demana que es dissenyin dues funcions diferents que resolguin aquest problema.

   La primera funció, de nom `acabaments1`, ha de retornar un diccionari complet, és a dir, amb els 10 possibles elements. Exemple:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> d = acabaments1(lot)
>>> d == {'8': 0, '9': 0, '0': 0, '6': 0, '3': 0,
...       '7': 0, '2': 2, '5': 3, '4': 1, '1': 3}
True
```

```
def acabaments1 (loteria):
    digits = '0123456789'
    dacab = {}
    for d in digits:
        dacab[d] = 0
    for num in loteria:
        acab = num[-1]
        dacab[acab] = dacab[acab] + 1
    return dacab
```

Aquest és un exemple de creació d'un diccionari que representa una taula de freqüències, que és una de les aplicacions dels diccionaris. En aquest cas, cal inicialitzar el diccionari amb totes les possibles claus ja que l'enunciat demana un diccionari complet. A més, com que els valor són comptadors, cal inicialitzar-los tots a `0`.

La segona funció, de nom `acabaments2`, ha de retornar un diccionari que no cal que sigui complet: només hi haurà les claus corresponents als acabaments que hi hagi en els números de la llista entrada. Exemple:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> d = acabaments2(lot)
>>> d == {'2': 2, '5': 3, '4': 1, '1': 3}
True
```

```
def acabaments2 (loteria):
    dacab = {}
    for num in loteria:
        acab = num[-1]
        if acab in dacab:
            dacab[acab] = dacab[acab] + 1
        else:
            dacab[acab] = 1
    return dacab
```

En aquest cas no cal inicialitzar el diccionari amb totes les possibles claus ja que el diccionari no cal que sigui complet. Els parells del diccionari es creen sobre la marxa: si la clau no és al diccionari, s'insereix aquesta clau amb el valor del comptador a `1`. Si la clau ja hi és, s'incrementa el corresponent valor en una unitat.

3. Es disposa d'una llista d'strings amb els números de loteria de Nadal que han rebut primers premis els anys anteriors. Dissenyeu la funció `premiats` que a partir d'una llista com la descrita, retorni un diccionari on la clau sigui la xifra amb la que acaba un número (string) i el valor una llista amb tots els números acabats en aquella xifra, en el mateix ordre en què estan a la llista donada. El diccionari no cal que sigui complet. Exemple:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> premiats (lot)
{'2': ['65432', '98762'], '5': ['12345', '98765', '01235'],
 '4': ['98654'], '1': ['54321', '98541', '76781']}
```

```
def premiats (loteria):
    premi = {}
    for num in loteria:
        acab = num[-1]
        if acab in premi:
            premi[acab].append(num)
        else:
            premi[acab] = [num]
    return premi
```

Aquest és un exemple de creació d'un diccionari on els valors són llistes. En aquest cas, no cal inicialitzar el diccionari amb totes les possibles claus ja que el diccionari no cal que sigui complet. Els parells del diccionari es creen sobre la marxa: si la clau no hi és s'insereix al diccionari i el valor corresponent és una llista amb un sol element: el número en curs. Si la clau ja és al diccionari, s'afegeix el número al final de la llista corresponent amb el mètode `append`.

4. Les despeses d'un viatger venen representades en un diccionari on la clau és un enter indicant el dia i el valor una llista de reals indicant les despeses.

   (a) Dissenya la funció `despesa_total` que, a partir d'un diccionari com l'indicat, retorni la suma total de les despeses. Exemple:

```
>>> d = {12:[2,3,4,5], 13:[6,7,6,8], 14:[50,10,4,5],
...      15:[4,6,10,12]}
>>> despesa_total(d)
142
```

   En aquest exercici apliquem un esquema de recorregut al diccionari.

```
def despesa_total (despeses):
    total = 0
    for dia in despeses:
        total = total + sum(despeses[dia])
    return total
```

   (b) Dissenyeu la funció `undiaesundia` que, a partir d'un diccionari com l'indicat i una determinada quantitat (tot en la mateixa moneda), retorni un enter corresponent al primer dia que es trobi en el diccionari en què la despesa hagi superat aquella quantitat. Si cap dia se supera la quantitat donada, la funció ha de retornar el valor `0`. Exemple:

```
>>> d = {12:[2,3,4,5], 13:[6,7,6,8], 14:[50,10,4,5],
...      15:[4,6,10,12]}
>>> undiaesundia(d, 60)
14
>>> undiaesundia(d, 100)
0
```

   En aquest exercici apliquem un esquema de recorregut al diccionari.

```
def undiaesundia (despeses, quant):
    trobat = False
    for dia in despeses:
        trobat = sum(despeses[dia]) > quant
        if trobat:
            eldia = dia
            break
    if trobat:
        return eldia
    else:
        return 0
```

# 10 Mòduls i fitxers

## 10.1 Mòduls

Un mòdul és un fitxer que conté definicions i sentències Python que poden ser usats en altres programes i funcions Python.

Python disposa de molts mòduls. La biblioteca estàndard de Python està formada per diversos mòduls. En aquesta biblioteca hi ha les funcions de conversió de tipus: `int`, `float`, `str`, `list`, etc., i altres funcions que ja s'han vist com `eval`, `sum`, `max`, `min`, etc. Hi ha altres mòduls com el mòdul `math`, amb les constants i funcions que es poden trobar en una calculadora científica. També hi ha el mòdul `random` amb funcions que permeten treballar amb nombres aleatoris, el mòdul `time` amb funcions per cronometrar els programes o el mòdul `keyword` que té funcions com `iskeyword` que indica si una paraula és reservada o no i la llista `kwlist` que mostra totes es paraules reservades.

Altres mòduls i biblioteques : `numpy` per treballar amb matrius, `pygame` per dissenyar jocs per computador, `pandas` per treballar amb bases de dades, *Python Imaging Library*, per treballar amb imatges, `networkx` per treballar amb grafs, etc.

Excepte per la biblioteca estàndard, per usar un mòdul cal importar-lo amb la sentència `import` i per usar un dels seus elements cal fer servir la notació del punt.

A continuació es mostren exemples amb els mòduls `keyword` i `math`:

```
>>> import keyword
>>> print(keyword.kwlist)
['and', 'as', 'assert', 'break', 'class', 'continue', 'def',
 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for',
 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',
 'while', 'with', 'yield']
>>> keyword.iskeyword('for')
True
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(math.pi/3)
0.5
```

## 10.2 Creació de mòduls

Les aplicacions informàtiques es divideixen en un determinat nombre de mòduls i això facilita el seu disseny.

Un mòdul és un fitxer amb l'extensió `.py` i, de fet, tots els fitxers que hem creat fins ara amb una o més funcions són mòduls. Suposem que el mòdul `modul1.py` conté la funció `fun1` i el mòdul `modul2.py` conté la funció `fun2`, i que `fun1` crida a `fun2`. En aquest cas, a l'inici del mòdul `modul1.py` hi ha d'haver la sentència:

```
from modul2 import func2
```

## 10.3 Àmbit de visibilitat (Espai de noms)

Un *espai de noms* és la col·lecció d'identificadors que està inclòs en una funció o en un mòdul. També ens podem referir a aquest concepte com *àmbit de visibilitat*.

## 10.4 Fitxers seqüencials de text (FST)

Un fitxer informàtic és un recurs informàtic per enregistrar dades en un dispositiu d'emmagatzematge d'un ordinador. Mentre s'executa un programa, les seves dades (variables, valors, etc.) s'emmagatzemen a la memòria d'accés aleatori (RAM). La memòria RAM és ràpida, però és volàtil, cosa que significa que quan finalitza el programa aquestes dades desapareixen. Les dades també es poden emmagatzemar en un dispositiu d'emmagatzematge no volàtil, com ara un disc dur o una memòria USB.

Sovint les aplicacions informàtiques necessiten treballar amb dades emmagatzemades en fitxers, de manera que les dades estiguin disponibles cada vegada que s'executa el programa. En concret, necessitem programes i funcions per poder obtenir (llegir) dades i per escriure dades en un fitxer. Usant fitxers, els programes poden desar informació entre execucions.

Els fitxers són bàsicament una seqüència de bytes. Hi ha una àmplia tipologia de fitxers informàtics, però en aquest curs tractarem un tipus específic: **fitxers seqüencials de text** (FST). Un FST és un fitxer que consta d'una seqüència de caràcters.

Un fitxer d'accés **seqüencial** és un fitxer que s'ha de recórrer sempre des del començament fins al final. No es pot anar enrere ni saltar cap a una determinada posició: aquesta restricció implica que no es pot llegir i escriure en un fitxer al mateix temps. En contrast, els **fitxers d'accés directe** permeten accedir directament a posicions especificades.

Un fitxer **de text** és un fitxer compost només per bytes que corresponen a caràcters imprimibles i, per tant, aquests fitxers es poden obrir i editar amb un editor de text sense format. En contraposició, els fitxers binaris són fitxers generals en els quals els bytes poden tenir altres significats que la codificació ASCII per a caràcters imprimibles. Per tant, no es poden editar fitxers binaris. Per exemple, un fitxer amb extensió `.py` que conté codi Python és un fitxer de text (codi font Python). D'altra banda, un fitxer amb extensió `.pyc` és un fitxer binari i no es pot llegir amb un editor de text: els bytes d'aquest fitxer corresponen a les instruccions de Python traduïdes a llenguatge màquina. Els fitxers binaris amb extensió `.pyc` els construeix automàticament Python quan importem un mòdul. Això fa que quan es torna a importar el fitxer, aquest procés sigui més ràpid.

Un FST està compost únicament per caràcters organitzats en **línies** (també dits **registres**). Les línies acaben amb el caràcter salt de línia (`'\n'`). Les línies poden ser de longitud fixa o variable. Els fitxers es poden estructurar com les taules usades en les bases de dades, és a dir, amb totes les línies estructurades en **camps d'informació**. Per exemple, un fitxer on s'emmagatzema la informació dels productes d'una empresa on a cada línia hi ha la mateixa informació de cada producte: el codi, el preu i el nombre d'unitats al magatzem.

Com tots els fitxers, els fitxers de dades també han de tenir un nom. És recomanable que els fitxers de dades estiguin en el directori de treball on hi ha el mòdul (amb les funcions Python) que s'està executant. Si no es fa així, el nom del fitxer haurà d'anar

precedit del Camí necessari per arribar al directori on és, a través de l'arbre de directoris (*path*). Els fitxers de dades no cal que tinguin una extensió, però com que són fitxers de text, els hi podem posar l'extensió `.txt`. Exemples de noms de fitxers: `dades.txt`, `productes.txt`, `estudiants.txt`, .... Des de fora del programa o funció Python, podem consultar, modificar i crear fitxers de text amb qualsevol editor de text com l'editor de l'aplicació `idle` o l'editor `emacs`, entre altres.

## 10.5 FST i Python

Python ofereix un tipus de dades que permet manipular fitxers. Un fitxer usat des d'una funció Python ha d'estar representat per una variable d'aquest tipus. El nom amb què Python anomena aquest tipus de dades és: `io.TextIOWrapper`; per simplificar, en aquest capítol parlarem de `tipus fitxer`.

El fitxer guardat en la memòria d'emmagatzematge s'anomena **fitxer extern** i a la variable de tipus fitxer usada en la funció s'anomena **fitxer intern**.

Un programa o funció Python abans de llegir o escriure en un fitxer ha d'establir un protocol de comunicació (canal, enllaç) entre el fitxer extern i el fitxer intern. Aquest canal s'estableix amb la crida a la funció de la biblioteca estàndard `open`. Quan s'acaba de treballar amb un fitxer, cal tancar el canal de comunicació. La funció `open` es pot cridar directament i aleshores cal tancar explícitament el fitxer. Si es crida la funció `open` amb la sentència `with` el fitxer es tanca implícitament.

La sintaxi de la funció `open` amb la sentència `with` és la següent:

```
with open(nom_extern, mode) as nom_intern
```

- `nom_extern` és una variable o valor de tipus string que conté el nom del fitxer.
- `mode` és també una variable o valor de tipus string que indica el mode en què s'obre el fitxer: lectura ('r') o escriptura ('w'). Quan s'obre un fitxer en mode lectura, el fitxer ha d'existir. Quan s'obre un fitxer en mode escriptura, es crea el fitxer. Si ja existia un fitxer amb aquell nom, en obrir-lo per escriptura es destrueix la informació que hi havia abans.
- `nom_intern` és una variable de tipus fitxer.

Exemple:

```
>>> with open ('primer.txt', 'r') as f1:
...     with open ('segon.txt', 'w') as f2:
```

Obre un fitxer de nom `primer.txt` en mode lectura i l'associa a la variable de tipus fitxer `f1`. Si el fitxer `primer.txt` no existeix o no és al directori de treball, dóna error. Després obre un altre fitxer de nom `segon.txt` en mode escriptura i l'associa a la variable de tipus fitxer `f2`. El fitxer `segon.txt` pot existir o no, però si existeix, el seu contingut desapareixerà.

La sentència `with` permet obrir diversos fitxers. Exemple:

```
>>> with open ('n.txt', 'r') as f, open ('m.txt', 'w') as g:
```

El tipus fitxer disposa dels mètodes següents.

Mètodes per llegir:

```
f.read(size)
    llegeix un màxim de 'size' caràcters del fitxer f
    que retorna com un string

f.readline(size)
    llegeix la línia següent del fitxer f
    que retorna com un string

f.readlines()
    llegeix tot el fitxer f
    retorna una llista d'strings, cadascun corresponent a una línia
```

Mètode per escriure:

```
f.write(str)
    escriu l'string str al fitxer f
    retorna un enter (la longitud de str)
```

Mètode per tancar

```
f.close()
    Tanca el fitxer. No retorna res
```

## 10.6 Recorregut d'un fitxer

Els fitxers de text consten d'una o diverses línies que finalitzen amb un caràcter salt de línia. La sentència `for` proporciona una forma natural per recórrer totes les línies d'un fitxer així com per aplicar-hi un esquema de cerca:

```
with open ('dades.txt', 'r') as f:
    for línia in f:
        tractar línia
```

En aquest exemple s'obre un fitxer de nom `dades.txt` en mode lectura i s'associa a la variable de tipus fitxer, `f`. La sentència `for` fa un recorregut seqüencial de les línies de manera similar a com fa un recorregut seqüencial pels caràcters d'un string o pels elements d'una llista.

La variable `línia` conté una línia del fitxer: a la primera iteració, la primera línia i a cada nova iteració, la línia següent. Els fitxers de text contenen caràcters i, per tant, la variable `línia` és un string amb tots els caràcters de la línia i el caràcter final de salt de línia inclòs. La sentència `for` obté una nova línia automàticament a cada iteració i també s'atura automàticament quan detecta el final del fitxer.

Per tal de processar una línia moltes vegades cal fer un procés de descodificació. Suposem que el fitxer de nom `dades.txt` de l'exemple anterior és un fitxer de productes on, a

cada línia, hi ha la informació d'un producte: el codi (`str`), el preu (`float`) i el nombre d'unitats al magatzem (`int`), separats per un guió. Suposem que un producte té el codi 'DR45', un preu de 56.75 i un nombre d'unitats de 200. A la corresponent iteració del bucle for, la variable `línia` és una variable `str` amb el contingut següent:

```
'DR45-56.75-200\n'
```

Descodificar aquesta cadena de caràcters vol dir obtenir els corresponents elements amb els corresponents tipus: el codi del producte (`str`), el preu (`float`) i el nombre d'unitats al magatzem (`int`). El proces de descodificar es pot descompondre en els passos següents:

- *netejar*: es treuen els caràcters no necessaris a l'inici o final de la línia, com el salt de línies. Es pot fer de diverses maneres:
  - amb el mètode `strip` del tipus `str`. Aquest mètode retorna una còpia de l'string sense els caràcters a l'inici i al final especificats. Si no s'especifiquen els caràcters, treu els caràcters whitespace.
  - `línia = línia[:-1]`: només elimina el darrer caràcter
  - `línia.replace(('\n','')`: elimina tots els possibles salts de línia
- *separar*: la línia es parteix en els camps d'informació usant amb el mètode `split` del tipus `str`. Recordem les dues modalitats de funcionament del mètode `split`. Si els camps d'informació estan separats per un caràcter diferent de l'espai en blanc, cal posar aquest caràcter com a paràmetre del mètode. Si els camps d'informació estan separats per espais en blanc, no cal posar l'espai en blanc com a paràmetre del mètode i quan s'aplica també s'eliminen els salts de línia. Per tant, en aquest cas, l'aplicació del mètode `split` no només separa sinó que també neteja.
- *descodificar*: els camps d'informació es converteixen al tipus adequat. Cal usar les funcions `int`, `float` i `eval` per convertir els strings a enter, real i booleà respectivament

Tornem a mostrar el contingut de la variable `línia`:

```
'DR45-56.75-200\n'
```

I ara es mostra l'aplicació de les tres operacions esmentades:

```
>>> línia = línia.strip()       # neteja
>>> línia
'DR45-56.75-200'
>>> linfo = línia.split('-')    # separació
>>> linfo
['DR45', '56.75', '200']
>>> codi = linfo[0]      # no cal descodificar els strings
>>> preu = float(linfo[1])      # descodificació
>>> unitats = int(linfo[2])     # descodificació
>>> codi, preu, unitats
('DR45', 56.75, 200)
```

Si els camps d'informació de `línia` estiguessin separats per espais en blanc, `'DR45 56.75 200\n'`, les operacions per descodificar la línia serien:

```
>>> linfo = línia.split()     # neteja i separació
>>> codi = linfo[0]
>>> preu = float(linfo[1])    # descodificació
>>> unitats = int(linfo[2])   # descodificació
```

Per altra banda, a l'hora d'escriure en un fitxer de text, també cal codificar la informació a escriure a cada línia com un string que ha de tenir com a darrer caràcter el salt de línia. Si la informació a escriure és de tipus enter, real o booleà, cal usar la funció `str` per convertir-la a string

## 10.7 Exercicis

1. Un fitxer conté un dni a cada línia. Com exemple, suposem que disposem d'un fitxer de nom `dni.txt` amb el contingut següent:

   ```
   38765898Y
   78654321P
   56898564L
   38560367K
   98546456H
   87564021G
   19823456F
   87632369K
   ```

   Dissenya la funció `dni1` que donat el nom d'un fitxer d'aquestes característiques i una lletra, retorni un enter indicant el nombre de dnis que acaben en aquesta lletra. Exemples:

   ```
   >>> dni1('dni.txt','K')
   2
   >>> dni1('dni.txt','L')
   1
   >>> dni1('dni.txt','X')
   0
   ```

   ```
   def dni1 (nomf, lletra):
       with open(nomf, 'r') as f:
           n = 0
           for linia in f:
               linia = linia.strip()
               if linia[-1] == lletra:
                   n = n + 1
       return n
   ```

   En aquest primer exemple cal insistir en els conceptes de fitxer extern i fitxer intern i repassar el concepte de paràmetre formal i actual d'una funció així com la sintaxi dels noms de variables i paràmetres i la dels valors de tipus `str` (col·loquialment: quan s'han de posar cometes i quan no). La funció `dni1` té dos paràmetres de tipus `str`, un dels quals és `nomf` que fa referència al nom del fitxer extern i l'altre és la

`lletra`. Són els paràmetres formals de la funció, són identificadors i, per tant, no van entre cometes. En canvi, a la crida a la funció, `dni1('dni.txt','K')`, hi ha els paràmetres actuals que són expressions de tipus `str`. En aquest cas cada expressió és un valor de tipus `str`: `'dni.txt'` i `'K'`, respectivament i, per tant, van entre cometes. En el cos de la funció es defineix la variable `f`, que és de tipus fitxer.

El fitxer de nom `dni.txt` ha d'existir en el directori de treball, sinó en intentar obrir-lo donaria error. Les línies d'aquest fitxer només tenen un camp d'informació. A més, en aquest exercici només s'han de fer operacions amb strings. Per tant, només cal netejar la línia; no cal separar ni descodificar. En realitat, tampoc caldria netejar substituint la sentència `if linia[-1] == lletra:` per `if linia[-2] == lletra:`. Aquest és un procés de síntesi on s'obté un comptador.

2. Dissenya la funció `dni2` que, donat el nom d'un fitxer de les característiques del de l'exercici anterior, un enter entre `1` i `99`, i el nom de dos altres fitxers, escrigui en el primer d'aquests fitxers els dnis tals que l'enter que defineixen les seves dues xifres inicials sigui més petit que l'enter donat i escrigui a l'altre fitxer la resta de dnis. Les línies d'aquests fitxers han d'estar en el mateix ordre que en el fitxer inicial. Exemples: la crida següent:

```
>>> dni2('dni.txt', 40, 'sortida1.txt', 'sortida2.txt')
```

crearà els fitxers de nom `'sortida1.txt'` i `'sortida2.txt'`, amb els continguts:

| 'sortida1.txt' | 'sortida2.txt' |
|---|---|
| 38765898Y | 78654321P |
| 38560367K | 56898564L |
| 19823456F | 98546456H |
| | 87564021G |
| | 87632369K |

```
def dni2 (nomf, nombre, nomf1, nomf2):
    with open(nomf, 'r') as f, open(nomf1, 'w') as f1, \
         open(nomf2, 'w') as f2:
        for linia in f:
            if int(linia[:2]) < nombre:
                f1.write(linia)
            else:
                f2.write(linia)
```

En aquesta funció obrim 3 fitxers, un per lectura i dos per escriptura. Usem la mateixa sentència `with` per tots tres. Cada fitxer s'ha d'associar a una variable de tipus fitxer diferent ja que tots tres estan oberts en paral·lel. El caràcter barra invertida permet continuar una sentència a la línia següent. Els dos fitxers d'escriptura no han d'existir; si existissin s'esborrarien.

En aquest cas, no netegem les línies perquè no cal i així quan s'escriuen als fitxers de sortida, no cal posar el salt de línia. Cal descodificar els dos primers caràcters de cada línia, convertint-los a enter, per poder-los comparar amb el nombre donat. També s'hagués pogut fer sense descodificar, convertint el nombre donat a string. En aquest disseny s'ha aplicat un procés de filtratge.

3. Dissenya la funció `dni3` que donat el nom d'un fitxer de les característiques del de l'exercici anterior, retorni `True` si hi ha algun dni que té alguna xifra 0 i `False` altrament. Exemple

```
>>> dni3.dni3('dni.txt')
True
```

```
def dni3 (nomf):
    with open(nomf, 'r') as f:
        trobat = False
        for linia in f:
            trobat = '0' in linia
            if trobat:
                break
    return trobat
```

En aquest exemple s'ha aplicat un esquema de cerca. Tampoc ha calgut processar la línia.

4. Una empresa té representat l'estat actual del seu magatzem de productes en un fitxer on, a cada línia, hi ha la informació d'un producte, que consta de 4 camps: el codi (`str`), les unitats (`int`), el tipus de producte, que tipifica la seva valoració (`str`) i un booleà que indica si aquest producte requereix d'un magatzem especial o no. Aquestes dades estan separades pel caràcter dos punts. També disposem d'un diccionari on la clau és el tipus de producte i el valor el preu actual del producte.

   Dissenya la funció `valor` que a partir del nom d'un fitxer i d'un diccionari de les característiques descrites, escrigui en un fitxer que s'ha de dir `'stockespecial.txt'` només els productes que requereixen emmagatzematge especial. A cada línia d'aquest fitxer hi ha d'haver la informació següent d'aquests productes: el seu codi i el seu valor total (les unitats multiplicades pel preu) separats pel caràcter dos punts. A més, aquesta funció ha de retornar el valor total dels productes guardats al magatzem especial. Exemple: per un fitxer de nom `'stock2016.txt'` amb el contingut següent:

```
FG89:25:A:True
HJ96:15:B:False
AD45:200:C:True
XZ65:124:C:False
TY76:89:A:True
TH89:250:A:True
HJ56:85:B:False
AH76:20:C:True
MV45:24:C:False
TY89:123:A:True
```

   la funció ha de passar el test següent:

```
>>> dicc = {'A': 245, 'B': 123, 'C': 89}
>>> valor.valor('stock2016.txt', dicc)
138895
```

i es crearà un fitxer de nom `'stockespecial.txt'` amb el contingut següent:

```
FG89:6125
AD45:17800
TY76:21805
TH89:61250
AH76:1780
TY89:30135
```

```
def valor (nomf1, dicc):
    suma = 0
    with open(nomf1, 'r') as f1, \
         open('stockespecial.txt', 'w') as f2:
        for linia in f1:
            linia = linia.strip()
            linfo = linia.split(':')
            codi = linfo[0]
            unitats = int(linfo[1])
            tipus = linfo[2]
            especial = eval(linfo[3])
            if especial:
                preu = dicc[tipus]
                val = unitats * preu
                suma = suma + val
                f2.write(codi + ':' + str(val)+ '\n')
    return suma
```

En aquest exemple notem que el fitxer d'entrada ens el donen com a paràmetre però el fitxer de sortida s'ha de dir sempre el mateix nom, `'stockespecial.txt'`. Observem l'ús diferent de la funció `open` en els dos casos.

En aquest disseny s'aplica un esquema de filtratge (els productes que necessiten emmagatzematge especial). Cal aplicar les tres operacions de neteja, separació i descodificació a les línies, obtenint 4 variables: el codi, les unitats, el tipus i el booleà (que s'ha de descodificar amb la funció `eval`). Per calcular els valors cal obtenir el preu del producte indexant el diccionari donat amb el tipus de producte.

Un cop s'ha obtingut la informació per cada línia que ha d'anar al fitxer de sortida, aquesta informació s'ha de codificar en un string. L'expressió:

```
codi + ':' + str(val)+ '\n'
```

converteix la variable `val` de real a string i concatena els 4 strings: la variable `codi`, el valor `:`, la variable `val` convertida a string i el valor salt de línia, `'\n'`.

# 11 Iteració: sentència while

## 11.1 Composició repetitiva: sentències for i while

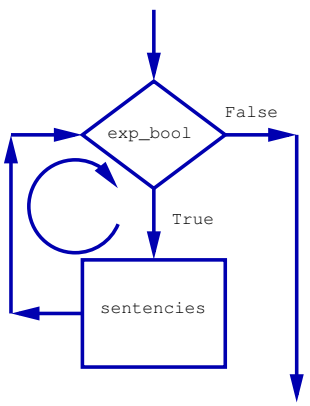
Fins ara hem aplicat la composició repetitiva en els esquemes de recorregut i cerca aplicats a les estructures de dades strings, llistes, tuples, diccionaris i fitxers. S'ha usat la sentència `for` especialment dissenyada per aquests casos. Ara bé, no tots els problemes que requereixen una composició repetitiva es poden resoldre usant la sentència `for`.

La sentència `for` permet fer repeticions de forma controlada i segura: recorre tots els elements de l'estructura indicada, des del primer fins al darrer:

```
for elem in estructura:
    tractar element
```

La sentència `while` és la sentència repetitiva general que permet dissenyar tot tipus de composicions iteratives. És una sentència de més baix nivell. La seva sintaxi és la que es mostra a continuació:

```
while exp_bool:
   sentències
```




La figura anterior mostra el diagrama de flux d'aquesta sentència. El flux del programa avalua l'expressió booleana. Si aquesta s'avalua a `True`, s'executa el grup de sentències i es torna a avaluar l'expressió booleana i així successivament. Quan l'expressió booleana s'avalua a `False`, el flux del programa ja no executa el grup de sentències i continua endavant. És a dir, mentre (`while`) es compleixi l'expressió booleana, s'executa el grup de sentències i el programa roman en el bucle fins que l'expressió booleana es deixa de complir.

## 11.2 Disseny de composicions iteratives amb while

La sentència `while` permet dissenyar composicions iteratives generals. En el disseny d'aquest tipus de composicions és molt fàcil cometre errors del tipus:

- no tractar tots els elements que es vol tractar

- no sortir mai del `while`: *bucle sense fi*

Exemple:

```
a = 5
while a < 10:
   a = a - 1
```

Aquest és un exemple de bucle sense fi: un cop el flux del programa entra dins el `while`, no en surt mai més.

En el disseny de composicions iteratives amb la sentència `while` la depuració amb la tècnica de prova i error és insuficient i cal una metodologia més sòlida per garantir la correctesa.

Atès que les dades que es manipulen en les composicions iteratives tenen estructura de seqüència, la metodologia que s'aplica es basa en identificar i caracteritzar aquesta seqüència i, tot seguit, aplicar un esquema de recorregut o cerca.

## 11.3 Seqüències

Una seqüència és un conjunt de dades de qualsevol tipus que es caracteritzen per estar disposades linealment, ja sigui des d'un punt de vista físic o conceptual.

Més formalment, una seqüència és un conjunt de dades que ve caracteritzada per tres propietats:

- conté un element que és el *primer* de la seqüència
- es pot definir la relació de *següent*: donat un element de la seqüència, permet obtenir l'element que ve a continuació. Aquest concepte també s'anomena *relació de successió*
- és finita. Es pot definir una propietat que compleixen tots els elements de la seqüència, o que no compleixen els elements que no són de la seqüència: propietat d'acabament.

La identificació d'una seqüència consisteix a determinar el tipus de les dades i la caracterització consisteix a determinar quin és el primer element, com s'obté el següent i la propietat d'acabament.

Exemples:

- la seqüència dels múltiples de *3* positius i inferiors a *100* és una seqüència d'enters. Una possible caracterització és la següent: el primer element d'aquesta seqüència és el *3*, el següent s'obté sumant *3* a l'anterior i tots els elements han de ser inferiors a *100*.
- la seqüència de les potències de *2* inferiors a *2000*. Una possible caracterització és la següent: el primer element d'aquesta seqüència és l'*1*, el següent s'obté multiplicant per *2* l'anterior i tots els elements han de ser inferiors a *2000*.

- els punts de la funció $y = f(x) = x^2 + 4x + 2$ dins l'interval $[-5, 5]$ amb increments d'abscissa d'una unitat són una seqüència de punts. El primer element és $(x, y) = (-5, f(-5))$; per obtenir el següent, cal sumar *1* a l'abscissa, $x$, i calcular $f(x)$. La propietat d'acabament es compleix quan l'abscissa és més gran que 5

## 11.4 Iteració amb while i seqüències

Tot problema que requereix l'ús d'una composició iterativa general té una seqüència associada i en el disseny de la composició iterativa hi intervenen tres elements que tenen una relació directa amb la caracterització de la seqüència associada.

```
sentències abans de while
while exp_bool:
    sentències dins de while
```

La metodologia de disseny seqüencial es basa en els següents punts:

- les sentències que s'executen abans del `while` tenen relació amb el primer element de la seqüència
- les sentències que s'executen dins del `while` tenen relació amb el següent element de la seqüència
- l'expressió booleana del `while` està relacionada amb la propietat d'acabament de la seqüència

Així doncs, per dissenyar una composició iterativa general, cal dur a terme els següents passos:

1. Identificar la seqüència associada
2. Caracteritzar la seqüència associada.
3. Identificar l'esquema: recorregut o cerca
4. Dissenyar la composició iterativa

## 11.5 Esquema de recorregut

L'esquema de recorregut és el que cal aplicar quan el problema a resoldre ha de generar o visitar sempre tots els elements de la seqüència. Aquest esquema amb la sentència `while` és el següent:

```
obtenir primer element
while not propietat acabament:
    tractar element
    obtenir següent element
```

Unes sentències que s'executen abans del `while` fan la tasca d'obtenir el primer element de la seqüència. Unes sentències que s'executen dins del `while` fan la tasca d'obtenir el següent element de la seqüència. I la `propietat d'acabament` és una expressió booleana que s'avalua a False per tots els elements de la seqüència i a True quan s'acaben els elements de la seqüència.

## 11.6 Esquema de cerca

L'esquema de cerca és el que cal aplicar quan el problema a resoldre no sempre ha de generar o visitar tots els elements de la seqüència. Aquest esquema amb la sentència `while` és el següent:

```
trobat = False
obtenir primer element
while not propietat acabament:
    trobat = condició_de_cerca (element)
    if trobat:
       break
    obtenir següent element
if trobat:
    tractament_trobat(element)
else:
    tractament_no_trobat (element)
```

## 11.7 Recorregut i cerca en estructures amb while

Totes les estructures vistes fins ara (strings, llistes, diccionaris tuples i fitxers) disposen de mecanismes de caracterització que la sentència `for` usa de forma automàtica.

En general no és necessari usar la sentència `while` per recórrer aquestes estructures però, exceptuant els diccionaris, també es pot utilitzar.

Totes les estructures seqüencials, strings, llistes i tuples, es poden recórrer usant la sentència `while` fent servir com a seqüència associada la seqüència dels índexs. Veieu l'exercici 4.

Els diccionaris, en no tenir índex, només es poden recórrer amb la sentència `for`.

Els fitxers també es poden recórrer usant la sentència `while` i fent servir com a seqüència associada la seqüència de les línies del fitxer. El mètode `readline` permet obtenir tant el primer element com els següents elements d'aquesta seqüència. També permet avaluar la condició d'acabament: quan el mètode `readline` retorna l'string buit vol dir que s'ha arribat al final del fitxer

```
linia = f.readline()
while linia != '':
    tractar linia
    linia = f.readline()
```

## 11.8 Comptador d'iteracions

Quan es caracteritzen determinades seqüències que sovint corresponen a successions matemàtiques, determinar el primer element i el següent pot deduir-se directament de la definició de la successió. Ara bé, en alguns casos és difícil determinar la condició d'acabament si, per exemple, aquesta seqüència té un comportament cíclic. Per tal d'evitar que la funció es quedi en un bucle sense fi, es pot usar la tècnica del **comptador d'iteracions**. Aquesta tècnica consisteix en comptar les iteracions i sortir del bucle o bé quan es compleix la condició desitjada o bé quan es supera un determinat nombre d'iteracions. Exemple:

Donats dos enters, $a$ i $b$ es defineix la successió matemàtica següent:

$x_1 = a$

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

Pels valors $a = 4$ i $b = 4$ els primers elements d'aquesta successió són:

$4, 2, 1, 5, 21, 85, 341, 1365, 5461, 21845, \ldots$

Aquesta successió, primer decreix però a partir del terme de valor 1 és monòtona creixent.

En canvi, pels valors $a = 1$ i $b = 3$ és una successió cíclica: $1, 4, 2, 1, 4, 2, 1 \ldots$

Pel primer cas es pot obtenir una condició d'acabament definint un valor màxim a partir del qual ja no es generin més termes.

Pel segon cas cal un comptador d'iteracions. Veieu l'exercici 5.

## 11.9 Exercicis

1. Dissenya una funció `sumamult` que donat un enter `n`, retorni la suma dels múltiples de `3` positius i inferiors a `n`. Exemple:

```
>>> sumamult(10)
18
>>> sumamult(30)
135
>>> sumamult(100)
1683
```

```
def sumamult (n):
    suma = 0
    mult = 3
    while mult < n:
        suma = suma + mult
        mult = mult + 3
    return suma
```

   Aquest disseny es basa en la seqüència dels múltiples de 3 positius i inferiors a $n$ caracteritzada a l'apartat 11.3. L'esquema és un recorregut: síntesi (sumatori).

Aquesta funció també es pot dissenyar amb la sentència `for` i la funció `range`:

```
def sumamult (n):
    suma = 0
    for mult in range(3, n, 3):
        suma = suma + mult
    return suma
```

En aquest cas, per triar correctament els paràmetres de la funció `range` també s'ha aplicat la metodologia explicada en aquest capítol.

Aquests dos dissenys no només resolen el mateix problema sinó que, en realitat, són equivalents ja que les sentències `mult = 3`, `mult = mult + 3` i l'expressió booleana `mult < n` obtenen la mateixa seqüència que la sentència `for mult in range(3, n, 3)`.

2. Dissenya una funció `potdos` que donat un enter $x$ que representa una xifra i un altre enter $n$, retorni el nombre de potències de *2* inferiors a `n` que continguin aquesta xifra. Exemple

```
>>> potdos(2, 1000)
5
>>> potdos(3, 1000)
1
>>> potdos(7, 10000)
0
>>> potdos(7, 100000)
1
```

```
def potdos (x, n):
        compt = 0
        pot2 = 1
        while pot2 < n:
                if str(x) in str(pot2):
                        compt = compt + 1
                pot2 = pot2 * 2
        return compt
```

Aquest disseny es basa en la seqüència de les potències de 2 inferiors a $n$ caracteritzada a l'apartat 11.3. L'esquema és un recorregut: síntesi (comptador).

Aquesta funció no es pot dissenyar amb la sentència `for` i la funció `range`, ja que la relació de successió no correspon a una progressió aritmètica (`+`) sinó a una progressió geomètrica (`*`).

3. Es disposa de la funció matemàtica $y = f(x) = x^2+4x+2$ mostrejada dins l'interval $[a, b]$ i amb increments d'abscissa de $d$. Dissenya la funció `posneg` que, donats els valors `a, b, d` (enters) indicats, retorni dos enters. El primer és el nombre de punts que tenen ordenada positiva o zero i el segon el nombre de punts que tenen ordenada negativa del conjunt de punts del mostreig de la funció matemàtica $f(x)$ esmentada. Exemple:

```
>>> posneg(-5, 5, 1)
(8, 3)
>>> posneg(-5, 5, 2)
(4, 2)
>>> posneg(-4, 4, 2)
(4, 1)
```

La seqüència associada a aquest exercici es la seqüència de les abscisses dels punts mostrejats. L'esquema és un recorregut: síntesi (comptador).

```
def posneg (a, b, d):
    npos = 0
    nneg = 0
    # primer element de la seqüència
    x = a
    while x <= b:
        y = x**2 + 4*x + 2
        if y >= 0:
            npos = npos + 1
        else:
            nneg = nneg + 1
        # següent element de la seqüència
        x = x + d
    return npos, nneg
```

4. Un interval es representa amb un tuple de dos elements que representen els seus extrems inferior i superior. Donada una llista d'intervals, dissenya una funció que retorni un real corresponent a la suma de les amplituds de tots els intervals. Dissenya tres variants d'aquesta funció: la funció `total1` que usi la sentència `for` recorrent els elements de la llista directament; la funció `total2` que usi la sentència `for` recorrent els elements de la llista a través de l'índex; i la funció `total3` que usi la sentència `while`. Exemple:

```
>>> lintervals = [(-3, 3), (1.2, 4.8), (-56, 24),
...  (-20, -10)]
>>> total1 (lintervals)
99.6
>>> total2 (lintervals)
99.6
>>> total3 (lintervals)
99.6
```

```
def total1 (lintervals):
    tot = 0
    for int in lintervals:
        tot = tot + (int[1]-int[0])
    return tot

def total2 (lintervals):
    tot = 0
    for i in range(len(lintervals)):
        tot = tot + (lintervals[i][1]-lintervals[i][0])
    return tot

def total3 (lintervals):
    tot = 0
    i = 1
    while i < len(lintervals):
        tot = tot + (lintervals[i][1]-lintervals[i][0])
        i = i +1
    return tot
```

5. Donats dos enters, $a$ i $b$ es defineix la successió matemàtica següent:

   $x_1 = a$

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

   Dissenya la funció `succ` que, donats quatre enters positius `a, b, fi` i `ni`, calculi la mitjana dels termes de la successió anterior inferiors a `fi`. El paràmetre `ni` de la funció `succ` és el nombre màxim d'iteracions permeses. La funció ha de retornar la mitjana i el nombre de termes usats. Exemple:

```
>>> serie(4, 4, 50, 10000)
(6.6, 5)
>>> serie(2, 2, 50, 10000)
(9.833333333333334, 6)
>>> serie(3, 1, 50, 10000)
(1.5004, 10000)
>>> serie(1, 3, 50, 10000)
(2.3332, 10000)
>>> serie(64, 5, 100, 10000)
(5.724, 10000)
```

```
def succ (a, b, fi, ni):
    suma = 0
    n = 0
    x = a
    while  x < fi and n < ni:
        suma = suma + x
        n = n + 1
        if x%2 == 0:
            x = x//2
        else:
            x = b*x + 1
    return suma/n, n
```

   En aquest exercici, la variable `n` que compta el nombre de termes per obtenir la mitjana es pot usar també com a comptador d'iteracions.

6. Dissenya la funció `xifres_iguals`, que donats dos enters, retorni el primer enter que hi hagi entre ells que tingui totes les xifres iguals. Si no n'hi ha cap, la funció ha de retornar l'enter `-1`. Exemple:

```
>>> xifres_iguals (1, 100)
1
>>> xifres_iguals (45, 100)
55
>>> xifres_iguals (699, 1000)
777
>>> xifres_iguals (699, 750)
-1
>>> xifres_iguals (100, 111)
111
```

```
def xifres_iguals_1 (a, b):
    x = a
    while x <= b:
        sx = str(x)
        if sx.count(sx[0]) == len(sx):
            return x
        x = x +1
    return -1

def xifres_iguals_2 (a, b):
    for x in range(a, b+1):
        sx = str(x)
        if sx.count(sx[0]) == len(sx):
            return x
    return -1

def xifres_iguals_3 (a, b):
    trobat = False
    x = a
    while x <= b:
        sx = str(x)
        trobat = sx.count(sx[0]) == len(sx)
        if trobat:
            break
        x = x + 1
    if trobat:
        return x
    else:
        return -1

#tria la funció
xifres_iguals = xifres_iguals_1
#xifres_iguals = xifres_iguals_2
#xifres_iguals = xifres_iguals_3
```

La seqüència associada a aquest exercici es la seqüència dels enters des de `a` fins a `b` (ambdós inclosos). L'esquema és una cerca.

La condició de cerca, que un enter tingui totes les xifres iguals, s'ha expressat convertint l'enter a string i aplicant el mètode `count`:

```
sx.count(sx[0]) == len(sx)
```

De les tres resolucions que es mostren, a la primera s'usa la sentència `for` ja que la seqüència a generar és una progressió aritmètica. La segona resolució és equivalent a la primera però usant la sentència `while` en lloc de `for`. I a la tercera s'usa l'esquema general de cerca amb la variable booleana `trobat`.

La condició de cerca es podria dissenyar ad hoc. Dissenya la funció `té_xifres_iguals` que, donat un enter, retorni `True` si té totes les xifres iguals i `False` en cas contrari. Després redissenya la funció `xifres_iguals` usant la funció `té_xifres_iguals`.

7. Dos nombres primers es diu que són *bessons* si difereixen en dues unitats. Per exemple, els dos nombres primers 29 i 31 són bessons, però els dos nombres primers 19 i 23 no ho són. A continuació es mostren els primers 19 parells de bessons:

```
(3,  5), (5,  7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61),
(71,  73), (101, 103), (107, 109), (137, 139), (149, 151), (179, 181),
(191, 193), (197, 199), (227, 229), (239, 241), (269, 271), (281, 283)
```

Ens demanen que dissenyem la funció `bessons` que, donats dos enters `a` i `b`, retorni el primer parell de nombres primers bessons que hi hagi entre ells. Si no n'hi ha cap, la funció ha de retornar l'enter `-1`.

Pel disseny d'aquesta funció hem de caracteritzar la seqüència dels nombres primers entre `a` i `b`. El primer element d'aquesta seqüència, `x`, és el primer nombre primer més gran o igual que `a`. El següent element d'aquesta seqüència és el primer nombre primer més gran que `x`. Com que estem buscant un parell de nombres primers consecutius (dins la seqüència dels nombres primers), aplicarem la tècnica de finestres de 2 elements. La condició de fi de seqüència és que el segon element d'una d'aquestes finestres sigui més petit o igual que `b`.

Exemple: suposant `a=600` i `b=643`, els nombres primers entre `a` i `b` són: 601, 607, 613, 617, 619, 631, 641 i 643. Els parells a considerar seran: (601, 607), (607, 613), (613, 617), (617, 619), (619, 631), (631, 641) i (641 643). En aquest cas hi ha dos parells que són bessons: (617, 619) i (641 643). La funció `bessons` ha de retornar el primer: (617, 619).

Ara bé, l'obtenció de la seqüència dels nombres primers no és directa i la manera més entenedora de generar-la (sinó la única) és generant una seqüència d'enters i comprovar per cadascun si és primer o no.

En el disseny de la funció `bessons` utilitzarem dues funcions més: la funció `és_primer` i la funció `següent_primer`, que s'especifiquen tot seguit i es dissenyaran a continuació, previ al disseny de la funció `bessons`.

(a) La funció `és_primer`, donat un enter positiu, retorna `True` si aquest és un nombre primer i `False` en cas contrari.

(b) La funció `següent_primer` que donat un enter positiu, retorna el primer nombre primer més gran o igual que l'enter donat.

(a) Dissenya la funció `és_primer(n)` que, donat un enter positiu, `n`, retorna `True` si `n` és un nombre primer i `False` en cas contrari.

Un nombre primer es defineix com un nombre més gran que `1` que només és divisible per `1` i per ell mateix. Seguint la definició, s'ha de generar la seqüència dels enters entre `2` i `n-1` i comprovar si `n` és divisible per algun d'ells. L'esquema a aplicar és una cerca ja que quan es troba un divisor de `n` ja es pot dir que `n` no és primer.

Aquesta seqüència es pot simplificar. Per un costat, exceptuant els casos especials dels nombres `1` i `2`, tots els nombres primers són senars i, per tant, cap d'ells és divisible per `2`. Per tant, només cal generar la seqüència dels nombres senars. Per altra banda, si existeix un nombre $x > \sqrt{n}$ tal que $n$ és divisible per $x$, es compleix que també existeix un nombre $y < \sqrt{n}$ tal que $n$ és divisible per $y$ i $n = xy$. Quan $n$ és un quadrat perfecte, $x = y = \sqrt{n}$. Per tant, no cal arribar fins a `n-1`, sinó només fins a $\sqrt{n}$.

A continuació es mostren alguns tests i el disseny de la funció `és_primer(n)`. Aquest disseny tracta a part els casos dels nombres `1` i `2` i dels nombres parells. Per la resta aplica el mètode indicat amb la simplificació esmentada.

```
>>> és_primer(1)
False
>>> és_primer(2)
True
>>> és_primer(53)
True
>>> és_primer(84)
False
>>> és_primer(63)
False
```

```
import math

def és_primer(n):
    if n == 1:
        return False
    elif n == 2:
        return True
    elif n%2 == 0:
        return False
    else:
        sqr = math.sqrt(n)
        trobat = False
        x = 3
        while x <= sqr:
            trobat = n%x == 0
            if trobat:
                break
            x = x + 2
        return not trobat
```

(b) Dissenya la funció `següent_primer(n)` que, donat un enter positiu, `n`, retorna el primer nombre primer més gran o igual que `n`. Aquesta funció ha de cridar la funció anterior `és_primer`.

Exceptuant l'`1` i el `2`, els nombres primers són senars. La funció `següent_primer(n)` genera la seqüència dels nombres senars, `x`, començant per `n` si aquest és senar o per `n+1` si és parell. La condició de fi de seqüència és que `x` sigui primer. Podem posar aquesta condició de fi ja que la seqüència dels nombres primers és infinita i, per tant, sempre existeix un nombre primer més gran que qualsevol enter positiu donat. Aquesta funció també tracta de forma especial els casos `1` i `2`.

A continuació es mostren alguns tests i el disseny de la funció `següent_primer(n)`:

```
>>> següent_primer(1)
2
>>> següent_primer(2)
2
>>> següent_primer(9)
11
>>> següent_primer(23)
23
>>> següent_primer(511)
521
>>> següent_primer(521)
521
```

```
def següent_primer(n):
    if n == 1 or n == 2:
        return 2
    else:
        if n%2 == 0:
            n = n + 1
        x = n
        while not és_primer(x):
            x = x + 2
        return x
```

(c) Dissenya la funció `bessons(a,b)` que, donats dos enters `a` i `b`, retorni el primer parell de nombres primers bessons que hi hagi dins l'interval $[a, b]$. Si no n'hi ha cap, la funció ha de retornar l'enter `-1`. Aquesta funció ha de cridar la funció anterior `següent_primer`.

Aquesta funció, partint de la seqüència dels nombres primers entre `a` i `b`, ha de generar parells de nombres primers consecutius (finestres de 2 elements).

Denotem per (`x`, `y`) un parell de nombres primers consecutius. El valor de `x` del primer parell s'obté fent `següent_primer(a)`. El valor de `y` del primer parell s'obté fent `següent_primer(x+1)`. Després, a cada iteració, el primer element del parell passa a ser el segon element del parell anterior, `x = y` i el valor de `y` del següent parell s'obté fent `següent_primer(x+2)` ja que en aquest cas tenim la garantia que `x` és senar. La condició de fi és que `y > b`. L'esquema a aplicar és de cerca i la condició de cerca és que `x` i `y` difereixin en dues unitats: `y == x + 2`.

A continuació es mostren alguns tests i el disseny de la funció `bessons(a, b)`:

```
>>> bessons.bessons(1,6)
(3, 5)
>>> bessons.bessons(29,60)
(29, 31)
>>> bessons.bessons(75,100)
-1
>>> bessons.bessons(858,1000)
(881, 883)
>>> bessons.bessons(900,1000)
-1
>>> bessons.bessons(620,643)
(641, 643)
>>> bessons.bessons(620,642)
-1
```

```
def bessons (a, b):
    trobat = False
    x = següent_primer(a)
    y = següent_primer(x + 1)
    while y <= b:
        trobat = y == x + 2
        if trobat:
            break
        x = y
        y = següent_primer(x + 2)
    if trobat:
        return (x, y)
    else:
        return -1
```

# 12 Càlcul amb valors reals

En aquest capítol es fa una molt breu introducció a la problemàtica referent a la representació dels valors reals en els ordinadors i del càlcul amb aquests valors.

## 12.1 Representació dels valors reals en els ordinadors

El tipus `int` que permet representar valors enters presenta una limitació en magnitud, però no en precisió (aritmètica amb enters). Per exemple, un enter representat en 8 bytes té el següent rang de valors $[-2^{63}, -2^{63}]$ (8 bytes són 64 bits, un dels quals s'utilitza pel signe de l'enter).

En canvi el tipus `float` que permet representar valors reals presenta limitacions en la magnitud i també en la precisió: la representació dels valors reals en els ordinadors no és exacta. Per un costat hi ha valors reals amb un nombre infinit de decimals (com el valor $1/3 = 0.\overline{3}$). També hi ha valors reals que en base 10 tenen un nombre finit de decimals, però en base 2 tenen un nombre infinit de decimals. El valor 0.1 en base 10 té la següent representació en base 2: $0.0\overline{0011}$. A més, molts valors reals amb un nombre finit de decimals tampoc es poden representar de forma exacta.

En la memòria d'un ordinador, un valor real, $n$, es representa amb notació científica en base 2, també referida com a notació amb **coma flotant** (**floating point** en anglès, d'on ve el nom del tipus `float`):

$$n = m \times 2^e$$

on $m$ és la mantissa i $e$ l'exponent. La mantissa és un valor real en base 2 tal que $abs(M) < 1$. L'exponent és un enter en base 2. Tant la mantissa com l'exponent poden ser positius o negatius.

Tant la mantissa com l'exponent venen limitats per la capacitat (en bits) d'un valor real. Aquesta capacitat depèn del sistema que s'estigui usant. El nombre de bits per representar un valor s'ha de repartir entre la representació de l'exponent (un valor enter) i la mantissa (els seus decimals) amb els signes corresponents. La limitació de bits de l'exponent implica una limitació de la magnitud dels valors mentre que la limitació de bits per la mantissa implica una limitació de la precisió. El tipus `float` de Python es representa en 8 bytes (64 bits) que es reparteixen de la forma següent: 1 bit pel signe del valor, 11 per l'exponent (1 pel signe de l'exponent i 10 pel valor) i 52 per la mantissa. Els llenguatges de programació ofereixen tipus de dades amb més bits per la mantissa per tal de treballar amb més precisió. Amb la biblioteca `numpy`, podem usar el tipus `numpy.float128` que permet representar reals amb 16 bytes (128 bits): 1 bit pel signe, 15 per l'exponent i 112 per la mantissa.

La inexactitud de la representació dels reals repercuteix en totes les operacions que es fan entre reals i, en particular, en la comparació entre reals.

## 12.2 Operacions amb valors reals

Quan s'opera amb valors reals es produeixen errors. Hi ha els errors d'arrodoniment deguts a la pròpia representació. Per exemple suposem que tenim una representació en base 10 i 5 xifres i sumem les quantitats 9.2654 i 7.1625, el resultat és 16.4279 que té 6 xifres. Com que s'ha d'arrodonir a 5 xifres queda aproximat pel valor 16.428. També hi ha els errors de propagació deguts a les operacions amb valors aproximats.

En aquest document ens centrarem en el problema més comú que és el de la comparació entre dos valors que teòricament haurien de ser iguals però que degut a que s'han obtingut a partir de processos diferents no són exactament iguals.

El mètode per tractar aquest problema es basa en la definició d'un valor `epsilon` suficientment petit de forma que pugui representar el valor 0 (zero) amb la precisió que es desitgi.

Donats dos valors `x, y` que teòricament haurien de ser iguals però que no ho són, l'expressió `x == y` donarà `False`. En canvi, l'expressió `abs(x-y) < epsilon` donarà `True`

## 12.3 Exercicis

1. Una estació meteorològica ha captat 3 temperatures pels 10 primers dies del més de gener d'un any determinat. Aquesta informació és en una llista de 10 subllistes corresponents a aquests dies de gener en ordre cronològic. Cada subllista té 3 valors reals corresponents a les 3 temperatures captades.

   Dissenya la funció `temp (ltemp, tref, epsilon)` que, a partir d'un llista com la indicada, `ltemp`, una temperatura de referència, `tref`, i un valor `epsilon`, retorni una altra llista amb els dies de gener en què la mitjana de les 3 temperatures captades és igual a `tref`. Recorda que per fer les comparacions entre nombres reals has de fer-ho amb la tolerància `epsilon` donada, és a dir, comparant el valor absolut de la diferència dels dos nombres amb `epsilon`.

   ```
   >>> ltemp = [[-1.2, 0.1, 1.1], [-2.2, 0.5, 1.7],
   ... [-10.5, 2.3, 6.2], [-3.7, 1.1, 2.6], [-7.6, 1.2, 3.4],
   ... [-9.8, -2.3, 12.1], [1.1, 1.1, -2.2], [0.1, -0.1, 0],
   ... [-4.5, -2.1, -0.8], [4.5, -3.2, -1.3]]
   >>> temp(ltemp, 0.0, 0.001)
   [1, 2, 4, 6, 7, 8, 10]
   ```

   El disseny de la funció és:

   ```
   def temp (ltemp, tref, epsilon):
       ldies = []
       for i in range(len(ltemp)):
           sl = ltemp[i]
           mit = (sl[0]+sl[1]+sl[2])/3
           if abs(mit-tref) < epsilon:
               ldies.append(i+1)
       return ldies
   ```

   Els valors de les 10 mitjanes són els següents:

```
7.401486830834377e-17
-7.401486830834377e-17
-0.6666666666666664
0.0
-0.9999999999999999
-5.921189464667501e-16
0.0
0.0
-2.4666666666666663
-7.401486830834377e-17
```

Observem que els únics que donen zero exactament són els dels dies 4, 7 i 8. Els dels dies 1, 2, 6 i 10 són zero aproximadament i, per tant, si es comparessin amb `tref` amb l'operador d'igualtat, (`mit == tref`), el resultat seria `False`.

2. El desenvolupament com a sèrie de Taylor de la funció exponencial és:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Dissenya la funció `exponencial` que, donats dos reals `x` i `epsilon`, retorni el valor de $e^x$ calculat com la suma dels termes del desenvolupament en sèrie superiors a `epsilon` en valor absolut.

Hem de generar la seqüència dels termes del desenvolupament. Un terme es pot expressar com $t_i = x^i/i!$. Per tant, generant la seqüència de valors enters es poden obtenir els corresponents termes. Una altra forma de caracteritzar la seqüència és la següent: $t_o = 1, t_i = t_{i-1} * x/i$. La condició de fi de seqüència es compleix quan el valor absolut d'un terme és igual o inferior a `epsilon`.

A continuació es mostren alguns tests d'aquesta funció:

```
>>> exponencial(1, 0.001)
2.7180555555555554
>>> exponencial(-5, 0.00000001)
0.006737943884298238
>>> math.e**3
20.085536923187664
>>> exponencial(3, 0.001)
20.08521256087662
>>> exponencial(3, 0.000001)
20.085536488379084
>>> exponencial(3, 0.00000001)
20.08553691196314
```

En els tres darrers tests s'observa com el valor $e^3$ es va aproximant més al que dóna Python, com més petit és el valor d'`epsilon`.

A continuació es mostra el disseny de la funció `exponencial` seguint la primera forma de caracteritzar la seqüència. El comentari indica l'acció a fer per obtenir el següent element si s'aplica l'altra forma de caracteritzar la seqüència.

```
import math
def exponencial (x, epsilon):
    suma = 0
    i = 0
    t = 1
    while abs(t) > epsilon:
        suma = suma + t
        i = i + 1
        t = x**i/math.factorial(i)      # t = t * x/i
    return suma
```