UNIVERSITAT DE BARCELONA

ALGORÍSMICA

Apunts de l'assignatura

Jordi Vitrià, Mireia Ribera

jordi.vitria@ub.edu | ribera@ub.edu

https://algorismica2018.github.io/

Índex

Presentació de l'assignatura

Què és aquesta assignatura?

Aquesta assignatura està adreçada a donar la formació bàsica als estudiants sobre l'anàlisi i disseny d'algorismes, tant des d'un punt de vista teòric com aplicat. No s'assumeix cap formació prèvia en programació de l'estudiant.

Què s'espera dels estudiants matriculats?

Els estudiants han de participar de forma activa durant les classes magistrals de teoria (**1,5 hores a la setmana**). Durant les hores teòrico-pràctiques (o de problemes, **10 sessions de 1,5 hores**) hauran de dissenyar solucions algorísmiques als problemes plantejats pels professors.

Durant les hores presencials de pràctiques (**10 sessions de 1,5 hores**) hauran de programar de forma individual una sèrie d'exercicis pràctics. Les hores no presencials de l'assignatura (4 hores a la setmana) les han de dedicar a l'estudi i a la preparació dels problemes i pràctiques.

Programarem?

Tot i que en aquesta assignatura no és estrictament necessari programar, ho farem amb un llenguatge d'alt nivell: Python.

Com s'organitza l'assignatura?

Usarem una única eina per distribuir la informació i organitzar la feina: el **Campus Virtual** de la UB.

Què hi ha al Campus Virtual?

- Un link als apunts de teoria.
- Conjunt de problemes.
- Enunciat de les pràctiques.
- Lliurament de les pràctiques.
- Avaluacions de les proves i pràctiques.

Com s'avaluarà l'assignatura? (I)

L'assignatura seguirà un esquema d'avaluació continuada, amb dos elements principals: proves presencials i lliurament remot d'exercicis.

- Lliurament via campus virtual de pràctiques (LP): Els professors proposaran una sèrie de pràctiques que hauran de ser lliurades via el campus virtual per part de l'alumne dins el període assenyalat pel professor. Cada un dels lliuraments serà avaluat pel professor amb una nota que pot anar de 0 (nota mínima) a 10 (nota màxima). En cas de no lliurar les pràctiques dins el període assenyalat, l'alumne obtindrà un 0. La nota final (LP) de la part de lliurament de pràctiques serà la mitja de tots els lliuraments.
- Proves presencials (PP): durant el curs, l'alumne realitzarà varies proves escrites (teòrico-pràctiques) davant del professor. Les proves s'avaluaran amb una nota de 0 (nota mínima) a 10 (nota màxima). La nota final (PP) d'aquesta part serà la mitja de totes les proves realitzades (una prova no realitzada = 0).

La nota segons l'avaluació continuada (NF) es calcularà de la següent manera:

- Si (LP>4,0 i PP>4,0): NF = PP * 0,6 + LP * 0,4
- Sinó, NF = min(4.0, PP * 0,6 + LP * 0,4).

Com s'avaluarà l'assignatura? (II)

Durant la segona prova presencial (Gener) es donarà l'opció de presentar-se de tota l'assignatura o només de la segona part. Si un alumne opta per tornar a presentar-se a la primera part de l'assignatura, en la nota de proves presencials es tindrà en compte la nota millor dels dos intents.

Tots aquells alumnes que no aprovin la prova presencial de Gener i obtinguin una NF>=3,5 tenen dret a una reavaluació al cap d'un dies de la publicació de NF. La reavaluació serà equivalent a un examen final. En aquests casos, la nota final de l'assignatura serà la nota de la reavaluació.

I el lliurament de problemes...?

No hi ha una activitat pròpia de lliurament de problemes, però l'alumne anirà construint un portafolis de problemes que el professor pot revisar en qualsevol moment.

La part pràctica de les proves presencials estarà basada en aquests problemes.

El campus virtual proporciona informació sobre QUAN s'ha resolt els problemes. Per valorar aquesta activitat el professor tindrà en compte tant el QUÈ com el QUAN.

Calendari de proves

Tots els exàmens es fan en període no lectiu.

- La primera prova presencial es farà el 5 de novembre de 15h a 18h.
- La segona prova presencial (examen final) es farà el 18 de gener de 15h a 20h.
- La reavalaució es farà el 31 de gener de 15h a 20h.

Bibliografia

Algorísmica

- T. H. Cormen et al. Introduction to algorithms, MIT Press, 2001.
- S.Dasgupta.Algorithms, McGrawHill, 2006.
- V. Levitin, Introduction to the Design and Analysis of Algorithms, ISBN: 0-201-74395-7, Addison-Wesley (2ond edition)
- S. Skiena. The Algorithm Design Manual, Springer; 2nd edition (August 21, 2008), Language: English, ISBN-10: 1848000693.

Python

- A. Downey, J. Elkner and C. Meyers. How to Think Like a Computer Scientist. Learning with Python..

Tema 1: Què és un algorisme?

Resum del tema 1

- **Conceptes**:
  - Algorisme,
  - Input(entrada),
  - Output(sortida),
  - Correcció,
  - Eficiència (memòria i cicles),
  - Errors (tipus),
  - Conceptes de llenguatge de programació:
    - Primitives (símbols),
    - sintaxi,
    - semàntica estàtica,
    - semàntica.

Què és un algorisme?

> Definició de la *Wikipedia*: Un algorisme és una seqüència finita, no ambigua i explícita, d'instruccions per a resoldre un problema.

La definició d'aquesta assignatura:

Un algorisme és qualsevol procediment computacional que pren un (o una sèrie) de dades/valors com a *entrada* i genera alguna dada/valor (o sèrie de dades/valors) com a *sortida*.

- Els algorismes són les **idees/estratègies** que hi ha darrera els programes per resoldre un determinat problema.
- Els algorismes són independents del llenguatge en que estan escrits. El mateix algorisme escrit en dos llenguatges diferents pot tenir una aparença superficial molt diferent.
- Els algorismes sí que depenen de la representació de les dades.
- Els algorismes interessants són els que resolen problemes generals. Els problemes específics es resolen reduïnt-los a problemes generals.

Exemple computacional (arrel quadrada)

Problema a resoldre mitjançant un algorisme:

> Entrada: Un nombre `a`
>
> Sortida: Un nombre `b` tal que `b*b=a`
>
> Requeriment: Volem una solució **correcta i eficient**!

Hi ha diversos algorismes per calcular aquest valor. El que s'explica a l'escola és un d'ells (i no és exactament simple!).

Heró d'Alexandria (10 dC-70 dC) ja en va proposar un altre:

- Comencem amb un nombre qualsevol `g`.
- Mentre `g*g` *no s'assembli prou* a `a`:
  - Calculem un nou candidat `(g+a/g)/2`.
- Donem com a resultat l'últim valor de `g`.

Exemple computacional (arrel quadrada)

Codificació en Python:

La condició "mentre `g*g` *no s'assembli prou* a `a`" la implementarem calculant un error d'aproximació: `g*g - a > valor_error`.

```
def hero(a,error):
    import math
    g = 1.0
    while math.fabs(g*g - a) > error:        #fabs vol dir nombre absolut en decima
        g = 1/2*(g+a/g)
    return g
```

Si executem `hero(49,0.0001)`, l'ordinador retorna `7.000000141269659`.

Correcció i Eficiència Algorísmica

Un algorisme és **correcte** si podem demostrar que retorna la sortida desitjada per a qualsevol entrada legal (per al problema de l'arrel quadrada, això vol dir nombres positius o 0).

> Demostrar la *correcció* és fàcil per alguns algorismes, difícil per la majoria i fins i tot impossible per alguns! Aquest aspecte de l'algorísmica no l'abordarem durant aquest curs.

Un algorisme és **eficient** si es fa amb el mínim nombre de recursos (cicles de càlcul de l'ordinador / temps, memòria de l'ordinador) possible.

> Fer servir algorismes eficients és sempre convenient i moltes vegades una necessitat!

Algorismes i ordinadors

Un ordinador fa només dues coses (però molt ben fetes!):

- calcular (combinar dades per obtenir altres dades);
- emmagatzemar (llegir/escriure dades a una memòria) els resultats del càlcul.

Un ordinador convencional fa més de 1.000.000.000 de càlculs per segon i pot emmagatzemar més de 1.000.000.000.000 de bits.

Els algorismes que veurem en aquest curs són procediments per a resoldre problemes que estan basats en el càlcul i emmagatzament de dades en un ordinador convencional.

No veurem algorismes:

- basats en càlcul paral·lel ni distribuït entre diversos ordinadors;
- basats en arquitectures no convencionals (p.e. quàntica).

Exemple: el problema del viatjant de comerç (TSP).

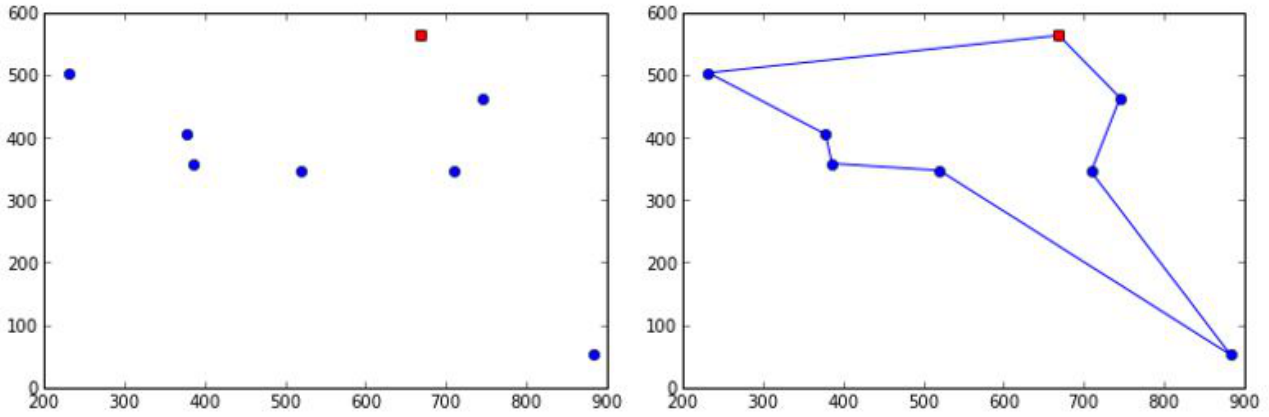
Aquest cartell correspon al concurs promogut per *Procter & Gamble* l'any 1962 per recorrer 33 ciutats dels EUA:

Anem a proposar algorismes per solucionar-ho!

Exemple: el problema del viatjant de comerç (TSP).

Suposem que hem de passar per un conjunt de punts definits i volem minimitzar la distància recorreguda.

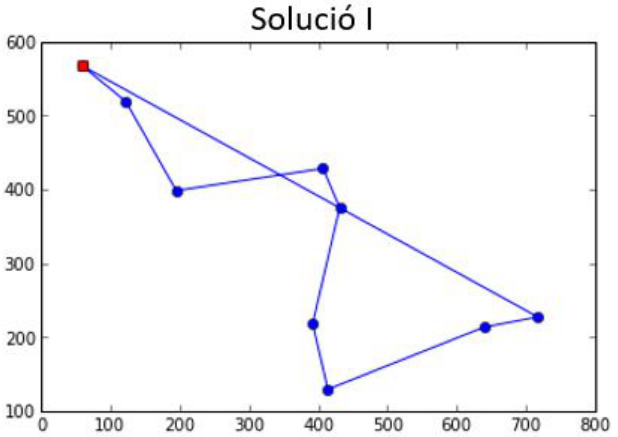
A la figura de la dreta tenim una possible instància del problema. A l'esquerra hi ha la millor solució d'aquesta instància, **la que voldriem trobar amb un algorisme que acceptés com a entrada qualsevol conjunt de punts***.

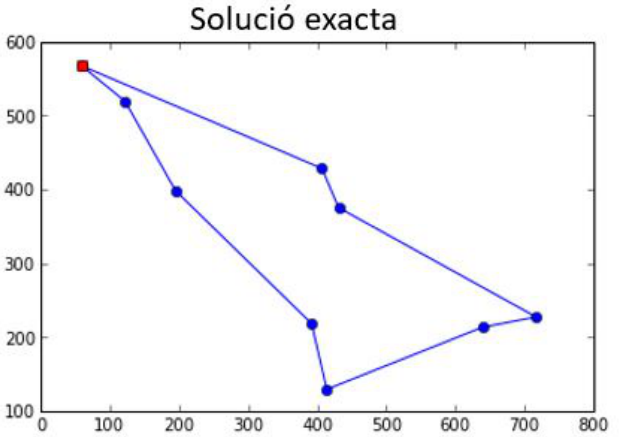


* A la major part dels casos que resoldrem en aquest curs o tindrem accés a la solució correcte del problema o podrem fer un programa molt simple que, donada una solució, comprovi que és correcte. En el cas del problema del viatjant de comerç no tindrem ni una cosa ni l'altra!

Propostes

Solució I: Escollim un punt aleatori per començar, i anem creant un recorregut seleccionant el *veí més proper* (la ciutat més propera entre les que no ha visitat encara) a cada pas.





És correcte?

> Sabem que **no ho és** perquè tenim accés a un *oracle* que ens diu quina és la solució correcte. Més endavant veurem quines opcions tenim si no tenim accés a l'oracle.

Propostes

Solució II:

- Considerem **tots el possibles passos parcials entre dues ciutats**, calculem la seva longitud i ho guardem en un conjunt.
- Mentre ens quedin passos parcials al conjunt:
  - Busquem al conjunt el pas parcial més petit p.
  - Afegim p al recorregut final sempre i quan no generi un cicle o una doble sortida per un punt.
  - Eliminem p del conjunt de passos parcials.

És correcte?

Solució III:

Considerem **totes les possibles ordenacions** del conjunt format per totes les ciutats, calculem la distància de cada una de les ordenacions i seleccionem la més curta.

És correcte?

Solucions correctes i eficients!

La solució III anterior **és correcta** però **no és eficient**.

- No cal demostrar que és correcta: és evident!
- El nombre de possibles ordenacions d'un conjunt de `n` elements ve donat pel concepte de factorial: `n! = n x (n-1) x (n-2) x ... x 1`.
- El factorial d'un nombre `n` creix molt ràpidament quan `n` es fa gran. Aquest pot ser un nombre molt gran fins i tot pels ordinadors.

Com expressem els algorismes?

Amb **llenguatges de programació**.

Un llenguatge de programació es defineix per unes **primitives** (símbols), una **sintaxi** (regles de combinació de símbols), una **semàntica estàtica** (combinacions de símbols amb significat) i una **semàntica** (el significat que nosaltres volem donar a l'algorisme).

Com expressem els algorismes?

Amb **llenguatges de programació**.

Un llenguatge de programació es defineix per unes **primitives** (símbols), una **sintaxi** (regles de combinació de símbols), una **semàntica estàtica** (combinacions de símbols amb significat) i una **semàntica** (el significat que nosaltres volem donar a l'algorisme).

Fins ara hem usat *paraules* o pseudocodi, però també podem usar un llenguatge d'alt nivell, **Python**, molt proper al pseudocodi.

El preu que hem de pagar és que haurem d'**especificar** una mica més les coses.

Els avantatges:

- aprenem un llenguatge útil;
- som més formals en les especificacions;
- podem executar-los i fer simulacions.

Llenguatges

- **Símbols**: Són la forma d'escriure variables (p.e. `a`), instruccions (p.e. `print()`), etc. Hi ha una sèrie de regles que els defineixen.
- **Sintaxi**: Són les regles que defineixen les combinacions vàlides de símbols: `3.2 + 4.5` és vàlida, però `3.2 a 2.3` no ho és.
- **Semàntica estàtica**: `3.2/'abc'` és sintàcticament correcte perquè l'expressió (`<literal><operador><literal>`) ho és, però no ho és des del punt de vista de la semàntica estàtica.

  Els errors més perillosos quan programem no són els sintàctics, atès que la majoria es poden detectar automàticament o són fàcils de veure!

  Alguns llenguatges detecten quasi tots els errors de semàntica estàtica, però Python només alguns!
- **Semàntica**: Es refereix a "què" fa el programa (p.e. aquest programa calcula l'arrel quadrada?) i per tant depèn totalment del programador.

Llenguatges

Si no hi ha errors sintàctics ni de semàntica estàtica el programa s'executarà i farà alguna cosa, però no necessàriament la que volem.

Els *entorns de programació* (IDE) ens poden ajudar a detectar els errors sintàctics i alguns errors de semàntica estàtica, però no els semàntics.

Aquest programa no donarà mai cap error:

```
def suma(a,b):
  '''
  Aquest programa no fa el que ha de fer!
  '''
  return a - b
```

Els únics que podem determinar que no és correcte sóm nosaltres.

Llenguatges

Si un programa té un error que no ha estat detectat:

1. Pot acabar inesperadament la seva execució i generar un error. La majoria de vegades no afecta a la resta de programes de l'ordinador, però hi ha errors que poden causar un error fatal a l'ordinador i aturar-lo*. En aquest cas, la majoria d'errors són errors de sintaxi o de semàntica estàtica no detectats per l'IDE.
2. Pot ser que mai s'aturi i per tant no generi la resposta. En aquest cas normalment estem davant d'un problema semàntic.
3. Pot aturar-se i generar una resposta que pot ser incorrecta. En aquest cas normalment estem davant d'un problema semàntic.

*Pot ser que els errors només es manifestin per alguna combinació específica dels valors d'entrada i per tant no es detectin sense fer moltes proves. Normalment no podem provar totes les possibilitats!

Possibles preguntes d'exàmen del Tema 1

- Quines són les dues característiques que defineixen un algorisme?
- Quins errors són més difícils de corregir i perquè?
- Donat un algorisme simple, determinar si té o no té algun error i de quin tipus és.
- Donat un algorisme simple, determinar si generarà una sortida (i quina) a partir d'una entrada determinada.
- Donats dos programes diferents, determinar si implementen o no el mateix algorisme.
- Donat un problema simple i dos possibles algorismes que el resolen, determinar quin dels dos és millor.

Tema 2: Python

Resum del tema 2 (I)

- **Conceptes de programació**:
  - Instrucció,
  - funció,
  - paràmetres,
  - crida/invocació,
  - noms,
  - expressions,
  - literals,
  - comentaris,
  - variables,
  - iteracions definides / bucles,
  - biblioteques (libraries) de funcions / mòduls,
  - assignacions:
    - asssignacions simples, d'entrada, simultànies.

Resum del tema 2 (II)

- **Llenguatge Python**:
  - com organitzar un programa modularment: funcions i blocs de codi
  - `def`,
  - `print()`,
  - `input` i `eval(input)`,
  - `for i in range(start,stop,step)`,
  - índex,
  - operació `**`,
  - `abs()`,
  - `%`,
  - `import`

Resum del tema 2 (III)

- **Llenguatge Python (2)**:
  - cadenes de caràcters o strings
    - slicing
    - concatenar(+) i repetir (*)
    - split, join, strip
    - islower, isupper, isalpha
  - ASCII i UniCode
    - ord i char
  - àmbit o scope
  - crida de funcions
  - operadors relacionals: <, <=, ==, >=, >, !=
  - operadors booleans: and, or
  - estructures de control:
    - if elif else
    - while

Resum del tema 2 (IV)

- **Llenguatge Python (3)**:
  - col·leccions de dades, mutables/immutables, homogènies/inhomogènies
    - llistes
    - diccionaris
    - tuples
  - referències
  - clonatge

Python

Funcions

Un programa en Python pot ser una simple seqüència d'instruccions vàlides, però si volem organitzar el codi de forma òptima podem **crear/definir funcions** (que en aquest cas s'anomena `hola`):

```
def hola():
    print("Hola!")
```

Les funcions serveixen per facilitar la **reutilització** del codi i també la seva **organització**.

Un cop la tenim definida la poden **cridar/invocar**:

```
hola()
> Hola!
```

El símbol `>` indica una resposta de l'ordinador a través de la pantalla.

Funcions

La definició d'una funció no té cap efecte visible, **no fa res si no la cridem**:

```
def hola():
    print("Hola!")
>
```

Però internament, Python l'emmagatzema.

La funció es podrà cridar dins de la sessió en que s'ha definit. Si canviem de sessió i la volem usar tenim dues opcions:

- Tornar-la a definir.
- Guardar-la en un fitxer al disc de l'ordinador i llegir-la des de Python.

Un programa en Python

Les funcions poden tenir **paràmetres** (que van entre els parèntesi):

```
def hola(persona):
    print("Hola", persona)
```

Un paràmetre és un contenidor, que en el moment de la definició no té valor.

Quan es crida la funció els paràmetres han de prendre un valor:

```
hola("Jordi")
> Hola Jordi
```

Consola Python: Exec

Un programa en Python

En aquest programa podem veure diversos elements del llenguatge:

- comentaris,
- variables,
- assignacions,
- iteracions,
- entrada de dades des del teclat,
- crida de la funció,
- etc.

```
def main():
    '''
    Comportament caòtic
    '''
    print("Aquest programa implementa un comportament caòtic")
    x = input("Entra un nombre entre 0 i 1: ")
    x = float(x)
    for i in range(10):
        x = 3.9 * x * (1-x)
        print(x)

main()
```

Un programa en Python

Els elements més importants que tenim per a construir un programa Python són:

- **Noms**. Els fem servir per anomenar les funcions i les variables.
  - Tècnicament s'anomenen identificadors. Han de començar per lletra o _ que pot ser seguit per qualsevol seqüència de lletres, dígits o subratllats (no espais!).
  - Distingirem entre majúscules i minúscules.
  - Hi ha noms reservats (`and`, `for`, `def`, etc.).
- **Expressions**: Són la part de codi que calcula o produeix nous valors de les dades.
  - L'expressió més simple s'anomena *literal*, i s'usa per especificar un valor: `3.6`, `"hola"`.
  - Les variables són literals.
  - Podem crear expressions complexes combinant expressions més simples amb operadors: `3.9 + x * (1-x)`
    - Els operadors matemàtics segueixen les precedències estàndard. Tot i que per tant no sigui necessari l'ús de parèntesi, és **molt** recomanable.

Un programa en Python

- **Sortides**. Són les dades o missatges que Python escriu com a producte de l'execució d'un programa. La forma més important de produïr una sortida és la funció `print`, amb els següents arguments:

  ```
  print(value1, value2,..., sep=' ', end='\n')
  ```

- **Assignacions**. N'hi ha dos tipus:

  - Assignacions simples: `x = 3 + x * (1-x)`, per assignar el valor d'una expressió a una variable.
  - Assignacions d'entrada: `x = input("Entra un valor: ")`, per assignar l'entrada de l'usuari (alfanumèrica) a través del teclat a una variable.

  Si volem introduir algun altre tipus de valor hem de fer servir la funció `eval` (que fins i tot ens permet evaluar expressions):

```
a = eval(input("Entrada: "))
a

> Entrada: 3+4+5
> 12
```

Un programa en Python

- En Python podem fer assignacions simultànies, com per exemple: `sum, diff = x+y, x-y`

  Aquest tipus d'assignació pot ser molt útil, com per exemple per *intercanviar els valors de dues variables*. Si volem intercanviar els valors de dues variables, això no funciona!:

```
x = 3
y = 4
x = y
y = x
print(x,y)

> 4 4
```

  Com ho faríeu?

```
x = 3
y = 4
x,y = y,x
print(x,y)
```

Un programa en Python

- **Iteracions** (*loops*) definides. Els *iteradors* ens permeten executar un conjunt d'instruccions un nombre definit de vegades. La instrucció `for` és l'iterador més simple:

```
for j in [0,1,2,3]:
  print(j*j, end=" ")

> 0 1 4 9

for i in range(10):
  print(i, end=" ")

> 0 1 2 3 4 5 6 7 8 9
```

L'expressió `[0,1,2,3]` representa una **llista** de nombres. La funció `range` ens permet generar llistes de forma automàtica.

Un programa en Python

Quan treballem amb nombres, la funció `range` pot ser molt útil. La seva sintàxi és `range(start, stop, step)`.

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

list(range(0,10,3))
> [0,3,6,9]

list(range(0,-4,-1))
> [0,-1,-2,-3]
```

Els nombres i Python

Les dades que un programa pot manipular i emmagatzemar són de diferents **tipus**. El tipus de la dada determina quins valors pot tenir i quines operacions es poden fer.

```
type(3)
> int
type(3.14)
> float
x = -32
type(x)
> int
print(3+4, 3+4.0)
> 7 7.0
print(10.0/3, 10/3)
> 3.3333333333335  3.3333333333335
```

Els operadors bàsics són: `+`, `-`, `*`, `/`, `**`, `%`, `abs()`. `**` indica la potència, `%` el mòdul, i `abs()` el valor absolut.

- Què dona `3**2`, `10%3`, i `abs(-4)`?

Els nombres i Python

Python també ens proporciona funcions matemàtiques dins d'una biblioteca (*module*) especial anomenada `math`.

Una biblioteca no és res més que un fitxer que conté un conjunt de definicions de funcions. Quan carreguem una biblioteca normalment no s'executa res, només es fan definicions de funcions i potser de variables.

```
import math
def main(a,b,c):
    x = (-b+math.sqrt(b**2-4*a*c))/2*a
    print(x)
```

Python té centenars de mòduls predefinits que l'usuari pot usar lliurement.

Exemple: el factorial d'un nombre.

```
def factorial(num):
    factorial = 1
    if num < 0:
      print("Entra un enter positiu! ")
    elif num == 0:
      print("El factorial de 0 és 1")
    else:
      for i in range(1,num+1):
          factorial *= i
      print("El factorial de ", num, "és", factorial)
```

Exemple explicat: el factorial d'un nombre.

```
def factorial(num): # definim la funció factorial amb un paràmetre
    factorial = 1
    if num < 0:    # condicional
      print("Entra un enter positiu! ")     # només s'executa si
                                            # es compleix la condició
    elif num == 0:    # condicional
      print("El factorial de 0 és 1") # només s'executa si
                                      # es compleix la condició
    else:    # alternativa als condicionals
      for i in range(1,num+1): # 1,2,3...num
          factorial *= i # factorial = factorial * i
      print("El factorial de ", num, "és", factorial)
```

Python Help

- Lloc Web de Python
- Documentació

Possibles preguntes d'exàmen del Tema 1

- Donat un programa simple en Python, detectar (sense executar-lo) possibles errors sintàctics i semàntics.
- Donat un programa simple en Python, determinar (sense executar-lo), si s'aturarà o té un bucle infinit.

Python (2)

Cadenes de caràcters (*strings*)

Un string és una seqüència de caràcters, que es pot emmagatzemar en variables:

```
a = 'Hola'
b = "Mireia"
print(a,b)
> Hola Mireia

type(b)
> str
```

Podem entrar *strings* des del teclat:

```
nom = input("Quin és el teu nom?")
```

De fet, tot el que entra pel teclat és una cadena de caràcters. Si volem entrar dades d'un altre tipus ho hem de fer així:

```
edat = eval(input("Quina és la teva edat?"))
```

`eval` interpreta el que entrem com una expressió Python i l'avalua.

Cadenes de caràcters (*strings*)

Per accedir a una cadena de caràcters hem de veure com Python les indexa:

H	**e**	**l**	**l**	**o**		**B**	**o**	**b**
0	1	2	3	4	5	6	7	8

Llavors podem accedir als valors de cada element de la cadena o fins i tot a subcadenes:

```
s = "Hello Bob"
x = 8
print(s[0], s[1], s[x-2])
> H e B
```

Una operació anomenada *slicing* ens permet accedir a subcadenes de caràcters:

```
print(s[0:3], s[6:9], s[:3], s[3:], s[:])
> Hel Bob Hel lo Bob Hello Bob
```

Cadenes de caràcters (*strings*)

També podem concatenar (+) i repetir (*) subcadenes:

```
print("Bread" + " & " * 3 + "Breakfast")
> Bread &  &  & Breakfast

len("Bread" + " & " * 3 + "Breakfast")
> 23
```

Exemple:

```
def mes():
    mesos = "GenFebMarAbrMaiJunJulAgoSetOctNovDes"
    n = eval(input("Quin mes vols?"))
    pos = (n-1) * 3
    m = mesos[pos:pos+3]
    print("L'abreviatura és: ", m)
```

Cadenes de caràcters (*strings*)

L'ordinador emmagatzema els caràcters de forma numèrica.

Una forma estàndard s'anomena codificació **ASCII** (*American Standard Code for Information Interchange*), però tal i com el nom indica, no considera els caràcters que no s'usen en l'anglès. Usa 7 bits per caràcter.

Per això hi ha el sistema **UniCode**, que considera els caràcters de totes els llengües. Usa 16 bits per caràcter. Per compatibilitat, és un superconjunt de l'ASCII.

Python ens dóna funcions per accedir a aquests codis:

```
ord('A')
> 65

ord('a')
> 97

chr(97)
> a
```

Taula ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Cadenes de caràcters (*strings*)

Amb la funció `split` puc separar una cadena en una llista en diferents parts, indicant el caràcter separador:

```
cadena = "458.342.120€"
llista = cadena.split('.')
> ['456', '342', '120€']
```

Si no indiquem el separador, per defecte és el caracter blanc.

```
cadena = "El gos i el gat, menjàven plegats"
llista = cadena.split()
> ['El', 'gos', 'i', 'el', 'gat,', 'menjàven', 'plegats']
```

Cadenes de caràcters (*strings*)

`join`: Un dels usos més estesos de `join` consisteix en convertir llistes a cadenes de caràcters.

```
llista = ['El', 'gos,', 'i', 'el', 'gat,', 'menjàven', 'plegats.']
cadena=" ".join(llista)  # l'espai farà de separador
print(cadena)
> El gos, i el gat, menjàven plegats.
```

`strip`: Elimina els caràcters indicats de l'inici i del final de la cadena, o els espais en blanc si no s'indiquen caràcters específics.

```
"introducció a Python".strip('nio')
> 'troducció a Pyth‘
" introducció   ".strip()
> 'introducció‘
```

`islower`, `isupper` i `isalpha`: Verifiquen si la cadena és majúscules, minúscules o tota de lletres respectivament.

`lower` i `upper`: Converteixen la cadena de majúscules a minúscules i de minúscules a majúscules respectivament.

Funcions

Fins ara hem escrit tots els programes en una única funció.

Per diverses raons (economia a l'escriure, manteniment del software, disseny) i sobretot quan resolem problemes més complexos val la pena fer servir diferents funcions. Cada funció resoldrà una part del problema, serà com un subprograma, o un programa dins del programa.

Recordem que les funcions no són res més que una seqüència d'instruccions amb un nom. Una funció es pot cridar des de qualsevol lloc del programa pel seu nom.

```
def sum(a,b):
    return a+b

a = 3
b = a*2
c = sum(a,b)
c

> 9
```

Funcions

L'àmbit o *scope* és el nom que donem als **llocs d'un programa** en els que *es pot fer referència* a una variable.

Només es pot fer referència a les variables definides dins d'una funció dins la pròpia funció, són locals, i per això poden tenir el mateix nom que variables externes.

L'única manera que té una funció per veure les variables d'una altra funció és passar-li com a paràmetre.

Exemple:

```
a = 7
def assignacio():
    a = 5
    print("dins, a val ", a)
assignacio()
print("fora, a val ",a)
```

Què es mostrarà a pantalla?

Crida de funcions

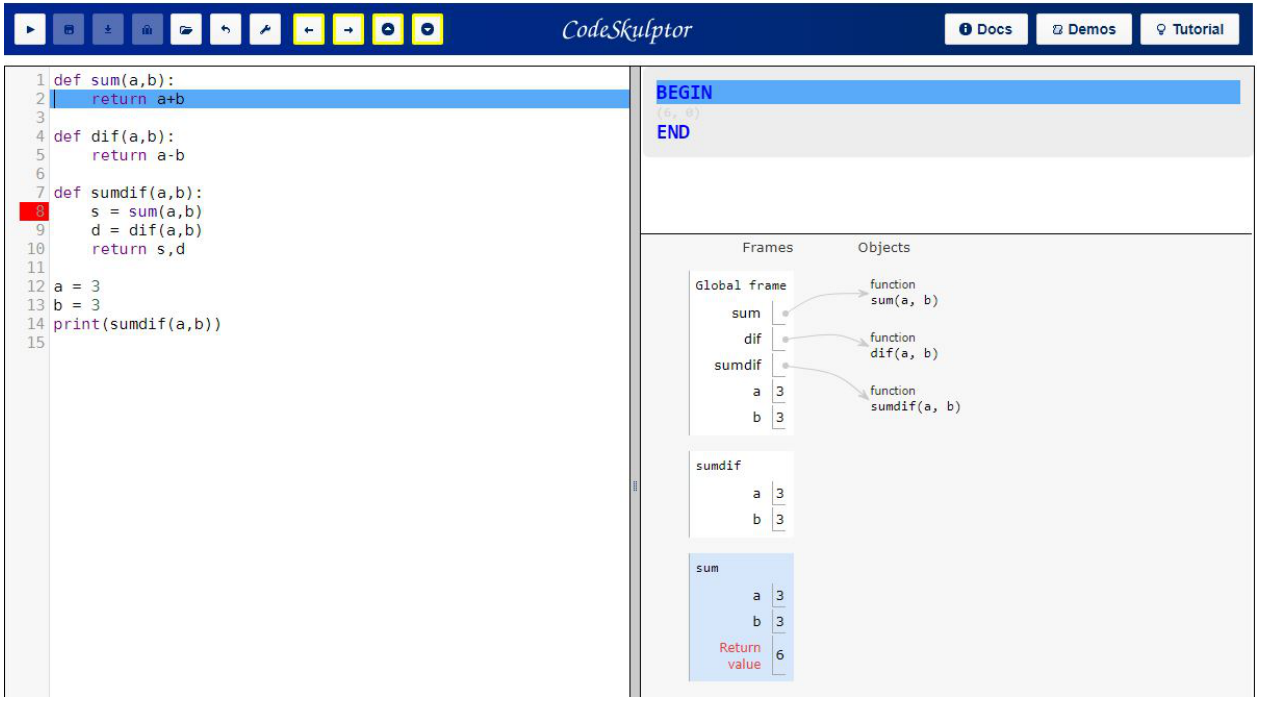
Quan Python rep la crida d'una funció, fa quatre coses:

- El programa que fa la crida se suspèn/congela en el punt de la crida.
- Els paràmetres de la funció passen a prendre els valors de la crida.
- S'executa el cos de la funció.
- Retorna el control al punt de programa posterior a la crida.

```
def sum(a,b):
    return a+b

def dif(a,b):
    return a-b

def sumdif(a,b):
    s = sum(a,b)
    d = dif(a,b)
    return s,d

a = 3
b = 3
print(sumdif(a,b))

> 6 0
```

Crida de funcions

Veiem-ho amb [Code Skulptor]



Operadors relacionals

Els operadors relacionals ens permeten combinar expressions. El seu resultat és un valor *booleà* (`True[1]` o `False[0]`). Podem comparar tot tipus de dades:

```
3<4
> 1
3*4 < 3+4
> 0
"hello" < "Hello"
> 0
```

Els operadors són:

```
<           # menor que
<=          # menor o igual que
==          # igual que
>=          # major o igual que
>           # major que
!=          # diferent que
```

Operadors booleans

Els operadors booleans ens permeten combinar expressions lògiques. El seu resultat és un valor booleà.

```
a = True
b = False
c = a and b
print(c)
print(a and b or c)
print(a or (not b) and c)
```

`and`: és veritat si i només si els dos operands ho són.

`or`: és veritat si al menys un dels dos operands ho és.

L'ordre de precedència és not, and i or.

> Quin és el resultat de cada expressió?

Estructures de control

Quan volem canviar el fil del programa en funció de si es compleix una condició o no, farem servir l'estructura `if`:

```
if a==0:
    print("valor neutre")
elif a < 0:
    print("valor negatiu")
elif a == 1:
    print("valor unitat")
else:
    print("altres valors")
```

Els `elif` són opcionals. En podem posar tants com vulguem.

`else` també és opcional.

Exemple

Càlcul del promig d'una seqüència de nombres:

```
def mitja():
    n = eval(input("Quants nombres tens?"))
    suma = 0.0
    for i in range(n):
        x = eval(input("Entra un nombre: "))
        suma = suma + x
    return("La mitja és: ", suma/n)
```

És correcte, però no gaire pràctic. Per què?

Estructures de control

Per solucionar el problema anterior podem usar un altre *estructura de bucle*: `while`.

```
i = 0
while i < 10:        #les instruccions s'executen mentre es compleix la condició
    print(i)
    i += 1
```

Llavors podem reescriure el programa anterior com:

```
def mitja():
    suma = 0.0
    comptador = 0
    mesnombres = "s"
    while mesnombres[0] == 's':
        x = eval(input("Entra un nombre: "))
        suma += x
        comptador += 1
        mesnombres = input("Hi ha més nombres (si o no)?")
    return suma/comptador
```

Però encara podem fer més eficient aquest codi!

Estructures de control

```
def mitja():
    suma = 0.0
    comptador = 0
    x = eval(input("Entra un nombre (negatiu per acabar): "))
    while x > 0:
        suma += x
        comptador += 1
        x = eval(input("Entra un nombre (negatiu per acabar): "))
    return suma/comptador
```

```
def mitja():
    suma = 0.0
    comptador = 0
    xStr = input("Entra un nombre (<Enter> per acabar): ")
    while xStr != "":
        x = eval(xStr)
        suma += x
        comptador += 1
        xStr = input("Entra un nombre (<Enter> per acabar): ")
    return suma/comptador
```

Python (3)

Col·lecions de dades

Exemples de col·leccions:

- Paraules d'un text.
- Estudiants d'un curs.
- Dades d'un experiment.
- Clients d'un negoci.
- Els gràfics que es poden dibuixar en una finestra.

Python ens dona suport per a la manipulació d'aquest tipus de dades.

Suposem que volem calcular la **mitja** i la **desviació estàndard** d'un conjunt de `n` nombres.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n-1}}$$

Col·lecions de dades

El que necessitem és emmagatzemar una col·lecció de coses (a priori no sabem quantes) en un “objecte”.

De fet, aquest tipus d’*objecte* ja l’hem fet servir, i es diu llista:

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

a='A B C D'
a.split()
> ['A','B','C','D']
```

Una llista és una **seqüència ordenada de coses**.

Els elements d'una llista s'indexen de la mateixa manera que una cadena de caràcters. De fet les llistes i els strings són conceptualment molt semblants, i podem aplicar-hi operadors semblants.

La diferència és el que contenen. Les llistes poden contenir **qualsevol tipus de dades**, incloent “classes” definides pel programador. Les llistes són **mutables**, és a dir, es poden canviar sobre la mateixa estructura (els strings no!).

Col·lecions de dades

Les llistes en Python són **dinàmiques**, poden créixer i decréixer durant l'execució del programa. Les llistes en Python són **inhomogènies**, poden contenir tipus diferents de dades. En resum, les llistes són **seqüències mutables d'objectes arbitraris**.

Es creen així:

```
a = [1,3,5,7,0]
b = ['spam', 0, 3.9]
c = []
d = [0] * 50
```

Podem afegir-hi o esborrar coses:

```
nums = []
x = eval(input("Entra un nombre: "))
while x >= 0:
    nums.append(x)
    x = eval(input("Entra un nombre: "))

del nums[1]
```

Col·lecions de dades

Donada una llista `l`:

- `l.append`: afegeix elements al final.
- `l.sort`: ordena els elements.
- `l.reverse`: inverteix la llista.
- `l.index(x)`: retorna l'índex del primer element igual a `x`.
- `l.count(x)`: retorna el nombre de vegades que apareix `x`.
- `l.remove(x)`: elimina la primera ocurrència de `x`.
- `l.pop(i)`: elimina l'ièssim element de la llista i retorna el seu valor.
- `x in l`: retorna una valor booleà en funció de si `x` és a la llista o no.

Suposant que tenim una llista formada per milers de milions d'elements, podriem ordenar aquestes operacions en funció del temps que trigarien a executar-se?

Exemple

```
def getNumbers():
    nums = []
    xStr = eval(input("Entra un nombre (<Enter> per acabar): "))
    while xStr != "":
        x = eval(xStr)
        nums.append(x)
        xStr = eval(input("Entra un nombre (<Enter> per acabar): "))
    return nums

def m(nums):
    suma = 0.0
    for num in nums:
        suma += num
    return suma/len(nums)

def stdDev(nums, mean):
    import math
    sumDev = 0.0
    for num in nums:
        dev = mean - num
        sumDev += dev * dev
    return math.sqrt(sumDev/len(nums)-1)

data = getNumbers()
```

Referències

Cada una de les dades que creem té una referència que podriem entendre com l'adreça de memòria on es pot localitzar.

Si executem:

```
a = "banana"
b = "banana"
```

`a` i `b` són dos noms diferents amb el mateix valor, però és la "mateixa" cadena de valors o guardem dues vegades a memòria la mateixa cadena de caràcters?

Cada objecte té un identificador únic, que podem obtenir amb la funció `id`:

```
id(a)
> 135044008

id(b)
> 135044008
```

Per tant, en aquest cas Python ha creat una estructura `banana` i les dues variables en fan referència.

Referències

Les llistes funcionen diferent (`a` i `b` tenen el mateix valor però no fan referència al mateix objecte):

```
a = [1,2,3]
b = [1,2,3]
print(id(a),id(b))
> 238870816, 245363636
```

Com que les variables fan referència a objectes, si una variable fa referència a una altra tenim:

```
a = [1,2,3]
b = a
print(id(a),id(b))
> 238870856, 238870856
```

Com que la llista té dos noms, direm que té un **àlies**. Això és perillós per objectes mutables!!! Pels immutables no hi ha problema.

Referències

El clonatge és una tècnica per la que fem una còpia de l'objecte en si, no de la referència. Pel cas de les llistes ho podem fer així:

```
a = [1,2,3]
b = a[:]
b[0] = 5
print(a,b)
> [1,2,3] [5,2,3]
```

Referències

Si passem una llista com a **argument** d'una funció, passem una **referència**, no una còpia. Considerem aquesta funció:

```
def head(l):
    return(l[0])

a = [1,2,3]
head(a)
> 1
a
> [1,2,3]
```

Considerem ara aquesta altra funció:

```
def deleteHead(l):
    del l[0]
    return l

a = [1,2,3]
deleteHead(a)
> [2,3]
a
```

Referències

Si retornem una llista també retornem una referència:

```
def tail(l):
    return l[1:]

a = [1,2,3]
rest = tail(a)
print(rest, a)
> [2,3] [1,2,3]
```

Com que la llista s'ha creat amb `:` és una nova llista. Qualsevol modificació de `rest` no té efectes en `a`.

En canvi en el següent cas:

```
numbers = [1,2,3]
def test(l):
    l.reverse()
test(numbers)
print(numbers)
> [3,2,1]
```

Llistes

Una llista imbricada és una llista que apareix com a element d'una altra llista.

`l = [0,1,3,['a','b']]`

Per obtenir un element d'una llista imbricada ho podem fer de dues maneres:

```
# amb dos passes...
li = l[3]
li[0]
> 'a'
# o bé ...
l[3][0]
> 'a'
```

I és que les llistes imbricades es fan servir per representar matrius:

```
m = [[1,2,3],[4,5,6],[7,8,9]]
m[0]
> [1,2,3]

m[1][1]
>5
```

Diccionaris

Python ens proporciona un altre tipus de col·lecció molt útil: els **diccionaris**.

La raó de la seva existència és que no sempre serà possible accedir a una dada pel seu índex, sinó per exemple, per algun valor que el defineix (p.e. un empleat pel seu DNI). És a dir, volem accedir a un valor per una **clau**.

Python crea els diccionaris així:

```
passwd = {'bill':'clinton', 'barack':'obama'}
```

I ens permet accedir-hi així:

```
passwd['bill']
> clinton
```

Els diccionaris són mutables:

```
passwd['bill'] = 'gates'
```

Exemple: omplir un diccionari des d'un fitxer.

Suposem que tenim una llista d'usuaris i els seus passwords en un fitxer. El format és una línia per usuari formada per dues paraules: el nom d'usuari i el password.

```
passwords = {}
f = open('passwords.txt','r')
for line in f.readlines():
    usr, passw = line.split()
    passwords[usr] = passw
f.close()
```

Diccionaris

```
p = {'a':'A', 'b':'B', 'c':'C', 'd':'D'}
for i in p.keys():
    print(i,end=',')
> a,b,c,d

for i in p.values():
    print(i,end=',')
> A,B,C,D

for i in p.items():
    print(i,end=',')
> ('a','A'),('b','B'),('c','C'),('d','D')

list(p.values())
> ['A','B','C','D']

'a' in p
> True
```

Tuples

Hi ha una altra classe de col·lecció a Python que és semblant a la llista, però que és immutable: la **tupla**.

```
t = 'a','b','c','d'
```

o

```
t = ('a','b','c','d')
```

Si només hi ha un element s'ha d'escriure amb una coma final:

```
t = ('a',)
type(t)
> tuple

t = ('a')
type(t)
> str
```

Les operacions son les mateixes que per les llistes (tenint en compte que són immutables!)

Exemple: Estadística de les paraules d'un document?

```
def numW():
    '''
    Aquest programa calcula l'estadística de les paraules d'un document
    '''
    fname = input("Nom del document")
    text = open(fname, 'r').read()
    text = text.lower()
    for ch in '!"·$%&/()=?¿|@#¢∞¬÷÷“”≠¡{}[]+-,.;:-_><':
        text.replace(ch,' ')
    words = text.split()
    counts = {}
    for w in words:
        if w in counts:
            counts[w] = counts[w] + 1
        else:
            counts[w] = 1
    n = eval(input("Quantes paraules vols analitzar (les més freqüents)?"))
    lfreq = []
    for w in counts:
        lfreq.append((counts[w],w))
    print(lfreq)
    lfreq.sort(reverse=True)
    print(lfreq)
```

Possibles preguntes d'exàmen relacionades amb el tema 2 (I)

1. Converteix a una iteració `while` la següent iteració `for`:

```
word = 'hola'
for a in word:
    print(a)
```

2. Quin és el valor de x?

```
x = 1 == 1 or 1 != 1
```

3. Què mostrarà el següent programa:

```
a = 2
if a + 1 > 3:
    print (a + 3)
elif a > 2:
    print(a + 2)
else:
    print(a + 1)
```

Possibles preguntes d'exàmen relacionades amb el tema 2 (II)

- Què es mostrarà per pantalla?

```
def funA(l):
    l.append(1)
    print(l)
l=[2,3,4]
funA(l)
print(l)

def funB(w):
    w=w+'s'
    print(w)
w="gat"
funB(w)
print(w)
```

  per què?

- Què sortirà a pantalla?

```
def funA(x,y,z):
    return (x + y) > z
```

Possibles preguntes d'exàmen relacionades amb el tema 2 (III)

- Fes un programa en Python que, donat un enter `N`, calculi totes les combinacions de 4 enters, `A`, `B`, `C`, `D`, que compleixen aquesta equació: `A^2+B^2+C^2+D^2 = N`. Nota: no s'han de considerar les combinacions repetides!
- Completa aquest programa (substituïnt ____):

```
define word_lengths(list_of_words):
    list_of_lengths = []
    for each ____ in ____:
            list_of_lengths.append(____)
    return list_of_lengths
```

  de manera que ens retorni una llista amb la longitud de les paraules:
  `word_lengths(["red", "green", "blue"]) => [3, 5, 4]`