Conceptes bàsics de programació i Python, orientat a 1r de Batxillerat

Enric X. Martín Rull,
Juny 2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Introducció

En aquest document trobarem de manera barrejada nocions bàsiques sobre el nostre llenguatge de programació, en aquest cas Python i nocions generals de programació i algorísmia. S'ha optat per aquest format perquè la programació és una matèria que de manera natural permet anar alternant els exercicis pràctics amb alguns conceptes teòrics o directament aplicar aprenentatge basat en problemes.

El llenguatge Python s'ha anat fent popular com alternativa lliure i gratuïta a altres plataformes de desenvolupament, especialment en aplicacions científiques. Pertany a la família dels llenguatges interpretats. Això vol dir que, en temps d'execució (mentre nosaltres fem córrer el programa), es van executant les instruccions d'una en una a una mena de màquina virtual. Això facilita la trobada de possibles errors de programació i la seva correcció, així com la portabilitat del codi d'una màquina a una altra o d'un sistema operatiu a un altre.

Com els altres llenguatges interpretats, Python permet declarar variables a mida que les necessitem, no cal declarar-les a priori ni fer un anàlisi de quins tipus ens caldran o quina mida i precisió han de tenir per contenir les nostres dades. Això es pot considerar pràctic, però és un bon exercici fer un plantejament previ de les variables, els seus tipus i dimensions per agafar **bons hàbits de programació**.
També és un bon costum posar noms descriptius a les variables, a les funcions i posar un mínim de comentaris al codi per facilitar-ne la llegibilitat, per altres persones, i per nosaltres mateixos en el futur.

Estructura d'aquest document

A continuació d'aquest apartat trobarem uns conceptes bàsics sobre com els ordinadors representen la informació en el que habitual coneixerem com variables, i els tipus de dades més habituals.

El capítol primer explica breument els **aspectes bàsics del llenguatge Python**. Veurem les particularitats d'aquest llenguatge amb els tipus de dades, les operacions que podem fer i l'estructura del llenguatge de programació. A partir d'aquest moment ja podríem començar a programar.

Caldrà però reflexionar una mica i pensar com encararem els exercicis de programació i la implementació de les solucions, d'aquí la necessitat del segon capítol. Aquest explora com l'**algorísmia** serveix per resoldre problemes i com hem d'encarar i estructurar els nostres programes perquè siguin solucions elegants i eficients pels problemes plantejats.

El capítol tercer ens mostrarà perquè Python ha esdevingut un llenguatge tan popular. Existeixen múltiples **llibreries** per poder resoldre tota mena de problemes i ampliar les funcionalitats dels nostres programes. Aprendrem a utilitzar-les, veurem algunes de les més útils i també com fer-ne de pròpies.

Finalment al capítol quart trobarem uns primers **programes** estàndard a desenvolupar en el que es treballen conceptes clau d'algorísmia i codificació. Seran un bon punt de partida per, a partir d'ells, estendre les seves funcionalitats o desenvolupar petits projectes.

Index d'aquest document:

Introducció

Estructura d'aquest document

Conceptes bàsics

1. Petita introducció a Python
    - 1.1 Objectiu
    - 1.2 Utilització de dades a Python, variables, tipus, constants…
    - 1.3 Operacions aritmètiques, aprenem a treballar amb les dades.
    - 1.4 Operacions amb altres tipus: booleans, cadenes, conjunts i llistes
    - 1.5 Entrada i sortida de dades, interacció amb l'usuari.
    - 1.6 Treball amb fitxers. Guardar les dades
    - 1.7 Funcions en Python.
    - 1.8 Estructures bàsiques de programació en Python: bifurcacions, repeticions.
    - 1.9 Comentaris en Python.

2. Bones pràctiques de programació. Algorísmica
    - 2.1 Algorísmia
    - 2.2 Estructura del codi
    - 2.3 Estructura de les dades
    - 2.4 Creació de classes
    - 2.5 Estil de programació, noms, comentaris.

3. Llibreries en Python, utilització i creació.
    - 3.1 Utilització de llibreries
    - 3.2 Creació de llibreries
    - 3.3 Algunes llibreries interessants

4. Primers projectes en Python.
    - 4.1 Programació d'un accés amb password. Identificar-nos.
    - 4.2 Programa endevinar nombre
    - 4.3 Programa per comptar lletres
    - 4.4 Programa de càlcul de probabilitat amb daus
    - 4.5 Programa per tornar canvi, l'algorisme de tornar canvi
    - 4.6 Programa per encriptar i desencriptar
    - 4.7 Programa per cercar el mínim a una paràbola.
    - 4.8 Primers i factors

Conceptes bàsics

Els ordinadors guarden la informació a memòria en format binari, és a dir, amb zeros i uns. Si imaginem que volem representar un nombre enter amb 4 bits, podem pensar en que tenim 16 possibilitats:

Codi binari	Valor enter
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

En aquest cas només ens ha servit per nombres positius, entre 0 i 15. El valor de cada bit ve donat per la seva posició. Llegint d'esquerra a dreta el seu valor és: 8, 4, 2 i 1.

Podem llegir un nombre binari com per exemple '0110' calculant:
0 x 8 + 1 x 4 + 1 x 2 + 0 = 6

En general si volem representar nombres positius amb n bits, el nombre màxim de valors representables serà 2^n-1.

Exercici 1.1.1: Representa en binari amb 4 bits el nombre 11.

Exercici 1.1.2: Representa en binari amb 6 bits el nombre 23.

Exercici 1.1.3: Quin és el nombre més gran que podem representar amb 6 bits?

Exercici 1.1.4: Quin és el nombre més gran que podem representar amb 8 bits?

Exercici 1.1.5: Representa en binari amb 8 bits el nombre 103.

La forma presentada de representar els nombres no permet representar nombres negatius, això ho podem fer, per exemple, indicant un bit de signe. Tindrem el primer bit (per l'esquerra) per indicar si és positiu (0) o negatiu (1) i després posarem el valor en els tres bits restants. Veiem-ho altre cop amb una taula:

Codi binari	Valor enter
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	0
1001	-1
1010	-2
1011	-3
1100	-4
1101	-5
1110	-6
1111	-7

Aquest sistema només pot codificar 15 nombres diferents (fixeu-vos que hi ha dos zeros!) però serveix per representar positius i negatius i s'entén fàcilment.
El sistema més utilitzat per representar enters s'anomena complement a 2. Té l'avantatge que al sumar nombres positius i negatius el resultat és correcte. En el cas anterior caldria fer per cada suma una comparació de magnituds i signes.

Anem primer a veure com suma un ordinador en binari, farem un exemple amb dos nombres positius de 4 bits (3 i 5), suposem que codifiquem tot com a nombres positius:

Nombre 1: 0011 (3 en base 10)
Nombre 2: 0101 (5 en base 10)

Nota. Anomenem base 10 la que acostumem a utilitzar, que té 10 dígits diferents (entre 0 i 9) i que ve del fet de que tenim 10 dits.

Farem la suma com en base 10, de dreta a esquerra i comptant si ens emportem una, la diferència és que la base és 2, per tant quan sumem 1 amb 1 dóna 0 i cal afegir 1 a la següent posició.

Nombre 1	0	0	1	1
Nombre 2	0	1	0	1
Resultat				

Sumem el primer bit (dreta), 1+1=0 i ens emportem 1.

			+1	
Nombre 1	0	0	1	1
Nombre 2	0	1	0	1
Resultat				0

Sumem ara els segons bits, juntament amb el que ens emportem, 1+1+0=0, ens emportem 1.

		+1		
Nombre 1	0	0	1	1
Nombre 2	0	1	0	1
Resultat			0	0

Seguim amb els tercers bits, juntament amb el que ens emportem, 1+0+1=0, ens emportem 1.

	+1			
Nombre 1	0	0	1	1
Nombre 2	0	1	0	1
Resultat		0	0	0

Acabem amb els quarts bits i el que ens emportem, 1+0+0=1. Hem acabat.

Nombre 1	0	0	1	1
Nombre 2	0	1	0	1
Resultat	1	0	0	0

El resultat és 1000 que codificat en 4 bits en positiu és 8, per tant 3+5=8. Correcte.

Codificació en complement a 2

Anem a veure com els ordinadors acostumen a representar els nombres enters. Farem exemples amb 8 bits. Si el nombre és positiu es representa directament.

Exemple:
Nombre 23 = 0 0 0 1 0 1 1 1 (8 bits) = 0x128 + 0x64 + 0x32 + 1x16 + 0x8 + 1x4 + 1x2 + 1x1

Si el nombre és negatiu, seguirem els següents passos.
- el representem en positiu.
- invertim els bits (0 a 1 i 1 a 0)
- sumem 1 a la codificació.

Exemple:
Nombre -17
- pas 1: 17 = 0 0 0 1 0 0 0 1 = 0x128 + 0x64 + 0x32 + 1x16 + 0x8 + 0x4 + 0x2 + 1x1
- pas 2, invertim: 0 0 0 1 0 0 0 1 → 1 1 1 0 1 1 1 0
- pas 3, sumem 1: 1 1 1 0 1 1 1 0 + 1 = 1 1 1 0 1 1 1 1

Hem acabat (per estranya que ens sembli, la representació de -17 és 1 1 1 0 1 1 1 1.

Nota: una característica dels nombres negatius és que sempre comencen (per l'esquerra) amb un bit a 1.

Anem a veure que la suma bit a bit de nombres representats en complement a 2 funciona i dóna el resultat esperat (recordeu que va de dreta a esquerra).

	+1	+1	+1	+1	+1	+1	+1	
Nombre 1 = 23	0	0	0	1	0	1	1	1
Nombre 2 = -17	1	1	1	0	1	1	1	1
Resultat	0	0	0	0	0	1	1	0

Veiem que el resultat ha donat (com esperàvem) 6, nombre positiu. En aquest cas sempre en hem emportat una a partir del primer bit.

Exercici 1.1.6: Representa en complement a 2 amb 8 bits el nombre -67.

Exercici 1.1.7: Representa en complement a 2 amb 8 bits el nombre -46.

Veiem un altre exemple:

					+1	+1		
Nombre 1 = 35	0	0	1	0	0	0	1	1
Nombre 2 = -42	1	1	0	1	0	1	1	0
Resultat	1	1	1	1	1	0	0	1

El resultat ha donat el nombre binari: 1 1 1 1 1 0 0 1

Per saber quin nombre és en base 10 farem els passos de codificació al revés:

- Restem 1 per tant ens queda: 1 1 1 1 1 0 0 0
- Invertim els bits i ens queda: 0 0 0 0 0 1 1 1

El resultat és la codificació de -7, per tant tot ha estat correcte!!

Exercici 1.1.8: Codifica en complement a 2 els nombres 105 i -45 i suma'ls. Comprova que el resultat és correcte.

Nombres reals.

Acabarem aquesta part inicial pensant com els ordinadors representaran els nombres reals (no entrarem en profunditat). Estem parlant de com representar nombres com:

0.000135 , -124.1, 3.141592854 , 6.12×10^{-3} , 6.023×10^{23} , etcètera.

La codificació més habitual és diferenciant les xifres significatives del nombre (que s'anomenarà mantisa) les xifres de l'exponent i els signes respectius de cada un. Sols es representen les xifres significatives, per exemple en el nombre:

0.000135

La mantisa a representar serà 135 i l'exponent -7.
S'acostumen a utilitzar cadenes llargues de bits (32, 64…) per representar els reals. Veiem un exemple de representació en 32 bits:

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	. . . .	1	0
Significat:	SE	E7	E6	E5	E4	E3	E2	E1	SM	M23	M22	M21	. . . .	M2	M1

Tenim el bit SE pel signe de l'exponent, 7 bits per codificar l'exponent (E7 a E0), el bit SM pel signe de la mantisa i 23 bits per representar la magnitud de la mantisa.

L'exponent representable en aquest real de 32 bits anirà de -127 a 127.
La mantisa representable en aquest real de 32 bits anirà de -8388607 a 8388607.

En cas de que amb 32 bits no tinguem prou precisió es faran servir 64, 128, etc.

Nota. Aquesta forma de representar els reals permetrà al maquinari de l'ordinador fer sumes, restes,multiplicacions i divisions de nombres fàcilment.

Caràcters

Anomenem caràcters els elements amb els que representem lletres, nombres i símbols diversos, parlarem del caràcter ‘a’, del caràcter ‘3’, del símbol ‘%’ o ‘$’, etc.
Habitualment els caràcters es representen en 8 bits, el que ens permet tenir-ne 256 de diferents. Existeix un estàndard anomenat codi ASCII (American of Code for Information Interchange) que assigna un codi de 8 bits i a cada caràcter. La següent taula en mostra uns quants:

Codi	Nombre	Caràcter	Codi	Nombre	Caràcter
00110000	48	‘0’	01000001	65	‘A’
00110001	49	‘1’	01000010	66	‘B’
00110010	50	‘2’	01000010	67	‘C’
00110011	51	‘3’	01000010	68	‘D’
00110100	52	‘4’	01000010	69	‘E’
00110101	53	‘5’	01000010	70	‘F’
00110110	54	‘6’	01000010	71	‘G’
…	…	…	…	…	…

Podeu fer una cerca de “Taula ASCII” per trobar la taula completa. Existeixen altres codificacions més extenses per codificar tota mena d’accents, lletres gregues, caràcters d’altres idiomes, etc.

Cadenes

Anomenem cadena (en anglès *string*) a una sèrie de caràcters que representaran frases, textos, etc. No són més que els caràcters posats un darrere l’altre a la memòria i un indicador de final, per saber que s’acaba la cadena (habitualment un codi 00000000).

Exemple. Codificació a memòria de ‘Hola’

Posició	1	2	3	4	5
Caràcter	‘H’	‘o’	‘l’	‘a’	NULL
Nombre	72	111	108	97	00
Codi	01001000	01101111	01101100	01100001	00000000

El caràcter 00 se l’acostuma a anomenar NULL i indica el final de la cadena.

Exercici 1.1.9: Cerca una taula ASCII i codifica la paraula ‘Python’ com a l’exemple anterior.

Exercici 1.1.10: Cerca una taula ASCII i codifica amb una cadena el teu nom i cognoms. Quants caràcters ocupa?

Veurem que Python a partir d'aquestes formes bàsiques de representar la informació ens oferirà mètodes cada cop més elaborats i fàcils d'utilitzar. Aquesta és una de les fortaleses d'aquest llenguatge.

1. Petita introducció a Python

1.1 Objectiu

Aquest apartat no pretén ser un manual extens del llenguatge Python, sino una guia mínima per la seva utilització, amb exemples i una selecció dels elements que considerem adequats per una assignatura de nivell de Batxillerat.

Una referència extensa es pot trobar a: https://docs.python.org/3/reference/

Al web docs.python.org es van actualitzant els documents de referència del llenguatge, les seves llibreries, novetats, etc. Normalment tota la documentació es troba en anglès i castellà entre d'altres idiomes.

El web general: www.python.org s'hi trobaran també descàrregues de materials, experiències d'usuaris, novetats, una wiki de treball amb Python, etc.

1.2 Utilització de dades a Python, variables, tipus, constants…

El llenguatge de programació Python s'encarregarà de la codificació a la memòria de l'ordinador de totes les dades que vulguem utilitzar.
Nosaltres farem servir variables (cada una té un nom de referència) per anar treballant i l'intèrpret la ubicarà a la memòria, posarà els 1 i 0 que calguin, i ens la mostrarà quan faci falta.

Les dades en Python es guarden en variables, quan assignem un valor a un símbol, creem una variable. A l'exemple creem una variable fent l'assignació "x=12".

```
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.

In [1]: x=12

In [2]: type(x)
Out[2]: int

In [3]: x
Out[3]: 12
```

A partir d'aquest moment existeix una variable x de tipus enter (la comanda *type* ens retorna el tipus assignat a la variable) i amb valor inicial 12. El nom de la variable el triarem nosaltres.

Tipus de variables

En Python podem crear variables de tipus: enter, real, complexe, booleà, cadena, tupla, conjunt, rang, llista i diccionari. Anem a veure com es defineixen aquests tipus:

Tipus	Exemple de creació	Descripció
enter (int)	x = 12	Variable per contenir nombres enters
real (float)	y = 4.2	Variable per contenir nombres reals
complexe (complex)	z = 5.0 + 2j	Variable per contenir nombres complexes (la lletra emprada és la j)
booleà (bool)	bit = False	Pot contenir estats lògics (True o False)
cadena (string)	s = ‘Maria’	Conté seqüències de caràcters. Es poden definir amb cometa simple o doble (essent coherents a la definició)
tupla (tuple)	a = (‘Pedro’, 17)	Conté una estructura amb diversos camps, que poden tenir tipus diferents. Accedim amb als camps amb [ ] els camps es comencen a comptar per 0.

Exemple d’utilització de les tuples, en aquest cas amb 3 camps:

```
In [37]: alumne=("Joan",17,True)

In [38]: type(alumne)
Out[38]: tuple

In [39]: nom = alumne[0]

In [40]: nom
Out[40]: 'Joan'

In [41]: edat = alumne[1]

In [42]: edat
Out[42]: 17

In [43]: aprovat = alumne[2]

In [44]: aprovat
Out[44]: True
```

Tipus	Exemple de creació	Descripció
conjunt (set)	c = { ‘blau’, ’verd’, ’vermell’, ‘groc’ }	Variable per contenir elements no ordenats i no repetits i treballar amb operacions de conjunts.
rang (range)	r = range (-3, 3, 1)	Els rangs es defineixen amb un interval i un pas, es proporciona el valor inicial, el final i el pas. A l’exemple tindrem que el rang conté: -3, -2, -1, 0, 1 i 2. *Atenció, el “final” no entra al rang!*
llista (list)	m = [2, 4, 1, 7, 3]	Seqüència ordenada d’elements, a diferència dels conjunts pot haver elements repetits. Els elements de la llista poden ser heterogenis, inclús pot haver llistes dins de les llistes!
diccionari (dict)	d = { 1:’a’, 2:’b’, 3:’c’ }	El tipus diccionari associa una clau a un o més elements, per exemple la clau 1 ens donarà ‘a’, la 2 ‘b’, etc.

Exemple d’utilització de llistes amb tipus heterogenis (atenció que pot ser complicat, s’aconsella crear-les amb seny):

```
In [80]: llista = [ 1, 3, [2,4,6], 'a']

In [81]: llista [0]
Out[81]: 1

In [82]: llista [2]
Out[82]: [2, 4, 6]

In [83]: llista [2][2]
Out[83]: 6

In [84]: llista [3]
Out[84]: 'a'
```

Constants

A diferència d’altres llenguatges de programació Python no permet definir constants. Existeixen símbols predeclarats com *True* o *False* i si dins d’un programa volem anomenar un símbol que no canviï el seu valor es segueix el conveni de posar el seu nom en majúscules i si cal, amb separació de _ (guions baixos).

Exemple: RADI_RODA = 12.2

El llenguatge Python té predefinides algunes constants com els valors booleans *True* i *False*, *None*, etc. Si volem emprar constants matemàtiques usarem llibreries com ara numpy.

Tipus mutables i immutables

Dels tipus principals (que són els que habitualment farem servir) n'hi ha que són mutables i n'hi ha que són immutables. Això té implicacions per entendre el funcionament de Python i sobretot quan passem paràmetres a funcions.
Un tipus mutable és aquell que permet, en temps d'execució, canviar-ne el contingut. En canvi, un tipus immutable no pot ser modificat una vegada l'hem creat. La següent taula mostra quins tipus són mutables i quins immutables:

	Tipus mutables	Tipus immutables
Simples		int, long, float, complex, bool
Compostos	list, range, set, dict	string, tuple

Anem a veure amb un exemple què significa això, a més de la funció de Python *type()* que ja hem utilitzat anteriorment, usarem la funció *id()* que ens retorna un identificador únic de la variable emprada.

Exemple 1. Tipus simple immutable, enter.

```
In [24]: x=12

In [25]: type(x)
Out[25]: int

In [26]: id(x)
Out[26]: 93856578502304

In [27]: x=6

In [28]: id(x)
Out[28]: 93856578502112
```

Hem creat una variable x de tipus enter, amb valor 12, i després li hem assignat un altre valor (6). Com podem veure, els identificadors de la variable han canviat. És a dir, ens ha creat una altra variable, que també es diu x, però no és la mateixa d'abans.

Exemple 2. Tipus compost mutable, llista.

```
In [30]: llista = [2,4,5,7,8]

In [31]: type(llista)
Out[31]: list

In [32]: id(llista)
Out[32]: 139999930582272

In [33]: llista[0]=1

In [34]: llista
Out[34]: [1, 4, 5, 7, 8]

In [35]: id(llista)
Out[35]: 139999930582272
```

Hem creat una llista (tipus list) amb valors inicials [2, 4, 5, 7,8] i després hem canviat el primer element (a la posició 0) per un 1. Com podem veure el id de la llista no varia, per tant ens permet en temps d'execució fer-li variacions.

Exemple 3. Tipus compost immutable, tupla.

```
In [37]: t = ('Antonio',15)

In [38]: type(t)
Out[38]: tuple

In [39]: id(t)
Out[39]: 139999942948608

In [40]: t[1]=17
Traceback (most recent call last):

  File "<ipython-input-40-7d563d5cff48>", line 1, in <module>
    t[1]=17

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

Com podem veure el tipus compost tupla, al ser immutable, directament no deixa modificar un dels seus camps generant un error en el moment en que ho intentem.
Caldrà tenir en compte això de cara als programes més complexes que crearem en el futur.

Quan passem un tipus mutable a una funció, si dins de la funció el modifiquem, quedarà modificat a fora. El darrer exemple, avança la creació de funcions i la comanda *print* però ens ajudarà a entendre el concepte:

Exemple 4. Llista mutable

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def canvi(l):
    l[2]=65

llista=[1,2,3,5,6]
canvi(llista)

print(llista)
```

Resultat:

```
In [49]: runfile('/home/enric/Escriptori/Treball Profe/Programes/ProgsPython,
enric/Escriptori/Treball Profe/Programes/ProgsPython')
[1, 2, 65, 5, 6]
```

Veiem que la llista ha quedat modificada després de la crida a la funció.

En canvi, si utilitzem un tipus immutable com els enters…

Exemple 5. Enter immutable

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def canvi(x):
    x=65

x=4
canvi(x)

print(x)
```

Resultat:

```
In [50]: runfile('/home/enric/Escriptori/Treball Profe/Programes/ProgsPython,
enric/Escriptori/Treball Profe/Programes/ProgsPython')
4
```

Veiem que no s'ha canviat el valor de la variable x.

Quan passem a una funció un paràmetre que és de tipus mutable, es passa la referència a la variable i per tant aquesta serà modificada (en altres llenguatges, com C, se'n diu pas per referència) , en canvi quan passem una variable de tipus immutable se'n passa una còpia (d'això en C se'n diu pas per valor).

1.3 Operacions aritmètiques, aprenem a treballar amb les dades.

Assignació

En Python utilitzarem el símbol ‘=’ per assignar un valor a una variable. Com hem vist, si la variable no existeix, també la crea i li atribueix un tipus.

Exemples:
x = 3 (int), y = 2.12 (float), z = [2,3,4] (llista).

Operadors aritmètics

Els operadors aritmètics són els següents:

- Suma ‘+’ :   x = x + 3, y = 4 + x
- Resta ‘-’ :   y = y - 1, x = 4.5 - 2.0
- Negació ‘-’:   y = - x
- Multiplicació ‘*’ :   x = x * 3, y = x * 4
- Divisió ‘/’ :   x = x / 2, z = z / 4.0
- Divisió entera ‘//’ :   x = x // 2 (només torna la part entera de la divisió)
- Potència ‘**’ :   x = 4 ** 2, y = x ** 3
- Mòdul (reste de la divisió):   x = 7 % 2, y = 11 % x

Els podrem aplicar a variables enteres, reals o complexes.

Precedència d’operadors

L’ordre en que s’executen les operacions aritmètiques és:

Potència (**) , Negació (-), Producte (*, /, %, //), Suma (+, -)

En cas de dubte o per facilitar la lectura del codi, es poden posar parèntesis.

Exemple 1:

x = 4 * 3 + 2 = 14
x = 4 * (3 + 2) = 20

Combinació assignació/operació

És possible combinar les assignacions i operacions en una única expressió, per exemple si volem incrementar x en 4, podem escriure:

x += 4

Aquesta combinació la podem fer amb tots els operadors aritmètics, per tant tindrem: ‘+=’, ‘-=’, ‘*=’, ‘/=’, ‘**=’, ‘//=’ i ‘%=’

Exemple 2:

```
x = 7
x **= 2
x +=1           (El valor final de x, serà 50: x = 7² + 1)
```

Exercici 1.3.1: Calcula el valor de la variable y després d’executar aquest codi,

```
x = 12
y = x % 5
y **= 4
```

Exercici 1.3.2: Calcula el valor de les variable x i y després d’executar aquest codi,

```
x = 10
y = x ** 2 + 3
y -= 5
x = ( y // x )
```

1.4 Operacions amb altres tipus: booleans, cadenes, conjunts i llistes

1.4.1 Operacions amb booleans

Les variables de tipus Booleà tenen dos valors possibles: cert i fals. Això en Python es representa amb les paraules reservades: *True* i *False*.

- Podem assignar valors a variables booleanes:

    b = *False*
    b = (a>3) (b valdrà *True* si a>3 i *False* en els altres casos)
    c = *True*

- Podem fer la and (torna cert si les dues són certes) entre dues variables booleanes:

    c = a *and* b

- Podem fer la or (torna cert si una és certa) entre dues variables booleanes:

    c = a *or* b

- Podem fer la not (negar, si és cert passa a fals i a l’inrevés) el valor d’una variable booleana:

    a = *not* b

Exercici 1.4.1.1: Calcula el valor final de la variable b després d'executar aquest codi,

```
x = 12
a = (x > 10) and (x < 20)
b = not a
```

1.4.2 Operacions amb cadenes

Les cadenes o *strings* són seqüències de caràcters, com hem vist es creen quan se'ls fa una assignació.

- Podem assignar valors a variables de tipus string:

  s = "Aprenc Python"

- Podem concatenar (+) dues variables de tipus *string*:

  s1 = "Hola"
  s2 = "amics"
  s3 = s1 + s2   (s3 valdrà "Holaamics")

- Podem repetir (*) trossos de variables de tipus *string*:

  s1 = "Hola"
  s2 = s1 * 3   (s2 valdrà "HolaHolaHola")

- Podem tallar ([ ]) trossos d'un *string*:

  s1 = "Hola amics"
  s2 = s1[3]   (s2 valdrà "a")
  s2 = s1[3:7]   (s2 valdrà "a am")
  s2 = s1[-2]   (s2 valdrà "c")

  Nota: Cal estar atents a la numeració dels caràcters de la cadenai veure que en les seqüències [:] el darrer element no s'extreu! Fixeu-vos en la taula:

Cadena	H	o	l	a		a	m	i	c	s
Posició +	0	1	2	3	4	5	6	7	8	9
Posició -	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Exercici 1.4.2.1: Donada la cadena “Hola bon dia, com esteu?”, trobeu la x i la y per extreure el segment “bon dia”

s1 = “Hola bon dia, com esteu?”
s = s1 [x : y]

Exercici 1.4.2.2: Quin resultat tindrem a la variable s si fem servir aquest codi?

s1 = “Hola bon dia, com esteu?”
s = s1 [4: -5]

- Podem cercar pertanyença (in / not in) dins d’un *string*:

    s1 = “Python és genial“
    b = “Python” in s1 (b valdrà True)
    b = “Java” in s1 (b valdrà False)

- Finalment podem comparar (== / !=) dos *strings* per saber si són iguals:

    s1 = “Python és genial“
    b = (“Python és fantàstic” == s1) (b valdrà False)

Funcions aplicables al tipus cadena

A més d’aquestes operacions, les cadenes tenen una sèrie de funcions implícites, que ens permeten treballar amb elles quan hem creat una variable del tipus cadena.
Per exemple si creem una cadena *s* fent:

s= ”Python és genial”

Podrem cridar (es mostren algunes de les funcions del tipus):

- len() x=len(s) ens tornarà la longitud de la cadena en un enter (en aquest cas 16).
- s.capitalize() convertirà el primer caràcter a lletres majúscules.
- s.upper() passa tota la cadena a lletres majúscules.
- s.lower() passa tota la cadena a lletres minúscules.
- s.count(cadena) retorna quants cops hi ha cadena dins de s.
- s.find(cadena) cerca cadena dins de s i ens torna la posició on és.

- s.isalpha() ens retorna *True* si tots els caràcters són de l'alfabet.

- s.isdigit() ens retorna *True* si tots els caràcters són nombres del 0-9.

- s.isalnum() ens retorna *True* si tots els caràcters són lletres o nombres.

- s.split(separador) ens retorna una llista amb s partit en trossos segons el separador indicat.
A l'exemple si cridem s.split(" ") ens tornarà una llista amb els elements ["Python","és","genial"]

1.4.3 Operacions amb conjunts

El tipus conjunt permet tenir elements no repetits i sense un ordre concret. Podem declarar un conjunt fent:

colors1 = { "blau", "verd", "vermell", "groc"}

i un altre conjunt fent:

colors2 = { "verd", "negre", "blanc"}

Els operadors bàsics de conjunts són:

- unió (|) permetrà unir dos conjunts:

colors3 = colors1 | colors2 (colors3 tindrà 6 elements)

- intersecció (&) calcularà els elements dels dos conjunts:

colors3 = colors1 & colors2 (colors3 tindrà 1 element, "verd")

- diferència (-) eliminarà del primer conjunts els elements del segon:

colors3 = colors1 - colors2 (colors3 tindrà 3 elements, "blau", "vermell" i "groc")

Exercici 1.4.3.1: Donats els següents conjunts:

s1 = { 'A', 'B', 'D'}
s2 = { 'B', 'C', 'D'}
s3 = { 'A', 'E', 'I', 'O', 'U'}

Quin resultat tindrem al conjunt s després de fer:

s = (s1 | s3) - s2

Funcions aplicables al tipus conjunt

Com amb les cadenes, amb el tipus conjunt també podem utilitzar una sèrie de funcions que ens permetran treballar fàcilment amb ells. Si tenim els conjunts:

```
c1 = { “blau”, ”verd”, ”vermell”, ”groc”}
c2 = { “blau”, ”verd”, ”vermell”}
```

Veiem-ne algunes:

- len() — x=len(c1) ens tornarà el nombre d’elements d’un conjunt (4)
- c1.add(elem) — Afegirà un element nou (elem) al conjunt c1.
- c1.remove(elem) — Eliminarà l’element (elem) del conjunt c1.
- c1.copy() — Ens retornarà una còpia del conjunt, amb tots els seus elements. Per exemple: d = c1.copy( )
- c1.clear() — Elimina tots els elements del conjunt c1.
- c1.intersection(c2) — Retorna la intersecció entre els conjunts c1 i c2. En aquest cas retornaria els tres elements “blau”, “verd”, “vermell”.
- c1.union(c2) — Retorna la unió entre els conjunts c1 i c2. En aquest cas retornaria els quatre elements “blau”, “verd”, “vermell”, “groc”.
- c1.difference(c2) — Retorna la diferència entre els conjunts c1 i c2. En aquest cas retornaria només l’element “groc”.

1.4.4 Operacions amb llistes

L’operador [ ] serveix per crear una llista. La podem crear buida, o directament amb uns elements inicials.

```
x =  [  ]
y =  [ 1, 2, 3, 4, 5]
```

La majoria d’operacions amb llistes, seran **mutables** és a dir, canvien el contingut de la llista en executar-les. Altres són **immutables** i retornaran una nova llista o un valor, sense modificar el contingut de la llista.
Veurem ara una sèrie de funcions sobre el tipus llista, ja implementades pel llenguatge Python, segons aquesta classificació. Existeixen més funcions (no s’han considerat imprescindibles) que poden ser trobades en els manuals de referència del llenguatge.

Operacions mutables de les llistes

- Afegir (append) un element a una llista. L'afegirà pel final.

  x = [1, 2, 3]
  x.append (5) (x tindrà ara [1, 2, 3, 5])

- Estendre (extend) afegeix una llista a una llista. L'afegirà pel final.

  x = [1, 2, 3]
  x.append ([4,5]) (x tindrà ara [1, 2, 3, 4, 5])

- Inserir (insert) afegeix un element a una posició concreta d'una llista.

  x = [1, 2, 3, 4, 5]
  x.insert (2, 12) (x tindrà ara [1, 2, 12, 3, 4, 5], hem afegit a la posició 2 (es compta des de 0) un 12)

- Eliminar (del) elimina un element d'una posició concreta d'una llista.

  x = [1, 2, 3, 4, 5]
  x.del (2) (x tindrà ara [1, 2, 4, 5], hem eliminat el 3 de la posició 2 (es compta des de 0) de la llista)

- Eliminar (remove) elimina la primera aparició d'un element d'una llista.

  x = [1, 2, 3, 2, 1]
  x.remove (2) (x tindrà ara [1, 3, 2, 1], s'ha eliminat la primera aparició de l'element 2)

- Girar (reverse) dóna la volta a una llista, el primer serà l'últim, etc.

  x = [1, 2, 3, 4, 5]
  x.reverse ( ) (x tindrà ara [5, 4, 3, 2, 1])

- Ordenar (sort) Ordena una llista de petit a gran (és la opció per defecte).

  x = [1, 4, 3, 5, 2]
  x.sort ( ) (x tindrà ara [1, 2, 3, 4, 5])

  La funció sort( ) pot tenir un paràmetre sort (reverse= True) per ordenar de gran a petit.
  Els elements de la llista han de ser del mateix tipus, si volem ordenar una llista de cadenes ho farà per ordre alfabètic.

  La funció sort, permet passar-li com a paràmetre una funció (key), per definir un criteri d'ordenació diferent (per exemple, llargada d'una cadena). Podem

usar una funció preexistent o definir una pròpia (això es veurà més endavant).

x = [‘Anna’, ‘Xi’, ‘Fatima’]
x. sort (key = len)
(x valdrà ara [‘Xi’, ‘Anna’, ‘Fatima’])

Exercici 1.4.4.1: Donades les següents llistes:

n1 = [‘Juan’, ‘Enric’, ‘Sarai’]
n2 = [‘Ahmed’, ‘Laura’]
n3 = [‘Liu’, ‘Antonio’, ‘Maria’]

Generar una única llista noms[ ] amb els noms de n1, n2 i n3 ordenats alfabèticament.

Operacions immutables de les llistes

Aquestes funcions i operadors, a diferència de les anteriors, no modificaran el contingut de les llistes, o bé retornaran una llista nova, o un element o algun valor relacionat.

- Ordenar (sorted) Retorna una nova llista ordenada de petit a gran amb els valors de la llista inicial.

  x = [1, 4, 3, 5, 2]
  y = sorted (x) (y tindrà ara [1, 2, 3, 4, 5] mentre que x seguirà tenint els valors [1, 4, 3, 5, 2])

  La funció sorted ( ) també pot tenir un paràmetre sort (reverse= True) per ordenar de gran a petit.
  Els elements de la llista han de ser del mateix tipus, si volem ordenar una llista de cadenes ho farà per ordre alfabètic.

- Afegir (+) permet afegir una llista a una altra.

  x = [1, 2, 3, 4]
  y = [5, 6]
  z = x + y (z tindrà ara [1, 2, 3, 4, 5, 6])

- Replicar (* n) Replica els elements d’una llista n vegades.

  x = [1, 2, 3]
  y = x * 3 (x tindrà ara [1, 2, 3, 1, 2, 3, 1, 2, 3])

- Mínim (min) i màxim (max) retorna l'element més petit o més gran d'una llista.

```
x = [ 3, 2, 3, 4, 7, 4 ]
a = min ( x )             ( a serà un enter, i valdrà 2 )
b = max ( x )             ( b serà un enter, i valdrà 7 )
```

- Localitzar (index) retorna la posició d'un element en una llista.

```
x = [ 3, 6, 2, 5, 1, 7, 4, 8 ]
a = x.index (5)           ( a valdrà 3, és a la posició 3 començant per 0)
```

- Comptar (count) torna quants cops un element és dins d'una llista.

```
x = [ 1, 2, 5, 4, 5, 1, 3, 5, 7, 7 ]
a = x.count (5)           ( a valdrà 3 perquè 5 apareix tres cops )
                          ( retornarà 0 si no hi és)
```

- Existència (in) informa amb un booleà si un element és dins d'una llista.

```
x = [ 1, 2, 5, 4, 5, 1, 3, 5, 7, 7 ]
a = 8 in x
b = 4 in x                ( a valdrà False, b valdrà True)
```

- Suma (sum) suma els elements d'una llista si són numèrics.

```
x = [ 1, 2, 5, 4, 5, 1, 3, 5, 7, 7 ]
a = sum ( x )
                          ( a valdrà 40)
```

- Llargària (len) compta el número d'elements d'una llista.

```
x = [ 1, 2, 5, 4, 5, 1, 3, 5, 7, 7 ]
a = len ( x )
                          ( a valdrà 10)
```

Exercici 1.4.4.2: Donades les següents llistes:

n1 = ['Juan', 'Enric', 'Sarai']
n2 = ['Ahmed', 'Laura']
n3 = ['Liu', 'Antonio', 'Maria', 'Sarai']

Generar una única llista noms[ ] amb els noms de n1, n2 i n3 ordenats alfabèticament, fent servir la funció de sumar llistes.

Comprovar si 'Lluís' és a la llista total.
Comprovar si 'Sarai' hi és més d'una vegada.

1.5 Entrada i sortida de dades, interacció amb l'usuari.

1.5.1 La funció print

El llenguatge Python disposa de la funció **print ( )** que mostrarà per pantalla allò que li passem com a paràmetre.

print (" Hola! ")

(mostrarà Hola! a la pantalla).

La funció print accepta com a paràmetre qualsevol dels tipus de Python, veiem aquest exemple:

```
x=12

llista=[1,2,3,4,5]

s="Patata"

LN=["Moha","Ke-hon","Pere","Jana","Alba","Ian"]

print ("Proves:")
print (3)
print (x)
print (llista)
print (s)
print (LN)
```

On mostra una cadena, un nombre, una variable entera, una llista d'enters, una variable de tipus cadena i una llista de cadenes:

```
Proves:
3
12
[1, 2, 3, 4, 5]
Patata
['Moha', 'Ke-hon', 'Pere', 'Jana', 'Alba', 'Ian']
```

La funció **print** ens permet passar més d'un paràmetre separat per comes, això és útil per decorar la impressió de variables:

```
print ("La llista és:",llista)
print ("La paraula és:",s)
print ("El nombre és:",x)
```

```
La llista és: [1, 2, 3, 4, 5]
La paraula és: Patata
El nombre és: 12
```

Si ens cal, podem definir un separador entre els elements del print amb la paraula 'sep', veiem un exemple:

```
a=2
b=4
c=6

print ( a, b, c)
print ( a, b, c, sep=';')
```

Obtenim: 2 4 6
2;4;6

D'aquesta manera podem substituir el separador per defecte (un espai) pel que nosaltres vulguem, en aquest cas un ';'.

Per defecte Python després de cada **print** acaba la línia d'impressió. En cas de que ho vulguem canviar podem usar la indicació 'end' i assignar-li un nou valor. Això vol dir que el terme 'end' per defecte és un caràcter de línia nova (' \n '). Veiem un exemple:

```
a=2
b=4
c=6

s="Patata"


print (a,b,c)
print (s)

print (a,b,c,end='---')
print (s)
```

Obtenim: 2 4 6
Patata
2 4 6---Patata

El primer parell de prints pinta 2 4 6 i a la línia següent la paraula Patata. A la segona parella, al modificar l'element 'end' per '---' pinta els dos a la mateixa línia.

Exercici 1.5.1.1: Escriure amb un sol print la frase "Python és divertit !!" deixant dues línies en blanc davant i dues darrere.

1.5.2 La funció input

Així com print permet mostrar coses a la pantalla, la funció **input ( )** ens permetrà agafar dades escrites amb el teclat.
El paràmetre que es passa a la funció input ( ) és un text que es pintarà a la pantalla abans de l'entrada.
Veiem-ne un exemple:

```
n = input ("Entra un nombre: ")
print ("Has entrat:",n)
```

Obtenim: Entra un nombre: 89
Has entrat: 89

La variable n tindrà el valor ''89''.

A l'utilitzar la funció **input** cal tenir en compte les següents consideracions:

- la funció és **bloquejant**. Fins que l'usuari no entri un valor i premi la tecla 'Enter' el programa quedarà en aquesta línia de codi.
- sempre torna una variable de tipus **cadena** (string). A l'exemple, no ha tornat el nombre 89 sinó que n tindrà una cadena amb dos caràcters: 8 i 9.

Haurem doncs de convertir el que ens retorna **input** al format que desitgem. Això s'anomena 'typecasting' i es fa utilitzant el nom del tipus al qual volem convertir la variable, veiem a l'exemple:

```
n = input ("Entra un nombre: ")
print (type(n))
x=int(n)
print (type(x))
print ("Has entrat:",x)
```

Obtenim: Entra un nombre: 89
<class 'str'>
<class 'int'>
Has entrat: 89

Per tant la variable n era de tipus string, i la variable x ja és de tipus enter, gràcies a la conversió: x=int (n)

Aquests són algunes de les **conversions de tipus** que podem fer en Python:

- a = int (s) Convertirà s (cadena) a nombre enter. Si s no correspon a un enter, per exemple s="hola", tornarà un error.

- a = int (f) Convertirà f (float, nombre real) a enter. Atenció: ho fa truncant!! per tant si f=23.99, a valdrà 29.

- b = float (s) Convertirà s (cadena) al nombre real més proper. Si s no correspon a un nombre, per exemple s=”hola”, tornarà un error.

- b = float (a) Si a és un enter, b serà el real equivalent (ex. a = 3 -> b = 3.0)

- c = str (a) Convertirà el nombre a, a la cadena de caràcters que el representa.

Veiem algun exemple de funcionament:

```
s1 = "45.4"
s2 = "Estrella"

x= float(s1)
print (x)

y= int(float(s1))
print (y)

z= float(s2)
print (z)
```

Obtenint:

45.4
45
ValueError: could not convert string to float: 'Estrella'

Tal com esperàvem.

Exercici 1.5.2.1: Demanar un nombre enter per pantalla i pintar el nombre i el seu doble.

Exercici 1.5.2.2: Demanar dos nombres enters per pantalla i pintar els nombres i la seva suma.

1.6 Treball amb fitxers. Guardar les dades

Fins ara hem vist com avaluar expressions amb Python, que ens mostren el resultat d'alguna operació com per exemple a>b, i com utilitzar print i input per tenir interacció amb els usuaris del nostre programa.
Hi haurà vegades però que voldrem anar més enllà i utilitzar fitxers. Algunes de les avantatges d'utilitzar fitxers són:

- podrem llegir dades originades per altres programes, baixades d'internet, provinents de sensors, etc.
- podrem guardar les dades o sortides del nostre programa per utilitzar-los amb altres eines, enviar-los a algú, imprimir-los tenir una bitàcora de funcionament, etc.
- podrem interrompre l'execució del nostre programa i que pugui continuar més endavant per allà on anava. Haurem de deixar el progrés del programa guardat en fitxers.

Fitxers en Python

El llenguatge Python disposa de comandes per crear, obrir, escriure i llegir fitxers. Com en tots els llenguatges de programació, aquestes comandes reben suport del sistema operatiu que tinguem a sota (Linux, Windows, macOS, Android, etc.) per tant els llocs on els nostres fitxers aniran a parar, estaran sota l'estructura del sistema operatiu.

1.6.1 La funció open ( )

Open ( ) és una funció implementada de Python que ens permetrà obrir un fitxer. És una funció que ens demana dos paràmetres: el nom del fitxer i què volem fer amb el fitxer. Anem a veure com funciona.

f = open (nom_fitxer, mode)

La funció open ens tornarà una variable especial (aquí anomenada f, però pot tenir el nom que vulguem), que ens permetrà utilitzar el fitxer.

El nom de fitxer serà una cadena on especificarem el nom que volem donar al fitxer:

Exemples:

f = open (“ Prova.txt “)

Obrirà el fitxer anomenat Prova.txt a la carpeta actual de treball.

f = open (“ C:\usuari\dades\Prova.txt”)

Obrirà el fitxer Prova.txt a la carpeta especificada, estil Windows.

f = open (“ ./data/Prova.txt”)

Obrirà el fitxer Prova.txt a la carpeta especificada, estil Linux.

- Quan especifiquem una carpeta, si aquesta no existeix, open fallarà.

- Normalment el nom consta de dues parts, separades per un punt (.) la primera part és el nom del fitxer en si mateix. La segona (a l'exemple txt) es diu extensió i indica al sistema operatiu quina mena de fitxer és (text, imatge, executable, etc.)

A més del nom, indicarem a la funció **open** un **mode** de funcionament, és a dir, què volem fer amb el fitxer. Els modes possibles, que es passaran com una cadena, són:

- r lectura (reading), és el mode per defecte. Volem llegir el fitxer.
- w escriptura (writing). Si no existeix el fitxer, el crea. Si ja existeix el sobreescriurà i destruirà el que hi hagi.
- a afegir (append). Si no existeix el crea. Si existeix manté el contingut i escriurà al final del fitxer (només escriurà).
- + actualitzar (update). Obre el fitxer per lectura i escriptura, podrem llegir el que hi ha i afegir coses noves.

- t texte (text). El fitxer estarà en mode text, és a dir amb caràcters alfanumèrics que també podrem veure amb qualsevol editor del nostre sistema operatiu (per exemple LibreOffice). És el mode per defecte.
- b binari (binary). El fitxer estarà en mode binari, 1 i 0 directament sense cap codificació especial.

- x exclusiu (exclusive). Obrirem el fitxer de manera exclusiva. Mentre hi estiguem treballant, cap altre programa del sistema operatiu el podrà utilitzar.

Aquests paràmetres de mode es poden combinar. Veiem alguns exemples:

f = open (" Dades_Classe.txt" , "wt")

Crearem un fitxer anomenat "Dades_Classe.txt" a la carpeta actual, per escriure en mode text. Si hi ha existia, es perdrà la informació.

f = open (" Notes_Mates.txt" , "rt")

Obrirem, si existeix, un fitxer anomenat "Notes_Mates.txt" de la carpeta actual, on només podrem fer lectures.

Recordem que el sistema operatiu és qui s'encarrega de donar suport als fitxer. Si intentem obrir un fitxer pel que no tenim permisos, a una carpeta que no existeix, que no és del nostre usuari, etc. la funció **open** fallarà i rebrem un missatge d'error.

1.6.2 La funció close ( )

Quan acabem acabat de treballar amb el fitxer: llegir-lo, escriure'l o afegir-hi contingut, el tancarem per informar al sistema operatiu que està disponibile per altres programes. **Sempre** cal tancar un fitxer que hem obert, sinó les dades es poden perdre. Per tancar el fitxer és tan senzill com cridar la funció **close ( )** associada al manegador de fitxers que ens ha tornat open ( ).

Exemple:

```
f = open ( " Dades_temperatura.txt", "wt")
# operacions de recollir i escriure dades al fitxer
f. close ( )
```

Cal assegurar-se de tancar els fitxers. Si el nostre programa es penja durant la seva execució, pot deixar fitxers oberts amb risc de perdre la informació.

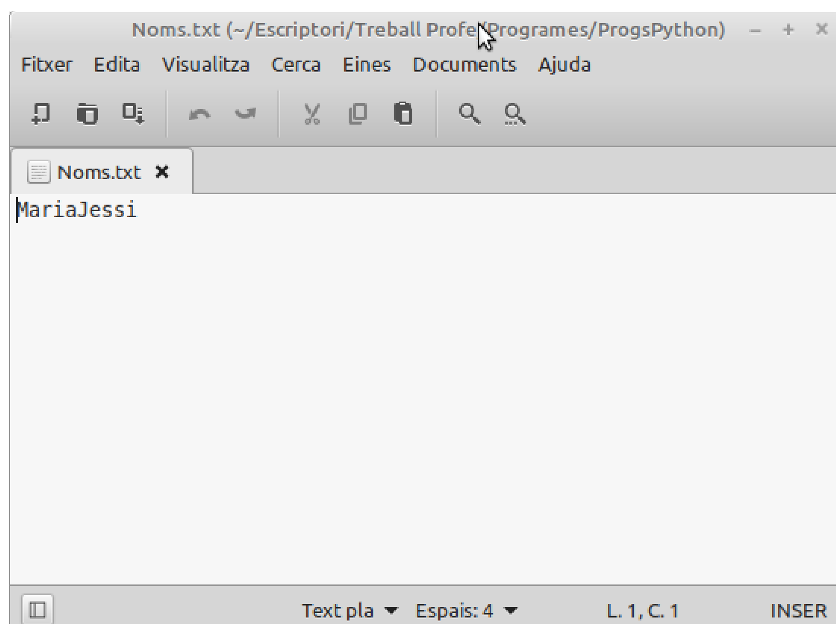
1.6.3 La funció write ( )

Aquesta és la funció que ens permetrà escriure dades al nostre fitxer. Com a la funció close, write està associat al manegador de fitxers que ens ha tornat open. Li passarem un paràmetre que seran les dades que aniran al fitxer. Es pot cridar diverses vegades.

Exemple.

```
f = open ( " Noms.txt", "wt")
f.write ( "Maria" )
f.write ( "Jessi" )
f. close ( )
```

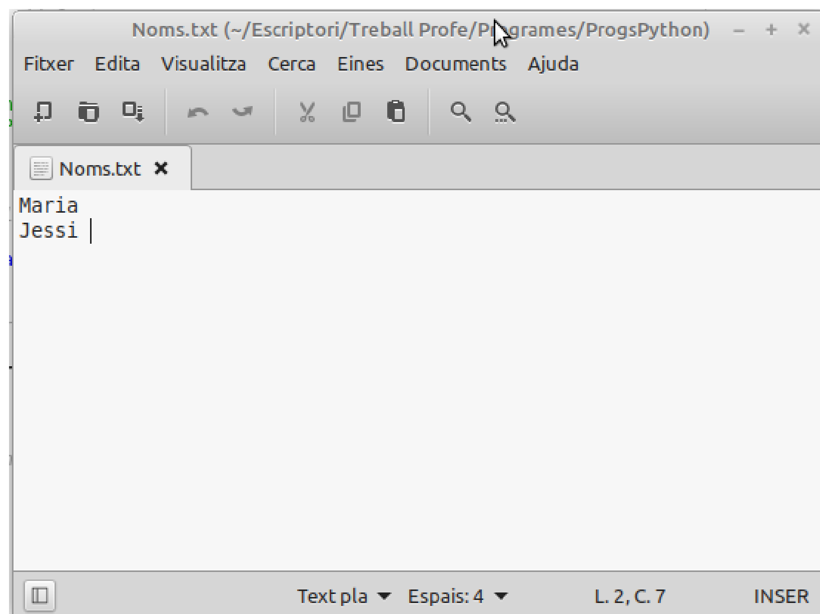
I veiem-ne el resultat:



Obviament no és el que esperàvem. Com hem vist amb la funció **print( )** a l'apartat anterior, existeix un caràcter '\n' que ens permetrà forçar un salt de línia.
Anem a provar-lo:

```
f = open ( " Noms.txt", "wt")
f.write ( "Maria \n" )
f.write ( "Jessi \n" )
f. close ( )
```

Obtenim:



Molt millor així, ja que anirem separant les dades que escrivim al fitxer.

1.6.4 Les funcions readline ( ) i read( )

La funció **readline ( )** llegeix directament una línia del fitxer i ens la torna com a cadena de caràcters (string), per tant és ideal com a complement del codi que acabem de veure. Si tenim un fitxer amb dos noms de persones com l'anterior, un programa com aquest:

```
f = open ( "Noms.txt", "rt")
s1 = f.readline ( )
s2 = f.readline ( )
print ( "El primer nom és: ",s1)
print ( "El segon nom és: ",s2)
f. close ( )
```

Mostrarà directament a pantalla:

```
El primer nom és: Maria
El segon nom és: Jessi
```

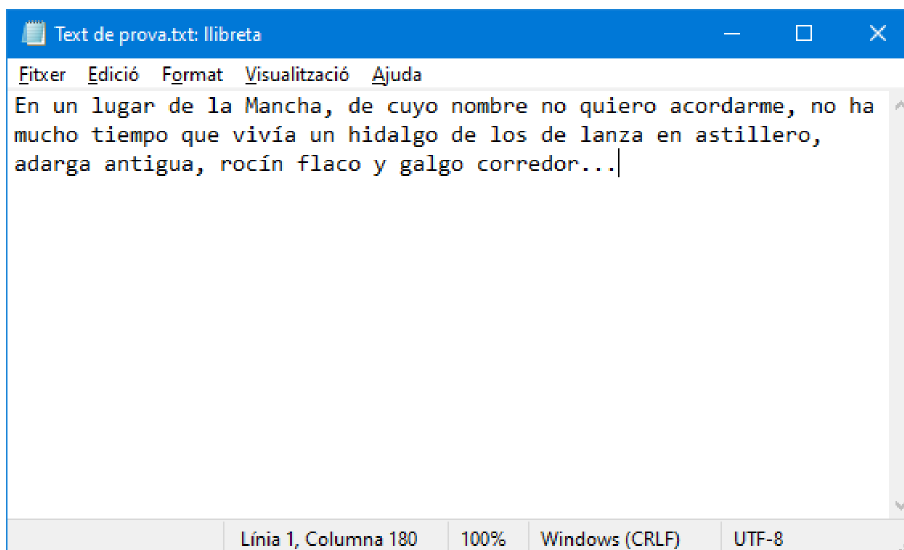
D'aquesta forma ja tenim un mecanisme per guardar i recuperar informació amb els nostres programes.

Exercici 1.6.4.1: Fer un programa que escrigui en un fitxer el nom d'unes quantes persones de la vostra classe. Visualitzar-lo amb un editor de text qualsevol (per exemple, notepad).

Exercici 1.6.4.2: Modificar des del notepad el contingut del fitxer anterior (per exemple, afegint un nom nou) i fer un programa que el carregui, afegint els noms a una llista de Python. Mostrar la llista a pantalla.

Qüestions. Ha funcionat bé? com controlem el nombre d'elements que té la llista? discutiu sobre aquest punt amb els companys.

La funció **read (n)** permet llegir n bytes d'un fitxer. Per fitxers de texte, podem assumir que cada caràcter ocupa un byte. Si tenim un fitxer com aquest:



I provem el següent programa:

```
f = open ( "Text de prova.txt", "rt")
s = f.read (24)
print ( s )
f. close ( )
```

Tindrem a la sortida:

En un lugar de la Mancha

La funció read ( ) normalment s'utilitza acompanyada de dues funcions més que ens ofereixen els fitxers i que ens ajuden a tenir control de la seva lectura:

- La funció: **f.tell ( )** que donat un fitxer f ens diu per quin caràcter anem.

- La funció: **f.seek (n)** que ens col·loca el punt de lectura del fitxer a la posició que li diguem nosaltres (n).

Veiem un exemple de com utilitzar tell( ) i seek ( ), aplicat al fitxer anterior. Utilitzem dues cadenes s1 i s2 per guardar text llegit del fitxer i una variable entera x per saber per on anem:

```
f = open ( " Text de prova.txt", "rt")
s1 = f.read (11)
print ( s )
x = f.tell ( )
f.seek ( x-5 )
s2 = f.read (11)
print (s1)
print (s2)
f. close ( )
```

El que obtindrem per pantalla són dues cadenes (dels dos print):

```
En un lugar
lugar de la
```

Veiem la repetició de la paraula "lugar" perquè seek ( ) ha tirat 5 posicions enere.

1.7 Funcions en Python.

El llenguatge de programació Python permet definir i cridar funcions, cosa que ens ajudarà a organitzar bé el codi. Per exemple, si hi ha un tros de codi que hem d'utilitzar moltes vegades, el millor és agrupar-lo dins una funció i cridar-la diverses vegades.

Les funcions es definiran sempre amb la comanda ' def ' seguida del nom de la funció, els paràmetres que rep i el símbol ' : '. Tot el codi dins de la funció estarà tabulat (és a dir, mogut uns quants caràcters cap a la dreta respecte la definició) i les funcions acabaran amb una comanda 'return ( )' on es pot posar allò que retorna la funció (si per exemple ha fet un càlcul).

Veiem-ne un exemple:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""


def suma (a , b):
    r = a + b
    return (r)

c = suma (12,3)

print (c)
```

Aquest codi ha creat una funció *suma* que rep dos paràmetres a i b per sumar-los i retornar el resultat. Si l'executem, veiem que s'escriu un 15, tal com ha de ser.

Atenció! les variables a, b i r de dins de la funció són locals a elles, i no afectarien a altres amb el mateix nom que pogués haver-hi fora de la funció.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""


def suma (a , b):
    r = a + b
    return (r)

a=7
b=12
r=13
c = suma (a,b)

print (c)
print (r)
```

Si executem aquest codi, veiem que c val 19 (la suma) i r segueix valent 13. perquè la r de la funció és local a ella.
Com hem dit, les funcions serveixen per agrupar un codi que utilitzarem moltes vegades; veiem aquest exemple:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""


def suma (a , b):
    r = a + b
    return (r)



a=7
a = suma (a,12)
a = suma (a,25)
a = suma (a,6)

print (a)
```

El resultat mostrat és 50, que serà 7 + 12 + 25 + 6.

A diferència d'altres llenguatges de programació, Python permet que una funció retorni diversos valors com a resultat. Veiem a l'exemple com una funció 'opera' torna el resultat de diferents operacions entre els paràmetres:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""

def opera (a , b):
    s = a + b
    r = a - b
    p = a * b
    q = a / b
    return (s,r,p,q)


a,b,c,d = opera (18,6)

print (a,b,c,d)
```

Els valors mostrats són: 24 12 108 3.0

Fixem-nos que a l'utilitzar l'operador ' / ' per la divisió, ens ha tornat un valor de tipus real o float. A l'apartat 1.3 hem vist que l'operador ' // ' forçaria l'operació amb enters.

Com hem vist, quan passem com a paràmetres a una funció tipus mutable, cal anar en compte perquè aquests **SI** que poden quedar alterats fora de la funció si dins d'ella es modifiquen.
Veiem un primer exemple on passem una llista com a paràmetre i només fem servir operadors immutables, per tant no tenim cap problema:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""

def valormig (a):
    n= len(a)
    s= sum(a)
    return (s/n)


l=[1,3,2,5,6,5]

print (valormig(l))
print (l)
```

La sortida és:

3.6666666666666665
[1, 3, 2, 5, 6, 5]

Per tant, veiem que s'ha calculat correctament el valor mig dels elements de la llista (hem dividit la suma dels elements pel seu nombre) i la llista no ha quedat modificada.
En canvi a l'exemple:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""

def valormig (a):
    a.sort()
    n= len(a)
    s= sum(a)
    return (s/n)


l=[1,3,2,5,6,5]

print (valormig(l))
print (l)
```

La sortida és:

```
3.6666666666666665
[1, 2, 3, 5, 5, 6]
```

Veiem que s’ha modificat el contingut de la llista. S’ha volgut posar èmfasi en aquest cas perquè és una font possible d’errors de programació.

1.8 Estructures bàsiques de programació en Python: bifurcacions, repeticions.

1.8.1 Bifurcacions el el codi

Els llenguatges de programació ofereixen una estructura per poder fer bifurcacions en l'execució del codi.

- S'avaluarà una condició (o una combinació de condicions) (and, or…)
- Es proposarà una o més instruccions a executar si la condició és certa
- Es proposarà (opcionalment) una o més instruccions a executar si la condició és falsa.

Python utilitza les paraules reservades **if** i **else** per l'estructura bàsica de bifurcació.

if condició :
instruccions
else:
instruccions

Veiem que la o les instruccions de després del 'if' o l''else' van un tabulador cap a la dreta. Els dos punts de després indiquen que comença el cas de la condició i el bloc tabulat indica el conjunt d'instruccions a executar.

Exemple, escriure si un nombre a és parell o senar:

```
if a%2==0:
    print("És parell")
else:
    print("És senar")
```

Per avaluar la condició hem fet servir la operació mòdul (%) que torna el reste de la divisió entera, en aquest cas, per 2.

Python permet també fer **bifurcacions amb múltiples casos**, afegint el terme **elif**.

```
if a<0:
    print("És negatiu")
elif a>0:
    print("És positiu")
else:
    print("És zero")
```

A l'exemple veiem l'avaluació de tres casos de la variable a, utilitzant un 'elif'.

Tot i que Python permet la utilització de tants elif com vulguem és desaconsellat per casos en que es dispara el nombre d'opcions, veiem un mal exemple:

```
if a==0:
    print("És zero")
elif a==1:
    print("És u")
elif a==2:
    print("És dos")
elif a==3:
    print("És tres")
else:
    print("És quatre")
```

Imaginem un cas com aquest on es volgués determinar un nombre entre 0 i 9 (l'exemple s'ha tallat a 4). En aquests casos és millor utilitzar el tipus **diccionari** vist anteriorment quan s'han presentat tipus bàsics de Python.

```
dic={0:"Zero",1:"U",2:"Dos",3:"Tres",4:"Quatre",5:"Cinc",
     6:"Sis",7:"Set",8:"Vuit",9:"Nou"}

print ("És " + dic[a])
```

Exercici 1.8.1.1: Fer un petit programa que demani un nombre i digui si és múltiple de 3.

Exercici 1.8.1.2: Fer un programa que, donada la següent llista de noms:

LN = [“Pepe”, “Javi”, “Joan”, “Fatu”, “Laia”, “Mar”]

Demani un nom nou i, si hi ha hi és, escrigui que ja hi és. Si no hi és, l'afegeixi a la llista.

1.8.2 Repeticions: while

L'estructura de programació amb **while** ens permetrà repetir un bloc de codi mentre es dongui una condició.

while condició :
instruccions

Com en el cas de l'if, pot haver una o diverses instruccions dins del bloc del ‘while’ que vindran agrupades també amb un tabulador (tot el bloc ha d'estar tabulat). Cal anar amb compte perquè si la condició sempre és certa, crearem un bucle infinit per tant caldrà pensar bé les condicions que s'escriuen.

Aquest exemple pintarà els nombres del 0 al 9 com a nombres:

```
i=0
while i<10:
    print(i)
    i=i+1
```

i aquest els pintarà en format text:

```
dic={0:"Zero",1:"U",2:"Dos",3:"Tres",4:"Quatre",5:"Cinc",
     6:"Sis",7:"Set",8:"Vuit",9:"Nou"}

i=0
while i<10:
    print (dic[i])
    i=i+1
```

En canvi aquest mal exemple:

```
dic={0:"Zero",1:"U",2:"Dos",3:"Tres",4:"Quatre",5:"Cinc",
     6:"Sis",7:"Set",8:"Vuit",9:"Nou"}

i=0
while i<10:
    print(i , "," , dic[i])
```

crea un bucle infinit perquè ens hem oblidat d'incrementar la variable 'i' i per tant mai deixa de ser més petita que 10.

El programa imprimeix per sempre: 0 , Zero

Es pot sortir d'un programa en execució prement les tecles **CTRL + C**.

La comanda **break** permet sortir d'un bucle en qualsevol moment:

```
i=0
while i<10:
    if i==6:
        break
    print(i)
    i=i+1
```

A l'exemple, el programa donarà com a sortida: 0 1 2 3 4 5
L'ús del 'break' es considera un mal hàbit de programació sempre trobarem una alternativa millor avaluant la condició del bucle adequadament (en aquest cas while i<6:).

La comanda **continue** salta el cas actual d'un bucle:

```
i=0
while i<10:
    i=i+1
    if i==6:
        continue
    print(i)
```

La sortida del programa serà ara: 1 2 3 4 5 7 8 9 10

Com veiem no ha imprès el cas de i=6 però ens ha obligat a canviar l'estructura del bucle i ara pinta de 1 a 10 !! perquè l'estructura anterior:

```
i=0
while i<10:
    if i==6:
        continue
    print(i)
    i=i+1
```

Crea un bucle infinit donat que 'continue' fa que no es pinti ni incrementi la i, per tant sempre val 6. Evitarem utilitzar 'continue' posant un if adequat:

if i != 6:
print (i)

igual que 'break' considerarem que 'continue' és un mal hàbit de programació.

Exercici 1.8.2.1: Fer un bucle amb while que pinti tots els números entre 0 i 25,

Exercici 1.8.2.2: Fer un bucle que demani un nombre a l'usuari i mentre no sigui 0, pinti el nombre entrat i el seu doble.

1.8.3 Repeticions: for

A banda de la instrucció **while** el llenguatge Python té una segona instrucció per definir blocs de codi i repetir la seva execució. Es tracta de la sentència **for**.

La sentència for, permetrà recórrer TOTS els elements d'una estructura iterable: una llista, una tupla, un conjunt, un diccionari, etc. La seva utilització és:

for element in tipus_iterable:
instruccions

Aquestes instruccions seran sobre tots els elements del tipus iterable, veiem que a diferència del *while*, no hi ha una condició de sortida. L'estructura *for* es fa servir pel que algorísmicament s'anomena fer un recorregut.

Exemple 1. Mostrar a pantalla tots els elements d'una llista

```
llista=["Primavera","Estiu","Tardor","Hivern"]

for el in llista:
    print(el)
```

La variable el serà un iterador que anirà prenent el valor de tots els elements de la llista i aquests s'aniran imprimint.

Exemple 2. Fer la intersecció entre dues llistes L1 i L2

```
L1=["Juan","Maria","Pere","Ian","Antònia"]
L2=["Moha","Ke-hon","Pere","Jana","Alba","Ian"]

for el in L1:
    if el in L2:
        print(el)
```

Aquest codi imprimirà: Pere
Ian

Com veiem la variable el prendrà tots el valors de L1 i un a un mirem si està a L2 i en cas afirmatiu, l'imprimim.

A vegades ens és necessari crear temporalment una sèrie numèrica per indexar alguna estructura o iterar-la amb un for. Python ens ofereix la funció **range(n)** que crea una variable del tipus **range** amb els elements de 0 fins a n-1.

Veiem el següent codi per veure com funciona el tipus **range**:

```
In [12]: x = range(12)

In [13]: print(x)
range(0, 12)

In [14]: type(x)
Out[14]: range

In [15]: for i in range(12):
    ...:     print (i)
    ...:
0
1
2
3
4
5
6
7
8
9
10
11
```

Exemple 3. Crear una llista amb els 10 primers nombres elevats al quadrat.

```
llista=[]

for i in range(1,11):
    llista.append(i**2)

print(llista)
```

La sortida del programa serà: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Hem creat una llista buida, a la que hem anat afegint elements. Els elements els hem tret dels nombres generats pel range que han anat de 1 a 10 (11-1) elevats al quadrat.

Observem aquest altre codi que fa exactament el mateix amb un *while*:

```
llista=[]

i=1
while i<=10:
    llista.append(i**2)
    i=i+1

print(llista)
```

La sortida del programa també serà: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Quin és millor? Si tenim una estructura de dades a recórrer usarem el *for*, però en aquest cas, en que l'hem creat expressament és més elegant i estalvia memòria fer-ho amb un *while* (range crea a memòria l'estructura dels 10 elements, que és prescindible!).

Exercici 1.8.3.1: Utilitzar **range** per generar i pintar tots els múltiples de 5 entre 0 i 1000.

Exercici 1.8.3.2: Agafar tots els nombres entre 0 i 1000 i ordenar-los en quatre llistes:

- múltiples de 2
- múltiples de 3
- múltiples de 5
- resta de nombres

Pintar aquestes 4 llistes per pantalla.

Exercici 1.8.3.3: Aquest exercici consta de dues fases:

- Fer un programa que demani noms de persones i els vagi afegint a una llista inicialment buida fins que el nom sigui "sortir".
- Escriure tots els noms de persones amb l'afegit "és a la llista".

Exemple:

Llista = ["Javi","Joana"]

Escriure:

Javi és a la llista.
Joana és a la llista.

1.9 Comentaris en Python.

El llenguatge de programació Python permet dues maneres de posar un comentari en el codi. És important afegir comentaris al codi, i que aquests siguin clars, per tal de que la seva lectura i comprensió sigui fàcil.
El primer mètode per posar comentaris en Python és amb el símbol ‘ # ‘ que indica que, en aquella línia, tot el que ve a continuació és un comentari.

Exemple 1:

```
num = 12  # creem una variable de tipus enter, anomenada num
```

El segon mètode és utilitzant el símbol ‘ “”” ‘ o ‘ ‘’’ ‘ (triple cometa o triple apòstrof) al principi i final d’un bloc de texte. Tot el que quedi enmig serà considerat un comentari.

Exemple de combinació de comentaris de bloc i de línia:

```
'''

Aquest codi crearà dues variables num1 i num2
de tipus enter i en calcularà la suma, que es
guardarà a res

'''

num1 = 23
num2 = 36

res = num1 + num2 # a res queda el resultat.
```

Molts entorns de programació de Python, quan creem un fitxer nou per treballar, ens l’inicien amb uns comentaris:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 17:33:06 2022

@author: professor
"""
```

La primera línia, indica el camí (dins de les carpetes de l’ordinador) d’on es troba l’intèrpret de Python (això és típic dels sistemes Linux).

La segona línia indica que la codificació dels caràcters és utf-8, això ens permet utilitzar accents i caràcters especials (com ç o ñ) dins del fitxer del nostre programa.

Finalment el bloc amb la data, hora i autor, és insertat automàticament i ens ajudarà a tenir ordre en els nostres programes i recordar quan va ser començat i per qui.

2. Bones pràctiques de programació. Algorísmica

Després de la presentació del funcionament bàsic de Python i la realització de petits exercicis, ens podem sentit temptats de llença-nos a programar sense més. El fet de que Python permeti anar avaluant línia a línia, declarant variables sobre la marxa, etc.facilita que els codis generats tendeixin a ser poc estructurats. Això fa que la comprensió del codi per altres persones (o per nosaltres mateixos més endavant), la detecció d'errors no pensats en el primer moment de programació o la solució d'aquests errors sigui realment complicada.

Abans de posar-nos a fer un programa, val la pena fer-nos una sèries de preguntes i un petit anàlisi del que anem a fer:

- quin problema anem a resoldre?
- quines dades d'entrada tindrà el nostre programa?
- podem definir totes les dades possibles a l'entrada?
- hi ha interacció amb usuaris?
- utilitzem dades guardades a fitxers?
- què s'ha de mostrar a pantalla?
- com acaba el programa?
- quin és el resultat de l'execució del programa?
- quina relació hi ha entre l'entrada i la sortida del programa? la podem representar d'alguna manera?
- la solució al problema que volem resoldre, quina estructura té? és una seqüència? va prenent decisions? repeteix operacions?
- si la solució (nombre de passos) és molt llarga, la podem dividir en blocs més petits?
- si hi ha conjunts d'operacions que es repeteixen, les podem agrupar en blocs que anirem cridant (funcions)?
- Les dades que utilitzarà el programa, tenen alguna estructura concreta? són una llista de valors? estan ordenades?
- …

La resposta a aquestes i altres qüestions ens ajudarà a estructurar-nos mentalment la forma en que començarem a programari millorar les possibilitats d'èxit!

2.1 Algorísmia

Anomenem algorisme una seqüència precisa d'operacions o pasos que resolen un problema en un temps determinat (que ha de ser finit). L'algorisme és el mètode i és independent del llenguatge de programació o de la màquina que l'executarà.

Exemple. Volem obrir la porta de casa nostra

- Agafem el clauer
- Busquem la clau de la porta de casa (sabem la forma o el color)
- Posem aquesta clau al pany
- Girem en sentit antihorari una o dues voltes (recordem com va el pany)
- Empenyem o estirem la porta segons com estigui muntada
- Treiem la clau del pany

Fixem-nos que tenim una certa quantitat de coneixements i **condicions d'abans** de començar a obrir:

{ tenim el clauer **i**
la clau de casa és al clauer **i**
sabem quina és la clau (per forma, color, és la única…) **i**
sabem que el pany és d'una volta o de dues, **o**
sabem que normalment el tanquem amb una o dues voltes, **i**
sabem que la porta obre cap a dins o cap a fora
}

Els coneixements que hem de tenir a priori, estan guardats a la nostra **memòria**, en el cas d'un ordinador caldrà definir on de la seva memòria estarà.

Podem definir una **condició d'èxit**, que indica que ha acabat l'operació:

{ la porta està oberta **i**
la clau està treta
}

Veiem també que les operacions són simples i es poden estructurar en format de seqüència, excepte la de de buscar que és una mica més complicada.

Agafem
V
Busquem
V
Posem
V
Girem (x1 o x2)
V
Empenyem
V
Treiem

Per tant sembla una bona idea que la solució sigui una seqüència d'operacions senzilles, i que al pas "Busquem la clau de la porta de casa" li dediquem una atenció especial.

Com es fa la cerca de la clau?

Procediment de **Busquem**:

Busquem té unes **condicions d'abans** que s'han de complir:

{ tenim el clauer **i**
la clau de casa és al clauer **i**
sabem quina és la clau (per forma, color, és la única…)
}

També té una **condició d'èxit**:
{
la clau de la porta és a la nostra ma
}

De quines operacions consta la cerca de la clau?

- Despleguem el clauer per poder veure totes les claus
- Mirem la clau de més a l'esquerra.
- Si no és la de casa, mirem la següent a la seva dreta. Si si que ho és, l'agafem amb la ma
- repetim el pas anterior fins trobar-la.

Veiem que la condició lògica de "la clau de casa és al clauer" ens garanteix que el procediment on anem repetint un pas acabarà. Si no, caldria escriure quelcom com:

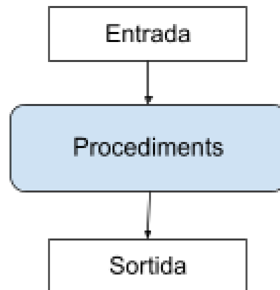
- repetim el pas anterior fins trobar-la o fins que arribem a la clau de més a la dreta.

La correcta definició dels passos i de les condicions d'entrada i de sortida, quan es creen subprocediments com el de "Busquem", l'especificació de què se li dóna i que s'espera que ens faci, etc. garantirà que l'algorisme i el programa que se'n pugui derivar, siguin correctes.

La definició de procediments i subprocediments, permetrà també que es puguin dividir les tasques entre diferents persones, per exemple, un programador implementa la funció de buscar la clau mentre un altre fa la resta.

És necessari que quedi clar què es dóna i què es retorna de cada pas per garantir un ensamblatge precís de totes les parts.

En general, cada algorisme o part d'algorisme es pot caracteritzar simplement amb (1) l'entrada: dades / coneixements necessaris, (2) els procediments: conjunt de passos per arribar a la solució desitjada i (3) la sortida: resultats esperats de l'algorisme.



Podem enllaçar dos algorismes o parts d'algorisme si les dades necessàries a l'entrada del pas (N) estan dins de la sortida del pas anterior (N-1).

Els procediments que podem fer en algorísmia es classifiquen formalment en tres categories:

2.1.1 Seqüencial

En aquest cas tindrem una composició seqüencial d'instruccions, una darrera l'altra. Les condicions d'abans de l'execució del nostre algorisme s'anomenen formalment *Precondicions: P*. Les condicions que s'han de complir a l'acabar l'algorisme o tros d'algorisme s'anomenen formalment *Postcondicions: Q*. Les accions de dins de la seqüència van portant de les Precondicions a les Postcondicions.

Formalment es pot caracteritzar així:

{P}	(precondicions)
S1	
S2	(accions)
…	
SN	
{Q}	(postcondicions)

Exemple: donat un nombre, volem calcular el seu doble més tres.

```
{ Precondició P: x és un nombre real }

y = 2 * x
y = y + 3

{Postcondicions Q: y és un nombre real, y = 2x + 3 }
```

La composició seqüencial és la més senzilla d'especificar i de programar.

2.1.2 Alternativa

Per evitar que els nostres programes siguin una única seqüència d'instruccions, ens cal definir alternatives, segons alguna expressió lògica agafarem un camí o un altre. Quan definim una composició alternativa d'operacions, es triarà executar unes instruccions o unes altres depenent de certes condicions B_1, B_2, B_3… B_N. La notació que es farà servir és aquesta:

{P} (precondicions)
si B_1 -> S1
si B_2 -> S2
si B_3 -> S3
…
si B_N -> SN
{Q} (postcondicions)

Com a mínim una de les expressions Booleanes Bi ha de ser certa per executar les instruccions Si. Per tant, a nivell lògic, la seqüència:

{P $\wedge$ B_i}
Si
{Q}

Ha de ser correcte per totes les expressions B_i.

D'altra banda, si (B_1 v B_2 v B_3 v … v B_N) no és cert per {P} pot ser que ens estiguem deixant alguns casos.

Exemple: donat un nombre, volem calcular el seu valor absolut.

{ Precondició P: x és un nombre real }
si x>0 -> y = x
si x<0 -> y= -x
{Postcondicions Q: y és un nombre real, y = | x | }

Podem verificar que pels dos casos es compleix la postcondició.

(1) { P: x és un nombre real $\wedge$ x>0 }
y = x
{Q: y és un nombre real, y = | x | }

(2) { P: x és un nombre real $\wedge$ x<0 }
y = -x
{Q: y és un nombre real, y = | x | }

En canvi veiem que: (x>0) v (x<0) no és cert per tot P: x és un nombre real, donat que ens deixem el cas x=0. Seria millor fer la primera condició: x >=0.

2.1.3 Iterativa

La darrera eina que ens cal és la repetició de segments de codi, ho podem fer seqüencialment repetint una operació N vegades, però altres vegades ens farà falta repetir unes operacions mentre es dongui una condició lògica.
Una manera de formalitzar les iteracions és aquesta:

```
{P}
mentre B fer
        S
fimentre
{Q}
```

Mentre es dongui que l'expressió booleana B és certa, s'anirà executant la seqüència S d'operacions. Aquesta seqüència i l'acabament de la condició B, hauran de portar la Precondició P a la Postcondició Q.

Exemple. Volem fer el producte de tots els nombres entre 1 i N (factorial).

```
{ Precondicions  P: N és un nombre natural, N>0 ; F = 1; i = 1   }
mentre ( i <= N )
        F = F * i
        i = i + 1
fimentre
{Postcondició Q: F = N !  }
```

La condició booleana de manteniment dins del bucle, és, B: i<=N

Si veiem el seu funcionament per N = 5, veiem com funciona el bucle:

Volta	i (a l'entrar)	F (a l'entrar)	i (al sortir)	F (al sortir)	N
1	1	1	2	1	5
2	2	1	3	2	5
3	3	2	4	6	5
4	4	6	5	24	5
5	5	24	6	120	5
6	6	Al valdre i=6 acaba i surt del mentre			

Les condicions que s'han de complir perquè ens mantinguem dins la iteració s'anomenen **invariant del bucle**. En aquest cas es compleixen dues condicions:

$$I: \ (1 \le i \le N+1) \quad i \quad F = (i-1)!$$

Quan al final es deixa de complir la condició de permanència al bucle B perquè i = N+1,

l'estat en que queda l'invariant del bucle I: (i = N+1) i (F = (i - 1) !

Fa que es compleixi la postcondició Q: F = N !

La combinació de les estructures seqüencials, alternatives i iteratives podran generar qualsevol algorisme. La comprovació mitjançant lògica, dels estats inicials i finals de cada bloc de codi i que formalment les operacions permetin anar d'uns als altres contemplant tots els casos permeten fer una programació matemàticament consistent i amb garanties de bon funcionament.
Aquestes tècniques de suport lògic a la programació s'utilitzen especialment en sistemes crítics: centrals nuclears, satèl·lits, avions, cotxes, medicina on la fallada d'un programa posaria en risc a persones. Malgrat que a la programació més comercial no es fa una verificació formal dels codis, és bo conèixer-la per ajudar-nos a estructurar la ment i la manera de plantejar els programes.

2.2 Estructura del codi

2.2.1 Translació d'algorisme a codi

En general, quan tenim una seqüència d'operacions que volem portar a codi, el procés és bastant senzill. Segons el llenguatge de programació (de més alt nivell o de molt baix nivell), cada operació que volem fer acabarà sent una instrucció. Aquestes instruccions aniran una darrera l'altra.
Seguint amb la idea de que cada algorisme o porció d'algorisme (que pot ser una funció, per exemple), tindrà unes dades d'entrada, les instruccions i la sortida (que pot ser deixada a memòria, pintada a pantalla o retornada d'una funció), intentarem ser ordenats:

- ubicar totes les variables al principi,
- després el codi i
- l'entrega dels resultats al final

Exemple en Python: mostrar el valor de la funció $f(x) = x^3 + x^2 - x + 4$ en el punt x=2.0

```
x=2.0                       # definim variable x i li donem valor d'entrada
resultat=0.0                # definim variable resultat pel final

resultat = x**3             # fem les operacions
resultat = resultat + x**2  # fem les operacions
resultat = resultat -x      # fem les operacions
resultat = resultat + 4     # fem les operacions

print ( resultat)           # mostrem el resultat
```

Quan hem de portar a codi una estructura alternativa, farem servir a la majoria de llenguatges la sentència **if**. Segons com utilitzem la combinació de **if - else - elif** s'avaluaran totes les alternatives una darrera l'altra o en el moment en que es compleixi una condició sortirem de l'avaluació dels casos.

Exemple d'alternatives en Python:

```
if (condició 1):
    S1
if (condició 2):
    S2
if (condició 3):
    S3
```

Amb aquesta estructura es poden executar **diverses** o **totes** les seqüències S1,S2, S3 segons si hi ha més d'una condició certa.

```
if (condició 1):
    S1
elif (condició 2):
    S2
elif (condició 3):
    S3
```

Amb aquesta estructura només es pot executar **una** de les seqüències, serà la primera que trobi la seva condició certa.

Les iteracions es poden traslladar amb la sentència **while** dels llenguatges de programació. Tot i que amb while es pot implementar qualsevol estructura d'iteració, alguns llenguatges, com Python, implementen dues instruccions diferents per generar bucles: **while** i **for**.

2.2.2 Cerques i recorreguts

En algorísmica hi ha dues estructures iteratives que es fan servir extensivament. L'estructura de cerca i l'estructura de recorregut.

L'estructura habitual de **cerca d'un element** és la següent:

```
while ( not final ):
        element = obtenir_següent ( )
        final = processar ( element )
resultat = element
```

quan detectem que hem trobat l'element, posem final a cert per sortir del bucle i el resultat és l'element trobat.
En cas de que no estigui garantida la presència de l'element és:

```
while ( not final ) and (queden_elements) :
        element = obtenir_següent ( )
        final = processar ( element )
if (final):
        resultat = element
else:
        resultat = ERROR
```

D'aquesta manera evitem un possible bucle infinit i al sortir indiquem si s'ha trobat o no.
Exemple en Python: Donada una llista de nombres, mirar si hi ha un nombre major que 50.

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

n = len (Llista)
i=0
trobat = False

while ( i < n ) and (trobat==False) :
        trobat  = ( Llista [ i ] > 50)
        i = i + 1

if (trobat):
        print (" si hi ha un nombre major de 50")
```

El programa en aquest cas sortirà quan amb i=5, **trobi** l'element 67 de la llista, que compleix la condició. Un cop trobat, no cal seguir cercant, perquè només demanaven si hi ha UN element major de 50, no quants.

Una altra possibilitat és quan sabem a priori, que hem de passar per tots el elements d'una estructura o d'una sèrie de valors. En aquest cas, que la única condició és passar per tots, s'anomena un **recorregut**. Com s'ha dit **while** serveix per implementar qualsevol estructura d'iteració:

Exemple en Python: Donada una llista de nombres, mirar quants n'hi ha majors que 50.

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

n = len (Llista)
i=0
nombre_majors = 0

while ( i < n ) :
    if ( Llista [ i ] > 50):
        nombre_majors = nombre_majors + 1
    i = i + 1

print (" Hi ha ", nombre_majors, " més grans de 50")
```

Veiem que és un recorregut perquè cal passar per tots els elements. Tenim una variable pensada: nombre_majors per fer de comptador i l'anem incrementant dins de la iteració.

Veiem ara una solució alternativa:

Exemple en Python: Donada una llista de nombres, mirar quants n'hi ha majors que 50 utilitzant la comanda for..

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

nombre_majors = 0

for element in Llista :
    if ( element > 50):
        nombre_majors = nombre_majors + 1

print (" Hi ha ", nombre_majors, " més grans de 50")
```

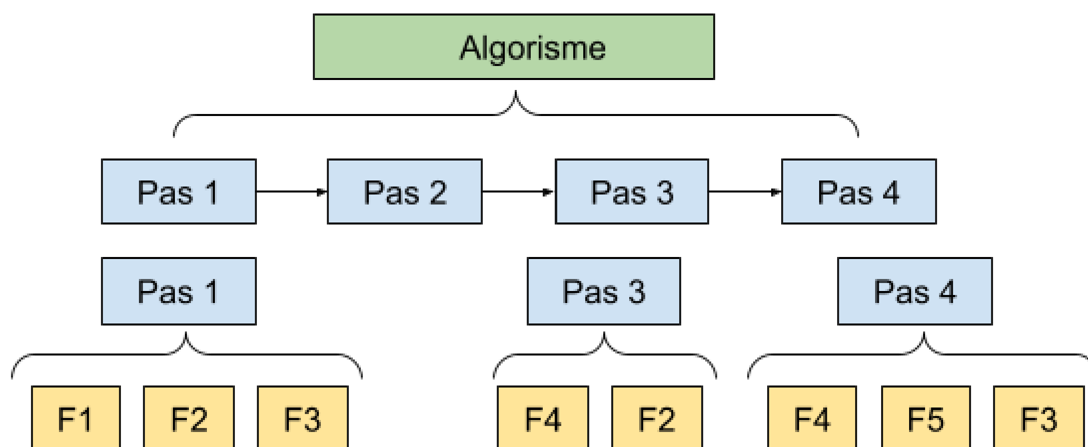
Veiem que el codi queda una mica més senzill perquè l'estructura del for, al aplicar-lo sobre una estructura iterable, permet prescindir de l'índex i (de fet ho farà ell internament).

En general, quan haguem de fer un recorregut sobre una estructura de dades preexistent i iterable, com ara una llista o un conjunt, és òptim fer servir la comanda **for**. Per fer cerques o seguir una sèrie de nombres (per exemple comptar de 0 a 60) és millor utilitzar el **while**.

2.2.3 Disseny descendent. Jerarquia i seqüenciació

Hem vist als apartats anteriors com es pensen i dissenyen algorismes i com es traslladen a codi les seqüències, alternatives o iteracions. Què passa quan el programa que hem de fer és molt gran? La idea és començar pensant en accions d'alt nivell tipus: demanar dades, posar a llista, consultar llista, etc. i després per cada una d'aquestes accions veure si internament s'ha de dividir en altres tasques.

Així doncs, intentarem pensar la solució al problema globalment i després, poc a poc, ens anirem preocupant de com portar cada part de la solució a codi. Aquesta estratègia de dalt cap a baix s'anomena **disseny descendent**.



Exemple: El nostre algorisme es pot dividir en quatre passos senzills (seqüenciació), alguns d'aquests passos (1, 3 i 4) encara són prou complexes com per dividir-los en funcionalitats més petites (jerarquització), a més quan fem aquest disseny podem veure que hi ha algunes d'aquestes funcionalitats que es poden utilitzar en diferents llocs, llavors té sentit crear unes funcions per reaprofitar el codi, que anirem cridant a diferents llocs.

2.2.4 Recursivitat

Una darrera tècnica de programació que permet implementar certes solucions algorísmiques és la recursivitat. Una solució recursiva és aquella on s'utilitza la seva pròpia definició, normalment, una forma més simple d'ella mateixa.

Veiem un exemple habitual de recursivitat:

El factorial d'un nombre N, escrit N! és N x (N-1) x (N-2) x … x 1, el factorial de 4 és 4! = 4x3x2x1

Veiem doncs que si definim f com la funció que calcula el factorial d'un nombre, la podem especificar així:

f (1) = 1
f (N) = N x f (N-1)

Per tant diem que el factorial d'un nombre és ell mateix, multiplicat pel factorial de ell menys u. Tenim el cas de que el factorial de 1 és 1, cosa que ens permetrà acabar el recorregut descendent de crides a la pròpia funció factorial.

Això ho podem portar al llenguatge de programació Python de la següent forma:

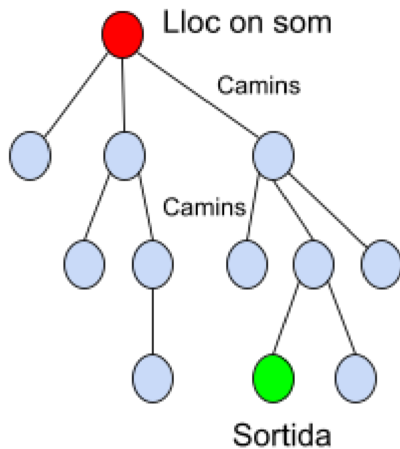
```
#Definició de funció
def factorial_recursiu (n):
   if n==1:
      return 1
   else:
      return n * factorial_recursiu (n-1)

#Programa
 = input (“Entra un nombre: “)
x = int (s)
print ( “El factorial de ”, x,” és ”, factorial_recursiu (x) )
```

Veiem que aquest programa demana un nombre i ens imprimeix el seu factorial, cridant una funció feta de forma recursiva.

La recursivitat és una tècnica molt útil en algorísmia perquè molts problemes es solucionen de forma senzilla de manera recursiva.

Per exemple si volem trobar la sortida a un laberint on anem trobant diferents desviacions (les habitacions són els cercles i els camins estan marcats com a ratlles), podem dir que la solució és, des del lloc on som agafar tots els camins possibles i repetir el procés (tornar a fer el mateix) fins que trobem la sortida.

Representació gràfica dels camins de sortida d'un laberint:



A nivell de pseudocodi podríem definir la sortida del laberint així:

funcio ***Sortir_laberint*** *(Habitació)*

si *Estem_a_la_Sortida (Habitació)*

Acabar ( )

sinó

per *C* ***en*** *Camins_possibles (Habitació)*

Sortir_laberint (Seguir_Camí (C))

Veiem que la funció *Sortir_laberint ( )* es va cridant a ella mateixa, i diverses vegades.

- Ens caldrà una funció addicional *Estem_a_la_Sortida ( )* que ens diu si som al final.
- Ens caldrà també una funció *Seguir_Camí ( )* que ens indiqui a quina habitació anem per un camí concret.
- Finalment ens caldrà una funció *Camins_possibles ( )* que ens torni la llista de camins de sortida d'una habitació.

La implementació del codi recursiu en un llenguatge com Python pot ser senzilla, però quan el nombre de crides a fer és molt gran (imaginem fer el factorial de 1000) és molt més costós que una solució iterativa com les que hem vist anteriorment. Cridar una funció requereix molta memòria a l'ordinador.

El següent codi de Python mostra les dues versions del factorial, recursiu i iteratiu.

```
def factorial_recursiu(n):
    if n==1:
        return 1
    else:
        return n*factorial_recursiu(n-1)

def factorial_no_recursiu(n):
    x=n
    while(n>1):
        n=n-1
        x=x*n
    return x


print (factorial_recursiu(125))
print (factorial_no_recursiu(125))
```

Amb la seva sortida (naturalment dóna el mateix en els dos casos):

1882677176888926099743767702491600857595403648714924258875982315083
5315633161359886688293288949592313364640544593005774063016191934138
0597818883457558547055524326375565007131770880000000000000000000000
000000000

2.3 Estructura de les dades

De la mateixa manera que amb el codi és una bona idea pensar quina sèrie d'operacions cal fer i com estructurar-les, amb les dades que farà servir el programa.
Per exemple si hem de fer un programa que calcula el producte de dos nombres, bàsicament ens caldrà una variable per emmagatzemar els dos factors i una tercera pel resultat. Si no se'ns especifica res més, caldrà preguntar si seran nombres enters o reals a qui ens encarrega la feina.
Si se'ns demana recórrer una llista cercant un nom, sabem que tindrem involucrades una llista, una variable de tipus booleà que determinarà si l'hem trobat o no i un índex per recórrer la llista.

Quan les variables són de tipus simples, com variables que guardaran un valor, les inicialitzarem al principi del programa per tenir clar què i perquè les farem servir. Quan les variables requereixen una estructura intentarem fer servir alguns dels tipus bàsic de Python o adaptar-los.

2.2.1 Taules, matrius, tuples

Una **taula** és un tipus de dades estructurat, homogeni, amb accés directe als elements i en principi de dimensió fixa.

Un exemple de taula podria ser:

Volem comptar el nombre de persones que visiten un museu cada mes.

Una solució seria tenir un conjunt de variables de tipus enter:

ComptadorGener = 0
ComptadorFebrer = 0
ComptadorMarç = 0
etc.

Però és més pràctic i elegant tenir una única taula de comptadors, que tindrà 12 elements:

TaulaVisitants[12]

0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

On cada element serà el comptador de persones de un dels mesos. Hi podrem accedir fent x = TaulaVisitants[3]

El llenguatge de programació Python no té un tipus taula directament però s'implementa a partir de llistes. Si a una llista posem un nombre determinat d'elements (ex. 12) i tots els elements que hi posem són del mateix tipus, ja complim les condicions d'una taula.

Imaginem ara, que volem guardar les dades dels visitants dels 12 mesos, però que també volem poder veure les dels darrers 10 anys. Naturalment podríem crear 10 variables de tipus taula:

TaulaVisitants2022[12]
TaulaVisitants2021[12]
TaulaVisitants2020[12]
etc.

Però podrem organitzar millor la informació en format de **matriu**. La matriu serà una estructura amb dades de tipus homogeni, amb accés directe als elements i de dimensió (nombre d'elements) finita.

En general una matriu es declara així:

TaulaVisitants[12][10]

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

A les files i columnes hi tindrem els comptadors de visitants dels diferents mesos i anys. Hi accedirem així: x = TaulaVisitants [2][5]

El llenguatge Python pot implementar les matrius utilitzant llistes també.

Exemple:

```
In [1]: Matriu = [[0,0,0],[0,0,0],[0,0,0]]

In [2]: Matriu[2][1]=5

In [3]: Matriu
Out[3]: [[0, 0, 0], [0, 0, 0], [0, 5, 0]]
```

Atenció !! veiem que els índexs compten des de 0.

L'ús de llistes per implementar taules i matrius funciona bé, però és poc eficient en us de memòria. Veurem que moltes vegades, especialment quan hem de fer processament de les dades, que les taules i matrius s'implementen utilitzant la llibreria **numpy**.

Quan volem treballar amb dades estructurades, amb accés directe, de dimensió fixa, però de tipus heterogenis organitzarem les dades en **tuples**. Veiem un exemple amb el tipus tupla de Python, on guardem 3 dades: dues cadena i una numèrica:

Alumne = ("Lucía", 13, "2n ESO")

En Python podem crear llistes de tuples i després treballar amb elles, fer cerques o diferents ordenacions.

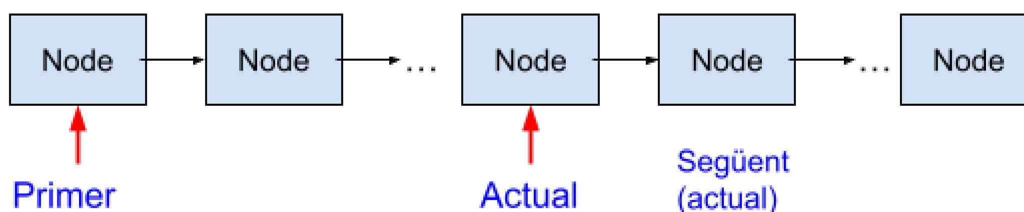
> *Grup = [("Lucía", 13, "2n ESO"), ("Mar", 14, "2n ESO"), ("Lee", 14, "3r ESO"), ("Jonathan", 16, "4t ESO")...]*

2.2.2 Llistes, piles, cues

En informàtica, quan es vol organitzar la informació de forma genèrica (sense preocupar-nos de què hi ha a l'interior o quina és la informació cocreta) es parla de tipus abstractes de dades. Els principals són les llistes, les piles i les cues (n'hi ha de més avançats com els arbres o els grafs).

2.2.2.1 Llistes

Des del punt de vista d'organització, una llista és una estructura on tenim un element que és el primer (tindrem un apuntador a ell), un element que és l'actual (serà un apuntador que va canviant) i una opcionalment una operació que donat un element ens pot dir quin és el seu següent.



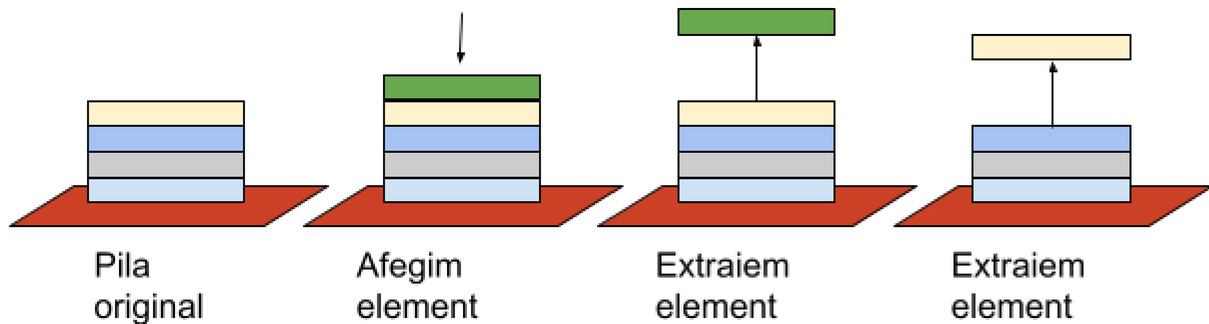
Veiem que amb això n'hi haurà prou per poder recórrer tots els elements i si cal, tornar al principi. Hi ha implementacions de llistes més elaborades que també implementen l'accés a l'últim i l'operació que ens torna l'element anterior de l'actual. També hi ha implementacions on l'actual pot ser modificat directament, sense recórrer a següent o anterior.

Per acabar, ens cal una operació per afegir elements a la llista, depenent de la implementació l'afegit es fa al final de la llista o a continuació de l'element actual (cosa que ajuda a mantenir l'ordenació, portant l'actual al lloc adequat).

El llenguatge **Python** permet l'accés a la llista pel primer Llista [0], pel darrer Llista [-1] o Llista [len(Llista) -1] o a un element concret Llista [i]. En el cas de Python haurem de recordar nosaltres per on anem de la llista si volem implementar funcions com següent( ).
Per afegir elements, Python disposa de append( ) que afegeix al final de la llista i de la funció insert (pos) que pot posar un element a una posició concreta.
L'accés directe als elements d'una llista que ofereix Python va més enllà de la definició conceptual del tipus llista, però ens donarà funcionalitats extra.

2.2.2.2 Piles

A nivell d'informàtica, una altra estructura de dades interessant és la **pila**. Imaginar-nos una pila és senzill, imaginem que estem posant plats un al damunt de l'altre. Els plats només es poden afegir per dalt (posar al damunt un altre) i treure també per dalt. En cas de que vulguem un plat que estigui al mig, haurem d'anar traient tots els de dalt fins arribar-hi.



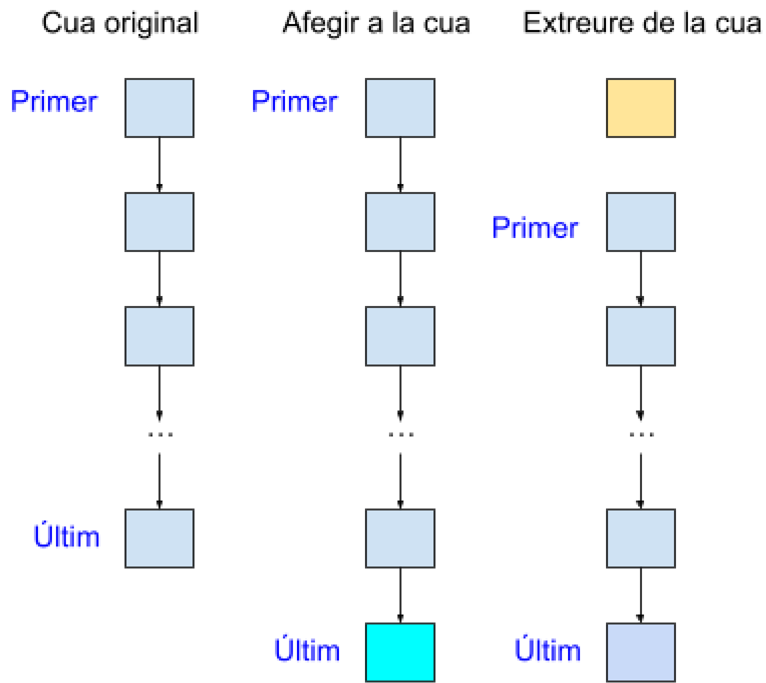
Per implementar una pila, ens caldrà una operació de crear la pila, una d'afegir element i una d'extreure element. Per controlar si la pila és buida, tindrem prou amb una operació que ens torni un booleà dient si és buida o no, o fer que la funció extreure torni un error si no hi ha res a extreure. És més elegant tenir la funció que ens digui si està buida.

Python no suporta el tipus pila de manera natural, però a partir d'una llista serà fàcil d'implementar. Per afegir (apilar) elements a la pila, farem: insert (0, element) és a dir, posarem un nou element a la posició 0, desplaçant la resta. Per extreure (desapilar) farem servir: pop (0) que extreu l'element de la posició 0. Per saber si la pila està buida, podem cridar la funció: len (llista). Si torna 0 és que està buida.

2.2.2.3 Cues

El darrer tipus abstracte per organitzar les dades és la cua. L'estructura d'una cua és fàcil d'entendre. Imaginem una cua qualsevol d'una botiga. Quan arriba una persona es posa al final, quan una persona és atesa, és la primera i surt de la cua. És a dir, afegim per darrera i extraiem per davant.

Veiem a la figura com funciona conceptualment una cua:



Python no suporta directament el tipus cua, però a partir del tipus llista i les seves funcions serà fàcil d'implementar. Per afegir elements a la cua, farem simplement un llista.append (element), que l'afegirà per darrera. Per extreure elements de la cua farem llista.pop(0) que extreurà el primer. Com amb les altres estructures, podem fer servir la funció len(llista) per saber si està buida.

2.4 Creació de classes

Fins ara hem vist per separat com es pot organitzar el codi (tipus de codis, funcions, organització de les funcions) i com es poden organitzar les dades en diferents estructures. La majoria de llenguatges de programació permeten crear **classes** o **objectes** on sota un concepte concret, podem agrupar-hi estructures de dades i les operacions per treballar-hi.

Anem a veure un exemple, acabem de veure el concepte de cua. Seria una bona idea agrupar tant les dades que conté la cua com les funcions per treballar-hi en un únic bloc que ens faciliti la seva utilització. Això serà una classe, anem a veure-la implementada en Python.

Exemple 2.4.1: Implementació d'una classe cua

```
class cua():

    def __init__(self):
        self.dades=[]

    def afegir(self, element):
        self.dades.append(element)

    def extreure(self):
        return self.dades.pop(0)

    def buida(self):
        return len(self.dades)==0
```

La paraula reservada **class** indica que anem a crear una classe, el seu nom serà cua. Veiem que això inicia un bloc de codi (pels :) on anirà tot el lligat amb la classe cua.

- La primera funció que veiem és una mica peculiar. S'anomena __init__( ): i és una funció que es cridarà per defecte cada cop que creem una variable del tipus cua. El codi que hi posem és associar un camp dades a la cua, que serà una llista buida [ ].
- La segona funció afegir ( ), rebrà un element i l'afegirà amb un append al final de la cua.
- La funció extreure ( ) ens retornarà l'element primer de la cua, eliminant-lo de la mateixa.
- Finalment, la funció buida ( ) ens torna un booleà comprovant si la cua està buida o no, utilitzant la funció len ( ) i mirant si torna 0.
- Ens queda comentar la paraula reservada **self**. *Self* es refereix a la propia classe que estem creant, per tant permetrà en qualsevol de les funcions, accedir a les dades que tinguem contingudes a la classe.

Veiem ara un exemple de com funciona la classe cua, tal com l'hem implementat:

```
class cua():

    def __init__(self):
        self.dades=[]

    def afegir(self, element):
        self.dades.append(element)

    def extreure(self):
        return self.dades.pop(0)

    def buida(self):
        return len(self.dades)==0


c = cua()
c.afegir("Joan")
c.afegir("Maria")
c.afegir("Kevin")

while not c.buida():
    print (c.extreure())
```

La sortida que tenim és aquesta: *Joan*
Maria
Kevin

On veiem que es respecta l'ordre tal com esperem a una cua.

Exemple 2.4.2: Implementació d'una classe llista

```
class llista ():

    def __init__(self):
        self.dades=[]
        self.actual=0
        self.num=0

    def seleccionar (self, nombre):
        self.actual = nombre

    def afegir(self, element):
        self.dades.insert(self.actual,element)
        self.num=self.num+1

    def extreure(self):
        self.num=self.num-1
        return self.dades.pop(self.actual)

    def consultar(self):
        return (self.dades[self.actual])

    def buida(self):
        return len(self.dades)==0

    def quants(self):
        return self.num
```

Veiem, com a diferències principals amb la cua, que, en aquesta implementació d'una llista a la inicialització iniciem tres camps: les dades, que serà una llista de Python, una variable actual que ens indicarà per quin element estem i una variable num que comptarà el nombre d'elements que té la llista.

- La funció seleccionar ( ) ens permet apuntar a algun dels elements de la llista.
- La funció afegir ( ), posa un element a la posició actual i incrementa el nombre d'elements.
- La funció extreure ( ) elimina l'element de la posició actual i decrementa el nombre d'elements.
- La funció consultar ( ) retorna l'element de la posició actual sense eliminar-lo.
- La funció buida ( ) retorna un booleà indicant si la llista està buida o no.
- La funció quants ( ) retorna un enter amb el nombre d'elements de la llista.

Veiem com podem utilitzar la classe llista:

```
l=llista()
l.afegir("Joanna")
l.seleccionar(1)
l.afegir("Maria")
l.seleccionar(2)
l.afegir("Moha")
l.seleccionar(1)
l.afegir("Kevin")

i=0
while i<l.quants():
    l.seleccionar(i)
    print (l.consultar())
    i=i+1
```

Obtenim a la sortida: *Joanna*
Kevin
Maria
Moha

Tal com esperàvem.

Exercici 2.4.1. Observem la implementació de la llista en Python. La funció buida ( ) comprova si hi ha elements a la llista cridant len ( ) sobre les dades guardades.

Proposa una forma alternativa d'implementar la funció buida ( ).

Exercici 2.4.2.

La funció seleccionar ( ) no verifica si el nombre apunta a un element de la llista o no. Imaginem que la llista té 10 elements i seleccionem el 30. Això provocarà una sèrie d'errors…

Millora la funció seleccionar ( ) perquè comprovi si el nombre apunta a un element o no, i retorni un booleà indicant si ha estat possible o no.

Exercici 2.4.3.

Després de veure la implementació als exemples 2.4.1 i 2.4.2 de les classes cua i llista…

a) Dissenya una classe **pila** amb les funcions:

   - apilar (element)
   - desapilar ( ) que retorna un element
   - buida ( ) que retorna un booleà indicant si està buida o no.

   Tot respectant el funcionament lògic de la pila.

b) Fes proves sobre la classe pila amb un programa que posi i tregui elements i pinti el funcionament per pantalla com en els exemples anteriors.

2.5 Estil de programació, noms, comentaris.

Quan fem programes hem de procurar que la feina feta sigui fàcil d'entendre per altres persones o per nosaltres mateixos en el futur. Com més clara sigui l'estructura, millor organitzat estigui el codi, més comentaris hi hagi, noms més clars tinguin les variables, etc. més fàcil serà d'entendre com funciona i de cercar i corregir possibles errors.

A continuació s'indicaran algunes pautes per fer programes elegants i funcionals.

2.5.1 Estructura general dels fitxers de codi

(1)
Comentaris inicials: Nom del programa, autor, data versió.

(2)
importació de mòduls o funcions (si cal)

(3)
declaració de funcions pròpies que farem servir (si cal)
declaració de variables que farem servir

(4)
programa que desenvolupem

Organització:

És una bona idea començar amb un comentari (una o diverses línies) on indiquem el nom del programa, la funcionalitat esperada, l'autor o autors, la data del darrer accés i la versió del programa si n'estem desenvolupant diverses.

En cas de que utilitzem mòduls estàndard de software (importació de llibreries que ens ajudin a tenir més funcionalitats) que hagin fet altres persones, les posarem al principi de tot.

A continuació posarem la declaració de funcions pròpies que hem decidit segmentar del programa per clarificar-lo o perquè l'utilitzem diferents vegades. Tot seguit fem la declaració o inicialització de les variables que utilitzarem, cosa que ens ajuda a planificar bé el nostre codi.

Finalment el codi principal del nostre programa, on posem les instruccions i crides a funcions que solucionaran el problema plantejat.

Exemple. El següent codi ens tornarà l'àrea d'un cercle a partir del radi. El radi ens el demanarà a l'usuari.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Calculadora d'area del cercle
Creat el divendres 10 de junt de 2022 a les 18:09:58
Autor. Profe Python
Versió: 1.0
"""

#importem la llibreria matemàtica numpy per disposar del nombre pi

import numpy as np
pi   = np.pi

# funció que rep el radi i torna l'area

def calcula_area_cercle(r):
    a=pi*r*r
    return(a)

# variables involucrades que utilitzarem

radi = 0.0     # el radi del cercle
area = 0.0     # l'area que serà el resultat
entrada = ""   # una cadena de caràcters per gestionar l'entrada

# programa

entrada = input("Entra el Radi: ")
radi = float(entrada)   # convertim la cadena rebuda a float
area = calcula_area_cercle(radi)

print ("L'àrea del cercle és: ", area)
```

2.5.2 Noms de les variables i estructures

Els noms de les variables seran generosos, com veiem a l'exemple, s'entén millor el codi si les variables tenen noms que fan referències a les seves funcions. Millor "radi", "àrea", "entrada" que no a,b,c… etc.

2.5.3 Identació del codi

Hi ha llenguatges com C que els blocs de codi (generats per if, while o for) s'han de posar entre claus ' { ' iniciar, ' } ' tancar. El llenguatge Python obliga a que els blocs estiguin tabulats: cada bloc va un tabulador cap a la dreta, el que facilita la lectura del que hi ha dins i la separació de la resta del codi.

```
while (condició):
    bloc de codi
    del while
continuació_programa
```

2.5.4 Comentaris

A banda del comentari inicial sobre l'autor, data i versió, serem generosos posant comentaris a les funcions, a parts del codi interessants, per indicar on comença o acaba un bloc, etc.

2.5.5 Expressions lògiques

Tot i que nosaltres podem conèixer perfectament com s'avaluen les expressions lògiques al llenguatge de programació que utilitzem, expressions molt llargues poden portar a confusió o ser font d'errors. S'aconsella posar parèntesis per fer més comprensible la seva lectura.

Exemple. Volem saber si un punt al pla R^2 està al primer o tercer quadrant.

```
if x>0.0 and y>0.0 or x<0.0 and y<0.0:
    print ("SI")
else:
    print (“NO”)
```

Millor escriure:

```
if  ( x>0.0 and y>0.0 )  or  ( x<0.0 and y<0.0 ):
    print ("SI")
else:
    print (“NO”)
```

2.5.6 Llargada i profunditat dels blocs

Si el codi de dins d'un bucle while, for, o el codi del cas d'un if o else, és tan llarg que no cap en una pantalla, això dificultarà la seva comprensió (anar pujant i baixant). És probable que algun tros es pugui extreure a nivell lògic cap a funcions.

Si els blocs de codi van aprofundint en excés, o s'acumulen massa avaluacions de casos i condicions, per exemple:

```
if (condicio1):
    while (condicio2):
        if (cas1):
            codi
        elif (cas2):
            while (condicio3):
                for recorregut_estructura:
                    etc…
```

És probable que hi hagi alternatives millors al codi, que valgui la pena estructurar-lo en funcions o veure si es poden fer pemutacions. La probabilitat d'equivocar-se i deixar-se casos en un codi com aquest és altíssima.

3. Llibreries en Python, utilització i creació.

Una de les grans forces de Python és la varietat de llibreries (també anomenades mòduls) existents que ens permetran, de manera senzilla, fer qualsevol cosa. Al voltant de Python hi ha una gran comunitat que en diferents àmbits: científics, tecnològics, matemàtics, lingüístics, ha anat desenvolupant llibreries estàndard i gratuïtes que tots podrem utilitzar.

3.1 Utilització de llibreries

3.1.1 Instal·lació

Per utilitzar una llibreria de Python cal, en primer lloc, que la tinguem instalada. Si instal·lem un paquet estàndard gran com Anaconda ja tindrem disponibles una gran varietat de llibreries. Sinó haurem de cercar "a mà" com es fa la instal·lació. Naturalment, cada llibreria tindrà una pàgina web explicant com procedir per tenir-la al nostre ordinador.

La informació genèrica sobre com funciona la instal·lació de 'packages' a Python la trobem a la referència: https://packaging.python.org/en/latest/tutorials/installing-packages/

Per saber quins mòduls o llibreries tenim instal·lats, podem fer:

- des d'una consola de Python escriure:

  >> *help ('modules')*

  I se'ns mostrarà la llista de mòduls instal·lats al nostre sistema.

- També des de la consola, o al programa podem escriure:

  >> *import nom_llibreria*

  en cas de que no estigui instal·lada rebrem un missatge d'error.

- Finalment, molts entorns gràfics com Anaconda, tenen una opció als menús per veure quins mòduls tenim instal·lats.

La forma estàndard d'instal·lar llibreries a Python és utilitzant **pip**. Pip cercarà la versió més actualitzada, resoldrà si ho ha de fer des del codi font (*sdist: source distribution*) o de versions precompilades (*wheels*) i ho posarà en el lloc correcte perquè la tinguem disponible.

3.1.2 Ús de les llibreries

Com s'ha apuntat a l'apartat anterior, per fer servir un mòdul o llibreria en els nostres programes, es farà servir la comanda **import**. A continuació de *import* posarem el nom de la llibreria.
Programant en Python veurem que hi ha unes formes habituals de fer servir la importació de llibreries. Anem a veure les quatre més habituals:

- import *llibreria*

  Estem portant tota la *llibreria* al nostre programa. Aquest serà el nom (*llibreria*) per fer-la servir, invocant les seves definicions o les seves funcions.

  Exemple, codi que demana el radi d'una circumferència i ens diu el perímetre:

```
import numpy     # importem la llibreria numpy per tenir el nombre pi

radi = float ( input (“ Entra el radi: “)
perimetre = radi * 2.0 * numpy.pi            # invoquem pi de numpy
print (perimetre)
```

- import *llibreria* as *lib*

  Portem la llibreria amb un nom diferent, normalment més curt, per estalviar escriptura.

  Exemple, el mateix d'abans, anomenem a numpy np:

```
import numpy as np     # importem la llibreria numpy amb nom np

radi = float ( input (“ Entra el radi: “)
perimetre = radi * 2.0 * np.pi          # invoquem pi de np (numpy)
print (perimetre)
```

- from *llibreria* import *part*

  Portem d'una llibreria una part, que pot ser una funció, grup de funcions. Va bé per tenir el codi més clar i no inundar l'espai de noms del nostre programa (si ho importem tot). A canvi si després ens calen més mòduls, caldrà afegir més línies.

Exemple, de la llibreria matplotlib (que és molt extensa) volem usar només les funcions de dibuixar:

```
from matplotlib import pyplot
```

- from *llibreria* import *part* as *nom*

Importem una part de la llibreria i li donem un nom que ens sigui còmode d'utilitzar:

Exemple: El mateix d'abans, però donant-li un nom que ens convingui més, en aquest cas plt, per dibuixar unes línies entre punts.

```
from matplotlib import pyplot as plt
                      # anomenem plt a les funcions de dibuix
x = [ -3, -2, -1, 0, 1, 2, 3 ]
y = [  5,  3.  4,  6, 3, 3, 2 ]

plt.plot( x, y, 'r' )         # definim punts de l'eix x i y, color vermell
plt.show( )                   # pintem
```

3.2 Creació de llibreries

Naturalment, nosaltres podem crear les nostres pròpies llibreries, els tipus abstractes de dades vistos a l'apartat 2.4 poden ser empaquetats en un mòdul nostre de Python que farem servir des dels nostres programes amb la comanda import. D'aquesta manera, se'ns facilitarà l'organització del codi, el reaprofitament a diversos projectes, etc.

Exemple. Creem un fitxer '*TADs.py*', amb el codi de les nostres classes cua i llista.

TADs.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Sat Jun 11 16:10:55 2022
5  @author: Profe Python
6  """
7
8  class llista ():
9
10     def __init__(self):
11         self.dades=[]
12         self.actual=0
13         self.num=0
14     def seleccionar (self, nombre):
15         self.actual = nombre
16     def afegir(self, element):
17         self.dades.insert(self.actual,element)
18         self.num=self.num+1
19     def extreure(self):
20         self.num=self.num-1
21         return self.dades.pop(self.actual)
22     def consultar(self):
23         return (self.dades[self.actual])
24     def buida(self):
25         return len(self.dades)==0
26     def quants(self):
27         return self.num
28
29 class cua():
30
31     def __init__(self):
32         self.dades=[]
33     def afegir(self, element):
34         self.dades.append(element)
35     def extreure(self):
36         return self.dades.pop(0)
37     def buida(self):
38         return len(self.dades)==0
```

I ara amb un fitxer diferent, per exemple '*ProvaTADs.py*', utilitzem la llista. El fitxer on hem creat la nostra llibreria, ha de ser "localitzable" des de '*ProvaTADs.py*'. Pot estar a la mateixa carpeta o a carpetes comunes compartides de Python.

ProvaTADs.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 22 14:41:19 2022

@author: Profe Python
"""
import TADs


l = TADs.llista()
l.afegir("Joanna")
l.seleccionar(1)
l.afegir("Maria")
l.seleccionar(2)
l.afegir("Moha")
l.seleccionar(1)
l.afegir("Kevin")

i=0
while i<l.quants():
    l.seleccionar(i)
    print (l.consultar())
    i=i+1



```

El programa funciona correctament:

Joanna
Kevin
Maria
Moha

En general podrem criar tantes llibreries com vulguem, però és una bona idea agrupar-les conceptualment: per una estructura d'informació (mapa, taula, biblioteca) i les seves funcions, per conjunts d'operacions matemàtiques, per funcions relatives a personatges de jocs, etc.

Sols caldrà vigilar que el nom que donem a la llibreria nostra no entri en conflicte amb les més habituals de Python.

3.3 Algunes llibreries interessants

A continuació anomenarem algunes de les llibreries més habituals en els inicis de Python, que ens donaran funcionalitats necessàries als nostres programes. Veurem alguns exemples de les llibreries, cadascuna té el seu manual amb una explicació detallada.

3.3.1 random

La llibreria *random* ens ofereix funcions per generar nombres aleatoris, que seran útils per estadística, inicialització de jocs, etc.

Exemple.

```
import random

valor = random.randint (1, 100)        # generem un valor de 1 a 100

print (“ El valor és: “, valor )
```

3.3.2 numpy

La llibreria *numpy* ens ofereix multitud d’operacions matemàtiques optimitzades perquè corrin el més ràpid possible en el processador. Per garantir un ús òptim dels recursos hardware i una bona organització a memòria, ens ofereix també estructures de dades (bàsicament *arrays*) on col·locar la nostra informació.

Exemple 1. Calculem el sinus d’una sèrie de 0 a 2*PI

```
import numpy as np

x = np.arange( 0.0, 2.0*np.pi, 0.01)   # creem un array de valors de 0 a 2 pi
                                       # amb pas 0.01
y = np.sin ( x )                       # calcula amb una sola crida el sinus de
                                       # tots els elements de l’array

print (x)
print (y)
```

Exemple 2. Treballem amb matrius

```
import numpy as np

m = [ [1.0,1.3,2.9] , [0.0,2.4,1.5] , [1.2,1.0,0.5] ]
M1= np.array(m)
print (M1)              # declarem les files d'una matriu i les passem a un
                        # array de numpy i el mostrem per pantalla
ID =np.identity(3)
print (ID)              # identity(n) crea una matriu identitat de la dimensió n

M2= np.linalg.inv(M1)  # provem de fer inversa de M1 a M2 amb les funcions
print (M2)             # de linear algebra de numpy

M3=np.dot(M1,M2)      # fem el producte de matrius per provar
print (M3)
```

I obtenim per pantalla:

```
[[1.  1.3 2.9]
 [0.  2.4 1.5]
 [1.2 1.  0.5]]
```

- La matriu declarada a M1

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

- La matriu identitat de dimensió 3

```
[[ 0.04752852 -0.35646388  0.79372624]
 [-0.2851711   0.4721166   0.23764259]
 [ 0.45627376 -0.0887199  -0.38022814]]
```

- La matriu inversa de M1, anomenada M2

```
[[1.00000000e+00 1.11022302e-16 2.22044605e-16]
 [0.00000000e+00 1.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.38777878e-17 1.00000000e+00]]
```

- El producte de M1 i M2, (quasi, per precisió) la identitat.

Atenció! amb les matrius l'operació *multiply* multiplica element a element. *dot* fa el producte matricial habitual.

Val la pena explorar les diferents opcions que ens dóna numpy per exercicis matemàtics.

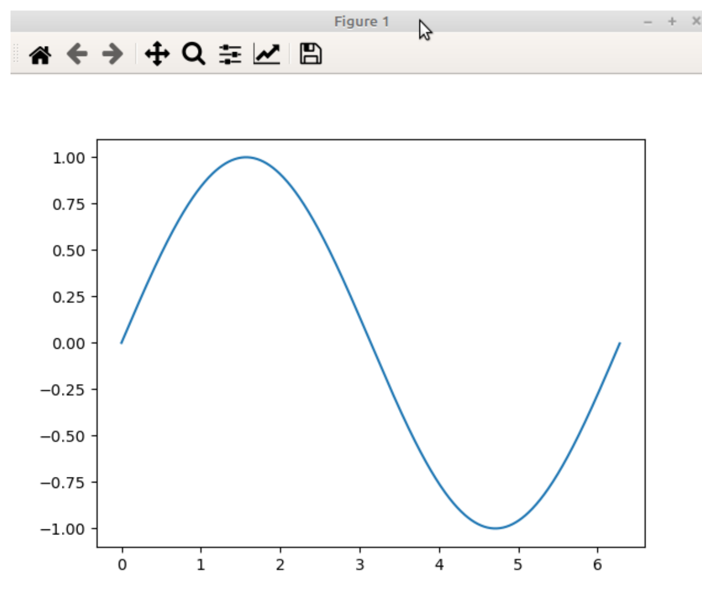
3.3.3 matplotlib

matplotlib és una llibreria multiplataforma que associada amb *numpy* ens permet visualitzar dades i fer tota mena de gràfics matemàtics. Es va crear com una alternativa lliure i gratuïta a eines com Matlab.

Exemple 1. Representem el sinus calculat amb numpy

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange( 0.0, 2.0*np.pi, 0.01)   # creem un array de valors de 0 a 2 pi
                                       # amb pas 0.01
y = np.sin ( x )                       # calcula amb una sola crida el sinus de
                                       # tots els elements de l'array
plt.plot( x, y)                        # presentarem y respecte a les x
plt.show( )                            # fem aparèixer el dibuix
```

Obtenint de forma senzilla la figura:



Matplotlib permet generar una gran varietat de gràfics 2D, 3D i animats.

3.3.4 tkinter

Fins ara hem estat veient exemples on el codi feia interacció amb l'usuari a través de la consola de Python. Allà mitjançant les funcions *print* i *input* podem imprimir i demanar dades a l'usuari. La llibreria *tkinter* permet generar programes amb interfícies d'usuari estàndard com botons, caixes de text, etiquetes, etc.

El següent exemple mostra com crear una finestra, posar-hi un botó, una capsa d'entrada d'informació, una capsa per mostrar informació i com associar una funció a un event de que s'ha premut el botó.

Exemple amb diversos elements de tkinter:

```
import tkinter as tk

def boto_pres (event):          # funció associada al botó. print a pantalla el
                                # nom i al textbox text saluda i compta lletres
    print ("Hola", entry.get() )
    text.insert ("1.0", "Hola "+ entry.get( ) + "\n")
    text.insert ("2.0","El nom té "+ str (len (entry.get() ) ) + " lletres\n")


w = tk.Tk( )                    # declarem una finestra de tkinter w i li
w.title ( "Prova tkinter" )     # donem un nom i una mida
w.geometry ( "300x450" )

# ara crearem diversos objectes: una etiqueta per veure com posar-ne,
# una caixa per entrar informació (entry), un botó activable amb el ratolí i
# una capsa de text on escriure. Per tots ells donarem mides i els ubicarem.

etiq = tk.Label ( text="Entra el teu nom: ", fg="white", bg="grey",
            width=30,height=3 )
etiq.place( x=50,y=50 )

entry = tk.Entry ( fg="blue", bg="white", width=30)
entry.place(x=50,y=100)

boto = tk.Button ( text= "Prem per analitzar",
    width=26, height=2, bg="lightgrey", fg="blue")
boto.bind( "<Button-1>", boto_pres)  # associem una funció al botó
boto.place( x=52, y=130)

text = tk.Text ( fg= "black", bg="white", width=30, height=20)
text.place(x=50,y=180)

w.mainloop()             # mainloop activa tot el mecanisme tkinter
```

Veiem que el mecanisme és bastant manual, per cada element triem colors, mides i el podem ubicar. La referència de *tkinter* ens ensenyarà com organitzar millor els elements si volem posar diversos botons (per fer una calculadora per exemple).

A l'executar el programa, obtenim:

Cal notar que tkinter pren el control de l'execució del programa amb *mainloop* i nosaltres escriurem el codi de resposta a events, com el botó.

3.3.5 pygame

La llibreria pygame ens dóna recursos per la gestió de petits jocs. Pinta imatges de fons, pinta objectes en primer pla, controla el temps d'actualització de la pantalla, detecta les tecles premudes, etc. A partir d'aquí tot dependrà de la nostra imaginació i consultant els manuals de la llibreria tindrem més exemples i explicacions de les funcions.

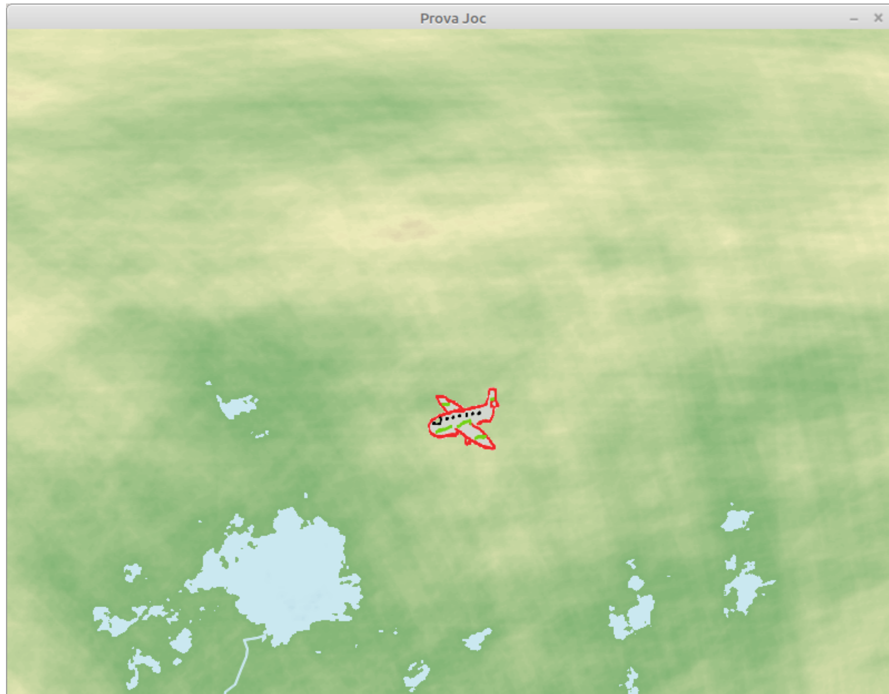
L'exemple plantejat a continuació mou un avió (dissenyat de forma senzilla per nosaltres en una imatge .png amb fons transparent) sobre un mapa pregenerat de 4000x2000 píxels. El fitxer de mapa (o el fons que volguem) i el fitxer de l'avió (o figures mòbils que volguem) han d'estar a la carpeta del programa perquè les pugui trobar la funció "load". En cas de que generem una gran quantitat de dibuixos i figures, serà una bona idea crear una carpeta de recursos del joc per tenir-ho tot ordenat.

Com en el cas de *tkinter*, passarem el control del programa al mòdul *pygame* i atendrem events com la pulsació de les tecles.

Petit exemple d'utilització de pygame:

```
import pygame
import pygame.key
from pygame.locals import *

pygame.init()                              # Creem la finestra pygame de 800x600
clock   = pygame.time.Clock()              # amb títol i inicialitzem els timers
screen   = pygame.display.set_mode((800,600))
pygame.display.set_caption("Prova Joc")
                                           # Carreguem imatge de fons i l'avió

img_fons = pygame.image.load("Mapa.png")
img_avio = pygame.image.load("avio.png")
img_f_sc = pygame.transform.scale(img_avio,(100,100))
                                           # Avió escalat a 100x100 (opcional)
x=-957
y=-439                                     # Posició original (mourem el fons)
vx=0
vy=0                                       # Velocitat inicial aturat

done = False
while not done:                            # mentre no acabem el joc

   for event in pygame.event.get():        # analitzem events: sortir (quit) o tecla
      if event.type == pygame.QUIT:
         done = True
      if event.type == pygame.KEYDOWN:

         if event.key==K_UP:               # segons les tecles donem més o menys
            vy=vy+1                        # velocitat a l'avió amb vx i vy
         if event.key==K_LEFT:
            vx=vx+1
         if event.key==K_DOWN:
            vy=vy-1
         if event.key==K_RIGHT:
            vx=vx-1

   x=x+vx                                  # Actualitzem x i y amb la velocitat
   y=y+vy
   screen.blit(img_fons, (x,y))            # pintem imatge de fons
   screen.blit(img_f_sc, (360,300))        # pintem avió al centre
   pygame.display.flip()                   # actualitzem la pantalla amb la imatge
   clock.tick(40)                          # nova i esperem 40 ms

pygame.quit()                              # a l'acabar, sortim de pygame
```

I aquest és el resultat obtingut, l'avió es mou sobre un fons:



4. Primers projectes en Python.

Els programes proposats a continuació són petits exercicis que pretenen tractar alguns aspectes bàsics de programació i de la interfície d’usuari, ús de llibreries, algorismes senzills, etc. Poden servir com a base per desenvolupaments més complexes.

Per cada programa es presenten uns petits objectius, una proposta d’estructura i uns exercicis proposats.

4.1 Programació d'un accés amb password. Identificar-nos.

Objectiu

Treballar amb funcions lògiques i casos. Treball amb cadenes. Fer una petita estructura iterativa.

Codi d'arrencada

Aquest és el codi inicial, on comprovem que una persona té accés a partir d'un password:

```
nom = "Maria"
password= "a322ZZ"

cadena_n = input ("Entra el teu nom d'usuari: ")
cadena_p = input ("Entra el teu password: ")

if (cadena_n == nom ) and (cadena_p == password ):
        print (" Endavant pots entrar ! ")
else:
        print (" Ho sento, identificació incorrecta! ")
```

Exercicis proposats

1) Executa el programa i comprova que funciona bé. Canvia les cadenes nom i password pel teu nom i un password qualsevol.

2) Personalitza el missatge d'entrada. Fes que escrigui: "Endavant Maria, pots entrar!" (canviant Maria pel nom posat)

3) Volem permetre que hi hagi dos usuaris admesos. Crea dues variables: nom1 i nom2 i dos passwords: password1 i password2 i fes que pugui entrar l'un o l'altre (naturalment cada un amb el seu password.

4) Modifica l'estructura del programa perquè hi hagi un màxim de tres intents. L'usuari podrà provar 3 cops d'entrar i sinó el programa sortirà. Anirem mostrant els intents disponibles cada vegada.

4.2 Programa endevinar nombre

Objectiu

Treballar amb llibreries. Aprendre a fer aproximacions successives. Treball amb cadenes. Fer una petita estructura iterativa.

Codi d'arrencada

Aquest és el codi inicial, on comprovem si una persona endevina un nombre entre 0 i 10:

```
import random

n = random.randint (0, 10)      # generem un nombre aleatori

resposta = input (“Entra un nombre: “)
nombre = int (resposta)         # convertim a enter

if (n == nombre ):
        print (“ Molt bé! l’has endevinat ! “)
else:
        print (“ Ho sento, el nombre era “, n)
```

Exercicis proposats

1) Executa el programa i comprova que funciona bé. L’has endevinat algun cop?

2) Canvia el programa perquè et vagi preguntant un nombre fins que l’endevinis.

   - Pensa si has de posar un bucle, quina és la condició del bucle?
   - Quina estratègia has seguit per endevinar-lo? anar provant o començant per 1,2,3…

3) Canviarem ara el programa perquè el nombre a endevinar estigui entre 0 i 100 (a la funció que genera el nombre aleatori).

   Per arribar més ràpid a la solució, se’ns donarà una pista: cada cop que entrem un nombre, ens dirà si el que hem d’endevinar és més gran o més petit. Quan l’haguem endevinat, acabarem.

4.3 Programa per comptar lletres

Objectiu

Treballar amb fitxers. Aprendre a fer recorreguts. Treball amb cadenes. Fer estadístiques.

Codi d'arrencada

Aquest és el codi inicial, on comptarem quantes ‘a’ hi ha a un text:

```
s= ” els gossos borden, la caravana passa.”

compt_a = 0     # definim un comptador
i = 0           # definim una variable per recórrer s

while i < len (s) :
        if  s[ i ] == ‘a’ :
                compt_a = compt_a + 1
        i = i + 1

print ( “ He trobat “, compt_a , “ lletres a ”  )
```

Exercicis proposats

1) Executa el programa i comprova que funciona bé. Prova a canviar el text definit a s.

2) Fes que el programa compti quantes vocals hi ha de cada. És a dir, quantes ‘a’,’e’,’i’,’o’ i ‘u’

3) Com podem arreglar el tema dels accents? detecta també ‘à’ com una a?

4) Mira’t el funcionament dels fitxers en Python. Canvia el programa perquè obri un fitxer de text i compti quantes vocals trobem de cada una.

5) Prova amb fitxers on hi escrivim textos en diferents idiomes (per exemple: català, castellà i anglès).

   Per una longitud igual dels fitxers, com ara 1000 lletres (podeu copiar i enganxar d’internet), detecteu alguna variació entre la utilització de cada vocal als diferents idiomes?

4.4 Programa de càlcul de probabilitat amb daus

Objectiu

Reforçar conceptes matemàtics. Aprendre a utilitzar taules. Aprende a utilitzar una llibreria de representació

Codi d'arrencada

Aquest és el codi inicial, on farem veure que llencem un parell de daus i comprovem si han sortit iguals:

```
import random

dau1 = random.randint (1, 6) # llencem un dau
dau2 = random.randint (1, 6) # llencem un altre dau

print (“ Ha sortit un “, dau1, “ i un “, dau2 )

if (dau1 == dau2 ):
        print (“ Això és un doble: “, dau1 )
```

Exercicis proposats

1) Modifica el codi perquè el llançament de daus i la comprovació de si són iguals es repeteixi 1000 vegades (crea un bucle).

2) Com que ara s’escriuen massa coses per pantalla, treurem els ‘prints’ i crearem un comptador que ens digui quantes, de les 1000 vegades, han sortit dos daus iguals. Per això:

   - caldrà crear un comptador fora del bucle i inicialitzar-lo a 0
   - cada cop que es dongui la condició de l’*if* incrementar-lo
   - al final, fora del bucle, escriure amb un print el valor del comptador, per saber quants dobles han sortit de les 1000 tirades.

3) Comprova que aquest nombre, correspon aproximadament amb el que s’espera fent càlcul de probabilitats. Pots provar-ho també amb un bucle de 10000 o 100000. *(continua)*

4) En comptes de comprovar si han sortit dos daus iguals, farem la suma dels dos daus. Això pot donar un nombre entre 2 i 12. Volem comptar quants cops la suma és 2, quants 3, quants 4, etc. Per fer-ho podríem declarar 11 variables comptador, però és millor:

   - inicialitzar una llista d'elements que seran els comptadors, tots a 0:
     comptadors = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

   - cada cop que fem la llançada de daus, calcular la suma i incrementar el comptador corresponent de la llista.

     Podem fer-ho així:

     suma = dau1 + dau2
     comptadors [suma] = comptadors [suma] + 1

     Fixeu-vos que és millor que anar fent 'ifs'. Té un problema però, els índexs de les llistes de Python comencen per 0. Arregleu-ho !!

   - Al final, pintar per pantalla la llista de comptadors. Es correspon al que esperem??

5) Utilitzeu la llibreria matplotlib per representar la llista de comptadors en un gràfic. Podeu basar-vos en aquest exemple.

```
import matplotlib.pyplot as plt

index        =    [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
comptadors   =    [ 0, 0, 5, 6, 7, 4, 3, 6, 7, 8,  9,  3,  2]
# Per dibuixar tenim una llista amb les ordenades i una altra amb valors

plt.grid (True)
plt.plot (index, comptadors, '--bo')
plt.show()
# Generem el gràfic amb matplotlib
```

Tot canviant els comptadors pels que calculeu vosaltres al programa.

   - Observeu el resultat, és el que s'espera matemàticament??

4.5 Programa per tornar canvi, l'algorisme de tornar canvi

Objectiu

Treballar conceptes d'algorísmia. Treballar amb xifres decimals.

Codi d'arrencada

Aquest és el codi inicial, tenim una botiga i figura que ens paguem amb bitllets de 50 euros. Haurem de tornar el canvi de la quantitat a pagar.

```
import random

pagat = 5000                       # Ens donen 50 euros o 5000 cèntims
degut = random.randint (1,5000)    # Diners a pagar en cèntims

canvi = pagat - degut

print (“ Han pagat “, pagat )
print (“ Ens deuen “, degut)
print (“ Hem de tornar “, canvi )
```

Exercicis proposats

1) Executa el programa diverses vegades i comprova que funciona bé. Heu trobat alguna cosa a millorar??

   Penseu com representar el valor de euros i cèntims d'una manera més habitual!! Programeu-la.

2) Suposem que volem tornar el canvi i que tenim unes capses amb monedes. Concretament, a l'inici tenim 100 de cada tipus de moneda que hi ha en circulació

```
mon_1c  = 100
mon_2c  = 100
mon_5c  = 100
mon_10c = 100
mon_20c = 100
mon_50c = 100
mon_1e  = 100
mon_2e  = 100
```

Fes un programa que torni el canvi necessari, indicant quantes monedes de cada tipus ha de tornar. Per exemple, la sortida ha de ser:

Canvi a tornar 1,22 euros
Tornem 1 moneda de 2 ct, 1 moneda de 20 ct, 1 moneda de 1e

I per tant ara de monedes de 2ct, 20ct i 1e ens en quedaran 99.

3) Pensa una estratègia per maximitzar el nombre de canvis que podrem tornar a la botiga. Discutiu-la amb els companys prèviament i després feu un programa que ho implementi.

   Podeu fer una petita competició, comptant el nombre de canvis que heu pogut tornar. Quan trobeu la versió òptima del programa, penseu que aquesta és la que fan servir les màquines de venda de productes.

4.6 Programa per encriptar i desencriptar

Objectiu

Treballar amb cadenes. Petites idees d'encriptació. Fer recorreguts. Treball amb diccionaris.

Codi d'arrencada

El cònsul i dictador romà Juli Cèsar, va idear un sistema per enviar els seus missatges xifrats, de manera que si algú l'interceptava, no podia entendre el que hi havia escrit.
https://ca.wikipedia.org/wiki/Xifratge_de_C%C3%A8sar
El seu sistema era molt senzill, simplement canviava una lletra per una altra desplaçant-la posicions a l'alfabet.
Aquest codi inicial, permet xifrar un missatge segons el codi de Cèsar:

```
missatge_orig = input (“Entra un text: “)
missatge_xifr = “”                    # cadena buida pel resultat
clau = 2                              # clau per encriptar

for c in missatge:                    # per tots els caràcters del missatge
        x = ord ( c )                 # el passem a numèric
        x = x + clau                  # sumem la clau
        missatge_xifr = missatge_xifr + (chr(x))
                                      # anem enganxant al resultat

print (“ El missatge encriptat és: “, missatge_xifr )
```

Exercicis proposats

1) Executa el programa i comprova que funciona bé. Mireu-vos què fan les funcions de Python *ord ( )* i *chr ( )*. Podeu provar a canviar la clau.

2) Mireu-vos com funciona la codificació dels caràcters. Veieu que al sumar una clau, podem anar a parar a símbols que no són les lletres habituals. Proposeu una forma d'arreglar això.

3) Mireu al manual com funcionen uns diccionaris. La codificació de Cèsar és molt fàcil de descodificar, utilitzeu un diccionari per crear un sistema de codificació més hàbil.

    Nota: un caràcter pot codificar a més d'un, ex. { 'a':'fs2' }

4) Serieu capaços de fer un programa que donat el missatge encriptat el desencripti?

5) Podeu fer un programa d'encriptar i desencriptar fitxers de text?

4.7 Programa per cercar el mínim a una paràbola.

Objectiu

Treballar amb les llibreries numpy i matplotlib. Fer una cerca condicionada en una taula i fer operacions matemàtiques. Representar funcions.

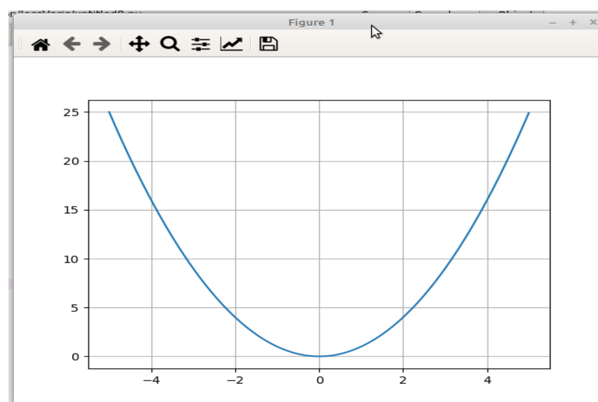
Codi d'arrencada

Aquest codi, crea dos vectors de numpy de 1000 elements i els omple amb els valors de x entre -5 i +5 (eix d'ordenades) i y del valor de x al quadrat, per tant una paràbola. Finalment la dibuixa.

```
import matplotlib.pyplot as plt
import numpy as np

x=np.zeros(1000)        # crear vector de 1000 elements
y=np.zeros(1000)        # crear vector de 1000 elements
i=-5.0                  # anirem des de -5.0
index=0
while index< 1000:
    x[index]= i
    y[index]= i*i
    i=i+0.01            # fins a 5.0, sumant de 0.01 en 0.01
    index=index+1

plt.grid(True)          # Dibuixem una quadrícula
plt.plot(x,y)           # afegim eixos x i y
plt.show()              # pintem
```

Exercicis proposats

1) Executa el programa i comprova que s'obri una finestra amb aquesta representació:



2) Observeu que la paràbola, a l'eix y, va des de 25 fins a 0, on té el mínim i torna a pujar fins a 25. En el codi, abans de fer el plt.show( ), afegeix:

```
plt.xlabel( 'eix x' )
plt.ylabel( 'eix y' )
plt.title( 'Paràbola' )
```

Amb això veiem que tenim control de l'eina de dibuix.

3) Canvieu el codi de dins del while, perquè la funció que es representi ara sigui:

$f(x) = x^2 -3x +1$

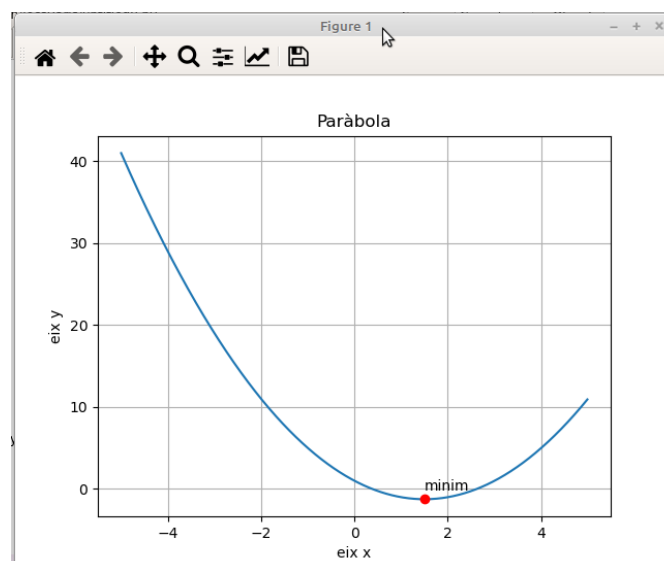
Fixeu-vos que si no tanquem la finestra de dibuix, es representa una paràbola sobre l'altre, és a dir, podem anar acumulant dibuixos si ens interessa.

4) Busqueu el mínim de la nova paràbola $x^2 -3x +1$. Pista: podem recórrer el vector y[ ], on tenim la imatge de la funció, cercant un punt que tingui un nombre major a la dreta i un major a l'esquerra; serà un mínim! Feu-ne un print de les coordenades:

El mínim és al punt (x,y) = (_____ , _____)

5) Representeu el punt del mínim al dibuix:
   - Podem pintar un punt individualment a una coordenada cx, cy fent:
     plt.plot(cx, cy, '--ro')
     on el paràmetre r es refereix al color 'red' i el 'o' a que posi una boleta.
   - Podem introduir text a un dibuix amb la comanda:
     plt.text(cx, cy, 'TEXT')

Esperem que quedi quelcom així:



4.8 Primers i factors

Objectiu

Treballar amb llistes. Aplicacions matemàtiques, nombres primers, factors.

Codi d'arrencada

Aquest codi consta d’una funció factoritzar que donat un nombre i sabent quins són els nombres primers, el factoritza, tornant una llista amb els seus factors.
El programa principal factoritza els nombres des de 4 fins a Num i omple una llista de nombres primers.

```
Primers = [2,3]                    # Inicialitzem la llista amb dos primers

def Factoritza (a):                # Funció que factoritza un nombre
        Factors = [ ]
        if a in Primers:           # Si és primer, ja estem
                return a
        else:
                for p in Primers:  # Sinó busquem els seus factors
                        while a % p==0:
                                a = a/p
                                Factors.append(p)
                if a!=1:
                        Factors.append(a)
        return Factors             # Tornem la llista de factors

# ******* Programa principal  ****************

Num = 1000
x = 4
while x<Num:

        f = Factoritza(x)          # Per cada nombre entre 4 i Num, factoritzem
        if len(f) == 1:            # Si només té un factor, és primer. L’afegim
                Primers.append ( f [0] )

        x=x+1                      # Anem pel següent

print ("Els primers són", Primers)  # Pintem la llista de primers
```

Exercicis proposats

1) Executa el programa i comprova que imprimeixi la llista de primers entre 2 i 1000.

2) Canvia Num per 10000 o 100000 i comprova que obtens una llista major de primers. Atenció!! per 100000 pot trigar més d'un minut!!!!

   Ara ens podem plantejar coses com ara:

   - La utilització de factoritzar és òptima per saber si és primer?
   - Cal recórrer tots els nombres (x=x+1) o ens podem anar saltant alguns, per exemple els parells, els que acabin en 5 o 0, etc.

3) Tornem a posar Num a 1000. A continuació del codi existent, afegim aquestes línies per comprovar que la funció *Factoritza( )* va bé.

```
print (Factoritza(320))
print (Factoritza(542))
print (Factoritza(6396))
```

```
[2, 2, 2, 2, 2, 2, 5]
[2, 271]
[2, 2, 3, 13, 41]
```

   Observant els resultats impresos per *Factoritza( )*, que torna una llista de factors.

   - Fes una funció que calculi el màxim comú divisor de dos nombres.
   - Fes una funció que calculi el mínim comú múltiple de dos nombres.

4) El nostre programa cada vegada ha de generar la llista de primers.

   Pensa maneres de tenir guardades la llista amb els nombres primers entre 1 i 1000 o 1 i 10000 de forma que ens estalviem la part inicial.