

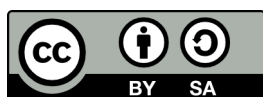
# Introducció a Python

Albert Gutiérrez Millà  
([albertgumi@gmail.com](mailto:albertgumi@gmail.com))

Juliol 2014



Aquest treball està llicenciat sota el Reconeixement Internacional 4.0 de Creative Commons.





# Consideracions inicials

El present llibre va ser creat a l'estiu de l'any 2014 pel curs *Introducció a Python* organitzat pel departament de matemàtiques de la Universitat Autònoma de Barcelona. Al finalitzar el curs, el contingut es va publicar sota llicència Creative Commons per ús i distribució lliure. Qualsevol correcció, contribució, suggeriment que es vulgui fer a aquest llibre, si us plau envieu-lo a l'adreça electrònica de l'autor<sup>1</sup>. Una versió actualitzada del llibre en format PDF, del codi font L<sup>A</sup>T<sub>E</sub>X i del codi d'exemples es mantenen en un repositori de versions GitHub<sup>2</sup>.

---

<sup>1</sup>albertgumi@gmail.com

<sup>2</sup><https://github.com/ertgumil/curs-python>



# Índex

<b>1</b>	<b>Introducció</b>	<b>1</b>
1.1	Què és Python? . . . . .	1
1.2	Estat de Python . . . . .	1
1.3	Sobre aquest llibre . . . . .	2
<b>2</b>	<b>Introducció a la programació en Python</b>	<b>3</b>
2.1	Terminal de Python . . . . .	3
2.1.1	Primer programa en Python . . . . .	4
2.2	Terminal iPython . . . . .	4
2.3	Executant el nostre codi Python . . . . .	6
2.4	Paraules reservades . . . . .	8
2.5	Comentaris de codi . . . . .	8
<b>3</b>	<b>Variables i estructures de dades</b>	<b>9</b>
3.1	Tipus de variables . . . . .	9
3.2	Assignació automàtica de tipus . . . . .	10
3.3	Variables i operacions . . . . .	11
3.3.1	Llibreria <i>math</i> . . . . .	13
3.4	Operacions amb cadenes de caràcters . . . . .	14
3.5	Estructures de dades . . . . .	15
3.5.1	Llistes . . . . .	15
3.5.2	Tuples . . . . .	19
3.5.3	Diccionaris . . . . .	19
<b>4</b>	<b>Control de fluxe</b>	<b>23</b>
4.1	Lògica booleana . . . . .	23
4.2	Operacions condicionals: if, elif, else . . . . .	24
4.3	Bucle for . . . . .	25

4.4	Bucle while . . . . .	26
4.5	Break continue pass . . . . .	27
<b>5</b>	<b>Funcions</b>	<b>29</b>
5.1	Funcions . . . . .	29
5.2	Funcions lambda, map i filter . . . . .	32
5.2.1	Lambda . . . . .	32
5.2.2	Map . . . . .	33
5.2.3	Filter . . . . .	33
<b>6</b>	<b>Esriptura i lectura de fitxers</b>	<b>35</b>
6.1	Lectura de fixers . . . . .	35
6.2	Esriptura de fitxers . . . . .	37
<b>7</b>	<b>Excepcions</b>	<b>39</b>
7.1	Gestionar excepcions . . . . .	39
7.2	Llençar excepcions . . . . .	41
<b>8</b>	<b>Mòduls i llibreries</b>	<b>43</b>
8.1	NumPy . . . . .	44
8.1.1	Declaració d'arrays . . . . .	44
8.1.2	Accés a les dades . . . . .	49
8.1.3	Operacions amb matrius . . . . .	51
8.2	Matplotlib . . . . .	53
8.2.1	Aproximació a Matplotlib . . . . .	53
8.2.2	Gràfics . . . . .	54
8.2.3	Scatter . . . . .	55
8.2.4	Gràfics de barres . . . . .	55
8.2.5	Gràfics pastís . . . . .	56
8.2.6	Gràfics amb fletxes . . . . .	56
8.2.7	Guardar gràfics en arxiu . . . . .	57
8.3	SciPy . . . . .	57
8.3.1	Exemple de SciPy . . . . .	58
<b>9</b>	<b>iPython Notebook</b>	<b>61</b>
9.1	Treballant amb el Notebook . . . . .	62
9.2	El format Markdown . . . . .	62
9.3	Inserir gràfics a iPython Notebook . . . . .	63



9.4	GitHub . . . . .	64
<b>10</b>	<b>Entorn de desenvolupament PyCharm</b>	<b>67</b>
10.1	Creant un projecte . . . . .	67
10.2	Executant . . . . .	68
10.3	Draceres útils . . . . .	71
<b>11</b>	<b>SAGE</b>	<b>73</b>
11.1	Algunes propietats . . . . .	73
11.2	Solucionar equacions . . . . .	76
11.3	Funcions . . . . .	77
11.4	Programació . . . . .	79



# Capítol 1

## Introducció

### 1.1 Què és Python?

Python és un llenguatge de programació interpretat creat per Guido Van Rossum l'any 1991, al qual li va posar aquest nom en honor al grup còmic *Monty Python*. El llenguatge és lliure i de codi obert; aquesta filosofia permet que tothom tingui accés al codi font per a contribuir-hi, modificar-lo per a les seves necessitats i distriuir-lo lliurement. Python és un llenguatge de *scripting* com els llenguatges de programació Bash o Perl i està programat en C i C++. Una de les avantatges de Python és la seva facilitat d'aprenentatge mitjançant la seva terminal. Python és indepent de plataforma. Podrem executar el nostre codi tan en Windows, Mac OS, Linux, etc. A més podrem compartir el nostre codi en plataformes on-line tal i com iPython Notebook. En l'actualitat té multitud de llibries: àlgebra lineal, bioinformàtica, visualització de dades, estadística, etc. S'empra en diferents àrees per la seva facilitat d'aprenentatge respecte a altres llenguatges de més baix nivell com C o Fortran, i per la seva rapidesa per a produir codi i entendre'l. És utilitzat a empreses, organitzacions i projectes tal i com Google, CERN, Amazon, YouTube, Anaconda Server o SAGE.

### 1.2 Estat de Python

Python es troba en transició de la versió 2 a la 3. Per a dur-la a terme hi han programes per a transformar el nostre codi d'una versió a l'altre com *2to3*. En el curs aprendrem Python 3.

## 1.3 Sobre aquest llibre

Aquest llibre tracta aspectes bàsics de la programació en Python i no arriba a temes tal i com les classes o la programació orientada a objectes. Busca apropar coneixements bàsics de programació.

Aquest llibre va acompanyat d'un fitxer amb diversos codi d'exemple del llibre. Es pot descarregar des del repositori principal del llibre. En gris es trobaran aquests exemples de codi que estan al fitxer *curs.py*. El contingut dels exemples serà el mateix, tot i que en el codi trobarem més impresions per pantalla dels resultats. Els exemples estan enumerats. *Exemple 0* significa que la funció que s'ha de cridar és la `ex000()`. Per a que el fitxer funcioni han d'estar instal·lades les llibreries NumPy, SciPy i Matplotlib, si no els `import` donaran error.

```
>>> import curs
>>> curs.ex000()
Has aconseguit cridar el fitxer de codi
```

La resta d'exemples es trobaran en color blau i els caràcters `>>>` indiquen que s'ha d'introduir la comanda a la terminal de Python.

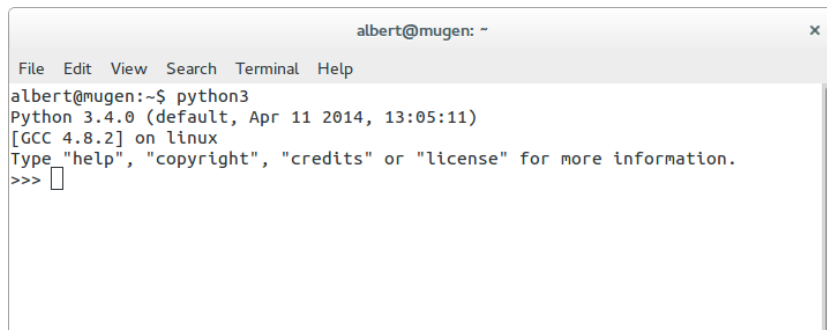
```
>>> Codi en Python
Sortida de l'execució
```

# Capítol 2

## Introducció a la programació en Python

### 2.1 Terminal de Python

Al contrari que amb llenguatges com C, C++ o Fortran no necessitem un compilador per a executar el codi en Python. Ser un llenguatge interpretat significa que un programa s'ocupa d'executar el codi Python. El programa que s'encarrega d'aquesta tasca és l'interpret. Nosaltres podem interactuar directament amb l'interpret emprant la terminal de Python. Per a obrir-la executem la terminal del sistema i introduïm la comanda `python3` o `python` en el cas de que la versió 3 estigui configurada per defecte al nostre sistema. A la Fig. 2.1 es mostra la terminal instal·lada amb Python 3. Per sortir de la terminal introduïrem `exit()` o premerem `Ctrl-D`

A screenshot of a terminal window titled 'albert@mugen: ~'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal content shows the command 'python3' being executed, resulting in 'Python 3.4.0 (default, Apr 11 2014, 13:05:11)' and '[GCC 4.8.2] on linux'. It also displays a prompt for help information. The prompt is '>>>' followed by a cursor.

```
albert@mugen: ~  
File Edit View Search Terminal Help  
albert@mugen:~$ python3  
Python 3.4.0 (default, Apr 11 2014, 13:05:11)  
[GCC 4.8.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

Figure 2.1: Terminal de Python 3

### 2.1.1 Primer programa en Python

A informàtica el primer programa que s'implementa s'anomena *Hello world*. Consisteix en mostrar per la terminal aquesta cadena de caràcters. Per a executar aquest exemple obrim un intèrpret de Python i hi introduïm el següent codi.

```
>>> print("Hello world")
Hello world
```

Exemple 1: Hello world

Veurem que el resultat és la cadena de caràcters passada com a paràmetre a la funció. En aquest exemple s'ha cridat la funció `print()` que imprimeix per pantalla cadenes de caràcters passades com a paràmetre. Si provem amb altres arguments veurem com sempre rebrem com a sortida la cadena de caràcters introduïda.

Com hem comentat abans Python es troba en una transició entre la versió 2 i 3, tot i que la versió 3 és completament funcional. Podem obrir un intèrpret de Python 2 executant la comanda `python2` o `python` (si la versió 2 es troba configurada per defecte) i introduïm el següent exemple de *Hello world* sense parèntesis.

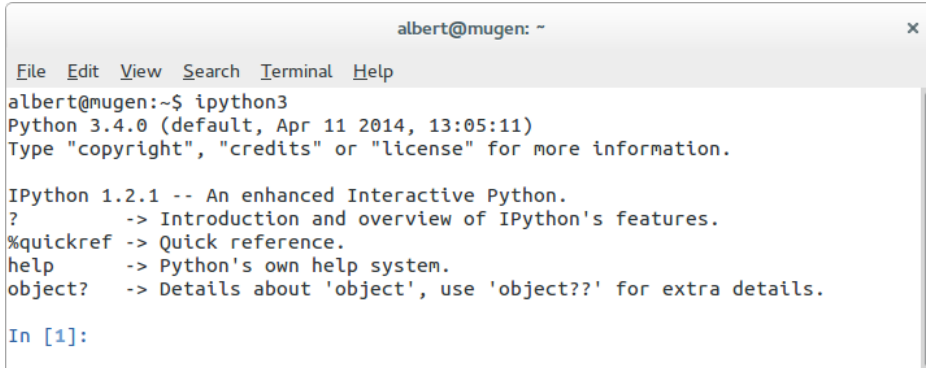
```
>>> print "Hello world"
Hello world
```

Ara provem a executar aquest mateix exemple amb Python 3 i comprovem els resultats. En Python 3 la funció `print` ha passat a ser considerada una funció més i els paràmetres han de ser cridats sempre entre parèntesis.

## 2.2 Terminal iPython

El software iPython és una terminal de Python enriquida. Té diversos mòduls implementats com la consola basada en la llibreria gràfica Qt, el quadern de codi iPython Notebook, suport per a la visualització integrada de gràfics a la terminal o eines per facilitar el còmput d'altres prestacions. No es troba instal·lada per defecte, així que s'haurà d'afegir al software del nostre sistema. Per a executar-la haurem de cridar la comanda `ipython` o `ipython3`. Té una llarga llista de funcionalitats i podem executar algunes comandes del sistema

des iPython. Per veure una referència breu intruir `%quickref`, i per a sortir d'aquest premerem la tecla *Q*. A la Fig. 2.2 es mostra la terminal iPython en funcionament.



```

albert@mugen: ~
File Edit View Search Terminal Help
albert@mugen:~$ ipython3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
Type "copyright", "credits" or "license" for more information.

IPython 1.2.1 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]:

```

Figure 2.2: Terminal de iPython

Un dels avantatges més productius de la terminal és l'autocompletat. Aquesta funcionalitat ens mostrarà totes les opcions que tenim donada un conjunt de lletres, un objecte, un mòdul, etc. Per a provar-ho introduir la lletra *p* i prémer la tecla *Tab*. Es veurà el conjunt de comandes que comencen amb la lletra *p*.

```

>>> p
%%perl      %%python    %paste      %pdef       %pinfo
%pprint     %prun       %pushd      %pylab      pow
%%prun      %%python3   %pastebin   %pdoc       %pinfo2
%precision  %psearch    %pwd        pass        print
%%pypy      %page       %pdb        %pfile      %popd
%profile    %psource    %pycat      plt         property

```

iPython permet guardar en un arxiu totes les comandes introduïdes durant una sessió. Per a desar-les s'ha de cridar a la funció `save` amb el següent format: `%save nom_sessió id_comandes`. A iPython cada comanda té un identificador numèric. Per a introduir rangs de comandes es separen per guió i per comandes individuals s'introdueixen soles com a l'exemple següent.

```

>>> %save sessio_avui 3-10 15-20 22 28

```

Podem consultar les propietats d'un objecte en Python que pot ser una funció, mòdul, variable, etc. Per a fer-ho introduïm el nom del objecte seguit del símbol d'interrogació. Per exemple consultem les propietats de la funció `print()`.

```
>>> print?
Type:          builtin_function_or_method
String Form:<built-in function print>
Namespace:    Python builtin
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current
sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

Quan estiguem treballant amb iPython i estem modificant el nostre codi font les llibreries importades no es modificaran per defecte i haurem de tancar la terminal i torar a entrar. Per evitar-ho activem l'opció `autoreload` de iPython amb les següents comandes.

```
>>> %load_ext autoreload
>>> %autoreload 2
```

## 2.3 Executant el nostre codi Python

La terminal és còmode per a realitzar probes i molt útil per a l'aprenentatge, però per a projectes ens interessarà crear el nostre propi programa en un fitxer. Per a desar el nostre codi creem un arxiu anomenat `programa.py` i hi introduïm el següent codi.



```
#!/usr/bin/env python3
print("Hello world")
```

La directiva `#!/usr/bin/env python3` s'ocupa de buscar on és l'interpret de Python 3 d'una manera portable entre plataformes Unix. Un cop tenim el nostre codi Python hi han dues maneres d'executar-lo. Una és passar-li la ruta al fitxer com a paràmetre a l'interpret de Python. Suposant que hem desat el fitxer a la nostra carpeta d'inici.

```
$ python3 programa.py
Hello world
```

Una altra manera és donar-li permisos d'execució al nostre programa. Per a això hem d'utilitzar la comanda `chmod` de Unix i amb el paràmetre `+x` que afegeix permisos d'execució a un fitxer. Un cop tenim permisos podem executar el binari especificant-li la ruta `./` més el nom `programa.py`.

```
chmod +x programa.py
./programa.py
Hello world
```

Un tercera manera és executar el programa des de la terminal. Per això hem de cridar a la funció `run` i especificar-li la ruta del programa i el nom.

```
>>> run "programa.py"
Hello world
```

## Exercici I

Executar l'exemple anterior amb la terminal iPython, guardar-lo i executar-lo des de la terminal.

## Exercici II

Executar l'exemple anterior amb la terminal iPython, desar-lo en una carpeta anomenada `exemple` i executar-lo des de la terminal. Com accedeixes a la ruta del arxiu?

## 2.4 Paraules reservades

Hi ha un conjunt de paraules que son reservades del llenguatge i no es poden emprar per a anomenar mòduls, llibreries, variables, funcions, etc. Si ho provem ens donarà un error.

```
and assert break class continue def del elif else except
exec finally for from global if import in is lambda not or pass
print raise return try while yield
```

## 2.5 Comentaris de codi

Els comentaris de codi permeten introduir text en el programa sense que sigui interpretat. Ens permet introduir descripcions als nostres programes i descriure'ls per a que qualsevol que el llegeixi entengui què fa el sistema. També s'utilitza per a fer que part del codi no s'executi. Hi han dos menes de comentaris.

- Comentaris de línia: emprem #
- Comentaris de bloc: emprem cometes triples `"""comentari"""`

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

"""
Autor: Albert Gutiérrez Millà
Data: 25 de juny del 2014
"""

# Aquesta funció retorna la cadena de caràcters per la stdout
print("Hello world")
```

# Capítol 3

## Variables i estructures de dades

Una variable és un element mutable que pot emmagatzemar valors. Per exemple en la funció  $f(x) = \sin(x)$  la variable  $x$  podrà tindre valors diferents depenent de quin li donem i a més pertanyerà a un conjunt o tipus de dada. En el cas de la programació en Python també.

### 3.1 Tipus de variables

Python emmagatzemarà i interpretarà els valors de les variables depenent del tipus de dades. A continuació mostrem alguns tipus de dades:

- *Integer*: nombres enters  
`x = 15`
- *Float*: nombres reals  
`x = -2.5`
- *Complex*: nombres complexes  
`x = -3 + 10.2j`
- *String*: llistes de caràcters  
`x = "python"`
- *Booleà*: lògics  
`x = True`

## 3.2 Assignació automàtica de tipus

Python és dèbilment tipat. Això significarà que podem assignar diversos tipus de dades a una mateixa variable. A continuació fem que una mateixa variable primer sigui un enter, després un nombre real i per últim una cadena de caràcters.

```
>>> x = 1
>>> x = 2.0
>>> x = "python"
>>> print(x)
python
```

Exemple 2: Variables dèbilment tipades

També podem realitzar operacions directament entre tipus diferents mentre aquests siguin compatibles. Per exemple realitzem una operació entre un enter i un nombre real.

```
>>> x = 1
>>> y = 2.0
>>> x/y
0.5
```

Exemple 3: Conversió implícita de dades

Al igual que podem comprovar les propietats d'una funció amb l'operador també podem consultar les variables.

```
>>> x = 3.3
>>> x?
Type:          float
String Form:3.3
Docstring:
float(x) -> floating point number
```

Convert a string or number to a floating point number, if possible.

Al ser un llenguatge dèbilment tipat, les conversions implícites de tipus són transparent per nosaltres. Per això ens pot interessar saber amb quin

tipus de dada estem treballant. La funció `type()` ens permet consultar el tipus.

```
>>> x = 1
>>> type(x)
<type 'int'>
>>> y = 2.0
>>> type(y)
<type 'float'>
>>> z = x/y
>>> type(z)
<type 'float'>
>>> print(z)
0.5
>>> a = (2+3j) * (3+5j)
>>> type(a)
<type 'complex'>
```

Exemple 4: La funció *type*

Combinada amb la funció `is` podem comprovar amb quina mena de dades estem treballant donat que podem voler evitar treballar amb tipus concrets de dades.

```
>>> x = 2
>>> type(x) is int
True
>>> x = 3.3
>>> type(x) is float
True
>>> type(x) is int
False
```

Exemple 5: Comprobació lògica de tipus

### 3.3 Variables i operacions

Els operadors de Python són semblants als que trobem en altres llenguatges. Com a operadors especials podem trobar la divisió entera i la potència. A continuació es llisten els operadors numèrics.

- Suma: +
- Resta: −
- Multiplicació : \*
- Divisió: /
- Divisió entera: //
- Reste del mòdul: %
- Potència: \*\*

Podem utilitzar el intèrpret de Python com si fos una calculadora. Fins i tot a Python podem definir els nostres propis operadors per a que realitzin les operacions que vulguem amb els nostres objectes.

```
>>> 2+3
5
>>> 2-3
-1
>>> 3*2
6
>>> 9/2
4.5
>>> 9//2
4
>>> 14%5
2
>>> 9**2
81
```

#### Exemple 6: Operacions Python

Al igual que en la calculadora la sintaxis de Python ens dona una jerarquia de les operacions.

```
>>> (2+3)*5
25
>>> 2+3*5
17
```

El tipus sencer no té limit de precisió a l'execució. Donat que Python realitza conversió implícita de dades ens trobarem que algunes operacions que impliquen canvi de tipus ens donaran error al executar.

```
>>> 2**1024
179769313486231590772930519078902473361797697894230657
273430081157732675805500963132708477322407536021120113
879871393357658789768814416622492847430639474124377767
893424865485276302219601246094119453082952085005768838
150682342462881473913110540827237163350510684586298239
947245938479716304835356329624224137216
```

Podem utilitzar el guió baix `_` per a referir-nos a `Ans`

```
>>> pi = 3.141592653589793
>>> r = 4
>>> A = pi * r**2
>>> _
50.26548245743669
```

### 3.3.1 Llibreria *math*

La llibreria `math` ens dona un conjunt de funcions matemàtiques i constants a per poder treballar. A continuació es llisten les que implementa la llibreria.

<code>acos</code>	<code>degrees</code>	<code>fsum</code>	<code>log2</code>
<code>acosh</code>	<code>e</code>	<code>gamma</code>	<code>modf</code>
<code>asin</code>	<code>erf</code>	<code>hypot</code>	<code>pi</code>
<code>asinh</code>	<code>erfc</code>	<code>isfinite</code>	<code>pow</code>
<code>atan</code>	<code>exp</code>	<code>isinf</code>	<code>radians</code>
<code>atan2</code>	<code>expm1</code>	<code>isnan</code>	<code>sin</code>
<code>atanh</code>	<code>fabs</code>	<code>ldexp</code>	<code>sinh</code>
<code>ceil</code>	<code>factorial</code>	<code>lgamma</code>	<code>sqrt</code>
<code>copysign</code>	<code>floor</code>	<code>log</code>	<code>tan</code>
<code>cos</code>	<code>fmod</code>	<code>log10</code>	<code>tanh</code>
<code>cosh</code>	<code>frexp</code>	<code>log1p</code>	<code>trunc</code>

Podem importar funcions concretes utilitzant la funció `from` i les podrem utilitzar directament en el nostre entorn.

```
>>> from math import sqrt, pow, sin
>>> sqrt(sin(pow(2,3)))
0.9946648916209829
```

O bé importar totes les funcions utilitzant el caràcter `*`.

```
>>> from math import *
sin(), cos(),
```

### Exercici III

Tenint la següent fórmula d'un tir parabòlic: quin valor de  $x$  prendrà el projectil amb una velocitat de 10m/s i un angle de 1 radian?

$$x = v_0 \cos(\alpha) \frac{2v_0 \sin(\alpha)}{g}$$

### Exercici IV

Escriu un programa en Python en un fitxer que s'encarregui de resoldre de forma genèrica el càlcul del problema del pla inclinat. Les variables han d'estar declarades al principi i comentades i al final de l'execució del programa ha de mostrar els resultats imprimint-los per pantalla.

## 3.4 Operacions amb cadenes de caràcters

En el cas de les cadenes de caràcters podem emprar operadors per a formatar la sortida del nostre text. Per exemple la concatenació empra l'operador `+` o la coma.

```
>>> var1 = "Hello"
>>> var2 = "World"
>>> print(var1 + " " + var2)
```



```
Hello World
>>> print(var1,var2)
Hello World
>>> print(var1 * 3)
HelloHelloHello
```

Si intentem realitzar la concatenació amb l'operador `+` i diversos tipus de dades que no son cadenes de caràcters rebrem una excepció. Per això emprarem la funció `str()` que converteix tipus de dades a cadena de caràcters..

```
>>> print("Resultat: " + str(3))
Resultat: 3
```

## 3.5 Estructures de dades

Podem estructurar conjunts de dades en una única variable. Per això creem les estructures de dades. Depenent de la forma d'organització i la interacció serà una o una altre tal i com llistes, conjunts, tuples, etc. Nosaltres podem crear els nostres tipus d'estructures en Python, però el llenguatge ja implementa les bàsiques.

### 3.5.1 Llistes

Les llistes són conjunts de dades ordenats unidimensionalment com per exemple una llista d'edats d'una població. Les llistes permeten emmagatzemar diversos tipus de dades. Per a la inicialització introduïrem valors entre claudàtors `[ ]`. Si volem que sigui una llista buida no inclourem cap element.

```
>>> llista_buida = []
>>> llista_edats = [10,5,6,23,65,23,2,21,41]
>>> llista_elements = [2, 4+3j, 'a', "bcd"]
```

Per accedir a la llista empram index. En gairebé tots els llenguatges de programació el primer element de la llista és l'element 0. Així doncs l'element 2 serà el tercer de la llista.

```
>>> llista = [1,3,4,2]
>>> llista[2]
4
```

Quan volem obtenir una subllista podem emprar l'operador : per indicar intervals d'índex. El primer índex ens indicarà el primer índex a retornar i el segon índex és el l'últim índex que no inclourà a la subllista. Si no incloem índex al principi o al final ens retornarà des del primer element o fins a l'últim, respectivament.

```
>>> llista[1:3]
[3,4]
>>> llista[:3]
[1, 3, 4]
>>> llista[1:]
[3, 4, 2]
```

Podem accedir des de l'últim element emprant nombres negatius. L'element -1 seria l'últim, -2 el penúltim, i així.

```
>>> llista[-1]
2
>>> llista[-3:-1]
[3, 4]
>>> llista[-4]
1
```

Una paraula és una cadena de caràcters. Per tant podrem accedir de la mateixa manera, tot i que no tindrem les mateixes funcions membre.

```
>>> c = "python"
>>> c[2:]
'thon'
>>> c[:-3]
'pyt'
>>> c[4]
'o'
```

Quan copiem una llista ho fem per referència. Això significa que no realitzarem una còpia exacta valor per valor a la nova variable, si no que tindrem una referència a la primera llista

```
>>> l = [-1,3,4,-9]
>>> m = l
>>> m[0] = 4
>>> l
[3,3,4,-9]
```

Com veiem estem modificant la primera llista `l` i no la llista `m`. Per a copiar el valor de la llista retornarem tota la llista de valors i la guardarem accedint a tots els elements amb `[:]`.

```
>>> l = [-1,3,4,-9]
>>> m = l[:]
>>> m[0] = 4
>>> m
[4,3,4,-9]
>>> l
[-1,3,4,-9]
```

Hi han un conjunt d'operacions que podem realitzar amb les llistes. Per a veure quines són podem prémer la tecla **tab** després d'haver escrit la variable més un punt `llista.` a la terminal.

- `append()`: Afegir elements
- `extend(l)`: Afegir elements d'una altra llista
- `insert()`: Inserir elements en una posició concreta
- `remove()`: Eliminar el primer element de la llista que coincideixi amb un valor donat
- `pop()`: Esborrar i tornar l'últim element
- `index()`: Tornar l'índex d'un element concret

- `count()`: Trobar el nombre de vegades que troba un element
- `sort()`: Ordenar la llista. Tenir en compte els tipus de variables
- `reverse()`: Donar la volta a la llista.

```
>>> l = [10,5,6,23,65,23,2,21,41]
>>> l.append(10)
>>> l
[10, 5, 6, 23, 65, 23, 2, 21, 41, 10]
>>> l.extend(l[:3])
>>> l
[10, 5, 6, 23, 65, 23, 2, 21, 41, 10, 10, 5, 6]
>>> l.insert(0,2)
>>> l
[2, 10, 5, 6, 23, 65, 23, 2, 21, 41, 10, 10, 5, 6]
>>> l.remove(23)
>>> l
[2, 10, 5, 6, 65, 23, 2, 21, 41, 10, 10, 5, 6]
>>> l.pop()
6
>>> l.index(65)
4
>>> l.count(10)
3
>>> l.sort()
>>> l
[2, 2, 5, 5, 6, 10, 10, 10, 21, 23, 41, 65]
>>> l.reverse()
>>> l
[65, 41, 23, 21, 10, 10, 10, 6, 5, 5, 2, 2]
```

#### Exemple 7: Operacions amb llistes

Dintre d'una llista podem crear a la seva vegada altres llistes i donar lloc a estructures com matrius.

```
>>> llista = [[1, 2], [3, 4]]
```

A la seva vegada nosaltres podem concatenar llistes diferents

```
>>> l = [[1,2],[3,4]]
>>> m = [[4,3],[2,1]]
>>> l+m
[[1, 2], [3, 4], [4, 3], [2, 1]]
```

En llistes així com en la resta d'estructures de dades tenim la funció `len()` que ens diu quants elements hi ha a l'estructura de dades.

```
>>> l = [[1,2,3],[4,5,6]]
>>> len(l)
2
>>> len(l[0])
3
```

### 3.5.2 Tuples

Les tuples són estructures de dades immutables, a diferència de les llistes. Per tant els valors que hi guardem seran constants. Hi podem accedir igual que amb les llistes introduint un índex, i sempre inicialitzarem els valors entre parèntesis ( ) i separats per comes. Només tenen dos funcions: `count` per contar el número de coincidències d'un element i `index` per retornar l'índex d'un element.

```
>>> t = (1,2,3,'a')
>>> t.count('a')
1
>>> t.index('a')
3
```

Exemple 8: Operacions amb tuples

### 3.5.3 Diccionaris

Un diccionari és una estructura de dades que guarda parelles de dades clau-valor. La clau sempre es declara primer i el valor segon després de dos punts. Les funcions `keys` i `values` ens retornen les claus i els valors.

```

>>> edat = {'salvador': 23, 'maria': 41, 'helena': 39}
>>> edat['helena']
39
>>> edat.keys()
dict_keys(['helena', 'salvador', 'maria'])
>>> edat.values()
dict_values([39, 23, 41])

```

#### Exemple 9: Operacions amb diccionaris

Si volem copiar els valors d'una llista a una altre haurem de fer servir la funció `copy`. Si no només tindrem una referència a la primera llista igual que ens passava amb les llistes.

```

>>> edat2 = edat.copy()
>>> edat2['salvador'] = 24
>>> edat
{'helena': 39, 'maria': 41, 'salvador': 23}
>>> edat2
{'helena': 39, 'maria': 41, 'salvador': 24}

```

```

>>> b = edat
>>> b['helena'] = 42
>>> edat
{'helena': 42, 'maria': 41, 'salvador': 23}

```

Per afegir un altre element només hem de declarar una nova clau i valor amb la variable ja existent.

```

>>> edat['lara'] = 53
>>> edat
{'helena': 42, 'lara': 53, 'maria': 41, 'salvador': 23}

```

#### Exercici V

Resol el problema de les 8 reines per a un nombre genèric de reines. Utilitza `try` i `except` per a controlar possibles excepcions, declara una funció per imprimir el resultat i guardar els resultats en un fitxer.

## **Exercici VI**

Calcular la desviació estàndard, la mitja i la mitjana d'una llista de 5 elements i mostrar-la per pantalla.





# Capítol 4

## Control de fluxe

Fins ara hem vist variables i operacions, però la part més intensa en còmput i la que determina la variabilitat de la nostra execució són els bucles i les operacions condicionals. Realitzar una acció o un altra, o iterar diverses longituds ens vindrà determinat per condicions lògiques.

### 4.1 Lògica booleana

La lògica booleana ens permet crear funcions lògiques i condicions que determinaran el camí que recorre el nostre codi. A continuació llistem les taules de la veritat de diversos operadors

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	$\neg y$
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Table 4.1: Taula de la veritat

En la taula anterior el valor

- $\wedge$ : conjunció (&). Tots valors han de ser certs.
- $x \vee y$ : disjunció(|). Un dels dos valors ha de ser cert.
- $x \oplus y$ : disjunció exclusiva(^). Els elements han de ser diferents.

- $\neg y$ : negació(**not**). Es nega el valor.

En Python "veritat" es representa amb la paraula **True** o el número 1 i "fals" amb la paraula **False** o el número 0.

```
>>> True & 1
True
>>> 0 and True
False
>>> True or False
True
>>> True ^ False
True
>>> not True
False
>>> not(True | False)
False
```

Exemple 10: Operador lògics

## 4.2 Operacions condicionals: if, elif, else

A continuació veurem les condicions. Python utilitza tabulacions per a mostrar diversos nivells del codi. Els operadors que fem per a l'execució de condicions són **if**, **elif**, **else**. L'estructura de les condicions és la següent.

```
if expressio1:
    accions1
elif expressio2:
    accions2
elif expressio3:
    accions3
else:
    resta de casos
```

La primera condició sempre ha de ser un **if** seguida d'una condició. Després, si volem, podem afegir més condicions amb **elif** i considerar la resta de casos (sense condició) amb **else**.

Per exemple en el cas que vulguem saber si una persona és jove, es adulta o si encara no ha nascut respondrem diferentment depenent de les edats. Així doncs creem diverses condicions i prenem una decisió quan aquesta és certa.

```
edat=-2
if edat < 18 and edat >= 0:
    print("Jove")
elif edat >= 18:
    print("Adult")
else:
    print("Encara no ha nascut")
```

Exemple 11: Rangs d'edats

## 4.3 Bucle for

El bucle `for` ens permet declarar un bucle. En Python s'utilitza, majoritàriament per recorre un conjunt d'elements tal i com diccionaris, llistes, tuples o altres elements iterables.

```
for i in llista:
    realitza acció sobre element i
```

L'element `i` (que pot ser anomenat d'altra manera) contindrà cada element de la llista a mesura que realitza les accions declarades dins del bucle. Quan hagi recorregut tots els elements sortirà de la iteració.

```
for i in [0, 1, 2]:
    print("Hola " + str(i))
print("Final")
```

Exemple 12: Bucle for

La funció `range()` ens permet crear dinàmicament llistes sobre les quals iterar. Si li passem un paràmetre crearà una llista desde 0 fins al valor passat. Amb dos valors li donem un interval, i amb tres un interval i un valor d'increment. El passem com a paràmetre a la funció `list()` per a forçar que sigui una llista.

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

Podem realitzar el codi anterior iterant directament sobre la llista d'elements creats.

```
for i in range(2):
    print("Hola " + str(i))
print("Final")
```

En el cas que volguem recorre una matriu que és una llista de llistes podem imbrincar els bucles `for`. Així doncs

```
m = [[1,2,3],[4,5,6]]
for i in m:
    for j in i:
        print(j)
```

Exemple 13: Bucles imbrincats

## 4.4 Bucle while

El bucle `while` és emprat en el cas de condicions d'execució i no tan per iteracions, tal i com el bucle `for`. S'executarà sempre mentre que la condició del bucle sigui veritat. Quan sigui fals, s'aturarà.

```
while condicio:
    accio
```

En el següent exercici calculem els quadrats de nombres sense realitzar multiplicacions fins arribar a 5 i imprimim els valors.

```

i = 0
c = 0
while i < 5:
    c = c + i + i + 1
    i = i + 1
    print(c)

```

Exemple 14: Càlcul de quadrats amb el bucle while

## 4.5 Break continue pass

Hi han un conjunt de funcions que ens permet sortir del bucle, saltar a la següent iteració o introduir una operació buida.

En el cas de la comanda **continue** volem que el programa segueixi l'execució, però que passem a la següent iteració. La comanda **break** sortirà del bucle en quan s'executi. La comanda **pass** no realitza cap acció. Pot ser utilitzat quan una instrucció és requerida sintàcticament però el programa no requereix cap acció, per exemple:

```

acc = 0
for i in range(1,10):
    if i % 2 == 0:
        pass
    elif i % 5 == 0:
        break
    elif i % 3 == 3:
        acc = acc + 1
    else:
        continue
    acc = acc + 1
print(acc)

```

Exemple 15: Break continue pass

### Exercici VII

Calcula la mitjana i la moda d'una llista de valors.



# Capítol 5

## Funcions

Una funció realitza un conjunt de tasques i pot tornar o no un valor. Si, per exemple tenim la funció  $f(x) = \sin(x)$ , llavors tenim que el nom de la funció és  $f$ , el paràmetre és  $x$  i la funcionalitat i sortida del sistema és el resultat de l'operació  $\sin(x)$ . Per a declarar una funció comencem amb la paraula clau **def** i seguit pel nom de la funció. Els valors calculats es retornen mitjançant la paraula clau **return**. La sortida del sistema retorna per **return** serà només una variable. Si volem tornar diversos valors haurem d'utilitzar llistes o altres estructures. Per tant la definició de la funció que em vist abans seria, en Python.

```
def f(x):  
    return sin(x)
```

Exemple 16: Definició funció f()

### 5.1 Funcions

Una funció realitza un conjunt de tasques i pot o no retornar un valor. En el cas que no retorni cap valor, per defecte tindrem el valor *None*. En el cas dels paràmetres podem definir tants com vulguem o cap. En el cas que no passem cap argument a la funció deixarem els parèntesis en blanc **def funcio()**

```
def f():  
    print("Dintre")
```

```
>>> r = f()
>>> print(r)
```

Podem declarar variables dins de les funcions que només tindran aquest àmbit

```
def funcio():
    var = 10
    print(var)
>>> funcio()
10
>>> var
```

```
-----
NameError                                Traceback (most recent call last)
/home/albert/<ipython-input-88-5d73f6e1684c> in <module>()
----> 1 var
```

```
NameError: name 'var' is not defined
```

La sortida del sistema retorna per **return** serà només una variable. Si volem tornar diversos valors haurem d'utilitzar llistes o altres estructures.

```
def fact(x):
    llista = []
    for i in range(1,x+1):
        llista.append(math.factorial(i))
    return llista
>>> l = fact(4)
>>> l
[1, 2, 6, 24]
```

Exemple 17: Retornar llista de valors

Una funció no ha de tindre obligatòriament la funció **return**, tot i que sempre tornarà un valor. Si no li especifiquem cap valor ens retornarà **None**.

```
def imprimeix_matriu (matriu):
    for i in range(0,len(matriu)):
        for j in range(0,len(matriu[0])):
```



```

        print(matriu[i][j])
>>> r = imprimeix_matriu([[1,2],[3,4]])
1
2
3
4
>>> print(r)
None

```

Podem tindre diverses instàncies de **return** i depenen del flux del programa retornar un valor o un altre.

```

def bool_a_binari(logic):
    if logic == True:
        return 1
    else:
        return 0
>>> curs.bool_a_binari(True)
1

```

#### Exemple 18: Diversos return

Podem especificar valors per defecte de la funció, i donar valors concrets al cridar-la. Per defecte anirà en ordre la crida, però podem especificar arguments en la crida.

```

def potencia (x = 2, y = 3):
    return x**y
>>> potencia(2)
8
>>> potencia(y=5, x = -4)
-1024

```

#### Exemple 19: Valors per defecte i ordre dels paràmetres

### Exercici VIII

Calcular els primers 1000 nombres prims emprant el sedàs d'Eratòstenes.

### Exercici IX

```
def mitjana(llista):
    if llista is not list:
        return None
    else:
        ret = 0
        for i in llista:
            ret = ret + i
        return ret/len(llista)
```

## Exercici X

Definir la següent funció:

$$\sum_{i=0}^n x^i = 1 + x + x^2 + x^3 + \dots + x^n$$

## Recursivitat

En Python podem definir funcions recursives. Una funció recursiva és aquella que es crida a si mateixa. Els beneficis de la recursivitat és l'expressivitat, tot i que computacionalment no sempre ens interessarà.

```
def factorial(n):
    if(n <= 0):
        return 1
    else:
        return n*fact(n-1)
```

Exemple 20: Càlcul recursiu d'un nombre factorial

## 5.2 Funcions lambda, map i filter

### 5.2.1 Lambda

Les funcions lambda són funcions anònimes que utilitzen un constructor anomenat `lambda`. Tenen un sintaxi pròpia i no les podem cridar ni definir de la mateixa forma que les funcions corrents de Python, sinó que té la seva pròpia sintaxis i semàntica.

```
f = lambda x: math.sin(x)
print(f(3))
g = lambda x,y: -1 * x**y
print(g(1,1))
```

Exemple 21: Funcions anònimes

### 5.2.2 Map

La funció `map()` ens és útil quan volem aplicar la mateixa operació a una llista de valors. Per exemple quan volem arrodonir una llista de números. Li passem com a paràmetre el nom de la funció i la llista de valors. Per això, els paràmetres de la funció seran:

```
>>> ll = list(range(1,10))
>>> def cub(x): return x**3
>>> list(map(cub,ll))
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Exemple 22: Calcular el cub d'una llista

### 5.2.3 Filter

La funció `filter()` descarta els valors d'una llista quedant-se només amb aquells valors que la funció retorna. Això ens serà útil quan volem descartar valors que no ens interessin d'una llista. El que determinarà si es retorna o no el valor és si la funció d'avaluació retorna cert o fals.

```
def fil(x):
    if(x < 0):
        return x
>>> v = list(range(-10,10))
>>> list(filter(fil,v))
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
```

Exemple 23: Retornar només valors negatius

Podem combinar les funcions anònimes i les funcions `filter()` i `lambda()`. Si en comptes de passar com a paràmetre el nom de la funció declarem la funció `lambda`, llavors tindrem una funció més expressiva.

```
>>> v = list(range(-10,10))
>>> list(filter(lambda x: x < 0,v))
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
```

Exemple 24: Retornar només valors negatius

### **Exercici XI**

Realitzar l'exercici anterior de càlcul de nombres primers emprant la funció `filter()`.

### **Exercici XII**

Definir la funció següent:

$$\sum_{i=0}^n x^i = 1 + x + x^2 + x^3 + \dots + x^n \quad (|x| < 1)$$

### **Exercici XIII**

Defineix l'algoritme del triangle de Tartaglia i executa'l per a 100 nivells.

### **Exercici XIV**

Calcular els factors per a qualsevol nombre donat

# Capítol 6

## Escriptura i lectura de fitxers

La terminal ens ofereix una interfície volàtil per als nostres càlculs. De vegades estem interessats en guardar fitxers amb els resultats del nostre processament, o bé carregar fitxers per a llegir l'entrada del nostre sistema.

### 6.1 Lectura de fixers

Per a obrir un fitxer emprarem la funció `open()` i per a tancar la funció `close()`. Al obrir hem d'escollir quina serà la interacció que tindrem amb el fitxer. Tenim diverses modes

- **a**: Afegeix contingut al final del fitxer.
- **r**: Només lectura.
- **w**: Sobreescriure el fitxer existent amb nou contingut.

Al obrir el primer paràmetre serà la ruta de l'arxiu i el segon el mode d'obertura. És important que sempre que s'obri un arxiu es tanqui, sinó el nostre descriptor de fitxer quedarà obert pel sistema operatiu.

```
f = open('arxiu.txt', 'r')  
f.close()
```

Un cop hem obert el fitxer podrem llegir. La lectura del fitxer anirà avançant a mesura que nosaltres anem obtenint línies de text. Tenim diverses maneres de llegir el fitxer. Podem utilitzar les funcions:

- `read()`: per a llegir tot el fitxer en una cadena de caràcters.
- `readline()`: per a llegir línia per línia. Haurem de cridar-la mentre no arribem al final del fitxer.
- `readlines()`: per a llegir tot en una llista.

Així doncs pel següent text guardat a *arxiu.txt*

```
La primera
la segona
i la tercera
```

Si utilitzem la funció `read()` tot el contingut serà una cadena de text.

```
>>> f = open('arxiu.txt', "r")
>>> text = f.read()
>>> print(text)
La primera
la segona
i la tercera
```

Amb `readline()` haurem de cridar diverses vegades la funció.

```
>>> text = f.readline()
>>> print(text)
La primera
>>> text = f.readline()
>>> print(text)
la segona
>>> text = f.readline()
>>> print(text)
i la tercera
```

La funció `readlines()` retornarà el contingut en una llista on cada element és una línia del text.

```
>>> text = f.readlines()
>>> print(text)
['La primera\n', 'la segona\n', 'i la tercera']
```

## 6.2 Escriptura de fitxers

L'escriptura és realitza en una forma similar que la lectura. Haurem d'utilitzar el mode `a` o `r`. Només tenim dues funcions en aquest cas.

- `write()`: Escriu directament tot el text
- `writelines()`: Escriu un text en format llista.

```
file = open("newfile.txt", "w")
file.write("hello world in the new file\n")
file.writelines(["linea n\n", "n+1\n"])
file.close()
```

### Exercici XV

Utilitzant el fitxer anterior *arxiu.txt* buscar la següent subcadena de caràcters programant una funció: " *a t*".





# Capítol 7

## Excepcions

### 7.1 Gestionar excepcions

Les excepcions són una manera de tractar casos en els quals Python considera que hi ha hagut una excepció de l'execució o interpretació. Per exemple una divisió entre 0 ens donarà una excepció de tipus `ZeroDivisionError`. Python parará l'execució del nostre programa i ens avisarà d'on ha succeït i quina mena d'error és.

```
>>> 2/0
-----
ZeroDivisionError      Traceback (most recent call last)
/home/albert/<ipython-input-2-897f73788bb0> in <module>()
----> 1 2/0

ZeroDivisionError: integer division or modulo by zero
```

A continuació es presenta una llista dels errors de Python extret de la documentació oficial de Python. En aquesta llista hi ha un ordre jeràrquic i algunes excepcions hereten de prèvies. Per exemple, dintre dels errors de sintaxis trobem l'excepció de sagnat.

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
```

```

+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        |   +-- BufferError
        |   +-- ArithmeticError
        |       |   +-- FloatingPointError
        |       |   +-- OverflowError
        |       |   +-- ZeroDivisionError
        |   +-- AssertionError
        |   +-- AttributeError
        |   +-- EnvironmentError
        |       |   +-- IOError
        |       |   +-- OSError
        |       |       +-- WindowsError (Windows)
        |       |       +-- VMSError (VMS)
        |   +-- EOFError
        |   +-- ImportError
        |   +-- LookupError
        |       |   +-- IndexError
        |       |   +-- KeyError
        |   +-- MemoryError
        |   +-- NameError
        |       |   +-- UnboundLocalError
        |   +-- ReferenceError
        |   +-- RuntimeError
        |       |   +-- NotImplementedError
        |   +-- SyntaxError
        |       |   +-- IndentationError
        |       |       +-- TabError
        |   +-- SystemError
        |   +-- TypeError
        |   +-- ValueError
        |       +-- UnicodeError
        |           +-- UnicodeDecodeError
        |           +-- UnicodeEncodeError
        |           +-- UnicodeTranslateError
    +-- Warning

```

```

    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning

```

Per a capturar una excepció utilitzarem les funcionalitats `try & except`. Per exemple podem dividir tots els nombres enters en el rang  $[10, -10]$  entre sí mateixos i esperar a que hi hagi una divisió entre zero.

```

for i in range(10, -10, -1):
    try:
        i/i
    except ZeroDivisionError:
        print ("Divisio entre 0")

```

Exemple 25: Divisió entre zero

També podem considerar diversos tipus d'error:

```

for i in range(10, -10, -1):
    try:
        i/i
    except TypeError:
        print ("Type Error")
    except ValueError:
        print ("Value Error")
    except:
        print ("Un altre error")

```

Exemple 26: Diversos errors

## 7.2 Llençar excepcions

Tot i que en els exemples previs hem mostrat l'error mitjançant la funció `print()`, la manera correcta és llençar una excepció, a no ser que ens interessi que l'excepció no aturi el nostre codi. Mitjançant la comanda `raise` nosaltres

podem llençar les nostres pròpies excepcions. Podem especificar el tipus d'error i quin és el missatge que volem que es mostri al usuari del nostre programa.

```
for i in range(10,-10,-1):  
    if(i==0):  
        raise ValueError("No s'accepta el valor 0")  
    i/i
```

Exemple 27: Llençar errors amb raise

### Exercici XVI

Buscar tots els casos pels que és cert l'últim teorema de fermat per a  $n \leq 10$  i valors de  $z \leq 10000$ .

$$x^n + y^n = z^n$$

## Capítol 8

# Mòduls i llibreries

Els mòduls ens permeten empaquetar les nostres funcions i organitzar-les modularment. Quan hem vist la llibreria `math` el que hem realitzat és una importació del mòdul. Quan utilitzem `import` fem que sempre hagem d'incloure el nombre del mòdul abans d'accedir als mètodes o atributs. Al incloure-les és un bon patró fer-ho de la manera següent.

```
import math
```

També podem crear un alias o abreviació per a treballar de manera més ràpida amb el mòdul utilitzant l'opció `as`. Així doncs amb la següent forma accediríem als mètodes amb `mt`.

```
import math as mt  
mt.sin(2)
```

És millor utilitzar `import` que `from modul import` degut a que la segona manera ens importarà funcions que quan tinguem projectes grans no sabrem d'on provenen i de vegades podem tenir col·lisions de noms, que és quan dues funcions o variables tenen el mateix nom i una sobreescriu l'altre.

```
from math import *
```

Com hem vist amb el mòdul `math` nosaltres podem importar el mòdul i després accedir a les funcions membre i als atributs mitjançant el punt

.'. Nosaltres també podem crear el nostre mòdul de manera senzilla. Podem desar en un arxiu un conjunt de funcions i una crida a aquestes per a que s'executin. Guardem les següents funcions dins d'un fitxer anomenat `funcions.py`.

```
def quadrat(x):  
    return x**2  
def cub(x):  
    return x**3
```

Un cop creat el fitxer nosaltres podem importar-lo emprant el nom del fitxer. Sempre buscarà en els fitxers dintre del nostre directori de treball.

```
>>> import funcions  
>>> funcions.cub(3)  
27
```

## 8.1 NumPy

NumPy és una llibreria de Python per a treballar amb arrays multidimensionals i especialment amb arrays grans. Implementa funcions d'àlgebra i està implementada a baix nivell per a un processament ràpid. Per a poder treballar amb NumPy haurem d'importar el mòdul.

```
>>> import numpy  
>>> import numpy as np  
>>> from numpy import *
```

Per a la sessió assumirem que importem el mòdul com `np` que degut al seu us intensiu és més breu i més còmode d'usar.

### 8.1.1 Declaració d'arrays

Amb NumPy declarem de la següent forma un array:

```
>>> arr = np.array([[1,2],[3,4]])
>>> arr
array([[1, 2],
       [3, 4]])
```

Exemple 28: Inicialització d'array amb NumPy

Per defecte el tipus de les dades serà detectat automàticament per Numpy. Podem comprobar el tipus de dades emprant l'atribut membre **dtype**.

```
>>> arr.dtype
dtype('int64')
```

Els tipus de dades estan homogeneïtzats. Així doncs si declarem totes les variables com a enters i una d'elles com a real, llavors tots els nombres seran reals.

```
>>> arr = np.array([[1,2],[3,4.3]])
>>> arr.dtype
dtype('float64')
>>> arr
array([[ 1.,  2.],
       [ 3.,  4.3]])
```

Al contrari que en les llistes de Python tots els tipus han de ser del mateix tipus. També podem especificar el tipus **int**, tant com a segon paràmetre, com utilitzant la declaració implícita **dtype=int**.

```
>>> arr = np.array([[1,2.9],[3,4.3]], dtype=int)
>>> arr.dtype
dtype('int64')
>>> arr
array([[1, 2],
       [3, 4]])
```

També podem especificar el tipus **complex**.

```
>>> arr = np.array([[1,2.9],[3,4.3]], dtype=complex)
>>> arr.dtype
```

```
dtype('complex128')
>>> arr
array([[ 1.0+0.j,  2.9+0.j],
       [ 3.0+0.j,  4.3+0.j]])
```

A continuació llistem els diversos tipus de dades que podem utilitzar a NumPy. Depenent de les necessitats del nostre programa utilitzarem un o altre.

- bool
- complex64
- complex128
- complex256
- float32
- float64
- float128
- int8
- int16
- int32
- int64
- uint8
- uint16
- uint32
- uint64

La funció `zeros()` ens permet inicialitzar una matriu. També li podem especificar el tipus de dades amb el que volem treballar.



```

>>> z = np.zeros((2,2), dtype=int)
>>> z
array([[0, 0],
       [0, 0]])
>>> z.dtype
dtype('int64')
>>> z = np.zeros((2,2), dtype=complex)
>>> z
array([[ 0.+0.j,   0.+0.j],
       [ 0.+0.j,   0.+0.j]])
>>> z = np.zeros((2,2), dtype=float)
>>> z
array([[ 0.,   0.],
       [ 0.,   0.]])

```

Exemple 29: Inicialització amb funcio zeros()

També tenim la funció anàloga **ones**.

```

>>> o = np.ones((3,3), np.float128)
>>> o
array([[ 1.,   1.,   1.],
       [ 1.,   1.,   1.],
       [ 1.,   1.,   1.]])

```

Exemple 30: Inicialització amb funcio ones()

També es pot donar el cas en que tenim una matriu **M** i volem crear una matriu amb les mateixes dimensions que **M** i inicialitzar-la amb zeros o uns. Per això tenim les funcions **ones\_like** i **zeros\_like**, que ens copien les dimensions d'una matriu donada.

```

>>> m = np.array([[1,2],[3,4],[5,6]])
>>> m
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> z = np.zeros_like(m)
>>> z
array([[0, 0],
       [0, 0],
       [0, 0]])
>>> o = np.ones_like(m)
>>> o
array([[1, 1],
       [1, 1],
       [1, 1]])

```

Exemple 31: Zeros like i ones like

Per a crear llistes tal i com fins ara fèiem emprant la funció **range** podem utilitzar la funció de NumPy **arange**

```

>>> np.arange(20,-20,-2)
array([ 20,  18,  16,  14,  12,  10,   8,   6,   4,   2,
        0,  -2,  -4,  -6,  -8, -10, -12, -14, -16, -18])

```

Degut a problemes de la representació de les dades, casos en els que utilitzem nombres reals pot succeir que es propaguin errors de representació i que el nombre d'elements retornat no sigui previsible. Per això utilitzem la funció **linspace** on primer especifiquem

```

>>> np.linspace(-2,2,6)
array([-2. , -1.2, -0.4,  0.4,  1.2,  2. ])

```

També podem emprar la funció **linspace** per a crear seqüències de nombres complexos.

```

>>> np.linspace(-3+4j,2+7j,9)
array([-3.000+4.j , -2.375+4.375j, -1.750+4.75j, -1.125+5.125j,
       -0.500+5.5j,  0.125+5.875j,  0.750+6.25j,  1.375+6.625j,
       2.000+7.j   ])

```

Podem crear una llista d'elements emprant la funció `linspace` i després convertir-la en una matriu emprant la funció `reshape`. Aquesta funció també redimensiona amb altres matrius.

```
>>> arr = np.linspace(-2,2,6)
>>> arr.reshape((2,3))
array([[ -2.  ,  -1.2,  -0.4],
       [  0.4,   1.2,   2. ]])
```

Exemple 32: Combinació de `linspace()` i `reshape()`

Si volem construir una matriu identitat cridem la funció la funció `identity`. Així doncs, per a crear la següent matriu

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Necessitem saber les dimensions de la matriu, que al ser la matriu identitat una matriu quadrada només és un paràmetre. En el cas previ 3.

```
>>> np.identity(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

### 8.1.2 Accés a les dades

Podem accedir a les dades tal i com accedíem a llistes bidimensionals, o amb els índex separats per comes a dins del claudators.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,2]
3.0
>>> a[0][2]
3.0
```

Hem de tindre en compte un concepte molt important en la còpia de les arrays, i es que al realitzar una assignació entre variables de numpy no estem copiant, sinó que estem creant una altre referència al mateix objecte.

```
>>> x = np.zeros((3,3))
>>> x
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> y = x
>>> y[1,1] = -1
>>> x
array([[ 0.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  0.]])
>>> y
array([[ 0.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  0.]])
```

Per evitar-ho el que fem és us de la funció `copy`

```
>>> x = np.zeros((3,3))
>>> x
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> y = x.copy()
>>> y[1,1] = -1
>>> x
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> y
array([[ 0.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  0.]])
```

Exemple 33: Còpia per valor mitjançant la funció `copy()`

Podem comprovar si un element es troba a dins d'una llista mitjançant la funció `in`

```
>>> 4 in a
True
>>> -4 in a
False
```

Trobar elements dins d'un array. Fem servir la funció `where`

```
>>> x = numpy.array([1,0,2,0,3,0,4,5,6,7,8])
>>> numpy.where(x == 0)[0]
array([1, 3, 5])
```

### 8.1.3 Operacions amb matrius

Per la suma de tots els elements de la matriu utilitzem la funció `sum` i per a realitzar el producte la funció `prod`

```
>>> o = np.ones((3,4), complex)
>>> o.sum()
(12+0j)
>>> o.prod()
(1+0j)
```

Aquestes funcions estan implementades a baix nivell amb C per a la seva rapidesa, i són molt més ràpides que si nosaltres declarem un bucle en Python que realitzi aquesta mateixa tasca. També tenim funcions per a trobar el mínim i el màxim a dins de la funció amb les funcions `min` i `max`, i per a saber els seus índex dintre de l'array emprarem les funcions `argmin` i `argmax`.

```
>>> m
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> m.min()
1
>>> m.max()
6
```

```
>>> m.argmin()
0
>>> m.argmax()
5
```

NumPy també inclou funcions estadístiques per a realitzar càlculs amb els arrays. Tenim per a la mitja la funció `mean` per a la variança la funció `var` i per a la desviació estàndard la funció `std`

```
>>> m
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> m.mean()
3.5
>>> m.var()
2.9166666666666665
>>> m.std()
1.707825127659933
```

La funció `diagonal` ens retorna els valors de la diagonal de la matriu.

```
>>> i = np.identity(3)
>>> i.diagonal()
array([ 1.,  1.,  1.])
```

Podem concatenar arrays utilitzant la funció `hstack`

```
np.hstack(v1, v2)
```

Per a multiplicar matrius utilitzem la funció `dot` i per a redimensionar una matriu utilitzem `reshape`

```
>>> x = np.arange(9).reshape((3,3))
>>> y = np.arange(3)
>>> np.dot(x,y)
array([ 5, 14, 23])
```

## 8.2 Matplotlib

Matplotlib proporciona una llibreria que permet graficar conjunts de valors amb diferents formes de representació. És utilitzada en l'àmbit científic i busca ser una alternativa a altres paquets com Matlab. Matplotlib també té la forma acceptada com estàndard *de facto* d'importar el mòdul al igual que Numpy. En aquest cas es crida `plt`. El mòdul que cridem és en veritat un submòdul que s'anomena `pyplot`.

```
import matplotlib.pyplot as plt
```

Per a tots els exemples que mostrarem assumirem que el mòdul NumPy i matplotlib estan importats com `np` i `plt`

### 8.2.1 Aproximació a Matplotlib

Per fer a generar els gràfics sempre haurem de cridar a una funció específica per a cada gràfic. El gràfic de línies s'utilitza mitjançant la funció `plot()` com es mostra a l'exemple 01. La funció `show` s'encarrega d'obrir una nova finestra que

```
>>> plt.plot([1,2,3,4,5])  
>>> plt.show()
```

Exemple 34: Plot sencill

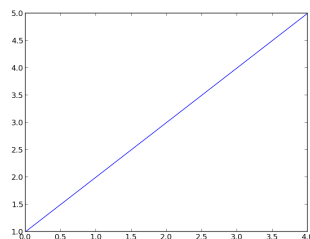


Figure 8.1: exemple plot simple

No només podem utilitzar les llistes pròpies de Python, sinó que podem utilitzar funcions i arrays de Numpy per a mostrar-les tal i com es mostra a l'exemple 02.

```
np.arange(0, 2*np.pi, 0.1);  
y = np.sin(x)  
plt.plot(x, y)
```

Exemple 35: Sinusoidal amb matplotlib

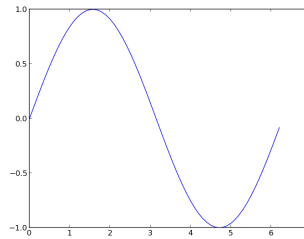


Figure 8.2: exemple sinusoidal

Si no volem tindre que cridar la funció `show()` cada cop que volem fer un `plot`, llavors haurem d'activar el mode interactiu amb la funció `ion()` (interactive on). Aquest mode és molt útil quan estem treballant amb una terminal i no volem cridar la funció `show` cada cop que volem veure els resultats.

```
plt.ion()
```

Podem apagar-lo amb la funció anàloga `iof`.

```
plt.ioff()
```

Per a tots els exemples que mostrarem assumirem que el mode interactiu està desactivat. Podem comprovar l'estat d'aquest mode mitjançant la funció

```
plt.isinteractive()
```

## 8.2.2 Gràfics

Abans hem vist la funció `plot` per realitzar gràfiques simples. A continuació veurem diverses funcions de matplotlib per a mostrar diferents menes de gràfics.



### 8.2.3 Scatter

En el cas de plot nosaltres tenim una distribució que va avançant seqüencialment. Quan volem repartir punts aleatòriament per l'espai utilitzarem la funció `scatter`. La funció rep dos paràmetres, l'eix de les x i de les Y

```
plt.scatter([2.5,2,3],[3,2,1])  
plt.show()
```

Exemple 36: Funcio scatter

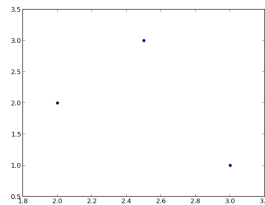


Figure 8.3: exemple scatter

### 8.2.4 Gràfics de barres

Pels gràfics de barres es crida a la funció `bar` i es passen els dos eixos (X,Y) com a paràmetre de la funció. També existeix una altre funció per barres que es específica per a divisions de colors o reparticions de valors tal i com histogrames que s'anomena `hist`.

```
plt.bar([1,2,3],[4,9,2])
```

Exemple 37: Gràfic de barres

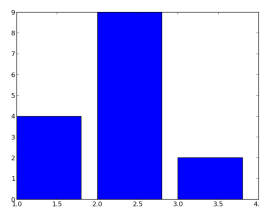


Figure 8.4: gràfic de barres

### 8.2.5 Gràfics pastís

Aquest gràfic ens sumarà tots els valors i cada part agafarà la part de l'angle que li correspongui.

```
plt.pie([1,2,3,4,5,6])
```

Exemple 38: Gràfic pastís

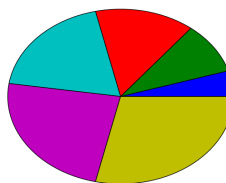


Figure 8.5: gràfic pastís

### 8.2.6 Gràfics amb fletxes

Aquests gràfics són utilitzats amb sistemes d'equacions diferencials o en valors de magnituds repartits per l'espai. Representen un vector amb una direcció i una llargària. Necessitem dues matrius, una amb l'eix de les X i una altre amb l'eix de les Y de les forces. Les fletxes van des de una coordenada (0,0) que està indicada en el punt de la matriu, al punt on està especificat el valor

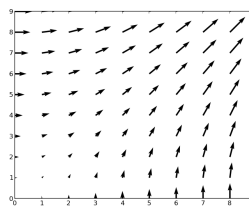


Figure 8.6: exemple gràfic fletxes

```
X, Y = np.mgrid[0:10, 0:10]  
plt.quiver(X, Y)
```

Exemple 39: Gràfic fletxes

### 8.2.7 Guardar gràfics en arxiu

Matplotlib permet guardar les gràfiques que hem realitzat dins d'una imatge. Per a poder guardar-la especifiquem la ruta del fitxer i el nom de la imatge

```
plt.savefig("exemple.png")  
plt.savefig("exemple.pdf")
```

## 8.3 SciPy

SciPy és una llibreria de Python que implementa un conjunt de funcions matemàtiques. A diferència que NumPy integra rutines numèriques com per exemple funcions per treballar amb vectors o manipulació d'imatges. La seva finalitat es proveir a la comunitat científica d'un ventall de funcionalitats per a processament de dades. Normalment trobarem SciPy importat de la següent manera

```
import scipy as sp
```

SciPy està organitzada en un conjunt de subpackets:

- *cluster* Algoritmes de clustering
- *constants* Constants matemàtiques i físiques
- *fftpack* Rutines de la transformada ràpida de Fourier
- *integrate* Integració i equacions diferencials ordinaries
- *interpolate* interpolació
- *io* Entrada i sortida
- *linalg* Àlgebra linial
- *ndimage* Processament d'imatges N-dimensionals
- *odr* Regressió de distància ortogonal.

- *optimize* Optimització i cerca d'arrels.
- *signal* Processament de senyal.
- *sparse* Matrius disperses.
- *spatial* Algoritmes i estructures de dades de problemes espaials.
- *special* Funcions especials.
- *stats* Funcions i distribucions estadístiques.
- *weave* Funcionalitats per integrar codi en C/C++ en el codi Python.

A la pàgina Documentació de Scipy podrem trobar les referències i descripcions de les funcionalitats.

```
>>> import scipy.optimize as optimize
>>> sp.info(optimize.fmin)
```

### 8.3.1 Exemple de SciPy

Per a veure un exemple de les moltes funcionalitats que proveu SciPy implementarem un codi utilitzant el paquet `linalg` i veurem com NumPy i SciPy es combinen.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5],[6]])
>>> b
array([[5],
       [6]])
>>> linalg.inv(A).dot(b) #slow
array([[ -4. ],
       [  4.5]])
>>> A.dot(linalg.inv(A).dot(b))-b #check
```

```
array([[ 8.88178420e-16],
       [ 2.66453526e-15]])
>>> np.linalg.solve(A,b) #fast
array([[ -4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A,b))-b #check
array([[ 0.],
       [ 0.]])
```



# Capítol 9

## iPython Notebook

Amb Python podem crear quaderns de codi i compartir-los amb qualsevol persona. Això és possible gràcies a iPython Notebook. És un mòdul de iPython que es penja a la web i es pot visualitzar des d'un navegador. És emprat per a docència, tutorials, quaderns de treball d'investigadors, etc. Podem penjar els nostres codis de forma lliure amb GitHub o a la nostra pàgina web persona i visualitzar-ho en línia a Nbviewer. A continuació és llisten un conjunt d'enllaços a exemples de iPython Notebooks.

- QuTiP lecture: Single-Atom-Lasing
- CFD Python: 12 steps to Navier-Stokes
- Aerodynamics-Hydrodynamics with Python
- Anàlisi d'ones i la transformada de Fourier
- Xarxes neurals
- (Llista completa d'exemples)

Per a executar-lo haurem de cridar a `ipython` especificant-li que volem que obri el `notebook` de la següent manera.

```
$ ipython notebook
```

Si tenim els nostres quaderns en un directori concret li podem especificar la ruta del directori de treball

```
$ ipython notebook --notebook-dir=/home/exemple
```

## 9.1 Treballant amb el Notebook

Tindrem dos tipus de cel·les: `code` i `markdown`. En una inclourem codi en Python i a l'altre text en format Markdown. Per a executar el codi en Python haurem de inserir el codi i prémer **Shift + Enter**.

A la columna de l'esquerra de la interfície trobarem principalment les següent seccions:

- Notebook: Crear o obrir quaderns. Sempre que vulguem descarregar abans l'hem de guardar
- Cell: operacions amb les cel·les. La secció **Format** ens permetrà definir el tipus de cel·la.
- Kernel: interrompre o tornar a executar el codi Python.
- Help: ens mostra ajuda i enllaços a projectes de Python. Per a mostrar totes les dreceres de teclat prémer **Ctrl + M + H**

## 9.2 El format Markdown

Markdown és un tipus d'escriptura que ens permet formatar el text. Ens permet crear llistes, diferents mides de lletres, inserir codi LaTeX, etc, escrivint només caràcters especials. La renderització de LaTeX que es mostri dependrà del nostre navegador.

Si volem crear un títol fem el caràcter `#`: Per tal ens transformarà:  
`# Títol`

### Títol

Si volem crear un títol fem el caràcter `##`:  
`## Subtítol`



## Subtítol

Si volem crear un títol emprem el caràcter `###`:

`###` Subsubítol

## Subsubítol

Per a inserir una llista introduïrem el caràcter `*`, `+` o `-`.

`* Element 1`

`* Element 2`

`* ...`

`• Element 1`

`• Element 2`

`• ...`

Sempre que vulguem codi LaTeX haurem d'emperrar al principi i al final de l'expressió el caràcter dolar `$` expressió `$`.

`\sum\limits_{i=1}^n i^2`

$$\sum_{i=1}^n i^2$$

## Exercici XVII

Realitzar un quadern amb iPython Notebook utilitzant NumPy on es descriguin les operacions algebraïques com inversa, multiplicació i on s'insereixi en les cel·les codi LaTeX descrivint el que està realitzant la matriu.

## 9.3 Inserir gràfics a iPython Notebook

iPython Notebook permet la integració d'imatges dintre dels quaderns. Per això podem utilitzar la llibreria matplotlib o bé les funcions `Image`. Per a poder inserir-les haurem d'executar el quadern amb l'opció `--pylab inline`.

`$ ipython notebook --pylab inline`

També podem executar la següent línia dintre d'un quadre d'execució de iPython, per tal de no reiniciar el servidor.

```
%pylab inline
```

Ara podrem executar els exemples anteriors mostrant les imatges integrades com es mostra a la Fig. 9.1.

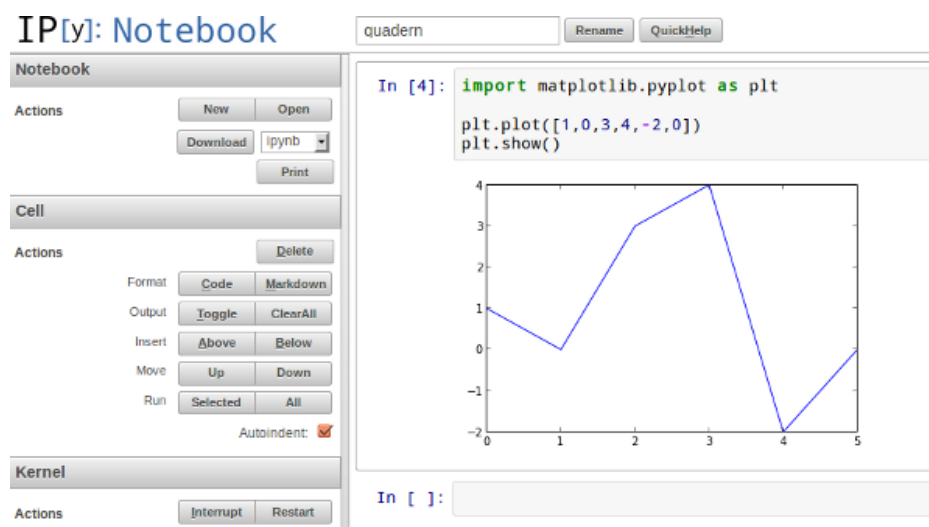


Figure 9.1: Integració d'imatges a iPython Notebook

## Exercici XVIII

Graficar la fórmula del sinus i el cosinus utilitzant NumPy i Matplotlib en un quadern de iPython Notebook.

## 9.4 GitHub

Els nostres quaderns poden ser publicats on-line i compartits a la xarxa. GitHub és un repositori de codi que ens permet compartir arxius i gestionar les versions d'aquests. Podem crear un perfil. Si pugem un quadern a GitHub després podem executar-lo automàticament amb el projecte NBviewer de iPython. Per a pujar els arxius a GitHub primer ens haurem de crear un compte. GitHub és gratuït mentre el nostre treball sigui públic i no excedim certs límits de capacitat. Obrim una terminal

Afegim l'arxiu al repositori.

```
$ git add arxiu.ipynb
```

Especifiquem un missatge a la nova versió del document.

```
$ git commit -m "Arxiu de notebook"
```

Pujem tots els canvis al servidor

```
$ git push
```

Ara tindrem el nostre projecte actualitzat i podrem passar la URL on es troba el nostre fitxer a la pàgina web NBviewer.



# Capítol 10

## Entorn de desenvolupament PyCharm

PyCharm és un entorn de desenvolupament integrat (IDE) per Python multiplataforma (Linux, Mac OS i Windows). Per descarregar PyCharm anem a la pàgina de descàrregues de JetBrains i seleccionem la nostra plataforma. PyCharm té moltes característiques interessants pel desenvolupament ràpid d'aplicacions en Python com autocompletat, autoidentificació, control de versions integrat, depurador, refactoring o suport pel testing.

### 10.1 Creant un projecte

Al crear un projecte hem d'especificar el nom del projecte, que serà el nom de la carpeta; la ruta del projecte dintre del sistema de fitxers i quin intèrpret usarem. Anem a *File - New project* i emplenem el diàleg.

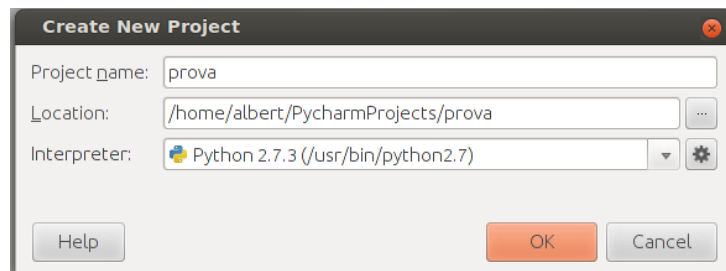


Figure 10.1: Terminal de Python

Fent click al projecte creat li diem *File - New* o cliquem a sobre del projecte amb el botó dret i li diem *New*. En ambdós casos seleccionem *Python file* tal i com es mostra a la Fig. 10.2.

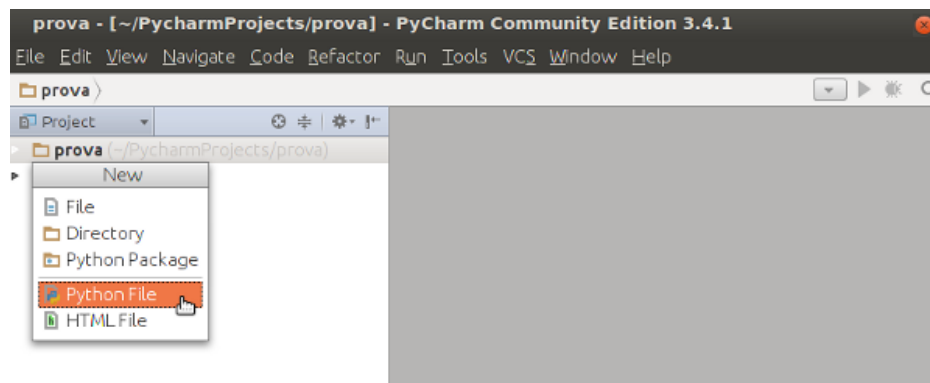


Figure 10.2: Nou fitxer Python

Emplenem el nom del fitxer (Fig. 10.3, que serà l'arxiu *.py* que crearem dins del sistema i que editarem).

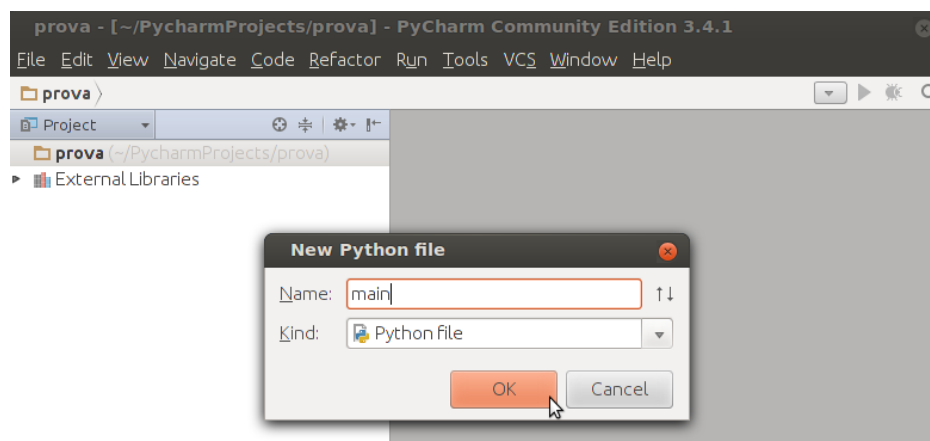


Figure 10.3: Emplenar nom del fitxer

## 10.2 Executant

Tenim diverses possibilitats per executar el nostre codi Python. PyCharm inclou l'opció d'execució des de la IDE, també integra la terminal, des de

la qual podem cridar al codi, o bé, fins i tot, podem seleccionar el codi amb el cursor, clickar amb el botó dret i seleccionar l'opció *Execute Selection in Console*. En l'exemple mostrat a continuació s'ha usat el següent codi

```
import numpy as np

def mitjana(tamany):
    v = np.random.rand(tamany)
    res = np.mean(v)
    return res

print(mitjana(1000))
```

Després d'haver creat el projecte podem programar directament a l'espai d'edició. Pycharm sagnarà el nostre codi automàticament de tal manera que no haguérem d'estar constantment tabulant cada línia. Per a poder executar el codi anem a Run - Run. Aquesta acció executarà el nostre codi

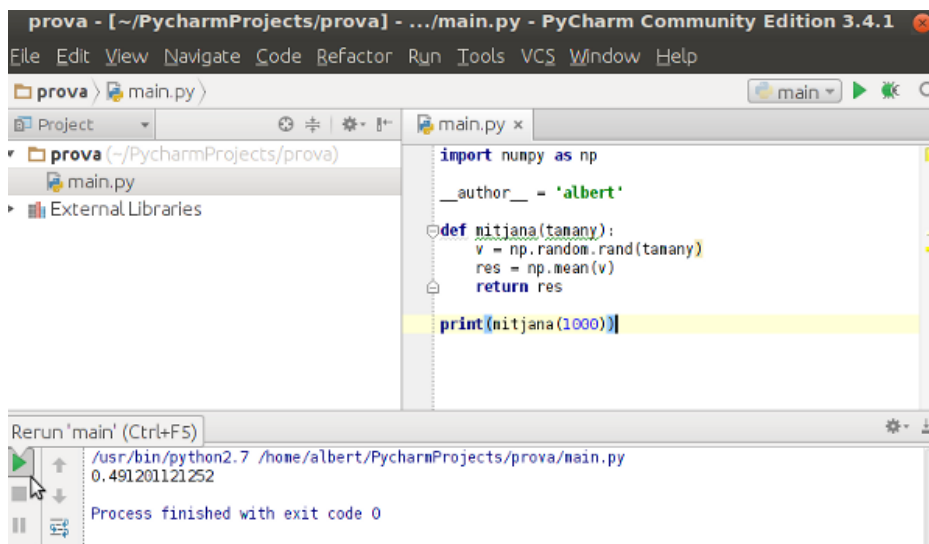


Figure 10.4: Execució d'un programa amb PyCharm

PyCharm integra la terminal iPython per a que podem treballar amb ella mentre estem desenvolupant el nostre codi. Per defecte la ruta del intèrpret serà la nostra carpeta de treball, així que podem importar amb la comanda

import el nostre fitxer i cridar les seves funcions tal i com es mostra a la Fig. 10.5.

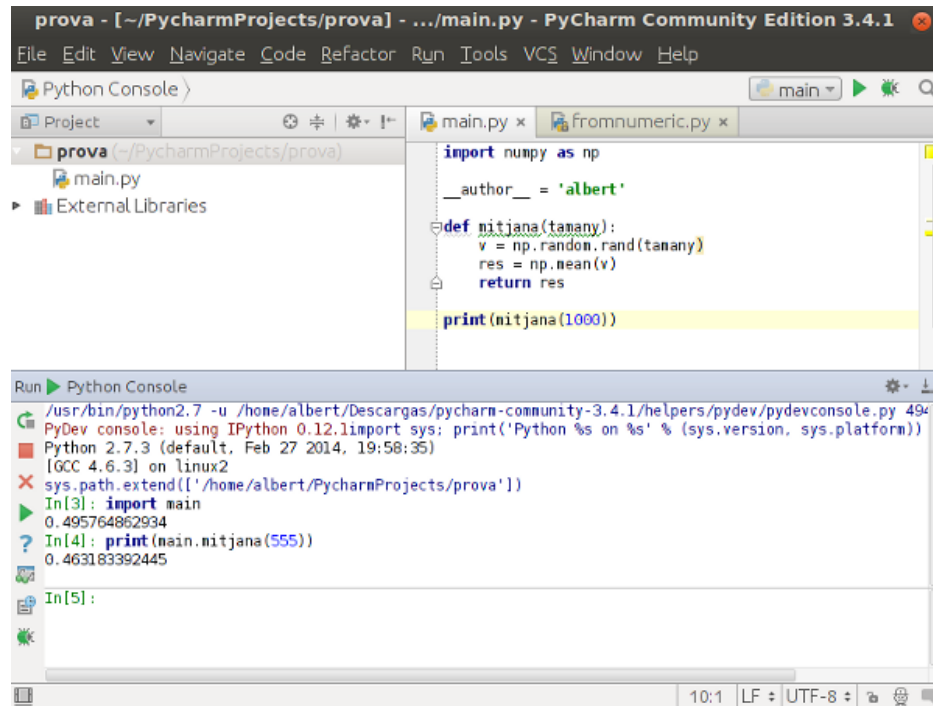


Figure 10.5: iPython integrat en PyCharm

Per a depurar el nostre codi anirem a Run - Debug. Abans d'executar-lo el que hem de fer és introduir un *breakpoint*. Per inserir-lo hem de clicar a l'àrea on es troba el punt vermell a la Fi. 10.6 i seleccionant la línia de codi on volem que s'aturi el codi. Per a avançar en el codi premerem els botons que hi han a l'àrea on es troba el cursor. Al col·locar-nos a sobre veurem la llegenda que ens explica la funcionalitat. Trobarem diverses:

- *Run to cursor*: executa el codi fins la posició del cursor.
- *Step into*: entra dintre de la definició de la funció.
- *Step over*: executar fins la següent línia del mètode on estem
- Quan volem parar l'execució haurem de prémer el botó quadrat vermell de l'esquerra o fer Ctrl + F2.



Trobarem dues pestanyes dintre del marc de depuració de Pycharm. Una s'anomena *Debugger* i l'altre s'anomena *Console*. Quan vulguem veure els resultats que el nostre programa estigui mostren per pantalla haurem d'anar a *Console*.

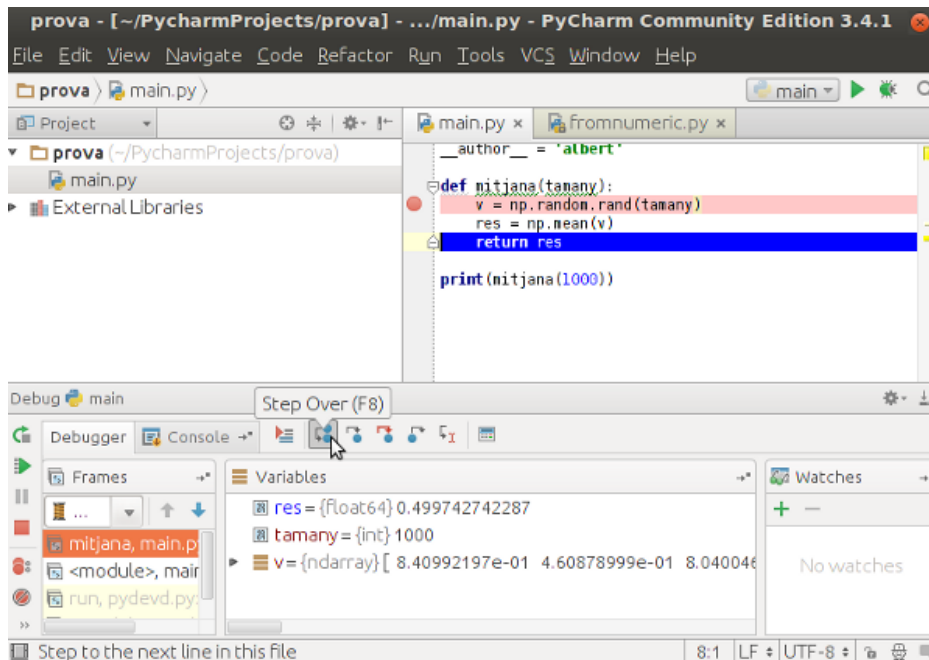


Figure 10.6: Depurant codi Python

## 10.3 Dracres útils

Les dreceres de teclat ens permeten realitzar accions que incrementen el temps de la nostra productivitat quan treballem. D'altra banda, de vegades hi han més de les que es poden recordar. A continuació es mostra un petit llistat que pot ser d'utilitat.

- Per a mostrar totes les opcions per autocompletar utilitzem les tecles **Ctrl + espai**
- Per veure la documentació d'un símbol pressionar **Ctrl + Q**.

- Per navegar a la declaració d'un símbol premem la tecla **Ctrl** i cliquem a sobre del símbol amb el ratolí.
- Quan hi ha la llista d'opcions per l'autocompletat premem **Tab** o **Enter** per a seleccionar-la.
- Quan estem definint una funció i ens trobem amb el cursor entre els parèntesis, al pressionar **Ctrl + P** veurem la paràmetres vàlids.
- Podem canviar entre pestanyes d'edició amb **Ctrl + Tab**.

## **Exercici XIX**

Agafant exercicis programats en sessions anteriors:

- Crear un nou projecte
- Executar des de la IDE
- Executar el codi des de iPython
- Introduir comentaris al nostre codi i després anar a la definició de les nostres pròpies funcions.
- Anar a les definicions de les funcions cridades
- Introduir breakpoints i depurar el codi

# Capítol 11

## SAGE

SAGE és un software de que càlcul matemàtic que busca ser una alternativa oberta a Maple, Mathematica, Magma o MATLAB. Té paquets d'àlgebra, geometria, teoria de nombres, criptografia, computació numèrica i altres àrees. Ofereix una closca per sobre de software como Maxima, NumPy, SciPy, Octave, etc i l'implementa amb Python. D'aquesta manera junta el software existent per tal d'oferir-lo d'una manera única. Inicialment va ser creat per a recerca en matemàtiques i actualment utilitzat també per docència. Ens centrarem en el quadern de SAGE tot i que la instal·lació també inclou una terminal.

### 11.1 Algunes propietats

En el quadern no hem d'importar mòduls tal i com requeríem amb la llibreria *math*, tot i que sí que podem importar els mòduls de Python com Numpy. Donat que SAGE integra diversos paquets, també permet l'execució de diversos llenguatges. Per exemple podem implementar *scripts* en llenguatge Bash especificant abans de la cel·la l'interpret escrivint `%sh` o seleccionant el llenguatge al menú d'interprets.

```
%sh
for i in {1..10}; do echo $i; done
```

Si volem especifica SAGE podem escriure `%sage`.

```
%sage
print("codi en sage")
```

SAGE inclou l'autocompletat. Si anem al quadern, assignem una variable, després introduïm un punt per accedir els mètodes i premem la tecla **Tab** veurem totes les possibilitats. Per exemple el mètode `.N()` ens tornarà el valor numèric. Aquesta opció serà útil degut a que SAGE ens retornarà sempre que pugui la forma simbòlica de les nostres operacions.

```
sage: x = 4
sage: x.
```

Al igual que amb Python podem obtenir més informació d'un element afegit el símbol `?` al final com es mostra a la Fig. 11.1.

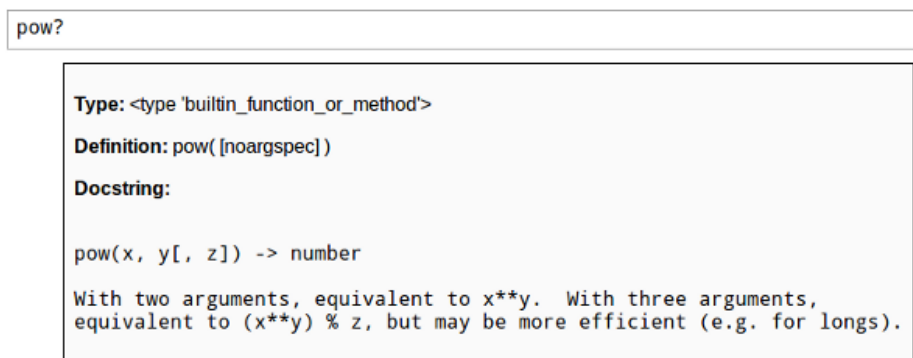


Figure 11.1: Ajuda d'un mètode de SAGE

Els operadors són els mateixos. En el cas d'una divisió en la que perdi precisió en el resultat la deixarà en forma de fracció. Els operadors `**` i `^` són equivalents

```
sage: 2*3
6
sage: (2+3j)-(4-8.3j)
-2.0000000000000000 + 11.300000000000000*I
sage: 3//2
1
```

```

sage: 238/4
119/2
sage: 236/2
118
sage: 3**10
59049
sage: 3^10
59049
sage: 10%3
1

```

Com veiem SAGE també realitza conversions implícites de tipus. Procurarà reduir les expressions fins que no en sàpiga més tal i com veiem amb les fraccions.

Hi han funcions que seran membre de la variable que té el valor tal i com la comprovació de nombres primers o la factorització, però el càlcul del màxim comú divisor (`gcd`) serà una funció a part.

```

sage: x = 3
sage: x.is_prime()
True
sage: y = 1001
sage: y.factor()
7 * 11 * 13
sage: gcd(123213,432432)
3

```

Entre les funcions comuns ens trobem l'arrodoniment. La funció `floor()` arrodoneix a la baixa i `ceil()` a l'alça.

```

sage: floor(3.9)
3
sage: ceil(3.1)
3

```

Trobarem funcions trigonomètriques com `cos`, `sin` `atan`, etc, o altres funcions comuns com `sqrt` incloses també dintre de la llibreria `math` de Python, però en aquest cas hi podem accedir directament. En el cas de

la trigonometria també entén les correspondències a fraccions tal i com el cas.

$$\sin\left(\frac{\pi}{3}\right) = \frac{1}{2}\sqrt{3}$$

```
sage: sin(pi/3)
1/2*sqrt(3)
```

En el cas de SAGE considera tan el logaritme com el logaritme neperià amb base  $e$  que és a la vegada una constant definida en l'entorn. Per a poder utilitzar una base diferent s'ha d'especificar com a segon paràmetre

```
sage: ln(e)
1
sage: log(e)
1
sage: log(125,5)
3
```

## 11.2 Solucionar equacions

Quan volem una igualtat i no una assignació a SAGE utilitzarem l'operand lògic de Python `==`. Per a poder utilitzar una variable per a resoldre equacions haurem d'inicialitzar-la emprant la funció `var()`. Quan volem fer que una variable de funció deixi de ser-ho utilitzem la funció `restore()`

```
sage: 9 == 9
True
sage: 9 <= 10
True
sage: x = var("x")
sage: 3*x - 10 == 5
3*x - 10 == 5
sage: restore('x')
```

SAGE procurarà retorna-nos el resultat de l'expressió en forma simbòlica abans que solucionar en una forma numèrica del conjunt real.

```
sage: solve( cos(x) - exp(x) == 0 , x)
[cos(x) == e^x]
sage: solve( exp(x) - x == 0 , x)
[x == e^x]
```

En el cas de que vulguem una solució numèrica haurem de passar com a paràmetre l'equació i un interval on trobar l'equació.

```
sage: find_root(sin(x) == x, -pi/2 , pi/2)
0.0
sage: find_root(sin(x) == cos(x), pi, 3*pi/2)
3.9269908169872414
```

Podem utilitzar la funció `solve()` també amb diverses variables de la funció.

```
sage: y = var("y")
sage: solve( [3*x - y == 2, -2*x -y == 1 ], x,y)
[[x == (1/5), y == (-7/5)]]
```

## 11.3 Funcions

En SAGE també tenim funcions matemàtiques a part de les funcions que em vist a programació. Podem definir funcions de la següent forma i detectarà quin tipus de funció declarem.

```
sage: f(x) = x*exp(x)
sage: f
x |--> x*e^x
sage: g(x) = (x^2)*cos(2*x)
sage: g
x |--> x^2*cos(2*x)
```

```
sage: h(x) = (x^2 + x - 2)/(x-4)
sage: h
x |--> (x^2 + x - 2)/(x-4)
```

Els caràcters |--> ens diuen que ens trobem amb una funció. Per cridar aquestes funcions les cridem amb el paràmetre numèric.

```
sage: f(1)
e
sage: g(2*pi)
4*pi^2
sage: h(-1)
2/5
```

Podem avaluar límits de funcions passant com a primer paràmetre el nom de la funció i com a segon el valor pel qual volem avaluar.

```
sage: limit(f, x=1)
e
sage: limit(g, x=2)
4*cos(4)
sage: limit(h, x = 4)
Infinity
```

També podem donar valor infinit a l'hora d'avaluar els límits.

```
sage: i(x) = x/x
sage: limit(i, x = Infinity)
1
```

També podem avaluar el límit per la dreta o l'esquerra d'una funció

```
sage: limit(h, x=4, dir="right")
+Infinity
sage: limit(h, x=4, dir="left")
-Infinity
```



Per a realitzar gràfics utilitzarem la funció `plot` i per a mostrar els resultat cridarem a la funció `show`. El resultat l'hem de recollir en un objecte. Implícitament estarem utilitzant llibreries gràfiques com GnuPlot, matplotlib, openmath o surf.

```
f(x) = sin(x)
p = plot(f(x), (x, -pi/2, pi/2))
p.show()
```

Al igual que en Matplotlib podem acumular resultats. En aquest cas haurem d'utilitzar l'operand `+` per a ajuntar els resultats.

```
sage: f(x) = sin(x)
sage: g(x) = cos(x)
sage: p = plot(f(x), (x, -pi/2, pi/2), color='black')
sage: q = plot(g(x), (x, -pi/2, pi/2), color='red')
sage: r = p + q
sage: r.show()
```

Pot realitzar plots paramètrics on declarem una funció i l'interval d'aquesta funció

```
t = var('t')
p = parametric_plot( [3*cos(t), 3*sin(t)], (t, 0, 2*pi) )
p.show()
```

## 11.4 Programació

Al igual que en Python ens trobem les estructures llista o diccionari. També trobem funcions pròpies de Python i mètodes dels elements que ens permeten interactuar amb les estructures de dades.

```
sage: d = {'1':True, '0': False}
sage: d['1']
True
sage: l = [3,5,1,3,5,2]
```

```

sage: l[2]
1
sage: len(l)
6
sage: type(l)
<type 'list'>
sage: l.count(5)
2

```

De la mateixa manera que en altres llenguatges com Bash, podem crear llistes donant un interval de nombres naturals i introduint dos punts entre mig.

```

sage: m = [1..10]
sage: m
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: 3 in [1..10]
True

```

Podem utilitzar funcions que treballen amb llistes tal i com vam veure amb NumPy

```

sage: sum([1,2,3])
6
sage: sum([1..100])
5050
sage: prod([1..4])
24

```

I les funcions especials com `map` o `filter` estan implementades, al igual que les funcions anònimes `lambda`.

```

sage: map(lambda x: x^x, [1..9])
[1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]

```

Les operacions condicionals es declaren igual.

```

z = 3
if z == 2:
    print "Dos"
else:
    print "no es tres"

```

Podem declarar també els bucles `for` i `while` com feiem amb Python.

```

for i in [0..3]:
    sin(float(i))

```

Les funcions venen implementades igualment per per Python.

```

def fun(x):
    print(log(x))
fun(e)

```

## Exercici XX

Per a provar SAGE el que farem serà hackejar una clau privada. A la criptografia actual la seguretat es basa en que és costosa la operació de calcular els factors d'un nombre. La clau pública és el resultat de la multiplicació de dos nombres primers molt grans. Per a obtenir la clau privada a partir de la clau privada haurem de cridar la funció `phi` que realitza l'operació.

$$\phi(k) = (p - 1)(q - 1)$$

On  $p$  i  $q$  són els factors de  $k$ . Tenim l'exponent  $e$  que compleix l'equació  $e = d^{-1} \bmod \phi(k)$  i d'on voldrem extreure  $d$  perquè  $e$  ja ens ve donada.

Haurem de trobar la identitat de Bezout de  $\phi(k)$  i l'exponent, i a partir d'aquí tenim l'invers, que només serà un degut a que els nombres són co-primers.

Així doncs, una clau privada és aquella que coneix els valors  $(d, p, q)$  essent i on la clau pública és un parell de valors  $(e, n)$ .

- exponent: 65537
- clau pública: 3229512820943487928047137895627953  
39960792410378105122617813793019657

Documentació de SAGE

## Exercicis finals

Com a últims exercicis es realitzen un conjunt de propostes per a solidificar els coneixements de Python.

- Solucionar el problema de la motxilla en Python.
- Realitzar un programa que calculi quadrats màgics en una matriu 4x4.
- Realitzar un quadern amb iPython Notebook descrivint les operacions matricials de suma, multiplicació, resta, inversa, transposada i escriure en els comentaris el codi LaTeX que explica les operacions.
- Escriure un programa que rebi un vector, l'ordini i el retorni. A l'interior de la funció hi ha que gestionar totes les possibles excepcions i llençar de pròpies.
- Calcular àrees i dibuixar-les utilitzant el paquet SAGE i comentant què és el que realitza la funció.