

# Introducció a Python:

## Sessió 5: Diversos temes pendents

### 1. Utilització de fitxers

La manera de treballar amb fitxers en Python té moltes semblances a la del llenguatge C.

Per poder utilitzar un fitxer, primer de tot s'ha d'obrir amb la funció nativa **open()**, que retorna un objecte **file**. La classe **file** és una classe predefinida de Python, que ofereix els mètodes necessaris per llegir, escriure i manipular el contingut d'un fitxer.

La funció **open()** té un segon paràmetre opcional per indicar el mode d'accés. Els modes són **r** per llegir (és el valor per defecte), **w** per escriure i **a** per actualitzar el fitxer escrivint a continuació de les dades existents. Per poder llegir i escriure al mateix temps, s'ha d'indicar **r+**.

**El sistema operatiu Windows** distingeix entre arxius amb contingut binari i de text. Per defecte **open()** obre els fitxers en mode text. Si es vol obrir com a fitxer binari s'ha d'afegir la lletra **b** al final del argument de mode.

Modos	Descripción
r	Abre un archivo de sólo lectura. El puntero del archivo se coloca en el principio del archivo. Este es el modo predeterminado.
rb	Abre un archivo de sólo lectura en formato binario.
r+	Abre un archivo para lectura y escritura. El puntero del archivo estará en el principio del archivo.
rb+	Abre un archivo para la lectura y la escritura en formato binario. El puntero del archivo estará en el principio del archivo.
w	Abre un archivo para escribir solamente. Sobrescribe el archivo si el archivo existe. Si el archivo no existe, se crea un nuevo archivo para escritura.
wb	Abre un archivo para escribir sólo en formato binario. Sobrescribe el archivo si el archivo existe. Si el archivo no existe, se crea un nuevo archivo para escritura.
w+	Abre un fichero para escritura y lectura. Sobrescribe el archivo existente si existe el archivo. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura.
wb+	Abre un archivo, tanto para la escritura y la lectura en formato binario. Sobrescribe el archivo existente si existe el archivo. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura.

a	Abre un archivo para anexar. El puntero del archivo se encuentra al final del archivo si existe el archivo. Es decir, el archivo está en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para escritura.
ab	Abre un archivo para adjuntar en formato binario. El puntero del archivo se encuentra al final del archivo si existe el archivo. Es decir, el archivo está en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para escritura.
a+	Abre un fichero para anexar y la lectura. El puntero del archivo se encuentra al final del archivo si existe el archivo. El archivo se abre en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura.
ab+	Abre un fichero para anexar y la lectura en formato binario. El puntero del archivo se encuentra al final del archivo si existe el archivo. El archivo se abre en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura.

## Utilització de fitxers: Activitats

Per realitzar les següents activitats, s'ha de crear previamente un fitxer de text anomenat **lista.txt**, i ubicat a la mateixa carpeta que els scripts de prova. El fitxer ha de contenir 5 línies de text.

### 1. Llegir i mostrar contingut.

Crear un fitxer d'script amb el següent contingut d'exemple:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  #Abrimos el fichero en modo lectura
5  fichero = open("lista.txt", "r")
6
7  contenido = fichero.read()
8
9  print contenido
10
11 fichero.close()
12
```

En executar-lo, ens mostrarà el contingut del fitxer.

### Comentaris:

Quan es deixa d'utilitzar un fitxer, cal utilitzar el mètode **close()**, per alliberar els recursos assignats. Això és especialment important quan s'hagi escrit al fitxer, per assegurar que els canvis es guarden.

El mètode **read()** admet com argument un enter que indica el nombre **màxim** de bytes que ha de llegir.

### Activitats:

Executar el programa. Explicar els resultats obtinguts.

Modificar-lo per llegir només el nombre de bytes que indiqui l'usuari.  
(Consultar a la web l'especificació del mètode **read()**)

## 2. Llegir línia a línia i mostrar contingut.

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  #Abrimos el fichero en modo lectura
5  fichero = open("lista.txt","r")
6
7  print fichero.readline()
8  print fichero.readline()
9  print fichero.readline()
10
11 fichero.close()
12
```

### Comentaris:

El llegeix utilitzant el mètode **readline()**, que retorna una línia sencera.

### Activitats:

Executar el programa. Explicar els resultats obtinguts. Per què hi ha línies en blanc?

Modificar el programa de manera que llegeixi les línies dins d'un bucle while.  
(Quan s'arriba a final de fitxer, **readline()** retorna un string buit "").

## 3. Carregar les línies en una llista i mostrar contingut.

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  #Abrimos el fichero en modo lectura
5  fichero = open("lista.txt","r")
6
7  lista_lineas = fichero.readlines()
8
9  print lista_lineas[1]
10 print lista_lineas[3]
11 print lista_lineas[4]
12
13 fichero.close()
14
```

**Comentaris:**

El llegeix utilitzant el mètode **readlines()**, que retorna una llista on cada element és una línia del fitxer.

**Activitats:**

Executar el programa. Explicar els resultats obtinguts.

Modificar el programa de manera que escrigui les línies en ordre invers.

**4. Llegir, aprofitant que l'objecte file és iterable, i mostrar contingut.**

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  #Abrimos el fichero en modo lectura
5  fichero = open("lista.txt","r")
6
7  for linea in fichero:
8      print linea
9
10 fichero.close()
11
```

**Comentaris:**

El llegeix utilitzant un bucle for que itera sobre l'objecte file.

**Activitats:**

Executar el programa. Explicar els resultats obtinguts. Per què no hi ha cap crida a **read()** ni **readline()**?

**5. Escriure en un fitxer.**

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  un_texto = "Hola mundo"
5  otro_texto = "Adiós mundo"
6
7  # Abrimos el fichero en modo escritura
8  # Si el archivo ya existe, se borrará el contenido
9  fichero = open("nuevo.txt","w")
10
11 fichero.write(un_texto)
12 fichero.write(otro_texto)
13
14 fichero.close()
15
```

**Comentaris:**

S'utilitza el mètode **write()**, que escriu un string en un fitxer. Si s'han d'escriure altres tipus de dades, s'han de passar a string previamente. S'escriu l'string tal com està, sense afegir cap tipus de separador. Si, per exemple, es vol saltar línia, s'ha d'afegir previamente a l'string un `\n`. Per poder escriure, el fitxer ha d'estar obert en mode **w**, **a** o **r+**.

**Activitats:**

Executar el programa. Explicar els resultats obtinguts.

**6. Escriure en un fitxer - 2.**

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  # Abrimos el fichero en modo escritura
5  # Si el archivo ya existe, se borrará el contenido
6  fichero = open("nuevo.txt","w")
7
8  for i in range(1,5):
9      texto = "Línea %i\n" % i
10     fichero.write(texto)
11
12  fichero.close()
```

**Activitats:**

Executar el programa. Explicar els resultats obtinguts.

**7. Escriure una seqüència en un fitxer.**

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  mi_lista = ["Primera","Segunda","Tercera"]
5
6  #Abrimos el fichero en modo escritura
7  fichero = open("nuevo.txt","w")
8
9  fichero.writelines(mi_lista)
10
11  fichero.close()
12
```

### Comentaris:

S'utilitza el mètode **writelines()**, que escriu al fitxer els elements d'una seqüència. Si s'han d'escriure altres tipus de dades, s'han de passar a string prèviament.

S'escriu l'string tal com està, sense afegir cap tipus de separador. Si, per exemple, es vol saltar línia, s'ha d'afegir prèviament a l'string un `\n`.

Per poder escriure, el fitxer ha d'estar obert en mode **w**, **a** o **r+**.

### Activitats:

Executar el programa. Explicar els resultats obtinguts.

## 8. Desplaçaments en un fitxer.

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  # Abrimos el fichero en modo escritura
5  # Si el archivo ya existe, se borrará el contenido
6  fichero = open("nuevo.txt", "w")
7
8  fichero.write("1234567890")
9
10 # movemos el puntero a 5 bytes
11 # desde el principio del fichero
12 fichero.seek(5)
13
14 fichero.write("XXX")
15
16 fichero.close()
```

### Comentaris:

Internament es guarda actualitzada la posició a partir de la qual es realitzarà una lectura o escriptura. Aquest valor **en bytes** es pot saber en qualsevol moment invocant el mètode **tell()**.

Per modificar la posició actual, es pot invocar el mètode **seek()**. Aquest mètode té dos paràmetres:

1. paràmetre obligatori que indica el nombre de bytes de desplaçament
2. paràmetre opcional que indica des d'on es compta el desplaçament:
  - 0 des del principi (valor per defecte)
  - 1 des de la posició actual
  - 2 des del final.

### Activitats:

Executar el programa. Explicar els resultats obtinguts.

## 9. Més funcions.

Crear un fitxer d'script amb el següent contingut d'exemple:

```
1     from sys import argv
2     from os.path import exists
3
4     script, from_file, to_file = argv
5
6     print "Copying from %s to %s" % (from_file, to_file)
7
8     # we could do these two on one line, how?
9     in_file = open(from_file)
10    indata = in_file.read()
11
12    print "The input file is %d bytes long" % len(indata)
13
14    print "Does the output file exist? %r" % exists(to_file)
15    print "Ready, hit RETURN to continue, CTRL-C to abort."
16    raw_input()
17
18    out_file = open(to_file, 'w')
19    out_file.write(indata)
20
21    print "Alright, all done."
22
23    out_file.close()
24    in_file.close()
```

### Activitats:

Analitzar el codi i explicar què fa el programa.

Quines dades es necessari tenir per poder executar-lo?

Quins paràmetres s'han de passar en la crida al script?

Executar el programa. Explicar els resultats obtinguts.

Modificar el programa perquè, en cas d'existir el fitxer de sortida, l'usuari pugui dir si sobreescriure'l, afegir les dades pel final o cancelar la operació.

## 2. Matrius en Python.

En Python no existeix explícitament el tipus de dada "array" o "vector", però es pot simular perfectament amb una llista. De la mateixa manera, no existeixen les matrius, però es poden construir mitjançant **l·listes niades** (traducció discutible de **nested lists** però, per no inventar terminologia, ho deixem així ...).

Una **l·lista niada** no és més que una llista que és un element dins d'una altra llista.

Per exemple:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

la llista "nested", té com a quart element **[10, 20]**, una altra llista (l·lista niada). L'accés als seus elements es pot fer així:

```
>>> elem = nested[3]
>>> elem[0]
10
```

o, de manera més compacta, així:

```
>>> nested[3][1]
20
```

Per representar la matriu

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

podríem escriure

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

Així, per exemple, podríem accedir a la segona fila:

```
>>> matrix[1]
[4, 5, 6]
```

o, directament al segon element (columna) de la segona fila

```
>>> matrix[1][1]
5
```

### Activitats:

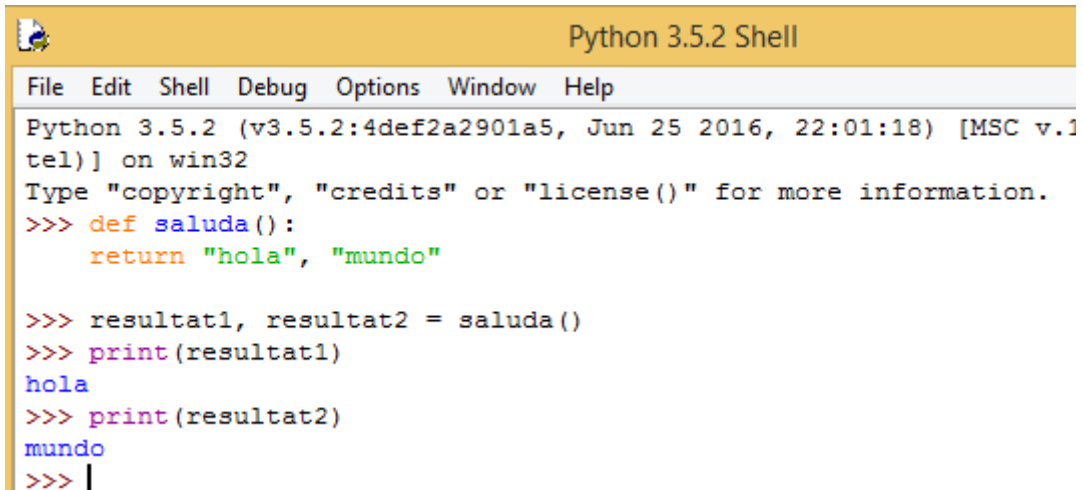
Escriure la funció **producte\_matriu\_escalar (mat, k)** que, donada una matriu d'enters **mat** de **m** files i **n** columnes (utilitzar la funció **len()**) multipliqui tots els seus elements pel nombre enter **k**. (Recordar que una llista és un objecte mutable. Quan es passa com paràmetre d'una funció, es passa per referència)



## 3. Més sobre funcions.

### 3.1. Retorns múltiples.

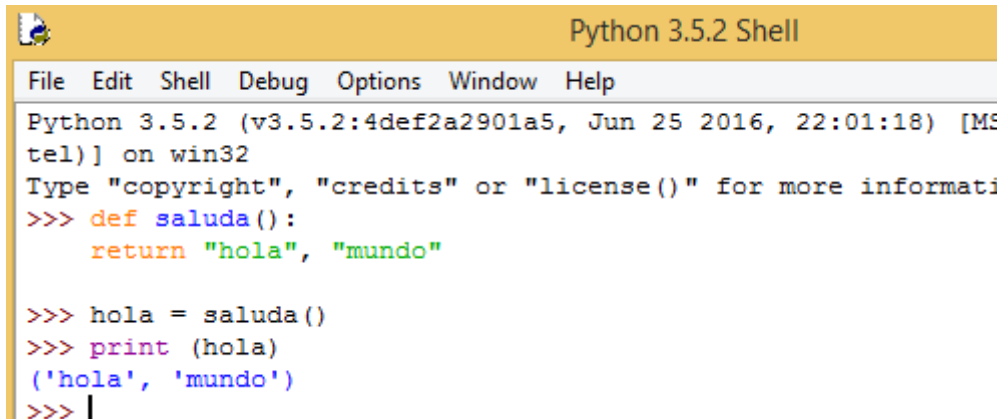
Les funcions en Python poden retornar més d'un valor. Anteriorment s'han vist exemples en els quals els valors retornats es recullen en diferents variables, tantes com valors retornats:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def saluda():
    return "hola", "mundo"

>>> resultat1, resultat2 = saluda()
>>> print(resultat1)
hola
>>> print(resultat2)
mundo
>>> |
```

Pot ser útil saber que, internament, es retorna una tupla amb la seqüència de valors retornats:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MS
tel)] on win32
Type "copyright", "credits" or "license()" for more informati
>>> def saluda():
    return "hola", "mundo"

>>> hola = saluda()
>>> print(hola)
('hola', 'mundo')
>>> |
```

### 3.2. Paràmetres amb valor per defecte.

Python permet donar un valor per defecte a un paràmetre, assignant-lo en la capçalera de la funció. Si en la crida no s'indica cap valor per aquest paràmetre, la funció agafarà el valor assignat per defecte.

Els paràmetres amb valor per defecte s'han d'escriure **després** dels que no en tenen.

```
>>>
>>> def f(a, b, c='A', d='B'):
        return list(str(a) + str(b) + str(c) + str(d))

>>> print (f(1,2))
['1', '2', 'A', 'B']
>>> print (f(1,2,3))
['1', '2', '3', 'B']
>>> print (f(1,2,3,4))
['1', '2', '3', '4']
>>>
>>>
```

#### Activitats:

- Probar l'exemple i explicar els resultats obtinguts.
- Modificar la funció i escriure els paràmetres amb valor per defecte primer. Tornar a reproduir l'exemple i explicar els resultats.

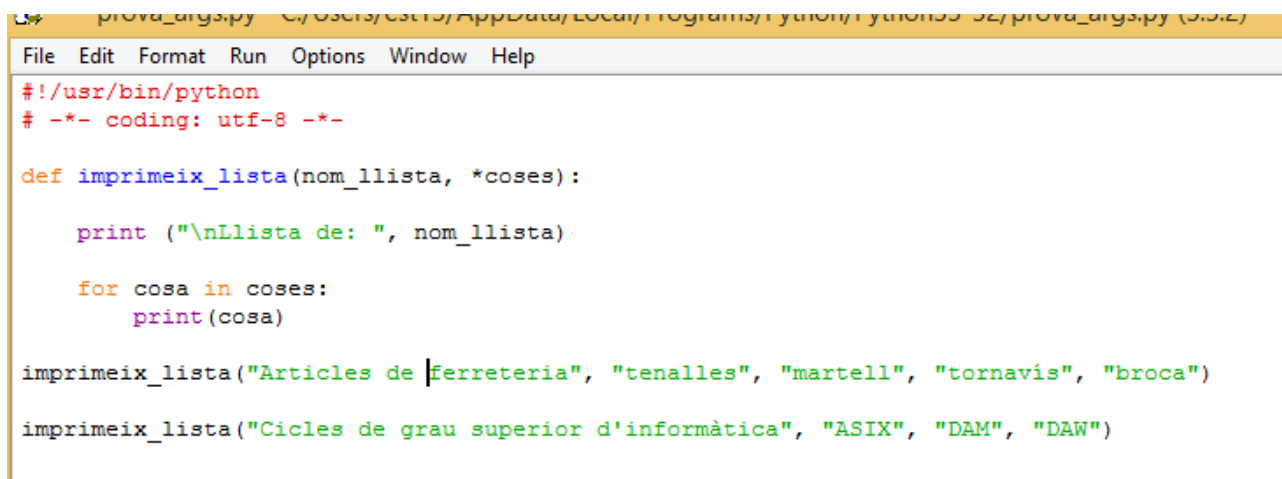
### 3.3. Funcions amb un nombre variable de paràmetres.

Python permet definir funcions que poden rebre en cada crida un nombre diferent de paràmetres.

Una manera de fer-ho és utilitzar els **"args"** (de "arguments"). Un paràmetre **"arg"** es defineix com qualsevol altre, però va **prefixat amb un asterisc**. Així Python interpreta que el paràmetre **és en realitat una llista** de paràmetres. Si hi ha paràmetres fixos, sempre s'han d'escriure abans dels paràmetre args.

La crida a la funció es fa de la manera habitual, indicant primer de tot els paràmetres fixos i després la part variable, que depèn del que volguem fer en cada crida.

Si executem aquest script



```
prova_args.py C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_args.py (3.5.2)
File Edit Format Run Options Window Help

#!/usr/bin/python
# -*- coding: utf-8 -*-

def imprimeix_lista(nom_llista, *coses):

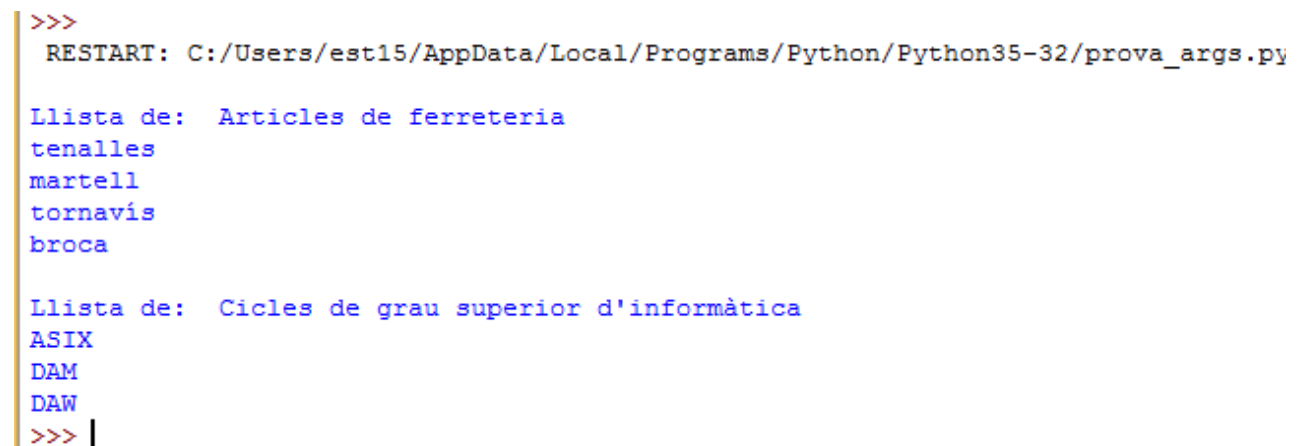
    print ("\nLlista de: ", nom_llista)

    for cosa in coses:
        print(cosa)

imprimeix_lista("Articles de ferreteria", "tenalles", "martell", "tornavis", "broca")

imprimeix_lista("Cicles de grau superior d'informàtica", "ASIX", "DAM", "DAW")
```

Obtindrem



```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_args.py

Llista de:  Articles de ferreteria
tenalles
martell
tornavis
broca

Llista de:  Cicles de grau superior d'informàtica
ASIX
DAM
DAW
>>> |
```

Una altra manera més sofisticada és utilitzar els **"kwargs"** (de "keyword arguments"). Un paràmetre **"kwarg"** va **prefixat de dos asteriscs** i la funció

hi rep **un diccionari** de parells clau-valor.

En la crida s'ha d'indicar primer de tot els paràmetres fixos i després la part variable, on cada paràmetre consta de clau i valor separats pel caràcter '='.

Si executem aquest script

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

def imprimir_dades(nom, **dades):

    print("\nDades de ", nom)

    for clau in dades:
        print (clau + ": " + dades[clau])

imprimir_dades("Antoni", edat = "20", matriculat = "si", beca = "no")
imprimir_dades("Joan", edat = "22", matriculat = "si", beca = "si", import_beca = "550")
```

Obtindrem

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_kwargs.p

Dades de  Antoni
matriculat: si
edat: 20
beca: no

Dades de  Joan
matriculat: si
edat: 22
beca: si
import_beca: 550
>>> |
```

## Activitats:

Probar els exemples i explicar els resultats obtinguts.

**Nota:** Per ser precisos amb la nomenclatura, s'anomenen **paràmetres** (o **paràmetres formals**) els que s'especifiquen en la **definició** d'una funció. En el moment de la **crida**, els valors que s'hi passen, s'anomenen **arguments** o **paràmetres reals**. En la pràctica, quan no hi ha ambigüïtat, és habitual utilitzar paràmetre i argument com a paraules sinònimes, encara que no ho siguin.

### 3.4. Àmbit de les variables i conflictes amb els noms.

L'àmbit d'una variable és la part del codi on és accessible. Les variables definides dins d'una funció s'anomenen **locals**, i les definides al programa principal s'anomenen **globals**.

Les variables **globals** són visibles a tot arreu, dins de qualsevol funció, però les variables **locals** només existeixen durant l'execució de la funció on s'han definit. Quan la funció acaba, les seves variables "queden fora d'àmbit" i no són accessibles:

```
>>>
>>> v_global = "Sóc global"
>>>
>>> def funció ():
    v_local = "Sóc local"
    print("la variable global diu:", v_global)
    print("la variable local diu:", v_local)

>>>
>>> funció()
la variable global diu: Sóc global
la variable local diu: Sóc local
>>>
>>> v_global
'Sóc global'
>>> v_local
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    v_local
NameError: name 'v_local' is not defined
>>>
```

Si tenim un nom de variable **repetit en diferents funcions**, Python considera que són diferents variables, amb àmbits diferents. En un determinat moment, només existeix la variable que pertany a la funció en execució.

Si tenim un nom de variable **repetit en una funció i en el programa principal**, Python torna a considerar que són diferents variables, amb àmbits diferents. Mentre s'executa la funció, la variable global homònima "queda fora d'àmbit".

```
>>>
>>> la_variable = "Sóc global"
>>>
>>> def funció ():
    la_variable = "Sóc local"
    print("la variable diu:", la_variable)

>>> funció()
la variable diu: Sóc local
>>>
>>> la_variable
'Sóc global'
>>>
```

Amb això veiem que si, dins d'una funció, volem modificar el valor d'una variable global, només aconseguirem crear una nova variable local amb el mateix nom, i la variable global quedarà igual...

La sentència "**global**" permet accedir des d'una funció a variables globals que ja existeixin i, fins i tot, crear-ne de noves:

```
>>>
>>> variable_global = "Sóc global"
>>>
>>> def funció ():
    #             modifica la variable global ...
    global variable_global
    variable_global = "Ja no sóc global"
    print("la variable variable_global diu:", variable_global)
    #             ... i en crea una altra
    global nova_global
    nova_global = "No desapareix després d'executar la funció"
    print(nova_global)

>>> funció()
la variable variable_global diu: Ja no sóc global
No desapareix després d'executar la funció
>>> variable_global
'Ja no sóc global'
>>> nova_global
"No desapareix després d'executar la funció"
>>>
```

### Activitats:

Probar els exemples i explicar els resultats obtinguts.

**Nota important:** NO es recomana ni crear ni manipular variables globals des de funcions. De fet és una bona pràctica minimitzar (o evitar) l'ús de variables globals.

## 4. Excepcions en Python.

En Python també es disposa d'un mecanisme per gestionar excepcions. És molt semblant al que proporciona Java.

L'execució d'aquest script

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

dividend = 1
divisor = 0

resultat = dividend/divisor
print ("El resultat de la divisió és: ", resultat)
```

provoca una excepció de divisió per 0:

```
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_
_excep00.py
Traceback (most recent call last):
  File "C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_e
xcep00.py", line 7, in <module>
    resultat = dividend/divisor
ZeroDivisionError: division by zero
>>> |
```

Per evitar que l'excepció aborti l'execució, utilitzem les sentències bàsiques **try** i **except**. Posem el codi que pot provocar excepcions en un bloc **try** i donem tractament a les excepcions en un bloc **except**:

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

dividend = 1
divisor = 0

try:
    resultat = dividend/divisor
    print ("El resultat de la divisió és: ", resultat)

except:
    if divisor == 0:
        print ("No es pot dividir por zero")
```

Això permet finalitzar el programa de manera controlada:

```
>>>
RESTART: C:\Users\est15\AppData\Local\Programs\Python\Python35-32\prova_
_excep01.py
No es pot dividir por zero
>>> |
```

La clàusula **except** permet distingir quin tipus d'error s'ha donat. Per exemple, la divisió no serà correcta si el divisor és 0 però tampoc si no és un valor numèric.

El següent script distingeix els dos casos esmentats:

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

dividend = "7"
divisor = 2

try:
    resultat = dividend/divisor
    print ("El resultat de la divisió és: ", resultat)

except ZeroDivisionError:
    print ("No es pot dividir por zero")

except TypeError:
    print ("Tipus de dades incorrecte")
```

i genera aquesta sortida

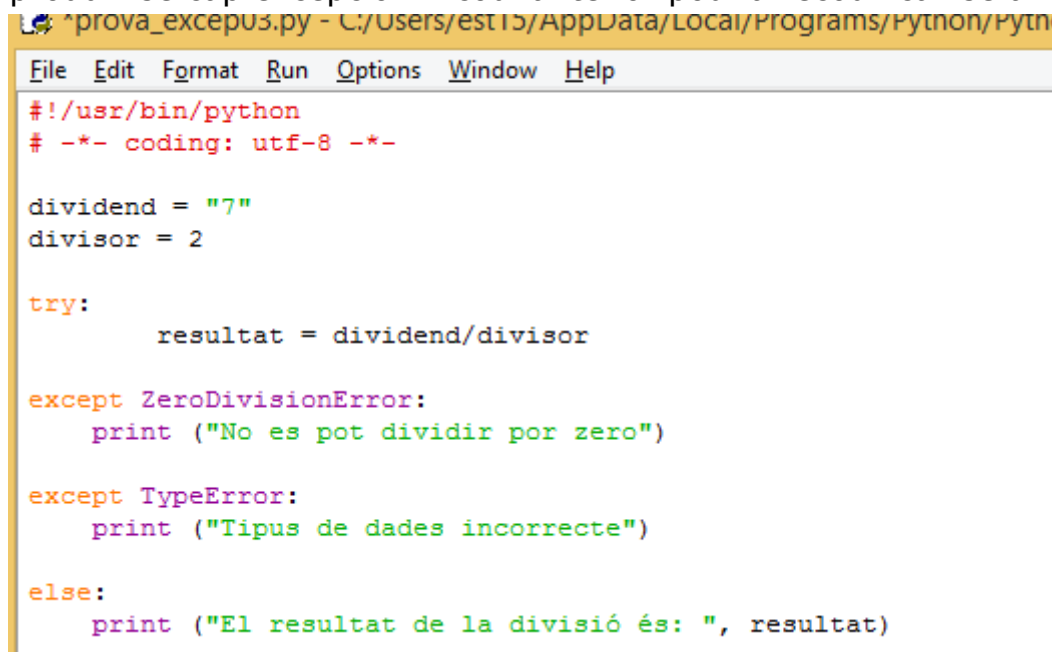
```
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova
_excep02.py
Tipus de dades incorrecte
>>> |
```

**Per consultar els codis d'excepcions predefinides veure**

<https://docs.python.org/3/library/exceptions.html>

<https://docs.python.org/2/library/exceptions.html>

També existeix la clàusula **else** que obre un bloc de codi pel tractament en cas de no produir-se cap excepció. El codi anterior podria recodificar-se així:



```
prova_excep03.py - C:/Users/est15/AppData/Local/Programs/Python/Python35-32
File Edit Format Run Options Window Help

#!/usr/bin/python
# -*- coding: utf-8 -*-

dividend = "7"
divisor = 2

try:
    resultat = dividend/divisor

except ZeroDivisionError:
    print ("No es pot dividir por zero")

except TypeError:
    print ("Tipus de dades incorrecte")

else:
    print ("El resultat de la divisió és: ", resultat)
```



i s'obtidria el mateix resultat.

També es disposa de la clàusula **finally**, el bloc de la qual s'executa sempre, hi hagi o no excepció.

També és possible aixecar excepcions per programa i definir excepcions d'usuari.

### Activitats:

- Probar els exemples i explicar els resultats obtinguts.
- Probar el següent script i explicar què fa

```
File Edit Format Run Options Window Help
#!/usr/bin/python
# -*- coding: utf-8 -*-

while True:
    try:
        n = input("Sisplau escrigui un enter: ")
        n = int(n)
        break
    except ValueError:
        print("El valor no es vàlid. Sisplau, torni-ho a intentar ...")
print ("Molt bé, el valor és vàlid !")
```

- Escriure un script que demani a l'usuari el nom d'un fitxer per obrir i visualitzar. Utilitzar control d'excepcions per detectar noms de fitxers inexistents i tornar a demanar-lo.

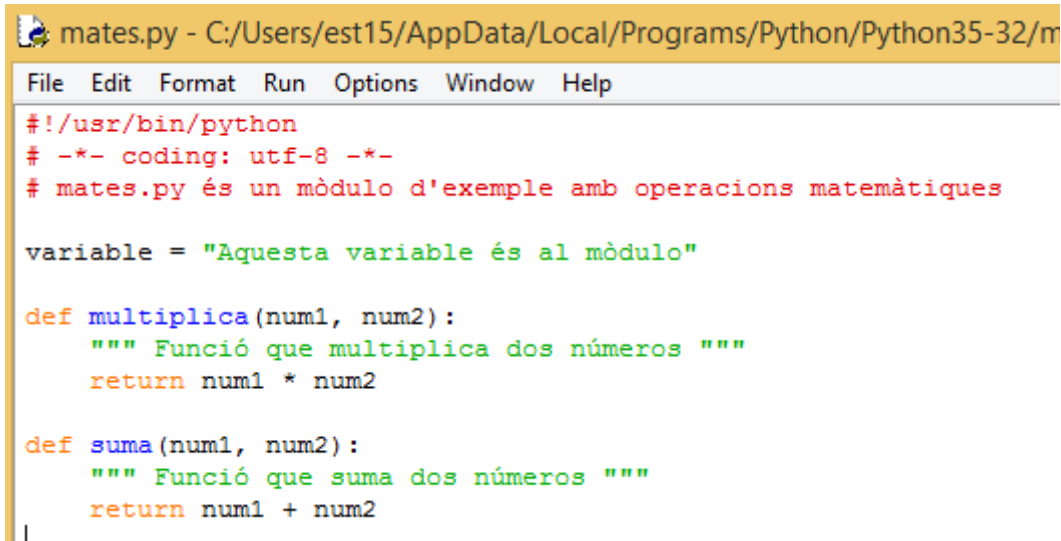
## 5. Mòduls i paquets.

### 1. Importar mòduls.

Un **mòdul** és un arxiu amb codi Python que pot ser reutilitzat per altres programes, després d'importar-lo utilitzant la sentència **import**.

Qualsevol arxiu amb extensió “.py” és un mòdul i pot ser importat. El nom del mòdul és el mateix nom del fitxer, sense l'extensió “.py”. A més dels que programem nosaltres, totes les instal·lacions de Python tenen disponibles centenars de mòduls que constitueixen la **biblioteca estàndard**.

Com a exemple, creem l'arxiu “mates.py”, que reutilitzarem:



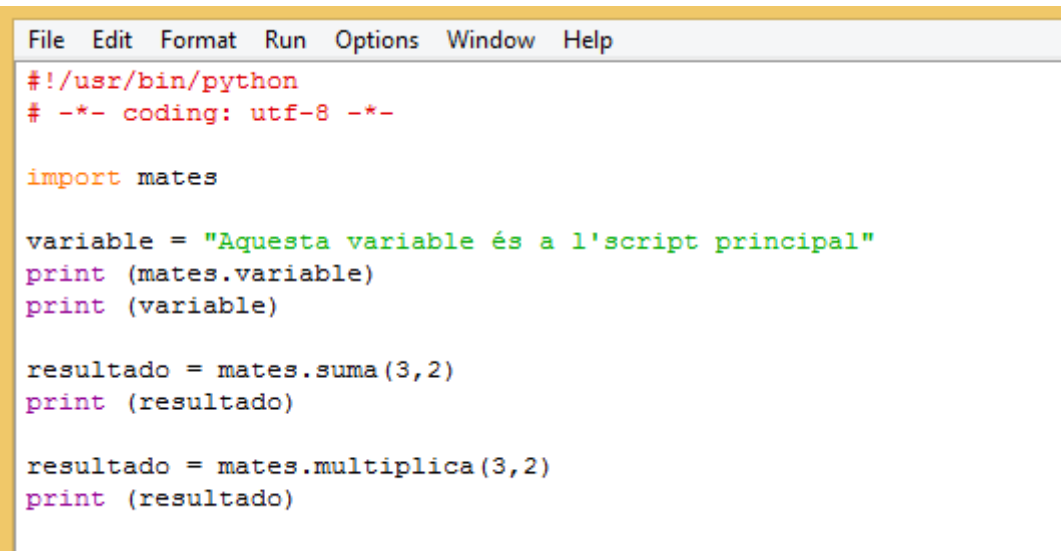
```
mates.py - C:/Users/est15/AppData/Local/Programs/Python/Python35-32/m
File Edit Format Run Options Window Help
#!/usr/bin/python
# -*- coding: utf-8 -*-
# mates.py és un mòdul d'exemple amb operacions matemàtiques

variable = "Aquesta variable és al mòdul"

def multiplica(num1, num2):
    """ Funció que multiplica dos números """
    return num1 * num2

def suma(num1, num2):
    """ Funció que suma dos números """
    return num1 + num2
```

En el següent script, s'importa el mòdul i s'utilitzen les seves variables i funcions, prefixant-les amb el nom del mòdul:



```
File Edit Format Run Options Window Help
#!/usr/bin/python
# -*- coding: utf-8 -*-

import mates

variable = "Aquesta variable és a l'script principal"
print (mates.variable)
print (variable)

resultado = mates.suma(3,2)
print (resultado)

resultado = mates.multiplica(3,2)
print (resultado)
```

L'execució de l'script produiria aquesta sortida:

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Py
Aquesta variable és al mòdul
Aquesta variable és a l'script principal
5
6
```

## 2. Espais de noms.

La tècnica que utilitza Python per mantenir aïllats els noms de variables, funcions, classes, objectes etc i evitar conflictes, són els **espais de noms** o namespaces. Cada mòdul és un espai de noms diferent, per això no hi ha confusió entre variables que es diuen igual, però partanyen a diferents mòduls. Per accedir a un objecte que partany a un mòdul, el nom s'ha de prefixar amb el nom del mòdul.

Quan s'importa un mòdul **es busca** un arxiu, amb extensió ".py" i amb el nom indicat, **al mateix directori** on és el programa que fa la importació. Si no se n'hi troba cap, es busca **en les rutes per defecte** que depenen de la instal·lació i del sistema operatiu.

Els camins on es busquen els mòduls són a la variable **path** del mòdul estàndard **sys**. Com qualsevol altre mòdul, es pot importar i utilitzar, en aquest cas per visualitzar una variable:

```
>>>
>>> import sys
>>> sys.path
['', 'C:\\Users\\est15\\AppData\\Local\\Programs\\Python\\Python35-32\\Lib\\idle
lib', 'C:\\Users\\est15\\AppData\\Local\\Programs\\Python\\Python35-32\\python35
.zip', 'C:\\Users\\est15\\AppData\\Local\\Programs\\Python\\Python35-32\\DLLs',
'C:\\Users\\est15\\AppData\\Local\\Programs\\Python\\Python35-32\\lib', 'C:\\Use
rs\\est15\\AppData\\Local\\Programs\\Python\\Python35-32', 'C:\\Users\\est15\\Ap
pData\\Local\\Programs\\Python\\Python35-32\\lib\\site-packages']
>>> |
```

Si tenim dos mòduls

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# primer.py es un mòdul d'exemple

variable = "Sóc al mòdul 'primer'"

def f_exemple():
    """ Funció que mostra un text """
    print (variable)
```

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# segon.py es un mòdul d'exemple

variable = "Sóc al mòdul 'segon'"

def f_exemple():
    """ Funció que mostra un text """
    print (variable)
```

amb variables i funcions del mateix nom, no hi ha cap confusió quan els utilitzem en el mateix programa:

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

# Importem els mòduls
import primer, segon

variable = "Sóc a l'script principal"

def f_exemple():
    """ Funció que mostra un text """
    print variable

print "Cridem a la funció d'aquest mòdul:"
f_exemple()

print 'Cridem a la funció del mòdul "primer":'
primer.f_exemple()

print 'Cridem a la funció del mòdul "segon":'
segon.f_exemple()

print "També podem accedir a les variables:"

print variable
print primer.variable
print segon.variable
```

En executar-lo obtenim la sortida següent:

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-3
--- Cridem a la funció d'aquest mòdul:
Sóc a l'script principal
--- Cridem a la funció del mòdul "primer":
Sóc al mòdul 'primer'
--- Cridem a la funció del mòdul "segon":
Sóc al mòdul 'segon'
--- També podem accedir a les variables:
Sóc a l'script principal
Sóc al mòdul 'primer'
Sóc al mòdul 'segon'
>>> |
```

## Importar objectes dins l'espai de noms

Per evitar haver de prefixar sempre el nom dels objectes, en comptes d'importar tot mòdul es pot importar només els objectes que necessitem:

```
from <nom_de_mòdul> import <nom_objecte1>, ... , <nom_objecteN>
```

Així, podem crear el mòdul "operacions":

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# operacions.py es un mòdul d'exemple

numero_pi = 3.14159

def multiplica(un, dos):
    """ Funció que multiplica i retorna el resultat"""
    return un * dos

def suma(un, dos):
    """ Funció que suma i retorna el resultat"""
    return un + dos
```

del qual importem les dues funcions i podem utilitzar-les sense prefixar:

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

# Importar les funcions "multiplica" i "suma".
from operacions import multiplica, suma

print (suma(21, 10))

print (multiplica(21, 10))
```

Quan executem l'script obtenim

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_mòduls2b.py
31
210
>>> |
```

Però la variable `numero_pi` no l'hem importat i no la podem utilitzar. En executar

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

# Importar les funcions "multiplica" i "suma".
from operacions import multiplica, suma

print (suma(21, 10))
print (multiplica(21, 10))
print (numero_pi + operacions.numero_pi)
```

S'obté un error:

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_mòduls2b.py
31
210
Traceback (most recent call last):
  File "C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_mòduls2b.py", line 11, in
<module>
    print (numero_pi + operacions.numero_pi)
NameError: name 'numero_pi' is not defined
>>>
```

Es poden importar tots els objectes d'un mòdul

```
from <nom_de_mòdul> import *
```

No es recomana per la possibilitat de pèrdua de control: es poden produir conflictes amb objectes del mateix nom de diferents mòduls i, per exemple, utilitzar una funció que no és la que pensem.

### 3. Execució d'un script com a mòdul i/o script independent.

En Python tots els mòduls tenen una variable privada **\_\_name\_\_** (comença i acaba amb dos guions baixos) que conté **el nom del mòdul** i que es pot consultar en execució. Quan l'script s'executa com a mòdul principal, **\_\_name\_\_** conté el valor **\_\_main\_\_**.

Consultant **\_\_name\_\_** podem decidir si fer o no fer tractaments (p.e. llegir paràmetres de la línia de comandes només si el mòdul s'executa com script independent) o, fins i tot, forçar que l'script només s'executi com a mòdul (o a l'inrevés).

### 4. Paquets.

Un **paquet** és un **directori que conté diversos mòduls**, en principi **relacionats** entre ells. Perquè Python consideri que un directori és un paquet, a més dels mòduls, ha de contenir un fitxer anomenat **\_\_init.py\_\_**. Aquest fitxer pot ser buit o contenir inicialitzacions.

Els paquets **poden contenir altres paquets**, de manera jeràrquica.

**Exemple:** dins la carpeta "matemàtiques" creem l'script "constants.py"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
constants.py conté constante matemàtiques
forma part del paquet "matemàtiques"
"""
pi = 3.14159
e = 2.71828
zeta = 1.64493
```

i l'script "operacions.py"

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
operacions.py conté operacions matemàtiques
forma part del paquet "matemàtiques"
"""

def multiplica(un, dos):
    """ Funció que multiplica i retorna el resultat"""
    return un * dos

def suma(un, dos):
    """ Funció que suma i retorna el resultat"""
    return un + dos

def quadrat(num):
    """ Funció que retorna el quadrat d'un número"""
    return num * num
```

També hi creem un fitxer **\_\_init\_\_.py** en blanc, perquè el directori "matemàtiques" sigui també el paquet "matemàtiques".

El següent script importa i utilitza els mòduls del paquet "matemàtiques":

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
utilitzar el paquet "matemàtiques"
"""

import matemàtiques.constants, matemàtiques.operacions

radi = 14

radi_quadrat = matemàtiques.operacions.quadrat(radi)
circunferencia = matemàtiques.operacions.multiplica(matemàtiques.constants.pi,
                                                    radi_quadrat)

print (circunferencia)
```

i produeix el resultat

```
>>>
RESTART: C:/Users/est15/AppData/Local/Programs/Python/Python35-32/prova_paquets1.py
615.75164
>>> |
```

### Comentaris:

S'han importat els mòduls individualment, especificant l'espai de noms sencer. Si intentem importar el paquet complet d'una tacada

```
import matemàtiques
```

no importarem res mentre el fitxer **\_\_init\_\_.py** estigui buit.

Si dins de `__init__.py` hi afegim clàusules d'importació, com per exemple

```
from .constants import e
from .operacions import suma
```

la importació anterior

```
import matemàtiques
```

tindrà per efecte importar la variable **e** i la funció **suma**.

Per evitar la incomoditat d'utilitzar molt llargs es poden definir **àlies**. En el següent script s'importa el mòdul de constants "**matemàtiques.constants**" i se li assigna com a àlies "**cons**". Al mòdul d'operacions se li assigna l'àlies "**ope**". Llavors en el codi es pot prefixar amb els àlies corresponents els objectes definits dins de cada mòdul.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
utilitzar el paquet "matemàtiques"
"""

import matemàtiques.constants as cons, matemàtiques.operacions as ope

radi = 14

radi_quadrat = ope.quadrat(radi)
circunferencia = ope.multiplika(cons.pi, radi_quadrat)

print (circunferencia)
|
```

El resultat és el mateix que abans.

### Activitats:

- Probar els exemples i explicar els resultats obtinguts.