

Bloc D: Programació en Python

pre-alpha 10/11/22 by XFF

Bibliografia

- Sweigart, A. (2015). *Automate the boring stuff with python: practical programming for total beginners*. San Francisco, No Starch Press.
- Matthes, E. (2015) *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (1st. ed.). No Starch Press, USA.
- Zelle, J. (2009) *Python Programming: An Introduction to Computer Science*. (2nd ed.) Franklin, Beedle & Associates INC. USA
- van Rossum, G. et al. (2018) *Python Tutorial*
- Guru99 Python tutorial (<https://www.guru99.com/python-tutorials.html>)
- W3 Schools Python tutorial (<https://www.w3schools.com/python/default.asp>)

Ampliació i reptes de programació pels més agosarats:

- <https://projecteuler.net/>
- <https://stackoverflow.com/questions/2573135/python-progression-path-from-apprentice-to-guru>

Metodologia de classe

Classes de 2h dividides en dues parts:

- Primera part: Part extensiva de teoria on es presenta el material de manera teorico-pràctica. i.e: mentre es presenten ints, llistes, for loops, etc s'ensenya al IDE o a VSCode el seu comportament. L'alumne s'espera que provi els exemples de classe o que prengui apunts de la teoria. La classe està pensada per a fer servir un híbrid entre diapositives/pissarra i demostracions en viu amb IDE.
- Sergona part: Part de treball en grup amb exercicis guiats a classe. Es pot dedicar part de la segona hora a explicar nous conceptes però aquests sempre aniran acompanyats d'exercicis. Al final de la classe se'ls donarà el projecte que tenen que fer per a la setmana vinent (a entregar online el dia que convingui)

Filosofia del bloc D

La programació és una disciplina que s'aprèn sobretot de manera autònoma i dedicant-li temps i interessant-se per un/a mateix/a. Tenint una quantitat limitada de classes (5 classes de 2h i escaix) i molt temari per a poder oferir una visió completa de la programació, es defineix el bloc D com un curs **intensiu**, assumint zero coneixement per part de l'alumne. Els coneixements explicats a classe estan pensats per a ser treballats durant la segona hora i fora de classe per mitjà d'exercicis i projectes entregables. Tantmateix, els mètodes d'avaluació canvien respecte els altres blocs i s'incentiva el treball en grup per a fer més amena la càrrega lectiva del bloc.

Índex per classe (Part 1: 1h teoria. Part 2: 1h teoria/problemes)

- Classe 0, 1:

- Part 1: Introducció a la programació. IDE de python, VSCode, conceptes fonamentals sobre programació (problema real -> formulació algorítmica -> traducció a llenguatge de programació). Bones praxis en programació: escriure pseudocodi -> programar. Hello World. StackOverflow.
- Part 2: Presentació de les estructures de dades: variables i tipus, llistes i diccionaris. Operadors. Control flow: for, while, if, else, elif, break i continue. Definició de funció i la seva sintaxi. Fer grups de treball i assignar primera entrega.

- Classe 2:

- Part 1: Introducció a la programació modular. Importar mòduls, utilitzar mètodes i funcions de tercers. Presentació del paquet std de python. Operacions sobre llistes, diccionaris (pop, append, key, etc...). Wrappers i casting de variables.
- Part 2: Funcions en detall: sintaxi, invocació i estructura. Control d'errors: estratègies de debugging, print(), type(), entendre els outputs d'error (SyntaxError, ValueError, NameError, etc). Documentació online de mòduls. Assignar segona entrega.

- Classe 3:

- Part 1: Mòduls de python en l'àmbit científic: NumPy, Matplotlib, SciPy (fsolve per exercicis de PH). STDIN, STDOUT i escriure resultats a fitxer. Carregar arxius csv amb NumPy i ús d'estructures de dades amb funcions (programació funcional). Generar gràfiques a partir de dades reals.
- Part 2: Introducció a les Classes de Python i al OOP. Estructura de classe, inicialització, objectes, invocar mètodes, atributs, herència. Assignar tercera entrega.

- Classe 4:

- Part 1: Classes i OOP en detall. RDKit i usos en química computacional. Classes aplicades a la química.
- Part 2: RDkit en profunditat: consolidació dels conceptes de classes i OOP amb la API de RDkit i exercicis guiats a classe. **A final de classe:** Tria projecte final: Chatbot Twitter, ML app, Joc (snake, tic-tac-toe, game of life). TO BE DETERMINED!

- Classe 5:

- Part 1: Presentació projecte final: explicació codi (50%), demostració funcionalitat (30%) estructura del codi (20%). Treball final 70% i entregues 30%.
- Part 2: Contingut addicional per a l'ampliació i comiat. Recollir crítiques/comentaris dels alumnes.

Classe 0: Instal·lació de VSCode (o Spyder) en environment de Conda. Què és un IDE?

- <https://code.visualstudio.com/download> (idealment ja hauria d'estat tot instal·lat a l'aula)
- Un IDE (Integrated Development Environment) és una interfase on podem crear els nostres programes i ens facilita les tasques relacionades amb el formateig del codi, la compilació, el debugging i d'altres.
- Es pot programar des d'un bloc de notes, però els IDEs estan pensats per a facilitar-nos la vida i tenen multitud d'opcions per a ser personalitzats i treure'n el màxim profit.
- Van des del IDE de python més senzill fins a IDEs on tens total control sobre totes les variables relacionades amb la programació.



EXPLORER

- ERIBLOCOD
 - > __pycache__
 - > .cache
 - classe_1_entregues
 - ex_1_test.py
 - ex_2_test.py
 - ex_4_test.py
 - ex_4-1_test.py
 - classe_1_exercicis
 - ex_class1_dicts.py
 - ex_class1_llistes.py
 - > classe_2_entregues
 - > classe_2_exercicis
 - README.md

Editor

```
class_1_exercicis > ex_class1_dicts.py
52
53
54
    "especie": "gos",
    "color": ['negre', 'taronja', 'blanc'],
    "good_boy": True}

als seus valors de la següent manera:
58
59
60
61
62
    dict_nested["animal_1"]["nom"] #retorna "Altramuz"
    dict_nested["animal_2"]["good_boy"] #retorna False
    dict_nested["animal_3"]["color"][0] #retorna "negre"
62
63
64
65
66
Els diccionaris són especialment útils quan treballem amb variables complexes i ens
ajuden a que el nostre codi sigui més llegible.
65
66
67
68
69
70
71
72
73
74
75
# Exercici 1
""" Fes un diccionari que sigui la fitxa d'un alumne de la uni: Nom, NIU, edat,
email, direcció i telefon.
"""
# Exercici 2
""" Amb el diccionari de l'exercici 1, canvia el valor del número de telèfon i el NIU
```

Terminal

```
bash - classe_1_exercicis
(base) xavi@xav-Hummer:~/eriBlocD/classe_1_exercicis$
```

Footer

main 0 0 0 Ln 82, Col 26 Spaces: 4 UTF-8 LF MagicPython 3.9.7 ('gaudi2_dev': conda)

Pestanyes obertes

Codi

Terminal

Carpetes i arxius

Versió de python

Classe 0: On anar a buscar informació quan tot falla?

Si en algun moment esteu perduts o perdudes, no enteneu alguna cosa o teniu un bug que no sabeu resoldre...

<https://stackoverflow.com/>

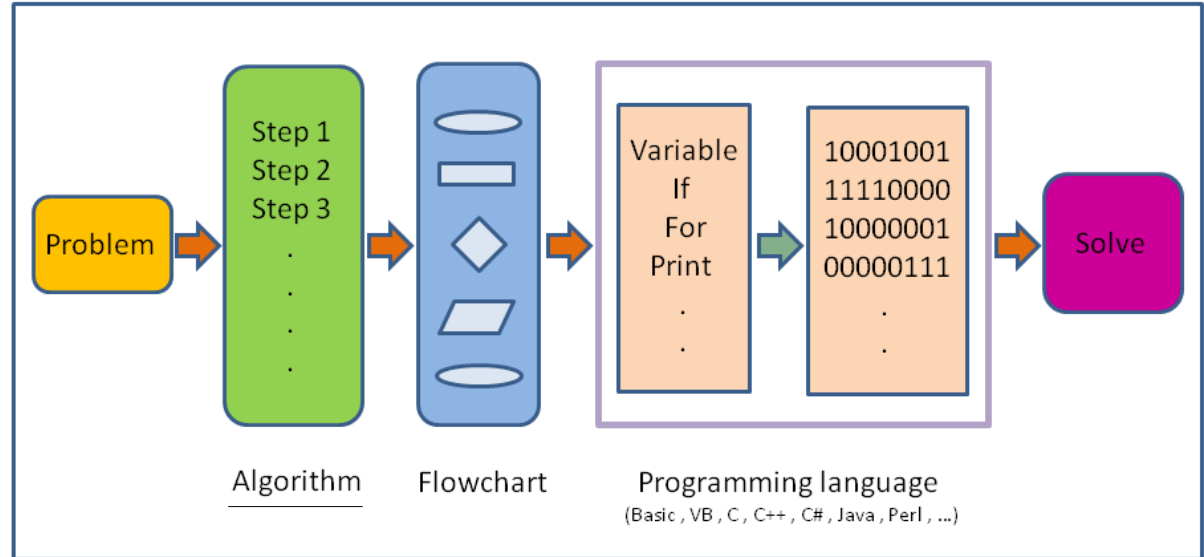
Un ~99% del codi que s'escriu a tot arreu s'agafa d'aquí. És el millor lloc per aprendre a programar i demanar dubtes o resoldre problemes que vagin sorgint.

Classe 1: Introducció a la programació. Plantejament del problema

Problema real → **Adaptació algorítmica (passos concrets)** → Control de flux (primer A, després B, ...) → Traducció a llenguatge de programació → END

I.e: Fer una truita:

- 1) Comprar ingredients
- 2) Treure estris (paella, bol, espàtula, etc)
- 3) Pelar patates I ceba
- 4) Tallar patates I ceba
- 5) Posar oli a paella
- 6) Cuinar ceba
- 7) Apartar la ceba
- 8) Cuinar patata
- 9) Batre ous
- 10) Posar sal
- 11) ...

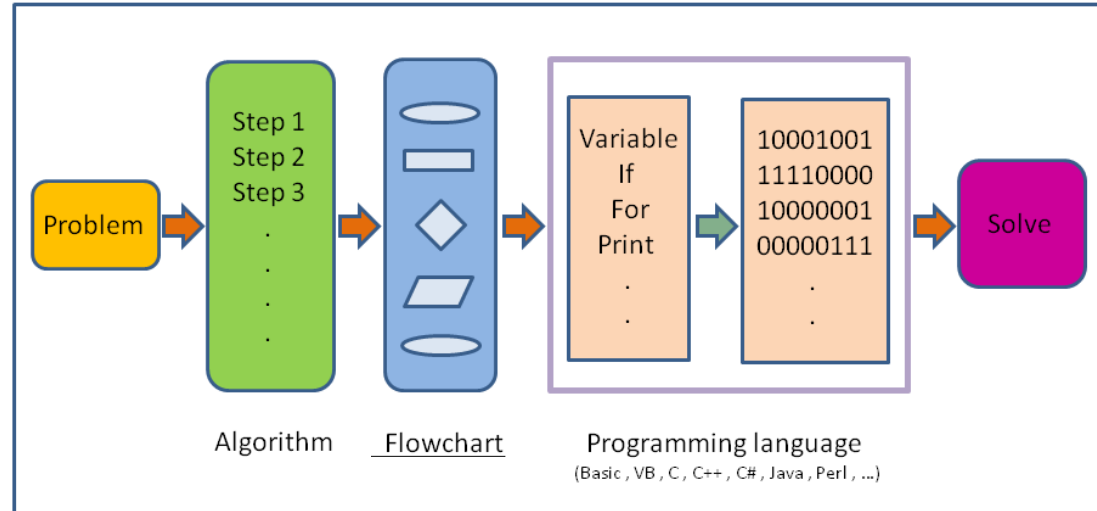


Classe 1: Introducció a la programació. Pseudocodi

Problema real → Adaptació algorítmica (passos concrets) → **Control de flux** (primer A, després B, ...) → Traducció a llenguatge de programació → END

I.e: Fer una truita:

- 1) Comprar ingredients
- 2) Treure estris (**si l'estri és necessari**: paella, bol, espàtula, etc)
- 3) Pre-tractament ingredients
-**Per cada ingredient**:
rentar, pelar (si cal), tallar (si cal)
- 4) Cuinar ingredients
- 5) Mesclar ingredients
- 6) Cuinar truita



Classe 1: Introducció a la programació. Adaptació

Problema real → Adaptació algorítmica (passos concrets) → Control de flux (primer A, després B, ...) → **Traducció a llenguatge de programació** → END

I.e: Fer una truita:

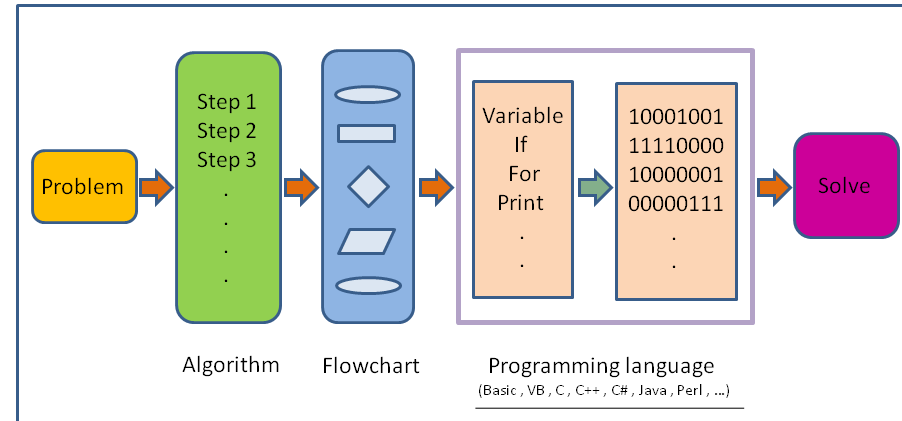
1) def buy_ingredients():

```
    ingredients=["potato", "potato", "onion", "egg", "egg", "egg", "egg"]  
    return ingredients
```

2) tools = ["pan", "oil", "salt"]

3) def cut(ingredients):

```
    peeled = ingredients.copy()  
    for i in range( len(ingredients)):  
        peeled[i] = "peeled " + ingredients[i]  
    return peeled
```



Classe 1: Introducció a la programació. Consells

A l'hora de començar a fer un programa complex, feu un primer esbós en paper sobre l'estructura que li voleu donar, les variables que tindrà i les operacions algorítmiques que fareu per a transformar el input A en el resultat que voleu B.

També és recomanable que les variables tinguin un nom que s'entengui i que aquest sigui representatiu del que són. Tots tenim pressa, però passats dos mesos, un d'aquests dos programes no sabreu què fa:

```
def cut(ingredients):  
    peeled = ingredients.copy()  
    for i in range( len(ingredients)):  
        peeled[i] = "peeled " +  
ingredients[i]  
    return peeled
```

```
def f(x):  
    tall = x.copy()  
    for i in range( len(x)):  
        tall[i] = "peeled " + x[i]  
    return tall
```

Classe 1: Introducció a la programació. Hello World!

- **Primer programa:** Hello world!
- Programa que imprimeix per pantalla les paraules "Hello World!"

```
>>> print("Hello World!")  
Hello World!  
>>>
```

print() → operació de STDOUT
(standard output)

- Funció que accepta un argument
- Accepta strings, nombres, etc...

"Hello World!" → String

- A python, les strings son variables que es fan servir per a representar paraules i frases. Similars a les llistes.

Classe 1: Introducció a la programació: Hello World!

- Print() és una **funció** que admet un argument entre parèntesis i imprimeix per pantalla el que li passem
- Podem emmagatzemar un valor donat en una **variable** i imprimir per pantalla la variable. Les variables s'escriuen sempre en minúscules i acostumem a escriure els espais amb barra baixa. I.e: my_var, atomic_distance_matrix, name, etc... Les majúscules les reservem per un altre tipus d'objectes.
- Per a crear una variable, li fem un nom i li assignem un valor. I.e:

```
>>> print("Hello World!")  
Hello World!  
>>>
```

```
var = "Howdy, World?"  
print(var)
```

```
>>> 'Howdy, World?'
```

Classe 1: Introducció a la programació: Paraules reservades

- Python es reserva certes paraules que nosaltres no podem fer servir com a variables. Generalment, ho veurem molt clar quan una paraula està reservada perquè si intentem fer-la servir com a variable, el nostre IDE l'escriurà amb un determinat color.
- Si intentem definir una variable amb una paraula reservada, ens donarà error. Algunes paraules reservades són: list, string, int, float, ...

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

```
my_string = "nom de variable acceptable, en blau"
string = "les paraules reservades surten en verd al VSCode"
```

Classe 1: Introducció a la programació: Tipus de variables

- Els tipus de variables més comuns que farem servir són números o lletres.
- **Int:** variables de tipus enter. Nombres enters, incloent el zero (1, 2, 7262, -6746, 0, etc...)
- **Float:** variables de “punt flotant”. Representen nombres decimals (1.2, -5.664334, 0.6, etc...)
- **String:** variables per a representar paraules, frases o caracters. S’escriuen entre cometes i es poden indexar (“Foo”, “a”, “Foo Bar”, etc...) Per a indexar strings, hem de donar-lis un nom. I.e:
 - `my_string = “ABCDEFGG”`
 - `my_string[0] >>> ‘A’`
 - `my_string[2] >>> ‘C’`
- **Bool:** Variables booleanes. Són True i False. Es fan servir amb els operadors per a veure si una afirmació és certa o falsa.
 - `Print(9>6) >>> True`
 - `Print(8>66) >>> False`

Classe 1: Introducció a la programació: Operadors

- Els operadors són eines que ens permeten treballar amb variables i obtenir-ne de noves o bé comprovar relacions entre elles.
- **Operadors d'assignació I**
 - **(+)**: Suma dos o més nombres o iterables (com llistes o strings)
 - **(-)**: Resta dos nombres. No funciona amb lletres o llistes.
 - **(*)**: Multiplica un nombre per un altre o un nombre per un iterable. (Proveu a fer paraula = "hola" i després feu paraula * 4).
 - **(/)**: Divideix dos nombres. No funciona amb iterables.
 - **(//)**: Divisió entera. I.e: 7 // 3 >>> 2
 - **(%)**: Operació mòdul. Ens retorna el mòdul d'un nombre
 - **(**)**: Exponencial. Eleva un nombre a l'exponent que sigui.

Classe 1: Introducció a la programació: Operadors d'assignació

- Els operadors d'assignació ens permeten donar un valor a una variable o bé modificar el seu valor existent.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Classe 1: Introducció a la programació: Operadors de comparació, lògics, d'identitat i de memebresia.

- Els operadors de comparació ens permeten comparar dos valors i ens retornen True o False depenent de si la comparació és certa o no.

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Classe 1: Introducció a la programació: Operadors de comparació, lògics, d'identitat i de memebresia.

- Els operadors lògics ens permeten crear enunciats complexos en els que demanem que certes variables compleixin tot un seguit de relacions concretes.
- Acostumem a fer-los servir dins de bloc condicionals.

Operator	Description	Example
and	Returns True if both statements are true	<code>x < 5 and x < 10</code>
or	Returns True if one of the statements is true	<code>x < 5 or x < 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

Classe 1: Introducció a la programació: Operadors de comparació, lògics, d'identitat i de memebresia.

- Els operadors d'identitat es fan servir per a comparar directament dues variables

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

- Els operadors de membresia serveixen per a saber si una variable en conté una altra o no

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Classe 1: Introducció a la programació: Llistes

- Les llistes són estructures de dades indexades (comencen al 0) que ens permeten emmagatzemar variables en posicions concretes i permeten valors duplicats. Súper útils a l'hora de programar. Es poden fer servir per a guardar valors que volguem conservar, com a vectors i en general per a guardar qualsevol cosa i modificar-la mentre corre el programa. Es defineixen entre claudators i els valors se separen amb comes. Es poden anidar!

```
my_list_1 = [1,2,[3,4,5],6]
```

```
my_list_2 = ["hola", "bon", "dia"]
```

```
my_list_3 = [1.33334, True, "Foo", 42]
```

```
my_list_1[2] >>> [3,4,5]
```

```
my_list_2[2] >>> 'dia'
```

```
my_list_3[0] >>> 1.33334
```

Classe 1: Introducció a la programació: Llistes

- Les llistes també accepten tot un seguit de funcions que existeixen per defecte a python i ens ajuden a manipular-les més fàcilment. El llenguatge de python ens permet utilitzar el punt (.) per a aplicar una funció a una llista. Veurem els funcions més endavant, però convé que ens familiaritzem amb la sintaxi i que veiem com les apliquem. Per exemple, la funció `append()` afegeix un valor concret a una llista. La funció `pop()` el·limina l'últim valor d'una llista.

```
my_list_1 = [1,2,[3,4,5],6]
```

```
my_list_2 = ["hola", "bon", "dia"]
```

```
my_list_1.append(300) >>> [1,2,[3,4,5],6, 300]
```

```
my_list_2.pop() >>> ['hola', 'bon']
```

Classe 1: Introducció a la programació: Llist comprehensions

- Python permet la creació i inicialització de llistes mitjançant el que s'anomena *list comprehension*. És una sintaxi similar al llenguatge humà que ens serveix per a compactar la definició d'una llista. I.e: posem que volem una llista amb 50 elements successius que siguin múltiples de dos:

Sense list comprehension

```
mult_dos = []  
  
for n in range(1, 51):  
    mult_dos.append(2 * n)
```

```
mult_dos = []  
  
for n in range(1, 51):  
    if n%3==0:  
        mult_dos.append(2 * n)
```

Amb list comprehension

```
mult_dos_comp = [2*n for n in range(1,51)]
```

```
mult_dos_comp = [2*n for n in range(1,51) if n%3==0]
```

Es llegeix:

2n per tot n des de 1 a 51.

Aquestes expressions es poden complicar més, afegint condicionals.

Classe 1: Introducció a la programació: Diccionaris

- Els diccionaris a Python són estructures de dades més ordenades que les llistes. Consten de parells de claus:valors (key:value). A diferència de les llistes, cada parell de valors clau:valor és **únic**! Els diccionaris també es poden indexar, però per claus en comptes de per índex. També es poden anidar. Es defineixen entre claus i els valors es separen amb comes. Els diccionaris, com les llistes, també accepten funcions que veurem més endavant.

```
my_dict = {  
    "marca": "Renault",  
    "model": "Clio",  
    "any": 1996,  
    "colors": ['blau', 'vermell', 'blanc']  
}
```

```
dict_nested = {  
    "animal_1": {"nom": "Altramuz",  
                 "especie": "Gos"},  
    "animal_2": {"nom": "Petunia",  
                 "especie": "Oca"}  
}
```

```
my_dict["marca"] >>> 'Renault'  
dict_nested["animal_2"]["nom"] >>> 'Petunia'
```

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Conditionals:**
 - **If / else:** condicional més bàsic. El bloc de codi que escrivim després d'un **if** només s'executarà si la condició que li donem és **certa**. Sinó, l'ordinador ignorarà el bloc i seguirà. Hem d'afegir indentació sota del "if". **else** serveix per a executar el bloc següent en cas de que el "if" sigui fals. I.e:

def funcio(x, y):

 if x > y:

 print(x)

 else:

 print(y)

```
i=23
if i%2==0:
    print("This is the if block")
    print("i is an even number")
else:
    print("This is the else block")
    print("i is an odd number")
```

```
This is the else block
i is an odd number
```

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Conditionals:**
 - **elif:** condicional complementari al **if**. Si el primer bloc “if” és fals, abans de passar al “else” final, si hi ha **elif**, el programa evaluarà totes les condicions intermitges per a veure si són certes.

def funcio(x, y):

 if x > y:

 print(x)

 elif x == y:

 print("Els dos valors són iguals!")

 else:

 print(y)

```
1
2   num = 5.8
3
4   if num > 0:
5       print("Positive number")
6   elif num == 0:
7       print("Zero")
8   else:
9       print("Negative number")
```

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Loop “for”:**
 - La seva sintaxi s'assembla molt a l llenguatge humà. Si volguessim iterar sobre una llista diríem: “per cada element d'aquesta llista...”. En python es tradueix a: “for element in my_list”, on my_list pot ser una variable iterable qualsevol.
 - Generalment, sempre que volguem modificar un seguit de valors dins una llista o diccionari, iterarem sobre ella mitjançant un bucle **for**.

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Loop for:**
 - El loop més utilitzat. Pot iterar sobre una llista o una string, sobre un diccionari, etc. Es pot fer servir conjuntament amb la funció range() per a que ens faci X iteracions. I.e:

```
my_list = [1,2,3]
```

```
for element in my_list:
```

```
    element +=1
```

```
print(my_list) >>> [2,3,4]
```

```
my_list = [1,2,3]  
for _ in range(0,5):  
    print(my_list)
```

```
>>>[1, 2, 3]
```

```
>>>[1, 2, 3]
```

```
>>>[1, 2, 3]
```

```
>>>[1, 2, 3]
```

```
>>>[1, 2, 3]
```

**Això ens permet modificar
l·listes o diccionaris!**

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Loop while:**
 - El loop **while** és un bucle que es va repetint fins que la condició que li donem de partida s'avalua com a False (fins que quelcom deixa de ser cert).

```
my_num = 0

while my_num < 5:
    print("not yet")
    my_num +=1
print("now!")
```

Resultat:

```
>>>not yet
>>>not yet
>>>not yet
>>>not yet
>>>not yet
>>>now!
```

Classe 1: Introducció a la programació: Control Flow

- Un cop tenim controlades les estructures de dades que podem fer servir per a emmagatzemar informació i els operadors bàsics ara hem d'aconseguir transformar un conjunt de variables en un altre. Per això podem fer servir els **condicionals** i els **loops**.
- **Loop while:**
 - **PRECAUCIÓ:** Els “while” poden arribar a generar bucles infinits! (si la condició sempre és certa, mai es sortirà del loop).

```
while 1:  
    print("ad nauseam")
```

Trivia: el valor 0 sempre s'avalua com a False, i el 1 com a True

Resultat:

```
>>>ad nauseam  
>>>ad nauseam  
>>>ad nauseam  
>>>ad nauseam  
>>>ad nauseam  
>>>...
```

Classe 1: Introducció a la programació: Introducció a les funcions

- Les funcions són una manera convenient d'organitzar codi per a fer-lo més fàcil d'entendre i importar-lo entre programes. Un cop hem fet una funció, la podem reutilitzar sempre que vulguem.
- Per a fer una funció, utilitzem la paraula clau **def**, li donem un nom i posem entre parèntesis quines variables agafa (cap, una, dues, etc...). Seguidament, en un bloc indentat, escrivim el que volem que faci i finalment amb la paraula reservada **return** li diem el que volem que ens torni. I.e:

def per a definir-la. Li posem de nom *cut* i accepta la llista *ingredients* com a única variable

```
def cut(ingredients):  
    peeled = ingredients.copy()  
    for i in range( len(ingredients)):  
        peeled[i] = "peeled " + ingredients[i]  
return peeled
```

Useu un **for** loop per iterar sobre els elements de la llista

La paraula reservada **return** ens especifica que la funció ens retorna la llista **peeled**

Classe 2: Funcions I Programació modular

- Les funcions, un cop definides, es poden executar si les invoquem pel seu nom, l.e:

```
def msg():
```

```
    print("Missatge de prova") → Per utilitzar-la fem msg(), o podem assignar el seu resultat a  
    una variable → missatge = msg()
```

- O bé es poden aplicar a un tipus concret d'objectes (com llistes, diccionaris, strings, etc). Llavors les anomenem **mètodes**. Els mètodes que farem servir habitualment ja estan definits dins de Python. Per a cridar-los, s'utilitza l'operador punt (.) **juntament amb l'objecte sobre el qual aplicarem el mètode**.

l.e: Les strings tenen un mètode anomenat **split()** que separa una string segons un delimitador que li proporcionem.

```
dot_string = "Missatge.de.prova"
```

```
slash_string = "Missatge/de/prova"
```

```
dot_string.split(".") >>> ["Missatge", "de", "prova"]
```

```
slash_string.split("/") >>> ["Missatge", "de", "prova"]
```

- El mètode **split()** és únic de les strings (no funcionarà amb altres objectes com diccionaris o llistes) i accepta un valor de separació (en aquest cas el punt). Retorna una llista amb la string separada.

Classe 2: Funcions I Programació modular

- Un cop hem escrit una funció la podem guardar en un **mòdul**. Un mòdul és un arxiu de python que conté una o més funcions però no té cap instrucció ni funció principal, només emmagatzema funcions. Els mòduls es poden importar per a fer-los servir en qualsevol programa que fem i així no haver de repetir codi.
- Per a tenir accés a les funcions que estan guardades a un mòdul concret, fem servir la paraula reservada **import**. A vegades, per a poder accedir a les funcions, hem d'especificar de quin mòdul provenen. Hi ha diverses maneres d'importar mòduls:

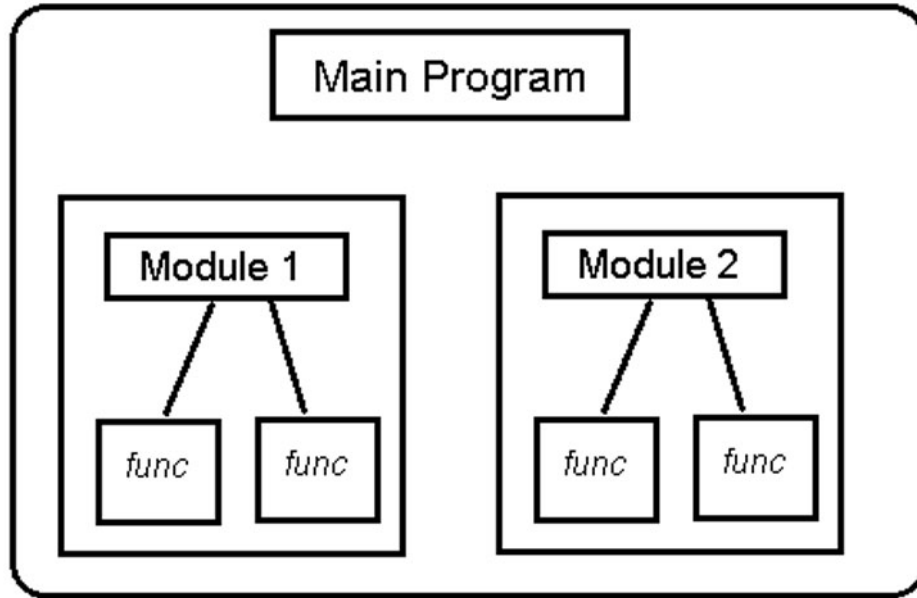
```
2_exercicis > ➤ funcions.py > 📁 foldl_str
def fancy_suma(x, y):
    """fancy_suma

    Args:
        x (int): un numero
        y (int): un altre numero

    Returns:
        int: suma formatada de x i y
    """
    return print("La suma de", x, "i", y, "és:", x + y)
```

Creem un arxiu que es digui `funcions.py` i allà definim totes les funcions que volgüem.

Classe 2: Funcions I Programació modular



Aquesta estratègia de dividir un programa en varis mòduls on s'agrupa el codi segons la seva funció és molt útil a l'hora d'organitzar un programa. També ens ajuda a veure on hi ha un error quan quelcom falla.

Classe 2: Funcions I Programació modular. Ampliació de funcions

- **Import** va seguit del nom del mòdul. En aquest cas d'exemple, el mòdul es diu **funcions.py** I podem cridar-lo amb **import funcions**. Sempre que volguem fer servir una funció del mòdul **funcions.py** haurem de fer servir l'operador punt (.) juntament amb el nom de la funció que volem invocar (com si fos un mètode).
- A més a més, per a tenir una sintaxi més llegible, podem importar un mòdul I posar-li un pseudònim per a cridar-lo de manera més senzilla (I.e: fx en comptes de funcions).
- Si només volem importar certes funcions d'un mòdul, fem servir: **from NOM_MODUL import NOM_FUNCIONS**

```
classe_2_exercicis > ex_imports.py > ...  
1 | import funcions as fx  
2 |  
3 | # Exemple  
4 | x = 7  
5 |  
6 | y = 99  
7 |  
8 | fx.fancy_suma(x, y)
```

Des d'un altre arxiu

Alternativament es podria haver fet:
from funcions import fancy_suma

O directament :
Import funcions

```
(gaudi2_dev) xavi@xav-Hummer:~/eriBlocD/classe_2_exercicis$ python ex_imports.py  
La suma de 7 i de 99 es 106
```

Classe 2: Funcions I Programació modular. Ampliació de funcions

- Les funcions poden admetre més d'un argument i cada argument pot ser d'un tipus diferent.

```
def update_list(mylist, update_number):  
    newlist = []  
    for element in mylist:  
        element += update_number  
        newlist.append(element)  
    return newlist
```

```
def update_list(mylist, update_number):  
    """Suma un nombre determinat a cada element d'una llista  
  
    Args:  
        mylist (list): llista de nombres enters  
        update_number (int): nombre a sumar  
  
    Returns:  
        newlist: nova llista amb els valors desitjats  
    """  
    newlist = []  
    for element in mylist:  
        element += update_number  
        newlist.append(element)  
    return newlist
```

Aquesta funció accepta una llista i un nombre i retorna una nova llista on a cada element se li ha sumat el nombre. I.e: `update_list([1,2,3], 5) → [6,7,8]`

El comentari amb informació sobre la funció ajuda a entendre millor què fa.

Classe 2: Funcions I Programació modular. Funcions lambda

- Les funcions lambda són un tipus de funcions que no segueixen l'estàndard que hem vist fins ara. Són funcions molt simples que no necessiten ser definides formalment i no tenen nom. Poden tenir més d'un argument i són útils quan volem fer operacions senzilles. I.e: dins d'una funció més gran i complexa que no necessita tenir més complexitat:

```
def average_grade(self):  
    from functools import reduce  
  
    sum = reduce(lambda a, b: a + b, self.grade_list)  
    return sum / len(self.grade_list)
```

```
def double(x):  
    return x+x  
  
lambda x: x+x
```

Sintaxi → *lambda* argument(s): expressió

```
greet = lambda: print("Hello!")  
  
greet() # Hello!
```

L'expressió també pot ser una funció!

Classe 2: Funcions I Programació modular. Casting de variables.

- El tipus d'una variable és de gran importància, ja que és el paràmetre pel qual una funció pot acceptar, o no, un argument que se li passi. Els tipus més bàsics són `int()` pels enters, `float()` per decimals, `str()` per strings, `bool()` per True/False, `list()` per llistes i `dict()` per diccionaris.
- Quan tenim una variable, diem que el seu tipus és un embolcall (*wrap*) que fa que la resta del programa la reconeixi d'una certa manera.

```
# Exemple
x = 7

y = 99

x_str = "7"

y_str = "99"

fx.fancy_suma(x, y)
fx.fancy_suma(x_str, y_str)
```

La funció `fancy_suma` et suma *dos elements que siguin sumables*. La suma de strings a python és la seva concatenació, i per tant dona un resultat que pot ser inesperat o la causa d'un problema.

```
(gaudi2_dev) xavi@xav-Hummer:~/eriBlocD/classe_2_exercicis$ python ex_imports.py
La suma de 7 i 99 és: 106
La suma de 7 i 99 és: 799
```

Classe 2: Funcions I Programació modular. Casting de variables.

- Haver de fer *cast* de variables és quelcom poc comú, però el concepte és important per entendre que el tipus (type) d'una variable és mutable. Pot canviar en un punt determinat del programa, el pot modificar una funció, pot ser incorrecte quan l'agafem d'un arxiu extern, etc.

```
# Exemple
x = 7

y = 99

x_str = "7"
y_str = "99"

fx.fancy_suma(x, y)
fx.fancy_suma(int(x_str), int(y_str))
```

El comportament inesperat de l'exemple anterior es pot corregir si fem un *cast* de string a int mitjançant la funció `int()`. Això farà que el tipus de les variables `x_str` i `y_str` ara sigui enter i es puguin sumar com nombres (un `float()` també serviria, però el resultat seria lleugerament diferent).

```
(gaudi2_dev) xavi@xav-Hummer:~/eriBlocD/classe_2_exercicis$ python ex_imports.py
La suma de 7 i 99 és: 106
La suma de 7 i 99 és: 106
```


Classe 2: Funcions I Programació modular. Funcions estàndards de Python

- Programar consisteix en resoldre un problema de manera analítica mitjançant algorismes. Quantes més eines tinguem a la nostra disposició, millor podrem plantejar la nostra estratègia de resolució. També és important a tenir en compte que, si ja existeix una funció que fa el que nosaltres volem (i possiblement ho fa més eficientment), no cal reescriure-la.
 - Tot seguit us presentem un recull de funcions que us ajudaran a programar millor. Es troben totes al paquet estàndard de python i no necessiten ser importades, ja que es troben per defecte quan obrim qualsevol editor.
 - La seva documentació completa es pot trobar aquí: <https://docs.python.org/3/library/functions.html>
-
- `print()`: imprimeix per pantalla
 - `range()`: genera una llista de nombres
 - `type()`: retorna el tipus d'una variable (int, str, list...)
 - `len()`: retorna nombre d'elements en un conjunt
 - `enumerate()`: converteix una llista en una enumeració
 - `list()`: assigna el tipus "llista" a una variable
 - `dict()`: assigna el tipus "diccionari" a una variable
 - `int()`: assigna el tipus "enter" a una variable
 - `float()`: assigna el tipus "decimal" a una variable
 - `str()`: assigna el tipus "string" a una variable
 - `max()`: retorna el valor més gran d'un iterable
 - `min()`: ídem que `max()` però amb el valor mínim.

Classe 2: Funcions I Programació modular. Mètodes estàndards de Python

- Python també inclou mètodes que s'apliquen a llistes, strings, diccionaris, etc... que ens poden resultar molt útils per a operar amb aquests objectes.
- Tot seguit us presentem un recull de mètodes que us poden ajudar a programar millor. Us recomanem que les proveu i veieu què fan cadascuna d'elles.
- La seva documentació completa es pot trobar aquí: <https://www.w3schools.com/python/default.asp>
 - **append()**: afegeix un element a una llista
 - **pop()**: el·limina un element concret d'una llista (o l'últim)
 - **keys()**: retorna totes les keys d'un diccionari
 - **values()**: retorna tots els valors d'un diccionari
 - **slice()**: divideix un iterable (llista, tuple, string...) de llargada determinada
 - **join()**: converteix iterables en strings (i afegeix una string)
 - **split()**: separa una string en dues
 - **replace()**: en un iterable, canvia un element determinat per un altre
 - **sort()**: ordena una llista segons un criteri
 - **map()**: associa una funció concreta a un iterable i l'aplica a cada membre del conjunt
 - **zip()**: converteix dos iterables en un de sol
 - **reduce()**: aplica recursivament una funció sobre els elements d'un iterable

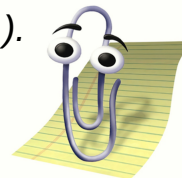
Classe 2: Funcions I Programació modular. Estratègies de debugging

- Tots cometem errors. Per sort, quan programem, aquests errors sempre es poden solucionar (més o menys fàcilment). El procés de descobrir per què un programa no funciona, arreglar-lo I assegurar que funciona correctament s'anomena *debugging*.
- Quan executem un programa I aquest funciona de manera inesperada, sabem que hi ha quelcom que no hem fet bé. La millor estratègia per a arreglar un *bug* és:
 - Llegir I entendre l'error que et proporciona el programa.
 - Dividir el codi en fragments petits que facin una funció concreta.
 - Monitoritzar el valor I tipus de totes les variables abans I després d'una operació (funció `print()`, `type()`, `len()`).
 - Comentar parts del codi fins que es trobi la mínima expressió del programa que funciona correctament.
 - Afegir el codi comentat fins que es trobi l'error. Aïllar-lo de nou I arreglar-lo.

Classe 2: Funcions I Programació modular. Estratègies de debugging

Tipus d'errors de Python I què signifiquen:

- **SyntaxError**: Clàssic quan s'aprèn a programar. Probablement una funció o variable no estigui ben escrita o s'hagi cridat malament.
- **ImportError**: Salta quan un mòdul no s'ha importat correctament, bé perquè està mal escrit, perquè l'ordinador no el troba (problemes amb el *path*) o perquè no existeix. De cara a minimitzar aquests errors, posa els mòduls que vols importar a la mateixa carpeta on tens els programa.
- **IndexError**: Molt comú quan es treballa amb llistes o iterables en general. Significa que quan estem iterant sobre un element, li estem demanant una posició que no existeix (l.e: demanar la posició 4 a la llista [1,2,3]).
- **KeyError**: Salta quan una clau no es troba en un diccionari.
- **OverflowError**: Error que apareix quan el número resultant d'una operació aritmètica es torna massa gran per a ser representat.
- **IndentationError**: Un altre clàssic quan es comença. Significa que no s'ha respectat la indentació dels blocs de codi (indentació dins un bucle *for*, després dels condicionals *if*, *else*,...).
- **TypeError**: Salta quan una variable té un `type()` que no concorda amb el que hauria de ser.



Classe 3: STDIN/STDOUT

Antigament, quan es volien entrar dades a un ordinador s'havien de fer servir targetes perforades que contenen la informació en un format que la màquina pogués entendre (~1950).

L'entrada de dades per teclat va ser un canvi radical que va facilitar enormement treballar amb ordinadors.

- El que entrem per teclat a l'ordinador s'anomena STDIN (Standard Input) i el que l'ordinador ens retorna (com ara el resultat d'un print) s'anomena STDOUT (Standard Output)



Classe 3: STDIN/STDOUT

El **STDIN** permet a l'usuari aportar informació al programa cada vegada que aquest s'executa. Mitjançant la funció **input()**, el valor que proporcionem s'assigna a una variable determinada.

ATENCIÓ: Els mètodes STDIN són poc fiables, ja que depenen de que no es cometi cap error a l'hora d'introduir dades per teclat (un error molt més comú del que es pot pensar). És molt més recomanable treballar amb dades extretes de fitxers.

El **STDOUT** escriu per pantalla el resultat de les operacions que fa l'ordinador. És la manera que té la màquina de comunicar-se amb nosaltres i hem de saber entendre què hi ha al STDOUT en tot moment.

És important durant el debug (per veure on falla el programa) o l'execució normal de programes (per a seguir el procés que estem executant)

Classe 3: STDIN/STDOUT. Exemple STDIN

Fent servir la funció **input()**, quan executem el programa, aquest pausarà la seva execució i ens demanarà que introduïm per teclat un valor que serà assignat a una variable. Hem de tenir en compte el **type()** de variable que entrem, ja que per defecte serà una string. Podem modificar-lo amb el cast corresponent.

```
# Taking input as string
color = input("What color is rose?: ")
print(color)

# Taking input as int
# Typecasting to int
n = int(input("How many roses?: "))
print(n)

# Taking input as float
# Typecasting to float
price = float(input("Price of each rose?: "))
print(price)
```

Output:

```
What color is rose?: red
red
How many roses?: 10
10
Price of each rose?: 15.50
15.5
```

Classe 3: Escriptura i lectura de fitxers

En comptes d'introduir dades per pantalla podem agafar-les d'un arxiu. També podem escriure els resultats d'un programa a un arxiu de text en comptes de fer que només ens doni el resultat per pantalla. La manera més eficient (generalment) de llegir un arxiu i guardar el seu contingut en una variable és aquesta:

```
with open('readme.txt') as f:  
    lines = f.readlines()
```

La funció `open()` obre un arxiu determinat. `Readlines()` llegeix cada línia de l'arxiu i la guarda dins la variable *lines*. *Lines* és una llista amb tot el text de l'arxiu `readme.txt`. Proveu-ho!

Classe 3: Escriptura i lectura de fitxers

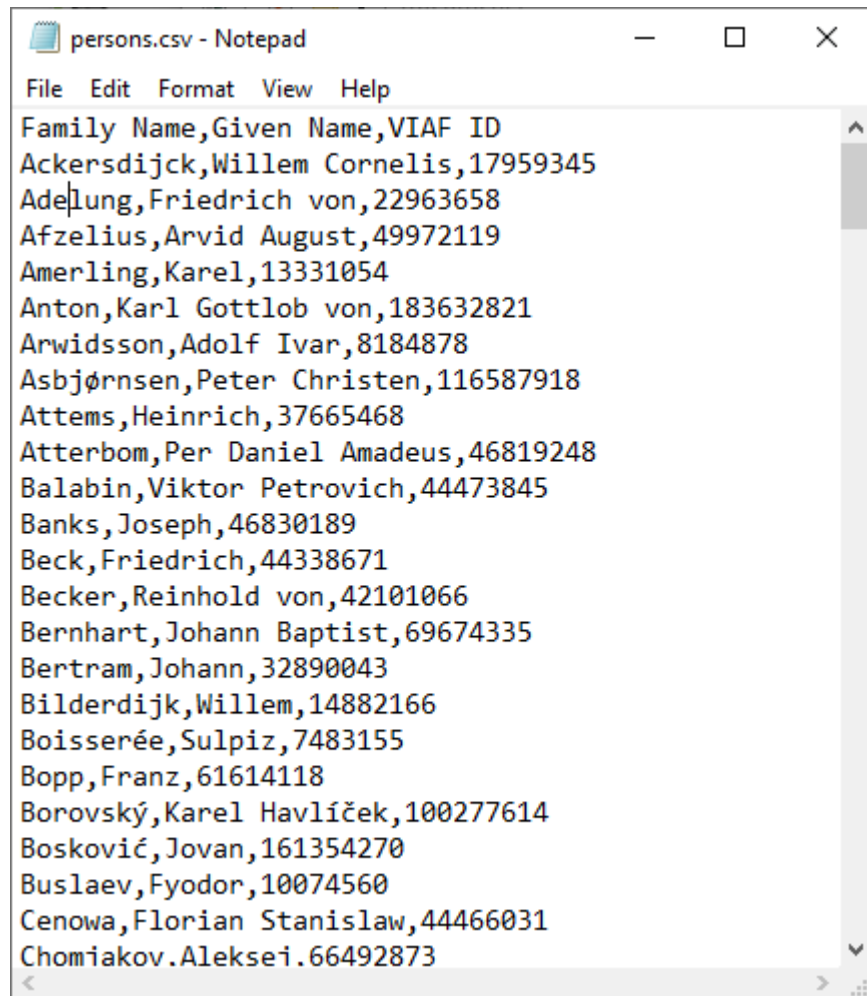
És útil poder guardar els resultats d'un programa en un arxiu. El mètode **write()** ens permet escriure un arxiu de text amb les dades que li proporcionem (en aquest exemple, les nostres dades estan a la llista *lines*):

```
lines = ['Readme', 'How to write text files in Python']
with open('readme.txt', 'w') as f:
    for line in lines:
        f.write(line)
        f.write('\n')
```

La funció **open()**, si no existeix l'arxiu que li donem, en crearà un de nou amb aquest nom. L'argument “w” ens indica que l'arxiu que obrirem és per a escriure-hi coses (“write”). També podem obrir-lo per a llegir-lo sense modificar-lo amb “r” (“read”). Proveu-ho!

Classe 3: Escriptura i lectura de fitxers. Format CSV

El format **CSV** (*Comma separated values*) ens permet guardar taules de valors en un format fàcil de recuperar. Si cada columna té un tipus de dada, el que tenim és bàsicament un full de càlcul que podem importar on volguem. Podem també transformar-los en diccionaris o altres tipus d'estructures de dades segons convingui.



```
persons.csv - Notepad
File Edit Format View Help
Family Name,Given Name,VIAF ID
Ackersdijck,Willem Cornelis,17959345
Adelung,Friedrich von,22963658
Afzelius,Arvid August,49972119
Amerling,Karel,13331054
Anton,Karl Gottlob von,183632821
Arwidsson,Adolf Ivar,8184878
Asbjørnsen,Peter Christen,116587918
Attems,Heinrich,37665468
Atterbom,Per Daniel Amadeus,46819248
Balabin,Viktor Petrovich,44473845
Banks,Joseph,46830189
Beck,Friedrich,44338671
Becker,Reinhold von,42101066
Bernhart,Johann Baptist,69674335
Bertram,Johann,32890043
Bilderdijk,Willem,14882166
Boisserée,Sulpiz,7483155
Bopp,Franz,61614118
Borovský,Karel Havlíček,100277614
Bosković,Jovan,161354270
Buslaev,Fyodor,10074560
Cenowa,Florian Stanislaw,44466031
Chomiakov,Aleksei,66492873
```

Classe 3: Mòduls d'importància en ciència: NumPy

NumPy és un mòdul molt eficient de python que eleva les llistes a *arrays*. A efectes pràctics, un array és una llista molt més eficient i amb més mètodes pròpis de NumPy. Transformar una llista a un array de numpy ens pot servir quan volem treballar amb matrius, llistes amb molts elements, volem llegir un csv directament amb numpy, etc... NumPy té mètodes molt semblants als de les llistes (https://www.w3schools.com/python/numpy/numpy_intro.asp), així com mètodes matemàtics de probabilitat, de nombres aleatoris...

Als exercicis podreu veure avantatges de numpy respecte llistes. Funciona especialment bé en conjunció amb altres mòduls de caire científic.

Classe 3: Mòduls d'importància en ciència: Math

El mòdul **math** conté una gran varietat de funcions comuns en l'àmbit científic. Hi ha funcions de trigonometria, estadística i distribucions, àlgebra, etc... Aquestes funcions estan optimitzades i convé fer-les servir.

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

Classe 3: Mòduls d'importància en ciència: Matplotlib

Matplotlib és un mòdul de Python que serveix per a fer gràfiques. Matplotlib ens dóna molta flexibilitat a l'hora de graficar diferents conjunts de valors i al treballar sobre conjunts de dades (fer regressions, tendències, histogrames, etc). Matplotlib té molts mètodes per a refinar gràfiques i fer anàlisis de resultats (https://www.w3schools.com/python/matplotlib_intro.asp), així com moltes opcions de formatat de gràfiques i tipus de gràfics diferents (línea, 2D i 3D, scatter, subplots, pie charts, etc...)

Matplotlib s'acostuma a fer servir juntament amb NumPy, ja que els valors que es a grafiquen acostumen a estar emmagatzemats en arrays de numpy.

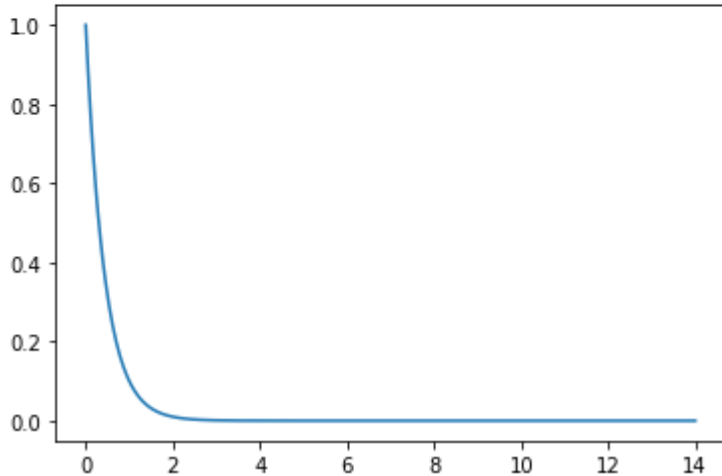
Classe 3: Mòduls d'importància en ciència: Matplotlib

```
import numpy as np  
pH = np.linspace(0.0, 14.0, 500)
```

```
H = [10**(-num) for num in pH]
```

```
plt.plot(pH, H)
```

```
[<matplotlib.lines.Line2D at 0x16aed269e08>]
```



Quan treballem amb matplotlib també farem servir altres mòduls per tal de preparar les dades que volem graficar.

Generalment, els dos arguments principals de `plt.plot` seran arrays de numpy o llistes.

Classe 3: Mòduls d'importància en ciència: SciPy

Afegir prac de fsolve amb eq de PH? Lligar el mòdul D a Fonaments de Química?
Altres vegades a fonaments no s'ha arribat al temari quan s'ha fet la pràctica a
ERI = problemes

Classe 3: Introducció a les classes i al OOP

Python és un llenguatge que ens permet fer servir **classes** i **objectes** per tal d'organitzar el nostre codi. El paradigma **OOP** (Object oriented programming) consisteix en agrupar les variables del nostre programa en classes segons les relacions que s'estableixin entre elles. La seva sintaxi és especial i s'ha de tenir en compte quan volem definir una classe (les classes van en majúscula). Les classes tenen **objectes** i **mètodes**: un objecte és un element d'aquella classe i un mètode és una funció que només es pot aplicar als objectes d'aquella classe.

Ho veurem més clarament amb un exemple:

- Definim la classe Person(), que tindrà dues propietats: nom i edat

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

La funció **__init__** és l'inicialitzador de la classe. Ens indica quines propietats tindrà cada objecte de la classe i garanteix que existeixin quan es crea un objecte.

Classe 3: Introducció a les classes i al OOP

- Ara que tenim una classe definida, podem crear **objectes** d'aquella classe.

```
persona_1 = Person("Carles", 23)  
persona_2 = Person("Sílvia", 24)
```

persona_1 i persona_2 són objectes de la classe Person. Aquesta és la sintaxi per a definir objectes d'una classe. El primer agument de l'inicialitzador és el nom i el segon l'edat. Podem modificar les propietats d'un objecte en qualsevol moment:

```
persona_1 = Person("Carles", 23)  
persona_2 = Person("Sílvia", 24)  
  
print(persona_1.name)  
  
persona_1.name = "Mohammed"  
  
print(persona_1.name)
```

Per accedir a les propietats d'un objecte, farem servir l'operador punt (.)

Classe 3: Introducció a les classes i al OOP. Mètodes de classe

- Dins d'una classe també podem definir **mètodes**, que són funcions exclusives pels objectes d'aquella classe. Els mètodes ens permeten modificar els objectes d'una classe i no es poden aplicar a cap altre element que no pertanyi a la classe.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def change_name(self, new_name):
        self.name = new_name

persona_1 = Person("Carles", 23)
persona_2 = Person("Sílvia", 24)

print(persona_1.name)  # Carles

persona_1.change_name("Mohammed")

print(persona_1.name)  # Mohammed
```

Per a aplicar un mètode a un objecte d'una classe, també fem servir l'operador punt (.)

Classe 3: Introducció a les classes i al OOP. Herència


```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def change_name(self, new_name):
        self.name = new_name

class Student(Person):
    def __init__(self, name, age, niu, grade_list):
        super().__init__(name, age)
        self.niu = niu
        self.grade_list = grade_list

    def average_grade(self):
        import functools

        sum = functools.reduce(lambda a, b: a + b, self.grade_list)
        return sum / len(self.grade_list)
```



The diagram shows an orange arrow pointing from the word 'Herència' to the 'Person' class in the code, indicating the concept of inheritance.

Herència

Podem crear una classe (o subclasse) que heredi propietats de la primera i que tingui mètodes propis. Per exemple, es pot definir la subclasse Student i dir que tots els Students també comparteixen les propietats de Person. Això ens permet crear una subclasse amb mètodes propis però que hereda les propietats de Person. Per a dir que una classe hereda les propietats d'una altra, posem el nom de la classe mare entre parèntesis quan definim la classe filla.

Classe 3: Introducció a les classes i al OOP. Herència

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def change_name(self, new_name):
        self.name = new_name

class Student(Person):
    def __init__(self, name, age, niu, grade_list):
        super().__init__(name, age)
        self.niu = niu
        self.grade_list = grade_list

    def average_grade(self):
        import functools

        sum = functools.reduce(lambda a, b: a + b, self.grade_list)
        return sum / len(self.grade_list)
```

Name i age, heretades de Person

Funció Lambda (funció simple sense definir)

```
person_3 = Student("Kyle", 22, 1329983, [5, 6, 7])
person_3.change_name("Johnny")

print(person_3.average_grade()) # 6.0
```

El mètode `change_name()` continuarà funcionant per als objectes de la classe `Student`, però el mètode `average_grade()` no funcionarà amb objectes de `Person` que no siguin `Student`. Un `Student` té totes les propietats de `Person` a més de les pròpies de `Student`. El mètode **`super()`** ens assegura que `Student` hereta els mètodes i propietats de `Person`.

Classe 4: Classes i OOP aplicades a la Química computacional

La **química computacional** és la branca de la química que engloba tots aquells procediments que estudien sistemes químics amb l'ajut d'eines de computació (ordinadors, supercomputadors, programes de simulació, etc...).

Per a poder treballar amb sistemes químics des d'un ordinador, s'ha de crear una interfase (programa) que ens permeti interactuar amb el sistema real (modificar-lo, obtenir informació sobre les seves propietats...).

Un exemple que veurem a continuació és l'ús d'una **API (application program interface)** per a treballar amb molècules. Aquest programa, anomenat **Rdkit**, ens proporciona un seguit de classes i funcions per a treballar amb dades reals de molècules (obtingudes del Protein Data Bank).



Classe 4: Classes i OOP aplicades a la Química computacional

Rdkit té tot un seguit de mòduls que permeten partir d'una estructura tridimensional obtinguda per difracció de raigs X (entre altres mètodes) i obtenir un objecte d'una classe amb diverses propietats que podem analitzar o modificar. Veiem-ho amb un exemple: un arxiu pdb (coordenades atòmiques) té aquesta pinta:

```
ATOM      1  N   ILE A  41      448.438  313.731  366.583  1.00125.46  N
ATOM      2  CA  ILE A  41      449.227  314.793  365.972  1.00125.46  C
ATOM      3  C   ILE A  41      450.200  315.373  366.989  1.00125.46  C
ATOM      4  O   ILE A  41      449.789  315.965  367.987  1.00125.46  O
ATOM      5  CB  ILE A  41      448.327  315.892  365.393  1.00125.46  C
ATOM      6  CG1 ILE A  41      447.459  315.327  364.267  1.00125.46  C
ATOM      7  CG2 ILE A  41      449.168  317.060  364.899  1.00125.46  C
ATOM      8  CD1 ILE A  41      446.341  316.251  363.836  1.00125.46  C
ATOM      9  N   LYS A  42      451.495  315.197  366.730  1.00131.57  N
ATOM     10  CA  LYS A  42      452.523  315.626  367.672  1.00131.57  C
ATOM     11  C   LYS A  42      453.547  316.534  367.002  1.00131.57  C
ATOM     12  O   LYS A  42      453.397  316.893  365.830  1.00131.57  O
```



Conté el número de l'àtom dins la molècula, el tipus d'àtom, el residu al qual pertany, el número de residu i les coordenades x y z, entre altres.

Classe 4: Classes i OOP aplicades a la Química computacional

QUANT VOLEM EXPLICAR DE RDKIT?

