Info 4 : Vérifier et approfondir nos connaissances en python
Proposition de correction

Contents

1 Épreuves et programme **1**

2 Bases de python **1**

3 Quelques notions plus complexes **12**

1 Épreuves et programme

Après une année (au minimum !) à pratiquer python, il est temps de faire le point sur les épreuves du concours et ce qu'il faut maîtriser pour réussir sereinement.

L'informatique sera évalué aux concours à l'écrit, dans 2 épreuves :

- l'épreuve d'informatique "pure", d'une durée de 3h, et de coefficient 4. Cette épreuve est commune avec la filière TSI, et repose uniquement sur le programme d'informatique.
- l'épreuve de modélisation, d'une durée de 4h et de coefficient 7. Cette épreuve est commune avec la filière PC, nécessite des des connaissances des programmes de physique et/ou chimie, et utilise l'informatique comme support. Cette épreuve met généralement en œuvre des méthodes d'Euler avec des tableaux numpy.

Il est important de noter que la somme des coefficients de ces deux épreuves dépasse celui de l'épreuve de mathématiques à l'écrit, pour seulement 2h de préparation par semaine ! Il est donc primordial que vous preniez le temps de re-travailler le travail effectué en classe par vous-même entre chaque séances. Il est aussi important de rappeler que ces épreuves se déroulent sur **papier** : il faut prendre l'habitude, lors des séances avec ordinateur, de raisonner au maximum à l'aide d'un brouillon, et non tenter *n'importe quoi* sur l'ordinateur et voir le résultat...

2 Bases de python

Lors de cette séance, nous allons faire le point sur les connaissances nécessaires pour l'épreuve d'informatique, en rapport avec le programme. Cette séance se déroulera de façon exceptionnelle **sans ordinateur** !

Commençons avec de brefs rappels, à réviser si besoin avec les cours de l'année dernière, agrémentés de quelques remarques :

- **Représentation des nombres** : écriture en binaires des nombres entiers, des flottants
- **Bases de données**
- **Écriture d'un code python** : on écrit un code python dans un logiciel nommé *éditeur* (*Thonny* ou ici *Jupyter lab*), ce code est ensuite interprété et le "dialogue" se fait via la *console* ou *shell*. Tout ce qui dans le code est écrit à la suite du caractère dièse #, sur la même ligne, **n'est pas interprété** : il s'agit de **commentaires**.
- **Calculs numériques** en python : +,-,*,/,**, et, pour les entiers // et %
- **Tests et logiques** en python : bool, et opération not or et and + comparaisons == != < > <= >= .

Question : A votre avis, que retourne les codes suivants ? **Ne pas les tester sur ordinateur !**

[2]:
```
type(2 == 3)
```

[2]: bool

[3]:
```
print(True == 1)
print(False == 0)
```

```
True
True
```

[4]:
```
print(2 == 3 and 2 > 1)
print(2 == 3 or 2 > 1)
```

```
False
True
```

[5]:
```
print(2 == 3)
print(2 != 3)
print(not 2 == 3)
print( 2<= 3)
```

```
False
True
True
True
```

[6]:
```
from random import uniform
x = uniform(0, 10) # nombre réel aléatoire compris entre 0 et 10
print( x <= 5 and x > 5)
print( x <= 5 or x > 5)
```

```
False
True
```

Remarque : dans l'expression `A and B`, `A` est évaluée en premier, et si le résultat est `False`, `B` n'est pas évaluée, ce qui peut avoir son utilité ! En voici un exemple :

[1]:
```
L = [2, 3, 1, 4]
while L != [] and L[-1] > 0 :
    L.pop()
```

Ce code ne fonctionnerait pas en inversant l'ordre des tests dans la boucle `while`, pourquoi ?

Toujours dans les tests : if avec possiblement else et même des elif

[7]:
```
x = uniform(0, 10)
if x < 3 :
    print("nombre inférieur strictement à 3")
elif x <= 7 :
    print("nombre compris entre 3 et 7 (inclus)")
else :
    print("nombre supérieur strictement à 7")
```

```
nombre supérieur strictement à 7
```

- **Variables** : une variable en informatique (en particulier ici en python) est un objet légerement différent des variables mathématiques. Alors qu'en mathématiques, une variable désigne une lettre qui peut souvent prendre n'importe qu'elle valeur (dans un intervalle donné), ici, en python il faut plutôt voir une variable comme un "nom" associé un objet informatique (plus précisément : associé à une adresse mémoire menant à cet objet). On peut faire varier l'objet associé à ce nom, ce qui explique la notion de variable. Cet objet peut être un nombre entier, un flottant, un booléen, une liste, etc. On **affecte** à une variable cet objet grâce au signe "=" : il est important ici de noter que, contrairement au signe égal mathématique, il n'y a pas ici de symétrie.

[8]:
```
a = 2
print(a)
```

```
2
```

Mais :

[9]:
```
2 = b
```

```
  File "/tmp/ipykernel_90562/2457820290.py", line 1
    2 = b
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='
```

Python, lors de l'interprétation du code, attribue automatiquement un **type** à chaque variable (ce qu'on nomme le *typage dynamique*):

[55]:
```
a = 2
b = 2.
c = a == b
d = 'python'
```

```
print("Type de a : ",type(a))
print("Type de b : ",type(b))
print("Type de c : ",type(c))
print("Type de d : ",type(d))
print(" a == b ? : ",c)
```

```
Type de a :  <class 'int'>
Type de b :  <class 'float'>
Type de c :  <class 'bool'>
Type de d :  <class 'str'>
 a == b ? :  True
```

Il faut être conscient que les concepteurs du langage python ont fait des choix, qui peuvent être différents de ceux d'autres langages que vous pouriez étudier plus tard.

- **Bibliothèques ou modules** : il est fréquent d'utiliser des fonctions python situé dans des modules (donc non présente dans la version de "base" de python), à l'aide de la commande import. Import d'une seule fonction d'un module :

[39]:
```
from random import randint
print("Nombre entier aléatoire entre 0 et 9 : ",randint(0,9))
```

```
Nombre entier aléatoire entre 0 et 9 :  9
```

Pour importer plusieurs fonctions d'un module :

[40]:
```
from random import randint, uniform
```

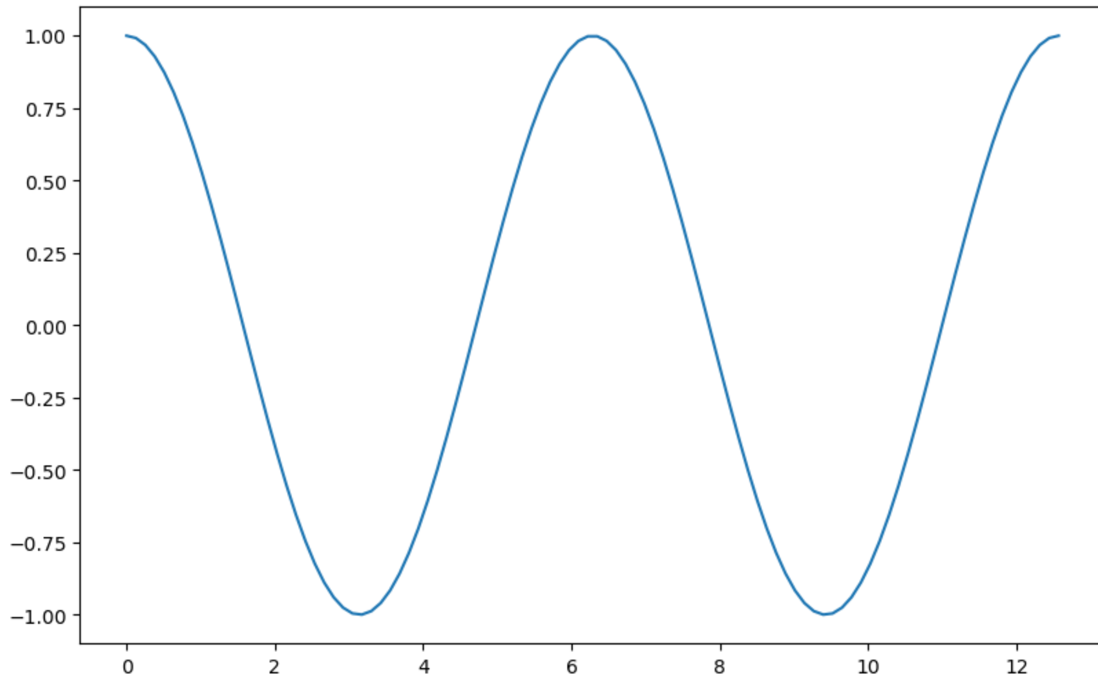
Pour importer toutes les fonctions d'un module :

[3]:
```
import math
math.cos(1)
```

[3]:
```
0.5403023058681398
```

Pour des modules au nom complexe, on peut utiliser un *alias*, permettant de ne pas écrire en entier le nom du module à chaque usage d'une fonction, comme ci-dessous avec *plt* et *np* :

[4]:
```
import matplotlib.pyplot as plt
import numpy as np

ratio = 1.2 # ratio de taille entre fig et texte (légende et axes), par défaut 1
plt.figure(figsize=(8*ratio,5*ratio),dpi = 100)
X = np.linspace(0,4*math.pi,100)
Y = np.cos(X)
plt.plot(X,Y)
plt.show()
```



On ne profite pour rappeler une fonciton très utile, la fonction help() :

```
[ ]: help(np.linspace)
```

Ou bien :

```
[5]: ? np.linspace
```

```
Signature:
 np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
Call signature:  np.linspace(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function linspace at 0x7ff5299c76a0>
File:            /usr/lib64/python3.12/site-packages/numpy/core/function_base.py
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
```

```
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

.. versionchanged:: 1.20.0
    Values are rounded towards ``-inf`` instead of ``0`` when an
    integer ``dtype`` is specified. The old behavior can
    still be obtained with ``np.linspace(start, stop, num).astype(int)``

Parameters
----------
start : array_like
    The starting value of the sequence.
stop : array_like
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded.  Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50. Must be non-negative.
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
dtype : dtype, optional
    The type of the output array.  If `dtype` is not given, the data type
    is inferred from `start` and `stop`. The inferred dtype will never be
    an integer; `float` is chosen even if the arguments would produce an
    array of integers.

    .. versionadded:: 1.9.0

axis : int, optional
    The axis in the result to store the samples.  Relevant only if start
    or stop are array-like.  By default (0), the samples will be along a
    new axis inserted at the beginning. Use -1 to get an axis at the end.

    .. versionadded:: 1.16.0

Returns
-------
samples : ndarray
    There are `num` equally spaced samples in the closed interval
```

```
    ``[start, stop]`` or the half-open interval ``[start, stop)``
    (depending on whether `endpoint` is True or False).
step : float, optional
    Only returned if `retstep` is True

    Size of spacing between samples.


See Also
--------
arange : Similar to `linspace`, but uses a step size (instead of the
         number of samples).
geomspace : Similar to `linspace`, but with numbers spaced evenly on a log
            scale (a geometric progression).
logspace : Similar to `geomspace`, but with the end points specified as
           logarithms.
:ref:`how-to-partition`

Examples
--------
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x…>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x…>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
```

```
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

- **Tableau 1d** : généralement, en python, on utilise des listes pour créer des tableaux unidimensionnels. Dans la suite, on rappelle quelques commandes à connaître à propos des listes, en commençant par la génération d'une liste :

[46]:

```python
# Création :
L = []

# Ajout d'éléments :
L.append(1)
L.append(2)
L.append(3)

print(L)
```

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

On peut aussi créer une liste contenant déjà des valeurs, puis les modifier :

[47]:

```python
M = [0] * 3
print(M)

M[0] = 4
M[1] = 5
M[2] = 6
print(M)
```

```
[0, 0, 0]
[4, 5, 6]
```

On rappelle quelques commandes utiles :

[48]:
```
# Taille d'une liste :
print(len(L))

# Accès au dernier élément (sans avoir besoin de connaître la taille) :
print(L[-1])

# Concaténation :
N = L + M
print(N)

# Extraction
print(N[1:4]) # éléments de N de l'indice 1 (le 2eme) inclus à l'indice 4 (le␣
↪5ème) exclu
print(N[:4]) # éléments de N du début (indice 0) à l'indice 4 exclu, équivalent␣
↪à N[0:4]
print(N[1:]) # éléments de N de l'indice 1 inclus à la fin, équivalent à N[1:
↪len(N)]

# Retranchement du dernier élément :
N.pop()
print(N)
```

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

IMPORTANT : Un rapport du jury du concours récent liste quelques erreurs récurrentes : utilisation de `L[i]=x` pour une liste vide, utilisation de `L=L+x` au lieu de `L=L+[x]` et utilisation de `L=L.append(x)`.

Pour un tableau (undimimensionnel ou de dimension supérieur…) contenant uniquement des flottants, dans le but de calculs numériques, on privilégiera les tableaux numpy. Normalement les commandes liés au tableaux numpy doivent être rappelées dans les énoncés… Au cas où, on créé généralement un tableau numpy en le remplissant de 0, puis on le modifie comme une liste :

[49]:
```
N = np.zeros(5)
N[3] = 4
print(N)
```

```
[0. 0. 0. 4. 0.]
```

Attention pas de `append` en numpy !

- **Boucles** : pour itérer (c'est-à-dire répéter) une instruction de nombreuses fois, on utilise des boucles. Attention alors à bien respecter l'**indentation**. Lorsqu'on connait le nombre d'itérations, on utilise généralement les instructions `for .... in ... .` Sinon : `while ...`

- **Boucles for et listes** : Il y a deux principale méthodes pour parcourir une liste. On peut soit parcourir la liste en suivant les indices :

[50]:
```
for i in range(len(L)):
    print(L[i])
```

```
1
2
3
```

Mais on peut aussi directement porcourir les éléments :

[51]:
```
for elem in L :
    print(elem)
```

```
1
2
3
```

Remarque : `elem` est ici une variable “muette”, comme l’était `i`, c’est-à-dire que son nom n’a aucune importance, tant qu’on utilise le même dans les 2 lignes.

Cela peut aussi servir à construire une liste :

[52]:
```
L1 = [i for i in range(10)]
L2 = [elem**2 for elem in L1]
print(L1)
print(L2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Profitons-en pour rappeler deux algorithmes essentiels : la recherche de l’indice d’un élément dans une liste, et la recherche du maximum :

[53]:
```
from random import randint

L = [randint(1,100) for i in range(20)]
L
```

[53]:
```
[95, 4, 3, 19, 12, 41, 33, 84, 67, 16, 29, 80, 26, 78, 100, 39, 63, 26, 3, 78]
```

[54]:
```
a = 79 # nombre entier
# Code permettant de tester si l'élément a est présent dans la liste. Si oui,␣
 ↪afficher son indice :
Indice_a = []
for i in range(len(L)):
    if a == L[i] :
        Indice_a.append(i)
if Indice_a == []:
    print(str(a)+" n'est pas dans la liste")
```

```
else :
    print("Indice(s) où trouver a dans la liste : ",Indice_a)
```

```
79 n'est pas dans la liste
```

[55]:
```
# Code permettant de déterminer le maximum de la liste, ainsi que son indice :
m = L[0] # max
ind_m = 0 # indice max

for i in range(1,len(L)):
        if L[i] > m:
            m = L[i]
            ind_m = i
print("Maximum : "+str(m)+" à la position "+str(ind_m))
```

```
Maximum : 100 à la position 14
```

- **Dichotomie** : Il faut maitriser cette méthode qui peut intervenir de différents algorithmes. Exemple : algorithme permettant de trouver l'indice auquel il faut insérer un élément dans une liste de nombre strictements croissants :

[1]:
```
from random import randint
L = [randint(1,100) for i in range(20)]
L.sort()
print(L)
```

```
[10, 11, 16, 17, 22, 26, 30, 38, 40, 41, 52, 52, 55, 61, 71, 72, 81, 87, 95,
100]
```

[4]:
```
a = 23 # nombre entier
# Code "naïf" permettant de déterminer l'indice où insérer a dans L en␣
↪respectant l'ordre
i = 0
while i < len(L) and a > L[i]: # Ici l'ordre est important dans le and
    i += 1
print("L'élément a doit être inséré entre les positions ",i-1," et ", i,".")
```

```
L'élément a doit être inséré entre les positions  4  et  5 .
```

[58]:
```
# Code utilisant la dichotomie permettant de déterminer l'indice où insérer a␣
↪dans L en respectant l'ordre

deb = 0 # début de portion de liste étudiée
fin = len(L) - 1 # fin de portion de liste étudiée
while fin - deb > 1 : # tant que l'intervalle étudié est plus long que 1
    m = (deb + fin) // 2 # milieu de l'intervalle
    if L[m] > a :
        fin = m
    else :
```

```
        deb = m

# Il reste trois positions possibles :
if a < L[deb] :
    print(deb)
elif a > L[fin] :
    print(fin+1)
else :
    print(fin)
```

```
14
```

- **Récursivité**
- **Fonctions** : Attention à bien maîtriser la syntaxe complète (`def`, `:`, `return` si nécessaire), et à bien respecter l'**indentation** !
- **Structures de données plus complexes** : tableau 2d (liste de listes, ou tableau numpy), dictionnaire
- **Méthodes numériques** : calcul d'intégrales / Méthode d'Euler (ordre 1 et 2) / Méthode de Gauss / Interpolation polynomiale de Lagrange

3 Quelques notions plus complexes

Dans la suite, nous allons voir ou revoir des notions plus complexes, généralement seulement abordées surperficiellement jusqu'à présent :

- **Effets de bord et Objets mutables / non mutables** : revenons à un problème déjà abordé l'année passée

[59]:
```
a = 3
b = a
b = 2
print("a :",a,", b :",b)
```

```
a : 3 , b : 2
```

Mais pour des listes :

[60]:
```
L = [1,2,3,4]
M = L
M[2] = 5
print(L)
```

```
[1, 2, 5, 4]
```

Cette différence de comportement, nommé *effet de bord*, s'explique par le fait que les nombres (int, float, …) sont des objets python dits *non mutables*, où *immutables*, alors que les listes sont des objets *mutables*.

Une variable non mutable est non modifiable : plus précisement, si on souhaite modifier la valeur assignée à cette variable, il faut refaire une affectation, cela revient à créer une nouvelle variable, dans un nouvel emplacement mémoire, avec le même nom que la précédente. Avec l'exemple précédent, en utilisant la fonction `id` permettant d'observer l'emplacement en mémoire des variables :

[61]:
```
a = 3
b = a
print(id(a),id(b))
```

```
139696778412336 139696778412336
```

Les variables a et b pointent donc vers le même emplacement mémoire, et donc le même objet, le nombre 3. Continuons :

[62]:
```
b = 2
print(id(a),id(b))
```

```
139696778412336 139696778412304
```

La valeur 2 a été ré-affectée à la variable b : b pointe vers une nouvel emplacement mémoire, différent du précédent, qui contenait 3.

Pour des listes, objets *mutables*, donc modifiables sans changer d'emplacement mémoire :

[63]:
```
L = [1,2,3,4]
M = L
print(id(L),id(M))
```

```
139695791063552 139695791063552
```

Sans surprise, les deux variables pointent vers le même emplacement mémoire, et donc le même objet constitué de 4 nombres.

[64]:
```
M[2] = 5
print(id(L),id(M))
```

```
139695791063552 139695791063552
```

M est une variable mutable : on peut modifier l'objet situé à l'emplacement pointé par M. L pointe encore vers le même emplacement que M, et donc vers le même objet modifié.

Pourquoi les listes sont *mutables* ? : c'est en réalité une caractéristique forte de cet objet python, et qui permet d'ajouter des variables à ces listes *à moindre coût* coût temporel, c'est-à-dire que l'opération est *rapide*), à l'aide de la fonction `append`, sans faire trop de modifications en mémoire (dans une certaine mesure seulement…). Les dictionnaires sont aussi des objets *mutables*.

Si on souhaite un ensemble de variables semblable à une liste, mais non mutables, on peut utiliser des **tuples** :

[65]:
```
T = (1,2,3,4)
type(T)
```

[65]: tuple

[66]:
```
T[3]
```

[66]: 4

Mais on ne peut alors pas le modifier :

[67]:
```
T[3] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_90562/2993086036.py in <module>
----> 1 T[3] = 0

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

[68]:
```
T.append(0)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_90562/100978339.py in <module>
----> 1 T.append(0)

AttributeError: 'tuple' object has no attribute 'append'
```

Une chaîne de caractères (type : *str*) est aussi un objet **non mutable** : malgré de nombreuses commandes similaires à celles des listes (`len`, accès à un élément), on ne peut pas la modifier, ni utiliser `append`.

[2]:
```
Chaine = 'Vive la TPC'
# Commandes possibles :
print(len(Chaine))
print(Chaine + '2 !') # Concaténation
print(Chaine[3]) # Accès à un élément
# Commandes impossibles
Chaine.append('!')
Chaine[3] = 'd'
```

```
11
Vive la TPC2 !
e
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[2], line 7
      5 print(Chaine[3]) # Accès à un élément
```

```
      6 # Commandes impossibles
----> 7 Chaine.append('!')
      8 Chaine[3] = 'd'

AttributeError: 'str' object has no attribute 'append'
```

Lorsqu'on utilise des listes, et qu'on veut garder une copie d'une telle liste, il faudra donc, du fait de son caractère **mutable**, en faire une copie :

[69]:
```
L = [1,2,3,4]

M = L.copy()
print(id(L),id(M))
```

```
139695792964992 139695792974784
```

On peut alors modifier M sans modifier L, et vis-versa.

- **Portée lexicale de python et Appel de fonction par valeur** : Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche en premier lieu la valeur définie à l'intérieur de la fonction ou en argument et, s'il ne la trouve pas, il ira chercher dans les variables préalablement définie avant l'exécution de la fonction. Voyons des exemples, mais auparavant effaçons toutes les variables préalablement définies :

[ ]:
```
%reset
```

[71]:
```
def fa():
    return a % 2 == 0

def fb(b):
    return b % 2 == 0
```

Ces fonctions ont pour but de ... ??? Testons :

[72]:
```
fa()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_90562/478878392.py in <module>
----> 1 fa()

/tmp/ipykernel_90562/3308266637.py in fa()
      1 def fa():
----> 2     return a % 2 == 0
      3
      4 def fb(b):
      5     return b % 2 == 0
```

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

Logique, a n'est une variable ni définie dans la fonction, ni définie précedemment, on ne peut donc pas éxécuter la fonction fa. Mais :

[73]:
```
print(fb(45))
print(b)
```

```
False
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_90562/450182923.py in <module>
      1 print(fb(45))
----> 2 print(b)

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

La fonction fb peut être exécutée, car elle prend un argument qu'on lui fournit, argument qui sera une variable b définie **localement** (uniquement dans l'éxécution de la fonction). En dehors de l'éxécution de la fonction, b n'est pas définie. Poursuivons en définissant des variables a et b avant l'éxécution des fonctions :

[74]:
```
a = 45
print(fa())
print(fb(a))
```

```
False
False
```

L'exécution des deux fonctions est possible. Par contre :

[75]:
```
b = 45
print(fb())
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_90562/3242630754.py in <module>
      1 b = 45
----> 2 print(fb())

TypeError: fb() missing 1 required positional argument: 'b'
```

L'éxécution “devrait” ici être possible, mais python la bloque car il attend un argument pour la fonction. Testons avec un autre argument :

[76]:
```
b = 45
print(fb(46))
```

```
True
```

On voit donc que python privilégie la variable local : il ne va donc chercher une variable en-dehors de la fonction que si celle-ci n'est pas définie localement. Terminons enfin avec ce test "idiot" :

[77]:
```
def fc(c):
    c = 46
    return c % 2 == 0

c = 45
print(fc(c),c)
```

```
True 45
```

On voit encore une fois que python a privilégié la variable locale, et n'a pas modifié la variable définie en-dehors de la fonction. Cela peut aussi se voir en parlant d'**appel de fonction par valeur** : L'exécution de f (x) évalue d'abord x puis exécute f avec la valeur calculée de x. Voyons cela :

[78]:
```
def f(a):
    a = a % 2
    return a == 0

a = 45
print(f(a),a)
```

```
False 45
```

On remarque ici que la fonction f s'est éxécutée non pas avec la **variable a** (qui aurait alors changé de valeur pour … ?), mais avec sa **valeur 45**. Ainsi a est inchangée.

Comme on peut s'en douter, ces comportements sont différents pour des listes, objets *mutables* :

[79]:
```
def fL(L):
    L[0] = 0
    return L
```

[80]:
```
L = [1,2,3,4]

print(fL(L),L)
```

```
[0, 2, 3, 4] [0, 2, 3, 4]
```

Ce comportement est finalement logique si on se rappelle que L ne désigne en réalité qu'un emplacement mémoire : la fonction `fL` se rend à cette emplacement, et modifie le premier objet pour lui donner la valeur 0. Comme la variable L pointe toujours vers ce même objet, l'affichage de L tient compte de la modification. On peut donc même simplifier la fonction :

[81]:
```
def fL(L):
    L[0] = 0

L = [1,2,3,4]

fL(L)
print(L)
```

```
[0, 2, 3, 4]
```

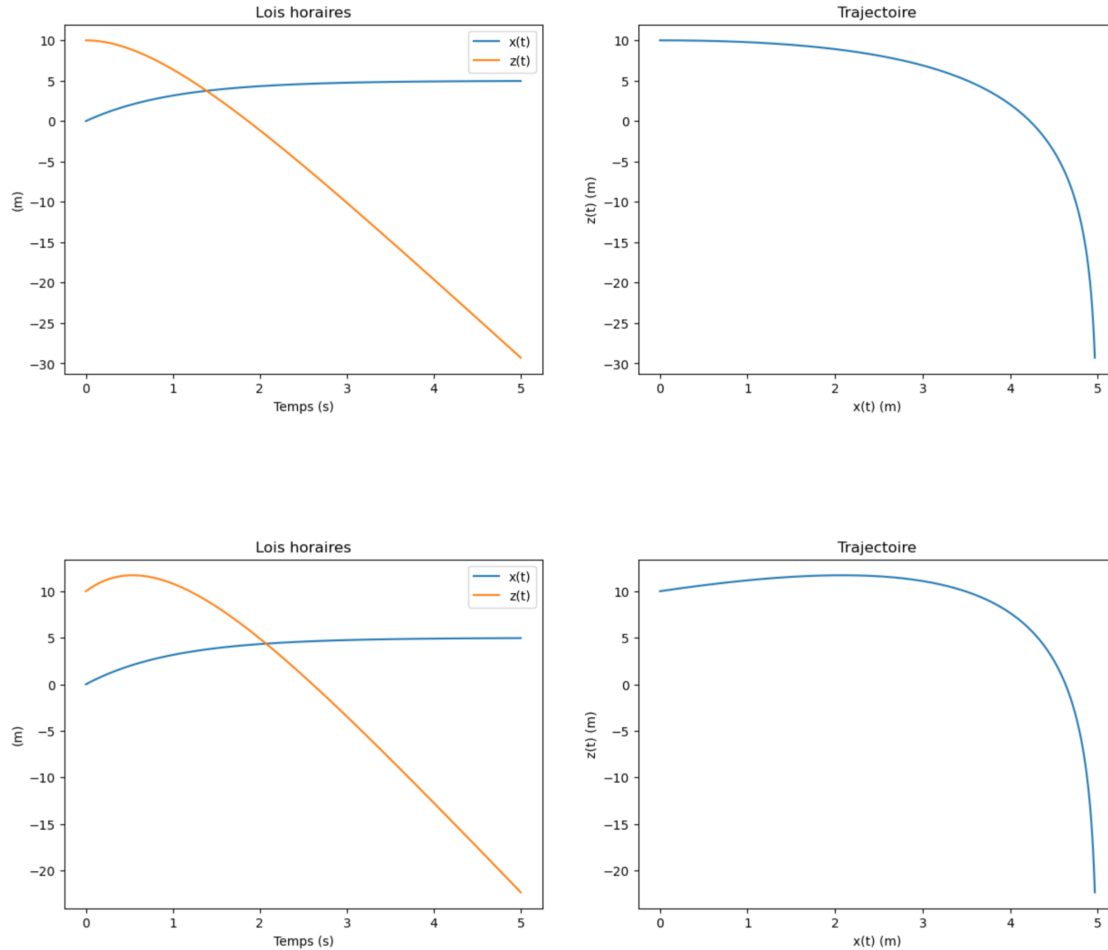
- **Fonctions et arguments** : En prenant en compte ce qui vient d'être décrit (portée lexicale), on peut se demander, lors de l'écriture d'une fonction, quelles données mettre en argument, ou quelles sont celles à définir a l'intérieur (localement) ou à l'extérieur (globalement) de la fonction. Ces choix dépendent en partie des habitudes et préférences de chacun.

Considérons l'exemple suivant : on souhaite réaliser le graphe de la trajectoire d'un mobile en chute dans un champ de pesanteur uniforme, soumis à une force de frottements fluide $\vec{f} = -\alpha\vec{v}$. Voici une partie du code :

[52]:
```
def calcul_traj(conditions_initiales) :
    """ Fonction retournant les tableaux des temps, abscisses et ordonnées pour␣
↪le problème de la chute libre avec frottements fluide
    conditions_initiale doit être un tuple contenant, dnas cet ordre : abscisse␣
↪initiale, ordonnée initiale, vitesse horizontale intiale, vitesse verticale␣
↪initiale"""

    g = 9.81
    xi, zi, vix, viz = conditions_initiales

    # fonction calculant l'abscisse :
    def x(t):
        return tau * vix * ( 1 - exp(-t / tau)) + xi

    # fonction calculant l'ordonnée :
    def z(t):
        return tau * ( viz + g * tau ) * ( 1 - exp(-t / tau))  - g * tau * t +␣
↪zi

    # Création des tableau contenant les valeurs numériques
    T = np.linspace(0, temps_final, N)
    X = [x(t) for t in T]
    Z = [z(t) for t in T]

    return T, X, Z

def graphe_traj(conditions_initiales) :
    """ Fonction affichant les graphes de slois hoarires et trajectoires
```

```
    conditions_initiale doit être un tuple contenant, dnas cet ordre : abscisse␣
↪initiale, ordonnée initiale, vitesse horizontale intiale, vitesse verticale␣
↪initiale"""

    T, X, Z = calcul_traj(conditions_initiales)

    plt.figure(figsize=(largeur,hauteur),dpi = 100)

    plt.subplot(121)
    plt.plot(T,X,label='x(t)')
    plt.plot(T,Z,label='z(t)')
    plt.title('Lois horaires')
    plt.xlabel('Temps (s)')
    plt.ylabel('(m)')
    plt.legend()

    plt.subplot(122)
    plt.plot(X, Z)
    plt.title('Trajectoire')
    plt.xlabel('x(t) (m)')
    plt.ylabel('z(t) (m)')
    plt.show()
```

[54]:
```
import numpy as np
import matplotlib.pyplot as plt
from math import exp

# Paramètres du graphe :
largeur = 15
hauteur = 5

# Paramètres physiques :
masse = 1
alpha = 1 # coefficient de frottements
tau = masse / alpha # temps caractéristique

# Paramètres modélisation
temps_final = 5
N = 100 # Nombre de points

# Affichage de graphes :
graphe_traj((0,10,5,0))
graphe_traj((0,10,5,7))
```



Sur cet exemple, on peut voir que *largeur*, *hauteur*, *masse*, *alpha*, *temps_final* et *N* sont définies comme des variables globales : on peut modifier leurs valeurs en modifiant le script, sans avoir besoin de modifier les fonctions. Ces variables sont utilisées dans les fonctions *calcul_traj* et *graphe_traj*. On peut ainsi aisément modifier leurs valeurs sans rentrer dans le code complexe des fonctions : même un utilisateur n'ayant pas réalisé ce code pourra l'adapter à son problème.

Il a été choisi de passer la grandeur *conditions_initiales* directement en argument des fonctions : il s'agit sûrement de la grandeur dont la variation nous intéresse le plus, la grandeur la plus à même d'être variable, toujours sans modifier les fonctions. Ainsi on présente aisément des graphes pour lesquels ces valeurs sont différentes.

En g est ici une variable locale, dont la variation ne nous intéresse pas : le calcul sera toujours effectué avec la même valeur numérique. On pourrait tout aussi bien la remplacer dans la fonction directement par sa valeur, mais cela rendrait le code moins lisible.

- **Intérêts des tuples** : Les *tuples* sont généralement utilisés pour effectué de multpiles affectations en simultané :

[82]:
```
a, b, c = 1, 2, 3
print(a,b,c)
```

```
1 2 3
```

Il faut bien comprendre ici qu'on affecte au tuple `(a, b, c)` le tuple `(1, 2, 3)`. Ainsi on peut s'en servir pour inverser des valeurs :

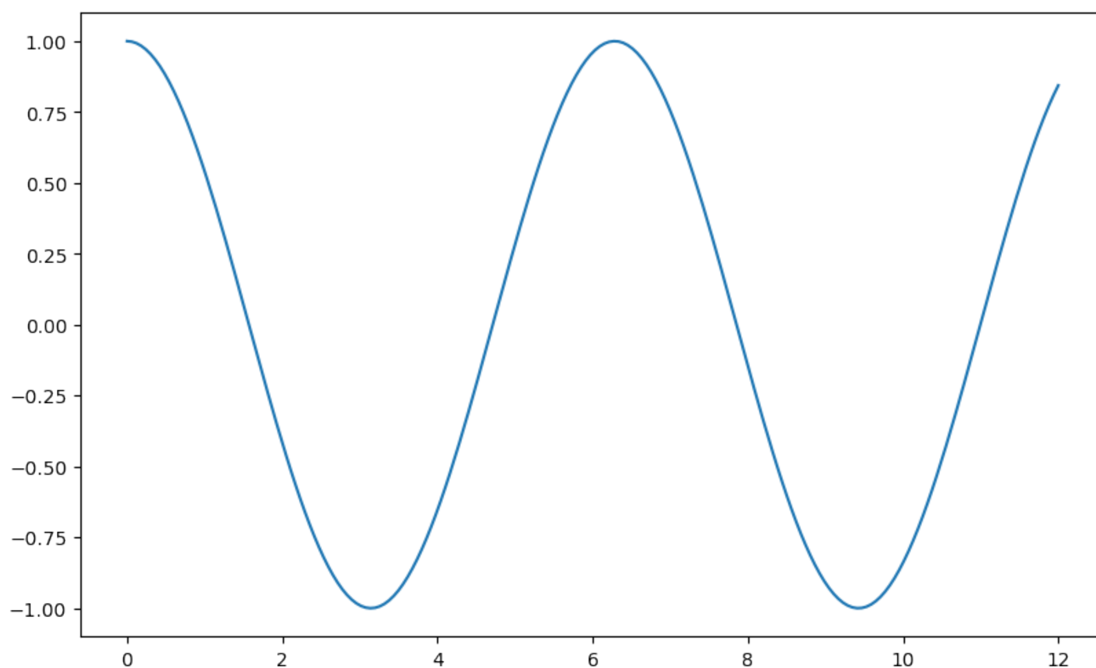
[83]:
```
b, a = a, b
print(a,b)
```

```
2 1
```

Cette propriété est très souvent utilisée pour "dépaqueter" les variables retournées par les fonctions :

[84]:
```
import matplotlib.pyplot as plt
import numpy as np

def tableaux_graphe_cos(x_inf, x_sup, N):
    X = np.linspace(x_inf, x_sup, N)
    Y = np.cos(X)
    return X, Y
```

[86]:
```
X, Y = tableaux_graphe_cos(0,12, 200)
ratio = 1.2 # ratio de taille entre fig et texte (légende et axes), par défaut 1
plt.figure(figsize=(8*ratio,5*ratio),dpi = 100)
plt.plot(X,Y)
plt.show()
```



- **Complément sur les boucles** : On peut interrompre l'exécution d'une boucle `for` ou `while` à l'aide de l'instruction `break` :

[87]:
```
for lettre in "python":
    print(lettre)
    if lettre == "o":
        break
```

```
p
y
t
h
o
```

On peut aussi placer dans une fonction plusieurs instructions `return`, souvent à la suite de test `if`, et possiblement dans une boucle :

[88]:
```
def plus_petit_diviseur(n):
    for div in range(2,n):
        if n % div == 0 :
            return("Plus petit diviseur de "+str(n)+" : "+str(div))
    return(str(n)+" premier")

print(plus_petit_diviseur(55))
print(plus_petit_diviseur(79))
```

```
Plus petit diviseur de 55 : 5
79 premier
```

Attention, il ne faut pas oublier que l'éxécution de la fonction s'arrête au premier `return` rencontré ! Ces dernières "astuces" sont à utiliser avec parcimonie.

Les boucle en `for elem in L` sont généralement utilisés avec `L` une liste. Mais on peut aussi parcourir un tuple, une chaîne de caractères, ou même un dictionnaire !

- **Terminaison** : Il est essentiel, pour l'écriture de boucle `while` ou de fonctions récursives, de tester la terminaison d'un programme (et ainsi ne pas rester bloqué dans une ("boucle infinie"). Dans le cas d'une boucle `for`, cela n'est que rarement problématique :

[89]:
```
L = [1,3,5]
for elem in L:
    print(elem**2)
```

```
1
9
25
```

Ici la liste `L` est une *invariant*, la boucle `for` va donc forcément se finir ! (Attention cependant aux boucles `for` qui parcourent des listes qu'on modifie…). Avec une boucle while :

[90]:
```
n = 5
fact = 1

while n > 0 :
    fact *= n
    n -= 1

print(fact)
```

```
120
```

Dans ce code, la suite des n est une suite strictement positive et strictement décroissante : elle va forçément se finir (à 1). On est donc assuré de la *terminaison* du code, et `n` est nommé *variant* de boucle.

- **Bonnes pratiques** : Il est évidemment utile, pour le correcteur **et pour vous** de commenter vos codes. Par ailleurs, il est nécessaire de passer suffisamment de temps à tester vos codes, avec des *jeux de tests* judicieux. enfin, mour l'écriture de fonctions, il est recommandé d'écrire une documentation et d'ajouter des assertions. Ainsi pour le code précédent, écrit sous forme d'une fonction :

[91]:
```
def fact(n):
    """ Retourne la valeur de n!, à condition que n soit un entier strictement␣
 ↪positif
    exemple :
    >>> fact(5)
    120
    """

    # On s'assure que n est un entier strictement positif :
    assert int(n) == n
    assert n > 0

    fact = 1

    while n > 0 : # réalisation du produit terme à terme
        fact *= n
        n -= 1

    return fact
```

[92]:
```
fact(0)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_90562/1238653140.py in <module>
----> 1 fact(0)

/tmp/ipykernel_90562/2609888048.py in fact(n)
```

```
      8     # On s'assure que n est un entier strictement positif :
      9     assert int(n) == n
---> 10     assert n > 0
     11 
     12     fact = 1

AssertionError: 
```

[93]:
```
fact(5)
```

[93]: 120

[94]:
```
fact(2.5)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_90562/2712522344.py in <module>
----> 1 fact(2.5)

/tmp/ipykernel_90562/2609888048.py in fact(n)
      7 
      8     # On s'assure que n est un entier strictement positif :
----> 9     assert int(n) == n
     10     assert n > 0
     11 

AssertionError: 
```

- **Coût temporel et Complexité**

Évaluer la complexité d'un algorithme revient à donner *approximativement* le nombre nécessaire d'opérations à effectuer. Il est difficile (pour ne pas dire impossible) et inutile d'en donner une valeur exacte, on se contente donc d'en donner un version *dominée* : un algorithme de variable n est dit de complexité en $O(n)$ si le nombre d'opérations nécessaires ne dépasse pas un multiple de n. Cette notion se rapproche de celle de l'équivalent vue en maths.

Remarques : De par la définition donnée, il n'est pas nécessaire d'ajouter des constantes multiplicatives : on ne notera pas $O(3n)$, mais bien $O(n)$. De même, comme pour les équivalents, seul le terme dominant à un intérêt : on ne notera pas $O(n^2 + n)$, mais bien $O(n^2)$

Le temps d'exécution des programmes est directement lié à la complexité : en effet, chaque opération est coûteuse en temps pour l'ordinateur (même si ceux-ci travail à l'échelle de la nanoseconde !). Dans le cas de calculs sur une liste de taille n : tout programme, même peu optimisé, peut effectuer des calculs simples de façon rapide une liste de 10 éléments. Mais si on considère des logiciels devant travailler sur les listes de millions d'éléments ? Il est alors cruciale d'optimiser son code, afin d'obtenir la complexité la plus faible possible.

On considérera que chaque opération simple (affectation, opération mathématique) à un coût de 1.

Type de complexité (les plus fréquents) :
- en $\mathcal{O}(1)$ ("temps constant") : temps nécessaire pour effectuer une opération "simple" (opération mathématique, affectation, …)

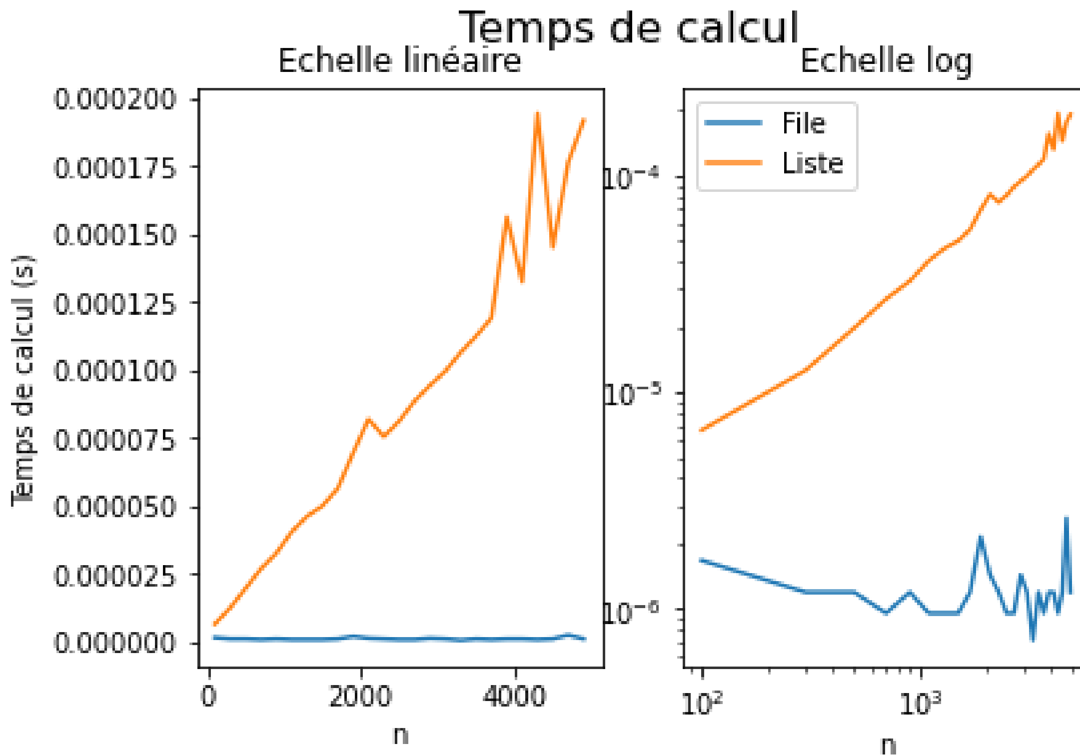
- en $\mathcal{O}(log(n))$ ("temps logarithmique") : le temps de calcul évolue comme $log(n)$, avec n le nombre d'éléments à traiter (souvent la taille de la liste)
- en $\mathcal{O}(n)$ ("temps linéaire") : le temps de calcul évolue comme n, avec n le nombre d'éléments à traiter. On est donc dans le cas ou le temps de calcul est proportionnel à la taille des données soumises.
- en $\mathcal{O}(n^2)$ ("temps quadratique") : le temps de calcul évolue comme n^2, avec n le nombre d'éléments à traiter. On est donc dans le cas ou le temps de calcul est proportionnel au carré de la taille des données soumises.
- …

Attention certaines opérations "simples à écrire" ne sont pas forcement "simples en temps". Par exemple, considérons une liste de longueur n, à laquelle on veut ajouter à gauche (ou au début) 10 nouveaux éléments, et comparons la même opération avec une file :

[95]:
```
import matplotlib.pyplot as plt
from collections import deque
from time import time

N, Tfile, Tliste = [], [], []
for n in range(100,5000,200):
    N.append(n)
    L = [0]*n
    F = deque()
    for i in range(n):
        F.append(0)
    time1=time()
    for i in range(10):
        F.appendleft(0)
    time2=time()
    for i in range(10):
        L = [0] + L
    time3=time()
    Tfile.append(time2-time1)
    Tliste.append(time3-time2)

ratio = 1.2 # ratio de taille entre fig et texte (légende et axes), par défaut 1
plt.figure(figsize=(8*ratio,5*ratio),dpi = 150)
fig, axs = plt.subplots(1, 2)
axs[0].plot(N,Tfile,label='File')
axs[0].plot(N,Tliste,label='Liste')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
```

```
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Tfile,label='File')
axs[1].loglog(N,Tliste,label='Liste')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```

```
<Figure size 1440x900 with 0 Axes>
```



On remarque donc bien que l'ajout d'élément au début de la file semble être à coût constant, alors que pour une liste, on semble plutôt être en temps linéaire $\mathcal{O}(n)$. D'où l'intérêt des **deque** pour créer des files, plutôt que des listes.

Faire de calculs sur une liste avec un algorithme de complexité $\mathcal{O}(n)$ va donc nous (enfin à l'ordinateur…) prendre un temps proportionnel au nombre d'éléments contenus dans la liste. Alors que faire ces mêmes calculs sur la même liste avec un algorithme de complexité $\mathcal{O}(n^2)$ va nous prendre un temps proportionnel au nombre d'éléments contenus dans la liste **au carré**.

Prenons par exemple un algorithme capable de trier une liste de 100 éléments en 1ms : s'il est de compléxité en $\mathcal{O}(n^2)$, il mettra environ 105s pour une liste d'un million d'éléments… Alors qu'avec une compléxité en $\mathcal{O}(n)$, il mettra seulement environ 10s. Bien plus agréable !

Remarque : un algorithme de compléxité en $O(n)$ n'est pas forcément plus rapide qu'un autre en $O(n^2)$. Seulement, quand on augmente n, le temps nécessaire pour chacun des programmes ne va pas évoluer de la même manière avec n, en faveur de celui en $O(n)$!

Essayons de déterminer la complexité de deux algorithmes "simples" : la fonction `carres(n)` est une fonction calculant le carré des entiers jusqu'à n, alors que la fonction `somme(n)` calcule la somme (a+b) avec pour a et pour b toutes les combinaisons d'entiers jusqu'à n.

[96]:
```
def carres(n):
    for i in range(1,n+1):
        i*i # on effectue seulement le calcul, sans même stocker la valeur,␣
 ↪inutile

def somme(n):
    for a in range(1,n+1):
        for b in range(1,n+1):
            a + b
```

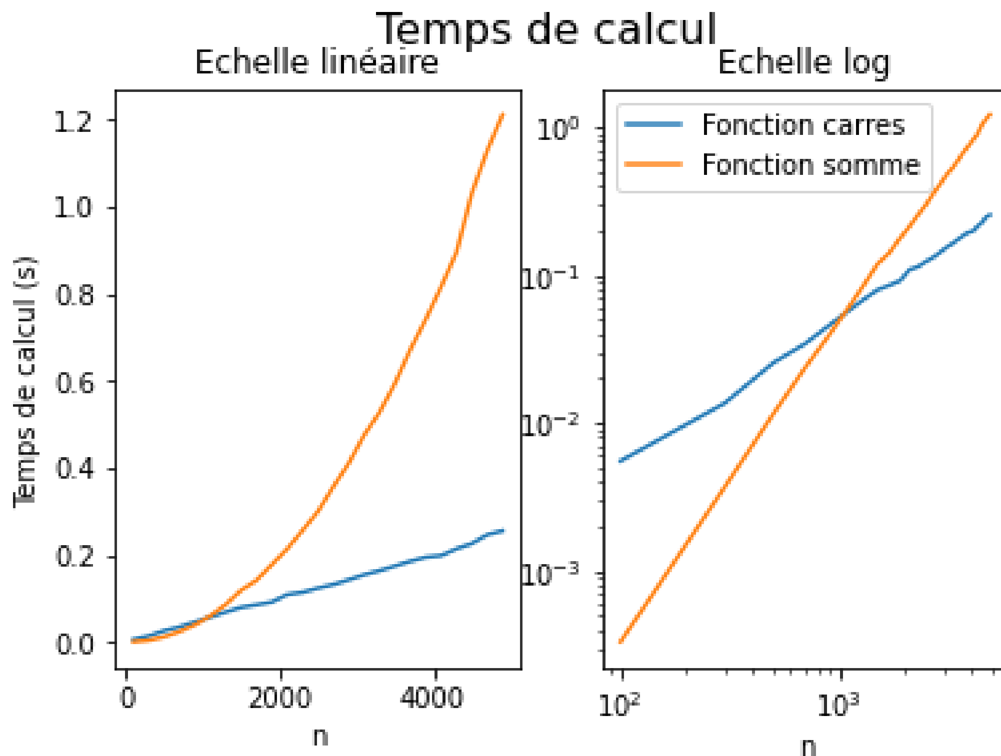
La complexité de chacune de ces fonctions est en :

- $\mathcal{O}(n)$ pour `carres(n)`
- $\mathcal{O}(n^2)$ pour `somme(n)`

On peut le vérifier en calculant le temps d'exécution pour différentes valeurs de n :

[97]:
```
Tcarres,Tsomme,N=[],[],[]
for n in range(100,5000,200):
    N.append(n)
    time1=time()
    for i in range(1000): # La fonction carres est tellement rapide
        carres(n)
    time2=time()
    somme(n)
    time3=time()
    Tcarres.append(time2-time1)
    Tsomme.append(time3-time2)

fig, axs = plt.subplots(1, 2)
axs[0].plot(N,Tcarres,label='Fonction carres')
axs[0].plot(N,Tsomme,label='Fonction carres_somme')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Tcarres,label='Fonction carres')
axs[1].loglog(N,Tsomme,label='Fonction somme')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
```

```
plt.legend()
plt.show()
```



On peut vérifier sur le deuxième graphique que la pente est proche de 1 pour `carres(n)`, et proche de 2 pour `somme(n)` : ceci n'est pas un hasard, la pente en échelle log est caractéristique de l'exposant de la complexité de l'algorithme.

Question subsidiaire : montrer que la complexité de l'algorithme dichotomique précédent (insertion au bon indice d'un nombre dans une liste triée) est en $\mathcal{O}(log(n))$.

Pour un algorithme de dichotomie agissant sur une liste de longueur n, il va être nécessaire de diviser en 2 la liste jusqu'à obtenir une sous-liste contenant seulement 1 ou 2 éléments. Pour simplifier, considéraons une sous-liste finale contenant 1 seul élément. A chacune de ces "divisions", la liste perd environ la moitié de ses éléments, sa taille est approximativement divisée par 2. En notant N le nombre de "divisions" de la liste initiale, la sous-liste finale sera alors de taille :

$$\frac{n}{2^N} = 1$$

On a donc :

$$log(n) = Nlog(2)$$

Soit :

$$N = \frac{log(n)}{log(2)} = \mathcal{O}(log(n))$$