

TP 0

INSTALLATION

0.1 INSTALLATION

Nous allons travailler avec la version 3 de Python en utilisant IDLE. Vous pouvez installer un environnement de travail satisfaisant de plusieurs manières.

§1 Installation sous GNU Linux ou BSD

- Installer Python en version 3.x.x
- Vérifier qu'IDLE est bien disponible. En ligne de commande, il faut lancer `idle` ou `idle3` selon votre distribution.
- Installer les modules `matplotlib`, `numpy`, `pandas`, `scipy` en version `python3`.

§2 Installation avec WinPython sous Windows

1. Téléchargez et installez la dernière version de WinPython sur

<http://winpython.github.io>

Si l'espace disque n'est pas un problème, prenez la version complète (790 Mo pour `Winpython64-3.9.5.0.exe`).

2. Lancer l'installation en cliquant sur le fichier téléchargé. Vous pouvez choisir d'installer WinPython où vous le souhaitez, c'est une distribution portable.

3. Facultatif : pour intégrer la distribution à l'environnement Windows, ouvrir

WinPython Control Panel

dans le dossier d'installation puis dans les menus

>Advanced puis >Register Distribution.

4. Démarrer IDLEX ou IDLE en cliquant sur l'icône appropriée dans le répertoire d'installation.

§3 Installation avec Anaconda sous MacOS et Windows

1. Jetez un coup d'œil au site de la distribution Anaconda:

<https://www.continuum.io>

et télécharger Anaconda.

2. Lancer l'installation.

3. Ouvrir une fenêtre Anaconda Prompt puis taper

```
conda install -c auto idlex
```

4. Toujours dans la fenêtre, taper `idlex`, ce qui devrait ouvrir une nouvelle fenêtre.

0.2 TESTER VOTRE INSTALLATION

Pour finir, vérifier que les modules `matplotlib`, `numpy`, `scipy` sont bien installés. Les trois lignes suivantes doivent rester muette.

Python 3 Shell

```
>>> import matplotlib
>>> import numpy
>>> import scipy
```

Si ce n'est pas le cas, vous avez vraisemblablement ce genre de message

Python 3 Shell

```
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'numpy'
>>>
```

Dans ce cas, demandez de l'aide.

TP

101

INTRODUCTIONS

101.1 IDLE COMME ENVIRONNEMENT DE DÉVELOPPEMENT INTÉGRÉ

§1 La console interactive ou Shell

Dans cette section, nous aurons besoin d'utiliser IDLE (Integrated DeveLopment Environment). Si tout va bien, on voit apparaître le message de copyright et le **prompt** `>>>`, soit à peu de chose près :

Python 3 Shell

```
Python 3.9.5 (default, May 24 2021, 12:50:35)
[GCC 11.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

§2 Interpréteur Python en ligne de commande

La fenêtre se nomme Python 3.9.5 Shell (ou autre version 3.x.x). Nous sommes dans un **shell Python**, autrement dit, un **interpréteur en ligne de commande**. On peut le reconnaître au **prompt** : ce sont les trois chevrons

Python 3 Shell

```
>>>
```

Le *prompt* signale l'emplacement où l'on peut écrire des instructions à faire exécuter par Python. Pour aujourd'hui, considérez que vous êtes face à une (grosse) calculatrice.

Exercice 1. Exécuter les commandes suivantes, observer, modifier, comprendre.

- | | | |
|-----------------------------|-------------------------------|---------------------------------|
| 1. <code>2 + 5</code> | 11. <code>float(3)</code> | 21. <code>2 +</code> |
| 2. <code>4 * 3</code> | 12. <code>abs(-34.5)</code> | 22. <code>2 < 4</code> |
| 3. <code>8 - 9</code> | 13. <code>int(3.9)</code> | 23. <code>2 > 4</code> |
| 4. <code>2 ** 8</code> | 14. <code>round(3.9)</code> | 24. <code>type(2 > 4)</code> |
| 5. <code>2 ^ 8</code> | 15. <code>2.4 + 3.7</code> | 25. <code>"4 * 3"</code> |
| 6. <code>35 / 3</code> | 16. <code>4 * 5 + 3</code> | 26. <code>type("4 * 3")</code> |
| 7. <code>35 // 3</code> | 17. <code>4 * (5 + 3)</code> | 27. <code>2x</code> |
| 8. <code>35 % 3</code> | 18. <code>4(5 + 3)</code> | 28. <code>2 * x</code> |
| 9. <code>type(35)</code> | 19. <code>2 + 3 + 12</code> | |
| 10. <code>type(35.0)</code> | 20. <code>2 + 3 # + 12</code> | |



À retenir : Le symbole `^` ne désigne pas la puissance. En Python, on utilise `**`.

§3 Définition d'une fonction numérique

Une fonction permet d'isoler une instruction qui revient plusieurs fois dans un programme.

Une fonction est définie par un nom, par ses arguments qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel et éventuellement une valeur de retour communiquée au programme par la fonction en fin d'exécution.

La fonction $g : x \mapsto x^4/4 - x^3/3 - 3x^2$ peut être définie ainsi dans la console Python :

Python 3 Shell

```
>>> def g(x):
...     return x ** 4 / 4 - x ** 3 / 3 - 3 * x ** 2
...
>>>
```



Il faut bien noter l'indentation avant le **return**, celle-ci fait partie intégrante de la syntaxe Python. Il est conseillé d'utiliser quatre espaces pour l'indentation.

Ensuite on peut évaluer cette fonction en quelques points.

Python 3 Shell

```
>>> g(5)
39.583333333333334
>>> g(8)
661.33333333333334
>>> g(0) + g(2)
-10.666666666666666
>>>
```

Tout comme le mathématicien, Python est allergique aux noms d'objets que l'on a pas préalablement introduit:

Python 3 Shell

```
>>> g(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

D'ailleurs, on peut utiliser n'importe quel nom de variable pour définir la fonction g . Ici, la variable t est (presque) muette.

Python 3 Shell

```
>>> def g(t):
...     return t ** 4 / 4 - t ** 3 / 3 - 3 * t ** 2
...
>>>
```

Exercice 2.

1. Définir la fonction $h : x \mapsto 2x^2 - 3/x$.
2. Tester votre fonction (c'est une étape indispensable) avec les valeurs 2, 3, -8.5 et 0. Vous devez obtenir

Python 3 Shell

```
>>> h(2)
6.5
>>> h(3)
17.0
>>> h(-8.5)
144.85294117647058
>>> h(0)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in h
ZeroDivisionError: division by zero
>>> h(1) + h(2) + h(3)
22.5
>>>
```



Raccourcis claviers. Alt+P et Alt+N permettent de naviguer entre les lignes déjà tapées.

§4 Import d'un module

Mis à part la fonction **abs** (valeur absolue) et **round** (arrondi), Python dispose de peu de fonctions numériques usuelles.

Heureusement, nous pouvons importer de nouvelles fonctions grâce à la directive **import**. On a alors à disposition une collection de constantes et fonctions usuelles. Par exemple, le sinus, le cosinus et la constante π :

Python 3 Shell

```
>>> import math
>>> math.sin(0)
0.0
>>> math.pi
3.141592653589793
>>> math.sin(math.pi / 2)
1.0
>>> math.pi + math.cos(1)
3.681894959457933
>>>
```

Pour obtenir la liste des fonctions et constantes importées :

Python 3 Shell

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

Pour obtenir l'aide sur une fonction particulière :

Python 3 Shell

```
>> help(math.sin)
```

Help on built-in function sin in module math:

```
sin(...)  
    sin(x)
```

Return the sine of x (measured in radians).
(END)

```
>>
```

Plus généralement, `help(math)` renvoie l'aide pour toutes les fonctions du module `math`.

Exercice 3.

1. Définir la fonction $k : x \mapsto \sqrt{1 + \sin(x)}$.
2. Tester votre fonction avec les valeurs 2, 3, -8.5 et π . Vous devez obtenir les résultats suivants :

Python 3 Shell

```
>>> k(2)  
1.3817732906760363  
>>> k(3)  
1.0682321882717574  
>>> k(-8.5)  
0.44890186831479073  
>>> k(math.pi)  
1.0  
>>>
```

101.2 LES FICHIERS *.py

Le système d'**import** permet de faire du code réutilisable et non de la maquette jetable.



Dans IDLE, nous allons souvent alterner entre la console interactive et l'éditeur de script. Il est donc important de savoir dans quel type de fenêtre on se trouve.



La console se différencie par les *prompt* `>>>` et par son titre `Python 3.x.x Shell`. Le texte est monochrome.

§1 Un premier fichier source

Exercice 4.

1. Créer un nouveau fichier (Menu File/New File ou Ctrl+N). Une nouvelle fenêtre apparaît, celle-ci est un éditeur de texte, ce n'est pas une console interactive. Sauvegarder le fichier *immédiatement* (Menu File/Save ou Ctrl+S) dans votre répertoire tp101 sous le nom `mesfonctions.py`.

Commencer par entrer les lignes suivantes dans l'éditeur de script (fichier `mesfonctions.py`).

```
1 #!/usr/bin/env python3
2 # encoding: utf-8
3 # PTSI2 <votre nom>
4
5 def g(x):
6     return x ** 4 / 4 - x ** 3 / 3 - 3 * x ** 2
```

Les deux premières lignes n'ont pas beaucoup d'importance pour le moment. Tout ce qui est placé après un # est un commentaire.

On retrouve ensuite la définition de la fonction $g : x \mapsto \frac{x^4}{4} - \frac{x^3}{3} - 3x^2$.

2. Exécuter votre fichier (F5 ou par le menu Run/Run Module). Dans l'interpréteur en ligne de commande apparaît

Python 3 Shell

```
>>> ===== RESTART =====
>>>
>>>
```

Ce qui se trouve dans le fichier `mesfonctions.py` a été exécuté comme s'il avait été tapé dans la console interactive. On peut d'ailleurs tester la fonction `g` avec quelques valeurs, par exemple `g(5)` et `g(8)`.

Python 3 Shell

```
>>> g(5)
39.58333333333334
>>> g(8)
661.3333333333334
>>>
```

3. Quitter IDLE (fermer toutes les fenêtres). Exécuter de nouveau IDLE, et charger votre fichier `mesfonctions.py` (Menu File/Open... ou Ctrl+O). Recommencer la question précédente. Que constate-t-on ?



Dans l'éditeur de script, *aucun prompt* n'est présent. Son titre est le nom du fichier en cours. Si vous avez bien pris la peine de nommer votre fichier avec une extension `.py`, le code bénéficie de la coloration syntaxique.

Exercice 5.

1. Dans le fichier `mesfonctions.py`, définir à nouveau la fonction

$$h : x \mapsto 2x^2 - 3/x$$

Ne pas oublier pas de la tester !

2. Toujours dans le fichier `mesfonctions.py`, définir la fonction

$$k : x \mapsto \sqrt{1 + \sin(x)}$$

Ne pas oublier pas de la tester !

§2 Fonctions de plusieurs variables

Il n'est pas plus difficile de définir une fonction de deux variables, par exemple, la fonction

$$(x, y) \mapsto x^y + y^x$$

peut se définir par

```
1 def puissances(x, y):
2     return x ** y + y ** x
```

Que l'on peut tester après exécution

Python 3 Shell

```
>>> ===== RESTART =====
>>>
>>> puissances(2, 3)
17
>>> puissances(0, 4)
1
>>>
```

Exercice 6.

1. À la suite de votre fichier `mesfonctions.py`, définir une fonction `hypothénuse`, avec deux paramètres a et b , calculant $\sqrt{a^2 + b^2}$.

2. Tester ! Par exemple...

Python 3 Shell

```
>>> ===== RESTART =====
>>>
>>> hypothénuse(3, 4)
5.0
>>> hypothénuse(11, 23)
25.495097567963924
```

```
>>> hypotenuse(7.8, 11.9)
14.228492541376266
>>>
```

101.3 INTRODUCTION AUX LISTES

§1 Listes

Tout comme les listes de courses dans le monde réel, les listes en Python sont très utiles. C'est la structure de données la plus utilisée, et elle peut l'être d'une multitude de façons pour modéliser et résoudre tout un tas de problèmes.

Dans la console interactive de IDLE...

Python 3 Shell

```
>>> s = [-3, -2, -1, 0, 1, 2, 3]
```

La partie droite est une **liste** d'éléments. Comme nous pouvons le constater, les listes sont dénotées par des crochets, et les valeurs d'une liste sont séparées par des virgules.

L'instruction `s = ...` permet de créer un **identifiant** pour un objet, ici `s` est un identifiant de la liste `[-3, -2, -1, 0, 1, 2, 3]`. Ceci permet de réutiliser la liste sans avoir besoin de la retaper !

C'est la notion de *variable* vue au lycée... En tout cas cela en est une approximation suffisante pour l'instant. Dans la console interactive, nous pouvons afficher la liste `s` :

Python 3 Shell

```
>>> s
[-3, -2, -1, 0, 1, 2, 3]
```

Et si nous souhaitons accéder à une valeur particulière de la liste

Python 3 Shell

```
>>> s[3]
0
>>> s[2]
-1
>>> s[5]
2
>>> s[0]
-3
```

Remarquons que les indices démarrent à 0. La **longueur** d'une liste (son nombre d'éléments) s'obtient grâce à la fonction `len`

Python 3 Shell

```
>>> len(s)
7
```

§2 Fonctions retournant une liste

Exercice 7. Dans votre fichier `mesfonctions.py`, écrire une fonction `sommeproduit`, avec deux paramètres `x` et `y`, et qui renvoie la *liste* contenant la somme $x+y$ et le produit $x*y$ des deux paramètres.

Indication : N'oubliez pas de tester votre fonction, par exemple

Python 3 Shell

```
>>> sommeproduit(7,4)
[11, 28]
>>> sommeproduit(1,9)
[10, 9]
>>> sommeproduit(0,123)
[123, 0]
```

§3 Fonctions ayant une liste pour argument

Exercice 8. Écrire une fonction `muladd`, avec un seul paramètre. Le paramètre de `muladd` doit être *une liste de trois éléments*. Cette fonction retourne le produit de ses deux premiers éléments additionné au troisième élément.

Python 3 Shell

```
>>> muladd([3, 7, 11])
32
>>> muladd([2, 5, -9])
1
>>> a = [1, 3, 2]
>>> muladd(a)
5
```

On a bien $3 \times 7 + 11 = 32$, $2 \times 5 - 9 = 1$ et $1 \times 3 + 2 = 5$.

Exercice 9. Dans votre fichier `mesfonctions.py`, écrire une fonction `dernier`, avec une liste `s` en paramètre, et qui renvoie le dernier élément de la liste.

101.4 LA STRUCTURES DE CHOIX

§1 Expressions booléennes

Une **expression booléenne** est une expression dont la valeur est soit vrai (**True**), soit faux (**False**). Ces expressions sont de type **bool**.

Python 3 Shell

```
>>> 12 == 12
True
>>> 12 != 12
False
>>> type(12 == 12)
<class 'bool'>
>>>
```

Enfin, voici les opérateurs de tests sur les valeurs numériques ainsi que les opérateurs booléens :

Égalité	==		
Différent	!=		
Inférieur strict	<	et	and
Inférieur ou égal	<=	ou	or
Supérieur strict	>	non	not
Supérieur ou égal	>=		

§2 Présentation du bloc d'instructions **if (elif)* [else]**

L'instruction **if** permet de vérifier des conditions avant d'exécuter un bloc d'instructions. La documentation python nous fournit la syntaxe de manière assez obscure : https://docs.python.org/3/reference/compound_stmts.html#the-if-statement

```
"if" expression ":" suite
( "elif" expression ":" suite )*
["else" ":" suite]
```

Voici plutôt un exemple simple : la valeur absolue

```
1 def vabs(x):
2     if x > 0:
3         y = x
4     elif x == 0:
5         y = 0
6     else:
7         y = -x
8     return y
```

- L'instruction **if** $x > 0$: exécute le bloc d'instruction $y = x$ si la condition $x > 0$ est vérifiée.

- Dans le cas où le bloc précédent n'est pas exécuté, l'instruction `elif x == 0` pour «else if» (sinon si en français) exécute le bloc `y = 0` si `x` est nul.
- Enfin, si aucune des conditions précédentes n'est vérifiées le bloc `y = -x` est exécuté.

Remarquez bien la présence du «:» et de l'indentation. Python est un langage visuel, les blocs d'instructions sont repérés par une indentation (quatre espaces). Normalement, inutile d'effectuer l'indentation à la main. Tout éditeur de code digne de ce nom (et même IDLE) s'en charge automatiquement lorsque qu'il croise un «:».

Dans la syntaxe générale, il peut y avoir plusieurs blocs `elif` successifs ; il peut également n'y en avoir aucun :

```
1 def vabs(x):
2     if x >= 0:
3         y = x
4     else:
5         y = -x
6     return y
```

Le bloc `else` est également facultatif, mais il y en a *au plus* un.

```
1 def vabs(x):
2     y = x
3     if x < 0:
4         y = -x
5     return y
```

§3 Exercices

Exercice 10.

1. Tester dans la console les instructions `15 % 3`, `15 // 3`, `15 // 4`, `15 % 4`. Expliquer le fonctionnement des opérateurs `//` et `%`.
2. Écrire une fonction `divise` qui prend comme arguments deux entiers relatifs a et $b \neq 0$ et qui renvoie le booléen `True` si b divise a , `False` sinon.
3. En déduire une implémentation d'une fonction `est_pair` qui teste la parité d'un entier relatif a .

Exercice 11.

1. Implémenter la fonction `second_degre` spécifiée ainsi:
 - La fonction `second_degre` a trois paramètres a , b , c .
 - Les paramètres a , b , c sont de type numérique (flottants) et $a \neq 0$
 - La fonction `second_degre` retourne une liste contenant les racines réelles du polynôme

$$aX^2 + bX + c$$

Vérifier ensuite que le programme fonctionne, en considérant les trinômes $X^2 - 1$, $X^2 - 2x + 1$ et $X^2 + X + 1$.

2. On se place maintenant dans le cas général (c'est-à-dire que a peut-être nul). Implémenter la fonction `liste_racines` spécifiée ainsi:

- La fonction `liste_racines` a trois paramètres a , b , c .
- Les paramètres a , b , c sont de type numérique (flottants) et non tous nuls.
- La fonction `liste_racines` retourne une liste contenant les racines réelles du polynôme

$$aX^2 + bX + c$$

Tester votre fonction avec les polynômes $X + 1$, 1 .

3. Que retourne votre fonction `liste_racines` pour le polynôme 0 ?

101.5 TESTER ET ÉCRIRE SES FONCTIONS

§1 Exemple de test automatisé d'une fonction

L'instruction `assert` permet de tester si une assertion (une expression booléenne) est vraie.

Python 3 Shell

```
>>> assert 12 == 12
>>> assert 12 != 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 1 < 3
>>>
```

Si celle-ci est fausse, apparaît une erreur.

En Python, la fonction `max` permet de retourner le maximum de deux nombres.

Python 3 Shell

```
>>> max(2, 8)
8
>>> max(8, 2)
8
>>> max(1, 1)
1
>>>
```

Dans cet exemple, nous allons implémenter notre propre fonction¹ `maximum` avec deux paramètres `a` et `b` et qui renvoie le maximum de ces deux valeurs. Mais nous allons écrire simultanément² la fonction et un **test fonctionnel**.

À la suite de votre fichier `mesfonctions.py`, ajouter une fonction `test_maximum` ainsi qu'une fonction `maximum` définie comme ci-dessous:

```
def test_maximum():
    assert maximum(1, 2) == 2
    assert maximum(-99999999, 1) == 1
    assert maximum(2, 1) == 2
    assert maximum(281, -5) == 281
    assert maximum(1, 1) == 1

def maximum(a, b):
    return b
```

Comme vous pouvez le constater, la fonction `maximum` retourne le second paramètre : pas terrible comme fonction `maximum`.

Exécutons le fichier source (F5), puis dans la console interactive, testons:

Python 3 Shell

```
>>> maximum(8, 12)
12
>>> maximum(203, 25)
25
```

Cela confirme que notre fonction n'est pas correcte. Mais au-dessus de la fonction `maximum`, se trouve une fonction `test_maximum`. Cette fonction permet d'effectuer un test automatisé. Toujours dans la console interactive, on obtient

Python 3 Shell

```
>>> test_maximum()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test_fonctions.py", line 8, in test_maximum
    assert maximum(2, 1) == 2
AssertionError
>>>
```

Elle signale que l'assertion `maximum(2, 1) == 2` est fausse, ce qui est normal puisque `maximum(2, 1)` retourne la valeur 1. De plus, elle signale que l'erreur a eu lieu lors de l'exécution de la ligne 8.

Exercice 12.

¹Ce qu'il ne faut bien sûr *jamais* faire, sauf si on vous le demande explicitement.

²On pourrait faire mieux! En effet, il est préférable d'écrire le test en premier, et seulement ensuite la fonction. Nous en reparlerons en cours.

1. Modifier la fonction `maximum` afin qu'elle retourne le maximum de ses deux arguments. L'utilisation de la fonction `max` de Python est bien évidemment interdite!
2. N'oublions pas qu'une fonction écrite se doit d'être testée!
3. Au fait, que donne l'appel suivant ?

Python 3 Shell

```
>>> maximum("Alice", "Bob")
```

Commentez rapidement.



Cela peut paraître évident, mais précisons qu'un test fonctionnel doit être écrit indépendamment de la fonction. Et donc, surtout pas en utilisant les résultats de la fonction à tester!

§2 Exercices

Exercice 13. Écrire une fonction `maximum3` et un test associé `test_maximum3` tels que:

- La fonction `maximum3` a trois paramètres `a`, `b`, `c`.
- La fonction `maximum3` retourne le maximum des trois valeurs données en paramètres.

Exercice 14. Écrire une fonction `maximum4` et un test associé `test_maximum4` tels que:

- La fonction `maximum4` a quatre paramètres `a`, `b`, `c`, `d`.
- La fonction `maximum4` retourne le maximum des quatre valeurs données en paramètres.

§3 Exercices avec des listes

Rappel

On rappelle que l'on peut accéder et modifier les éléments d'un tableau `A` en utilisant la notation `A[index]`.

Python 3 Shell

```
>>> A = [8, 11, 4]
>>> A[0]
8
>>> A[1]
11
>>> A[2]
4
>>> A[0] = 212
>>> A[2] = 314
>>> A
[212, 11, 314]
>>>
```

Exercice 15. Écrire une fonction `tri` prenant une liste de 3 nombres triplet en paramètre et retournant une liste contenant les même trois nombres dans l'ordre croissant.

Python 3 Shell

```
>>> tri([4, 12, 8])
[4, 8, 12]
>>> tri([12, 4, 8])
[4, 8, 12]
>>> tri([8, 12, 4])
[4, 8, 12]
```

On écrira bien sûr le test fonctionnel `test_tri` associé.

101.6 L'ORDRE ET LA PAGAILLE

Il y a des tâches pour lesquelles la marche que doit suivre l'ordinateur n'est pas du tout évidente. Nous consacrerons une bonne partie de l'année aux façons de dégager ces marches à suivre, de telle sorte que nous puissions en garantir la validité. Mais, allant du simple au compliqué, nous allons considérer d'abord les cas où la marche à suivre est évidente, quand on la dit en français, mais ne se transforme pas facilement en un programme.

Il y a en effet des cas où la difficulté de la programmation vient de ce que le travail à faire faire par l'ordinateur est régi par un grand nombre de règles plus ou moins bien organisées. Ce que nous considérons ici relève plus des tâches que des problèmes, la difficulté venant du grand nombre de cas pouvant se produire, et qui ne sont pas nécessairement répertoriés de façon claire. La difficulté sera donc *de mettre de l'ordre dans la pagaille*.

§1 Dire si une date est valide

Nous partirons d'un exemple concret. On donne une date par trois entiers : `jour`, le quantième jour, `mois` le numéro du mois, `an` le millésime. Il faut dire si c'est une date valide : il n'y a pas de 31 avril, ni de 29 février 1900...

La première difficulté est de réunir les règles disant si une date est valide ou non. On sait que le numéro de mois doit être compris entre 1 et 12. Le numéro de jour doit être au moins égal à 1, et au plus égal à un nombre qui dépend du mois (en première analyse): appelons ce nombre `long_mois` (longueur du mois). Nous avons fait apparaître une variable intermédiaire, ce qui va éclater le problème en posant le sous-problème: calculer la longueur du mois.

Si on pratique la **programmation descendante**, on remettra ceci à plus tard, et on essaiera de voir si toutes les autres règles ont été réunies. On constate que tout a été dit concernant le quantième et le numéro de mois. Quelles règles concernent le millésime? Seul celui qui sait qu'il y a eu des réformes de calendrier peut soupçonner qu'une telle réforme peut modifier les conditions de validité, et poser qu'il est prudent de s'en tenir au calendrier actuel. Une encyclopédie lui apprendra qu'il a été établi par le pape Grégoire XIII en 1582. Il supposera donc `an > 1582`. Mais la réforme n'a pas été appliquée en même temps dans tous les pays (si vous êtes curieux, vous en apprendrez plus dans «Le pendule de Foucault» de Umberto Eco...). Par prudence, prenons donc `an > 1600`.

Il n'est pas évident que l'étude descendante s'impose ici. Si l'on avait attaqué tout de suite le problème de la longueur du mois, nous aurions constaté qu'il fait apparaître la notion d'année bissextile. La consultation d'un encyclopédie pour connaître les règles qui les commandent aurait très certainement mentionné la réforme grégorienne... Quel que soit le chemin suivi, le résultat est le même: la longueur du mois est connue pour les mois autres que février ; pour celui-ci, elle est de 28 jours pour les années non bissextiles, de 29 pour les bissextiles. Notons au passage qu'il est maintenant clair que la longueur du mois de février dépend de l'année, ce qui modifie ce que l'on avait annoncé plus haut «en première analyse»: la longueur du mois dépend du mois et de l'année... Sont bissextiles les années dont le millésime est multiple de 4. Mais les années séculaires (celles dont le millésime se terminent par 00) ne sont pas bissextiles, sauf si le millésime est multiple de 400.



Exercice 16.

1. Écrire une fonction `date_valide` ayant pour arguments trois entiers `jour`, `mois`, `an`.

- Cette fonction retourne `True` (le booléen) lorsque la date est une date valide du calendrier grégorien.
- Cette fonction retourne `False` sinon.

Indication : On rappelle que `a % b` renvoie le reste de la division euclidienne de `a` par `b`. Donc `b` divise `a` (c'est-à-dire `a` est multiple de `b`) si, et seulement si `a % b == 0`.

2. Où est votre test `test_date_valide` associé ?

C'est un bel exemple d'*embrouillamini*: beaucoup de règles, sans aucune organisation. Il y a bien un découpage en sous-tâches : longueur du mois, validité du jour, validité du mois, validité de l'année. Et que faire pour les années bissextiles? Elles n'apparaissent pas dans ce découpage, et pourtant nous savons qu'elles jouent un rôle. Dans quel ordre prendre ces sous-tâches? On ne peut déterminer la longueur du mois si le mois ou l'année ne sont pas valides...

Il est conseillé d'appliquer systématiquement la règle suivante:

Traiter d'abord les cas les plus simples.

Elle vient tout droit du «Discours de la méthode» de Descartes: «conduire par ordre mes pensées, en commençant par es objets les plus simples et les plus aisés à connaître...»

TP

102

LES STRUCTURES DE RÉPÉTITION CONDITIONNELLE



À rendre avant le

- Les exercices 2, 3, 4, 6, 5, 7, sont à rendre dans un fichier nommé `partie2.py`
- L'exercice 8, est à rendre dans un fichier nommé `partie3.py`



- Les questions ☕ sont plus difficiles. Vous pouvez donc les passer dans un premier temps.
- Chaque fichier devra comporter un commentaire dans l'en-tête avec vos nom, prénom, classe, groupe et numéro du TP.
- Une exécution du fichier avec «F5» dans IDLE ne devra pas provoquer d'erreur.
- Les noms des fichiers et des fonctions demandées doivent être respectés.
- La note ne valide pas seulement le résultat de votre programme, mais également son style
 - choix approprié des noms de variables,
 - présence de documentation et de tests pour les fonctions importantes,
 - commentaires pertinents.
- Vous pouvez ajouter des résultats obtenus dans la console interactive (Shell) à la suite de vos fonctions, sur plusieurs lignes, entre triple quote (`"""....."""`).

Le non-respect d'un des points précédents peut retrancher jusqu'à 50 points sur la note finale.



La section 102.1 est à étudier *avant* le TP.

Nous allons maintenant étudier une des structures les plus puissantes de la programmation, à savoir la **répétition TantQue (while)**: elle permet d'exécuter à plusieurs reprises une suite d'instructions. Dans la plupart des langages, ces répétitions se classent en deux catégories :

- Les répétitions conditionnelles : la poursuite de la répétition des instructions concernées dépend d'une certaine condition qui peut être examinée.
- Les répétitions inconditionnelles : les instructions concernées sont répétées un nombre donné de fois.

102.1 EXEMPLE INTRODUCTIF : LA DIVISION EUCLIDIENNE



Exemples téléchargeables.

<http://www.prepa-baggio.fr/info/tp102/division.py>

§1 Problème

Pour $a, b \in \mathbb{N}$ avec $b \neq 0$, on veut calculer l'unique couple d'entiers (q, r) tels que

$$a = bq + r \text{ et } 0 \leq r < b.$$

Les entiers q et r sont respectivement le **quotient** et le **reste** de la division euclidienne de a par b .

Cela se fait facilement avec Python: il existe des opérateurs ou des fonctions pour obtenir le quotient et/ou le reste d'une division euclidienne !

Python 3 Shell

```
>>> 235 // 23
10
>>> 235 % 23
5
>>> divmod(235, 23)
(10, 5)
>>>
```

La dernière valeur renvoyée est un **couple**; vous pouvez considérer que c'est une liste non modifiable (on dit plutôt non **mutable** ou non **muable**).

On cherche ici à écrire une fonction semblable à la fonction `divmod`, mais qui utilise pour seule opération sur les entiers des additions, des soustractions et des comparaisons. Bien sûr, à moins que ce soit le thème de l'exercice (comme ici), on utilise les fonctions standard.

§2 Algorithme des soustractions successives

On présente ici l'algorithme de division euclidienne des entiers naturels tel qu'il est présenté à l'école primaire. C'est (presque) la manière dont elle est implémentée dans les processeurs car elle évite les coûteuses multiplications.

Pour $a, b \in \mathbb{N}$ avec $b \neq 0$, on veut calculer l'unique couple d'entiers (q, r) tels que

- Si $a > b$, on retranche b à a , on note r le résultat.
- Tant que $r \geq b$, on recommence: si $r > b$, retranche b à r , on note (encore) r le résultat.
- Lorsque $r < b$, nous avons obtenu le reste de la division euclidienne de a par b ; le quotient est le nombre de soustractions effectuées.

Ainsi, avec $a = 235$ et $b = 23$, les soustractions successives donnent pour r (en partant de 235)

étape	0	1	2	3	4	5	6	7	8	9	10
r	235	212	189	166	143	120	97	74	51	28	5

Une première idée, on utilise une succession d'instruction `if`:

```

1  def division_if(a, b):
2      q = 0
3      r = a
4      if r >= b:
5          r = r - b
6          q = q + 1
7
8      if r >= b:
9          r = r - b
10         q = q + 1
11
12     if r >= b:
13         r = r - b
14         q = q + 1
15
16     if r >= b:
17         r = r - b
18         q = q + 1
19
20     if r >= b:
21         r = r - b
22         q = q + 1
23
24     if r >= b:
25         r = r - b
26         q = q + 1
27
28     return (q, r)

```

Testons:

Python 3 Shell

```
>>> division_if(34, 7)
(4, 6)
>>> division_if(41, 12)
(3, 5)
>>> division_if(5, 21)
(0, 5)
>>> division_if(235, 23)
(6, 97)
>>>
```

Les premiers tests sont concluants:

$$34 = 4 * 7 + 6 \text{ et } 6 < 7$$

$$41 = 3 * 12 + 5 \text{ et } 5 < 12$$

$$5 = 0 * 21 + 5 \text{ et } 5 < 21$$

Mais le dernier test montre les limites de notre solution: on a bien $235 = 6 * 23 + 97$ mais $97 \geq 23$. Il manque quelques soustractions pour arriver au résultat! Nous pourrions rajouter une centaine de blocs `if`, ce qui fonctionnerait pour la division de 235 par 23 (et de nombreux autres cas), mais cela serait toujours insuffisant pour certains nombres.

Les ordinateurs sont très forts pour gérer les tâches répétitives. Il y a donc heureusement des structures de répétitions prêtes à l'emploi en Python (et dans la quasi totalité des langages de programmation).

L'algorithme de division euclidienne par division successives peut-être implémenté de la manière suivante

```
1 def division(a, b):
2     q = 0
3     r = a
4     while r >= b:
5         r = r - b
6         q = q + 1
7     return (q, r)
```

Avec $a = 235$ et $b = 23$, on obtient

Python 3 Shell

```
>>> division(235, 23)
(10, 5)
```

La ligne 4, `while r >= b:`, permet d'exécuter le **bloc** formé par les lignes 5 et 6 *tant que* la condition $r \geq b$ reste vérifiée. Lorsque cette condition n'est plus vérifiée (on a $r < b$), la boucle s'arrête et l'on passe à la ligne 7 qui retourne le résultat.

§3 La notion de compteur de boucle

Il est possible de compter les «tours de boucle» à l'aide d'une variable entière qu'on initialise à 0 et dont on augmente la valeur de 1 à chaque tour. On peut ensuite utiliser ce compteur de deux façons différentes:

- soit simplement pour en exploiter la valeur, aussi bien au sein des instructions de la boucle qu'après la fin de la boucle; nous parlerons «d'exploitation passive» du compteur. C'est le cas par exemple de la variable `q` de la fonction `division`.
- Soit pour limiter effectivement le nombre de tours de boucle en introduisant une condition de poursuite faisant intervenir le compteur: nous parlerons «d'exploitation active» du compteur.

§4 L'instruction `print` pour le déboguage du pauvre

Il est parfois difficile d'obtenir le résultat voulu à la première écriture d'une boucle tant que l'on n'a pas les outils adéquats. Nous abandonnerons bien vite les tâtonnements successifs pour adopter une démarche rigoureuse¹.

La seule façon de rendre compte d'un programme, c'est d'explicitier les situations qu'il engendre. Dès que le programme comporte une boucle, les choses se compliquent singulièrement.

Il est pour cela utile de dérouler les premières étapes «à la main». Si j'effectue l'appel `division(235, 23)`, alors

- $q = 0$ et $r = 235$;
- puisque $r = 235 \geq b = 23$, on obtient $r = 235 - 23 = 212$ et $q = 0 + 1 = 1$,
- puisque $r = 212 \geq b = 23$, on obtient $r = 212 - 23 = 189$ et $q = 1 + 1 = 2$,
- ...

S'il y a 300 tours de boucle, c'est un peu long... On peut également utiliser l'instruction `print` afin de connaître les valeurs associées à certaines variables en cours d'exécution. Par exemple, ajouter la ligne `print("q=", q, "r=", r)` dans le corps de la boucle.

```
1 def division(a, b):
2     q = 0
3     r = a
4     while r >= b:
5         r = r - b
6         q = q + 1
```

¹Voici un extrait du programme disant, en gros, que tout doit déjà être planifié avant d'écrire la moindre ligne de code:

L'introduction à l'algorithmique contribue à apprendre à l'étudiant à analyser, à spécifier et à modéliser de manière rigoureuse une situation ou un problème. Cette démarche algorithmique procède par décomposition en sous-problèmes et par affinements successifs. L'accent étant porté sur le développement raisonné d'algorithmes, leur implantation dans un langage de programmation n'intervient qu'après une présentation organisée de la solution algorithmique, indépendante du langage choisi.

Les invariants de boucles sont introduits pour s'assurer de la correction des segments itératifs.

```

7     print("q=", q, "r=", r)
8     return (q, r)

```

Ce qui donne,

Python 3 Shell

```

>>> division(235, 23)
q= 1 r= 212
q= 2 r= 189
q= 3 r= 166
q= 4 r= 143
q= 5 r= 120
q= 6 r= 97
q= 7 r= 74
q= 8 r= 51
q= 9 r= 28
q= 10 r= 5
(10, 5)
>>>

```

La syntaxe élémentaire pour l'instruction `print` est

```
print(arg1, arg2, arg3, arg4, ...)
```

chaque argument étant affiché à la suite des autres.

- Si l'argument est une chaîne de caractères (délimitée par `".."` ou `'..'`), alors le texte s'affiche tel quel.
- Si l'argument est une expression^a, alors, elle est évaluée et représentée.

^aDisons expression numérique pour l'instant, car une chaîne de caractères est une expression

Remarquez que tout n'est pas toujours bien aligné (comme le `q= 10 r= 5` précédent), mais nous réglerons ça dans un prochain TP.



Il faut bien comprendre que l'instruction `print` ne modifie pas la valeur de retour de la fonction qui est celle obtenue par l'instruction `return`.

On peut le constater avec les tests suivants

Python 3 Shell

```

>>> result = division(235, 23)
q= 1 r= 212
q= 2 r= 189
q= 3 r= 166
q= 4 r= 143
q= 5 r= 120
q= 6 r= 97
q= 7 r= 74

```



```

q= 8 r= 51
q= 9 r= 28
q= 10 r= 5
>>> result
(10, 5)
>>>

```

Essayez d'enlever la ligne return. Que se passe-t-il ?

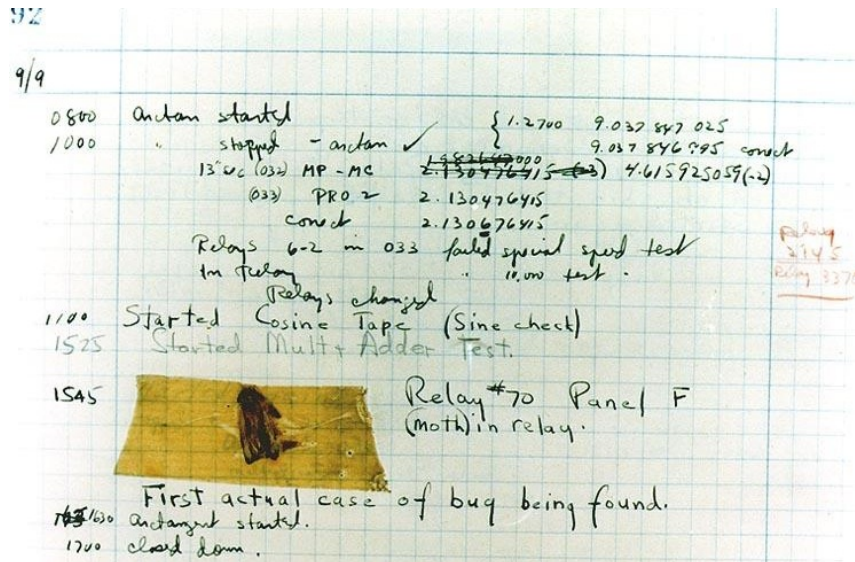


Figure 102.1: Le premier bug (9 septembre 1947)

§5 Notion de bloc d'instruction



- Vous pouvez télécharger les fonctions de l'exercice sur <http://www.prepa-baggio.fr/info/tp102/partie1.py>

- Vous placerez vos explications dans le fichier `partie1.py`, entre triple quote.

Comme c'est le cas pour la structure conditionnelle, l'indentation fait partie intégrante de la syntaxe Python. Seule la partie indentée sous le mot-clef `while` fait partie du corps de la boucle.

Exercice 1. Dans les questions suivantes, on modifie l'indentation du programme `division`. Expliquer à chaque fois le résultat obtenu (on pourra utiliser une instruction `print` pour suivre le déroulement du programme).

1.

```

1 def division1(a, b):
2     q = 0
3     r = a
4     while r >= b:
5         r = r - b

```

```

6     q = q + 1
7     return (q, r)

```

2.

```

1 def division2(a, b):
2     q = 0
3     r = a
4     while r >= b:
5         r = r - b
6     q = q + 1
7     return (q, r)

```

3.

```

1 def division3(a, b):
2     q = 0
3     r = a
4     while r >= b:
5         r = r - b
6         q = q + 1
7     return (q, r)

```

102.2 EXERCICES

§1 Premières boucles



Les exercices de cette partie sont à rendre dans le fichier `partie2.py`

Exercice 2. Écrire une fonction `temps_placement` ayant trois arguments

- `capital_ini` représentant le capital initial,
- `taux` le taux (annuel) auquel sera effectué le placement,
- `capital_cible` le capital que l'on souhaite obtenir.

Elle retourne le nombre d'années (un entier) de placement nécessaire pour dépasser le capital cible. Par exemple, il me faudra 142 ans pour devenir millionnaire en plaçant 1000€ à un taux de 5%.

Python 3 Shell

```

>>> temps_placement(1000, 0.05, 1000000)
142

```

Indication : Utiliser un compteur de boucle `annee`.

Exercice 3.

Dans cet exercice, on n'utilisera pas la fonction `sqrt` ou des puissances $1/2$.

1. Écrire une fonction `racine_int` ayant un paramètre n entier (positif ou nul) et qui retourne la racine carrée entière par défaut (entier positif ou nul), c'est-à-dire l'entier r vérifiant

$$r^2 \leq n < (r+1)^2.$$

Pour cela, on teste chaque valeur de r possible (en partant de 0) jusqu'à trouver la bonne.

On aura par exemple:

Python 3 Shell

```
>>> racine_int(16)
4
>>> racine_int(21)
4
>>> racine_int(35000)
187
```

2. Compléter la fonction `test_racine_int` afin de prendre en compte quelques cas limite supplémentaires.



Exercice 4. Écrire une fonction `sommechiffre` avec un paramètre entier positif n , qui retourne la somme des chiffres de l'entier n .

Indication :

- $q // 10$ retourne le quotient de la division euclidienne de q par 10 : c'est donc le nombre q auquel on a ôté son dernier chiffre.
- $q \% 10$ retourne le reste de la division euclidienne de q par 10 : c'est donc le chiffre des unités de l'entier q .
- `divmod(q, 10)` retourne le couple formé des deux valeurs précédentes.

§2 Quelques célébrités

Exercice 5. *Conjecture de Syracuse (ou de Collatz, ou Kakutani, ou Ulam).*

On considère une suite récurrente définie par $u_0 \in \mathbb{N}^*$ et

$$\forall n \in \mathbb{N}, u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

La conjecture de Syracuse affirme

Quel que soit le nombre u_0 , il existe un $n \in \mathbb{N}$ tel que $u_n = 1$, puis on retrouvera toujours la succession 4, 2, 1.

Par exemple, pour $u_0 = 11$, on a

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
u_n	11	34	17	52	26	13	40	20	10	5	16	8	4	2	1	4	2	1

1. Écrire une fonction `syracuse` avec deux paramètres `u0` et `n` et qui retourne le terme d'indice `n` de la suite de syracuse de premier terme `u0`. Par exemple

Python 3 Shell

```
>>> syracuse(11, 9)
5
>>>
```

2. On appelle **vol** la liste des u_n jusqu'à l'obtention du premier 1. Par exemple, le vol pour $u_0 = 11$ est

11	34	17	52	26	13	40	20	10	5	16	8	4	2	1
----	----	----	----	----	----	----	----	----	---	----	---	---	---	---

La **durée du vol** est le nombre de termes du vol moins 1, c'est-à-dire l'indice du premier 1 obtenu. Par exemple pour $u_0 = 11$, la durée du vol est 14.

Écrire une fonction `duree_vol` avec un paramètre `u0` qui retourne la durée du vol démarrant à `u0`.

Python 3 Shell

```
>>> duree_vol(11)
14
```

3. On appelle **altitude du vol**, le terme maximum atteint par la suite u_n , par exemple, pour $u_0 = 11$, l'altitude du vol est 52.

Écrire une fonction `altitude_vol` avec un paramètre `u0` qui retourne l'altitude du vol de premier terme `u0`.

Python 3 Shell

```
>>> altitude_vol(11)
52
```

Exercice 6. Écrire une fonction `pgcd` qui retourne le pgcd de deux entiers naturels.

On utilisera uniquement les relations suivantes, valables pour tout $(a, b) \in \mathbb{N}^2$:

$$\begin{aligned} \text{pgcd}(0, b) &= b, \\ \text{pgcd}(a, b) &= \text{pgcd}(b, a), \\ a \leq b &\implies \text{pgcd}(a, b) = \text{pgcd}(a, b - a). \end{aligned}$$



Exercice 7. On souhaite utiliser l'algorithme de babylone pour déterminer la racine carrée d'un nombre réel $a > 0$. Partant d'une approximation x_0 de \sqrt{a} , on construit par récurrence la suite

$$x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k} \right).$$

Écrire une fonction `racine_bab` avec deux paramètres `a` et `epsilon` qui retourne la première approximation x_k de \sqrt{a} telle que

$$|x_k - x_{k-1}| \leq \epsilon.$$

Vous pouvez choisir n'importe quelle première approximation, par exemple 1, a , ou $\frac{a+1}{2}$.

Python 3 Shell

```
>>> racine_bab(2, 1e-3)
1.4142135623746899
>>> sqrt(2)
1.4142135623730951
>>>
```

Indication : La condition d'arrêt dépendant de x_k et de la valeur précédente x_{k-1} , vous aurez donc besoin des deux références pendant votre boucle. On peut donc utiliser une variable `x_prec` pour stocker la «valeur précédente»...

102.3 ALGORITHME EN LIGNE



Les exercices de cette partie sont à rendre dans le fichier `partie3.py`

Un algorithme est dit **en ligne** (**on line**) lorsque les données nécessaires sont fournies au fur et à mesure de son exécution. Pour simuler ce phénomène, nous allons utiliser la fonction `input`, qui permet de demander à l'utilisateur d'entrer une chaîne de caractères au clavier.

Exercice 8. *Devinez le nombre.*

Écrire une fonction `deviner_le_nombre` ayant un paramètre entier `val_max` (entier supérieur ou égal à 1) et qui produit une session du jeu «Devinez le nombre». Le nombre à deviner est compris entre 1 et `val_max` inclus. Cette fonction retourne le nombre d'essais effectués pour trouver le nombre.

Le dialogue avec l'utilisateur se présentera ainsi:

Python 3 Shell

```
>>> nb_tentatives = deviner_le_nombre(10000)
Un nombre entre 1 et 10000 a été choisi au hasard...
Devinez le nombre : 2345
Trop petit
Devinez le nombre : 8180
Trop grand
Devinez le nombre : 6321
Trop grand
Devinez le nombre : 4084
Bravo !
>>> nb_tentatives
4
>>>
```

La valeur 4 apparaissant à la fin est la valeur retournée par la fonction. On pourra utiliser les instructions suivantes:

- `proposition = int(input("Devinez le nombre : "))` permet de demander à l'utilisateur d'entrer un nombre au clavier. Le résultat obtenu est un entier (de type `int`) référencé par le nom `proposition`.

- La fonction `randint` du module `random`. L'appel `randint(a, b)` retourne un entier de $[a, b]$, bornes incluses.

Exercice 9. Les générateurs de nombres aléatoires peuvent être utilisés pour créer des quizz simples. Par exemple, on peut vérifier la connaissance de vos tables de multiplications : un programme choisit deux nombres aléatoires entre 1 et 9 et pose une question semblable à

Combien font 6 fois 8 ?

L'utilisateur entre la réponse et l'ordinateur vérifie si le résultat est correct, puis renvoie un message approprié.

1. Implémenter ce quizz dans une fonction `table_de_multiplications`, sans paramètre.

- Si le résultat est correct, on recommence avec une nouvelle question.
- Si le résultat n'est pas correct, on affiche le nombre de bonnes réponses consécutives obtenues.

2. Jouer et obtenir un score supérieur à 100.

TP

103

LA RÉPÉTITION «POUR CHAQUE»



À rendre avant le

- Les exercices **1, 3, 4, 5, 6, 7, 8, 10, 11**, sont à rendre dans un fichier nommé `tp103.py`



- Les questions ☹ sont plus difficiles. Vous pouvez donc les passer dans un premier temps.

- Chaque fichier devra comporter un commentaire dans l'en-tête avec vos nom, prénom, classe, groupe et numéro du TP.

- Une exécution du fichier avec «F5» dans IDLE ne devra pas provoquer d'erreur.

- Les noms des fichiers et des fonctions demandées doivent être respectés.

- La note ne valide pas seulement le résultat de votre programme, mais également son style

- choix approprié des noms de variables,

- présence de documentation et de tests pour les fonctions importantes,

- commentaires pertinents.

- Vous pouvez ajouter des résultats obtenus dans la console interactive (Shell) à la suite de vos fonctions, sur plusieurs lignes, entre triple quote (`"""....."""`).

Le non-respect d'un des points précédents peut retrancher jusqu'à 50 points sur la note finale.

103.1 PARCOURS D'UN ITÉRABLE

Dans la suite, nous allons implémenter quelques opérations usuelles sur les listes. Ces fonctions concentrent quelques techniques classiques appliquées aux tableaux à une dimension. Conformément au programme, vous devez être capable d'écrire des fonctions équivalentes par vous-même.



Dans cette partie, on s'interdira donc l'usage de `min`, `max`, `sum`, l'opération `in`, `index`, `count`.

§1 L'instruction `for`

Voici un bloc d'instruction qui calcule le produit des 7 premiers nombres premiers

$$1 \times 2 \times 3 \times 5 \times 7 \times 11 \times 13$$

```
1 acc = 1
2 for elem in [2, 3, 5, 7, 11, 13]:
3     acc = acc * elem
```

La variable `elem` prend les valeurs successives de la liste `[2, 3, 5, 7, 11, 13]` et *pour chaque* valeur de `elem`, le bloc d'instruction `acc = acc * elem` est exécuté.

On peut tracer l'exécution de ce programme ainsi:

Tour de boucle	elem	acc (début)	acc (fin)
1	2	1	2
2	3	2	6
3	5	6	30
4	7	30	210
5	11	210	2310
6	13	2310	30030

- On dit que la liste `[2, 3, 5, 7, 11, 13]` est un **itérable**. Plus généralement, un **itérable** est un objet qui accepte l'itération, ce sur quoi on peut itérer, c'est à dire une collection dont on peut prendre les éléments un à un.

- L'instruction `for` sert à parcourir un itérable. À chaque itération (c'est-à-dire «tour de boucle»), la variable prend la valeur suivante de l'itérable.

- La variable `acc` joue le rôle d'**accumulateur**: c'est-à-dire une variable qui est modifiée avec chaque nouvelle donnée. Un accumulateur est très utile pour calculer des totaux par exemple.

Test 1

Au fait, pourquoi a-t-on initialisé la variable `acc` avec la valeur 1?

§2 L'itérable range

Rappelons que `range(p, q)` est une version «virtuelle» de la liste

`[p, p+1, p+2, ..., q - 2, q-1]`

qui va de `p` inclus à `q-1` exclu!

Python 3 Shell

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

Voici par exemple une fonction qui calcule $n! = \prod_{k=1}^n k$.

```
1 def factorielle(n):
2     produit = 1
3     for i in range(1, n + 1):
4         produit = produit * i
5     return produit
```

§3 Éléments d'un tableau

Rappelons qu'en Python, les éléments d'une liste de longueur n sont indicé de 0 à $n-1$. On peut également utiliser des indices négatifs. Par exemple, avec la liste

`A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]`.

On a les indices

i	0	1	2	3	4	5	6	7	8	9
A[i]	23	17	80	39	58	63	96	42	0	74
i	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Pour calculer le produit des éléments d'une liste `A`, on peut itérer sur les indices

```
1 n = len(A)
2 acc = 1
3 for i in range(n):      # Équivalent à range(0, n)
4     acc = acc * A[i]
```

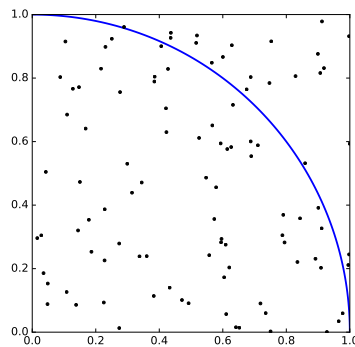
103.2 COMPTAGE

Exercice 1. Écrire une fonction de prototype `compte(A, x)` qui dénombre le nombre d'occurrences de l'élément `x` dans le tableau `A`.



Exercice 2. La but de cet exercice est d'écrire une fonction qui calcule expérimentalement la valeur de π en utilisant la méthode de Monte-Carlo. Elle consiste à tirer des points au hasard dans un carré de côté 1, dont l'aire vaut donc 1, et à compter le nombre de ces points qui tombent dans le quart de disque de rayon 1 inscrit dans ce carré, dont l'aire vaut $\pi/4$. Si ce nombre est assez élevé, la proportion de points contenus dans le quart de cercle approche le rapport de l'aire du quart de disque sur l'aire du carré, c'est-à-dire $\pi/4$.

La figure ci dessous illustre cette idée.



Calcul de π par la méthode de Monte-Carlo

Sur cette figure, 100 points ont été tirés au hasard, 77 sont dans le disque. Il y a donc une proportion de 77 pour cent de points dans le disque. On utilise alors l'approximation,

$$\text{Aire}(\text{disque}) = \frac{77}{100} \text{Aire}(\text{carré}).$$

Ce qui donne l'approximation

$$\frac{\pi}{4} \approx \frac{77}{100} = 0.77, \quad \text{c'est-à-dire} \quad \pi \approx 3.08.$$

Écrire une fonction `approx_pi` avec pour argument le nombre de points à tirer et qui retourne une approximation de π .

On utilisera la fonction `random` du module `random`: `random.random()` retourne un flottant de l'intervalle $[0, 1[$.

Cette méthode ne permet cependant pas d'espérer plus de quelques décimales. Ainsi, avec 10 millions de points, on n'obtient généralement que 3 décimales correctes.

103.3 ACCUMULATION

§1 Accumulation systématique

Exercice 3. Dans cet exercice, nous ré-implémentons la fonction `sum` de Python.

Écrire une fonction `somme_tableau` ayant pour argument un tableau et qui retourne la somme de ses éléments. On suppose que tous les éléments du tableau sont de type numérique.

Python 3 Shell

```
>>> somme_tableau([3, 4, 8])
15
>>> somme_tableau([])
0
>>> somme_tableau([0, 1, 2, 3])
6
```

Rappel

Pour une série statistique discrète $x = (x_1, \dots, x_p)$, la moyenne (arithmétique) μ et la variance μ_2 sont données par

$$\mu = \bar{x} = \frac{1}{p} \sum_{i=1}^p x_i \quad \text{et} \quad \mu_2 = \frac{1}{p} \sum_{i=1}^p (x_i - \mu)^2$$

L'écart-type (ou écart quadratique moyen) étant $\sigma = \sqrt{\mu_2}$.

Exercice 4. Écrire une fonction `somme_partielle` avec trois paramètres

- une fonction `f` ayant un paramètre entier;
- deux entiers `p` et `q`.

L'appel `somme_partielle(f, p, q)` retourne la valeur de la somme

$$\sum_{k=p}^q f(k).$$

Rappelons que, par convention, si $p > q$, cette somme égale 0.

Pour écrire une fonction de test, on pourra définir les fonctions `identite` et `carre` :

Python 3 Shell

```
>>> somme_partielle(identite, 12, 310)
48139
>>> somme_partielle(carre, 9, 130)
740601
>>>
```

103.4 RECHERCHE SÉQUENTIELLE

Dans cet exercice, nous ré-implémentons les fonctions `min` et `max`. Celles-ci sont donc interdites!

Exercice 5.

1. Écrire une fonction `maximum` ayant pour argument une liste non vide et qui retourne le maximum de cette liste. On suppose que tous les éléments de la liste sont comparables avec l'opérateur `<=`.

2. Écrire une fonction `minimum` ayant pour argument une liste non vide et qui retourne le minimum de cette liste. On suppose que tous les éléments de la liste sont comparables avec l'opérateur `<=`.

Python 3 Shell

```
>>> A = [1, 2, 3, 8, 12, 23, -5, -42, 7, 11]
>>> maximum(A)
23
```

Exercice 6.

1. Écrire une fonction `maximant` ayant pour argument une liste non vide et qui retourne un maximant de cette liste, c'est-à-dire la position d'un maximum.

2. Que se passe-t-il lorsque le maximum est atteint plusieurs fois?

Python 3 Shell

```
>>> A = [1, 2, 3, 8, 12, 23, -5, -42, 7, 11]
>>> maximant(A)
5
```



Exercice 7. Écrire une fonction `doubletop` ayant pour argument un tableau `A`.

- Pré condition. Le tableau `A` contient au moins deux éléments.
- Pré condition. Tous les éléments du tableau `A` sont comparables avec l'opérateur `<=`.
- Résultat. Cette fonction retourne un couple `(M1, M2)` où `M1` est le plus grand élément du tableau `A` et `M2` le second plus grand élément du tableau `A`.

Python 3 Shell

```
>>> doubletop([36, 7, 25, 45, 9, 9, 35, 42])
(45, 42)
>>> doubletop([28, 38, 24, 20, 22, 7, 23, 41, 43, 33])
(43, 41)
>>> doubletop([50, 20])
(50, 20)
```

```
>>> doubletop([46, 21, 28, 27, 49, 43, 35, 49])
(49, 49)
>>>
```

103.5 RÉPÉTITIONS IMBRIQUÉES

Considérons ces deux boucles imbriquées:

```
1 for i in range(3):
2     for j in range(5):
3         print("i = ", i, ", j = ", j)
```

L'instruction d'affichage (`print`) se trouve exécutée cinq fois pour chaque valeur de `i`. À chaque fois, la variable `j` prend successivement les valeurs 0, 1, 2, 3, 4. Nous obtenons le résultat

Python 3 Shell

```
i = 0 , j = 0
i = 0 , j = 1
i = 0 , j = 2
i = 0 , j = 3
i = 0 , j = 4
i = 1 , j = 0
i = 1 , j = 1
i = 1 , j = 2
i = 1 , j = 3
i = 1 , j = 4
i = 2 , j = 0
i = 2 , j = 1
i = 2 , j = 2
i = 2 , j = 3
i = 2 , j = 4
```



Exercice 8. Écrire une fonction `somme_double` avec trois paramètres

- une fonction `f` à deux paramètres;
- deux entiers `p` et `q`.

L'appel `somme_double(f, p, q)` retourne la valeur de la somme

$$\sum_{\substack{0 \leq i < p \\ 0 \leq j < q}} f(i, j).$$

Python 3 Shell

```
>>> somme_double(max, 5, 7)
125
>>> somme_double(min, 5, 7)
50
>>> somme_double(pow, 3, 11)
2059
>>> somme_double(math.gcd, 3, 121)
7563
>>> somme_double(math.atan2, 313, 211)
60815.333861253435
```



Exercice 9. Écrire des fonctions `somme_tsup` et `somme_tinf` avec deux paramètres

- une fonction `f` à deux paramètres;
- un entiers `n`.

Les appels `somme_tsup(f, n)` et `somme_tinf(f, n)` retournent respectivement

$$\sum_{1 \leq i \leq j < n} f(i, j) \quad \text{et} \quad \sum_{1 \leq j \leq i < n} f(i, j).$$

Python 3 Shell

```
>>> somme_tsup(max, 10)
285
>>> somme_tsup(min, 10)
165
>>> somme_tsup(pow, 23)
534487089095774230975553821392
>>> somme_tsup(math.gcd, 313)
207514
>>> somme_tsup(math.atan2, 313)
21549.17975955823
>>> somme_tinf(max, 13)
650
>>> somme_tinf(min, 13)
364
>>> somme_tinf(pow, 23)
363933571397116795153052659085
>>> somme_tinf(math.gcd, 313)
207514
>>> somme_tinf(math.atan2, 313)
55149.663285182796
```

103.6 SUITES RÉCURRENTES

Exercice 10. On considère la suite récurrente définie par $u_0 = 0$ et

$$\forall n \in \mathbb{N}, u_{n+1} = \sqrt{u_n + n + 1}$$

Écrire une fonction `suite_recurrente` avec un paramètre `n` et qui renvoie le terme u_n .

Python 3 Shell

```
>>> suite_recurrente(10)
3.6759796411289134
>>> suite_recurrente(100)
10.509990605994101
>>> suite_recurrente(1000)
32.12647919677667
>>>
```



Exercice 11. Écrire une fonction `fibonacci` prenant un entier n en entrée, et retournant F_n le terme d'indice n la suite de Fibonacci. On rappelle

$$F_0 = 0$$

$$F_1 = 1$$

$$\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n.$$

Python 3 Shell

```
>>> fibonacci(12)
144
```

TP

104

TABLEAUX À UNE DIMENSION

Exercice 1. Écrire une fonction `est_croissante` ayant pour argument une liste d'éléments comparables. Cette fonction

- retourne `True` si les entrées sont triées par ordre monotone croissant,
- retourne `False` sinon.

N'oubliez pas de tester quelques cas extrêmes. On considérera que la liste vide est croissante.

104.1 SUITES RÉCURRENTES

Exercice 2. Écrire une fonction `fibonacci` prenant un entier n en entrée, et retournant la liste des termes

$$[F_0, F_1, \dots, F_n]$$

de la suite de Fibonacci définie par

$$F_0 = 0$$

$$F_1 = 1$$

$$\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n.$$

Python 3 Shell

```
>>> fibonacci(0)
[0]
>>> fibonacci(1)
[0, 1]
>>> fibonacci(12)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
>>>
```

104.2 TRI PAR PROPAGATION (TRI À BULLES)

Le principe du tri par propagation, généralement connu comme le tri à bulles est simple. La métaphore est que le début de la liste symbolise le fond de l'eau et la fin de la liste la surface. Chaque nombre représente le diamètre d'une bulle et la bulle la plus grosse est celle qui remonte le plus vite à la surface. Plus formellement, on parcourt la liste de la gauche vers la droite en échangeant les termes contigus $L[i]$ et $L[i+1]$ s'ils sont mal rangés, c'est-à-dire si $L[i+1] < L[i]$ (on propage la bulle $L[i]$). À ce stade, la bulle la plus grosse (la plus grande valeur) est au bout de la liste en dernière position. Voici le déroulé de cette première étape avec la liste $L = [1, 4, 3, 5, 6, 0, 2]$. À chaque étape, les deux termes comparés sont matérialisés sur un fond rouge ou fond vert selon que leurs indices constituent une inversion et qu'il faut les échanger ou non.

1	4	3	5	6	0	2
1	3	4	5	6	0	2
1	3	4	5	6	0	2
1	3	4	5	6	0	2
1	3	4	5	0	6	2
1	3	4	5	0	2	6

Le terme le plus grand étant à présent placé au bout de la liste, il suffit de recommencer le processus de propagation, mais cette fois sur la sous-liste $L[0:n-1]$ contenant les $n - 1$ premiers termes de la liste et ainsi de suite en éliminant le dernier terme de la liste à chaque passe. L'algorithme est constitué d'une boucle principale qui fait varier la taille de la zone $[0, d]$ à parcourir dans la liste L en commençant par la liste entière et en appelant à chaque étape l'algorithme de propagation.

1	3	4	5	0	2	6
1	3	4	5	0	2	6
1	3	4	5	0	2	6
1	3	4	0	5	2	6
1	3	4	0	2	5	6

1	3	4	0	2	5	6
1	3	4	0	2	5	6
1	3	0	4	2	5	6
1	3	0	2	4	5	6

1	3	0	2	4	5	6
1	0	3	2	4	5	6
1	0	2	3	4	5	6

0	1	2	3	4	5	6
0	1	2	3	4	5	6

0	1	2	3	4	5	6
---	---	---	---	---	---	---

Exercice 3.

1. Appliquer en mode «papier-crayon» l'algorithme de tri à bulles au tableau

[2, 1, 6, 9, 8, 4]

2. Écrire une fonction `tri_bulles` ayant pour argument une liste `L` et qui la trie par l'algorithme de tri à bulle.

Python 3 Shell

```
>>> L = [1, 4, 3, 5, 6, 0, 2]
>>> tri_bulles(L)
>>> L
[0, 1, 2, 3, 4, 5, 6]
```

104.3 TRANSFORMÉE DE FOURIER DISCRÈTE

Dans la suite, i désigne le nombre complexe tel que $i^2 = -1$.

En python, celui-ci est représenté par `1j` (il n'y a pas de signe `*`). On peut par exemple définir les nombres complexes $a = 3i$, $b = 2 - 5i$ par

Python 3 Shell

```
>>> a = 3j
>>> b = 2 - 5j
>>> type(a), type(b)
(<class 'complex'>, <class 'complex'>)
>>> a * b
(15+6j)
```

L'exponentielle complexe définie pour $(x, y) \in \mathbb{R}^2$ par

$$\exp(x + iy) = e^x (\cos y + i \sin y)$$

peut être importée du module `cmath`

Python 3 Shell

```
>>> from cmath import exp, pi
>>> exp(0)
(1+0j)
>>> exp(1)
(2.718281828459045+0j)
>>> exp(pi / 2 * 1j)
(6.123233995736766e-17+1j)
```

La représentation en flottant du nombre π n'étant pas exacte, vous remarquerez que `exp(pi / 2 * 1j)` ne retourne pas exactement $\exp\left(\frac{\pi}{2}i\right) = i$.

Dans la suite, on appelle **vecteur** une liste python contenant des nombres complexes. Une **transformée de Fourier discrète** est une transformation linéaire de \mathbb{C}^n dans \mathbb{C}^n définie de la manière suivante. Soit x un vecteur de nombres complexes de longueur n . Les éléments de x sont donc indexés par l'intervalle d'entier $\llbracket 0, n-1 \rrbracket$. Pour tout entier $k \in \llbracket 0, n-1 \rrbracket$, la valeur $X[k]$ de la transformée de Fourier discrète du vecteur x se définit par:

$$X[k] = \sum_{m=0}^{n-1} x[m] \cdot \exp\left(-\frac{2\pi \cdot i}{n} \cdot m \cdot k\right)$$


Exercice 4. Construire une fonction `DFT(x)` qui calcule la transformée de Fourier discrète X d'un vecteur x de nombres complexes.

Python 3 Shell

```
>>> DFT([3, 6, 1j])
[(9+1j),
 (-0.866025403784437-5.696152422706632j),
```

(0.8660254037844365+4.696152422706631j)]

104.4 COEFFICIENTS BINOMIAUX

 **Exercice 5.** On souhaite écrire une fonction `binom` ayant pour paramètre un entier naturel n et qui retourne la liste des coefficients binomiaux $\binom{n}{k}$ où $0 \leq k \leq n$.
Pour cela, on utilisera uniquement les relations

$$\binom{0}{k} = \begin{cases} 1 & : k = 0 \\ 0 & : k \neq 0 \end{cases} \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Python 3 Shell

```
>>> binom(0)
[1]
>>> binom(1)
[1, 1]
>>> binom(2)
[1, 2, 1]
>>> binom(7)
[1, 7, 21, 35, 35, 21, 7, 1]
>>>
```

104.5 MÉMENTO POUR LES LISTES

§1 Éléments d'un tableau

Les exemples sont donnés avec la liste

`A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]`.

Rappelons qu'en Python, les éléments d'une liste de longueur n sont indicé de 0 à $n-1$. On peut également utiliser des indices négatifs.

i	0	1	2	3	4	5	6	7	8	9
A[i]	23	17	80	39	58	63	96	42	0	74
i	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Création d'une liste, nombre d'éléments de la liste:

Python 3 Shell

```
>>> A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> type(A)
<class 'list'>
```

```
>>> len(A)
10
>>>
```

- On peut créer une liste de 12 éléments, ici tous les éléments sont initialisés à 0:

Python 3 Shell

```
>>> B = [0] * 12
>>> B
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> len(B)
12
>>>
```

- Accéder à l'entrée d'indice k de la liste : `A[k]`.

Python 3 Shell

```
>>> A
[23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> A[3]
39
>>> A[2000]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

N'oubliez pas que l'indexage d'une liste commence à 0. Son dernier élément est donc à l'indice `len(A) - 1`.

- Pour modifier une entrée, on utilise une affectation

Python 3 Shell

```
>>> A
[23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> A[3] = 300003
>>> A
[23, 17, 80, 300003, 58, 63, 96, 42, 0, 74]
>>>
```

§2 Les méthodes `append` et `pop`

Étant donnée une liste `A`, la méthode `append` ajoute un élément à la fin de la liste `A`.

Python 3 Shell

```
>>> A = [1, 8, 3, 4, 2]
>>> A
[1, 8, 3, 4, 2]
>>> id(A)
140587527769480
>>> A.append(99)
>>> A.append(-1)
>>> A
[1, 8, 3, 4, 2, 99, -1]
>>> id(A)
140587527769480
>>> res = A.append(23)
>>> res
>>> type(res)
<class 'NoneType'>
>>> A
[1, 8, 3, 4, 2, 99, -1, 23]
>>>
```

Remarquer que l'appel `A.append(23)` retourne la valeur `None`.

La méthode `append` est utile pour créer des listes, notamment lorsque l'on ne connaît pas par avance la longueur de celle-ci. Pour cela, on peut commencer par initialiser une liste vide `A = []`.

La méthode `pop` supprime le dernier élément de la liste et retourne sa valeur.

Python 3 Shell

```
>>> A
[1, 8, 3, 4, 2, 99, -1, 23]
>>> A.pop()
23
>>> A.pop()
-1
>>> A
[1, 8, 3, 4, 2, 99]
>>> x = A.pop()
>>> x
99
>>> A
[1, 8, 3, 4, 2]
```

§3 Opérations classiques avec Python

Voici quelques opérations classiques sur les listes et déjà définies en Python.

- Déterminer si un élément appartient à une liste et la position de sa première occurrence:

Python 3 Shell

```
>>> A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> 63 in A
True
>>> 69 in A
False
>>> A.index(63)
5
>>> A.index(69)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 69 is not in list
>>>
```

- Compter le nombre d'occurrences d'un élément dans une liste:

Python 3 Shell

```
>>> A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> A.count(63)
1
>>> A.count(69)
0
>>> B = [0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0]
>>> B.count(0)
5
>>> B.count(1)
6
>>> B.count(2)
0
>>>
```

- Déterminer le minimum, le maximum, la somme des éléments d'une liste:

Python 3 Shell

```
>>> A = [23, 17, 80, 39, 58, 63, 96, 42, 0, 74]
>>> min(A)
0
>>> max(A)
96
>>> sum(A)
492
```

§4 Copie et slicing

On peut extraire une partie d'une liste A à l'aide d'un *slicing*

Python 3 Shell

```
>>> A
[13, 66, 72, 79, 80, 81, 84, 90, 92, 95]
>>> A[3:7]    # Extraction des éléments d'indices 3 à 6.
[79, 80, 80, 84]
>>> A[3:]     # Extraction des éléments de à partir de l'indice 3.
[79, 80, 81, 84, 90, 92, 95]
>>> A[:7]     # Extraction des éléments jusqu'à l'indice 6.
[13, 66, 72, 79, 80, 81, 84]
>>> A[1:9]    # Extraction des éléments d'indices 1 à 8.
[66, 72, 79, 80, 81, 84, 90, 92]
>>> A[1:9:2]  # Extraction des éléments d'indices 1 à 8 avec un pas de 2.
[66, 79, 81, 90]
>>> A[1:9:3]  # Extraction des éléments d'indices 1 à 8 avec un pas de 3.
[66, 80, 90]
```

On peut obtenir une copie superficielle d'une liste A en utilisant la méthode `copy` (depuis Python 3.3) ou en utilisant un slicing `A[:]`.

Python 3 Shell

```
>>> A
[1, 8, 3, 4, 2]
>>> A
[1, 8, 3, 4, 2]
>>> B = A.copy()
>>> B
[1, 8, 3, 4, 2]
>>> C = A[:]
>>> C
[1, 8, 3, 4, 2]
>>> id(A), id(B), id(C)
(139964017379264, 139964015600704, 139964015661248)
>>> A == B
True
>>> A is B
False
```

On obtient trois objet différents (leurs identifiants sont distincts) mais qui ont même valeur.

TP

105

ALGORITHMES OPÉRANT SUR UNE STRUCTURE SÉQUENTIELLE PAR BOUCLES IMBRIQUÉES

105.1 RECHERCHE DES DEUX VALEURS LES PLUS PROCHES DANS UN TABLEAU

Exercice 1.

1. Écrire une fonction `valeurs_plus_proches` prenant en argument une liste de nombres `A` et retournant les deux valeurs les plus proches.

- Pré-condition : `A` est au moins de longueur 2.

Python 3 Shell

```
>>> valeurs_plus_proches([-2, 3, 8, 5, -9])  
(3, 5)    # la valeur (5, 3) est également acceptable
```

2. Combien de fois la fonction `abs` est-elle appelée lors de l'appel de `valeurs_plus_proches` sur une liste de longueur n ?

105.2 RECHERCHE D'UN MOTIF DANS UN TEXTE

§1 Introduction aux chaînes de caractères

- En python, un caractère se place entre guillemets droits (guillemets anglais ou «double quote») ou d'apostrophe droite («single quote»). On peut comparer deux caractères à l'aide de ==

Python 3 Shell

```
>>> 'a' == 'a'
True
>>> 'a' == 'A'
False
>>> 'a' == 'z'
False
>>> 'a' == "a"
True
```

- Pour créer un chaîne de caractère, il suffit d'entourer le texte de guillemets droits (guillemets anglais ou «double quote») ou d'apostrophe droite («single quote»).

```
1 bagage = 'Le Bagage ne répondit rien, mais plus fort, cette fois.'
2 mort = "il n'arriverait pas à comprendre les hommes de son vivant."
```

L'existence des deux formes permet d'intégrer des apostrophes dans une chaîne délimitée avec des guillemets.

- On peut également comparer les chaînes de caractères

Python 3 Shell

```
>>> "Bonjour" == 'Bonjour'
True
>>> "Bonjour" == 'bonjour'
False
>>> "Alice" < "Bob"
True
>>> "Bob" <= "Alice"
False
>>>
```

La comparaison à l'aide d'une inégalité se fait par ordre lexicographique. Ce n'est pas tout à fait l'ordre du dictionnaire, mais presque...

- Pour du texte long, on peut le scinder sur plusieurs lignes en utilisant les triples guillemets (ou triple apostrophe).

```
1 chou = """Le fait est connu dans tous les univers: quel
2 que soit le soin apporté au choix des couleurs d'un décor
3 d'établissement, on aboutit inmanquablement à du vert
```

```

4  dégueulis, du brun innommable, de jaune nicotine ou du
5  rose orthopédique. Par un phénomène encore mal compris de
6  résonance sympathique, ces couleurs dégagent toujours une
7  vague odeur de chou bouilli... Même si aucun chou ne cuit
8  dans les environs.""

```

- Une chaîne de caractère est de type `str`, pour `string`. Sa longueur est obtenue avec la fonction `len`

Python 3 Shell

```

>>> type('Alice')
<class 'str'>
>>> len('Alice')
5

```

- Les chaînes de caractères sont des itérables.

Python 3 Shell

```

>>> for caract in 'Alice':
...     print('*', caract, '*')
...
* A *
* l *
* i *
* c *
* e *
>>> list('Alice')
['A', 'l', 'i', 'c', 'e']

```

- On accède à une lettre par son indice. Comme pour les listes, les indices démarrent à 0 et on peut utiliser des indices négatifs.

i	0	1	2	3	4	5	6	7	8	9	10	11
s[i]	A	l	i	c	e		e	t		B	o	b
i	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Python 3 Shell

```

>>> s = 'Alice et Bob'
>>> s[3]
'c'
>>> s[-3]
'B'
>>> s[33]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

```

- Contrairement aux listes, on ne peut pas modifier les éléments d'une chaîne de caractères.

Python 3 Shell

```
>>> s[6] = 'o'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

§2 Échauffement : recherche dans une chaîne de caractères

Exercice 2. Dans cet exercice, on utilisera pour les chaînes de caractères, uniquement des opérations de comparaisons *entre caractères* et l'accès à un caractère par son indice.

1. Écrire une fonction `compte_lettre` ayant pour arguments une chaîne de caractère `texte` et un caractère `lettre` et qui retourne le nombre d'occurrences de `lettre` dans la chaîne `texte`.

2. Écrire une fonction `position_lettre` ayant pour arguments une chaîne de caractère `texte` et un caractère `lettre` et qui retourne la liste des positions de `lettre` dans le `texte`.

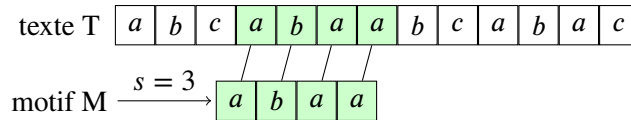
Python 3 Shell

```
>>> bagage
'Le Bagage ne répondit rien, mais plus fort, cette fois.'
>>> compte_lettre(bagage, 'e')
6
>>> position_lettre(bagage, 'e')
[1, 8, 11, 24, 45, 48]
```

§3 Recherche d'un motif dans un texte

Trouver toutes les occurrences d'une chaîne de caractère (motif) dans un texte est un problème qui se rencontre fréquemment dans les éditeurs de texte. Le plus souvent, le texte est un document en cours d'édition et la chaîne recherchée est un mot particulier fourni par l'utilisateur. Les algorithmes de recherche de chaîne de caractères sont également utilisés, par exemple, pour trouver une chaîne de caractères dans une séquence ADN, ou encore pour rechercher les pages Web correspondant à une requête soumise à un moteur de recherche Internet.

En nous référant à la figure



nous dirons que la chaîne M apparaît avec un décalage $s = 3$ dans le texte T.

Si M apparaît avec un décalage s dans T, alors le décalage s est un **décalage valide** ; sinon, s est un **décalage invalide**. Le **problème de la recherche d'une chaîne de caractères** revient à trouver tous les décalages valides pour lesquels un motif M apparaît dans un texte T donné.

Exercice 3. Dans cet exercice, on utilisera pour les chaînes de caractères, uniquement des opérations de comparaisons *entre caractères* et l'accès à un caractère par son indice.

1. Écrire une fonction `recherche_decalage` ayant pour arguments deux chaînes de caractères `texte` et `motif` et qui retourne le premier décalage valide du motif dans le texte.

- Précondition : Le motif apparaît dans le texte.

2. Écrire une fonction `recherche` ayant pour arguments deux chaînes de caractères `texte` et `motif` et qui retourne la liste des décalages valides du motif dans le texte.

Python 3 Shell

```
>>> ADN = "GATCCTCCATATACAACGGTATCTCCACCTCAGGTTTAGATCTCAACAAC"
>>> decalage(ADN, 'AA')
14
>>> recherche(ADN, 'AA')
[14, 44, 47]
>>> recherche(ADN, 'ACT')
[]
>>> recherche(ADN, 'AACG')
[14]
```

105.3 TRI PAR SÉLECTION

On considère le tri suivant de n nombres rangés dans un tableau A:

- On commence par trouver le plus petit élément de A et on le permute avec A[0].
- On trouve ensuite le plus petit élément de A[1:] (c'est le deuxième plus petit élément de A) et on le permute avec A[1].
- On continue de cette manière pour les $n - 1$ premiers éléments de A.

Ce tri s'appelle le tri par sélection ou tri par extraction.

La trace de l'exécution du tri par sélection pour la liste A ci dessous permet de comprendre le processus. Les deux termes qui seront permutés à l'issue de chaque étape sont distingué par un fond coloré.

1	4	3	5	6	0	2
0	4	3	5	6	1	2
0	1	3	5	6	4	2
0	1	2	5	6	4	3
0	1	2	3	6	4	5
0	1	2	3	4	6	5
0	1	2	3	4	5	6

Exercice 4.

1. Appliquer en mode «papier-crayon» l'algorithme de tri par sélection au tableau

[2, 1, 6, 9, 8, 4]

2. Écrire une fonction `tri_selection` ayant pour argument une liste A et qui la trie par l'algorithme de tri à bulle.

Python 3 Shell

```
>>> L = [1, 4, 3, 5, 6, 0, 2]
>>> tri_bulles(L)
>>> L
[0, 1, 2, 3, 4, 5, 6]
```

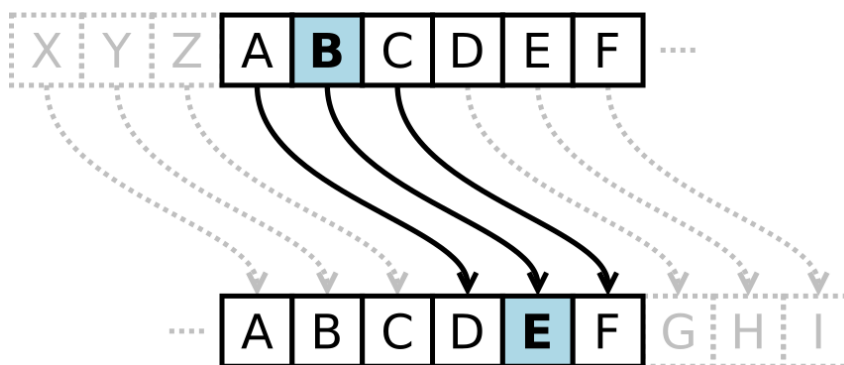
105.4 CHIFFREMENT PAR DÉCALAGE



Pour les plus rapides

En cryptographie, le chiffrement par décalage, aussi connu comme le chiffre de César ou le code de César, est une méthode de chiffrement très simple utilisée par Jules César dans ses correspondances secrètes (ce qui explique le nom «chiffre de César»).

Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Par exemple avec un décalage de 3 vers la droite, A est remplacé par D, B devient E, et ainsi jusqu'à W qui devient Z, puis X devient A etc... Il s'agit d'une permutation circulaire de l'alphabet. La longueur du décalage, 3 dans l'exemple évoqué, constitue la clé du chiffrement qu'il suffit de transmettre au destinataire — s'il sait déjà qu'il s'agit d'un chiffrement de César — pour que celui-ci puisse déchiffrer le message. Dans le cas de l'alphabet latin, le chiffre de César n'a que 26 clés possibles (y compris la clé nulle, qui ne modifie pas le texte).



Dans la suite, on supposera que les chaînes de caractères considérées ne contiennent que les lettres minuscules de a à z ou des espaces. On définira également la liste

```
1 alphabet = "abcdefghijklmnopqrstuvwxyz"
```

comme variable globale, ce qui signifie que l'on peut l'utiliser dans n'importe quelle fonction sans le prendre en argument.

Exercice 5. Écrire une fonction `code` prenant en argument une chaîne de caractère `texte` et un entier `c` et renvoyant le message obtenu par chiffrement de César.

Python 3 Shell

```
>>> texte
'personne ne remarque ce qui marche trop bien'
>>> code(texte, 3)
'shuvrqqh qh uhpduxh fh txl pdufkh wurs elhq'
>>>
```

On ignore quelle était l'efficacité du chiffre de César à son époque. La première trace de techniques de cryptanalyse date du IX^{ème} siècle avec le développement dans le monde arabe de l'analyse fréquentielle.

Pour décoder sans la clé un message suffisamment long, on peut essayer une attaque statistique. Si on suppose que le message est initialement écrit en français, la lettre la plus fréquente a de fortes chances d'être «e», ce qui permet alors de déterminer la clé.

Exercice 6.

1. Écrire une fonction `indice_max` prenant en argument une chaîne de caractères `message` et renvoyant l'entier associé à une lettre de fréquence maximale dans `message`.

Python 3 Shell

```
>>> indice_max("xsidxlxcxluxmxcalxyxaxjxlx")
23
```

car la lettre la plus fréquente est le «x», qui a pour indice 24 dans `alphabet`.

2. En déduire une fonction `decode` prenant en argument un chaîne de caractères `message` correspondant à un message codé, et tentant de le décoder par l'attaque statistique présentée.

Python 3 Shell

```
>>> indice_max('shuvrqqh qh uhpdutxh fh txl pdufkh wurs elhq')
7
>>> decode('shuvrqqh qh uhpdutxh fh txl pdufkh wurs elhq')
'personne ne remarque ce qui marche trop bien'
```
