CNESC Informatique

Python

Cours 1CB

2020–2021

Versions utilisées pour les tests
Python 3.8.3, 3.7.7, 3.6.1
Thonny 3.2.7
pygame 1.9.6, pillow 7.2.0, matplotlib 3.2.2, numpy 1.19.0

Auteurs de la version originale
Ben Kremer, David Mancini, Nino Silverio, Pascal Zeihen, François Zuidberg

à partir de la version 2.0
Ben Kremer, Pascal Zeihen, François Zuidberg

Version 2.1 – Luxembourg, le 2 juillet 2020

Table des matières

1 Réutilisation de code

1.1 Remarque préliminaire

Dans ce manuel, les conventions suivantes sont respectées quant au coloriage des codes-sources :

- ⋆ Les valeurs littérales de strings sont toujours imprimées en vert.
- ⋆ Python se réserve les noms suivants (liste exhaustive) qui ne peuvent pas être redéfinis : `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`, `False`, `None`, `True`

  Dans ce cours, ces mots réservés (*keywords*) sont toujours imprimés en bleu.
- ⋆ Python connaît en outre un certain nombre d'instructions, de fonctions, de types... qui sont **toujours** disponibles, sans qu'on ait besoin d'importer un module. Le programmeur peut redéfinir ces noms pour autre chose (p. ex. des variables), mais alors il risque de perturber, voire de perdre totalement la fonctionnalité initialement associée au nom. Voici la liste de ces noms, qu'on trouve dans la documentation officielle de Python : `abs`, `all`, `any`, `ascii`, `bin`, `bool`, `breakpoint`, `bytearray`, `bytes`, `callable`, `chr`, `classmethod`, `compile`, `complex`, `delattr`, `dict`, `dir`, `divmod`, `enumerate`, `eval`, `exec`, `filter`, `float`, `format`, `frozenset`, `getattr`, `globals`, `hasattr`, `hash`, `help`, `hex`, `id`, `input`, `int`, `isinstance`, `issubclass`, `iter`, `len`, `list`, `locals`, `map`, `max`, `memoryview`, `min`, `next`, `object`, `oct`, `open`, `ord`, `pow`, `print`, `property`, `range`, `repr`, `reversed`, `round`, `set`, `setattr`, `slice`, `sorted`, `staticmethod`, `str`, `sum`, `super`, `tuple`, `type`, `vars`, `zip`, `__import__`

  Dans ce cours, ces mots sont toujours imprimés en pourpre.
- ⋆ Finalement, les identificateurs les plus importants se rapportant aux classes sont imprimés en orange.

Concernant le niveau de difficulté des différentes parties du cours, quelques sections et une partie des exercices sont marqués par des astérisques.

Les exercices **sans astérisque** doivent tous être traités, soit en classe, soit par l'élève comme devoir à domicile.

Un astérisque * indique que la section du cours traite une matière plus difficile ou complémentaire à la matière de base. Quant aux exercices, **un astérisque *** indique que l'enseignant est libre de les traiter en classe ou non et qu'il peut faire un choix parmi eux, sachant que le niveau de difficulté de ces exercices est le niveau à atteindre pour bien préparer les élèves à l'examen de fin d'études secondaires.

Deux astérisques ** indiquent que l'exercice en question est destiné aux élèves les plus motivés ; il ne sera probablement pas possible de le traiter en classe, faute de temps.

1.2 Connaissances

1.2.1 Rappel - Fonctions

Une fonction est une partie de code d'un programme que nous pouvons utiliser (appeler) plusieurs fois. Chaque fois que nous l'utilisons, nous pouvons lui transmettre d'autres valeurs comme arguments qu'elle utilisera durant son exécution. Le code de la fonction est placé plus haut dans le code-source que l'appel effectif de la fonction (c'est-à-dire au moment de l'appel de la fonction par l'interpréteur du programme sa définition doit déjà avoir été lue).

```
def <name_of_function>(param_1, ..., param_n):
    <instruction(s)>
```

Normalement, le corps d'une fonction contient une ou plusieurs occurrences de l'instruction **return**, qui permet de renvoyer la réponse déterminée par la fonction. Après l'instruction **return**, l'exécution de la fonction est terminée ; il est donc possible de quitter une fonction à différents endroits du corps.

```
def <name_of_function>(param_1, ..., param_n):
    <instruction(s)>
    return <answer>
```

Lorsqu'une fonction ne contient aucune instruction **return**, la valeur implicite **None** est renvoyée. Dans d'autres langages de programmation (mais pas dans Python), de telles fonctions sont parfois appelées *procédures*.

Une fonction qui ne reçoit aucun argument garde néanmoins ses parenthèses, qui indiquent qu'il s'agit bien d'une fonction.

```
def <name_of_function>():
    <instruction(s)>
    return <answer>
```

Pour appeler une fonction, on écrit par exemple :

```
my_result = <name_of_function>(arg_1,..., arg_n)
```

Il importe de distinguer les deux notations suivantes et la terminologie correspondante :

- ⋆ `f(x)` : la **fonction** f est appelée avec un argument x ;
- ⋆ `x.f()` : la **méthode** f est appliquée à l'object x ;
- ⋆ `f(x, y)` : la **fonction** f est appelée avec les arguments x et y ;
- ⋆ `x.f(y)` : la **méthode** f est appliquée à l'object x, en tenant compte de l'argument y.

1.2.2 Modules

Lorsqu'on désire réutiliser une fonction existante dans un autre programme, on a tendance à recopier son code-source. Cette façon de procéder est la plus simple mais, en même temps elle est moins efficace et de loin plus dangereuse :

- ⋆ lors d'une mise à jour, on doit modifier et rectifier toutes les occurrences (copies) de la fonction, avec le risque d'en oublier ;
- ⋆ le code-source est agrandi de façon inutile.

Pour résoudre ce problème on peut utiliser les *modules*, qui sont des fichiers dans lesquels on regroupe les différentes fonctions. Ces modules peuvent alors être importés dans un nouveau programme Python à l'aide de l'instruction **import**. À l'intérieur d'un module, une fonction est déclarée comme d'habitude.

1.2.3 Librairies (*packages*)

Pour établir une collection plus vaste d'éléments réutilisables, la gestion des modules peut devenir très embrouillée. Pour ce cas on peut utiliser une librairie (package), qui est un dossier complet pour gérer les modules. Ces dossiers peuvent aussi contenir d'autres dossiers (structure imbriquée). Le dossier principal de la librairie doit contenir un fichier vide nommé `__init__.py` pour communiquer à l'interpréteur Python qu'il s'agit d'une librairie et non pas d'un dossier ordinaire. Pour créer sa propre librairie il faut donc suivre ces étapes :

1. créer un dossier et lui donner un nom approprié ;
2. ajouter les modules ;
3. créer le fichier vide `__init__.py` dans le dossier.

1.3 Mise en pratique

1.3.1 Création d'un module

1. Créer un fichier `my_math_functions.py` et entrer le code-source de la factorielle :

```
def fact(n):
    x = 1
    for i in range(2, n + 1):
        x *= i
    return x
```

2. Créer à côté du fichier (module) `my_math_functions.py` un fichier `my_calculations.py` et taper le code-source suivant :

```
from my_math_functions import fact
n = int(input("Enter␣an␣integer:␣"))
m = fact(n)
print(m)
```

Attention ! Lors de l'exécution (compilation) du module un nouveau fichier avec l'extension .pyc est éventuellement créé (dans un dossier nommé `__pycache__`).

Si des modifications dans le module deviennent nécessaires, il faudra d'abord supprimer le fichier avec l'extension `.pyc`, s'il existe, à l'aide du file manager du système d'exploitation (soit le *File Explorer* sous Windows, soit le *Finder* sous macOS) pour que les changements dans le module soient pris en compte par l'interpréteur.

1.3.2 Création d'une librairie

Il s'agit de créer une librairie contenant des fonctions utilisées lors du cours de 2CB. Cette librairie personnelle pourra être utilisée dans nos projets futurs.

1. Créer un nouveau dossier et nommez ce dossier `functions_2B`
2. Copier dans ce dossier le fichier `my_math_functions.py` de l'exemple précédent et ajouter un nouveau fichier `my_string_functions.py`. Taper la fonction suivante dans le nouveau fichier :

```
def string_is_empty(s):
    if len(s) == 0:
        return True
    else:
        return False
```

Créer dans ce même dossier un fichier `__init__.py` qui reste vide.

3. À côté du dossier `functions_2B` créer un fichier `my_use_of_my_functions.py` et taper le code-source suivant :

```
from functions_2B.my_math_functions import fact
from functions_2B.my_string_functions import string_is_empty
n = int(input("Enter␣an␣integer:␣"))
m = fact(n)
print(m)
text = "Hello␣World"
print(string_is_empty(text))
```

1.4 Exercices

Exercice 1.1 Ajouter cinq fonctions mathématiques du cours de 2CB (voir les exercices correspondants) au fichier `my_math_functions.py`.

Exercice 1.2 Ajouter cinq fonctions de manipulation de strings du cours de 2CB au fichier `my_string_functions.py`.

Exercice 1.3 * Ajouter un nouveau fichier `my_other_functions.py` dans lequel on copiera cinq fonctions diverses élaborées au cours de 2CB.

Exercice 1.4 * Écrire un programme qui utilise au moins une fonction de chaque fichier des trois exercices précédents.

2 Initiation à l'algorithmique

2.1 Définition et conditions

En toute généralité, un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou d'obtenir un résultat.

Le mot algorithme vient du nom d'un mathématicien perse du IX^{e} siècle, Al-Khwârizmî. Le domaine qui étudie les algorithmes est appelé l'**algorithmique**.

En programmation, un algorithme est une suite d'instructions dans un programme, servant à résoudre un type de problèmes. Il est dit correct lorsque, pour toutes les entrées possibles, soit il se termine en renvoyant le bon résultat (c'est-à-dire qu'il résout le problème posé), soit il renvoie un message d'erreur lorsque les entrées n'étaient pas adaptées au problème.

Donald E. Knuth liste les cinq conditions suivantes comme étant les caractéristiques d'un algorithme :

1. *finitude* : un algorithme doit toujours se terminer après un nombre fini d'étapes ;
2. *définition précise* : chaque étape d'un algorithme doit être définie précisément ; les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas ;
3. *entrée(s)* : l'ensemble des données transférées comme arguments à l'algorithme doit être clairement spécifié. En mathématiques, on parlerait d'ensemble de départ d'une fonction ou application (ici le mot « application » est utilisé au sens mathématique, c'est-à-dire en allemand *Abbildung* et non pas *Anwendung*). Cet ensemble peut être vide, c'est-à-dire il existe des algorithmes ne recevant aucun argument lors de leur appel ;
4. *sortie(s)* : un algorithme doit produire et renvoyer un résultat. En mathématiques, on parlerait d'ensemble d'arrivée d'une fonction ou application. Cet ensemble ne peut donc pas être vide. De plus, si l'algorithme a reçu des arguments en entrée, le résultat doit en dépendre et l'algorithme ne peut pas simplement les ignorer ;
5. *efficacité* : Toutes les opérations élémentaires que l'algorithme doit accomplir successivement doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant un papier et un crayon. Les miracles n'existent généralement pas en informatique.

Voici un exemple d'algorithme très simple : l'algorithme d'Euclide permettant de calculer le pgcd (*plus grand commun diviseur*) de deux nombres naturels non nuls a et b donnés. Il a déjà été introduit au cours de 2CB.

Mathématiquement, on sait que

- ⋆ si b divise a, alors $\text{pgcd}(a, b) = b$;
- ⋆ sinon, soit r le reste (non nul alors) de la division euclidienne de a par b ; on a : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$.

En code Python :

```
def pgcd(a, b):
    while a % b != 0:
        (a, b) = (b, a % b)
    return b
```

Ou, de façon encore plus proche de la description mathématique précédente :

```
def pgcd(a, b):
    if a % b == 0:
        return b
    return pgcd(b, a % b)
```

Ou, encore plus élégant, avec l'utilisation du fameux opérateur ternaire qui combine toute une structure alternative dans une seule expression :

```
def pgcd(a, b):
    return b if a % b == 0 else pgcd(b, a % b)
```

Cet algorithme vérifie les cinq conditions énoncées ci-dessus :

1. Comme le reste d'une division euclidienne est toujours strictement plus petit que le diviseur, on a $r < b$, et les dividendes et diviseurs intervenant successivement dans les calculs deviennent de plus en plus petits ; au plus tard lorsque $r = 1$, le reste de la division euclidienne suivante s'annule et le procédé se termine. La *finitude* est donc assurée.
2. La *définition* précise découle de la description mathématique qui est exacte (facile à démontrer, cf. cours 6C de mathématiques).
3. Les *entrées* sont clairement définies : deux nombres naturels non nuls.
4. La *sortie* produite est le pgcd des deux nombres entrés et en dépend bien sûr.
5. Tous les calculs (essentiellement des divisions euclidiennes) peuvent être réalisés de manière *efficace*.

2.2 Mesures d'efficacité d'un algorithme

Même s'ils répondent tous à la 5^{e} condition, il existe bien sûr des algorithmes qui sont plus rapides ou consomment moins de mémoire que d'autres. On peut ainsi quantifier l'efficacité suivant différents aspects, par exemple :

- ⋆ la durée d'exécution de l'algorithme ;
- ⋆ la quantité de mémoire nécessaire (penser p. ex. aux variables locales !) ;
- ⋆ le nombre de lectures ou d'écritures de données dans la mémoire : pour certains types de mémoire (p. ex. mémoire flash des clés USB), un nombre élevé d'écritures risque d'user, voire d'abîmer le matériel utilisé ;
- ⋆ le nombre d'accès au réseau intranet ou Internet (p. ex. lorsque les données sont stockées dans une Cloud) et la quantité de données y transmises ; en effet, ces transferts risquent de se faire lentement ou d'être payants, en fonction de l'infrastructure disponible et de l'abonnement d'Internet conclu.

Dans la suite, nous nous intéresserons principalement au temps d'exécution des algorithmes étudiés. Nous parlerons alors de **complexité temporelle** de ces algorithmes.

Pour beaucoup d'algorithmes, le temps d'exécution dépend de la taille des arguments fournis comme entrée. Pour décrire avec rigueur mathématique cette dépendance, la *notation de Landau* (ou notation O) est généralement utilisée. La section suivante contient les définitions et notations principales, ainsi que quelques exemples dont la plupart réapparaîtront dans ce cours à un moment donné. Pour les définitions des limites y utilisées, revoir le cours de mathématiques II de 2CB, 1er trimestre (limites).

2.3 Notation de Landau (notation O) *

Les notations suivantes sont fréquemment utilisées en analyse mathématique et en informatique (surtout en algorithmique).

Soit $a \in \mathbb{R} \cup \{-\infty; +\infty\}$, et soient f et g deux fonctions définies dans un voisinage V_a de a. On veut comparer le comportement asymptotique de f et g, si $x \to a$.

⋆ On note $\boxed{f(x) = o(g(x))}$, lorsque $\lim_{x\to a} \left|\frac{f(x)}{g(x)}\right| = 0$. Cela revient à dire que

$$\forall\, C > 0 : \exists\, V_a : \forall\, x \in V_a : |f(x)| \leq C \cdot |g(x)|$$

et on dit que f est négligeable par rapport à g.

⋆ On note $\boxed{f(x) = \omega(g(x))}$, lorsque $\lim_{x\to a} \left|\frac{f(x)}{g(x)}\right| = +\infty$. Cela revient à dire que $g(x) = o(f(x))$, et on dit que f domine g.

⋆ On note $\boxed{f(x) = O(g(x))}$, lorsque

$$\exists\, C > 0 : \exists\, V_a : \forall\, x \in V_a : |f(x)| \leq C \cdot |g(x)|\,.$$

On dit que f est majorée par g à un facteur près.

En particulier, $f = O(1)$ signifie que f est bornée.

⋆ On note $\boxed{f(x) = \Omega(g(x))}$, lorsque $g(x) = O(f(x))$.

C'est la définition moderne de Ω. On dit que f est minorée par g à un facteur près.

⋆ On note $\boxed{f(x) = \Theta(g(x))}$, lorsque $f(x) = O(g(x))$ et $g(x) = O(f(x))$.

Cela est équivalent à $f(x) = O(g(x))$ et $f(x) = \Omega(g(x))$.

On dit intuitivement que f est dominée et soumise à g.

Les symboles $o, \omega, O, \Omega, \Theta$ jouent en quelque sorte le rôle de $<, >, \leq, \geq, =$ au niveau des comportements asymptotiques.

Voici, en guise d'exemples, quelques durées d'exécution d'algorithmes classiques :

⋆ Accès à un élément d'un tableau à n éléments de même taille en mémoire : $\Theta(1)$.

⋆ Recherche dichotomique d'une clé dans un tableau à n éléments triés : $O(\ln n)$.

⋆ Test de primalité simple (p. ex. essayer tous les diviseurs impairs $\leq \sqrt{n}$) : $O(\sqrt{n}\,)$.

⋆ Recherche séquentielle d'une clé dans un tableau à n éléments non triés : $O(n)$.

⋆ Tri rapide (*Quicksort*) d'un tableau approprié à n éléments : $O(n \ln n)$.

⋆ Tri par insertion d'un tableau à n éléments : $O(n^2)$ et $\Omega(n)$.

⋆ Tri par sélection d'un tableau à n éléments : $\Theta(n^2)$.

⋆ Détermination du chemin le plus court : $O(n!)$ malheureusement.

2.4 Récursivité

En toute généralité, la **récursivité** est une démarche qui fait référence à l'objet même de la démarche à un moment du processus. En d'autres termes, c'est la propriété de pouvoir appliquer une même règle plusieurs fois à elle-même.

En informatique et en logique, une fonction/méthode ou plus généralement un algorithme qui s'appelle lui-même est dit **récursif**.

Nous venons de rencontrer un algorithme récursif, à savoir la 2$^{\text{e}}$ version de l'algorithme d'Euclide :

```
def pgcd(a, b):
    if a % b == 0:
        return b
    return pgcd(b, a % b)
```

Il s'agit d'un algorithme récursif, car à la 2e ligne la fonction pgcd s'appelle elle-même. Par contre, l'autre version

```
def pgcd(a, b):
    while a % b != 0:
        (a, b) = (b, a % b)
    return b
```

n'est pas récursive, mais dite **itérative** : la boucle `while` est exécutée (réitérée) autant de fois que nécessaire pour obtenir le bon résultat.

Lors d'appels récursifs, il faut faire attention à ne pas générer un nombre infini d'appels. Il faut donc prévoir une **condition d'arrêt** afin de briser le cercle vicieux. Dans l'algorithme d'Euclide, la condition « si b divise a, alors le pgcd cherché est b » permet de quitter la boucle, et nous avons déjà vu plus haut que cette condition sera forcément remplie à partir d'un certain moment.

Voici un autre exemple très simple : le calcul de la factorielle $n!$ d'un nombre naturel n. Comme d'habitude, on définit encore : $0! = 1! = 1$

Version itérative :

```
def factorielle(n):
    p = 1
    for i in range(2, n + 1):
        p *= i
    return p
```

La version récursive utilise la propriété mathématique $n! = n \cdot (n-1)!$:

```
def factorielle(n):
    if n > 1:
        return n * factorielle(n - 1)
    else:
        return 1
```

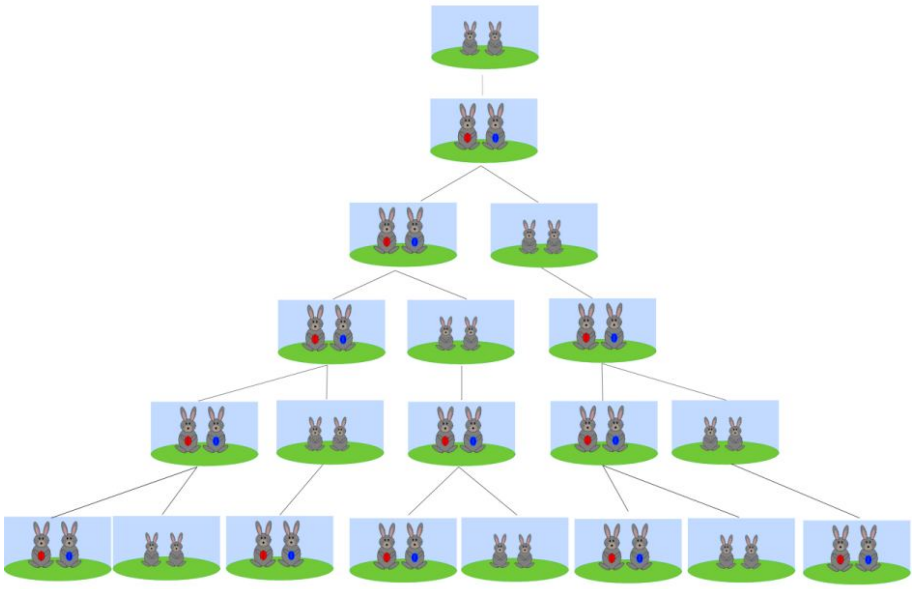
La condition d'arrêt utilisée ici est : $n! = 1$ dès que $n \leq 1$.

2.5 Exercices

Exercice 2.1 Considérons la suite des nombres de **Fibonacci** : c'est la suite de nombres naturels dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence par les termes 0 et 1 et ses premiers termes sont donc :

$F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, etc.

Elle doit son nom à Leonardo Fibonacci qui, dans un problème récréatif publié en 1202, décrit la croissance d'une population de lapins : « Un homme met un couple de lapins dans un lieu isolé. Combien de couples obtient-on après n mois si chaque couple engendre tous les mois un nouveau couple à compter du 3e mois de son existence ? »



Mathématiquement, on peut définir la suite ainsi :

$$F_0 = 0, \quad F_1 = 1, \quad \text{et} \quad F_n = F_{n-1} + F_{n-2} \text{ pour tout } n \geq 2$$

1. Écrire une fonction qui calcule F_n de façon itérative.
2. Écrire une fonction qui calcule F_n de façon récursive.
3. ***** Étudier la complexité temporelle de la fonction récursive. Si l'algorithme conçu s'avère très lent, essayer d'y remédier avec une liste-cache contenant les nombres de Fibonacci déjà calculés antérieurement.
4. Écrire une fonction qui calcule directement F_n, compte tenu de la formule suivante :

$$F_n = \frac{1}{\sqrt{5}} \cdot (\varphi^n - \psi^n) \qquad \text{avec} \quad \varphi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \psi = -\frac{1}{\varphi} = \frac{1-\sqrt{5}}{2}$$

Exercice 2.2 Soit la suite de Collatz (ou suite de Syracuse) définie par :

- ⋆ u_0 est un nombre naturel non nul quelconque,
- ⋆ pour tout n naturel :

$$u_{n+1} = \begin{cases} \frac{1}{2} \cdot u_n & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La *conjecture de Collatz* est l'hypothèse mathématique selon laquelle la suite de Collatz de n'importe quel naturel non nul u_0 atteint 1. En dépit de la simplicité de son énoncé, cette conjecture défie depuis de nombreuses années les mathématiciens.

1. Écrire une fonction qui à partir de u_0 calcule u_{100} de façon récursive.
2. Écrire une fonction qui à partir de u_0 calcule u_{100} de façon itérative.
3. Écrire une fonction qui à partir de u_0 détermine l'indice n le plus petit pour lequel $u_n = 1$.

4. Tester la fonction précédente pour toutes les valeurs de $u_0 \leq N$, pour une borne N raisonnablement élevée. Jusqu'à quelle valeur approximative de N peut-on monter si on veut que le programme s'arrête après une nuit (disons 8 heures) de calculs ?

Exercice 2.3 * Soit la fonction d'Ackermann définie par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

Écrire une fonction qui calcule de façon « efficace » $A(m,n)$ pour $m \geq 0$ et $n \geq 0$ donnés. Attention à la complexité temporelle de l'algorithme ! Jusqu'à quelle valeur de m peut-on monter raisonnablement ?

Exercice 2.4 – Algorithmes de recherche séquentielle et dichotomique

Soit une liste L contenant un certain nombre de strings, et c un string, appelé « clé » (*key*) dans la suite.

1. Écrire une fonction itérative qui vérifie (sans tricher, c'est-à-dire sans utiliser des fonctions ou méthodes Python préfabriquées !) si la liste L contient la clé c. Si c'est le cas, la fonction renvoie l'indice de la 1re occurrence de c dans L. Sinon, la fonction renvoie la valeur -1 par convention.

2. Écrire une fonction récursive qui réalise la même opération.

3. Modifier les deux fonctions précédentes de la manière suivante : elles ne renvoient plus un seul nombre (indice), mais la liste de tous les indices des éléments de L égaux à c. Lorsque L ne contient pas c, la liste renvoyée est la liste vide.

4. Supposer maintenant que la liste L est triée alphabétiquement. Écrire une fonction itérative, plus efficace (du point de vue complexité temporelle) que celle de la partie 1., qui effectue une **recherche dichotomique** de c dans L.

   Le principe est le suivant : comparer c avec la valeur au milieu de la liste L ; si les valeurs sont égales, la tâche est accomplie, sinon on recommence la recherche dans la bonne moitié de la liste.

5. Écrire une fonction récursive, plus efficace (du point de vue complexité temporelle) que celle de la partie 2., qui effectue une recherche dichotomique de c dans la liste L supposée triée.

Exercice 2.5 – Triangle de Pascal *

Écrire un programme qui calcule les nombres du **triangle de Pascal**.

Rappel : soient n le numéro de la ligne et k le numéro de la colonne du triangle, avec $0 \leq k \leq n$. Alors

$$p_{n,k} = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n \\ p_{n-1,k-1} + p_{n-1,k} & \text{si } 0 < k < n \end{cases}$$

Par convention, on définit $p_{n,k} = 0$ à l'extérieur du triangle, c'est-à-dire si $n < 0$ ou $k \notin [0, n]$.

1. Écrire une fonction `pascal_r` qui calcule $p_{n,k}$ de façon récursive, en utilisant la formule de récurrence

$$p_{n,k} = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n \\ p_{n-1,k-1} + p_{n-1,k} & \text{si } 0 < k < n \end{cases}$$

2. Écrire une fonction `pascal_rc` qui calcule $p_{n,k}$ de façon récursive en utilisant un cache contenant toutes les valeurs calculées précédemment. À partir de $n \geq 20$, cette fonction devrait être visiblement plus rapide que la version sans cache.

3. Écrire une fonction `pascal_i` qui calcule $p_{n,k}$ de façon itérative (sans appel récursif).

4. Écrire une fonction `pascal_f` qui calcule $p_{n,k}$ de façon directe, sachant que

$$p_{n,k} = \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

5. Le programme principal demande à l'utilisateur d'entrer un nombre naturel $N > 0$ et affiche les N premières lignes du triangle de Pascal. Il calcule et affiche ensuite la somme de chacune des dix lignes suivantes du triangle. Finalement il demande à l'utilisateur d'entrer deux nombres naturels a et $M > 0$ et affiche les M premières lignes du triangle sous la forme suivante : dans chaque ligne, les nombres qui sont multiples de a sont représentés par une espace et ceux qui ne sont pas multiples de a par le symbole `*`.

Exemple d'exécution du programme :

```
Entrez N (le nombre de lignes à afficher) : 20
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1
1 17 136 680 2380 6188 12376 19448 24310 24310 19448 12376 6188 2380 680 136 17 1
1 18 153 816 3060 8568 18564 31824 43758 48620 43758 31824 18564 8568 3060 816 153 18 1
1 19 171 969 3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628 3876 969 171 19 1
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912

Entrez a : 3
Entrez M (le nombre de lignes à afficher) : 27
*
**
***
*  *
** **
******
*  *  *
```

```
** ** **
*********
*       *
**      **
***     ***
*  *    *  *
** **   ** **
******  ******
*  *  *  *  *  *
** ** ** ** ** **
******************
*       *       *
**      **      **
***     ***     ***
*  *    *  *    *  *
** **   ** **   ** **
******  ******  ******
*  *  *  *  *  *  *  *  *
** ** ** ** ** ** ** ** **
**************************
```

2.6 Application : algorithmes de tri

2.6.1 Introduction

Dans ce sous-chapitre, nous allons étudier quelques algorithmes de tri (*sorting algorithms*), notamment l'algorithme non trivial du « tri rapide » (*Quicksort*).

Considérons une liste dont tous les éléments peuvent être comparés les uns aux autres, p. ex. une liste de nombres ou une liste de strings. La liste est dite **triée**, lorsque ses éléments sont rangés dans un ordre déterminé, le plus souvent croissant ou décroissant. Dans tous les cas, il faut bien spécifier

- ⋆ le **type** de tri : tri *numérique* pour les nombres, tri *lexicographique* (alphabétique) pour les strings, etc. ;
- ⋆ le **sens** du tri : ordre *croissant* (pour les strings : A–Z), ordre *décroissant* (Z–A), etc.

Dans la suite du chapitre, nous considérerons le plus souvent des tris numériques croissants, par pure convention et sans restreindre l'applicabilité des algorithmes aux autres cas de figure.

Voici une esquisse de programme qui illustre une méthode de tri très simple, à savoir le **tri par sélection** (*Selection Sort*) : il consiste à chercher le plus petit élément de la liste et le placer en début de liste, de chercher ensuite le plus petit élément parmi ceux qui restent pour le placer à la 2^{e} position, et ainsi de suite jusqu'à l'avant-dernière position. Le dernier élément de la liste est alors forcément le plus grand et la liste est complètement triée.

Pour générer rapidement des listes de nombres entiers, nous utiliserons une fonction aléatoire :

```
from random import randrange

def make_random_array(n):
    a = [randrange(10 * n) for _ in range(n)]
    return a
```

Les éléments d'une liste de taille n sont donc des nombres compris entre 0 et $10n - 1$. Ainsi il arrivera qu'une liste contienne plusieurs fois la même valeur, mais pas trop souvent.

À noter que le code plus « simple »

```
a = [randrange(10 * n)] * n
```

ne donne pas le résultat attendu : on obtiendrait n fois la même valeur aléatoire.

Pour afficher une liste, on utilisera la fonction triviale

```
def print_array(a):
    print(list(a))
```

Plus loin dans le chapitre, lorsqu'on étudiera la complexité temporelle à l'aide de listes de tailles très élevées, nous remplacerons cette fonction par

```
def print_array(a):
    print(list(a[:100]))
```

Ainsi nous limiterons l'affichage au 100 premiers éléments de la liste, tout en espérant que si le début de la liste apparaît correctement trié, ce sera le cas pour toute la liste.

Voici la fonction effectuant le tri par sélection :

```
def selection_sort(a):
    for i in range(len(a) - 1):
        p = i
        for j in range(i + 1, len(a)):
            if a[j] < a[p]:
                p = j
        if p > i:
            (a[i], a[p]) = (a[p], a[i])
```

Le compteur i parcourt la liste du début jusqu'à l'avant-dernier élément. Dans la boucle de compteur j, on recherche l'élément le plus petit parmi ceux qui n'ont pas déjà été triés. L'emplacement de l'élément le plus petit est sauvegardé dans la variable p. Finalement, lorsque l'élément le plus petit ne se trouve pas déjà au bon endroit ($p > i$), il y est copié et en même temps la valeur qui s'y trouvait est transférée à l'emplacement devenu vacant. Ainsi, la dernière ligne de ce code effectue un échange (*swap*) de deux éléments.

La fonction ne renvoie pas de résultat explicite à l'aide de **return**. Elle travaille sur la liste reçue comme argument et modifie donc la liste originale (passage par référence des structures composées qui sont *mutables* !).

Voici la partie principale du programme, permettant de tester l'algorithme :

```
n = int(input("Size: "))
a = make_random_array(n)
print_array(a)
selection_sort(a)
print_array(a)
```

L'algorithme du tri par sélection est facile à comprendre et à mettre en œuvre, mais il n'est pas rapide. Pour s'en convaincre, il suffit de compter p. ex. le nombre de comparaisons d'éléments de la liste effectuées à la 5e ligne de la fonction : pour i allant de 0 jusqu'à $n-2$, on compte

$$(n-1)+(n-2)+\cdots+2+1=\sum_{i=1}^{n-1} i=\frac{n\cdot(n-1)}{2} \quad \text{comparaisons.}$$

Ce nombre est indépendant de la liste reçue comme argument. Une liste déjà triée engendrera le même nombre de comparaisons, rien que pour constater qu'aucun élément de la liste ne doit être déplacé. Ainsi, la complexité temporelle du tri par sélection est $\Theta(n^2)$.

2.6.2 Best case, worst case, average case *

Lorsqu'on étudie la complexité d'un algorithme, on s'intéresse souvent à différentes configurations initiales des données. On distingue notamment le *best case* (cas le plus favorable), le *worst case* (cas le plus défavorable) et l'*average case* (cas d'une configuration typique, aléatoire).

Concernant le tri par sélection de la sous-section précédente, nous avons déjà vu que le nombre de comparaisons est indépendant des données initiales ; en effet, pour trier n éléments, on effectuera toujours $\frac{n\cdot(n-1)}{2}$ comparaisons.

Par contre, le nombre de *swap* (échanges des données, voir la dernière ligne du code-source de `selection_sort`) dépend de la configuration initiale des données. On identifie :

- ⋆ Le *best case* : la liste est déjà triée dans le bon sens, avant l'appel de l'algorithme de tri ; il n'y a rien à faire, donc aucun échange n'aura lieu.
- ⋆ Le *worst case* : la liste initiale est triée presque complètement, à l'exception de l'élément maximal qui se trouve du mauvais côté (à la première position). Exemple : on veut trier en ordre croissant les cinq nombres $5, 1, 2, 3, 4$. S'il y a n éléments au total, l'élément maximal sera échangé $n-1$ fois avec l'élément suivant de la liste, ce qui représente le cas le plus défavorable : l'échange a lieu pour toutes les étapes de la boucle sur i.
- ⋆ L'*average case* : si l'on dispose d'une liste avec n valeurs aléatoires, on peut admettre que l'élément le plus petit d'une sous-liste à k éléments se trouve déjà au bon endroit (au début de cette sous-liste) avec une probabilité égale à $\frac{1}{k}$. Un échange sera donc nécessaire avec une probabilité de $1-\frac{1}{k}$. Comme dans l'algorithme de tri par sélection, k prend successivement les valeurs $n, n-1, \ldots, 2$, on obtient comme nombre moyen d'échanges la valeur
$$\sum_{k=2}^{n}\left(1-\frac{1}{k}\right) = n-1-\sum_{k=2}^{n}\frac{1}{k} = n-\sum_{k=1}^{n}\frac{1}{k}$$
Or, les nombres $H_n = \sum_{k=1}^{n}\frac{1}{k}$, appelés *nombres harmoniques* en mathématiques, croissent lentement, à la même vitesse que $\ln n$ (intuitivement). On peut dire que dans l'expression $n - H_n$, le terme H_n est négligeable par rapport à n ; ainsi l'*average case* se trouve très rapproché du *worst case*, pour le tri par sélection.

2.6.3 Présentation du tri rapide (*Quicksort*)

Le tri rapide est un algorithme de tri inventé par Tony Hoare en 1961 et fondé sur la méthode de conception « divide et impera » (*diviser pour régner*, c'est-à-dire il s'agit d'un algorithme récursif).

La méthode consiste à placer **un seul** élément de la liste (appelé *pivot*) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche (mais dans n'importe quel ordre) et que tous ceux qui sont supérieurs au pivot soient à sa droite (dans n'importe quel ordre). Cette opération s'appelle le *partitionnement*.

Pour chacune des sous-listes restant à gauche et à droite du pivot, on appelle la même fonction, c'est-à-dire on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que la liste initiale soit triée.

Concrètement, pour partitionner une sous-liste :

- ⋆ on choisit un pivot de façon arbitraire, par exemple le dernier élément de la liste (ou de la sous-liste actuellement considérée) ;
- ⋆ tous les éléments inférieurs au pivot sont placés à gauche des éléments supérieurs au pivot ; des éléments trouvés du mauvais côté respectif sont échangés ;

⋆ le pivot est déplacé au bon endroit après les réarrangements précédents.

Il faut bien comprendre qu'à ce moment, le pivot se trouve **définitivement** au bon endroit dans la liste, mais que les sous-listes à sa gauche et à sa droite doivent encore être réordonnées, chacune indépendamment de l'autre.

Voici le code Python, correspondant à la description précédente :

```
def partition(a, g, d):
    p = a[d]
    (i, j) = (g, d - 1)
    while True:
        while i <= j and a[i] < p:   # a[i] on the left side?
            i += 1
        while i <= j and a[j] > p:   # a[j] on the right side?
            j -= 1
        if i <= j:
            (a[i], a[j]) = (a[j], a[i])   # put a[i] and a[j] on the correct sides
            i += 1
            j -= 1
        else:   # i et j crossed each other => done
            (a[i], a[d]) = (p, a[i])   # put the pivot in the correct position
            return i   # index of pivot
def quicksort(a, g = 0, d = ""):
    if d == "":
        d = len(a) - 1
    if g >= d:   # let's stop
        return
    p = partition(a, g, d)
    quicksort(a, g, p - 1)   # sublist to the left   of the pivot
    quicksort(a, p + 1, d)   # sublist to the right of the pivot
```

L'appel du tri rapide se fait par

```
quicksort(a)
```

Ainsi les indices g et d, pointant toujours vers les extrémités de la (sous-)liste à considérer, sont générés par défaut : $g = 0$ et $d = n - 1$, et toute la liste passée comme argument est traitée.

Comme pivot, nous avons choisi simplement le dernier élément de la (sous-)liste. Nous verrons plus loin comment améliorer ce choix.

À noter que le bon emplacement du pivot dans la fonction `partition` est donné par l'indice i (et non pas j ou un autre indice) !

2.6.4 Exercices

Exercice 2.6 Implémenter le tri par sélection et le tri rapide. Vérifier à l'aide de tests que les algorithmes fonctionnent correctement, c'est-à-dire que les listes obtenues sont bien triées.

Exercice 2.7 Tester la vitesse d'exécution des deux algorithmes. Jusqu'à quelle valeur approximative de n peut-on monter pour le tri par sélection, sans que la durée d'exécution ne dépasse une minute ? Une heure ? Une journée entière ? Quelle est la durée d'exécution du tri rapide pour ces mêmes valeurs de n ?

Indication : utiliser le fait que la complexité temporelle du tri par sélection est $\Theta(n^2)$; ainsi on pourra estimer les seuils sans faire fonctionner le tri par sélection pendant une heure, voire une journée entière.

Exercice 2.8 Que se passe-t-il lorsqu'on applique l'algorithme du tri rapide, tel que décrit ci-dessus, à une liste déjà entièrement triée ? Quelle est alors la complexité temporelle de l'algorithme ? Expliquer de façon théorique et vérifier à l'aide du programme.

Exercice 2.9 – Choix d'un bon pivot

Plutôt que de choisir toujours le dernier élément de la liste comme pivot, on préférera la démarche suivante : on compare le 1er élément, l'élément du milieu et le dernier élément de la liste, et on retiendra comme pivot la médiane des trois valeurs. Lorsque le pivot retenu ne se trouve pas déjà à la dernière position dans la liste, il faut échanger le pivot et la valeur qui s'y trouve.

Adapter ainsi l'algorithme du tri rapide (c'est-à-dire la fonction `partition`).

Exercice 2.10 Réimplémenter le tri par sélection, sous forme récursive.

2.6.5 Le tri par insertion *

Dans cette sous-section facultative, nous présentons un autre algorithme de tri, qui dans certains cas favorables est aussi rapide, parfois même plus rapide que *Quicksort*. C'est le tri par insertion (*Insertion Sort*).

Dans cet algorithme, on parcourt la liste à trier du début à la fin. Au moment où on considère l'i-ième élément, noté X, les éléments qui le précèdent sont déjà triés, par hypothèse. L'objectif de chaque étape est d'insérer X à la bonne place parmi ceux qui précèdent. Il faut pour cela trouver le bon emplacement où X doit être inséré en le comparant aux autres, puis décaler les éléments plus grands que X afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont effectuées en même temps, c'est-à-dire on fait remonter les éléments au fur et à mesure jusqu'à rencontrer un élément plus petit que X ou se retrouver en début de liste.

Voici le code Python correspondant :

```
def insertion_sort(a):
    for i in range(1, len(a)):
        (x, j) = (a[i], i)
        while j > 0 and a[j - 1] > x:
            a[j] = a[j - 1]
            j = j - 1
        if j < i:
            a[j] = x
```

L'appel se fait bien sûr par

```
insertion_sort(a)
```

Comme il est souvent utile de mesurer les temps d'exécution de différents algorithmes, nous illustrons encore le chronométrage du tri par insertion. Il faudra d'abord importer la fonction `process_time` qui permet de mesurer le temps écoulé en secondes :

```
from time import process_time
```

Ensuite, l'appel du tri peut se faire ainsi :

```
t1 = process_time()   # current time
insertion_sort(a)
t2 = process_time()   # current time, so t2 - t1 is the elapsed time
print("Time used:", t2 - t1)
```

2.6.6 Exercices supplémentaires *

Exercice 2.11 * Implémenter le tri par insertion.

Vérifier à l'aide de tests qu'il fonctionne correctement, c'est-à-dire que les listes obtenues sont bien triées.

Étudier le *best case*, le *worst case* et l'*average case* du tri par insertion. Compter pour cela le nombre de comparaisons d'éléments de la liste (noté n_c) et le nombre d'affectations de données de la liste (noté n_a). Comme ces deux types d'opérations nécessitent souvent un temps d'exécution similaire, on peut simplement additionner les deux nombres et étudier $N = n_c + n_a$ pour différentes configurations initiales.

Exercice 2.12 * Implémenter le tri par insertion sous forme récursive.

Exercice 2.13 * Comparer les vitesses d'exécution du tri par sélection et du tri par insertion :

- ⋆ pour des listes générées aléatoirement ;
- ⋆ pour des listes déjà triées (ou presque triées).

Conclure et constater notamment que la complexité temporelle du tri par insertion est $O(n^2)$ et $\Omega(n)$.

Exercice 2.14 * Compléter le code des algorithmes de tri implémentés jusqu'à présent, en y rajoutant des compteurs permettant de comptabiliser :

- ⋆ le nombre total d'affectations d'éléments de la liste qui sont faites lors d'un tri complet,
- ⋆ le nombre total de comparaisons d'éléments de la liste qui sont faites lors d'un tri complet.

Vérifier à l'aide d'exemples que les valeurs des compteurs correspondent bien aux prévisions mathématiques, concernant la complexité temporelle des algorithmes étudiés.

**Exercice 2.15 ** On peut s'attendre à ce que l'algorithme du tri rapide ne fonctionne plus très efficacement sur des sous-listes de taille très réduite, parce que les appels récursifs et la gestion des indices g, d, i, j prennent un temps relatif de plus en plus important dans des sous-listes petites. L'on pourra être tenté de limiter le travail de *Quicksort* à des (sous-)listes de taille suffisamment élevée, disons $d - g \geq 10$, et ne pas ordonner les sous-listes de taille < 10. Après les opérations de ce pseudo-tri, le tri par insertion permettra de trier définitivement la liste. Comme la liste fournie au tri par insertion est déjà presque triée, l'algorithme devrait être ultra-rapide.

Implémenter les idées de l'alinéa précédent, et comparer les vitesses d'exécution du tri rapide original et du tri rapide modifié.

**Exercice 2.16 ** Comme le tri de données est une opération élémentaire et très fréquente, Python fournit une fonction préfabriquée de tri, basée notamment sur… le tri rapide ! En effet, la fonction `sorted(a)` renvoie la liste triée des éléments de la liste a, sans modifier a, tandis que la méthode `a.sort()` trie la liste a *in situ*.

Comparer les vitesses d'exécution du tri rapide implémenté « manuellement » dans les exercices précédents et du tri rapide mis à disposition par Python.

Remarque : Il est tout à fait normal que les routines préfabriquées de Python soient plus efficaces que le code écrit manuellement, qui devra constamment être interprété et exécuté ligne après ligne.

3 Programmation orientée objet (POO)

3.1 Introduction

Jusqu'à présent nous avons vu des programmes composés de données et d'algorithmes qui les traitent. En appliquant la méthode top-down, on a décomposé les problèmes en sous-problèmes jusqu'à arriver à des instructions et des fonctions simples. Ces sous-programmes ont travaillé avec les données ou encore des structures de données.

Si cette approche est acceptable pour de petits programmes, elle s'avère nettement moins appropriée, voire coûteuse et dangereuse pour des logiciels de plus grande envergure. Voici les inconvénients majeurs :

- ⋆ si on veut agrandir ou modifier les structures de données, on est amené à changer toutes les parties de code qui les traitent (avec le risque d'injecter des bugs dans une partie de code correcte jusque-là) ;
- ⋆ plus l'envergure est grande, plus la durée, la complexité et les coûts de développement augmentent de façon démesurée ;
- ⋆ les données traitées ne sont pas du tout (ou très peu) protégées, ce qui représente un sérieux problème de sécurité et de confidentialité.

La programmation fonctionnelle (ou procédurale) telle qu'on l'a vue jusqu'à présent est axée sur la logique et les techniques de manipulation des données. Dans un monde connecté où la vie privée (et donc les données) deviennent de plus en plus compromises, il y a lieu de changer de perspective et de mettre en valeur les données elles-mêmes. La **programmation orientée objet** essaye de réaliser ce concept, ce qui nous amène à changer notre approche.

Prenons comme exemple un menuisier qui produit des articles de décoration comme des boules, des cônes, des parallélépipèdes et des pyramides. Afin de gérer son entreprise il se sert d'un programme pour les différentes facettes :

- ⋆ la gestion du stock des matières premières (bois, peintures, cires, ...),
- ⋆ les coûts (prix des matières, main-d'œuvre, utilisation des machines, ...)
- ⋆ la facturation,
- ⋆ ...

Si à un certain moment, il décide d'élargir l'éventail des articles de décoration, alors il sera amené à modifier tout son programme de gestion afin de pouvoir en tenir compte, puisque chaque article est différent (volume, superficie, main-d'œuvre requise, ...). Ainsi, le sous-programme pour calculer le prix de revient doit être modifié pour calculer le volume et la superficie du nouvel article. Il en est de même pour tous les autres sous-programmes.

Dans la programmation orientée objet, l'approche ne se base pas sur les procédures (comme scier, cirer, calculer le prix de revient, ...) mais sur les **objets**, c'est-à-dire les articles (la boule, la pyramide, ...). En modélisant chaque type d'article avec ses caractéristiques (rayon, longueur, ...) et ses actions (calculer le volume, calculer la superficie, calculer le prix de revient, ...) on arrive à modulariser en même temps le programme, mais d'une manière fondamentalement différente de l'approche décrite au début de la section. En mettant l'accent sur les articles, on peut très facilement ajouter un type d'article : il suffit de le modéliser avec ses caractéristiques et ses actions. Le reste du programme reste inchangé !

3.2 Définition

Selon Wikipédia, la programmation orientée objet (POO) est un paradigme de programmation informatique qui consiste en :

- ⋆ la définition et l'interaction de briques logicielles appelées **objets**, un objet représentant un concept, une idée ou une entité du monde physique, comme une voiture, une personne ou encore un article de décoration. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs ;
- ⋆ la représentation de ces objets et de leurs relations. L'interaction entre les objets via leurs relations permet de concevoir et de réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes ;
- ⋆ la modélisation permettant de transcrire les éléments du réel sous forme virtuelle. Elle a une importance majeure et nécessaire pour la POO.

3.2.1 Exemple

Reprenons l'exemple de notre menuisier et modélisons deux types d'articles de décoration, à savoir la pyramide et la boule :

`Pyramid`	`Sphere`	*nom de la classe*

`width : float`
`height : float`
`wood_price : float`
`working_time : float`
`tool_time : float`
	`radius : float`
	`wood_price : float`
	`working_time : float`
	`tool_time : float`
		attributs (propriétés)

`calculate_volume() : float`
`calculate_surface() : float`
`calculate_price() : int`
	`calculate_volume() : float`
	`calculate_surface() : float`
	`calculate_price() : int`
	`set_radius(new_radius:float)`
		méthodes (actions, calculs)

`Pyramid` est une **classe** (donc la description d'une chose de la vie réelle), qui dispose de différents **attributs** (donc ses propriétés) et d'un certain nombre de **méthodes** (donc des actions ou des comportements). Il en est de même pour la classe `Sphere`.

Afin de formaliser les descriptions de classe, on se sert souvent des diagrammes UML (*Unified Modeling Language*), qui regroupent dans un encadré le nom de la classe, les attributs avec leur type et les méthodes avec leur(s) paramètre(s) et leur type de valeur de retour (comme illustré ci-dessus).

Imaginons maintenant un client qui commande deux pyramides (p_1, p_2) et une sphère (s_1). Le menuisier créera les **instances** p_1, p_2 et s_1 (donc les objets informatiques représentant chacun un tel article) et encodera les valeurs pour les attributs respectifs. Ensuite il pourra se servir des méthodes des objets pour calculer le prix total :

```
total = p1.calculate_price() + p2.calculate_price() + s1.calculate_price()
```

Les méthodes `calculate_price()` se serviront des autres méthodes et attributs de ***leur*** instance pour calculer aires et volumes, sachant bien que la méthode `p1.calculate_volume()` s'appuyera sur la formule $\frac{\texttt{width}^2 \cdot \texttt{height}}{3}$, alors que la méthode `s1.calculate_volume()` appliquera la formule $\frac{4}{3}\pi \cdot \texttt{radius}^3$.

Il est important de noter ici que les méthodes de même nom, mais appartenant à des instances de différentes classes calculent chacune la **même chose** (le prix), mais d'une **manière différente**. Ceci est particulièrement intéressant lorsque le nombre d'instances augmente et qu'elles sont gérées dans une liste :

```
total = 0
for article in production_list:
    total += article.calculate_price()
```

À ce niveau on n'a plus besoin de se soucier de la nature des éléments de la liste. Chaque élément de la liste (donc chaque objet ou instance) calculera son prix selon sa méthode, peu importe que ce soit une sphère, une pyramide ou encore un autre article !

3.3 Classes, attributs et méthodes

En programmation orientée objet, une classe est une description formelle d'un ensemble d'objets avec leurs attributs et leurs méthodes.

Les attributs d'une classe décrivent l'état de l'objet, ses propriétés. Celles-ci peuvent être de nature très différente : nombres, textes, booléens, listes, ou même des instances d'autres classes.

Les méthodes d'une classe décrivent les (inter)actions avec les objets (calculs, tests ou autres). Elles sont similaires à des fonctions (sauf que les méthodes sont attachées à une classe) et peuvent, au besoin, retourner un résultat.

3.4 Abstraction, encapsulation et masquage

Le monde réel dispose de plein de facettes dont certaines sont difficiles, voire impossibles à être modélisées informatiquement. On doit donc se contenter des aspects nécessaires et laisser de côté les détails marginaux. Ce type de modélisation informatique est appelé abstraction.

À l'étape suivante on regroupe les données dans des structures convenables et on définit les méthodes d'accès à ces données (que ce soit en consultation ou en modification). L'idée fondamentale de la POO est de garder ensemble (dans la même structure) les données et les méthodes qui les manipulent. C'est le principe de l'encapsulation. Elle va souvent de pair avec le masquage qui essaye de garder secrètes les données et de n'y donner accès que lorsque c'est nécessaire. De plus, au lieu de donner accès directement aux données, on intercale souvent des méthodes (des accesseurs et des modificateurs), qui effectuent des contrôles (de plausibilité, de sécurité, ...). De cette manière la classe obtenue (données et méthodes) forme une sorte de boîte noire ayant le comportement et les propriétés désirés.

3.5 Instances

Lorsque nous programmons en POO, nous devons d'abord définir une classe (p. ex. : la classe `Pyramid`) avant de pouvoir en créer des instances (`p1`, `p2`) avec lesquelles nous savons travailler. Chaque instance est alors une entité en soi, chacune avec ses attributs et ses méthodes. Ainsi `p1` pourrait avoir une hauteur de 25 cm alors que celle de `p2` ne pourrait être que 13 cm. Il est important de noter qu'après avoir défini une classe, on n'y touche plus et qu'on ne se servira que des instances pour réaliser le travail de programmation.

Remarque : Contrairement à des variables de type standard comme `int` ou `float`, une instance est une structure de données, ce qui implique aussi une attention particulière à apporter lors de la copie. Ainsi l'instruction de copie entre entiers `y = x` crée une variable `y` complètement indépendante de `x`, alors que l'instruction de copie entre instances `p2 = p1` crée bien une variable `p2`, mais elle est attachée à la **même structure de données** que `p1` ! Si on veut créer une copie indépendante, alors il faut utiliser la méthode `deepcopy` du module `copy`. Il en est de même pour toutes les autres structures de données qui sont *mutables*, comme p. ex. les listes et les dictionnaires.

3.6 Constructeurs et autres méthodes spécifiques

Lors de la création d'une instance, Python effectue un certain nombre de démarches (p. ex. réserver un bloc de mémoire pour y mettre les données, relier ce bloc de données à la variable, ...). Mais il reste aussi des actions que nous devons faire nous-mêmes.

Sphere
radius : float wood_price : float working_time : float tool_time : float
__init__() __str__() : str calculate_volume(): float calculate_surface(): float calculate_price(): int set_radius(new_radius:float)

Reprenons pour cela la sphère de notre menuisier.

La classe `Sphere` dispose de quatre attributs. Ces attributs sont déclarés et initialisés dans une méthode spéciale qui est le *constructeur* (anglais : *initializer*). Cette méthode, si elle existe, porte toujours le nom `__init__` et est appelée automatiquement lors de la création de l'instance. Un constructeur ne retourne jamais une valeur explicite, mais crée implicitement l'instance en question.

Comme il y a un certain type de sphère qui se vend très bien, notre menuisier aimerait initialiser chaque nouvelle instance d'une sphère avec ces données.

Le code suivant en montre une implémentation possible :

```
1  from math import pi
2
3  class Sphere:
4
5      def __init__(self):
6          self.radius = 6                 # radius [cm]
7          self.wood_price = 2.5           # wood cost per dm^3
8          self.working_time = 1.5         # needed manpower in hours
9          self.tool_time = 1.25           # needed tool-time in hours
10
11     def __str__(self):
12         return f"Sphere␣with␣radius␣of␣{self.radius}␣cm."
13
14     def set_radius (self, new_radius):    # sets radius if it is within limits
15         if 3 <= new_radius <= 20:
16             self.radius = new_radius
17
18     def calculate_volume(self):         # returns volume in dm^3
19         return 4 / 3 * pi * self.radius ** 3 / 1000
20
21     def calculate_surface(self):        # returns surface in dm^2
22         return 4 * pi * self.radius ** 2 / 100
23
24     def calculate_price(self):          # returns manufacturing price + 25% win
25         surface_price = self.calculate_surface()
26         volume_price  = self.calculate_volume() * self.wood_price
27         time_price = self.working_time * 25 + self.tool_time * 8
28         return round((surface_price + volume_price + time_price) * 1.25)
29
30     def compare_volume_to(self, other):
31         return self.calculate_volume() - other.calculate_volume()
```

La déclaration et l'initialisation se font dans le *constructeur* (anglais : *initializer*), c'est-à-dire la méthode `__init__(self)` (lignes 5 à 9) ; on y voit une particularité du langage Python, à savoir le mot-clé `self`. Il sert de référence à la classe elle-même et doit être utilisé en tant que paramètre pour toute méthode de la classe et aussi pour différencier les attributs des variables locales. Il est important de noter que le mot-clé `self` doit être utilisé pour accéder aux attributs et méthodes, mais ne doit plus être utilisé dans la liste des arguments lors d'un appel ! Ceci est illustré dans la méthode `calculate_price(self)` qui utilise trois variables locales intermédiaires et fait appel à différentes méthodes.

Pour créer une nouvelle instance (c'est-à-dire un nouvel objet) de la classe, on appelle le constructeur sans passer d'argument pour `self` (car `self` est simplement une référence à l'objet qui sera construit). En l'occurrence, une nouvelle sphère est créée et stockée dans la variable `sp` comme suit : `sp = Sphere()`

La méthode `__str__(self)` (lignes 11 et 12) est une autre méthode spécifique à Python. Elle sert à retourner une description succinte, mais parlante de l'instance, sous forme de string. Ainsi la commande `print(s)` (`s` étant une instance de la classe `Sphere`) retourne `Sphere with radius of 6 cm.` au lieu d'un message obscur du genre `<__main__.Sphere object at 0x02E6C850>` si cette méthode n'existait pas.

Les attributs sont généralement publics, cela veut dire qu'on peut les accéder directement, de la manière `<instance>.<attribut>`, mais il y a des situations qui exigent la vérification d'une valeur avant de l'affecter à un attribut. Dans ces cas on passe par une méthode qu'on appelle modificateur (anglais : *setter*) qui effectue ce travail.

Dans l'exemple de la sphère de notre menuisier, les rayons possibles vont de 3 à 20 cm à cause de la machine utilisée. Il a donc fallu ajouter le modificateur `set_radius(new_radius)` (lignes 14–16), qui vérifie que la valeur de `new_radius` est bien dans les limites avant de l'affecter à l'attribut en question.

De la même manière, on peut se servir de méthodes dédiées pour lire la valeur d'un attribut. Ce sont des accesseurs (anglais : *getter*). En Python, on ne s'en sert que très peu, notamment lorsque la valeur de l'attribut doit être transcrite ou convertie (p. ex. une température à convertir de °C en °F).

Finalement il existe des méthodes qui prennent comme paramètre une instance de classe (que ce soit la même ou une autre). Dans l'exemple de notre menuisier, la méthode `compare_volume_to(other)` (lignes 30 et 31) retourne la différence des volumes de l'instance courante et de celle passée comme argument, peu importe que ce soit une sphère ou une pyramide.

Exercice 3.1 – Fractions

Créer la classe `Fraction` qui permet de représenter informatiquement une fraction :

Fraction
n : int d : int
__init__(new_n:int, new_d:int) __str__() : str set_d(new_d:int) simplify() invert() add(other:Fraction) subtract(other:Fraction) multiply_by(other:Fraction) divide_by(other:Fraction)

Le constructeur initialise la fraction avec les arguments respectifs. Si `new_d` est 0 alors d prendra la valeur 1.

`__str__` renvoie une chaine de type « `n/d (valeur décimale)` », p. ex. : `3/4 (0.75)`.

`set_d` fixe le dénominateur sur la nouvelle valeur si celle-ci est différente de 0.

`simplify` rend la fraction irréductible en divisant les deux membres par leur pgcd.

`invert` inverse la fraction si possible.

Les quatre autres méthodes effectuent les opérations arithmétiques indiquées si possible. Le résultat doit être rendu irréductible chaque fois.

Exercice 3.2 – Compte bancaire

Écrire un programme qui définit une classe `BankAccount`, permettant d'instancier des objets tels que `account1`, `account2`, etc. Le constructeur de cette classe initialisera deux attributs `name` et `balance` avec les valeurs par défaut `"Bill"` et 1000.

Trois autres méthodes seront définies :

⋆ `deposit(amount)` permettra d'ajouter une certaine somme au solde ;

⋆ `withdraw(amount)` permettra de retirer une certaine somme du solde ;

⋆ `__str__` servira à afficher le nom du titulaire et le solde de son compte.

Voici, en guise d'exemple, un programme principal utilisant la classe susmentionnée :

```
account1 = BankAccount("Tim", 800)
account1.deposit(350)
account1.withdraw(200)
print(account1)
account2 = BankAccount()
account2.deposit(25)
print(account2)
```

À l'exécution, l'affichage suivant est produit :

```
Le solde du compte bancaire de Tim est de 950.00 euros.
Le solde du compte bancaire de Bill est de 1025.00 euros.
```

Exercice 3.3 – Nombres *

Créer la classe `Number` qui permet d'effectuer des calculs sur un nombre naturel :

Number
value : int
__init__(new_value:int) __str__() : str set_value(new_value:int) factorial() : int sum_of_divisors() : int reverse() : int is_square() : bool is_prime() : bool is_perfect() : bool is_palindrome() : bool is_amicable_to(other) : bool

- ⋆ Le constructeur initialise `value` sur la valeur absolue de l'argument.
- ⋆ `__str__` renvoie une chaine de type « `value is x` », p. ex. : `value is 7`.
- ⋆ `set_value` fixe `value` sur la valeur absolue de l'argument.
- ⋆ `factorial` retourne la factorielle de `value` ($0! = 1$).
- ⋆ `sum_of_divisors` retourne la somme des diviseurs de `value`, y compris les diviseurs triviaux 1 et `value`.
- ⋆ `reverse` retourne le nombre écrit à l'envers. Ainsi 123 donnera 321 et 150 donnera 51.
- ⋆ `is_square` vérifie si `value` est un carré parfait (carré d'un nombre entier).
- ⋆ `is_prime` vérifie si `value` est un nombre premier.
- ⋆ `is_perfect` vérifie si `value` est un nombre parfait. Un nombre n est parfait si la somme de ses diviseurs est égale à $2 \cdot n$.
- ⋆ `is_palindrome` vérifie si `value` est un nombre palindrome. Un nombre palindrome a la même valeur dans les deux sens de lecture ($\rightarrow$ et $\leftarrow$).
- ⋆ `is_amicable_to` vérifie si `value` est amicale avec la valeur de l'argument `other`. Deux nombres naturels distincts x et y sont dits *amicaux* ssi leur propre somme ainsi que la somme des diviseurs de chacun d'eux sont trois nombres égaux :

$$x + y = \text{somme des diviseurs de } x = \text{somme des diviseurs de } y$$

Exercice 3.4 – Triangles *

Créer la classe `Point` qui permet de représenter un point avec son nom et ses coordonnées (entières par hypothèse) :

Point
name : str x : int y : int
__init__(name:str, a:int, b:int) __str__() : str distance_to(other:Point) : float

- ⋆ Le constructeur initialise les attributs sur les valeurs des arguments, mais si l'argument `name` est égal à `"random"`, alors le nom sera une lettre majuscule choisie au hasard et x et y seront des entiers aléatoires compris entre les arguments a et b (bornes comprises).
- ⋆ `__str__` renvoie une chaine de type « `name(x,y)` », p. ex. : `P(5,4)`.
- ⋆ `distance_to(other)` retourne la distance vers le point `other`.

Utiliser cette classe `Point` pour créer la classe `Triangle` qui permet de gérer des triangles et de faire des calculs avec eux :

Triangle
pa : Point pb : Point pc : Point
__init__(pa:Point, pb:Point, pc:Point) __str__() : str set_random_data(min:int, max:int) calculate_circumference() : float calculate_surface() : float show_type()

- ⋆ Le constructeur initialise les attributs sur les valeurs des arguments, les valeurs par défaut étant $A(0,0)$, $B(3,0)$ et $C(0,4)$.
- ⋆ `__str__` retourne une chaine « `triangle: (name(x,y), name(x,y), name(x,y))` », p. ex. : `triangle: (G(2,3), A(6,3), O(6,4))`.
- ⋆ `set_random_data(min, max)` recrée les trois points avec des noms aléatoires et des coordonnées aléatoires entre `min` et `max`.
- ⋆ `calculate_circumference` retourne le périmètre du triangle.
- ⋆ `calculate_surface` retourne la surface du triangle par la méthode de Héron.
- ⋆ `show_type` affiche dans la console le triangle avec ses sommets, son type (équilatéral, isocèle, scalène) et indique s'il est rectangle avec mention du sommet en question.

  Exemple :

  `The triangle: (G(2,3), A(6,3), O(6,4)) is scalene and right-angled at A.`

Exercice 3.5 – Jeu de cartes

Créer une classe `DeckCards` servant à décrire un jeu de 52 cartes. On définira d'abord une classe `Card` composée de deux attributs, à savoir

- ⋆ la valeur (A, 2, 3, …, 9, X, J, Q, K) ;
- ⋆ la couleur (on pourra écrire S, H, C, D au lieu de ♠, ♡, ♣, ♢).

À côté du constructeur, cette classe sera dotée de la méthode `__str__` permettant d'afficher ses attributs.

La classe `DeckCards` sera pourvue des méthodes `shuffle_cards` et `__str__` servant à mélanger le talon des 52 cartes resp. à afficher son contenu. Enfin la méthode `draw_card` sert à renvoyer (et enlever) la première carte du talon.

Le mélange des cartes peut se faire à l'aide de la fonction `shuffle` contenue dans le module `random`.

Le programme principal affiche le talon complet généré par le constructeur (c'est-à-dire avant le mélange), et puis le talon mélangé. Il enlève et affiche les dix premières cartes du talon mélangé et réaffiche le talon restant.

Voici un exemple d'exécution du programme.

```
Talon ordonné :
SA S2 S3 S4 S5 S6 S7 S8 S9 SX SJ SQ SK
HA H2 H3 H4 H5 H6 H7 H8 H9 HX HJ HQ HK
CA C2 C3 C4 C5 C6 C7 C8 C9 CX CJ CQ CK
```

```
DA D2 D3 D4 D5 D6 D7 D8 D9 DX DJ DQ DK

Talon mélangé :
S7 H6 H8 C5 SA S3 D5 SQ H2 CX H3 C8 S4
S5 D4 SK C7 S6 DJ C6 D2 DK CQ S9 DX D3
C9 CA HX D6 S2 HA DA D7 SJ D9 DQ HK C2
H4 SX H5 CJ C3 H9 D8 C4 HJ H7 CK HQ S8

Carte tirée : S7
Carte tirée : H6
Carte tirée : H8
Carte tirée : C5
Carte tirée : SA
Carte tirée : S3
Carte tirée : D5
Carte tirée : SQ
Carte tirée : H2
Carte tirée : CX

Talon restant :
H3 C8 S4 S5 D4 SK C7 S6 DJ C6 D2 DK CQ
S9 DX D3 C9 CA HX D6 S2 HA DA D7 SJ D9
DQ HK C2 H4 SX H5 CJ C3 H9 D8 C4 HJ H7
CK HQ S8
```

Exercice 3.6 * – Classes d'objets géométriques

Définir d'abord une classe `Point` caractérisée par son abscisse et son ordonnée. Définir les méthodes suivantes pour cette classe :

- ⋆ le constructeur initie l'objet (avec valeurs par défaut si l'utilisateur omet les arguments lors de l'appel) ;
- ⋆ `get_x` retourne l'abscisse du point ;
- ⋆ `get_y` retourne l'ordonnée du point ;
- ⋆ `set_x` remplace l'abscisse du point par l'argument fourni ;
- ⋆ `set_y` remplace l'ordonnée du point par l'argument fourni ;
- ⋆ `distance` retourne la distance entre le point en cours et le point fourni comme argument ;
- ⋆ `is_identical` retourne `True` si le point en cours est le point fourni comme argument, et `False` sinon ;
- ⋆ `__str__` sert à afficher les coordonnées du point en cours.

Définir ensuite une classe `Segment` caractérisée par deux points, le point d'origine et le point d'extrémité. Définir les méthodes suivantes pour cette classe :

- ⋆ le constructeur initie l'objet (avec valeurs par défaut si l'utilisateur omet les arguments lors de l'appel) ;
- ⋆ `get_boundary` retourne l'extrémité du segment ;

- ⋆ `get_origin` retourne l'origine du segment ;
- ⋆ `set_boundary` remplace l'extrémité du segment par le point fourni comme argument ;
- ⋆ `set_origin` remplace l'origine du segment par le point fourni comme argument ;
- ⋆ `get_length` retourne la longueur du segment ;
- ⋆ `get_midpoint` retourne le point-milieu du segment ;
- ⋆ `__str__` sert à afficher l'origine et l'extrémité du segment.

Définir finalement une classe `Quadrilateral` caractérisée par deux diagonales d_1 et d_2. Définir les méthodes suivantes pour cette classe :

- ⋆ le constructeur initie l'objet ;
- ⋆ `get_diagonal1` retourne la première diagonale d_1 du quadrilatère ;
- ⋆ `get_diagonal2` retourne la deuxième diagonale d_2 du quadrilatère ;
- ⋆ `set_diagonal1` remplace la première diagonale d_1 du quadrilatère en cours par la diagonale fournie comme argument ;
- ⋆ `set_diagonal2` remplace la deuxième diagonale d_2 du quadrilatère en cours par la diagonale fournie comme argument ;
- ⋆ `is_parallelogram` retourne **`True`** si le quadrilatère en cours est un parallélogramme et **`False`** sinon ;
- ⋆ `is_rectangle` retourne **`True`** si le quadrilatère en cours est un rectangle et **`False`** sinon ;
- ⋆ `__str__` sert à afficher les informations relatives au quadrilatère en cours.

Voici un programme principal qui sert simplement à tester les classes susmentionnées :

```
p1 = Point(1, 9)
p2 = Point(2, 5)
print(p1)
print(p2)
print(Point())
p1.set_x(p2.get_y())
print(p1)
print(p1.distance(p2))
print(p2.is_identical(Point(2, 5)))
s1 = Segment(p1, p2)
print(s1.get_boundary())
print(s1)
s1.set_origin(Point(3, 7))
print(s1)
print(f"{s1.get_length():.3f}")
print(s1.get_midpoint())
print(Segment())
s2 = Segment(Point(7, 8), Point(5, 1))
q1 = Quadrilateral(s1, s2)
print(q1)
print(q1.is_parallelogram())
```

```
q1.set_diagonal1(Segment(Point(7, 1), Point(5, 8)))
print(q1)
print(q1.is_rectangle())
```

Voici l'affichage produit lors de l'exécution :

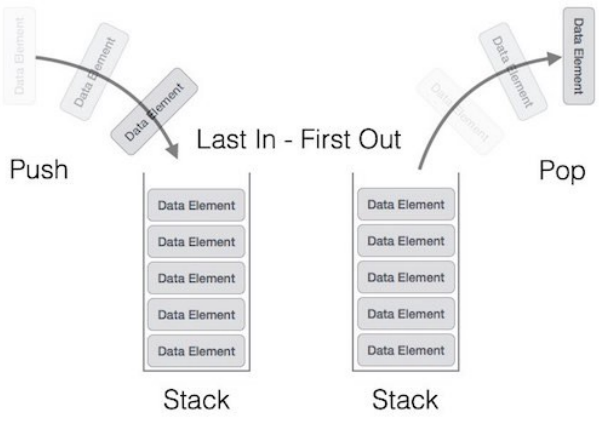
```
(1; 9)
(2; 5)
(0; 0)
(5; 9)
5.0
True
(2; 5)
segment [ (5; 9), (2; 5) ]
segment [ (3; 7), (2; 5) ]
2.236
(2.5; 6.0)
segment [ (0; 0), (1; 0) ]
quad [ (3; 7), (7; 8), (2; 5), (5; 1) ]
False
quad [ (7; 1), (7; 8), (5; 8), (5; 1) ]
True
```

3.7 Application : la pile (*stack*)

On appelle *structure de données* toute organisation logique de données permettant de simplifier ou d'accélérer leur traitement. À côté des structures de données prédéfinies en Python (p. ex. les listes et les dictionnaires), le programmeur peut créer ses propres structures adaptées aux problèmes à résoudre.

Il s'agit donc de programmer (simuler) une structure qui existe réellement avec les moyens de bord du langage de programmation Python. Dans le cadre de ce cours, nous n'envisageons que deux structures de données linéaires, à savoir la pile (*stack*) et, à titre facultatif dans la section suivante, la file (*queue*).

Une **pile** est une structure de données linéaire basée sur le principe du « dernier arrivé, premier sorti » ou LIFO (*Last In, First Out*). Ainsi avec une pile de livres, les livres s'entassent toujours sur le sommet de la pile et le dernier livre placé sur le sommet de la pile sera le premier à être enlevé par la suite.



Afin de manipuler une telle structure, les méthodes suivantes sont nécessaires :

- ⋆ `__init__` pour créer une pile vide correctement initialisée ;
- ⋆ `push` pour placer un objet sur le sommet de la pile ;
- ⋆ `is_empty` pour vérifier si la pile est vide ou non ;
- ⋆ `pop` pour enlever et renvoyer l'élément se trouvant sur le sommet de la pile ;
- ⋆ `top` pour renvoyer l'élément se trouvant sur le sommet de la pile sans l'enlever ;
- ⋆ `size` pour renvoyer le nombre d'objets placés dans la pile ;
- ⋆ `__str__` pour pouvoir afficher la pile de façon implicite (conversion automatique en un string).

Voici une implémentation possible pour ce nouveau type de données :

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def push(self, x):
        self.items.append(x)
    def pop(self):
        if len(self.items) > 0:
            return self.items.pop()
        else:
            print("Empty␣stack!")
    def top(self):
        if len(self.items) > 0:
            return self.items[-1]
        else:
            print("Empty␣stack!")
    def size(self):
        return len(self.items)
    def __str__(self):
        s = "Stack␣(from␣top␣to␣bottom):␣[␣"
        for x in reversed(self.items):
            s += f"{x}␣"
        s += "]"
        return s
```

Le programme suivant utilise cette implémentation du type abstrait pile.

```
from Stack import *

stack = Stack()

print("Pushing")
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
print("Stack␣size:", stack.size())
```

```
print(stack)
print("Stack␣size:", stack.size())
while not stack.is_empty():
    print(stack.top(), end = "␣")
    stack.pop()
print("\nStack␣size:", stack.size())
```

À l'exécution ce programme produit l'affichage suivant :

```
Pushing
Stack size: 4
Stack (from top to bottom): [ 4 3 2 1 ]
Stack size: 4
4 3 2 1
Stack size: 0
```

Exercice 3.7 – Parenthèses balancées

Il faut réaliser un programme qui vérifie si une suite de symboles de regroupement [,], {, }, (,) est correctement formée.

Voici trois exemples d'expressions bien formées :

```
{ ( { ( [ ] [ ] [ ] ) } ) ( ) }
( ) ( ) [ ] { }
( [ { ( [ { } ( ) ] ( ) ) } [ ] ] )
```

et trois exemples d'expressions mal formées :

```
[ { ] }
( ( ) ) )
{ [ } [ ] }
```

L'idée est de placer le symbole ouvrant (, [, { sur une pile. Une fois qu'un symbole de fermeture),] ou } est rencontré lors de l'analyse de l'expression, il faut vérifier qu'il correspond bien au symbole se trouvant au sommet de la pile. Si ces deux symboles diffèrent, alors l'expression est mal formée.

Par contre, si l'entièreté de l'expression a été examinée sans avoir eu de discordance et que la pile est finalement vide, l'expression est bien formée.

Écrire un programme avec une fonction `check` qui répond par un booléen si une chaîne de caractères fournie comme argument contient des parenthèses correctement balancées. Tous les autres caractères de la chaîne sont ignorés, c'est-à-dire la chaîne `"2*[3+4*(5-x)**2-y]/sin(3)"` est traitée de la même façon que la chaîne `"[()]()"` et la fonction `check` renvoie donc `True` pour cet exemple.

Exercice 3.8 – Conversion d'un nombre en notation binaire

Écrire un programme qui convertit un nombre décimal en base 2.

Voici le procédé à utiliser : on divisera le nombre par deux (division euclidienne) et le reste de cette division sera empilé. Le quotient sera quant à lui de nouveau divisé par deux, le reste empilé, etc. jusqu'à ce que le quotient devienne 0.

```
100 // 2 = 50 reste 0
50 // 2 = 25 reste 0
25 // 2 = 12 reste 1
12 // 2 = 6 reste 0
6 // 2 = 3 reste 0
3 // 2 = 1 reste 1
1 // 2 = 0 reste 1
```

La représentation binaire s'obtient alors en dépilant tous les chiffres empilés préalablement. Ainsi, la représentation binaire de 100_{10} est 1100100_2.

Exercice 3.9 * Compléter l'exercice précédent pour qu'on puisse convertir le nombre décimal entré en une base allant de 2 à 36. On utilisera les symboles A, B, ..., Z pour représenter les chiffres de 10 à 35.

Écrire également une fonction permettant la conversion réciproque : un nombre entré en une base b, avec $2 \leq b \leq 36$ est converti en notation décimale.

Voici un exemple d'exécution du programme complété :

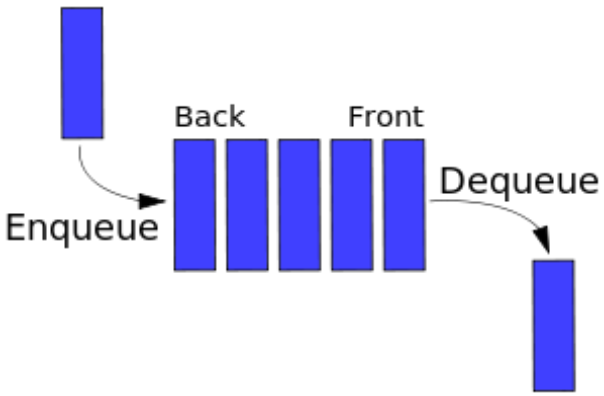
```
Entrez un nombre en base 10 : 64738
(64738)_10 = (1111110011100010)_2
Entrez la base : 16
(64738)_10 = (FCE2)_16
Entrez un nombre en base 16 : D020
(D020)_16 = (53280)_10
```

Exercice 3.10 * Réimplémenter le tri rapide (*Quicksort*) sous forme non récursive, c'est-à-dire la fonction `quicksort` ne devra plus s'appeler elle-même.

Indication : Avoir recours à une pile en y empilant toutes les sous-listes à droite des pivots déjà placés. Après avoir placé un pivot, continuer à examiner la sous-liste à gauche (s'il en reste une) à l'intérieur d'une boucle qu'on quitte lorsqu'il ne reste plus de sous-liste à gauche à traiter. Dépiler ensuite et traiter les sous-listes à droite des pivots. La fonction `partition` reste inchangée.

3.8 Application : la file (*queue*) *

Une file est une structure de données linéaire basée sur le principe du « premier arrivé, premier sorti » ou FIFO (*First In, First Out*). Ainsi dans une file d'attente devant un guichet, la première personne entrée dans la file sera la première personne a être servie.



Afin de manipuler une telle structure, les primitives suivantes sont nécessaires :

- ⋆ `__init__` pour créer une file vide correctement initialisée ;
- ⋆ `enqueue` pour placer un objet à l'arrière de la file ;
- ⋆ `is_empty` pour vérifier si la file est vide ou non ;
- ⋆ `dequeue` pour enlever et renvoyer l'élément se trouvant en tête de la file ;
- ⋆ `top` pour renvoyer l'élément se trouvant en tête de la file sans l'enlever ;
- ⋆ `size` pour renvoyer le nombre d'objets placés dans la file ;
- ⋆ `__str__` pour pouvoir afficher la file de façon implicite (conversion automatique en un string).

Voici une implémentation possible pour ce nouveau type de données :

```
class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def enqueue(self, x):
        self.items.insert(0, x)
    def dequeue(self):
        if len(self.items) > 0:
            return self.items.pop()
        else:
            print("Empty queue!")
    def top(self):
        if len(self.items) > 0:
            return self.items[-1]
        else:
            print("Empty queue!")
    def size(self):
        return len(self.items)
    def __str__(self):
        s = "Queue ==> : ["
        for x in self.items:
            s += f"{x} "
        s += "]"
        return s
```

Le programme suivant utilise cette implémentation du type abstrait pile.

```
from Queue import *

q = Queue()
for i in range(1, 5):
    q.enqueue(i)
print(q)
print("Queue size:", q.size())
while not q.is_empty():
```

```
    print(q.top())
    print(q.dequeue())
print(q)
print(q.dequeue())
```

À l'exécution ce programme produit l'affichage suivant :

```
Queue (from last to first) : [ 4 3 2 1 ]
Queue size: 4
1
1
2
2
3
3
4
4
Queue (from last to first) : [ ]
Empty queue!
None
```

Exercice 3.11 – Le problème de Joséphus

En mathématiques et en informatique, le *problème de Joséphus* est lié à certaines formulettes d'élimination. Il a été énoncé sous différentes formes, mais sa première formulation est due à Flavius Josephus.

Des soldats juifs, cernés par des soldats romains, décident de former un cercle. À partir d'une position choisie au hasard, on compte les soldats dans un sens donné et chaque troisième est exécuté. Tant qu'il y a des soldats, la sélection continue. Le but est de trouver à quel endroit doit se tenir un soldat pour être le dernier. Joséphus, peu enthousiaste à l'idée de mourir, parvint à trouver l'endroit où se tenir. Quel est-il ?

Écrire une fonction `josephus` admettant comme arguments le nombre de personnes dans le cercle initial et le numéro k tel qu'une personne sur k soit tuée à tour de rôle dans le cercle (dans l'énoncé précédent, $k = 3$).

Voici deux exemples d'exécution du programme :

```
Entrez le nombre de personnes : 23
Entrez k : 3
3 6 9 12 15 18 21 1 5 10 14 19 23 7 13 20 4 16 2 17 11 22
Dernière personne dans le cercle : 8

Entrez le nombre de personnes : 100
Entrez k : 2
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54
56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 3 7 11
15 19 23 27 31 35 39 43 47 51 55 59 63 67 71 75 79 83 87 91 95 99 5 13 21 29
37 45 53 61 69 77 85 93 1 17 33 49 65 81 97 25 57 89 41 9
Dernière personne dans le cercle : 73
```

3.9 Simulations

Les exercices de cette section (dont au moins les deux premiers doivent être traités en classe) peuvent être résolus à l'aide des structures de données vues récemment (les files notamment), mais le recours à cette classe n'est pas obligatoire. En effet, les mécanismes utilisés peuvent être (ré)implémentés avec des listes Python, sans compliquer le code du programme.

Exercice 3.12 – Car-wash

Écrire un programme qui simule une station de lavage de voitures. Nous nous intéressons au temps moyen d'attente passé dans l'unique file à l'entrée de la station en fonction du nombre d'arrivées de clients, de la durée d'un lavage (un programme de lavage supposé unique) et de la durée de la simulation.

Pour simplifier, nous supposons qu'une nouvelle arrivée dans la station se fait en moyenne toutes les x secondes, par exemple $x = 240$. On fixe la durée d'un lavage à 5 minutes et celle d'une journée à 10 heures. Le programme simulera 10 journées avant d'afficher les statistiques obtenues.

Le programme principal effectue un tour de boucle pour chaque seconde de simulation. À chaque tour de boucle :

- ⋆ on vérifie si une arrivée a lieu. Si tel est le cas, nous mémorisons le temps actuel (la seconde en cours) dans la file d'attente ;
- ⋆ si la machine de lavage n'est pas active et que la file d'attente n'est pas vide, la voiture en tête de la file d'attente est sortie de la file et son lavage démarre. De plus, les compteurs pour le calcul du temps moyen passé dans la file sont mis à jour ;
- ⋆ finalement, le processus de lavage est avancé d'une seconde.

Pour réaliser la simulation, nous nous aidons de la classe `CarWash` définie ainsi :

```
class CarWash:
    def __init__(self, tw):
        self.time_wash = tw
        self.time_remaining = 0
    def busy(self):
        return self.time_remaining > 0
    def tick(self):
        if self.busy():
            self.time_remaining -= 1
    def start_wash(self):
        self.time_remaining = self.time_wash
    def __str__(self):
        s = f"lavage, total = {self.time_wash},"
        return s + f" reste = {self.time_remaining}"
```

Voici un exemple d'exécution du programme :

```
Temps pour laver une voiture : 300 secondes
Une voiture arrive toutes les 240 secondes en moyenne.
Durée de la simulation : 36000 secondes
Nombre de simulations : 10
Nombre de voitures lavées : 1184
Nombre de voitures non lavées : 317
Temps moyen d'attente : 3907.61 secondes
```

Exercice 3.13 – Simulation d'un guichet

Écrire un programme qui calcule le pourcentage de clients qui repartent sans avoir été servis par l'unique guichet présent. Ce pourcentage sera mis en relation avec le nombre de clients qui arrivent et forment une file d'attente avant d'être servis.

Nous supposons qu'un client arrive toutes les 40 secondes en moyenne. La patience d'un client dans la file est limitée, elle varie aléatoirement entre 2 et 10 minutes. Le guichet sert les clients dans un temps compris entre une demi-minute et 5 minutes. Toute la simulation fonctionnera sur un jour de travail normal de 8 heures. Pour établir les statistiques demandées, le programme simulera 10 journées avant d'afficher les résultats.

Voici un exemple d'exécution du programme :

```
Un client arrive toutes les 40 secondes en moyenne.
Durée de la simulation : 28800 secondes
Nombre de simulations : 10
Nombre de clients servis : 1728
Nombre de clients impatients partis : 5458 (75.10%)
Nombre de clients en attente après la fermeture du guichet : 82 (1.13%)
Temps moyen d'attente : 393.52 secondes
```

Exercice 3.14 * Compléter le programme précédent : au lieu d'avoir un guichet unique, on dispose de n guichets (le nombre $n \geq 1$ est entré par l'utilisateur). Les clients forment encore une file d'attente unique, et dès qu'un guichet est libre, le client en tête de file est servi par ce guichet.

Voici deux exemples d'exécution du programme complété :

```
Entrez le nombre de guichets : 3
Un client arrive toutes les 40 secondes en moyenne.
Durée de la simulation : 28800 secondes
Nombre de simulations : 10
Nombre de clients servis : 5233
Nombre de clients impatients partis : 1828 (25.70%)
Nombre de clients en attente après la fermeture du guichet : 52 (0.73%)
Temps moyen d'attente : 203.92 secondes

Entrez le nombre de guichets : 5
Un client arrive toutes les 40 secondes en moyenne.
Durée de la simulation : 28800 secondes
Nombre de simulations : 10
Nombre de clients servis : 7062
Nombre de clients impatients partis : 133 (1.85%)
Nombre de clients en attente après la fermeture du guichet : 11 (0.15%)
Temps moyen d'attente : 44.82 secondes
```

Exercice 3.15 ** Écrire un programme qui permet de simuler le **débit de bières** dans un bistrot.

1. Implémenter une classe `Customer` qui décrit l'état actuel d'un client.
   - ⋆ Le constructeur initialise le taux d'alcoolémie (mesuré en ‰) à 0 et retient que ce client n'a pas encore commandé de bière.
   - ⋆ La méthode `waiting` répond par `True` ssi ce client vient de commander une bière qu'il n'a pas encore reçue.

- ⋆ La méthode `coma` répond par `True` ssi ce client a un taux d'alcoolémie ≥ 3.
- ⋆ La méthode `order_beer` est appelée lorsque ce client commande une bière.
- ⋆ La méthode `drink_beer` est appelée lorsque ce client boit une bière. Son taux d'alcoolémie augmente tout de suite d'une valeur aléatoire comprise entre 0,2 et 0,3. (On pourra calculer un nombre naturel aléatoire, compris entre 200 et 300, et le diviser par 1000.)
- ⋆ La méthode `tick` est appelée à chaque seconde de la simulation. Si le client n'est pas dans le coma, le taux d'alcoolémie diminue de $\frac{1}{36000}$, sans pouvoir devenir négatif. On ne modifie pas le taux d'un client dans le coma, pour bien retenir l'information qu'il est dans le coma.

2. Implémenter la simulation suivante :
   - ⋆ Durée : 10 soirées, dont chacune dure 6 heures.
   - ⋆ Nombre fixe de clients présents au bistrot : 50 (tous sobres au début de la soirée).
   - ⋆ Toutes les 45 secondes **en moyenne**, une bière est commandée par un client choisi au hasard, qui n'attend pas déjà une bière et n'est pas dans le coma. Ce client se place alors à la fin d'une file d'attente.
   - ⋆ Durant les périodes où tous les 50 clients sont dans la file d'attente ou dans le coma, aucune bière supplémentaire n'est commandée.
   - ⋆ Lorsqu'au moins un client se trouve dans la file d'attente, une bière est préparée. Elle sera prête après exactement 30 secondes et sera alors servie au client au début de la file qui la boit instantanément et quitte la file. S'il reste des clients dans la file, une nouvelle bière est préparée (ce qui durera à nouveau 30 secondes).
   - ⋆ Si après avoir bu une bière, un client atteint un taux d'alcoolémie ≥ 3, il tombe dans le coma et ne commandera plus de bière pendant le reste de la soirée.
   - ⋆ À la fin de la soirée, la bière partiellement préparée n'est plus servie et tous les clients dans la file d'attente sont à considérer comme « assoiffés ».
3. Écrire le programme principal qui, après la simulation des 10 soirées, affichera le nombre total de bières servies, le nombre total de clients tombés dans le coma et le nombre total de clients assoiffés.

Exemple d'exécution :

```
Un client arrive toutes les 45 secondes en moyenne.
Durée d'une soirée : 21600 secondes
Nombre de soirées : 10
Nombre de clients par soirée : 50
Nombre total de bières servies : 4694
Nombre total de clients dans le coma : 39
Nombre total de clients assoiffés : 12
```

Exercice 3.16 ** Écrire un programme qui permet de simuler la **circulation de caddies** devant les caisses d'un supermarché.

1. Implémenter une classe `Shopper` qui décrit l'état actuel d'un client, auquel est associé un seul caddie.

- ⋆ Le constructeur initialise le nombre d'articles achetés (un nombre aléatoire entre 1 et 50) et deux attributs (=variables) booléens : le premier (`unweighted`) vaut `True` ssi ce client a oublié de faire peser les légumes (valeur choisie au hasard : 1 chance sur 20) ; le deuxième (`bad_card`) vaut `True` ssi la carte de crédit de ce client est défectueuse (valeur choisie au hasard : 1 chance sur 50). En outre, l'attribut `waiting_time` indique la durée d'attente de ce client dans sa file d'attente actuelle, l'attribut `total_waiting_time` indique la durée totale d'attente et l'attribut `check_out` contient le numéro de la caisse associée à la file d'attente actuelle.
- ⋆ La méthode `checkout_time` calcule et renvoie le temps total nécessaire pour encoder les articles de ce client et pour le faire payer. Il faut compter 5 secondes par article pour l'encodage. Lorsque les légumes n'ont pas été pesés préalablement, il faut compter 1 minute (au lieu de 5 secondes) pour cet article. Pour payer, il faut compter 15 secondes pour un client normal, et 3 minutes pour un client dont la carte de crédit est défectueuse.
- ⋆ La méthode `nervous` est appelée lorsque ce client est nerveux et change de file d'attente.
- ⋆ La méthode `tick` est appelée à chaque seconde de la simulation. Elle actualise les attributs des temps d'attente.

2. Implémenter la simulation suivante :
   - ⋆ Durée : une semaine de 5 jours, de 8 heures chacun (avec éventuellement des heures supplémentaires).
   - ⋆ Nombre de caisses ouvertes chaque jour : 6 (numérotées : $1, 2, \ldots, 6$).
   - ⋆ Toutes les 30 secondes **en moyenne**, un nouveau client se range à la fin d'une file d'attente avec son caddie plus ou moins rempli. Il choisit la file dont le nombre de caddies en attente est le moins élevé. En cas d'égalité, il choisit par convention celle dont le numéro est le moins élevé.
   - ⋆ Tout client qui se trouve **à la fin d'une même file pendant exactement 1 minute** regarde à sa gauche et à sa droite, et si le nombre total de caddies en attente y est strictement moins élevé que devant lui dans sa propre file, il change instantanément de file, avec en cas d'égalité une préférence pour la file de gauche (indice moins élevé), et se range alors à la fin de la nouvelle file.

     Après exactement une minute, le même procédé recommence.

     Les files 1 et 6 n'ont évidemment qu'une seule file voisine ($1 \to 2$ et $5 \leftarrow 6$).
   - ⋆ Le temps nécessaire pour l'encodage des articles et le paiement n'est pas considéré comme temps d'attente pour ce client, mais pour tous les autres clients dans la file d'attente de cette caisse. Ainsi, lorsque l'encodage des articles d'un client a commencé, ce client ne change plus de file.
   - ⋆ À la fin de la journée, aucun nouveau client n'arrive plus devant les caisses. Ceux qui s'y trouvent déjà devront tous passer, avec les mêmes durées d'encodage des articles et de paiement qu'auparavant, mais il n'y aura plus de changements de file. Le temps de travail de chaque caissière, dépassant les 8 heures initiales jusqu'à ce que le dernier client ait quitté sa caisse, est comptabilisé comme temps « supplémentaire ». À la fin de la semaine, il sera exprimé en heures supplémentaires, arrondi à 0,001 près.

3. Écrire le programme principal qui, après la simulation des 5 jours, affichera le nombre total de clients servis, le nombre total d'articles vendus, le nombre total de minutes d'attente des clients (à 0,01 près), le nombre total de changements de file par des clients nerveux et le nombre total d'heures supplémentaires prestées par les caissières.

Exemple d'exécution :

```
Un client arrive toutes les 30 secondes en moyenne.
Durée d'un jour : 28800 secondes
Nombre de jours : 5
Nombre de caisses ouvertes : 6
Nombre total de clients servis : 4766
Nombre total d'articles vendus : 122164
Nombre total de minutes d'attente : 4610.83
Nombre total de changements de file : 176
Nombre total d'heures supplémentaires : 1.501
```

4 Graphisme avec Pygame

4.1 Introduction

Pygame est une bibliothèque libre multi-plateforme permettant la création de jeux vidéo (surtout en 2D) avec Python. Elle a été développée par Pete Shinners en 2000.

Elle contient des modules permettant de gérer des éléments graphiques, du son ainsi que des routines pour la gestion de périphériques d'entrée (clavier, souris, manette de jeu).

Pygame est basé sur la bibliothèque Simple-DirectMedia-Layer (SDL). Son but est de développer des jeux vidéo sans devoir recourir à un langage de programmation de bas niveau tel que C ou ses dérivés. Cela se fonde sur la supposition que la partie graphique, souvent la plus contraignante à programmer dans un jeu, est réalisée séparément de la logique même du jeu. On peut donc utiliser un langage de haut niveau comme Python pour la logique du jeu.

4.2 Mise en place d'un programme Pygame

Le squelette Pygame suivant apparaît sous cette forme dans quasiment tous les programmes que nous réaliserons :

```
import pygame, sys
from pygame.locals import *
pygame.init()
size = (400, 300)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Hello world!")
pygame.display.update()
pygame.event.get()
```

- ⋆ `import pygame, sys` – Cette première ligne est obligatoire et permet d'importer les bibliothèques `pygame` et `sys` pour que notre programme puisse utiliser les fonctions y incluses. Toutes les fonctions qui permettent la gestion des éléments graphiques, du son, ainsi que d'autres outils spécifiques à Pygame se trouvent dans ces bibliothèques.
- ⋆ `from pygame.locals import *` – Cette ligne est optionnelle et sert à rendre publiques certaines constantes et fonctions de Pygame, notamment les noms (en anglais) des couleurs (`red`, `yellow`, ...).
- ⋆ Ensuite pour pouvoir utiliser Pygame, il faut initialiser cet environnement. La façon habituelle de le faire est d'écrire : `pygame.init()` – Cette fonction initialise tous les modules de la bibliothèque `pygame`. Elle doit obligatoirement être appelée après avoir importé la bibliothèque `pygame` et avant d'appeler d'autres fonctions de Pygame.
- ⋆ `size = (400, 300)` – Cette ligne crée un couple (*tuple*) contenant les dimensions (largeur et hauteur) de la fenêtre Pygame, exprimées en pixels.
- ⋆ `screen = pygame.display.set_mode(size)` – Cette ligne crée l'écran (en fait la fenêtre) utilisé par notre programme. On pourra accéder plus tard à l'écran via la variable `screen`, de type `Surface`.
- ⋆ `pygame.display.set_caption("Hello world!")` – Cette ligne écrit à l'en-tête de la fenêtre Pygame le texte fourni comme argument.
- ⋆ `pygame.display.update()` et `pygame.event.get()` – Ces deux lignes rafraîchissent l'affichage à l'écran physique et attendent une réaction de l'utilisateur (dont nous reparlerons lors de la gestion des événements).

4.3 La boucle principale

Voici une représentation schématique de la boucle principale d'un programme Pygame :

```
done = False
while not done:
    handle_events()
    update_game_state()
    update_screen()
```

Attention ! Il s'agit d'un code générique, c'est-à-dire les trois fonctions dans la boucle ne sont pas des fonctions prédéfinies. Le programmeur peut les remplacer par quelques lignes de code ou bien définir ces fonctions et y inclure le code nécessaire.

La boucle principale susmentionnée est donc structurée en trois parties :

- ⋆ gestion des événements ;
- ⋆ actualisation de l'état du jeu ;
- ⋆ actualisation de l'affichage.

L'état du jeu fait référence à un ensemble de valeurs pour toutes les variables du programme, par exemple la position des différents éléments, le score... À chaque fois qu'une de ces variables est modifiée, l'état du jeu change.

En général, l'état du jeu change en réponse à des événements (comme un clic de souris ou le fait d'appuyer sur une touche du clavier) ou après un laps de temps préétabli. La boucle principale vérifie continuellement si un nouvel événement s'est produit et actualise l'état du jeu. Ce processus est appelé « gestion des événements ».

4.4 Gestion des événements

À l'intérieur de la boucle principale, la gestion des événements se fait à l'aide d'une boucle **for**. Cette boucle contient une chaîne d'instructions **if** et **elif** qui, selon le type d'événement, exécutent le code correspondant :

```
for event in pygame.event.get():
    if event.type == <event_type_1>:
        <do_something>
    elif event.type == <event_type_2>:
        <do_some_other_thing>
    ...
```

On voit que les tests sont tous regroupés à l'intérieur d'une seule boucle **for**.

Les objets de type `Event` ont un attribut nommé `type` qui représente le type d'événement qui se produit. Le module `pygame.locals` contient des constantes pour chaque type d'événement possible.

Par la suite, nous examinerons les événements les plus importants.

4.4.1 Événement QUIT

Lorsque l'utilisateur clique sur le symbole de fermeture dans l'en-tête de la fenêtre (la croix à droite sous Windows, ou le bouton rouge à gauche sous macOS), celle-ci doit se fermer. La constante correspondant à ce type d'événement est nommée `QUIT`.

Considérons maintenant une boucle principale gérant uniquement cet événement :

```
done = False
while not done:
    for event in pygame.event.get():
        if event.type == QUIT:
            done = True
```

Le problème est que, même si le programme sort de la boucle principale, il n'y a pas d'instruction qui ordonne la fermeture de la fenêtre.

Les lignes

```
pygame.quit()
sys.exit()
```

écrites à la fin du code font le nécessaire pour terminer le programme correctement. La fonction `pygame.quit()` est en quelque sorte le contraire de la fonction `pygame.init()` : elle exécute du code qui désactive les modules de la bibliothèque Pygame. Il est conseillé de faire suivre cette fonction par la fonction `sys.exit()` pour terminer « officiellement » le programme. Sans cette dernière ligne, la fenêtre Pygame risque de rester figée à l'écran, notamment sous macOS.

4.4.2 Les événements de réaction aux touches du clavier

L'événement `KEYDOWN` correspond à l'enfoncement d'une touche du clavier, l'événement `KEYUP` au relâchement d'une touche du clavier. Les objets de type `Event` ont également un attribut nommé `key` qui correspond à la touche appuyée (ou relâchée). On peut donc récupérer la touche appuyée (ou relâchée) grâce à `event.key` qui peut prendre différentes valeurs. Voici les plus importantes :

- ⋆ `K_a` pour la touche a (et pareil pour le reste de l'alphabet) ;
- ⋆ `K_0` pour la touche 0 en haut du clavier (et pareil pour les autres chiffres) ;
- ⋆ `K_KP0` pour la touche 0 du pavé numérique (et pareil pour les autres chiffres)
- ⋆ `K_LALT` et `K_RALT` pour la touche alt correspondante ;
- ⋆ `K_LSHIFT` et `K_RSHIFT` pour la touche SHIFT correspondante ;
- ⋆ `K_LCTRL` et `K_RCTRL` pour la touche ctrl correspondante ;
- ⋆ `K_SPACE` pour la touche espace ;
- ⋆ `K_RETURN` pour la touche « return » (passage à la ligne) ;
- ⋆ `K_ESCAPE` pour la touche esc ;
- ⋆ `K_UP`, `K_DOWN`, `K_LEFT` et `K_RIGHT` pour les touches du curseur (flèches).

Exemple d'une gestion d'événements réagissant à l'enfoncement de la flèche vers la gauche du clavier :

```
for event in pygame.event.get():
    if event.type == KEYDOWN and event.key == K_LEFT:
        <do_something>
```

4.4.3 Les événements de réaction à la souris

L'événement `MOUSEBUTTONDOWN` correspond à l'enfoncement d'un bouton de la souris, l'événement `MOUSEBUTTONUP` au relâchement d'un bouton de la souris, et l'événement `MOUSEMOTION` au déplacement de la souris.

Il est possible de vérifier à tout moment l'état des boutons (afin de savoir quel bouton a été pressé) grâce à la fonction `pygame.mouse.get_pressed`. Celle-ci retourne une séquence de trois valeurs booléennes représentant l'état des trois boutons de la souris (de gauche à droite). La valeur `True` veut dire que le bouton en question est pressé au moment de l'appel de la fonction.

Une autre manière d'interpréter les clics de souris consiste à lire l'attribut `event.button` après un événement `MOUSEBUTTONDOWN` ou `MOUSEBUTTONUP`. Le nombre lu indique le bouton ayant provoqué l'événement : 1 (bouton gauche), 2 (bouton du milieu s'il existe), 3 (bouton droit), 4 (scroll-up), 5 (scroll-down).

Il est également possible de récupérer la position du curseur au moyen de la fonction `pygame.mouse.get_pos`.

Exemple d'une gestion d'événements affichant dans la console la position du curseur lorsqu'on enfonce le bouton gauche de la souris (et aucun autre bouton) :

```
for event in pygame.event.get():
    if event.type == MOUSEBUTTONDOWN:
        if pygame.mouse.get_pressed() == (True, False, False):
            print(pygame.mouse.get_pos())
```

4.5 Les éléments graphiques

Pygame possède plusieurs fonctions qui nous permettent de dessiner des figures simples.

4.5.1 La surface de dessin

La surface de dessin peut être vue comme une matrice de points, dont l'origine se trouve dans le coin supérieur gauche.

En admettant que la taille de la fenêtre Pygame a été définie par

```
size = (400, 300)
screen = pygame.display.set_mode(size)
```

le point supérieur gauche a comme coordonnées $(0, 0)$, le point supérieur droit de la fenêtre a comme coordonnées $(399, 0)$, le point inférieur gauche $(0, 299)$ et le point inférieur droit $(399, 299)$. Les couples de coordonnées contiennent d'abord l'abscisse (avec 0 comme valeur de départ à gauche), et puis l'ordonnée (avec 0 comme valeur de départ en haut). Contrairement aux conventions mathématiques, l'axe des ordonnées est orienté **vers le bas** dans Python comme dans la plupart des langages de programmation.

4.5.2 Les couleurs

Dans Pygame, nous disposons de la fonction `Color(name)`, qui renvoie la couleur correspondant à l'argument `name` (string correspondant à la dénomination anglaise de la couleur).

Il est également possible de définir une couleur à partir du modèle RGB, défini à partir de ses trois composantes rouge (*red*), verte (*green*) et bleue (*blue*). Chacune des composantes peut avoir une valeur de 0 à 255. La valeur $(0, 0, 0)$ correspond au noir.

Voici quelques exemples de couleurs ainsi que leur équivalent RGB :

```
Color("black")      # Color(0, 0, 0)
Color("white")      # Color(255, 255, 255)
Color("red")        # Color(255, 0, 0)
Color("green")      # Color(0, 255, 0)
Color("blue")       # Color(0, 0, 255)
Color("yellow")     # Color(255, 255, 0)
```

4.5.3 Les formes géométriques

Avant de commencer à dessiner, il est recommandé d'effacer le contenu de la fenêtre en remplissant celle-ci avec une couleur précisée à l'aide de la fonction `fill(color)`. En effet, même si par défaut, la fenêtre a un arrière-plan noir, on ne devrait pas partir du principe que le canevas sur lequel on dessine soit toujours vide au moment où on y accède.

À titre d'exemple, les instructions

```
screen.fill(Color("white"))
```

et

```
screen.fill(Color(255, 255, 255))
```

ont le même effet net de remplir la fenêtre avec un arrière-fond blanc.

Pour dessiner un **segment** droit, Pygame dispose de la fonction

```
pygame.draw.line(surface, color, start_point, end_point[, width])
```

Les paramètres `start_point` et `end_point` sont des coordonnées du type (x, y).

Exemple :

```
pygame.draw.line(screen, Color("green"), (0, 0), (100, 100), 5)
```

dessinera dans la fenêtre appelée `screen` un segment droit de couleur verte et d'épaisseur 5 pixels du point $(0, 0)$ au point $(100, 100)$. Les extrémités font partie de la ligne.

Remarques :

- ⋆ Lorsqu'on omet l'épaisseur, elle prend la valeur 1 par défaut.
- ⋆ Pour dessiner un seul point (un pixel), on pourra soit dessiner un segment dégénéré dont le point de départ et le point d'arrivée sont identiques, soit recourir à la méthode `set_at` applicable à la fenêtre Pygame :

  ```
  screen.set_at((x, y), Color("yellow"))
  ```

  dessine un point (pixel) jaune à la position (x, y) de la fenêtre.

Pour dessiner un **rectangle**, Pygame dispose de la fonction

```
pygame.draw.rect(surface, color, rectangle_tuple[, width])
```

Le paramètre `rectangle_tuple` est un tuplet de quatre entiers (coordonnées x et y du coin supérieur gauche du rectangle, ainsi que la largeur et la hauteur du rectangle).

Au lieu d'un tuplet de quatre entiers, on peut également créer préalablement un objet de type `pygame.Rect` à l'aide de `pygame.Rect(x, y, width, height)` et passer celui-ci en paramètre. L'avantage d'utiliser un `Rect` est que la classe possède des méthodes pour obtenir facilement les coordonnées d'autres points (p. ex le centre ou le coin inférieur droit).

Exemple :

```
pygame.draw.rect(screen, Color("red"), (20, 20, 250, 100), 2)
```

dessine dans la fenêtre appelée `screen` un rectangle de couleur rouge et d'épaisseur 2 pixels, de largeur 250 pixels, de hauteur 100 pixels et dont le coin supérieur gauche est le point $(20, 20)$.

En utilisant un `Rect`, cela donnerait :

```
my_rect = pygame.Rect(20, 20, 250, 100)
pygame.draw.rect(screen, Color("red"), my_rect, 2)
```

Remarques :

- ⋆ Lorsqu'on omet l'épaisseur ou fournit 0 comme argument, un rectangle plein est dessiné.
- ⋆ Pour dessiner un carré, on indiquera simplement deux valeurs identiques pour la largeur et la hauteur du rectangle.

Enfin, pour dessiner une **ellipse**, Pygame dispose de la fonction

```
pygame.draw.ellipse(surface, color, bounding_rect[, width])
```

Cette fonction dessine l'ellipse inscrite (tangente intérieurement) dans le rectangle `bounding_rect` défini par un tuplet de quatre entiers comme susmentionné. Le rectangle `bounding_rect` peut également être un objet de type `Rect`.

Exemple :

```
pygame.draw.ellipse(screen, Color("blue"), (20, 20, 250, 100), 3)
```

dessine dans la fenêtre appelée `screen` une ellipse de couleur bleue et d'épaisseur 3 pixels, inscrite dans un rectangle de largeur 250 pixels (grand axe de l'ellipse), de hauteur 100 pixels (petit axe de l'ellipse) ; le coin supérieur gauche du rectangle est le point $(20, 20)$ (bien sûr, l'ellipse elle-même ne passe pas par ce point !).

Pour dessiner un **cercle**, on pourrait décrire une ellipse dont le grand axe et le petit axe ont des valeurs identiques. Ou bien, on utilise la fonction

```
pygame.draw.circle(surface, color, center_point, radius[, width])
```

L'argument `center_point` décrit les coordonnées du centre du cercle et `radius` est le rayon (demi-diamètre) exprimé en pixels.

Exemple :

```
pygame.draw.circle(screen, Color("green"), (150, 50), 15, 1)
```

dessine dans la fenêtre appelée `screen` un cercle de couleur verte et d'épaisseur 1 pixel, centré au point $(150, 50)$ et de rayon 15 pixels.

Remarque : Lorsqu'on omet l'épaisseur ou fournit 0 comme argument, une ellipse pleine (ou un disque) est dessinée.

4.5.4 Mise à jour de la surface de dessin

L'ordinateur n'affichera pas le dessin au fur et à mesure de sa construction, car cela causerait des scintillements (*flickering*) désagréables.

Après avoir appelé toutes les fonctions de dessin voulues afin d'obtenir le résultat souhaité à l'écran, il faut **rafraîchir** l'écran. Ceci peut se faire à l'aide de la fonction

```
pygame.display.update()
```

Appelée sans argument, la fonction fait rafraîchir toute la surface de dessin ; appelée avec un paramètre optionnel de type `Rect`, la mise à jour de la surface est limitée à la partie décrite par le rectangle.

Il est indispensable d'écrire une telle instruction, de préférence tout au début ou à la fin de la boucle principale. Si on l'oublie, aucun dessin réalisé par les techniques susmentionnées ne sera visible à l'écran physique !

4.5.5 Affichage de textes

Il existe bien sûr d'autres manières d'afficher un texte dans un programme Pygame que de l'afficher à l'en-tête de la fenêtre en écrivant

```
pygame.display.set_caption("My␣Message")
```

Il est possible d'afficher n'importe quel texte directement sur la surface de dessin grâce au module `pygame.font`.

Pygame ne dessine pas directement un texte sur la surface de dessin existante. Une image du texte est d'abord créée sur une nouvelle surface de dessin. Celle-ci est ensuite copiée sur la surface de dessin existante grâce à la méthode

```
blit(source, dest[, area, special_flags])
```

Nous allons utiliser dans la suite les méthodes `pygame.font.SysFont` et `font.render` :

- ⋆ `pygame.font.SysFont(name, size[, bold, italic])` crée un objet de type `Font` à partir des polices disponibles dans le système d'exploitation. Les paramètres `bold` et `italic` ont la valeur `False` par défaut.
- ⋆ `font.render(text, antialias, color[, background])` dessine le texte sur une nouvelle surface de dessin. La méthode retourne la surface créée. Le paramètre `background` a la valeur `None` par défaut.

Le code suivant écrit le texte « Hello World ! » au milieu de l'écran en utilisant la police Palatino, la taille 72 points et la couleur rose.

```
import pygame, sys
from pygame.locals import *

pygame.init()
w = 640
h = 480
size = (w, h)
screen = pygame.display.set_mode(size)
screen.fill(Color("white"))

font = pygame.font.SysFont("Palatino", 72)
text = font.render("Hello World!", True, Color("pink"))
screen.blit(text, ((w - text.get_width()) // 2, (h - text.get_height()) // 2))
pygame.display.update()

done = False
while not done:
    for event in pygame.event.get():
        if event.type == QUIT:
            done = True

pygame.quit()
sys.exit()
```

4.6 Gestion du temps

La fréquence de rafraîchissement (fréquence d'image), exprimée en Hertz [Hz = s^{-1}], définit le nombre d'images qu'un programme affiche par seconde (*frames per second, FPS*). Une fréquence trop basse provoquera une animation saccadée, une fréquence trop élevée un programme qui s'exécute si rapidement que le joueur n'a pas le temps de réagir.

Un objet `pygame.time.Clock` nous permet de faire en sorte que notre programme tourne avec une fréquence maximale donnée. Cet objet ajoutera des petites pauses lors de l'exécution de la boucle principale afin que le programme ne tourne pas trop vite, peu importe la vitesse de l'ordinateur sur lequel il tourne.

L'objet de type `Clock` doit être créé avant la boucle principale moyennant l'instruction suivante :

```
fps_clock = pygame.time.Clock()
```

À la fin de la boucle principale, de préférence après l'appel de `pygame.display.update()`, on appelle la méthode `tick(framerate)` de l'objet. Le programme insère alors des pauses afin que le programme tourne à la fréquence voulue. Par exemple :

```
fps_clock.tick(30)
```

Afin de voir l'influence de la fréquence sur le programme, il est intéressant de définir une constante `FPS` au début du programme et de remplacer l'appel précédent par

```
fps_clock.tick(FPS)
```

Le module `time` de la bibliothèque `pygame` possède également les fonctions intéressantes suivantes :

- ⋆ `pygame.time.delay(my_delay)` crée une attente dont la durée est exprimée en ms ;
- ⋆ `pygame.time.get_ticks()` renvoie le temps passé (exprimé en ms) depuis l'initialisation `pygame.init()`.

4.7 Résumé : structure d'un programme Pygame

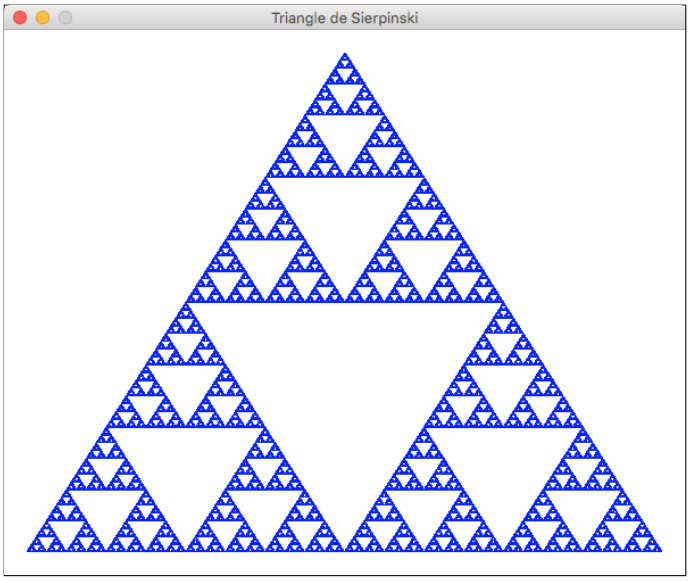
Voici le squelette revisité du début du chapitre : de nombreux programmes Pygame peuvent être construits à partir de la structure suivante.

```
import pygame, sys
from pygame.locals import *

pygame.init()
size = (400, 300)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Hello world!")
screen.fill(Color("white"))

FPS = 30
clock = pygame.time.Clock()

done = False
while not done:
    for event in pygame.event.get():
        if event.type == QUIT:
            done = True
        # elif event.type == # check for user interaction
    # do some cool stuff
    pygame.display.update()
    clock.tick(FPS)

pygame.quit()
sys.exit()
```

4.8 Exercices

Exercice 4.1 – Triangle de Sierpinski

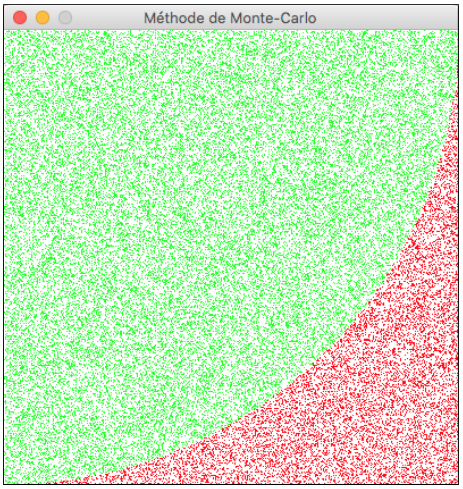
Créer le programme Pygame suivant :



- ⋆ le programme tourne dans une fenêtre de taille 600×480 pixels avec un arrière-fond blanc ;
- ⋆ le titre de la fenêtre est « Triangle de Sierpinski » ;
- ⋆ on se donne un triangle (ABC) tel que les trois sommets se trouvent à l'intérieur de la fenêtre, ainsi qu'un point P_0 de la fenêtre aux coordonnées aléatoires ;
- ⋆ dans la boucle principale, le programme calcule le point P_i tel que P_i soit le milieu du segment $[P_{i-1}, S]$, où S est un des trois sommets du triangle choisi chaque fois au hasard. Les points P_i (à partir de $i > 9$) sont dessinés en bleu ;
- ⋆ il n'est pas nécessaire de faire tourner le programme à une vitesse particulière ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture.

Exercice 4.2 – Méthode de Monte-Carlo

Créer un programme Pygame qui calcule une valeur approchée de π par le hasard (méthode de Monte-Carlo).



À cette fin on calcule une suite de points du carré unité et on calcule le rapport entre le nombre de points situés dans le quart de disque unité et le nombre total de points. Ce rapport se rapproche de $\frac{\pi}{4}$ qui est l'aire du quart de disque en question.

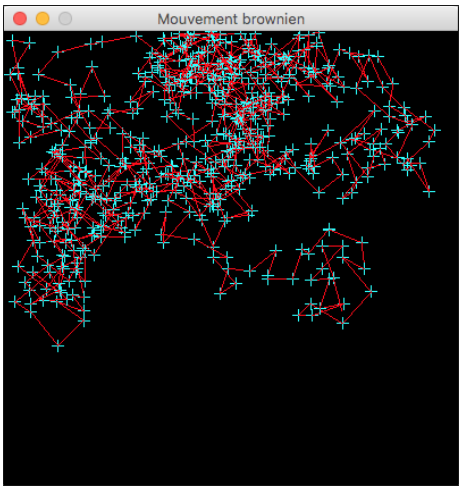
- ⋆ Le programme tourne dans une fenêtre de taille 400×400 pixels avec un arrière-fond blanc ;
- ⋆ le titre de la fenêtre est « Méthode de Monte-Carlo » ;
- ⋆ dans la boucle principale, le programme choisit au hasard un point du carré unité ; les points inscrits dans le quart de disque unité sont dessinés en vert, les autres points en rouge ;
- ⋆ il n'est pas nécessaire de faire tourner le programme à une vitesse particulière ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture. Il affiche ensuite dans la console le nombre total de points dessinés, le nombre de points inscrits (points verts), l'approximation de π qui en résulte et la différence avec la valeur réelle de π.

Exercice 4.3 – Mouvement brownien *

Le mouvement brownien (ou processus de Wiener) est la description mathématique du mouvement aléatoire d'une particule en apesanteur (molécule de gaz, particules dans un liquide, ...).

La description physique la plus élémentaire du phénomène est la suivante :

- ⋆ entre deux chocs, la particule se déplace en ligne droite avec une vitesse constante ;
- ⋆ la particule est accélérée ou réfléchie lorsqu'elle rencontre une molécule de fluide ou une paroi.



Écrire un programme Pygame simulant le mouvement brownien selon les indications suivantes :

- ⋆ le programme tourne dans une fenêtre de taille 400×400 pixels avec un arrière-fond noir ;
- ⋆ le titre de la fenêtre est « Mouvement brownien » ;
- ⋆ au début du programme la particule est placée à des coordonnées aléatoires à l'intérieur de la fenêtre ;

- ⋆ la position actuelle de la particule est marquée par une croix symétrique de 11 pixels, en couleur turquoise (`"cyan"`) ;
- ⋆ chaque mouvement est simulé par un déplacement en x et y d'amplitude aléatoire entre 0 et 30 pixels ;
- ⋆ si, lors d'un mouvement, la particule risque de sortir du dessin, alors elle est réfléchie ;
- ⋆ le mouvement est affiché par un segment rouge ;
- ⋆ le programme tourne à 20 FPS ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture.

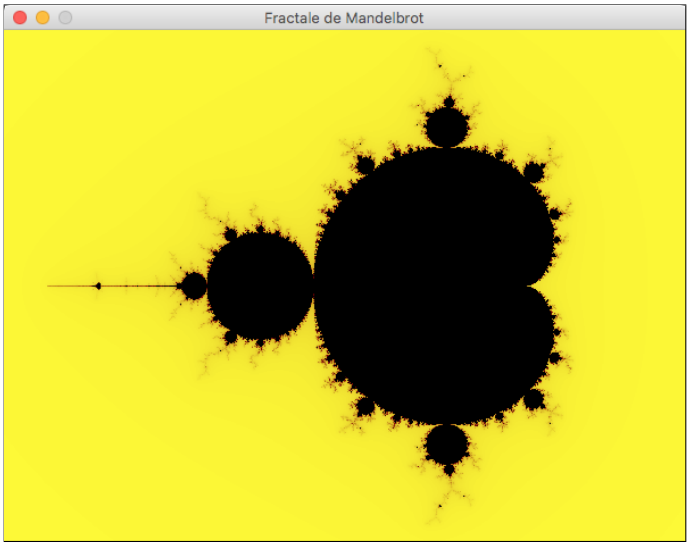
Exercice 4.4 – Fractale de Mandelbrot * L'ensemble de Mandelbrot $\mathcal{M}$ est une fractale définie comme l'ensemble des points d'affixe $c \in \mathbb{C}$ du plan de Gauß pour lesquels la suite définie par la récurrence

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \quad \text{pour } n \in \mathbb{N} \end{cases}$$

ne tend pas vers l'infini complexe (c'est-à-dire le module ne tend pas vers l'infini).

Cet ensemble définit, dans le plan complexe, une surface finie à bord infini ! Le bord est en effet brisé/fracturé (d'où le nom *fractale*) à toute échelle. Afin de pouvoir calculer et afficher en un temps fini cet ensemble, on se donne un nombre maximal d'itérations `maxiterations` et un rayon maximal `maxradius`.

Si pour un point d'affixe c, on a $|z_{\texttt{maxiterations}}| < \texttt{maxradius}$, on admet que $c \in \mathcal{M}$.



Créer un programme qui calcule et affiche cet ensemble selon les indications suivantes :

- ⋆ le programme tourne dans une fenêtre de taille 600×450 pixels ;
- ⋆ le titre de la fenêtre est « Fractale de Mandelbrot » ;
- ⋆ on se donne les valeurs

```
xmin = −2.2
xmax = 1.0
ymin = −1.2
ymax = 1.2
```

représentant la partie du plan pour laquelle l'ensemble est initialement dessiné ainsi qu'un nombre maximal d'itérations `maxiterations = 255` et un rayon maximal `maxradius = 10` ;

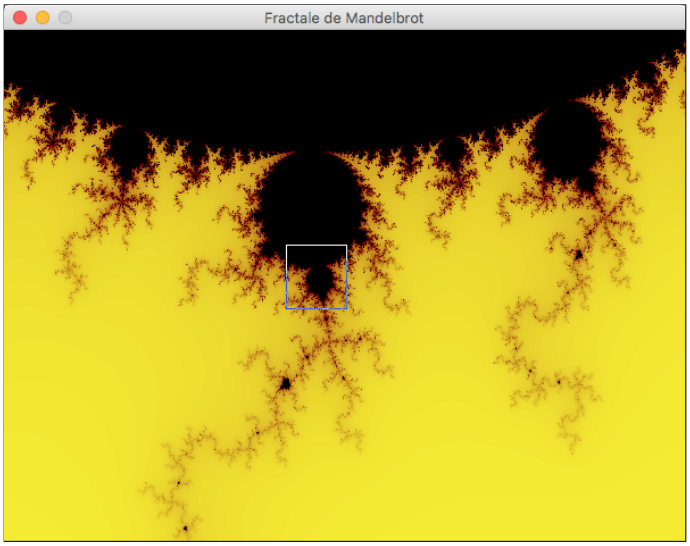
- ⋆ le programme affiche la partie de l'ensemble de Mandelbrot relative à la partie du plan concernée. La couleur d'un point sera noire s'il appartient à l'ensemble, sinon sa couleur sera égal à la couleur RGB

```
color = (255 - iterations, max(255 - 2 * iterations, 0), 0)
```

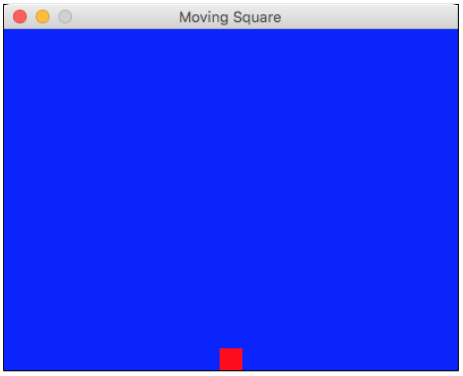
(la couleur passe doucement du jaune vers le rouge foncé, puis vers le noir, en fonction du nombre d'itérations) ;

- ⋆ comme on ne veut pas attendre trop longtemps, mais quand même voir la progression de l'affichage, le dessin est mis à jour après chaque ligne dessinée (mais non pas après chaque point !) ;
- ⋆ le programme se termine proprement lorsque le dessin est complet et que l'utilisateur clique ensuite sur le bouton de fermeture.

** Voici une version du programme plus difficile : lorsque le dessin est complet, l'utilisateur peut tracer un rectangle de zoom-in en reliant deux sommets diagonalement opposés de ce rectangle avec le bouton gauche de la souris enfoncé. Ce rectangle doit être visible pendant les manipulations de l'utilisateur (déplacement de la souris avec bouton gauche enfoncé). Lorsque l'utilisateur relâche le bouton gauche et qu'il a effectivement décrit avec la souris un rectangle valide (non dégénéré en un point ou un segment horizontal/vertical), un zoom est effectué, c'est-à-dire les valeurs de `xmin`, `xmax`, `ymin` et `ymax` sont mises à jour en fonction des coordonnées des sommets du rectangle désigné par l'utilisateur, et l'ensemble $\mathcal{M}$ est redessiné.

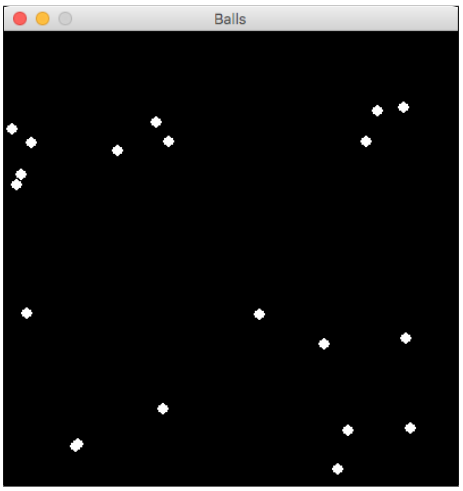


Exercice 4.5 Écrire un programme Pygame d'après les indications suivantes :



- ⋆ le programme tourne dans une fenêtre de taille 400×300 pixels, avec un arrière-fond bleu ;
- ⋆ le titre de la fenêtre est « Moving Square » ;
- ⋆ au lancement du programme, un carré plein de côté 20 pixels et de couleur rouge se situe au milieu du bord inférieur de la fenêtre ;
- ⋆ le programme réagit aux touches de direction du clavier : le carré se déplace de 5 pixels dans la bonne direction sans dépasser le bord de la fenêtre ; on traitera de façon intelligente les cas spéciaux lorsque plusieurs touches sont enfoncées en même temps ;
- ⋆ le programme tourne à 20 FPS ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture.

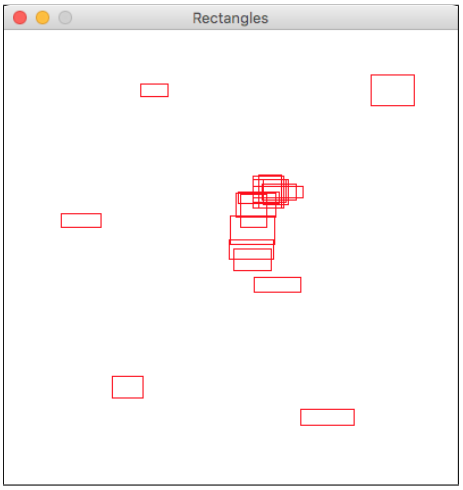
Exercice 4.6 Écrire un programme Pygame selon les instructions suivantes :



- ⋆ le programme tourne dans une fenêtre de taille 400×400 pixels avec un arrière-fond noir ;
- ⋆ le titre de la fenêtre est « Balls » ;
- ⋆ au début, le programme génère une liste `ball_list` contenant 20 coordonnées correspondant aux centres de 20 balles de rayon 5 pixels chacune. Les coordonnés sont choisies de manière aléatoire, mais de façon à ce que chacune des balles se trouve à l'intérieur de la fenêtre ;
- ⋆ définir une fonction `verify_mouse` à un paramètre. Cette fonction retourne la première balle de la liste pour laquelle les coordonnées passées comme argument se trouvent à l'intérieur de la balle, et `None` sinon ;

- ⋆ le programme réagit à un clic gauche de la souris : il vérifie si les coordonnées de la souris se trouvent à l'intérieur d'une des balles et supprime dans ce cas la balle correspondante de la liste ;
- ⋆ le programme dessine les balles sous forme de disques blancs de rayon 5 pixels ;
- ⋆ il n'est pas nécessaire de faire tourner le programme à une vitesse particulière ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture.

Exercice 4.7 * Créer un programme qui gère des rectangles d'après les indications suivantes :



- ⋆ le programme tourne dans une fenêtre de taille 400×400 pixels avec un arrière-fond blanc ;
- ⋆ le titre de la fenêtre est « Rectangles » ;
- ⋆ il n'est pas nécessaire de faire tourner le programme à une vitesse particulière ;
- ⋆ un clic sur le bouton gauche de la souris ajoute à la liste `rectangles` (vide au début du programme) les coordonnées d'un rectangle centré autour de la position de la souris et de taille aléatoire (largeur de 20 à 50 points, hauteur de 10 à 30 points) ;
- ⋆ un clic sur le bouton droit de la souris supprime de la liste `rectangles` tous les rectangles à la position de la souris, s'il y en a ;
- ⋆ les rectangles de la liste `rectangles` sont dessinés en rouge et ont une épaisseur de 1 pixel ;
- ⋆ le programme se termine proprement lorsqu'on clique sur le bouton de fermeture.

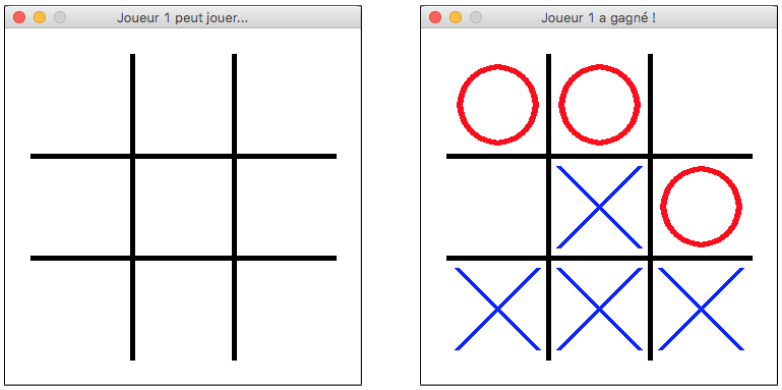
Exercice 4.8 Écrire une fonction `flytext` à deux paramètres `msg` (le message à afficher) et `duration` (la durée en secondes). La fonction fait défiler le texte de gauche à droite et de droite à gauche ; la durée de chacun des deux passages correspond à `duration`.

Le programme tourne dans une fenêtre de taille 600×200 et l'écran est rafraîchi 30 fois par seconde.

Exercice 4.9 – Tic-tac-toe *

Écrire un programme qui assure l'arbitrage du jeu de Tic-tac-toe entre deux joueurs humains.

Règles du jeu : Deux joueurs s'affrontent. Ils doivent remplir chacun à leur tour une case d'une grille 3×3, vide au départ, avec le symbole qui leur est attribué : une croix bleue ou un cercle rouge. Le gagnant est celui qui arrive à aligner trois symboles identiques, horizontalement, verticalement ou en diagonale. Lorsque la grille est remplie et qu'aucun joueur n'a su aligner trois symboles identiques, la partie est dite « remise » (nulle).



Indications :

1. La fenêtre Pygame mesure 350×350 pixels. Chaque case de la table de jeu mesure 100×100 pixels.
2. La fonction auxiliaire `draw_coin` dessine soit une croix bleue soit un cercle rouge, de dimensions 80×80 et d'épaisseur 5, au milieu de la case choisie.
3. La fonction `draw_board` dessine la table de jeu, c'est-à-dire 4 segments noirs d'épaisseur 5 pixels sur fond blanc.
4. La fonction `victory` analyse si le joueur passé comme argument a trois symboles alignés dans la table de jeu et renvoie comme résultat `True` ou `False`.
5. La boucle principale du jeu permet aux joueurs de jouer à tour de rôle, jusqu'à ce que la partie se termine. À chaque instant, l'un des cinq messages suivants est affiché à l'en-tête de la fenêtre : « Joueur 1 peut jouer… », « Joueur 2 peut jouer… », « Joueur 1 a gagné ! », « Joueur 2 a gagné ! », « Partie remise. »
6. Suite à l'événement `QUIT`, le programme ferme sa fenêtre et se termine.

Exercice 4.10 – Alien Invasion *

Écrire un programme simulant une invasion de vaisseaux extra-terrestres que le joueur tentera d'anéantir. L'écran Pygame a comme dimensions 500×600 pixels et est rafraîchi 60 fois par seconde.

1. La classe `Alien` représente un vaisseau extra-terrestre.
   (a) Le constructeur initialise l'attribut `x` à une valeur aléatoire entre 12 et 487, l'attribut `y` à -12, l'attribut `color` à une couleur RGB dont le canal R prend une valeur aléatoire entre 128 et 255, le canal G prend une valeur aléatoire quelconque et le canal B est nul. La vitesse `v` est initialisée à l'argument optionnel ; lorsque l'argument n'est pas fourni, la vitesse est initialisée à une valeur aléatoire entre 100 et 199. La composante verticale de la vitesse `vy` est initialisée à une valeur aléatoire comprise entre 20 et `v` $- 1$, et la composante horizontale en est déduite, de façon à ce que `v` représente la norme du vecteur-vitesse (`vx`, `vy`).

(b) La méthode `draw` dessine le vaisseau à l'écran Pygame fourni comme argument, sous forme de disque de rayon 12 pixels et de centre (`x`, `y`).

(c) La méthode `move` déplace le vaisseau à la vitesse de `v` pixels/seconde. Lorsque le vaisseau touche ainsi le bord gauche ou le bord droit de l'écran, la composante horizontale du vecteur-vitesse est remplacée par son opposé. Lorsque le vaisseau a complètement quitté le bord inférieur de l'écran, son ordonnée `y` est réinitialisée à -12.

(d) La méthode `collision` reçoit un objet de type `Starship` comme argument et teste s'il y a une collision entre `self` et ce vaisseau.

(e) La méthode `hit` reçoit un objet de type `Torpedo` comme argument et teste s'il y a une collision entre `self` et cette torpille.

2. La classe `Starship` représente le vaisseau piloté par le joueur.

   (a) Le constructeur initialise l'attribut `x` à la valeur du milieu de l'écran, et l'attribut `y` à 30 pixels au-dessus du bord inférieur de l'écran. Ces coordonnées représentent le centre de gravité du vaisseau.

   (b) La méthode `draw` dessine le vaisseau à l'écran Pygame fourni comme argument, sous forme d'un carré rempli blanc dont le côté mesure 41 pixels.

   (c) La méthode `move` reçoit un argument `dx` et déplace le vaisseau de `dx` pixels vers la droite (ou vers la gauche si `dx` < 0). Le vaisseau ne pourra pas quitter l'écran, mais devra y rester entièrement visible.

3. La classe `Torpedo` représente une torpille lancée par le joueur.

   (a) Le constructeur initialise les attributs `x` et `y` au milieu du bord supérieur du vaisseau `Starship` fourni comme argument. Ces coordonnées représentent le centre de gravité de la torpille.

   (b) La méthode `draw` dessine la torpille à l'écran Pygame fourni comme argument, sous forme d'un rectangle rempli de largeur 3 pixels et de hauteur 11 pixels, de couleur `"cyan"`.

   (c) La méthode `move` fait monter la torpille vers le haut de 400 pixels/seconde.

4. La fonction auxiliaire `final_message` affiche le message fourni comme argument à l'en-tête de la fenêtre Pygame et attend jusqu'à ce que le joueur clique à l'intérieur de la fenêtre ou la ferme.

5. Le programme principal initialise la fenêtre Pygame, une armée de 10 vaisseaux `Alien` et un vaisseau `Starship`.

   (a) Le joueur peut mettre fin au jeu en cliquant dans la fenêtre.

   (b) Le joueur peut déplacer son vaisseau à l'aide des touches de curseur, vers la gauche ou vers la droite, avec auto-repeat, à une vitesse de 200 pixels/seconde.

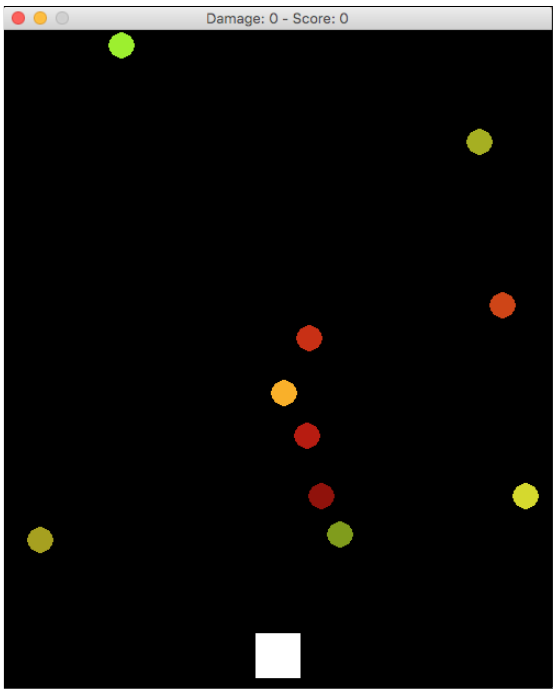
   (c) Le joueur peut lancer une torpille en frappant la touche « espace » (sans auto-repeat), lorsque le nombre de torpilles finalement visibles à l'écran ne dépasse pas 6.

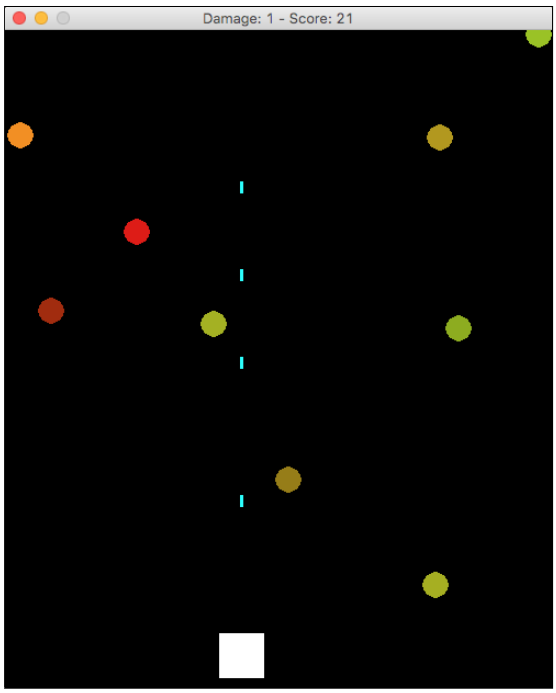
   (d) Une torpille qui a quitté entièrement l'écran par le bord supérieur est anéantie.

   (e) Lorsqu'un vaisseau `Alien` est touché par une torpille, ce vaisseau et cette torpille sont anéantis. Un nouveau vaisseau `Alien` est créé par l'appel du constructeur, et sa vitesse sera celle du vaisseau détruit, augmentée de 10 %.

(f) Lorsqu'un vaisseau `Alien` touche le vaisseau du joueur, le vaisseau `Alien` est anéanti, et le compteur des dégâts est augmenté d'une unité. Lorsque ce compteur atteint la valeur 3, le jeu se termine.

(g) Pendant le jeu, l'en-tête de la fenêtre Pygame affiche tout le temps le texte suivant : « Damage : n – Score : x », où n est le compteur des dégâts déjà enregistrés (0, 1 ou 2), et x est le nombre de vaisseaux `Alien` détruits par des torpilles. Lorsque le jeu se termine, le message final est : « GAME OVER – Score : x ».





4.9 Exercices de synthèse *

Les exercices de cette section sont tous facultatifs, mais comme leur niveau de difficulté correspond plus ou moins à celui des questions de l'examen de fin d'études secondaires, il est hautement recommandé d'étudier au moins une partie d'entre eux.

Nous regroupons ici des exercices qui mélangent différentes techniques de programmation vues dans ce chapitre-ci et dans les chapitres précédents. En effet, tous les exercices Pygame de la section précédente, sauf le dernier, servaient à appliquer directement les notions vues dans ce chapitre. Nous dépassons maintenant ce cadre et proposons quelques exercices de synthèse.

Les énoncés sont formatés comme de véritables énoncés de devoirs en classe ; ainsi le coloriage des codes-sources Python, tel qu'appliqué ailleurs dans ce manuel, est désactivé dans cette section.

Au-delà des exercices de cette section, nous renvoyons l'élève intéressé aux questions d'examen antérieures ; les énoncés y relatifs sont distribués séparément sous forme de fichiers PDF.

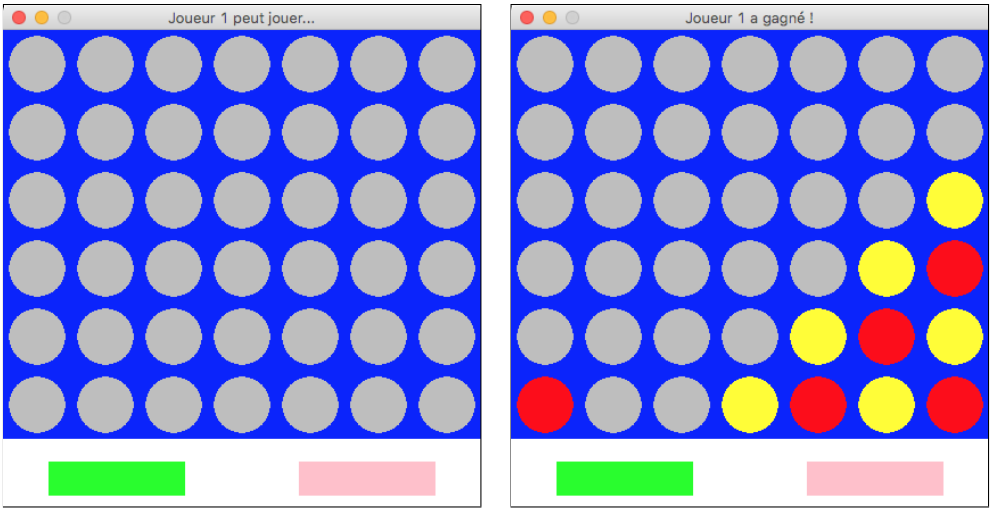
Exercice 4.11 – Le jeu de « Puissance 4 » (*Vier gewinnt*) *

Dans le jeu « Puissance 4 », deux adversaires essaient chacun de réaliser un alignement horizontal, vertical ou diagonal de quatre jetons de leur couleur respective.

La table de jeu consiste en un plateau rectangulaire **vertical** de 7 colonnes pouvant contenir chacune, de bas en haut, 6 jetons. Ce plateau est vide en début de jeu (voir la capture d'écran à gauche). Le premier joueur dispose de jetons de couleur jaune et son adversaire de jetons de

couleur rouge. Chacun pose à tour de rôle un jeton sur la colonne de son choix, sous condition que cette colonne ne soit pas déjà remplie jusqu'au 6e niveau. Soumis à la pesanteur, le nouveau jeton s'immobilise à la position libre **la plus basse** de la colonne.

En réalisant un alignement de 4 jetons, un joueur met fin à la partie, qu'il gagne alors (voir la capture d'écran à droite : le premier joueur vient d'aligner 4 jetons jaunes en diagonale). Si toute la table de jeu est remplie de jetons sans qu'il y ait d'alignement, la partie est déclarée « remise ».



L'écran Pygame comprend :

- ⋆ une table de jeu bleue, de 420×360 pixels ; chacun des 7×6 emplacements de la table de jeu mesure donc 60×60 pixels ;
- ⋆ l'en-tête de la fenêtre contient toujours l'un des messages suivants : « Joueur n peut jouer… », « Joueur n a gagné ! », « Partie remise ! » ;
- ⋆ un bouton (rectangle) vert de 120×30 pixels, qui permet de commencer une nouvelle partie de jeu : la table de jeu est réinitialisée et le premier joueur pourra recommencer à jouer ;
- ⋆ un bouton (rectangle) rose (`pink`) de 120×30 pixels, qui permet d'annuler le dernier coup joué s'il existe ; la table de jeu est mise à jour et le joueur dont le coup vient d'être annulé pourra jouer à nouveau.

Écrire un programme Pygame en suivant ces instructions :

1. Écrire une fonction `draw_coin` qui accepte comme arguments l'écran Pygame, le numéro de colonne, le numéro de ligne et un numéro de couleur ($0 =$ jaune, $1 =$ rouge, $2 =$ gris), et qui dessine un disque de diamètre 50 pixels au bon emplacement dans la couleur indiquée.
2. Écrire une fonction `draw_board` qui dessine la table de jeu, y compris les 42 emplacements sous forme de disques gris, et les deux rectangles vert et rose sur fond blanc, semblables à la capture d'écran à gauche.
3. Écrire une fonction `victory` qui détermine si le joueur actuel a réalisé un alignement de 4 jetons et renvoie comme résultat un booléen.
4. Écrire le programme principal, y compris l'arbitrage du jeu :

   Un joueur peut placer un nouveau jeton en cliquant avec la souris sur la table de jeu : le programme détecte la colonne choisie à l'aide de l'abscisse du pointeur de la souris,

l'ordonnée pouvant être ignorée. Lorsqu'il est possible d'effectuer un coup dans la colonne proposée par le joueur, le programme exécute le coup, c'est-à-dire il dessine le jeton et affiche à l'en-tête de la fenêtre le message approprié (« Joueur n peut jouer… », « Joueur n a gagné ! » ou « Partie remise ! »).

Lorsque le joueur clique avec la souris sur le rectangle vert, le jeu est réinitialisé.

Lorsque le joueur clique avec la souris sur le rectangle rose, le dernier coup joué (s'il existe) est annulé. Il est permis de cliquer à plusieurs reprises sur le rectangle rose et d'annuler ainsi plusieurs coups joués dans l'ordre inverse, jusqu'à la position initiale du jeu.

Lorsque le jeu est fini, soit par la victoire d'un joueur, soit parce que la partie est remise, le programme attend jusqu'à ce que l'utilisateur clique sur le bouton de fermeture de la fenêtre, avant se terminer.

Exercice 4.12 – Le « Jeu de la vie » **

Le « Jeu de la vie » est un automate cellulaire imaginé par John Conway en 1970. Il se déroule sur une grille à deux dimensions, théoriquement infinie (mais de longueur et de largeur finies dans la pratique), dont les cases carrées, qu'on appelle des *cellules*, peuvent prendre deux états distincts : *vivantes* ou *mortes*.

Chaque cellule à l'intérieur de la grille possède 8 voisines : celles qui l'entourent. Lorsque la grille est finie, les cellules de son bord ne possèdent que 5 voisines, et les 4 cellules aux coins de la grille n'en possèdent que 3.

À chaque étape (appelée *génération*), l'évolution d'une cellule est entièrement déterminée par l'état de ses voisines de la façon suivante :

- ⋆ Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
- ⋆ Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Dans notre programme, la grille consiste en un plateau carré de 100×75 cellules, dont toutes les cellules sont mortes initialement. L'utilisateur pourra modifier l'état des cellules de la grille, c'est-à-dire les rendre vivantes ou mortes. Il dispose en outre de trois boutons servant à faire évoluer la population d'une ou de plusieurs générations ou à réinitialiser la grille.

L'écran Pygame comprend :

- ⋆ une grille carrée noire de 800×600 pixels ; chacune des 100×75 cases contenant une cellule mesure donc 8×8 pixels ;
- ⋆ l'en-tête de la fenêtre contient toujours le message « Génération x ; population y », où x est le numéro de la génération actuelle (0 au début du jeu ou après une réinitialisation) et y est le nombre de cellules actuellement vivantes ;
- ⋆ au-dessous de la grille : une zone blanche de hauteur 60 pixels, contenant les trois boutons suivants ;
- ⋆ un bouton (rectangle) vert de 200×30 pixels, pour faire évoluer la population d'une seule génération ;
- ⋆ un bouton (rectangle) rouge de 200×30 pixels, pour faire évoluer rapidement la population de plusieurs générations successives ;
- ⋆ un bouton (rectangle) bleu de 200×30 pixels, pour réinitialiser la grille (toutes les cellules sont alors mortes).

Écrire un programme Pygame, en suivant ces instructions :

1. Écrire une fonction `draw_cell` qui accepte comme arguments l'écran Pygame, le numéro de colonne, le numéro de ligne et la couleur (jaune pour les cellules vivantes et noire pour les cellules mortes), et qui dessine un carré de côté 6 pixels au centre de la case précisée dans la couleur indiquée.

2. Écrire une fonction `draw_board` qui accepte comme arguments l'écran Pygame et la variable contenant la population actuelle, et qui (re)dessine la grille avec toutes ses cellules vivantes et mortes.

3. Écrire une fonction `draw_buttons` qui accepte comme argument l'écran Pygame et qui y dessine les trois boutons vert, rouge et bleu sur fond blanc.

4. Écrire une fonction `evolution` qui accepte comme arguments la variable contenant la population actuelle et qui la remplace par la population de la génération suivante. Il est recommandé de compter d'abord pour chaque cellule ses cellules voisines vivantes et de stocker ces nombres dans une variable auxiliaire. Ensuite, on repasse en revue toutes les cellules de l'ancienne population et en fonction du contenu de la variable auxiliaire on modifie l'état des cellules vivantes ou mortes directement dans la variable reçue comme argument.

5. Écrire une fonction `invert_cells` qui reçoit comme arguments l'écran Pygame, la variable contenant la population actuelle et les numéros de colonne et de ligne de la case cliquée. Elle attend jusqu'à ce que le bouton de la souris soit relâché et détermine les numéros de colonne et de ligne de la case pointée en ce moment. Les deux couples de coordonnées déterminent alors les sommets diagonalement opposés d'un rectangle, éventuellement réduit à une seule case (p. ex. lors d'un bref clic de souris). Pour toutes les cases à l'intérieur de ce rectangle (bord compris), l'état des cellules est **inversé** : une cellule morte devient vivante, et une cellule vivante devient morte. La fonction actualise l'écran et la population actuelle passée comme argument.

   Attention ! Lorsque le bouton est relâché sur la zone blanche avec les boutons, il faut remplacer l'ordonnée du pointeur par celle de la ligne la plus basse de la grille, afin de ne pas placer des cellules à l'extérieur de la grille.

6. Écrire une fonction `green_button` qui accepte comme arguments l'écran Pygame et la variable contenant la population actuelle. Elle fait évoluer la population d'une génération, grâce à la fonction précédente, augmente le compteur des générations d'une unité et affiche la nouvelle population à l'écran.

7. Écrire une fonction `red_button` qui accepte comme arguments l'écran Pygame et la variable contenant la population actuelle. **Aussi longtemps** que le bouton de la souris n'est pas relâché, elle (re)fait évoluer la population d'une génération, augmente le compteur des générations d'une unité et affiche la nouvelle population à l'écran.

8. Écrire le programme principal :

   L'utilisateur peut changer l'état d'une cellule en cliquant avec la souris sur la grille : le programme détecte la colonne et la ligne choisies à l'aide des coordonnées du pointeur de la souris et appelle la fonction `invert_cells`.

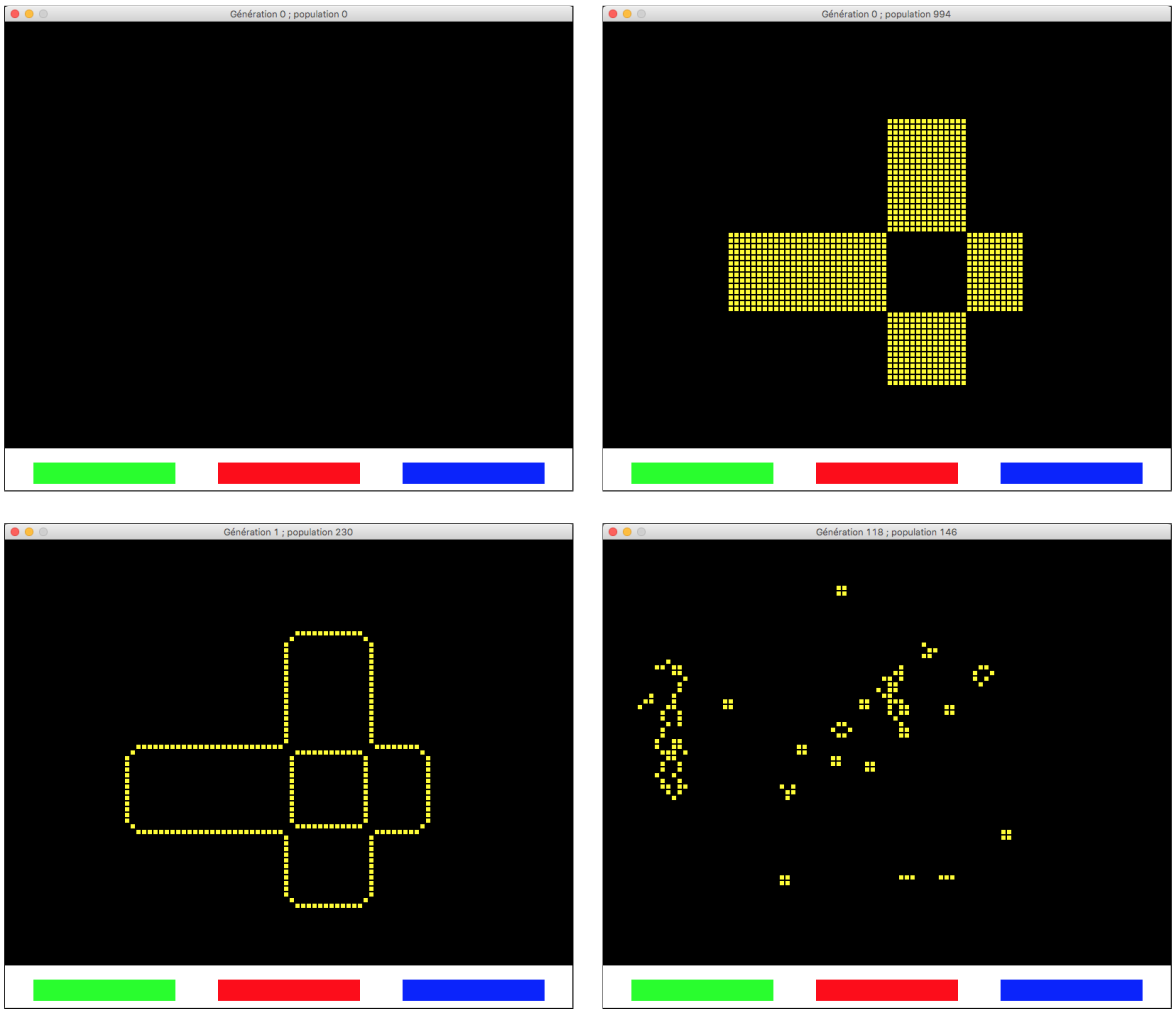
   Lorsque l'utilisateur clique sur le rectangle vert, la fonction `green_button` est appelée.

   Lorsque l'utilisateur clique sur le rectangle rouge, la fonction `red_button` est appelée.

   Lorsque l'utilisateur clique sur le rectangle bleu, la population actuelle est réinitialisée (toutes les cellules sont alors mortes), le compteur des générations est remis à 0 et l'affichage est actualisé.

   L'utilisateur peut terminer le programme en fermant la fenêtre Pygame.

Captures d'écran : (1) après le démarrage du programme (ou un clic sur le bouton bleu) ; (2) après avoir dessiné deux blocs rectangulaires de cellules ; (3) après un clic sur le bouton vert ; (4) après un clic de quelques dixièmes de seconde sur le bouton rouge.



Exercice 4.13 – Les tours de Hanoï (version console) *

Écrire un programme qui implémente les *tours de Hanoï*.

Une tour de Hanoï, d'ordre $n \in \mathbb{N}^*$, consiste en n disques entassés de diamètres $n, \ldots, 2, 1$ (du bas vers le haut). Il faut déplacer ces disques de l'emplacement de départ (à gauche) vers l'emplacement d'arrivée (à droite) en utilisant un emplacement auxiliaire (au milieu), et ceci en un minimum d'étapes. On ne peut déplacer plus d'un disque à la fois, et on ne peut placer un disque que sur un autre disque de diamètre **supérieur**, ou à un emplacement vide.

1. Écrire une fonction `move` qui réalise (p. ex. par des appels récursifs) le déplacement des n disques en $2^n - 1$ étapes.

2. Écrire une fonction auxiliaire `show_towers` qui affiche les trois tours sous forme de trois listes à l'écran. Par exemple, pour $n = 3$, la tour initiale sera affichée sous la forme `[3, 2, 1]`.

3. Écrire le programme principal qui demande à l'utilisateur d'entrer n, exécute l'algorithme pour déplacer toutes les tours de la gauche vers la droite, et affiche le nombre d'étapes nécessaires.

Exemple d'exécution :

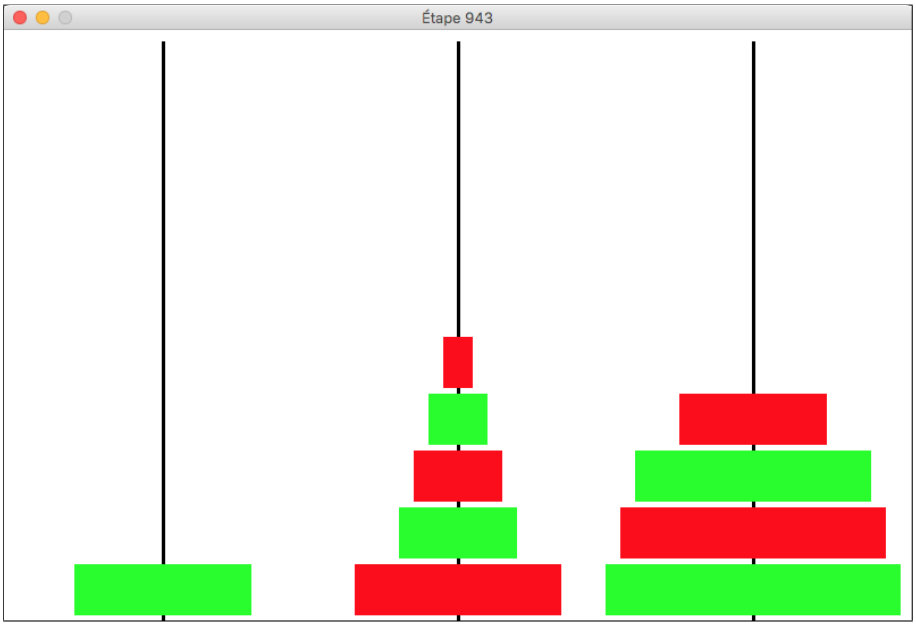
```
Entrez le nombre de tours : 3
[3, 2, 1] - [] - [] au départ
[3, 2] - [] - [1]
[3] - [2] - [1]
[3] - [2, 1] - []
[] - [2, 1] - [3]
[1] - [2] - [3]
[1] - [] - [3, 2]
[] - [] - [3, 2, 1]
Terminé après 7 mouvements.
```

Exercice 4.14 – Les tours de Hanoï (version Pygame) *

Modifier le programme précédent pour que l'affichage des tours ne se fasse plus dans la console, mais dans une fenêtre Pygame de taille 800×520 pixels. Chaque tour est représentée par un rectangle plein, de couleur rouge ou verte (suivant la parité de l'indice du disque), de hauteur 45 pixels et de largeur égale à un multiple de 26 pixels.

Le programme doit fonctionner pour les valeurs de $n = 1, \ldots, 10$. La vitesse de succession des étapes sera telle que l'algorithme se termine après 10 secondes (sauf si l'ordinateur utilisé est trop lent pour afficher une centaine d'étapes par seconde).

Capture d'écran pour $n = 10$, à l'étape 943 (sur 1023).



Exercice 4.15 – Les doubles tours de Hanoï (version console) **

Écrire un programme qui implémente les *doubles tours de Hanoï*.

Une double tour de Hanoï, d'ordre $n \in \mathbb{N}^*$, consiste en $2n$ disques entassés de diamètres $n, n, \ldots, 2, 2, 1, 1$ (du bas vers le haut). Il faut déplacer ces disques de l'emplacement de départ (à gauche) vers l'emplacement d'arrivée (à droite) en utilisant un emplacement auxiliaire (au milieu), et ceci en un minimum d'étapes. On ne peut déplacer plus d'un disque à la fois, et on ne peut placer un disque que sur un autre disque de diamètre **supérieur ou égal**, ou à un emplacement vide.

1. Écrire une fonction `move2` qui réalise (p. ex. par des appels récursifs) le déplacement des $2n$ disques en $2^{n+1} - 2$ étapes. Les disques de même diamètre peuvent être placés l'un

sur l'autre dans **n'importe quel ordre**, aussi bien pendant les déplacements qu'à la configuration finale.

2. Écrire une fonction `move2_ordered` qui réalise (p. ex. par des appels de la fonction `move2` et/ou des appels récursifs) le déplacement des $2n$ disques en $2^{n+2} - 5$ étapes. Les disques de même diamètre peuvent être placés l'un sur l'autre dans n'importe quel ordre pendant les déplacements, mais ils doivent avoir retrouvé l'**ordre initial** à la configuration finale.

3. Écrire une fonction auxiliaire `show_towers` qui affiche les trois tours sous forme de trois listes à l'écran. Lors de la configuration initiale, les disques de même diamètre k sont représentés par k et k', et lors des déplacements cette distinction (avec ou sans $'$) permet de suivre le parcours exact de chaque disque. Voir l'exemple d'exécution à la page suivante.

4. Écrire le programme principal qui demande à l'utilisateur d'entrer n, exécute successivement les deux algorithmes 1. et 2. pour déplacer les doubles tours de la gauche vers la droite, et affiche le nombre d'étapes nécessaires.

Exemple d'exécution, pour $n = 3$. On indique par `<==` les déplacements des disques les plus grands (cela peut être utile pour mettre au point l'algorithme récursif et pour le débogage).

```
Entrez le nombre de tours : 3
[ 3 3' 2 2' 1 1' ] - [ ]              - [ ]              au départ (sans ordre)
[ 3 3' 2 2' 1 ]    - [ ]              - [ 1' ]
[ 3 3' 2 2' ]      - [ ]              - [ 1' 1 ]
[ 3 3' 2 ]         - [ 2' ]           - [ 1' 1 ]
[ 3 3' ]           - [ 2' 2 ]         - [ 1' 1 ]
[ 3 3' ]           - [ 2' 2 1 ]       - [ 1' ]
[ 3 3' ]           - [ 2' 2 1 1' ]    - [ ]
[ 3 ]              - [ 2' 2 1 1' ]    - [ 3' ]           <==
[ ]                - [ 2' 2 1 1' ]    - [ 3' 3 ]         <==
[ 1' ]             - [ 2' 2 1 ]       - [ 3' 3 ]
[ 1' 1 ]           - [ 2' 2 ]         - [ 3' 3 ]
[ 1' 1 ]           - [ 2' ]           - [ 3' 3 2 ]
[ 1' 1 ]           - [ ]              - [ 3' 3 2 2' ]
[ 1' ]             - [ ]              - [ 3' 3 2 2' 1 ]
[ ]                - [ ]              - [ 3' 3 2 2' 1 1' ]
Terminé après 14 mouvements.

[ 3 3' 2 2' 1 1' ] - [ ]              - [ ]              au départ (avec ordre)
[ 3 3' 2 2' 1 ]    - [ 1' ]           - [ ]
[ 3 3' 2 2' ]      - [ 1' 1 ]         - [ ]
[ 3 3' 2 ]         - [ 1' 1 ]         - [ 2' ]
[ 3 3' ]           - [ 1' 1 ]         - [ 2' 2 ]
[ 3 3' ]           - [ 1' ]           - [ 2' 2 1 ]
[ 3 3' ]           - [ ]              - [ 2' 2 1 1' ]
[ 3 ]              - [ 3' ]           - [ 2' 2 1 1' ]    <==
[ 3 1' ]           - [ 3' ]           - [ 2' 2 1 ]
[ 3 1' 1 ]         - [ 3' ]           - [ 2' 2 ]
[ 3 1' 1 ]         - [ 3' 2 ]         - [ 2' ]
[ 3 1' 1 ]         - [ 3' 2 2' ]      - [ ]
[ 3 1' ]           - [ 3' 2 2' 1 ]    - [ ]
[ 3 ]              - [ 3' 2 2' 1 1' ] - [ ]
[ ]                - [ 3' 2 2' 1 1' ] - [ 3 ]            <==
[ ]                - [ 3' 2 2' 1 ]    - [ 3 1' ]
[ ]                - [ 3' 2 2' ]      - [ 3 1' 1 ]
[ 2' ]             - [ 3' 2 ]         - [ 3 1' 1 ]
[ 2' 2 ]           - [ 3' ]           - [ 3 1' 1 ]
[ 2' 2 1 ]         - [ 3' ]           - [ 3 1' ]
[ 2' 2 1 1' ]      - [ 3' ]           - [ 3 ]
[ 2' 2 1 1' ]      - [ ]              - [ 3 3' ]         <==
```

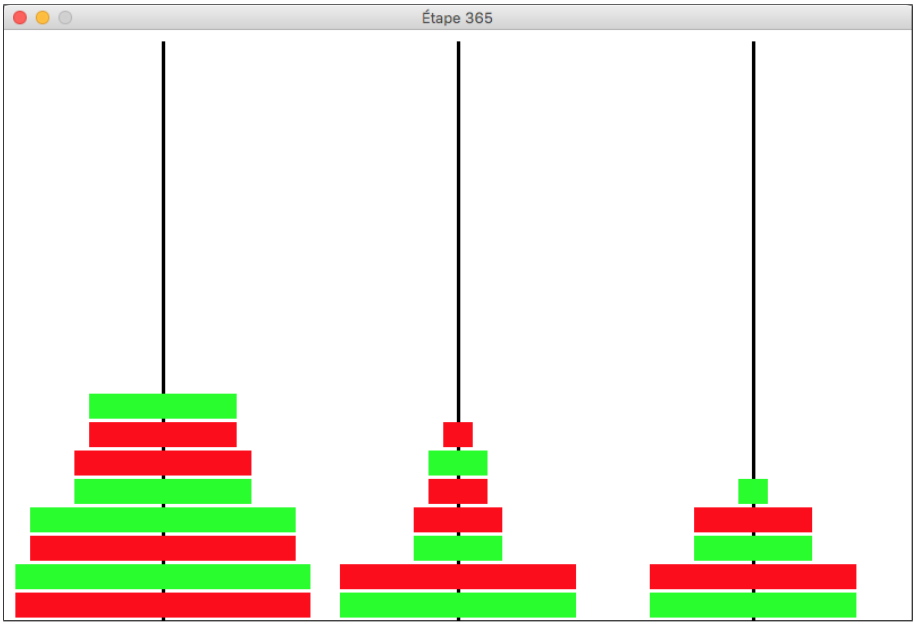
```
[ 2' 2 1 ]          - [ 1' ]            - [ 3 3' ]
[ 2' 2 ]            - [ 1' 1 ]          - [ 3 3' ]
[ 2' ]              - [ 1' 1 ]          - [ 3 3' 2 ]
[ ]                 - [ 1' 1 ]          - [ 3 3' 2 2' ]
[ ]                 - [ 1' ]            - [ 3 3' 2 2' 1 ]
[ ]                 - [ ]               - [ 3 3' 2 2' 1 1' ]
Terminé après 27 mouvements.
```

Exercice 4.16 – Les doubles tours de Hanoï (version Pygame) **

Modifier le programme précédent pour que l'affichage des tours ne se fasse plus dans la console, mais dans une fenêtre Pygame de taille 800×520 pixels. Chaque tour est représentée par un rectangle plein, de couleur rouge ou verte (dans la configuration initiale, le disque vert se trouve au-dessus du disque rouge de même taille), de hauteur 22 pixels et de largeur égale à un multiple de 26 pixels.

Le programme doit fonctionner pour les valeurs de $n = 1, \ldots, 10$. Le temps d'attente entre deux étapes successives sera d'envion $\frac{5}{2^n}$ seconde.

Capture d'écran pour $n = 10$ disques (2^e algorithme), à l'étape 365 (sur 4091).



Exercice 4.17 – Gestion de rectangles *

Créer un programme qui gère des rectangle dans une fenêtre Pygame. Il est basé sur les classes `Rectangle` et `RectangleList` et fait appel à la bibliothèque `pygame`.

I) Initialisation

1. Effectuez le chargement des bibliothèques nécessaires.
2. La fenêtre d'application mesure 640 x 480 pixels et est rafraîchie 30 fois par seconde.

II) La classe Rectangle

La classe `Rectangle` décrit un rectangle par les coordonnées réelles du coin supérieur gauche (`x1,y1`), du coin inférieur droite (`x2,y2`) et d'un indicateur booléen de sélection (`selected`).

1. Le constructeur initialise les coordonnées sur les valeurs passées par paramètres et sélectionne ce rectangle. A défaut de coordonnées, les coordonnées sont initialisées sur $(20, 10)$ et $(50, 30)$.

2. La méthode `sort_coordinates` réarrange (trie) les coordonnées de façon à ce (`x1,y1`) soit le point supérieur gauche et (`x2,y2`) le point inférieur droite.

3. Les méthodes `get_width` et `get_height` retournent respectivement la largeur et la hauteur du rectangle.

4. La méthode `draw` au paramètre `pcolor` dessine sur la toile Pygame le rectangle avec la couleur indiquée et un intérieur blanc. La largeur du trait est de 1 pixel.

5. La méthode `is_inside` aux paramètres `px` et `py` vérifie si le point de coordonnées (`px,py`) se trouve dans le rectangle (bords inclus) et en retourne le résultat sous forme booléenne.

6. La méthode `move` aux paramètres `pdx` et `pdy` déplace le rectangle horizontalement de `pdx` pixels et verticalement de `pdy` pixels.

7. La méthode `update` aux paramètres `px` et `py` ajuste les coordonnées de `x2` et `y2` sur les valeurs respectives en les remplaçant.

III) La classe RectangleList

La classe `RectangleList` gère une liste de rectangles de type `Rectangle`. À cet effet elle dispose d'un seul attribut qui est la liste `list`.

1. Le constructeur initialise la liste `list`.

2. La méthode `add` aux paramètres `px` et `py` crée un rectangle de taille zéro aux coordonnées indiquées et ajoute ce rectangle à la liste.

3. La méthode `update_last_rectangle` aux paramètres `px` et `py` ajuste les coordonnées du dernier rectangle de la liste sur les valeurs respectives.

4. La méthode `draw_all` dessine sur la toile Pygame tous les rectangles non sélectionnés de la liste en noir. Si la liste comporte un rectangle sélectionné, alors celui-ci est dessiné en dernier lieu et en rouge. La largeur du trait est de 1 pixel pour tous les rectangles.

5. La méthode `process_last_rectangle_at_coordinates`, aux paramètres `px`, `py` et `paction`, recherche dans la liste le dernier rectangle dessiné qui contient le point (`px, py`). Dans le cas affirmatif, ce rectangle est traité selon la valeur du paramètre `paction` :

   ⋆ il est sélectionné si `paction` a la valeur `"select"`,

   ⋆ il est supprimé si `paction` a la valeur `"delete"`.

   Si aucun rectangle n'est trouvé, alors il n'y a aucun traitement à faire.

6. La méthode `unselect_all` désélectionne tous les rectangles de la liste.

7. La méthode `move_selected_rectangle` aux paramètres `pdx` et `pdy` déplace le rectangle sélectionné de la liste horizontalement de `pdx` pixels et verticalement de `pdy` pixels. S'il n'y a aucun rectangle sélectionné, alors aucun traitement n'est effectué.

IV) Le programme principal

1. Effectuer les initialisations nécessaires pour créer une fenêtre d'application avec une surface de dessin de 640 x 480 pixels et dont l'entête sera « Rectangles ». La surface de dessin, initialement blanche, est rafraîchie 30 fois par seconde. Finalement une liste vide de rectangles (donc de type `RectangleList`) est créée.

2. La boucle principale du programme se limitera à intercepter et interpréter et gérer proprement les actions de la souris :

   (a) lorsqu'on appuie sur le bouton gauche de la souris, on ajoute à la liste un rectangle de taille zéro et aux coordonnées de la souris,

   (b) lorsqu'on appuie sur le bouton du milieu de la souris, le dernier rectangle dessiné et contenant le point cliqué de la souris (s'il y en a un) est supprimé de la liste,

   (c) lorsqu'on appuie sur le bouton droit de la souris, on sélectionne le dernier rectangle dessiné et contenant le point cliqué de la souris (s'il y en a un) de la liste,

   (d) lorsqu'on déplace la souris avec le bouton gauche enfoncé, on ajuste les coordonnées (`x2,y2`) du dernier rectangle de la liste sur les coordonnées de la souris,

   (e) lorsqu'on déplace la souris avec le bouton droit enfoncé, on déplace le rectangle sélectionné de la liste vers les coordonnées de la souris,

   (f) lorsqu'on relâche un des trois boutons de la souris, on fait désélectionner tous les rectangles de la liste,

   (g) après chaque action de la souris, l'écran est mis à jour (redessiné)

   (h) un clic sur le bouton de fermeture de la fenêtre permet de quitter l'environnement Pygame et de terminer l'application à tout moment.

   Remarque : les programmeurs travaillant sous macOS pourront remplacer l'action du bouton du milieu de la souris (inexistant sur les Macs) par l'enfoncement d'une touche du clavier, par exemple la touche d'espace.

5 Pillow (PIL) [chapitre optionnel]

5.1 Introduction

Pillow est une libraire graphique. Elle constitue la continuation de la librairie populaire PIL, dont les dernières mises-à-jour remontent à 2009. Pillow élargit les performances de Python dans le domaine du traitement d'images. Elle permet d'effectuer une multitude d'opérations dont les plus utilisées sont les suivantes :

- ⋆ lecture, affichage, sauvegarde, conversion de format ;
- ⋆ recadrage, copie, collage ;
- ⋆ décomposition d'une image en bandes rouge, vert, bleu ou niveaux de gris ;
- ⋆ transformations géométriques (redimensionnements, rotations, symétries) ;
- ⋆ filtres divers (flou, netteté, ...) ;
- ⋆ traitements par lot ;
- ⋆ création d'images et accès aux pixels ;
- ⋆ fonctions personnalisées (traitement ponctuel, traitement local).

À part les deux derniers points, toutes les opérations citées ne nécessitent que quelques lignes de code et sont donc faciles d'accès. Pour la plupart des fonctions personnalisées, il existe également des fonctions prédéfinies, mais dans ce cours on essaye de mettre l'accent plutôt sur les techniques de mise en œuvre, que sur le résultat en tant que tel.

5.2 Installation

Afin de pouvoir utiliser la librairie Pillow, il faut la télécharger et installer. Les ordinateurs de votre lycée disposent en principe déjà de la librairie Pillow. Pour les installations privées à domicile (Windows, MacOS, Linux) rendez-vous sur le site `edupython.script.lu` où vous trouverez des liens respectifs. Pour vérifier l'installation de la librairie, il suffit de l'utiliser :

```
>>> from PIL import Image
```

5.3 Lecture, affichage, sauvegarde, conversion

À part de charger la bibliothèque, il suffit de trois instructions pour lire une image, pour l'afficher et pour la sauvegarder sous un autre format (convertir) :

```
from PIL import Image

imgFrog = Image.open("frog.jpg")
imgFrog.show()
imgFrog.save("frog.png")
```

Python permet de lire la plupart des formats de fichiers d'images actuels (png, jpg, bmp, gif, ...).

En important le module `Image` de la libraire, on peut lire une image depuis un fichier externe. Comme ni Pillow, ni Python ne disposent d'un afficheur d'images intégré, la méthode `show()` fait appel à l'afficheur standard de Windows (probablement Photo Viewer). Ceci est d'ailleurs

un point faible de cette librairie : on ne pourra pas voir en direct l'action de nos programmes, mais uniquement leur résultat à la fin de l'exécution.

La procédure de lecture détecte automatiquement le format de l'image. (Cette action semble être très rapide et indépendante de la taille de l'image. En effet, l'instruction `open` ne fait que lire l'en-tête (meta-données) de l'image, le reste n'étant chargé qu'en cas de besoin.) Lors de la sauvegarde de l'image, Python examine l'extension du fichier indiqué pour utiliser le convertisseur approprié. Comme Python est un langage orienté objet, la conversion sans affichage peut se faire en une seule instruction : `Image.open("frog.jpg").save("frog.png")`

Attention : `save` ne demande jamais une confirmation. Si un fichier de même nom existe déjà, il sera remplacé et l'ancien contenu sera définitivement perdu !

5.4 Accès aux informations d'une image

Après le chargement d'une image les informations suivantes sont accessibles en accédant à ses attributs :

information	**nom de l'attribut**	**type**	**exemple**
nom du fichier	filename	str	`myFile = imgFrog.filename`
mode	mode	str	`myMode = imgFrog.mode`
largeur	width	int	`w = imgFrog.width`
hauteur	height	int	`h = imgFrog.height`
taille	size	2-tuple	`(w, h) = imgFrog.size`

5.5 Recadrage copie, collage

Si on désire extraire une partie d'une image en vue d'un traitement (comme le recadrage par exemple), il suffit d'un quadruplet (4-tuplet) de type (x_1, y_1, x_2, y_2) pour extraire la région rectangulaire comprise entre (x_1, y_1) et (x_2, y_2), l'origine se trouvant en haut à gauche :

```
corners = (600, 150, 1400, 750)
imgRegion = imgFrog.crop(corners)
            # imgRegion represente le recadrage de
            # imgFrog (taille : 800x600 px),
            # imgFrog reste inchange.
```

N.B. : – Il est nécessaire que $x_2 > x_1$ et que $y_2 > y_1$!
– La ligne et la colonne du point d'arrivée ne font pas partie de l'extrait.

Pour copier une image entière la chose est bien plus simple : `imgClone = imgFrog.copy()`

Après avoir modifié une région on peut la recoller sur l'image originale en utilisant `paste` :

```
imgFrog.paste(imgRegion)              # insertion de imgRegion au point (0,0)
imgFrog.paste(imgRegion, (x,y))       # insertion de imgRegion au point (x,y)
imgFrog.paste(imgRegion, corners)     # insertion au point (x1,y1)
                                      # imgRegion doit alors avoir la taille
                                      # exacte de corners
```

5.6 Décomposition d'une image en bandes rouge, vert, bleu ou en niveaux de gris

Si on désire travailler individuellement sur les composants ROUGE, VERT et BLEU d'une image, on a intérêt à subdiviser l'image en ses bandes de couleurs r, g et b :

```
r, g, b = imgFrog.split()
```

Les trois bandes r, g et b constituent maintenant des images monochromatiques, dont les contenus représentent respectivement les intensités des couleurs rouge, vert et bleu.


image originale


bande rouge


bande verte


bande bleue

La commande `merge` permet de recombiner ces bandes pour en recréer une image RGB :

```
imgNewFrog = Image.merge("RGB",(r,g,b))
```

Il faut noter qu'ici on fait appel à la classe `Image` pour créer une nouvelle image.

Tout comme le réassemblage des bandes, on peut créer une image en niveaux de gris :

```
imgGrayFrog = imgFrog.convert('L')
```

Le paramètre 'L' indique le type de la cible de conversion (niveaux de gris) et la conversion se base sur la clé pondérée suivante pour générer les différents niveaux de gris :

```
G = R * 0.299 + G * 0.587 + B * 0.114
```

5.7 Transformations géométriques (redimensionnements, rotations, symétries)

Parmi les opérations les plus usuelles figurent les redimensionnements. On les réalise en définissant une taille (par un couple) et en appelant la méthode correspondante :

```
size = (400, 300)
imgSmallFrog = imgFrog.resize(size)
imgSmallFrog.save("smallFrog.jpg")
```

La méthode `resize` permet en outre, par des paramètres optionnels, de définir un filtre de ré-échantillonnage [1] à utiliser et de limiter la région à redimensionner de l'image :

```
imgSmallFrog = imgFrog.resize(size, Image.HAMMING, corners)
```

Reste à noter que `resize` ne respecte pas les proportions de l'image originale, mais fait tenir l'image résultante exactement dans les dimensions fournies par le paramètre `size` ! Il peut donc y avoir une déformation de l'image résultante, tandis que l'image originale reste inchangée.

Les transformations de rotation orthogonale et de symétrie peuvent toutes se faire avec `transpose` :

[1] Le ré-échantillonnage est le processus permettant d'interpoler les valeurs des pixels en transformant votre jeu de données raster. Il est utilisé lorsque l'entrée et la sortie ne s'alignent pas exactement, lorsque la taille de pixel change, lorsque les données sont déplacées, ou pour plusieurs de ces raisons.

```
imgResult = imgFrog.transpose(Image.FLIP_LEFT_RIGHT)
imgResult.show()
imgResult = imgFrog.transpose(Image.FLIP_TOP_BOTTOM)
imgResult.show()
imgResult = imgFrog.transpose(Image.ROTATE_90)
imgResult.show()
imgResult = imgFrog.transpose(Image.ROTATE_180)
imgResult.show()
```

flip vertical

Pour les rotations arbitraires on peut utiliser `rotate` :

`imgResult = imgFrog.rotate(72)` *#en degrés, dans le sens math. positif)*

5.8 Filtres divers (flou, netteté, ...)

La bibliothèque Pillow contient le module `ImageFilter` qui propose une série de filtres de traitement d'images dont les plus usuels sont les suivants :

`BLUR`	rendu flou
`FIND_EDGES`	détection de contours
`SHARPEN`	accentuation de la netteté
`SMOOTH`	élimination d'impuretés

Ils sont mis en œuvre en utilisant la méthode `filter` du module `ImageFilter` :

```
from PIL import ImageFilter
imgResult = imgFrog.filter(ImageFilter.FIND_EDGES)
imgResult.show()
```

détection de contours

5.9 Création de nouvelles images et accès aux pixels

Au lieu de travailler avec des images existantes, on peut aussi créer des images nouvelles. Dans ce cas on utilise la méthode `new` et on fournit les données suivantes :

- ⋆ le mode (`L"` pour niveaux de gris, `'RGB'` pour couleurs et `'RGBA'` pour couleurs avec canal alpha),
- ⋆ la taille (couple),
- ⋆ la couleur initiale (un nombre pour une image en simple bande, un tuplet adéquat pour une image multibande).

```
imgNew = Image.new("RGB", (800,600), (255,255,255))   # image RGB blanche
imgHistogram = Image.new("L", (256,500), 0)           # image monobande noire
```

L'accès aux pixels d'une image peut se faire par `getpixel` et `putpixel` de la classe `Image`, mais c'est très lent. Une manière plus rapide utilise la propriété `pixels` de l'objet `PixelAccess` :

```
pixels = imgFrog.load()
```

Ensuite les pixels peuvent être accédés de manière cartésienne (en lecture et écriture) :

```
value = pixels[x,y]
pixels[0,0] = value // 2
```

L'exemple ci-dessous montre le code pour créer un histogramme de l'image `frog.jpg` (convertie en niveaux de gris).

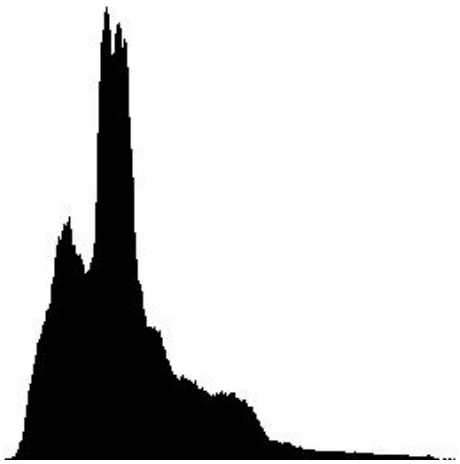
Pour cela on effectue les étapes suivantes :

- ⋆ on charge l'image à traiter, on crée une image pour l'affichage de l'histogramme et on crée les deux accesseurs aux images,
- ⋆ la liste contenant les nombres des différents niveaux de gris est créée et initialisée,
- ⋆ les différents niveaux de gris sont comptabilisés dans la liste `histogram`,
- ⋆ on recherche le maximum de la liste pour normaliser ses valeurs, de façon à ce que l'histogramme tienne dans l'image `imgHistogram`,
- ⋆ pour chaque niveau de gris (0...255) on dessine une ligne noire verticale correspondante sa la valeur dans la liste,
- ⋆ on remplace l'image de l'histogramme par son symétrique vertical pour neutraliser le fait que l'axe des ordonnées est inversé par rapport au sens mathématique,
- ⋆ finalement l'image obtenue est sauvegardée et affichée.

```
from PIL import Image

# input and initialisation
imgFrog = Image.open("frog.jpg").convert('L')   # load image,convert to gray
imgHistogram = Image.new('L',(256,256),255)     # create new image for the histogram
frogPixels = imgFrog.load()                     # create access matrix for the frog
histogramPixels = imgHistogram.load()           # create access matrix for histo.
histogram = [0] * 256                           # create, initialise histogram list

# processing
for x in range(imgFrog.width):                  # count the grayscale of every pixel
    for y in range(imgFrog.height):
        histogram[frogPixels[x, y]] += 1

max = histogram[0]                              # detect the max. of the histogram
for i in range(len(histogram)):
    if histogram[i] > max:
        max = histogram[i]

for i in range(len(histogram)):
    histogram[i] = round(histogram[i]*255/max) # scale the values to fit in image

for x in range(len(histogram)):                 # paint the histogram
    for y in range(histogram[x]):
        histogramPixels[x,y] = 0                # every pixel of histogram is black
```

```
imgHistogram = imgHistogram.transpose(Image.FLIP_TOP_BOTTOM)
                                        # flip the image to make y point up
# output
imgHistogram.save("grayFrogHistogram.jpg")      # save the histogram image
imgHistogram.show()                             # show the histogram image
```

image originale



histogramme

Exercice 5.1 Prendre l'image `frog.jpg` et entourez la grenouille d'un cadre jaune. L'épaisseur du cadre est fournie par l'utilisateur et doit être comprise entre -10 et $+10$. Une épaisseur positive (négative) emboîte les cadres successifs vers l'intérieur (l'extérieur). Afin de bien visualiser la direction, colorier le premier cadre en rouge.

5.10 Traitement par lots (Batch processing)

Il existe beaucoup de situations ou des traitements identiques sont à appliquer à une multitude d'images.

Prenons l'exemple de devoir créer des images miniatures (angl. : *thumbnails*) pour une page web. Dans ce cas on peut se servir de la bibliothèque `glob` pour gérer les expressions génériques comme `*.jpg` par exemple et en créer une liste.

Nous avons vu déjà que `resize` permet de redimensionner une image, mais cette méthode ne tient pas compte des proportions de l'image initiale.

La méthode `thumbnail` est similaire à `resize`, mais elle tient compte des proportions initiales de l'image de façon à ce que l'image résultante ne dépasse pas la taille passée comme argument.

Cependant il faut noter que contrairement à `resize` (qui retourne une image), `thumbnail` agit sur l'image originale. Il faut donc, au besoin, d'abord créer une copie de sauvegarde.

```
imgMiniFrog = imgFrog.copy()    # taille : 1600 x 1200 px
size = (200, 200)
imgMiniFrog.thumbnail(size)     # taille : 200 x 150 px
```

Pour traiter plusieurs fichiers à la fois, il suffit de faire appel à `glob` pour rechercher dans le dossier courant toutes les images à traiter et de les mettre dans une liste :

```
from PIL import Image
import glob

size = (200, 200)                                    # cadre de limite de taille

for imageFile in glob.glob("*.jpg"):                 # prendre tous les '.JPG'
    file = imageFile[:-4]                            # eliminer '.jpg' du nom
    imgCurrentImage = Image.open(imageFile)          # acceder a l'image
    imgCurrentImage.thumbnail(size, Image.ANTIALIAS)          # creer thumbnail
    imgCurrentImage.save(file + "_thumbnail.jpg", "JPEG")     # sauvegarder
```

Le paramètre `*.jpg` de l'instruction de recherche `glob.glob` comporte un joker (caractère substituant). Il existe deux jokers :

? : caractère substituant pour exactement un caractère,

$*$: caractère substituant pour 0 ou plusieurs caractères (sauf ? et $*$).

Une description plus détaillée de `glob` peut être consultée sur la toile à l'adresse : `https://docs.python.org/3/library/glob.html`.

Exercice 5.2 Afin de redresser les portraits d'une prise de vue, créer un programme Python qui demande à l'utilisateur une direction (gauche ou droite) et qui fait pivoter ensuite de 90° toutes les photos de type BMP du dossier actuel.

5.11 Fonctions personnalisées

De nos jours le traitement d'images est devenu une partie intégrante et importante dans de nombreux domaines, que ce soit l'imagerie aérienne et spatiale (réalité virtuelle, analyse, cartographie, ...), les technologies biomédicales (scanner, IRM, échographie, ...) ou encore la robotique (contrôle de qualité, détection de formes, assemblage, ...). Chaque domaine traite l'image différemment, mais on peut distinguer différentes classes de traitement :

- ⋆ Le **traitement ponctuel** s'occupe d'un seul pixel et le traite indépendamment de tous les autres pixels de l'image.
- ⋆ Le **traitement local** s'occupe d'un seul pixel, mais prend aussi en compte l'entourage direct (pixels voisins).
- ⋆ Le **traitement global** s'occupe d'un bloc de pixels, souvent pour corriger des distorsions géométriques ou encore pour créer des superpositions d'images.
- ⋆ Le **traitement (multi-)spectral** s'occupe d'un seul pixel, mais prend en compte son homologue dans des images de bande spectrale différente.

Dans ce cours, nous nous limiterons aux traitements ponctuels et locaux.

5.11.1 Le traitement ponctuel

Parmi les traitements ponctuels les plus utilisés, on trouve l'inversion (négative) d'une image. Il existe bien sûr une méthode pour le faire (`invert`), mais on voudrait mettre l'accent ici sur le traitement et non pas sur le résultat.

```python
from PIL import Image

# input and initialisation
imgFrog = Image.open("frog.jpg").convert('L')    # load image, convert to grayscale
frogPixels = imgFrog.load()                      # create access matrix for the frog

# processing
for x in range(imgFrog.width):                   # count grayscale of every pixel
    for y in range(imgFrog.height):
        frogPixels[x,y] = 256 - frogPixels[x,y]

# output
imgFrog.save("invertedFrog.jpg")                 # save the inverted frog
imgFrog.show()                                   # show the inverted frog
```

image originale

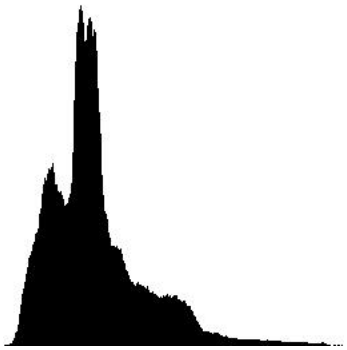
image inversée

Exercice 5.3 Effectuer une expansion (dilatation) des nuances (all. : *Tonwertspreizung*) sur une image.

- ⋆ calculer et afficher l'histogramme de l'image,
- ⋆ demander le seuil pour le noir (niveau de gris en-dessous duquel tous les pixels seront noirs) et le seuil pour le blanc (niveau de gris au-dessus duquel tous les pixels seront blancs.
- ⋆ remplacer tous les pixels hors seuil par le noir, respectivement le blanc. Pour tous les autres pixels effectuer un ajustement linéaire entre les seuils blanc et noir rapportés à l'intervalle $[0, 255]$.

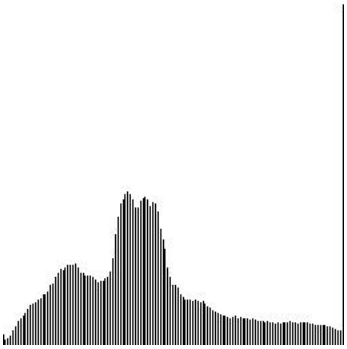
Exemple pour seuil noir = 15 et seuil blanc = 150 avec , avec histogrammes respectifs :

image originale



histogramme

image après dilatation



histogramme après dilatation

Exercice 5.4 Effectuer une quantification (réduction) linéaire sur une image, transformée en niveaux de gris :

- ⋆ demander le nombre de niveaux de gris,
- ⋆ subdiviser la plage nuances en conséquence,
- ⋆ affecter à chaque pixel le seuil de son intervalle,
- ⋆ ajuster la valeur du pixel de façon à ce que le seuil le plus élevé soit le blanc.

L'image ci-contre montre une quantification sur 4 niveaux de gris.

5.11.2 Le traitement local

Le traitement local est utilisé principalement pour des opérations comme le lissage, l'élimination de bruit ou encore la détection de contours. Le principe de traitement est toujours similaire : Chaque pixel de l'image cible est le résultat d'un calcul effectué sur le voisinage du pixel de l'image source. Il peut s'agir des voisins 4- ou 8-connexes, ou encore des matrices carrées impaires centrées autour du pixel de l'image source.

Exemple : Lissage d'une image

Le filtre le plus simple qui existe est celui du lissage. Il remplace le pixel par la moyenne des pixels qui l'entourent. De cette manière on élimine les pixels de bruit impulsionnel. Le côté négatif est que l'image résultante sera moins nette.

On note que le bord de l'image (d'épaisseur de la moitié de la taille de la matrice), n'est pas traité, mais uniquement copié.

```
from PIL import Image

# input and initialisation
imgFrog = Image.open("frog.jpg").convert('L')           # load, convert to grayscale
imgSmoothedFrog = Image.open("frog.jpg").convert('L')   # open a copy of the image,
                                                        # so no need to copy edges!

frogPixels = imgFrog.load()                             # create matrix for the frog
smoothedPixels = imgSmoothedFrog.load()                 # and one for smoothed frog

size = 0
while not ((3 <= size <= 19) and (size % 2 == 1)):      # get an odd matrix size
    shades = int(input("Please enter (ODD) the size of the matrix [3..19] : "))

# processing
halfSize = size // 2                                    # half size for the 4 sides
for x in range(halfSize,imgFrog.width-halfSize):        # do not process the border
    for y in range(halfSize,imgFrog.height-halfSize):
        sum = 0
        for co in range (x-halfSize,x+halfSize+1):      # calculate sum of all
            for li in range(y-halfSize,y+halfSize+1):   # elements of given matrix
                sum += frogPixels[co,li]
            smoothedPixels[x,y] = sum // (size*size)    # average is the new pixel

# output
imgSmoothedFrog.save("smoothedFrog.jpg")                # save the smoothed frog
imgSmoothedFrog.show()                                  # show the smoothed frog
```

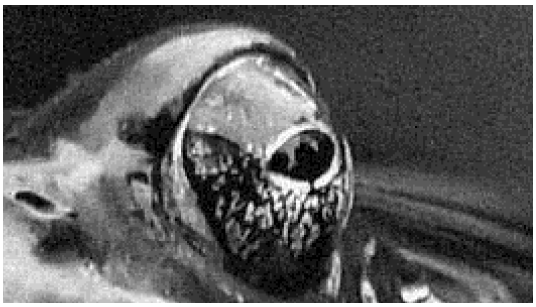


image bruitée

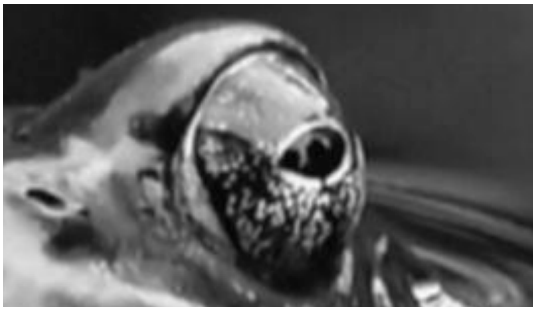


image lissée

Exercice 5.5 Il existe parmi les opérateurs locaux un représentant connu, celui de Sobel. Quelle est son utilité ? Faire une recherche sur Internet et réaliser une implémentation.

Exercice 5.6 Appliquer la réduction linéaire vue plus haut à l'image couleur.

Exercice 5.7 Créer une image bruitée de la grenouille en donnant à 10% des pixels le double de la valeur maximale de leur entourage 5×5. Veiller à ne pas dépasser le maximum (255).

Exercice 5.8 Rechercher sur Internet les informations sur l'opérateur local de dérivée seconde de Laplace. Quelle est son utilité ? Réaliser une implémentation.

5.12 Aide-mémoire

```
# import package and open image :
from PIL import Image
imgFrog = Image.open("frog.jpg")

# show image (default photo viewer) :
imgFrog.show()

# save image in specific format(bmp, gif, jpg, png, ...):
imgFrog.save("frog.png")

# crop image :
corners = (600, 150, 1400, 750)
imgRegion = imgFrog.crop(corners)

# copy and paste image into another image at position (x, y) :
imgClone = imgFrog.copy()
imgLake.paste(imgClone, (x, y))

# split image in color bands :
r, g, b = imgFrog.split()

# recompose / merge image from splitted bands :
imgNewFrog =Image.merge("RGB", (r,g,b))

# convert image to grayscale :
imgGrayFrog = imgFrog.convert('L')

# resize image :
size = (400, 300)
imgSmallFrog = imgFrog.resize(size)

# transpose, rotate, filter :
imgResult = imgFrog.transpose(Image.ROTATE_90)
imgResult = imgFrog.rotate(45)
imgResult = imgFrog.transpose(Image.FLIP_TOP_BOTTOM)
from PIL import ImageFilter
imgResult = imgFrog.filter(ImageFilter.SHARPEN)

# create new white image :
imgNew = Image.new("RGB", (800, 600), (255, 255, 255))
# create new monoband black image
imgHistogram = Image.new("L", (256, 500), 0)

# access specific pixels :
pixels = imgFrog.load()
value = pixels[x,y]
pixels[0,0] = value // 2
```

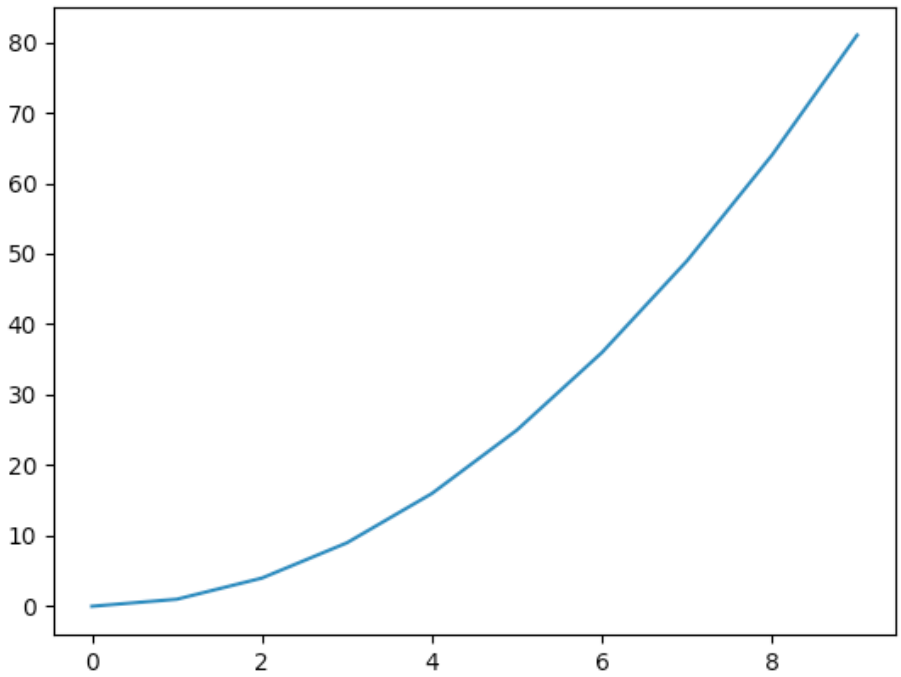
6 Représentation de données [chapitre optionnel]

6.1 La collection `matplotlib.pyplot`

La collection de commandes `matplotlib.pyplot` permet de représenter des graphiques de façon très aisée et standardisée sur la plupart des plateformes pour lesquelles Python est disponible. Il suffit d'installer préalablement la librairie `matplotlib` et celles dont cette dernière dépend (notamment `numpy` pour réaliser efficacement les calculs numériques) ; de plus la plateforme utilisée devra mettre à disposition une application auxiliaire permettant de visualiser les graphiques générés.

Le petit programme suivant, à trois lignes seulement, relie dix points du plan à l'aide de segments de droite ; il s'agit donc d'une approximation de la partie de parabole $y = x^2$ sur l'intervalle $[0, 9]$.

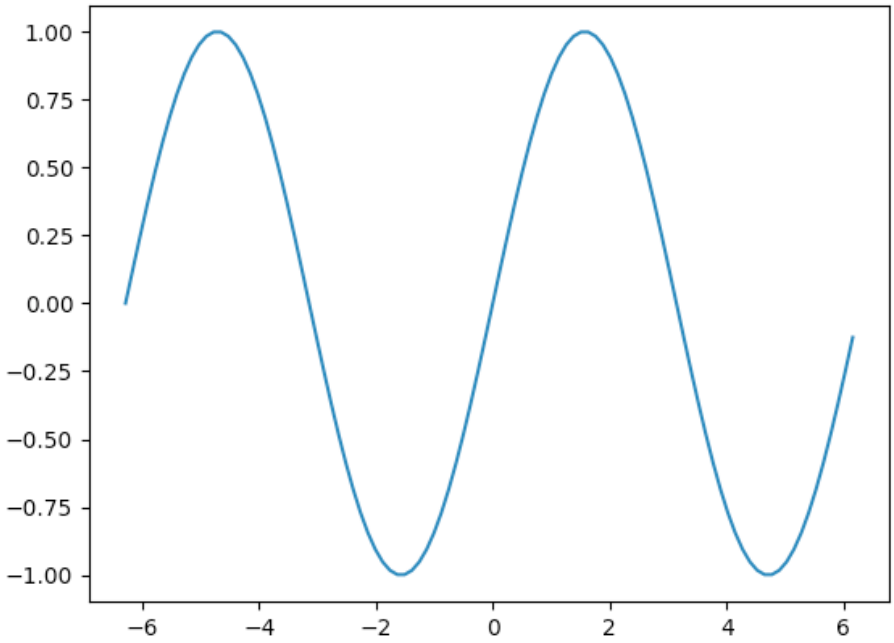
```
import matplotlib.pyplot as plt
plt.plot(range(10), [x ** 2 for x in range(10)])
plt.show()
```



La première ligne fait l'importation nécessaire et permettra dans la suite d'abréger `matplotlib.pyplot` par `plt`. La deuxième ligne réalise le graphique, tandis que la troisième demande à l'utilisateur de fermer la fenêtre du graphique manuellement ; sans cette ligne le programme risque de se terminer aussitôt sans laisser le temps à l'utilisateur de contempler le graphique.

Voici la sinusoïde qui correspond au sinus, représentée sur l'intervalle $[-2\pi, 2\pi[$:

```
from math import sin, pi
import matplotlib.pyplot as plt
coord_x = [-2 * pi + n * pi / 25 for n in range(100)]
plt.plot(coord_x, [sin(x) for x in coord_x])
plt.show()
```



Lorsqu'on importe explicitement la librairie `numpy`, le calcul des abscisses peut être simplifié par un appel à la méthode `numpy.arange`. Voici une nouvelle version du programme qui trace la même sinusoïde :

```
from math import sin, pi
import numpy as np
import matplotlib.pyplot as plt
coord_x = np.arange(−2 * pi, 2 * pi, pi / 25)
plt.plot(coord_x, [sin(x) for x in coord_x])
plt.show()
```

Ou, mieux encore : l'avant-dernière ligne du programme peut s'écrire

```
plt.plot(coord_x, np.sin(coord_x))
```

c'est-à-dire `numpy` met à disposition des fonctions mathématiques qui s'appliquent directement aux listes de nombres.

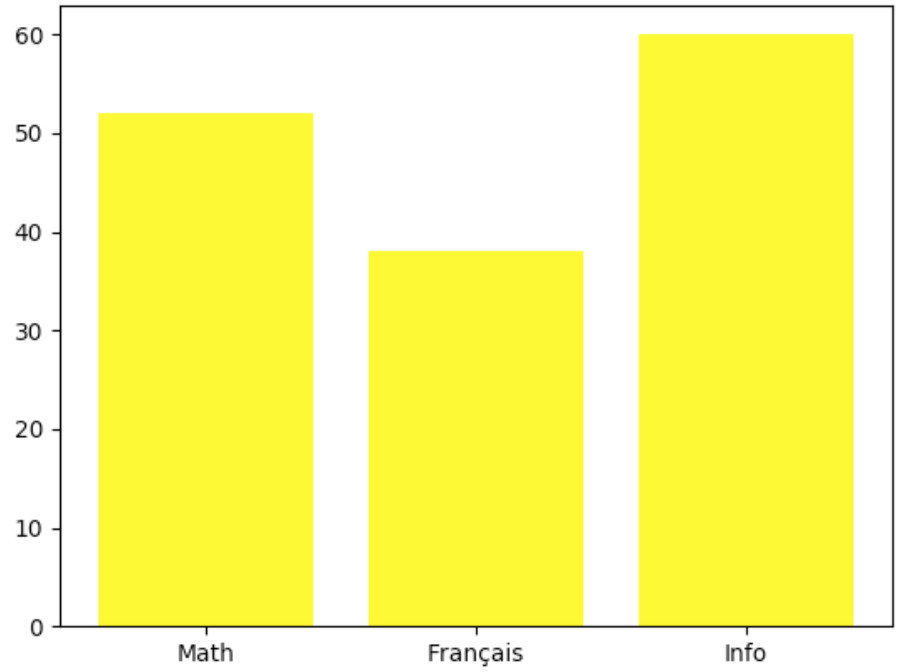
Un ou plusieurs arguments supplémentaires à `plot()` permettent de modifier directement l'apparence du graphique. Par exemple,

```
plt.plot(coord_x, np.sin(coord_x), color = "red")
```

trace la sinusoïde en couleur rouge.

Un diagramme à bâtons est dessiné comme suit :

```
import matplotlib.pyplot as plt
plt.bar(range(3), [52, 38, 60], color = "yellow")
plt.xticks(range(3), ("Math", "Français", "Info"))
plt.show()
```



Il importe de préciser des abscisses, pour que l'emplacement des bâtons et la correspondance entre bâtons et étiquettes soient bien définis.

6.2 Exercices

Exercice 6.1 Écrire un programme qui demande à l'utilisateur d'entrer les coefficients d'un polynôme P à une variable x, par exemple sous forme de liste (voir cours de 2CB), ainsi que des abscisses $x_1 < x_2$. Le programme représentera ensuite le graphique du polynôme entré entre x_1 et x_2.

Exercice 6.2 Compléter le programme de l'exercice précédent : le programme représentera non seulement le polynôme $P(x)$, mais aussi ses première et deuxième dérivées $P'(x)$ et $P''(x)$, sur un même graphique.

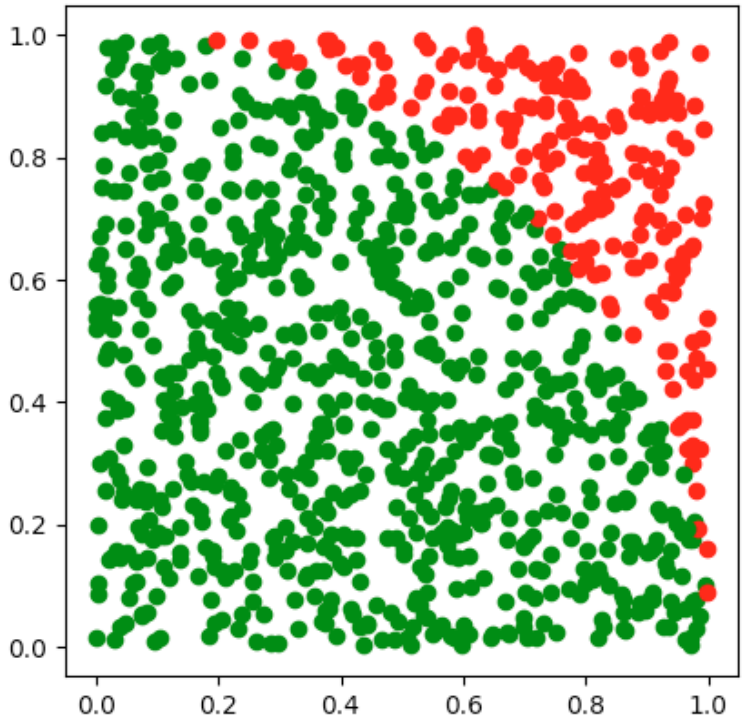
Exercice 6.3 Reprendre le premier exercice de cette section et compléter le programme pour qu'il représente le polynôme $P(x)$ ainsi que sa primitive $\int P(x)\,dx$ qui s'annule en x_1, sur un même graphique.

6.3 Application : processus de Monte-Carlo

Le programme suivant applique la méthode de Monte-Carlo pour approcher la valeur de π. Ce même processus a déjà été illustré dans le 4e chapitre du cours, dans `pygame`.

```
import matplotlib.pyplot as plt
from random import random

def generate(n):
    inside = 0
    for i in range(n):
        x, y = random(), random()
        if x * x + y * y <= 1:
```

```
            plt.plot(x, y, "go")   # point vert (g = green, o = disque)
            inside += 1
        else:
            plt.plot(x, y, "ro")   # point rouge (r = red, o = disque)
    return inside

n = 1000
plt.axes().set_aspect("equal")
inside = generate(n)
my_pi = 4 * inside / n
print("f\textcolor{deepgreen}{$\pi\;\approx$}f", my_pi)
plt.show()
```



6.4 Illustration du fonctionnement des algorithmes de tri

Afin d'illustrer le fonctionnement du *tri par sélection*, nous représentons graphiquement les données de la liste à trier à l'aide de bâtons. Ainsi, nous complétons le code avant l'appel du tri :

```
plt.bar(range(len(a)), a)
plt.pause(3)
selection_sort(a)
```

La première ligne représente la liste initiale. La deuxième ligne fait attendre le programme pendant 3 secondes. Contrairement aux effets de la méthode `plt.show()`, le programme n'est pas interrompu et l'intervention manuelle de l'utilisateur n'est donc pas requise.

À l'intérieur de l'algorithme de tri, nous rajoutons, à l'intérieur de la boucle d'indice i, les lignes suivantes :

```
        plt.close()
        plt.bar(range(len(a)), a)
        plt.bar([i, p], [a[i], a[p]], color = "red")
        plt.pause(1)
```

La première ligne « ferme », c'est-à-dire efface, le graphique précédent. Le nouveau contenu de la liste est représenté en couleur bleue (par défaut), et les éléments échangés sont représentés en rouge. Voici le programme complet :

```
from random import randrange
import matplotlib.pyplot as plt

def make_random_array(n):
    a = [randrange(10 * n) for _ in range(n)]
    return a

def selection_sort(a):
    for i in range(len(a) - 1):
        p = i
        for j in range(i + 1, len(a)):
            if a[j] < a[p]:
                p = j
        if p > i:
            (a[i], a[p]) = (a[p], a[i])
        plt.close()
        plt.bar(range(len(a)), a)
        plt.bar([i, p], [a[i], a[p]], color = "red")
        plt.pause(1)

def print_array(a):
    print(list(a[:100]))

n = int(input("Size: "))
a = make_random_array(n)
print_array(a)
plt.bar(range(len(a)), a)
plt.pause(3)
selection_sort(a)
print_array(a)
plt.close()
plt.bar(range(len(a)), a)
plt.show()
```

Voici encore une illustration graphique du fonctionnement de l'algorithme de *tri rapide*. La sous-liste actuellement triée est représentée en rouge, tandis que le pivot est marqué en jaune. La fonction `partition` reste inchangée. Il suffit de compléter la fonction `quicksort` :

```
def quicksort(a, g = 0, d = ""):
    if d == "":
        d = len(a) - 1
    if g >= d:  # let's stop
        return
```

```
    p = partition(a, g, d)
    plt.close()
    plt.bar(range(len(a)), a)
    plt.bar(range(g, d + 1), a[g : d + 1], color = "red")
    plt.bar(p, a[p], color = "yellow")
    plt.pause(0.5)
    quicksort(a, g, p - 1)   # sublist to the left  of the pivot
    quicksort(a, p + 1, d)   # sublist to the right of the pivot
```

Le graphique illustre le fonctionnement du tri rapide pour une liste de taille 40.