2. La Programmation Orientée Objet (POO)

2.1 Paradigmes de programmation

Cette première partie est inspirée d'un document proposée par Frédéric Manson, professeur NSI.

2.1.1 Différents paradigmes

Les langages de programmation sont nombreux et variés. On peut les regrouper dans plusieurs classes, correspondantes à des schémas de pensée différents : ce sont les **paradigmes de programmation**. Il est d'ailleurs assez usuel maintenant qu'un langage appartienne à plusieurs de ces classes, c'est par exemple le cas de Python (ainsi que de C++, Ruby, OCaml, ...). Certains de ces paradigmes sont mieux adaptés que d'autres pour traiter des problèmes spécifiques. On verra ultérieurement qu'il est possible d'utiliser plusieurs paradigmes à l'intérieur d'un même programme.
Les principaux paradigmes sont :

- impératif (Fortran, COBOL, BASIC, Pascal, C, PHP, ...),
- fonctionnelle (Lisp, OCaml, Haskell, ...)
- événementielle (JavaScript, Processing, Scratch, ...)
- orientée objet (C++, PHP, Python, ...)

2.1.2 Programmation impérative

Le **paradigme impératif** est le paradigme le plus traditionnel. Les premiers programmes ont été conçus sur ce principe :

- Une suite d'instructions qui s'exécutent séquentiellement, les unes après les autres.
- Ces instructions comportent :
  Des affectations
  Des boucles (pour ..., tant que ..., répéter ... jusqu'à ...)
  Des conditions (si ... alors ... sinon)
  Des Branchements/sauts sans condition
- La programmation impérative actuelle limite autant que possible les sauts sans condition. Ce sous-paradigme est appelé programmation structurée. Les sauts sont utilisés en Assembleur (`BR adr`, « branch to adress »), en BASIC (`GOT TO`).
  Un programme utilisant de nombreux sauts est qualifié de « programmation spaghetti » [1], pour la clarté toute relative avec laquelle on peut le dérouler. Certains langages peuvent donner facilement ce style de code (BASIC, FORTRAN, ...)
- L'usage des fonctions, comme on a pu le voir en première, est aussi une variante de la programmation impérative, appelée programmation procédurale. Elle permet de mieux suivre l'exécution d'un programme, de le rendre plus facile à concevoir et à maintenir, et aussi d'utiliser des bibliothèques.

1. https://linuxfr.org/news/encore-un-exemple-de-code-spaghetti-toyota

2.1.3 Programmation fonctionnelle

Ce type de programmation sera vu en fin d'année. Tout ce que l'on peut dire pour le moment c'est que dans ce paradigme tout est fonction. De plus, en programmation fonctionnelle on évite totalement ce qu'on appelle les « effets de bords [2] ».

2.1.4 Programmation événementielle

Comme son nom l'indique, ce type de programmation correspond à des interactions avec des événements extérieurs. Par exemple on a vu en première que suite à un `onclick` on pouvait déclencher un programme écrit en JavaScript qui lui-même pouvait interagir avec l'utilisateur.
De même, on a vu un peu de programmation événementielle avec la carte **micro :bit**. Rappelez-vous, nous avions des instructions du type `if button_a.is_pressed():`.
Enfin, avec un peu de nostalgie, on peut se souvenir du langage Scratch qui est bien événementielle.

2.1.5 Programmation objet

Comme son nom l'indique, le paradigme objet donne une vision du problème à résoudre comme un ensemble d'objets. Ces objets sont définies par :

- leurs propriétés ou caractéristiques. On les appelle les **attributs**.
- leurs comportements ou interactions. On les appelle les **méthodes**.

Les objets interagissent entre eux en respectant leur interface.
L'**encapsulation** introduit une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de sous-programmes destinés à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'**une seule et même entité**.

2.2 Un premier exemple de POO

On souhaite programmer un jeu qui s'inspire [3] de la série de films ***Les gendarmes à Saint-Tropez***.
Pour commencer, nous allons réfléchir à l'**interface** ensuite nous effectuerons son **implémentation** pour en améliorer sa **modularité**.
Ça fait beaucoup de nouveaux mots pour vous mais ne vous inquiétez pas, on y revient tranquillement dans le détail.

2.2.1 L'interface

Hedy et John veulent programmer un jeu inspiré de la série de films ***Les gendarmes à Saint-Tropez***. Pour commencer, ils ont besoin de créer un nouvel objet `Gendarme`.
Hedy décide de s'attaquer à la programmation de ce nouvel objet pendant que John s'occupera de l'environnement graphique. Hedy et John doivent donc se mettre d'accord sur ce que doit être un gendarme et ce que doit faire un gendarme. Ils mettent alors au point un **interface**. Pour le moment, on ne s'occupe pas de savoir comment on va programmer cet objet `Gendarme`.
Cet interface pourrait être le suivant :

2. Par exemple lorsque l'on modifie un tableau dans une fonction sans passer par un `return`.
3. Pourquoi ? Personne ne le sait !

Attribut	Description
`nom`	le nom du gendarme, type `String`
`grade`	le grade du gendarme, type `String`
`qi`	le quotient intellectuel du gendarme, type `integer`, donné à la création entre 40 et 80 [4], donné de manière aléatoire
`nb_pv`	le nombre de procès verbaux donnés depuis le début de la partie, type `integer`, vaut 0 en début de partie

Méthode	Description
`get_nom`	renvoie le nom du gendarme
`get_grade`	renvoie le grade du gendarme
`get_qi`	renvoie le quotient intellectuel du gendarme
`get_nb_pv`	renvoie le nombre de procès verbaux donnés par le gendarme
`set_nom`	modifie le nom du gendarme
`set_grade`	modifie le grade du gendarme
`set_qi`	modifie le quotient intellectuel du gendarme
`set_nb_pv`	modifie le nombre de procès verbaux donnés par le gendarme
`donner_pv`	fonction qui permet de donner un procès verbal. L'attribut `nb_pv` doit pouvoir augmenter de 1. Cette fonction dépend du quotient intellectuel du gendarme.
`monter_grade`	fonction qui permet de monter en grade si on a donné plus de 100 procès verbaux. Le gendarme voit aussi son quotient intellectuel augmenté de 10 points et le nombre de procès verbaux repasse à 0. De plus si le quotient intellectuel est supérieur à 80, le gendarme monte de deux grades. Enfin, le gendarme ne peut pas dépasser le garde de général. (la partie est alors gagnée)

Rmq Les méthodes commençant par `get_...` sont des **accesseurs** (ou getters en franglish). Ce sont des méthodes qui permettent d'obtenir la valeur de l'attribut associé. Nous ne sommes pas obligés de les appeler `get_...` mais c'est une convention très largement utilisée.
Les méthodes commençant par `set_...` sont des **mutateurs** (ou setters en franglish). Ce sont des méthodes qui permettent de modifier la valeur de l'attribut associé. De la même manière nous ne sommes pas obligés de les appeler `set_...`.

Une fois qu'ils se sont décidés, Hedy et John s'attaquent chacun à leurs tâches respectives.

2.2.2 L'implémentation

Maintenant que l'interface est décidé, Hedy va pouvoir le programmer, c'est ce qu'on appelle l'**implémentation**. Qu'importe comment Hedy va effectuer son implémentation, avec une liste, un dictionnaire, une classe ? Grâce à l'interface, John connaît le résultat final attendu et va pouvoir effectuer son travail en s'appuyant sur les résultats attendus d'Hedy. Finalement c'est ce que vous faites vous aussi quand par exemple vous utilisez l'objet `String`. Vous ne savez pas comment il a été implémenté, mais vous savez que si vous faites `'MaJuSCuLe'.lower()` vous obtiendrez la même chaîne de caractères mais tout en minuscule car vous connaissez l'interface.

2.2.3 La modularité

Le découpage en plusieurs modules d'un projet permet de se répartir la tâche à plusieurs. Il permet aussi de s'assurer qu'un changement de structure de données sur une des parties ne nécessite pas de modifier un grand nombre de fonctions. En séparant l'interface de l'implémentation, on augmente la modularité du programme et on facilitera la recherche et la correction de bugs, ainsi que l'ajout de nouvelles fonctionnalités. Mais cela nécessite aussi de bien définir à l'avance les interfaces des principales fonctions et de bien documenter le code pour savoir clairement à quoi sert chaque fonction, et idéalement, comment elle fonctionne. La modularité est essentielle dans tout projet vu le nombres [5] de lignes de code nécessaire.

4. Oui, les Gendarmes de Saint-Tropez ne sont pas très intelligents.
5. Vous trouverez sur ce lien https://www.informationisbeautiful.net/visualizations/million-lines-of-code/ l'ordre de grandeur du nombre de lignes de code de quelques programmes très courants

2.2.4 Création de l'objet `Gendarme`

Le « moule » avec lequel on va fabriquer un objet est appelé une **classe**.
La classe `Gendarme` comprend par exemple les **attributs** (propriétés) et les **méthodes** (comportements).
Quand on crée un objet du type `Gendarme`, l'ordinateur crée ce que l'on appelle une **instance** de la classe. C'est-à-dire que tous les objets de la classe auront les mêmes **attributs** et **méthodes**. Autrement dit, deux `gendarmes`, bien que différentes, sont deux **instances** de la même classe et à ce titre auront toutes les deux les attributs `nom`, `garde`, `qi` et `nb_pv` ainsi que les méthodes `get_nom`, `get_grade`, ... **encapsuler** dans l'objet lui même.

2.2.4.1 Le constructeur

Nous créons notre premier moule, notre **classe** `Gendarme`.
A ce propos, il y a à nouveau une convention très largement respectée qui est qu'une classe doit s'écrire en commençant par une majuscule. Ce n'est pas une obligation, juste une convention.
Ensuite nous allons définir notre première méthode, qui est une méthode particulière, le **constructeur**. Comme son nom l'indique, le constructeur est la méthode qui va permettre de créer une **instance** de la classe.
En Python, le constructeur se nomme toujours `__init__`.

```
1  from random import randint
2
3  class Gendarme :
4      '''
5      La super classe des gendarmes à
       Saint-Tropez
6      '''
7
8      # constructeur
9      def __init__(self, nom, grade):
10         self.nom = nom
11         self.grade = grade
12         self.qi = randint(40,80)
13         self.nb_pv = 0
```

Quelques explications supplémentaires :
Le constructeur est une fonction qui à toujours besoin, au minimum de l'argument `self`. C'est tout simplement pour dire que le constructeur s'adresse à lui-même. Cela semble évident, mais il faut le préciser.

Ensuite la ligne 10 peut se comprendre ainsi :

$$\underbrace{\texttt{self.nom}}_{\text{le nom du gendarme créé}} = \underbrace{\texttt{nom}}_{\text{la valeur nom donnée en argument à la ligne 9}}$$

Pour la ligne 12, le constructeur va automatiquement affecter à l'attribut `qi` un entier aléatoire entre 40 et 80.
A la ligne 13, on affecte à l'attribut `nb_pv` la valeur 0.

Une fois le constructeur donné, on peut créer toutes les instances voulues de notre classe :

```
1 >>> gendarme1 = Gendarme("Cruchot","maréchal des logis")
2 >>> gendarme2 = Gendarme("Gerber","adjudant")
3 >>> gendarme3 = Gendarme("Fougasse","gendarme")
4 >>> gendarme4 = Gendarme("Merlot","gendarme")
5 >>> gendarme5 = Gendarme("Tricard","gendarme")
6 >>> gendarme6 = Gendarme("Berlicot","gendarme")
```

Nous avons donc réussi à créer un nouvel objet qui nous permet de stocker en même temps le nom du gendarme, son grade, son quotient intellectuel et le nombre de procès verbaux qu'il a donnés. C'est plutôt sympathique, mais c'est encore très limité.
On peut néanmoins avoir facilement accès aux attributs de chaque gendarme en écrivant le nom de l'instance de type `Gendarme` suivi d'un point puis de l'attribut voulu.

```
>>> gendarme1.nom
'Cruchot'
>>> gendarme1.grade
'maréchal des logis'
>>> gendarme2.qi
62
>>> gendarme3.qi
74
>>> gendarme3.nb_pv
0
```

ALERTE ROUGE!!!

Python étant un langage très large d'esprit dans ses pratiques [6], il permet une grande ouverture d'esprit. Malheureusement, il permet aussi d'écrire des choses assez bizarres voire choquantes pour ceux ayant découvert la POO avec un autre langage.
Vous pouvez ainsi modifier, l'air de rien, votre instance comme dans le code ci-contre.
Ce genre de code est totalement impossible à faire dans la plupart des langages orientés objets sans passer par ce qu'on appelle des **accesseurs** et des **mutateurs**.
De manière générale, ne choquez personne et ne faites jamais ça.

```
>>> gendarme1.nom = 'Louis de Funès'
# maintenant le nom du gendarme1 a
    changé

>>> gendarme1.truc = 'bizarre'
# on a rajouté un attribut chelou à
    notre gendarme

>>> gendarme1.nom = 32
# maintenant l'attribut nom n'est plus
     une chaîne de caractères,
    attention danger
```

La bonne pratique est donc de **TOUJOURS** utilisé les méthodes **accesseurs** (getters) ou **mutateurs** (setters) pour respectivement accéder ou modifier les attributs d'une instance.

exercice 2.1 Améliorer le constructeur de la classe `Gendarme` de la manière suivante :

- En utilisant la commande `assert` on va s'assurer que pour le gendarme créé, le nom et le grade sont bien des chaînes de caractères.
- En utilisant à nouveau la commande `assert` vérifier que le grade proposé est bien dans la liste suivante `["élève gendarme", "gendarme", "maréchal des logis", "adjudant", "adjudant chef", "major", "élève officier", "sous-lieutenant", "lieutenant", "capitaine", "commandant", "lieutenant colonel", "colonel", "général"]`.
- Rajouter un attribut de votre choix.

■

6. Le fondateur de Python, Guido van Rossum, est fréquemment citée pour cette phrase : « We are all consenting adults here ». Elle signifie : « Nous sommes tous des adultes consentants ». Sous entendu : si vous voulez vous tirer une balle dans le pied, allez-y, vous êtes adulte après tout.
D'après le cours Python d'OpenClassrooms :
https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python/4313211-comprenez-lencapsulation

2.2.4.2 Méthodes

Les méthodes sont les fonctions qui vont être directement encapsulées à l'objet `Gendarme`. On va donc les écrire directement après le constructeur, dans le même bloc.

```
from random import randint

class Gendarme :
    '''
    La super classe des gendarmes à Saint-Tropez
    '''

    # constructeur
    def __init__(self, nom, grade):
        self.nom = nom
        self.grade = grade
        self.qi = randint(40,80)
        self.nb_pv = 0

    # méthodes
    def get_nom(self):
        ''' accesseur pour l'attribut nom '''
        return self.nom

    def set_nom(self, nouveau_nom):
        ''' mutateur pour l'attribut nom '''
        assert isinstance(nouveau_nom,str), "le nouveau nom n'est pas une chaîne de caractères"
        self.nom = nouveau_nom

    ...
```

Pour appliquer la **méthode** `get_nom()` à l'**instance** `gendarme1`, on écrira `gendarme1.get_nom()` et non pas `get_nom(gendarme1)` pour bien préciser que l'on applique la méthode à `gendarme1`.

```
>>> gendarme1.get_nom()
'Cruchot'
>>> gendarme1.set_nom('louis')
>>> gendarme1.get_nom()
'louis'
```

exercice 2.2 Rajouter toutes les méthodes prévues dans l'interface conçu par Hedy et John. ■

exercice 2.3 Rajouter une ou plusieurs méthodes utilisant l'attribut que vous avez rajouté à l'exercice 2.1. ■

2.3 Hors programme mais diaboliquement sympathiques : les fonction spéciales

Maintenant que nous avons créé notre nouvel objet de type `Gendarme`, on peut essayer de voir ce qui se passe avec la fonction `print` :

```
>>> print(gendarme1)
<__main__.Gendarme object at 0x7fd1fe7ecf10>
```

Cette réponse est plutôt décevante, même si elle nous donne plein d'information :

- `__main__.Gendarme object` signifie que `gendarme1` est une instance de la classe `Gendarme`
- `at 0x7fd1fe7ecf10` signifie que cette instance a pour adresse mémoire `0x7fd1fe7ecf10`

On pourrait néanmoins se dire que ce serait plus intéressant d'avoir dans la console un affichage du type : `'Bonjour, je suis le maréchal des logis Cruchot, j'ai un qi de 62 et j'ai déjà dressé 3 pv.'` Il suffit d'écrire une méthode pour cela :

```
    def affichage(self) :
        ''' affiche une belle phrase de présentation dans la console '''
        voyelles = "aeéèêiouy"
        texte = "Bonjour, je suis l"
        if self.grade[0] in voyelles :
            # la première lettre est une voyelle
            texte = texte + "'" + self.grade
        else :
            texte = texte + "e " + self.grade
        texte = texte + " " + self.nom + ", j'ai un qi de " + str(self.qi)
        texte = texte + " et j'ai déjà dressé " + str(self.nb_pv) + " pv."
        return texte
```

Nous obtenons bien le résultat attendu :

```
>>> gendarme1.affichage()
"Bonjour, je suis le maréchal des logis Cruchot, j'ai un qi de 62 et j'ai déjà
    dressé 3 pv."
```

Mais ce serait tout de même mieux si on pouvait obtenir le même résultat directement avec un `print(gendarme1)`. Pas de problème, il suffit de faire ce qu'on appelle une **surcharge d'opérateur**. Autrement dit, nous allons apprendre à la fonction `print` à afficher correctement les objets du type `Gendarme` sans pour autant qu'elle oublie ce qu'elle savait déjà faire. Pour ce faire, il suffit de remplacer le nom de notre méthode `affichage` par la **fonction spéciale**ment [7] associée à la fonction `print` : `__str__`.

```
    def __str__(self) :
        ''' affiche une belle phrase de présentation dans la console '''
        voyelles = "aeéèêiouy"
        texte = "Bonjour, je suis l"
        if self.grade[0] in voyelles :
            # la première lettre est une voyelle
            texte = texte + "'" + self.grade
        else :
            texte = texte + "e " + self.grade
        texte = texte + " " + self.nom + ", j'ai un qi de " + str(self.qi)
        texte = texte + " et j'ai déjà dressé " + str(self.nb_pv) + " pv."
        return texte
```

7. On parle aussi de **méthode spéciale**.

On peut maintenant faire notre `print` tranquillement :

```
>>> print(gendarme1)
Bonjour, je suis le maréchal des logis Cruchot, j'ai un qi de 62 et j'ai déjà
    dressé 3 pv.
```

Voici la liste des **fonctions spéciales** qui permettent de surcharger les opérateurs de base :

Opérateur	Notation	Méthode à définir
Signe positif	+	`__pos__`
Signe négatif	-	`__neg__`
Addition	+	`__add__`
Soustraction	-	`__sub__`
Multiplication	*	`__mul__`
Division	/	`__truediv__`
Exponentiation	**	`__pow__`
Division entière	//	`__floordiv__`
Reste de la division entière	%	`__mod__`
Égal	==	`__eq__`
Différent	!=	`__ne__`
Strictement plus petit	<	`__lt__`
Plus petit ou égal	<=	`__le__`
Strictement plus grand	>	`__gt__`
Plus grand ou égal	>=	`__ge__`
« non » logique	not	`__not__`
« et » logique	and	`__and__`
« ou » logique	or	`__or__`
Affichage d'un truc	`print(truc)`	`__str__`
Contenu d'un truc	retour charriot	`__repr__`
Longueur d'un truc	`len(truc)`	`__len__`
n-ième élément d'un `truc`	`truc[n]`	`__getitem__`

Par curiosité, vous pouvez regarder le fichier `maFraction.py` que l'on retrouvre dans mon GitHub. Dans le script j'ai défini une classe `Fraction`. En utilisant les fonctions spéciales, je peux définir des fractions, elles sont enregistrées sous leur forme irréductible, je peux effectuer toutes les opérations (addition, soustraction, multiplication, division, puissance), je peux les comparer (égalité, supériorité stricte ou large, infériorité stricte ou large) et, enfin, je peux les afficher avec la barre de fraction si besoin.

2.4 Hors programme : héritage, polymorphisme, notion de public privé

Désolé, c'est hors programme et nous n'avons pas le temps d'aborder ce point ... Donc je n'en parlerai pas.

2.5 La PEP 8

Une PEP (ou *Python Enhancement Proposals*) est comme son nom l'indique une proposition d'amélioration de Python. Une fois une PEP validée, elle est publiée avec son numéro. La PEP 8 est celle auquel on fait le plus référence car elle conseille sur les bons usages d'écriture du code. Respecter la PEP 8 n'est pas une obligation, mais si on la respecte du mieux possible, son code est plus rapidement accessible à une autre personne.
D'une manière générale, on doit utiliser la convention dite **CamelCase**, où la « casse de chameau ». Cette convention consiste à écrire sans espace, sans _, en mettant en lettre capital le début de chaque mot, ce qui provoque des ondulations comme un dos de chameau : desOndulationsCommeUnDosDeChameau.

- On n'utilise jamais de caractères spéciaux (accents, espace, ..)

- Le nom d'une variable doit commencer par une minuscule : `maVariable`
- Le nom d'une classe doit commencer par une majuscule : `MaClasse`
- Le nom d'une fonction doit commencer par une minuscule : `maFonction()`
- Les constantes s'écrivent en majuscule : `CONSTANTE`

https://www.python.org/dev/peps/pep-0008/

2.6 Exercices

exercice 2.4 1. Écrire la classe `Eleve` qui permettrait pour un élève de stocker les données suivantes : son nom, son prénom, ses notes obtenues à trois DS, ainsi qu'une méthode qui calculerait la moyenne des trois notes.

2. a. Définir une instance pour Hedy Lamarr qui a obtenu les notes suivantes : 18 au DS1, 19 au DS2 et 20 au DS3.
   b. Encapsuler une méthode afin d'obtenir sa moyenne.

■

exercice 2.5 Programmer un QCM à l'aide d'une classe et de méthodes. ■

exercice 2.6 — *. Créer une classe `Horloge`, puis la tester.
Les attributs sont :
- `heures` ;
- `minutes` ;
- `secondes`.

Les méthodes sont :
- `ticTac` : cette méthode augmente l'horloge d'une seconde ;
- `reveil(h , mn , s)` : sonne le réveil à une heure donnée par l'utilisateur ;
- `__repr__` : représente l'objet.

■

exercice 2.7 — **. *Exercice proposé par Frédéric Manson, professeur NSI.*

1. Créer une classe `Boite`. Cette classe a pour attributs :
   - `longueur`
   - `largeur`
   - `hauteur`
   - Ces trois attributs sont dans un ordre décroissant longueur $\geq$ largeur $\geq$ hauteur (on peut choisir des dimensions entre 1 et 50)

   Elle a pour méthodes :
   - `volume`, qui comme son nom l'indique donne le volume d'une boite
   - `rentreDans(autreBoite)`, qui renvoie vrai si l'objet `Boite` rentre dans `autreBoite`.
2. Créer aléatoirement une liste d'une vingtaine de boîtes.

3. A l'aide d'un algorithme glouton, donner une suite de boîtes tirées de la liste obtenue à la question précédent. Cette suite sera aussi grande que possible avec des boîtes qui rentrent les unes dans les autres. Rappel : pour trier les boîtes, on fera appel à la fonction `sorted()` ou la méthode `sort()` avec en argument une fonction qui donne la clef pour effectuer le tri. Cette fonction à passer en paramètre prend un élément de la liste et retourne ce sur quoi doit s'effectuer le tri.

```
def laClef(objet):
    """
    Renvoie la valeur qui sera examinée pour le tri
    """
    return valeur de l'objet
liste_tries = sorted(liste, key = laClef)
```

On peut rajouter `reverse = True` pour avoir l'ordre décroissant.

2.7 Mini-projet : programmer une fontaine de balles rebondissantes avec pyxel

2.7.1 Installation du module

pyxel est un moteur de jeu pour Python qui permet de programmer des jeux dans une ambiance rétro : les pixels peuvent être très gros, il n'y a que 16 couleurs différentes et on ne peut jouer que 3 sons à la fois. Cette volonté minimale permet au moteur de jeu d'être simple, efficace et rapide.
Vous trouverez en suivant ce lien un tutoriel :
https://nuitducode.github.io/DOCUMENTATION/PYTHON/01-presentation/
Pour commencer nous allons installer pyxel si ce n'est déjà fait. Il suffit d'écrire dans la console la commande :

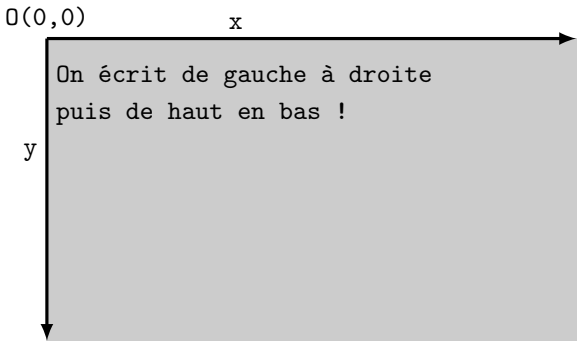
```
>>> pip install pyxel
```

Rmq

Pour programmer notre animation ou un jeu, il faudra donner toutes les coordonnées de tous les objets visibles, mais attention, il y a une chose très particulière à savoir. En informatique, dans la grande majorité des cas, le repère n'est pas le repère habituellement vu en mathématique.

- L'origine du repère est sur le coin en haut à gauche de la fenêtre.
- L'axe des abscisses est orienté classiquement vers la droite.
- L'axe des ordonnées est orienté vers le bas !

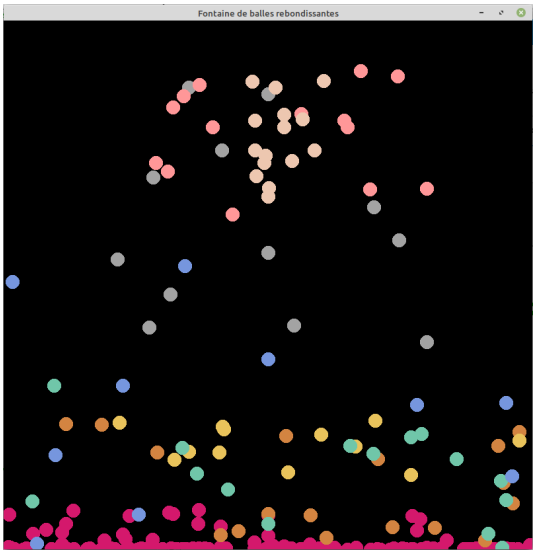
Pensez à l'écriture : on va de droite à gauche (axe des abscisses) et ensuite de haut en bas (axes des ordonnées).



2.7.2 Le cahier des charges

Nous allons programmer une fontaine de balles rebondissantes.

- les balles vont toutes partir d'un même point et être projetées en haut dans des directions aléatoires ;
- les balles vont être attirées par la gravité et vont retomber au sol ;
- si les balles touchent le cadre gauche ou droit de la fenêtre elles rebondissent sans perte d'énergie (elles gardent la même vitesse) ;
- si les balles touchent le sol, elles vont rebondir en perdant de l'énergie (elles monteront donc moins haut) ;
- les balles ont une durée de vie limitée ;
- si on appuie sur la touche espace du clavier, toutes les nouvelles balles changent de couleur.



2.7.3 Les imports et constantes

Pour commencer, on importe les modules nécessaires et 2 variables globales (autrement dit des variables qui seront lisibles par toutes les fonctions sans avoir à les mettre en attribut). L'intérêt des variables globales est que si on désire changer la taille de l'écran ou le rayon des balles, il n'y aura que très peu de choses à modifier.
Par convention, une constante est écrite uniquement avec des caractères majuscules.

```
1 import pyxel
2 from random import randint
3
4 # rayon des balles
5 global RAYON ; RAYON = 10
6
7 # taille de l'écran
8 global COTE ; COTE = 800
```

2.7.4 La classe `Balle`

Nous définissons donc notre classe `Balle`. Nous pourrons ainsi définir un très grand nombres de balles sur le même modèle. J'ai choisi d'avoir une classe sans méthode.

```
11 class Balle :
12     '''
13     objet Balle avec ses coordonnées, son différentiel en x et en y
14     '''
15     # le constructeur
16     def __init__(self, couleur):
17         self.x = COTE//2
18         self.y = COTE//3
19         self.dx = randint(-10,10)
20         self.dy = randint(-20,-10)
21         self.couleur = couleur              # couleurs entre 0 et 15
22         temps_secondes = 8                  # secondes
23         self.temps_vie = 30*temps_secondes  # il y a 30 images par seconde
```

Explications :

- lignes 17 et 18 nos balles vont toutes partir du même point d'abscisse `COTE//2` et d'ordonnée `COTE//3` ;
- lignes 19 et 20 à chaque étape, la balle va se déplacer suivant les abscisses de `dx` et suivant les ordonnées de `dy`. Ces deux valeurs sont choisies au hasard.
  Pour `dx`, c'est un nombre négatif ou positif qui déplacera la balle respectivement à gauche ou à droite par rapport au point de départ.
  Pour `dy`, c'est un nombre négatif, donc pour commencer la balle est lancée vers le haut (on appliquera ensuite la gravité pour faire retomber la balle) ;
- lignes 16 et 21 la couleur de la balle est le seul attribut de la classe paramétrable ;
- lignes 22 et 23 la durée de vie des balles est limitée à 8 secondes. On tient compte ici du nombre d'images par seconde de l'animation (on en aura 30 par seconde).

2.7.5 L'animation

2.7.5.1 Le constructeur

Nous allons maintenant pouvoir utiliser les outils du module pyxel. Vous allez voir, il s'agit bien de POO. Pour commencer le constructeur :

```
26 # =====================================================
27 # == La class qui définit l'animation
28 # =====================================================
29
30 class Anim:
31     # le constructeur
32     def __init__(self):
33         # taille de la fenêtre COTE * COTE pixels
34         pyxel.init(COTE, COTE, title = "Fontaine de balles rebondissantes")
35
36         # les balles
37         self.couleur = 1
38         # notre liste de balles qui n'en contient qu'une pour commencer
39         self.balles = [Balle(self.couleur)]
40
41         # pour exécuter le jeu
42         pyxel.run(self.update, self.draw)
```

Explications :

- ligne 34 c'est une commande obligatoire du module pyxel, à placer au début du constructeur. On définit en pixels la taille de la fenêtre et le titre qui sera affiché ;
- ligne 42 c'est à nouveau une commande obligatoire, à placer à la fin du constructeur. On annonce quelles sont les méthodes qui vont permettre le bon déroulement du jeu (pour nous de l'animation) : `self.update` pour définir ce qui est modifié 30 fois par seconde et `self.draw` ce qu'il faut dessiner 30 fois par seconde ;
- lignes 37 et 39 on crée une liste pour toutes nos balles. Pour commencer il n'y a qu'une balle dans notre liste. Sa couleur est donnée par `self.couleur`.

2.7.5.2 Quelques méthodes

```
45     # =====================================================
46     # == Des méthodes
47     # =====================================================
48     def change_color(self):
49         """change les couleurs de toutes les nouvelles balles
50         si on appuye sur la touche espace"""
51         if pyxel.btnr(pyxel.KEY_SPACE):
52             self.couleur += 1
53             # self.couleur ne peut pas dépassser 15
54             if self.couleur == 16:
55                 # self.couleur ne doit pas valoir 0 sinon les balles sont noires
56                 self.couleur = 1
```

Explications :

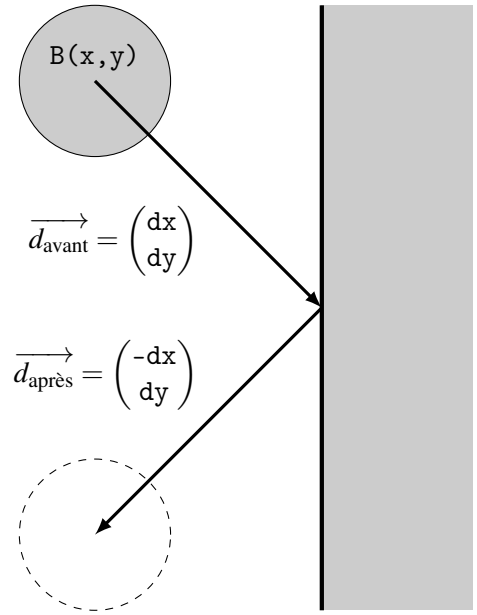
- lignes 51 et 52 on pourra changer de couleur en incrémentant de 1 `self.couleur` lorsque l'on appuie sur la touche ESPACE ;
- lignes 54 et 56 attention, avec le module pyxel il n'y a que 16 couleurs numérotés de 0 à 15. On doit donc s'assurer que l'on ne dépasse pas 16 et éviter la valeur 0 (balle noire sur fond noire).

Rmq Si vous avez bien tout suivi à la ligne 51 nous faisons de la programmation événementielle.

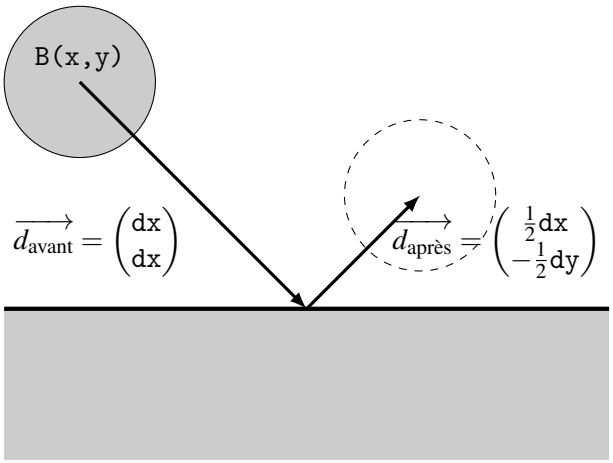
```
58      def deplacement(self):
59          """déplacement des balles avec les rebonds sur les côtés ou au sol"""
60          for b in self.balles :
61              if b.x + b.dx > COTE or b.x + b.dx < 0:
62                  # on touche le mur droit ou gauche
63                  # on modifie le sens de dx
64                  b.dx = b.dx * (-1)
65              if b.y + b.dy > COTE :
66                  # on touche le sol
67                  # on modifie le sens de dy et on le divise par 2
68                  b.dy = b.dy // (-2)
69                  b.dx = b.dx // 2
70              # la balle se déplace
71              b.x = b.x + b.dx
72              b.y = b.y + b.dy
73              # on simule la gravité en accélérant dy
74              b.dy = b.dy + 1
```

Explications :

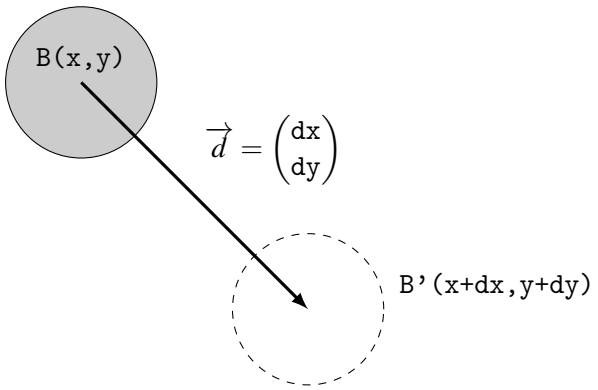
- lignes 61 et 64 si une balle touche un des deux bords de la fenêtre, elle rebondit.
  Pour programmer ce rebond, on constate qu'il suffit de changer le signe de `dx`.



- lignes 65 et 69 si une balle touche le sol, elle rebondit avec moins d'énergie.
  Pour programmer ce rebond, on constate qu'il suffit de changer le signe de `dy` mais aussi de diminuer les valeurs de `dx` et `dy`.
  Ici nous avons décidé de diviser ces valeurs par 2.



- lignes 71 et 72 les coordonnées de la balle sont modifiées pour créer un déplacement. En termes mathématiques on dira qu'on a appliqué une translation de vecteur $\begin{pmatrix} dx \\ dy \end{pmatrix}$.
  Les nouvelles coordonnées sont alors `(x+dx,y+dy)`.



- ligne 74 à chaque étape la valeur de dy est incrémentée de 1. Le déplacement par rapport à la hauteur n'est pas constant. On simule alors une accélération induite par la gravité : la balle va de plus vite vers le bas.

```
76      def vieillir(self) :
77          """ vieillissement et mort des balles"""
78          # on crée la liste des balles qui sont périmées
79          liste_dead = []
80          for i in range(len(self.balles)) :
81              self.balles[i].temps_vie -= 1
82              if self.balles[i].temps_vie == 0 :
83                  liste_dead.append(i)
84          # on supprime les balles périmées de la liste en commençant par la fin
85          for i in range(len(liste_dead)-1,-1,-1) :
86              self.balles.pop(i)
```

Explications :

- lignes 80 et 81 chaque balle vieillit et donc perd 1 en temps de vie ;
- lignes 82 et 83 si le temps de vie est à 0, on rajoute dans la liste `liste_dead` cette balle comme étant une balle périmée ;
- ligne 85 et 86 on supprime ensuite toutes les balles périmées en partant de la fin de la liste. Mais pourquoi ? ? ?

2.7.5.3 La méthode `update`

La méthode `update` va informer Python des nouveautés qui doivent s'appliquer 30 fois par seconde.

```
89      # ======================================================
90      # == UPDATE
91      # ======================================================
92      def update(self):
93          """mise à jour des variables (30 fois par seconde)"""
94          # on effectue les mises à jour éventuelles
95          self.change_color()
96          self.deplacement()
97          self.vieillir()
98          # on rajoute une nouvelle balle
99          self.balles.append(Balle(self.couleur))
```

Explications :

- lignes 95, 96 et 97 on applique les changements (éventuels) sur la couleur des balles, sur leur déplacement et sur leur temps de vie ;
- ligne 99 on rajoute une nouvelle balle à la liste des balles. Nous créons donc 30 balles à la seconde.

2.7.5.4 La méthode `draw`

La méthode `draw` va s'occuper de l'affichage et de l'animation.

```
102     # ========================================================
103     # == DRAW
104     # ========================================================
105     def draw(self):
106         """création et positionnement des objets (30 fois par seconde)"""
107         # vide la fenêtre
108         pyxel.cls(0)
109 
110         # on dessine les balles
111         for b in self.balles :
112             pyxel.circ(b.x, b.y, RAYON, b.couleur)
```

Explications :

- ligne 108 on commence par vider la fenêtre en appliquant une couleur uniforme (0 pour noire) sur toute la fenêtre. C'est cette étape, répétée 30 fois par seconde qui va donner l'impression d'une animation comme si à l'instar des premiers dessins animés on repartait d'une nouvelle feuille ;
- lignes 111 et 112 on dessine chaque balle sous la forme d'un disque de coordonnées `(b.x, b.y)`, de rayon `RAYON` et de couleur `b.couleur`.

2.7.6 Pour lancer l'animation

Il suffit de créer une instance de la classe `Anim` :

```
115 Anim()
```

2.7.7 A vous de jouer

Modifier ce programme selon votre choix. Vous pouvez par exemple avoir des balles qui se déplacent toutes de manières rectilignes en rebondissant sur les murs. Vous pouvez aussi créer des balles qui rebondissent les unes sur les autres. On peut aussi imaginer que vous créez les balles en cliquant avec la souris sur la fenêtre. On peut remplacer les balles par des têtes de gendarmes ...
N'ayez comme limite que votre imagination et épatez moi !