

📍 Avenue V. Maistriau 8a
B-7000 Mons
📞 +32 (0)65 33 81 54
✉️ scitech-mons@heh.be

WWW.HEH.BE

Programmations avancées - Pratique

Cours POO - UML

Bachelier en informatique et systèmes – Finalité Télécommunications et réseaux –Bloc 1.

Chapelle Joakim – joakim.chapelle1@heh.be

Depreter Johan – johan.depreter@heh.be

Desmet Erwin – erwin.desmet@heh.be

Scopel Fabrice – fabrice.scopel@heh.be

Table des matières

1.	Introduction / Rappels des concepts	4
1.1	Les Concepts de base de la POO	4
1.2	Déférence entre l'approche procédurale et la POO	5
1.3	Les classes et la Hiérarchie.....	6
1.4	Exemple 2	7
1.5	Qu'est-ce qu'un objet ?.....	8
1.6	L'héritage	8
1.7	De quoi est composé un objet ?	9
1.8	Notre première classe.....	10
1.9	Notre premier Objet	11
1.10	Principaux points à retenir.....	11
2	Les piles pour comprendre l'approche objet.....	13
2.1	L'approche Objet	13
2.2.1	L'encapsulation et ce qui l'entoure	15
2.2.1	Notion de Visibilité.....	18
2.2.2	Le self	19
2.3	Testons et comprenons notre exemple sur les piles complété	19
2.3	Principaux points à retenir.....	20
3	Propriétés de la POO.....	21
3.1	Variables d'instance	21
3.1.1	Examinons le code suivant :	23
3.2	Variables de classes	24
3.3	Vérification de l'existence d'un attribut.....	26
3.4	Principaux points à retenir.....	28
4	Les méthodes en détails.....	29
4.1	Récapitulatif de tout ce qu'on sait sur les méthodes	29
4.2	Les atouts de self.....	30
4.3	le constructeur	30
4.4	La vie cachée des classes et des objets	32
4.4.1	Name	33
4.4.2	Module	33
4.4.3	Bases	33
4.5	Le python une introspection et une réflexion qui paye	34
4.6	Analysons une classe.....	35
4.7	Principaux points à retenir.....	36
5	L'héritage, Pourquoi et comment ?.....	38
5.1	Contextualisation.....	39

5.2 issubclass()	40
5.3 isinstance()	42
5.4 L'opérateur is	43
5.5 Comment python utilise les méthodes et les attributs hérités	44
5.6 Accès aux variables de classes	46
5.7 Accès aux variables d'instances	46
5.7 Résumé de l'héritage jusqu'ici	47
5.8 L'héritage multiple	48
5.9 Comment construire une hiérarchie de classes	51
5.10 Le polymorphisme	52
5.11 La composition	54
5.12 L'héritage simple vs l'héritage multiple	56
5.13 Principaux points à retenir	57

1. Introduction / Rappels des concepts

1.1 Les Concepts de base de la POO

Essayons de nous éloigner de la programmation informatique et des ordinateurs en général, et discutons du problème de la programmation objet.

Presque tous les programmes et techniques que vous avez utilisés jusqu'à présent relèvent du style de programmation procédural. Certes, vous avez utilisé certains objets intégrés(notamment sur les listes), mais sans vraiment savoir ce que cela impact et pourquoi on appelle cela un objet.

La programmation procédurale a été l'approche dominante du développement de logiciels pendant des décennies, et elle est toujours utilisée aujourd'hui. De plus, il ne va pas disparaître à l'avenir, car il fonctionne très bien pour des types de projets spécifiques (généralement, pas très complexes et pas volumineux, mais il y a beaucoup d'exceptions à cette règle).

L'approche objet est « encore » assez jeune (beaucoup plus que l'approche procédurale) et est particulièrement utile lorsqu'elle est appliquée à des projets importants et complexes réalisés par de grandes équipes composées de nombreux développeurs.

La structure d'un projet en POO facilite de nombreuses tâches importantes, par exemple, la division du projet en petites parties indépendantes et le développement indépendant de différents éléments du projet.

Python est un outil universel pour la programmation objet et procédurale. Il peut être utilisé avec succès dans les deux sphères. Dans ce cours, nous nous contenterons de parler du Python comme langage.

De plus, en Python POO, vous pouvez créer de nombreuses applications utiles, même si vous ne connaissez rien aux classes et aux objets. Mais vous devez quand même garder à l'esprit que certains problèmes (par exemple, la gestion de l'interface utilisateur graphique) peuvent nécessiter une approche objet stricte. Heureusement, la programmation objet est relativement simple si on garde un esprit logique.



1.2 Différence entre l'approche procédurale et la POO

Dans l'approche procédurale, il est possible de distinguer deux mondes différents et complètement séparés : le monde **des données et le monde du code**. Le monde des données est peuplé de variables de différents types, tandis que le monde du code est composé de code regroupé en modules et fonctions. Les fonctions ont la capacité d'utiliser des données, mais pas l'inverse. En outre, les fonctions sont capables d'abuser des données, c'est-à-dire d'utiliser la valeur de manière non autorisée (par exemple, lorsque la fonction sinusoïdale obtient un solde de compte bancaire comme paramètre, l'intérêt est léger mais c'est pour l'exemple).

Nous venons de se mettre d'accord sur le fait que les données ne peuvent pas utiliser de fonctions. Mais est-ce tout à fait vrai? Existe-t-il des types spéciaux de données qui peuvent utiliser des fonctions ?

Oui, il y a, elles sont nommées méthodes. Ce sont des fonctions qui sont invoquées à l'intérieur des données, pas simplement dans le même programme. Si vous pouvez voir cette distinction, vous avez fait le premier pas dans la POO.

L'approche objet suggère une façon de penser complètement différente. Les données et le code sont enfermés ensemble dans le même monde, divisés en classes.

Chaque **classe est comme une recette qui peut être utilisée lorsque vous voulez créer un objet utile** (c'est de là que vient le nom de l'approche). Vous pouvez produire autant d'objets que nécessaire pour résoudre votre problème. Chaque objet a un ensemble d'éléments (ils sont appelés propriétés ou attributs - nous utiliserons les deux mots synonymes) et est capable d'effectuer un ensemble d'activités (appelées méthodes).

Les recettes comme dans la vraie vie, peuvent être modifiées si elles sont inadéquates à des fins spécifiques et dès lors, de nouvelles classes peuvent être créées. Ces nouvelles classes héritent des propriétés et des méthodes des originaux et en ajoutent généralement de nouvelles, créant ainsi de nouveaux outils plus spécifiques.

Les objets sont des incarnations d'idées exprimées dans les classes, comme un bon gâteau au chocolat dans votre assiette est une incarnation de l'idée exprimée dans une recette imprimée dans un vieux livre de cuisine.

Les objets interagissent les uns avec les autres, échangeant des données ou activant leurs méthodes. Une classe correctement construite (et donc ses objets) est capable de protéger les données sensibles et de les cacher des modifications non autorisées. A chacun son jardin privé !! Ou le fameux : « Touche pas à ça petit con »

Il n'y a pas de frontière claire entre les données et le code : elles ne font qu'un dans les objets. Nous sommes donc dans un monde différent de celui que vous connaissiez.

Tous ces concepts ne sont pas aussi abstraits que vous pourriez le soupçonner à première vue. Au contraire, ils sont tous tirés d'expériences de la vie réelle et sont donc extrêmement utiles en programmation informatique: ils ne créent pas de vie artificielle - **ils reflètent des faits, des relations et des circonstances réels.**

1.3 Les classes et la Hiérarchie

Le mot *classe* a plusieurs significations, mais toutes ne sont pas compatibles avec les idées que nous allons expliquer ici. Le mot **classe** qui nous intéresse est comme une **catégorie**, classé en raison de similitudes précisément définies. Nous allons essayer de reprendre quelques exemples de classes qui sont de bons exemples de ce concept

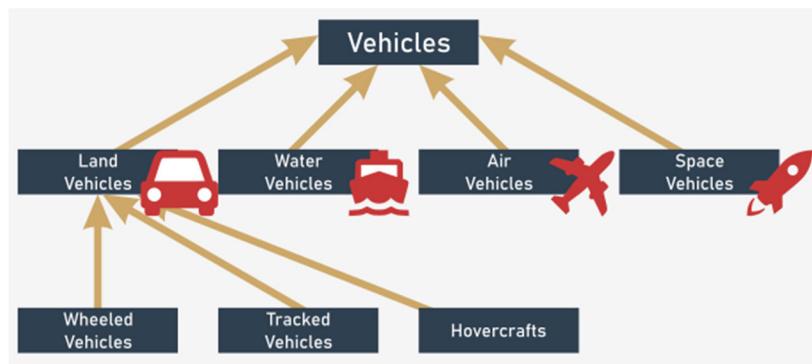


Figure 1 : Exemple de hiérarchie de classes

Regardons un instant les différents véhicules. Tous les véhicules existants (et ceux qui n'existent pas encore) sont **liés par une seule caractéristique importante**: la capacité de se déplacer. Vous pouvez débattre entre vu sur le fait qu'un chien bouge aussi; Mais un chien est-il un véhicule? Non, donc cette caractéristique pour un chien sera différente.

Nous allons donc améliorer la définition, c'est-à-dire l'enrichir d'autres critères, distinguer les véhicules des autres êtres/objets pouvant se déplacer et créer une connexion plus forte.

Prenons en considération les circonstances suivantes : les véhicules sont des entités artificiellement créées utilisées pour le transport, déplacées par les forces de la nature et dirigées (conduites) par les humains.

Si on se réfère à cette définition, un chien n'est donc pas un véhicule.

La classe *des véhicules* est très large. Trop large. Nous devons donc définir des classes plus **spécialisées**. Les classes spécialisées sont des **sous-classes**. La classe *des véhicules* sera dite **superclasse** pour tous.

Remarque: **La hiérarchie se développe de haut en bas, comme les racines des arbres, pas comme les branches.**

La classe la plus générale et la plus large est toujours au sommet (la superclasse) tandis que ses descendants sont situés en dessous (les sous-classes).

À présent, vous pouvez probablement indiquer certaines sous-classes potentielles pour la superclasse Véhicules.

Il existe de nombreuses classifications possibles.

Nous allons choisir des sous-classes en fonction de l'environnement, et disons qu'il y a (au moins, nous allons nous limiter pour l'exemple) quatre sous-classes:

- les véhicules terrestres;
- les véhicules aquatiques;
- les véhicules aériens;
- les véhicules spatiaux.

Dans cet exemple, nous allons uniquement nous concentrer sur la première sous-classe, la classe : véhicules terrestres. Si vous le souhaitez, vous pouvez continuer avec les autres comme exercice.

Les véhicules terrestres peuvent être divisés en davantage de sous-classes, selon par exemple, la méthode avec laquelle ils vont rouler.

Ainsi, nous pouvons énumérer:

- véhicules à roues;
- véhicules à chenilles;
- aéroglisseurs.

La hiérarchie que nous avons créée est illustrée par la figure 1.

Notez la direction des flèches - elles pointent toujours vers la superclasse. La classe de niveau supérieur est une exception si on veut, car elle n'a pas de superclasse.

1.4 Exemple 2

Un autre exemple est la hiérarchie du règne taxonomique des animaux.

Nous pouvons dire que tous les *animaux* (notre classe de premier niveau) peuvent être divisés en cinq sous-classes:

- mammifères;
- Reptiles;
- oiseaux;
- poisson;
- Amphibiens.

Nous allons prendre la première classe pour une analyse plus approfondie. Nous avons identifié les sous-classes suivantes :

- mammifères sauvages;
- mammifères domestiques.

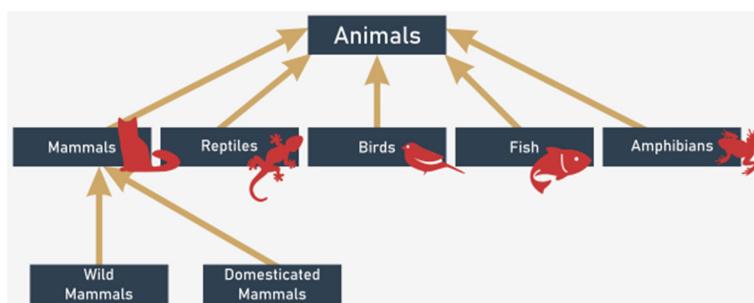


Figure 2 : Exemple de hiérarchie avec les animaux
Essayez d'étendre la hiérarchie comme vous le souhaitez et trouvez le bon endroit pour les humains.

1.5 Qu'est-ce qu'un objet ?

Une classe (parmi d'autres définitions) est un **ensemble d'objets**. Un objet est **un être appartenant à une classe**.

Un objet est une **incarnation des exigences, des traits et des qualités attribués à une classe spécifique**. Cela peut sembler simple, mais notez les circonstances . Les classes forment une hiérarchie.
Cela peut signifier qu'un objet appartenant à une classe spécifique appartient à toutes les superclasses en même temps. Cela peut également signifier que tout objet appartenant à une superclasse ne peut appartenir à aucune de ses sous-classes.

Par exemple: toute voiture personnelle est un objet appartenant à la classe des *véhicules à roues*. Cela signifie également que la même voiture appartient à toutes les superclasses de sa classe d'origine; Par conséquent, il est également membre de la classe *des véhicules*.

Votre chien (ou votre chat) est un objet inclus dans la classe des *mammifères domestiques*, ce qui signifie explicitement qu'il est également inclus dans la classe *des animaux*.

Chaque **sous-classe est plus spécialisée** (ou plus spécifique) que sa superclasse. Inversement, chaque **superclasse est plus générale** (plus abstraite) que n'importe laquelle de ses sous-classes.

Notez que nous avons supposé qu'une classe ne peut avoir qu'une seule superclasse - ce n'est pas toujours vrai, mais nous discuterons de cette question plus tard.

1.6 L'héritage

Définissons l'un des concepts fondamentaux de la programmation objet, nommé **héritage**. Tout objet lié à un niveau spécifique d'une hiérarchie de classes **hérite de tous les traits (ainsi que les exigences et les qualités) définis à l'intérieur de l'une des superclasses**.

La classe d'origine de l'objet peut définir de nouveaux traits (ainsi que des exigences et des qualités) qui seront hérités par l'une de ses sous-classes.

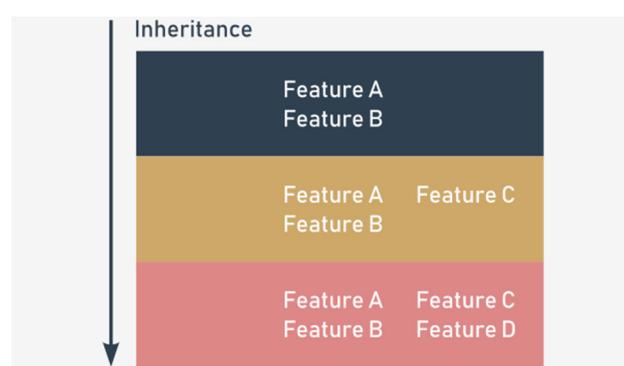


Figure 3 : Schématisation de l'héritage

1.7 De quoi est composé un objet ?

La convention de programmation objet suppose que **chaque objet existant peut être équipé de trois groupes d'attributs** :

- Un objet a un **nom** qui l'identifie de manière unique dans son espace de noms d'origine (bien qu'il puisse également y avoir des objets anonymes)
- Un objet possède **un ensemble de propriétés(attributs) individuelles** qui le rendent original, unique ou exceptionnel (bien qu'il soit possible que certains objets n'aient aucune propriété)
- Un objet a **un ensemble de capacités(méthodes) pour effectuer des activités spécifiques**, capable de changer l'objet lui-même, ou certains des autres objets.

Il y a un indice (bien que cela ne fonctionne pas toujours) qui peut vous aider à identifier l'une des trois sphères ci-dessus. Chaque fois que vous décrivez un objet et que vous utilisez :

- un nom – vous définissez probablement le nom de l'objet;
- un adjectif – vous définissez probablement la propriété de l'objet;
- un verbe – vous définissez probablement l'activité de l'objet.

Deux exemples de phrases devraient servir à mieux comprendre :

Une Cadillac rose roulait vite.

Nom de l'objet = Classe Cadillac

Classe Principale = Véhicules à roues

Propriété = Couleur (rose)

Activité = Rouler (rapidement)

MiaouMiaou est un gros chat qui dort toute la journée.

Nom de l'objet = MiaouMiaou

Classe principale = Chat

Propriété = Taille (grande)

Activité = Sommeil (toute la journée)

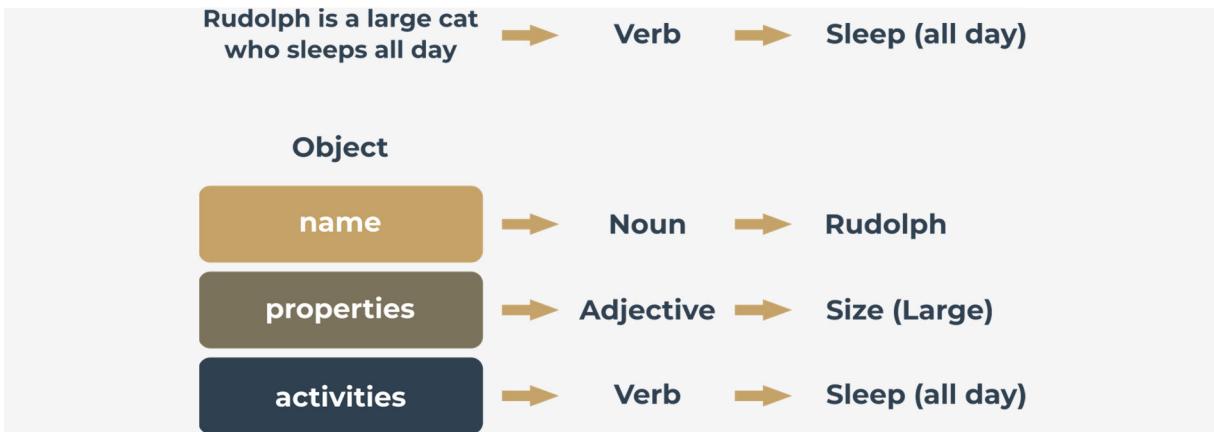


Figure 4 : Composition d'un objet

1.8 Notre première classe

La programmation objet **est l'art de définir et de développer des classes**. Une classe est un modèle d'une partie très spécifique de la réalité, reflétant les propriétés et les activités trouvées dans le monde réel.

Les classes définies au début sont trop générales et imprécises pour couvrir le plus grand nombre possible de cas réels.

Rien ne s'oppose à la définition de nouvelles sous-classes plus précises. Ils hériteront de tout de leur superclasse, afin que le travail qui a été consacré à sa création ne soit pas gaspillé.

La nouvelle classe peut ajouter de nouvelles propriétés et de nouvelles activités, et peut donc être plus utile dans des applications spécifiques. De toute évidence, il peut être utilisé comme superclasse pour un nombre quelconque de sous-classes nouvellement créées.

Le processus n'a pas besoin d'avoir une fin. Vous pouvez créer autant de classes que nécessaire.

La classe que vous définissez n'a rien à voir avec l'objet : **l'existence d'une classe ne signifie pas que l'un des objets compatibles sera automatiquement créé**. La classe elle-même n'est pas capable de créer un objet - vous devez le créer vous-même, et Python vous permet de le faire.

Il est temps de définir la classe la plus simple et de créer un objet. Jetez un coup d'œil à l'exemple ci-dessous :

```
class MaPremiereClass :
    pass
```

Nous venons de définir notre première classe. La classe est plutôt pauvre : elle n'a ni propriétés ni méthodes. C'est **vide**, en fait, mais cela n'a pas d'importance pour l'instant. Plus la classe est simple, plus elle sera facile à comprendre.

La définition commence par le mot clé : `class`. Il est suivi d'un **identifiant qui**

nommera la classe (*note: ne le confondez pas avec le nom de l'objet - ce sont deux choses différentes*).

Ensuite, vous ajoutez deux **points** (:), car les classes, comme les fonctions, forment leur propre bloc imbriqué par la dé-indentation. Le contenu à l'intérieur du bloc définit toutes les propriétés et méthodes de la classe.

Le mot-clé *pass* remplit notre classe de « rien ». Il ne contient aucune méthode ou propriété.

1.9 Notre premier Objet

La classe nouvellement définie devient un outil capable de créer de nouveaux objets. L'outil doit être utilisé explicitement, à la demande.

Imaginez que vous souhaitez créer un (exactement un) objet de la classe `MaPremiereClass`.

Pour ce faire, vous devez affecter une variable pour stocker l'objet nouvellement créé de cette classe et créer un objet en même temps. Nous procéderons comme suit :

```
premier_objet = MaPremiereClass()
```

Remarques :

- Le nom de la classe Nous fait croire qu'il s'agit d'une fonction - Pouvez-vous voir cela?
- l'objet nouvellement créé est équipé de tout ce que la classe comporte; Comme cette classe est complètement vide, l'objet l'est également.

L'acte de créer un objet depuis une classe est également appelé une **instanciation** (car l'objet devient une **instance de la classe**).

Laissons les classes tranquilles pendant un court instant, car nous allons maintenant vous dire quelques mots sur *les piles*. Nous savons que le concept de classes et d'objets n'est peut-être pas encore tout à fait clair. Ne vous inquiétez pas, nous vous expliquerons tout très bientôt. Ceci n'est en soit qu'un petit rappel du cours théorique.

1.10 Principaux points à retenir

1. Une **classe** est une idée (plus ou moins abstraite) qui peut être utilisée pour créer un certain nombre d'incarnations – une telle incarnation est appelée un **objet**.

2. Lorsqu'une classe est dérivée d'une autre classe, sa relation est nommée **héritage**. La classe qui dérive de l'autre classe est nommée sous-classe. Le deuxième côté de cette relation est nommée **superclasse**. Une façon de présenter une telle relation est un **diagramme d'héritage**, où:

les superclasses sont toujours présentées **au-dessus de** leurs sous-classes;
Les relations entre les classes sont représentées sous forme de flèches dirigées
de la sous-classe vers sa superclasse.

3. Les objets sont équipés de:

- un **nom** qui les identifie et nous permet de les distinguer;
- un ensemble de **propriétés** (l'ensemble peut être vide)
- un ensemble de **méthodes** (peut également être vide)

4. Pour définir une classe Python, vous devez utiliser le mot-clé class. Par exemple:

```
Classe This_Is_A_Class:  
    passer
```

5. Pour créer un objet de la classe précédemment définie, vous devez utiliser la classe comme s'il s'agissait d'une fonction. Par exemple:

```
this_is_an_object = This_Is_A_Class()
```

2 Les piles pour comprendre l'approche objet

Une pile est une structure développée pour stocker des données d'une manière très spécifique. Imaginez une pile de pièces de monnaie. Vous ne pouvez pas mettre une pièce ailleurs que sur le dessus de la pile.

De même, vous ne pouvez pas obtenir une pièce de monnaie de la pile à partir d'un endroit autre que le haut de la pile. Si vous voulez obtenir la pièce qui se trouve sur le fond, vous devez retirer toutes les pièces des niveaux supérieurs. Le nom alternatif pour une pile (mais uniquement dans la terminologie informatique) est **LIFO**.

C'est une abréviation pour une description très claire du comportement de la pile: **Last In - First Out**. La pièce qui est arrivée en dernier sur la pile partira en premier.

Une pile est un objet avec deux opérations(**méthodes**) élémentaires, classiquement appelées **push** (lorsqu'un nouvel élément est placé sur le dessus) et **pop** (lorsqu'un élément existant est retiré du haut).

Les piles sont très souvent utilisées dans de nombreux algorithmes classiques, et il est difficile d'imaginer la mise en œuvre de nombreux outils largement utilisés sans l'utilisation de piles.

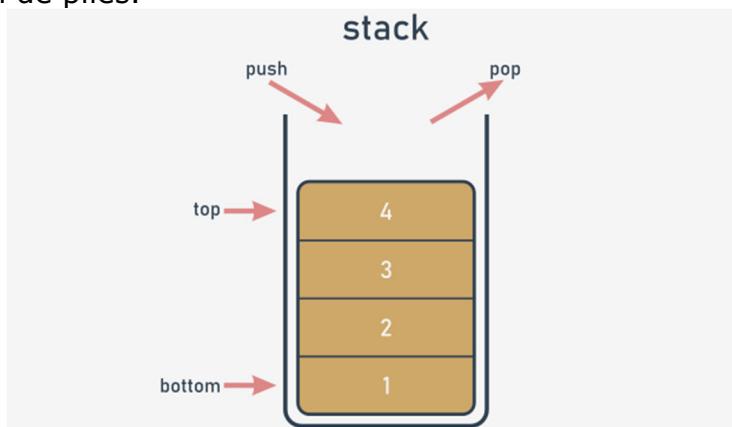


Figure 5 : Une pile

2.1 L'approche Objet

Contrairement à d'autres langages de programmation, Python n'a aucun moyen de vous permettre de déclarer une méthode par défaut.

Au lieu de cela, vous devez ajouter une instruction spécifique. Les méthodes doivent être ajoutées manuellement à la classe.

Comment garantissez-vous qu'une telle activité ait lieu chaque fois que la nouvelle pile est créée ?

Il existe un moyen simple de le faire, vous devez **doter la classe d'une fonction spécifique** - sa spécificité est double:

- Elle doit être nommée de manière stricte;

- Elle est invoquée implicitement, lors de la création du nouvel objet.

Une telle fonction est appelée **constructeur** (ça on l'a vu en théorie), car son objectif général est de **construire un nouvel objet**. Le constructeur doit tout savoir sur la structure de l'objet et doit effectuer toutes les initialisations nécessaires.

Pour une pile sera pourrait être :

```

1 class Stack: # Defining the Stack class.
2     def __init__(self): # Defining the constructor function.
3         print("Hi!")
4
5
6 stack_object = Stack() # Instantiating the object.
7

```

Conclusion :

- Le nom du constructeur est toujours `__init__`;
- il doit avoir **au moins un** paramètre (nous en discuterons plus tard); le paramètre est utilisé pour représenter l'objet nouvellement créé - vous pouvez utiliser le paramètre pour manipuler l'objet et l'enrichir avec les méthodes ou attributs nécessaires; vous l'utiliserez bientôt;
- Remarque: Le paramètre obligatoire est généralement nommé `self` - ce n'est qu'une **convention, mais vous devez la suivre** - il simplifie le processus de lecture et de compréhension de votre code.

→Attention, à la ligne 6 on remarque bien qu'aucun appel explicite du constructeur n'a été fait, ceci est automatique et implicite.

- Toute Modification apportez au constructeur impactera l'état du paramètre `self` et sera répercuté sur l'objet nouvellement créé.
- L'appel d'un attribut ou d'une méthode de notre objet se fera avec la commande : **`self.Methode` ou `self.attribut`**
- Quand on appelle un attribut dans un constructeur, on l'instancie et il est donc disponible dans toute notre classe.

Testons notre class :

```

1 class Stack:
2     def __init__(self):
3         self.stack_list = []
4
5
6 stack_object = Stack()
7 print(len(stack_object.stack_list))

```

Nous avons donc bien un constructeur qui va instancier une liste vide.

Nous essayons d'accéder à cette liste, et est-ce que cela fonctionne ? (Je vous rappel que nous voudrions que nos attributs soient cachés(privés) et donc cela ne devrait pas.

Et bien oui cela fonctionne... Il va donc falloir changer cela, c'est très facile mais ... pas du tout intuitif en python.

Nous allons donc revenir sur la notion d'encapsulation.

2.2.1 L'encapsulation et ce qui l'entoure

En programmation orientée objet, on distingue différents types d'attributs: les **attributs publics**, les **attributs protégés** et les **attributs privés**. Nous reviendrons sur les attributs protégés rapidement lorsque nous parlerons **d'héritage** mais la différence importante est entre les attributs privés et les attributs publics.

Un attribut privé n'est accessible **qu'à l'intérieur de la définition de la classe**:

Je ne pourrais y accéder (pour lecture ou écriture) que dans la définition des différentes méthodes. A contrario, les attributs publics sont accessibles partout et toujours.

Dans un langage comme Java, cette différence est très stricte. Lorsqu'on utilise Python, on ne rend jamais totalement un attribut privé: Python est un langage beaucoup plus ouvert pour "adultes consentants", c'est-à-dire qu'on fait en quelque sorte confiance à l'utilisateur final pour qu'il ne cherche pas à détruire le code.

Mais il existe tout de même des façons de faire en sorte que les attributs soient moins accessibles.

- Attributs privés en Python

Nous allons prendre un exemple qui va nous permettre de créer un attribut privé **âge** (après tout, pourquoi aurait-on le droit de demander son âge à un animal qu'on ne connaît pas ?).

Pour définir un attribut privé, on va nommer cet attribut en commençant par `_` mais faites attention, car si vous terminez le nom de l'attribut par `_` aussi, alors il n'est plus considéré comme privé: il a simplement un nom plus compliqué.

```
class Animal:  
    def __init__(self):  
        self.__age = 0  
  
    def vieillir(self):  
        self.__age = self.__age + 1  
animal = Animal()  
  
animal.vieillir()  
  
print(animal.__age)
```

La dernière ligne devrait générer une exception **AttributeError**:
L'attribut `__age` n'existe pas... Mais on remarque que la méthode **vieillir** a bien fonctionné: l'attribut `__age` existe bien à l'intérieur de la définition de la méthode.

On a donc bien créé un attribut privé... ou alors il est simplement caché. En fait, il est disponible mais sous un autre nom: `_Animal__age`.

```
class Animal:  
    def __init__(self):  
        self.__age = 0  
  
    def vieillir(self):  
        self.__age = self.__age + 1  
animal = Animal()  
  
animal.vieillir()  
  
print(animal._Animal__age)
```

Dans d'autres langages de programmation, c'est ici que les **accesseurs** et les **mutateurs** jouent un rôle très important puisque c'est grâce à eux qu'**on va pouvoir lire ou modifier ces attributs**.

Mais nous pouvons tout de même les définir nous-mêmes:

```
class Animal:  
    def __init__(self):  
        self.__age = 0  
  
    def vieillir(self):  
        self.__age = self.__age + 1  
  
    def get_age(self):  
        print('accesseur')  
        return self.__age  
  
    def set_age(self, valeur):  
        print('mutateur')  
        if type(valeur) == int:  
            self.__age = valeur  
        else:  
            print('mauvais type')  
animal = Animal()  
  
animal.vieillir()  
  
print(animal.get_age())  
  
animal.set_age(1.5)
```

Dans cet exemple, les méthodes **get_age** et **set_age** servent d'accesseur et de mutateur. On voit l'intérêt d'un mutateur pour contrôler le type d'un attribut, ce qui pour un langage typé dynamiquement comme Python peut présenter un avantage.

- Définir proprement accesseurs et mutateurs

Dans cette partie, nous allons utiliser les décorateurs (un design pattern bien connu que vous verrez en bâ2 mais que vous allez comprendre).

Pour forcer l'accès aux attributs via les getters et les setters, on peut utiliser la classe pré-construite **property**:

```
class Animal:  
    def __init__(self):  
        self.age = 0  
  
    @property  
    def age(self):  
        print('on retourne la valeur depuis le getter')  
        return self._age  
  
    @age.setter  
    def age(self, valeur):  
        print('on modifie la valeur depuis le setter')  
        if type(valeur) == int:  
            self._age = valeur  
        else:  
            print('mauvais type')  
  
animal = Animal()  
  
print(animal.age)  
  
animal.age = 10  
  
animal.age = -1.5
```

La première définition de la méthode **age** permet de définir le **getter** et la seconde le **setter**. On remarque alors dans les lignes suivantes, que ce sont bien ces méthodes qui sont appelées lorsqu'on modifie ou appelle ces attributs.

Attention: dans ce code, l'attribut age est toujours public. Ce code permet simplement de montrer comment définir proprement et simplement accesseur et mutateur.

- Encapsulation des méthodes

Le même principe d'encapsulation s'applique aux méthodes: on peut définir des méthodes privées, protégées ou publiques. Les méthodes publiques sont toujours accessibles alors que les méthodes privées ne sont accessibles qu'à l'intérieur de la classe.

Le principe en Python est le même: on utilise __ au début du nom de la méthode (et pas à la fin) et la méthode est toujours retrouvable sur le modèle de _NomDeLaClasse__NomDeLaMethode.

Il faut toutefois garder à l'esprit que ce sont des concepts de Programmation Orientée Objet "pure". Python ne respecte pas vraiment ces principes puisqu'on peut toujours avoir accès à des attributs ou méthodes privés. Dans un langage vraiment orienté objet comme Java ou c#, on ne peut pas tricher.

2.2.1 Notion de Visibilité

La visibilité d'un attribut ou d'une opération permet de déterminer qui pourra y accéder. En outre, C'est une notion fondamentale pour le concept objet d'encapsulation. On a 3 types de visibilité:

- « + » ou public : tout le monde peut y accéder
- « # » ou protected : seules la classe possédant l'élément et les classes filles peuvent y accéder (voir notion d'héritage)
- « - » ou private : seule la classe possédant l'élément peut y accéder

Règle :

- Les attributs doivent être privés
- Les opérations définissant l'interface d'une classe doivent être publique (services offerts)
- Des opérations peuvent aussi être privées si elles ne participent qu'au fonctionnement interne de la classe

2.2.2 Le self

Toutes les méthodes doivent avoir ce paramètre. Il joue le même rôle que le premier paramètre du constructeur.

Il permet à la méthode d'accéder aux entités (attributs et méthodes) effectuées par l'objet réel. Vous ne pouvez pas l'omettre. Chaque fois que Python appelle une méthode, il envoie implicitement l'objet courant comme premier argument.

Cela signifie qu'une **méthode est obligée d'avoir au moins un paramètre, qui est utilisé par Python : self** - vous n'avez aucune influence sur lui.

Si votre méthode n'a besoin d'aucun paramètre, celui-ci doit être spécifié de toute façon. S'il est conçu pour traiter un seul paramètre, vous devez en spécifier deux, et le rôle du premier est toujours de s'appeler lui-même.

2.3 Testons et comprenons notre exemple sur les piles complété

```
1  class Stack:
2      def __init__(self):
3          self.__stack_list = []
4
5      def push(self, val):
6          self.__stack_list.append(val)
7
8      def pop(self):
9          val = self.__stack_list[-1]
10         del self.__stack_list[-1]
11         return val
12
13
14 little_stack = Stack()
15 another_stack = Stack()
16 funny_stack = Stack()
17
18 little_stack.push(1)
19 another_stack.push(little_stack.pop() + 1)
20 funny_stack.push(another_stack.pop() - 2)
21
22 print(funny_stack.pop())
```

Que se passe-t-il ? Etais-ce prévisible ?

2.3 Principaux points à retenir

1. Une **pile** est un objet conçu pour stocker des données à l'aide du modèle **LIFO**. La pile accomplit généralement au moins deux opérations, nommées **push()** et **pop()**.
2. Une **méthode** de classe est en fait une fonction déclarée à l'intérieur de la classe et capable d'accéder à tous les composants de la classe.
3. La partie de la classe Python responsable de la création de nouveaux objets est appelée le **constructeur**, et elle est implémentée comme une méthode du nom `__init__`.
5. Chaque déclaration de méthode de classe doit contenir au moins un paramètre (toujours le premier) habituellement appelé `self`, et est utilisé par les objets pour s'identifier.
6. Si nous voulons cacher l'un des composants d'une classe au monde extérieur, nous devons commencer son nom par `_`. Ces composants sont appelés **privés**.

3 Propriétés de la POO

3.1 Variables d'instance

En général, une classe peut être équipée de deux types de données différents pour former les propriétés d'une classe.

Ce type de propriété de classe existe lorsque et uniquement lorsqu'elle est explicitement créée et ajoutée à un objet. Comme vous le savez déjà, cela peut être fait lors de l'initialisation de l'objet, effectuée par le constructeur. Mais, cela peut aussi être fait à n'importe quel moment de la vie de l'objet et toute propriété existante peut être enlevée à tout moment.

Une telle approche a des conséquences importantes :

- Différents objets de la même classe **peuvent posséder différents ensembles de propriétés**;
- Il doit y avoir un moyen de **vérifier en toute sécurité si un objet spécifique possède la propriété** que vous souhaitez utiliser (sauf si vous voulez provoquer une exception - cela est toujours possible mais rare)
- Chaque objet **porte son propre ensemble de propriétés** - elles n'interfèrent en aucune façon les unes avec les autres.

Ces variables (propriétés/attributs) sont appelées **variables d'instance**.

Le mot *instance* suggère qu'ils sont étroitement liés aux objets (qui sont des instances de classe), pas aux classes elles-mêmes.

Regardons cela de plus près avec un exemple :

```
1 class ExampleClass:  
2     def __init__(self, val = 1):  
3         self.first = val  
4  
5     def set_second(self, val):  
6         self.second = val  
7  
8  
9 example_object_1 = ExampleClass()  
10 example_object_2 = ExampleClass(2)  
11  
12 example_object_2.set_second(3)  
13  
14 example_object_3 = ExampleClass(4)  
15 example_object_3.third = 5  
16  
17 print(example_object_1.__dict__)  
18 print(example_object_2.__dict__)  
19 print(example_object_3.__dict__)
```

Il faut une explication supplémentaire avant d'entrer dans les détails. Jetez un coup d'œil aux trois dernières lignes du code.

Les objets Python, lorsqu'ils sont créés, sont dotés **d'un petit ensemble d'attributs et de méthodes prédéfinies**. Chaque objet les a, que vous le vouliez ou non. L'une d'elles est une variable nommée `dict` (c'est un dictionnaire).

La variable contient les noms et les valeurs de tous les attributs (variables) que l'objet possède actuellement. Vous pouvez l'utiliser pour présenter en toute sécurité le contenu d'un objet.

Plongeons dans le code maintenant:

- la classe nommée `ExampleClass` possède un constructeur, qui **crée inconditionnellement une variable d'instance** nommée `first`, et la définit avec la valeur transmise par le premier argument (du point de vue de l'utilisateur de la classe) ou le second argument (du point de vue du constructeur) ; notez la valeur par défaut du paramètre - toute astuce que vous pouvez faire avec un paramètre de fonction en procédural peut être appliquée aux méthodes, aussi;
- La classe possède également une **méthode qui crée une autre variable d'instance**, nommée `second` ;
- nous avons créé trois objets de la classe `ExampleClass`, mais toutes ces instances sont différentes :
 - `example_object_1` seulement la propriété est nommée `en premier`;
 - `example_object_2` a deux propriétés: `première` et `deuxième`;
 - `example_object_3` a été enrichi d'une propriété nommée `troisième` juste à la volée, en dehors du code de la classe - c'est possible et tout à fait admissible.

Les résultats du programme montrent clairement que nos hypothèses sont correctes - les voici:

```
{first: 1}  
{'second': 3, 'first': 2}  
{'third': 5, 'first': 4}
```

Il y a une conclusion supplémentaire qui devrait être tirée : **la modification d'une variable d'instance d'un objet n'a aucun impact sur tous les autres objets de cette instance**. Les variables d'instance sont parfaitement isolées les unes des autres.

3.1.1 Examinons le code suivant :

```
1  class ExampleClass:
2      def __init__(self, val = 1):
3          self.__first = val
4
5      def set_second(self, val = 2):
6          self.__second = val
7
8
9  example_object_1 = ExampleClass()
10 example_object_2 = ExampleClass(2)
11
12 example_object_2.set_second(3)
13
14 example_object_3 = ExampleClass(4)
15 example_object_3.__third = 5
16
17
18 print(example_object_1.__dict__)
19 print(example_object_2.__dict__)
20 print(example_object_3.__dict__)
```

Jetez un coup d'œil à l'exemple ci-dessus :

C'est presque la même chose que le précédent. La seule différence réside dans les noms des attributs. Nous avons **ajouté deux traits de soulignement (__)** devant eux.

Comme vous le savez, un tel ajout rend l'instance **variable privée** - elle devient inaccessible depuis le monde extérieur.

Le comportement réel de ces noms est un peu plus compliqué, alors exécutons le programme. Voici le résultat :

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

On remarque des noms étranges pleins de traits de soulignement. D'où viennent-ils?

Lorsque Python voit que vous souhaitez ajouter une variable d'instance à un objet et que vous allez le faire dans l'une des méthodes de l'objet, il **perturbe l'opération** de la manière suivante :

- il met le nom de classe avant le nom de votre attribut;
- Il met un trait de soulignement supplémentaire au début.

C'est pourquoi le `__first` devient `_ExampleClass__first`.

Le nom est maintenant entièrement accessible depuis l'extérieur de la classe.

Vous pouvez exécuter un code comme celui-ci :

```
print(example_object_1.__ExampleClass__first)
```

et vous obtiendrez un résultat valide sans erreurs ni exceptions.

Comme vous pouvez le constater, rendre une propriété privée est donc très limité en python.

La manipulation ne fonctionne pas si vous ajoutez une variable d'instance privée en dehors du code de classe.

Dans ce cas, il se comportera comme n'importe quel autre attribut traditionnel.

3.2 Variables de classes

Une variable de classe est **un attribut qui existe en une seule copie et qui est stockée en dehors de tout objet**.

Remarque : aucune variable d'instance n'existe s'il n'y a pas d'objet dans la classe ; Une variable de classe existe dans une copie même s'il n'y a aucun objet dans la classe.

Les variables de classe sont créées différemment des instances de classes.

Un exemple pour mieux voir cela :

```
1 class ExampleClass:
2     counter = 0
3     def __init__(self, val = 1):
4         self.__first = val
5         ExampleClass.counter += 1
6
7
8 example_object_1 = ExampleClass()
9 example_object_2 = ExampleClass(2)
10 example_object_3 = ExampleClass(4)
11
12 print(example_object_1.__dict__, example_object_1.counter)
13 print(example_object_2.__dict__, example_object_2.counter)
14 print(example_object_3.__dict__, example_object_3.counter)
```

- Il y a une affectation dans la première ligne de la définition de classe - elle définit le `compteur` de variable à 0; l'initialisation de la variable à l'intérieur de la classe mais en dehors de l'une de ses méthodes fait de la variable une variable de classe;

- L'accès à une telle variable ressemble à l'accès à n'importe quel attribut d'instance - vous pouvez le voir dans le corps du constructeur ; Comme vous pouvez le voir, le constructeur incrémente la variable d'une unité; En effet, la variable compte tous les objets créés

La sortie de ce code nous donnera :

```
{'__ExampleClass__first': 1} 3
{'__ExampleClass__first': 2} 3
{'__ExampleClass__first': 4} 3
```

Deux conclusions importantes ressortent de l'exemple :

- Les variables de classe ne sont pas affichées dans le `__dict__` d'un objet (c'est naturel car les variables de classe ne font pas partie d'un objet) mais vous pouvez toujours essayer de regarder dans la variable du même nom, mais au niveau de la classe - nous vous le montrerons très bientôt;
- Une variable **de classe présente toujours la même valeur** dans toutes les instances de classe (objets)
- On peut ajouter les deux underscore devant une variable de classe, cela aura le même effet que précédemment.

Nous vous avons déjà dit que les variables de classe existent même lorsqu'aucune instance de classe (un objet) n'a été créée.

Maintenant, nous allons profiter de l'occasion pour vous montrer la **différence entre les deux variables** `__dict__`, celle de la classe et celle de l'objet. Regardez le code ci-dessous.

```
1 class ExampleClass:
2     varia = 1
3     def __init__(self, val):
4         ExampleClass.varia = val
5
6
7 print(ExampleClass.__dict__)
8 example_object = ExampleClass(2)
9
10 print(ExampleClass.__dict__)
11 print(example_object.__dict__)
```

- Nous définissons une classe nommée `ExampleClass` ;
- La classe définit une variable de classe nommée `varia` ;
- Le constructeur de la classe définit la variable avec la valeur du paramètre

- La dénomination de la variable est l'aspect le plus important de l'exemple car :
 - Changer l'affectation en `self.varia = val` créerait une variable d'occurrence du même nom que celle de la classe ;
 - Changer l'affectation en `varia = val` fonctionnerait sur la variable locale d'une méthode; (Nous vous encourageons fortement à tester les deux cas ci-dessus - cela vous permettra de vous souvenir plus facilement de la différence et de bien la comprendre)
- La première ligne du code hors classe imprime la valeur de l'attribut `ExampleClass.varia` ; remarque : nous utilisons la valeur avant que le tout premier objet de la classe ne soit instancié.

Si on exécute le code ci-dessus, on peut voir, le `__dict__` de la classe contient beaucoup plus de données que son homologue objet.
 La plupart des données sont inutiles - celui que nous voulons que vous vérifiiez soigneusement montre la valeur actuelle de notre variable `varia`.
 Notez que l'objet `__dict__` est vide --> l'objet n'a pas de variables d'instance.

3.3 Vérification de l'existence d'un attribut

L'attitude de Python vis-à-vis de l'instanciation d'objets soulève un problème important - contrairement à d'autres langages de programmation, vous **ne pouvez pas vous attendre à ce que tous les objets de la même classe aient les mêmes ensembles d'attributs**.

Tout comme dans l'exemple de ci-dessous.

L'objet créé par le constructeur ne peut avoir qu'un seul des deux attributs possibles : `a` ou `b`.

```

1 class ExampleClass:
2     def __init__(self, val):
3         if val % 2 != 0:
4             self.a = 1
5         else:
6             self.b = 1
7
8
9 example_object = ExampleClass(1)
0
1 print(example_object.a)
2 print(example_object.b)

```

L'exécution du code produira la sortie suivante :

```

1
Traceback (most recent call last):
File ".main.py", line 11, in 
print(example_object.b) AttributeError: 'ExampleClass'
object has no attribute 'b'

```

On constate donc que la tentative d'accès à un attribut non existant provoque une erreur.

Pour éviter ce cas, il faudrait peut-être faire appel à notre bon vieux try-except

```
1 class ExampleClass:  
2     def __init__(self, val):  
3         if val % 2 != 0:  
4             self.a = 1  
5         else:  
6             self.b = 1  
7  
8  
9 example_object = ExampleClass(1)  
10 print(example_object.a)  
11  
12 try:  
13     print(example_object.b)  
14 except AttributeError:  
15     pass
```

Comme vous pouvez le voir, cette action n'est pas très sophistiquée, ça fonctionne mais nous venons en fait de balayer la question sous le tapis. Un peu comme un étudiant lors d'un examen !

Heureusement, il existe une autre façon (la bonne) pour faire face à ce problème.

Python fournit une **fonction capable de vérifier en toute sécurité si un objet / classe contient une propriété spécifiée**. La fonction est nommée `hasattr` et s'attend à ce que deux arguments lui soient transmis :

- la classe ou l'objet contrôlé ;
- le nom de la propriété dont l'existence doit être signalée (remarque : il doit s'agir d'une chaîne contenant le nom de l'attribut, pas seulement du nom)

La fonction renvoie True ou False.

Voici un exemple d'utilisation pour l'attribut d'un objet :

```
1 class ExampleClass:  
2     def __init__(self, val):  
3         if val % 2 != 0:  
4             self.a = 1  
5         else:  
6             self.b = 1  
7  
8  
9 example_object = ExampleClass(1)  
10 print(example_object.a)  
11  
12 if hasattr(example_object, 'b'):  
13     print(example_object.b)
```

Voici un autre exemple pour le test sur une classe :

```
1 class ExampleClass:  
2     attr = 1  
3  
4  
5 print(hasattr(ExampleClass, 'attr'))  
6 print(hasattr(ExampleClass, 'prop'))
```

Console >_

True
False

3.4 Principaux points à retenir

1. Une **variable d'instance** est une propriété dont l'existence dépend de la création d'un objet. Chaque objet peut avoir un ensemble différent de variables d'instance. De plus, ils peuvent être librement ajoutés et retirés des objets au cours de leur vie. Toutes les variables d'instance d'objet sont stockées dans un dictionnaire dédié nommé `__dict__`, contenu dans chaque objet séparément.
2. Une variable d'instance peut être privée lorsque son nom commence par `_`, mais n'oubliez pas qu'une telle propriété est toujours accessible depuis l'extérieur de la classe à l'aide d'un **nom complet** construit comme :
`_ClassName__PrivatePropertyName.`
3. Une **variable de classe** est une propriété qui existe dans exactement une copie et qui n'a besoin d'aucun objet créé pour être accessible. Ces variables ne sont pas affichées en tant que contenu de `__dict__`.
Toutes les variables de classe d'une classe sont stockées dans un dictionnaire dédié nommé `__dict__`, contenu dans chaque classe séparément.
4. Une fonction nommée `hasattr()` peut être utilisée pour déterminer si un objet/une classe contient un attribut spécifié.

Petit résumé en code :

```
class Sample:  
    gamma = 0 # Class variable.  
    def __init__(self):  
        self.alpha = 1 # Instance variable.  
        self.__delta = 3 # Private instance variable.  
  
obj = Sample()  
obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)  
print(obj.__dict__)
```

Sortie : `{'alpha': 1, '_Sample__delta': 3, 'beta': 2}`

4 Les méthodes en détails

4.1 Récapitulatif de tout ce qu'on sait sur les méthodes

Comme vous le savez déjà, une méthode est une fonction **intégrée dans une classe**.

Il y a une exigence fondamentale - une **méthode est obligée d'avoir au moins un paramètre** (il n'existe pas de méthodes sans paramètre - une méthode peut être invoquée sans argument, mais pas déclarée sans paramètres).

Le premier (ou le seul) paramètre est généralement nommé `self`. Nous vous suggérons de suivre la convention - elle est couramment utilisée, et vous aurez probablement quelques surprises en utilisant un autre nom.

Le nom `self` suggère explicitement l'objectif du paramètre qu'il représente, il identifie l'objet pour lequel la méthode est appelée.

Si vous invoquez une méthode, vous ne devez pas passer l'argument pour le paramètre `self` - Python le définira pour vous.

```
1 class Classy:  
2     def method(self):  
3         print("method")  
4  
5  
6 obj = Classy()  
7 obj.method()
```

Le code produit : `method`

Notez la façon dont nous avons créé l'objet - nous avons traité le nom de la classe comme une fonction, renvoyant un objet nouvellement instancié de la classe.

Si vous souhaitez que la méthode accepte des paramètres autres que `self`, vous devez :

- les placer après le `self` dans la définition de la méthode;
- les spécifier pendant l'appel sans spécifier `self` (comme s'il n'existait pas)

```
1 class Classy:  
2     def method(self, par):  
3         print("method:", par)  
4  
5  
6 obj = Classy()                                     method: 1  
7 obj.method(1)                                       method: 2  
8 obj.method(2)                                       method: 3  
9 obj.method(3)                                     en sortie →
```

4.2 Les atouts de `self`

Le paramètre `self` est utilisé pour accéder aux variables d'instance et de classe de l'objet.

L'exemple montre les deux façons d'utiliser `self`:

```
1 class Classy:  
2     varia = 2  
3     def method(self):  
4         print(self.varia, self.var)  
5  
6  
7 obj = Classy()  
8 obj.var = 3  
9 obj.method()                                sortie → 2 3
```

Le paramètre `self` est également utilisé pour appeler d'autres méthodes objet/classe à partir de la classe.

```
1 class Classy:  
2     def other(self):  
3         print("other")  
4  
5     def method(self):  
6         print("method")  
7         self.other()  
8  
9  
10 obj = Classy()                            method  
11 obj.method()                             sortie → other
```

4.3 le constructeur

Une méthode comme nommée: `__init__`, ne sera pas une méthode régulière - ce sera un **constructeur**.

Si une classe possède un constructeur, il est appelé automatiquement et implicitement lorsque l'objet de la classe est instancié.

Notre constructeur :

- est **obligé d'avoir le paramètre self** ;
- **peut (mais n'a pas besoin de) avoir plus de paramètres** que `self` ; si cela se produit, la façon dont le nom de classe est utilisé pour créer l'objet doit refléter la définition du `__init__` ;
- **peut être utilisé pour configurer l'objet**, c'est-à-dire initialiser correctement son état interne, créer des variables d'instance, instancier d'autres objets si leur existence est nécessaire, etc.

Regardons un exemple pour mieux comprendre :

```
1 class Classy:  
2     def __init__(self, value):  
3         self.var = value  
4  
5  
6 obj_1 = Classy("object")  
7  
8 print(obj_1.var)           sortie → object
```

Notez que le constructeur :

- **ne peut pas renvoyer une valeur**, car il est conçu pour renvoyer un objet nouvellement créé et rien d'autre ;
- **ne peut pas être appelé directement à partir de l'objet ou de l'intérieur de la classe** (par contre, vous pouvez appeler un constructeur à partir de n'importe quelle sous-classe de l'objet, mais nous aborderons ce problème dans le chapitre suivant.)

Comme déjà dit le constructeur est une méthode et une méthode est une fonction. On peut donc faire tout ce qu'on peut faire avec une fonction avec un constructeur.

```
1 class Classy:  
2     def __init__(self, value = None):  
3         self.var = value  
4  
5  
6 obj_1 = Classy("object")  
7 obj_2 = Classy()  
8  
9 print(obj_1.var)           object  
.0 print(obj_2.var)          Sortie → None
```

Tout ce que nous avons dit à propos de **la manipulation** des noms des attributs s'applique également aux noms des méthodes et donc une méthode dont le nom commence par `__` est (partiellement) cachée (privée).

```

1 class Classy:
2     def visible(self):
3         print("visible")
4
5     def __hidden(self):
6         print("hidden")
7
8
9 obj = Classy()
10 obj.visible()
11
12 try:
13     obj.__hidden()
14 except:           visible
15     print("failed")      failed
16
17 obj._Classy__hidden()      Sortie → hidden

```

4.4 La vie cachée des classes et des objets

Chaque classe Python et chaque objet Python est pré-équipé d'un ensemble d'attributs utiles qui peuvent être utilisés pour examiner ses capacités.

Vous connaissez déjà l'un d'entre eux, la propriété `__dict__`. Qui est une espèce de dictionnaire de notre objet ou classe.

Observons comment il traite les méthodes dans le code suivant et recherchez toutes les méthodes et attributs définis. Localisez le contexte dans lequel ils existent : à l'intérieur de l'objet ou à l'intérieur de la classe.

```

1 class Classy:
2     varia = 1
3     def __init__(self):
4         self.var = 2
5
6     def method(self):
7         pass
8
9     def __hidden(self):
10        pass
11
12
13 obj = Classy()
14
15 print(obj.__dict__)
16 print(Classy.__dict__)

```

4.4.1 Name

Une autre propriété intégrée qui mérite d'être mentionnée est `__name__`, qui est une chaîne de caractère. Cet attribut contient **le nom de la classe**. Ce n'est rien d'excitant, juste une chaîne de caractère (string).

Remarque : l'attribut `__name__` est absent de l'objet - **il n'existe qu'à l'intérieur des classes**.

Si vous voulez trouver **la classe d'un objet particulier**, vous pouvez utiliser une fonction nommée `type()`, qui est capable (entre autres) de trouver une classe qui a été utilisée pour instancier n'importe quel objet.

```
1 class Classy:  
2     pass  
3  
4  
5 print(Classy.__name__)  
6 obj = Classy()           Classy  
7 print(type(obj).__name__) Sortie → Classy
```

Attention qu'une déclaration comme celle-ci, provoquerait une erreur :

```
print(obj.__name__)
```

4.4.2 Module

`__module__` est aussi une chaîne de caractères , elle **stocke le nom du module qui contient la définition de la classe**.

```
1 class Classy:  
2     pass  
3  
4  
5 print(Classy.__module__)      _____  
6 obj = Classy()                __main__  
7 print(obj.__module__)        Sortie → __main__
```

Comme vous le savez, tout module nommé `__main__` n'est en fait pas un module, mais en fait le **fichier en cours d'exécution**.

4.4.3 Bases

`__bases__` est un tuple. Un **tuple qui contient des classes** (pas des noms de classe) qui sont des superclasses directes pour la classe.

L'ordre est le même que celui utilisé dans la définition de classe.
Nous ne vous montrerons qu'un exemple très basique, car nous voulons mettre en évidence un léger **fonctionnement de l'héritage avant les chapitres dédiés**.

De plus, nous allons vous montrer comment utiliser cet attribut en détail lorsque nous discuterons des exceptions.

Remarque : **seules les classes ont cet attribut** - les objets n'en ont pas.
Nous allons définir une fonction nommée `printbases()`, conçue pour présenter clairement le contenu du tuple.

```
1 class SuperOne:  
2     pass  
3  
4  
5 class SuperTwo:  
6     pass  
7  
8  
9 class Sub(SuperOne, SuperTwo):  
0     pass  
1  
2  
3 def printBases(cls):  
4     print('(', end='')  
5  
6     for x in cls.__bases__:  
7         print(x.__name__, end=' ')  
8     print(')')  
9  
0  
1 printBases(SuperOne)          ( object )  
2 printBases(SuperTwo)          ( object )  
3 printBases(Sub)              Sortie → ( SuperOne SuperTwo )
```

Remarque : une classe **sans superclasses explicites pointe vers object** (une classe Python prédéfinie) comme ancêtre direct. Dans le diagramme de classe, on pourra donc ne définir aucune superclasse.

4.5 Le python une introspection et une réflexion qui paye

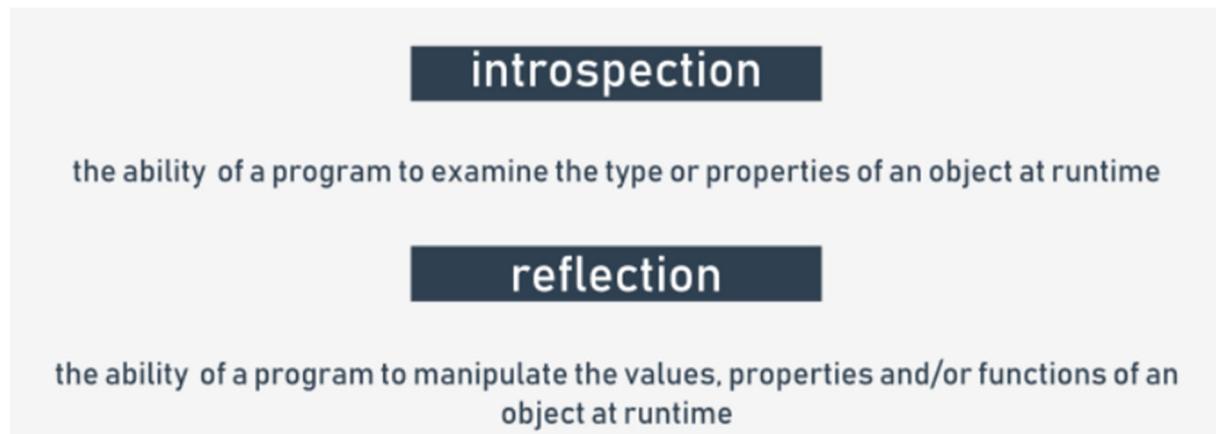


Figure 7 : Schématisation de termes

Tous ces moyens permettent au programmeur Python d'effectuer deux activités importantes spécifiques à de nombreux langages objectifs. Il s'agit des éléments suivants :

- **L'introspection**, qui est la capacité d'un programme à examiner le type ou les propriétés d'un objet au moment de l'exécution;
- **La réflexion**, qui va plus loin, est la capacité d'un programme à manipuler les valeurs, les propriétés et/ou les fonctions d'un objet au moment de l'exécution.

En d'autres termes, vous n'avez pas besoin de connaître une définition complète de classe/objet pour manipuler l'objet, car l'objet et/ou sa classe contiennent les métadonnées vous permettant de reconnaître ses caractéristiques pendant l'exécution du programme.

4.6 Analysons une classe

Que pensez-vous pouvoir trouver au niveau d'une classe python ?

La réponse est assez simple : TOUT CE QUE VOUS VOULEZ !

La réflexion et l'introspection permettent à un programmeur de faire n'importe quoi avec chaque objet, peu importe d'où il vient.

Analysons le code suivant :

```
1  class MyClass:  
2      pass  
3  
4  obj = MyClass()  
5  obj.a = 1  
6  obj.b = 2  
7  obj.i = 3  
8  obj.ireal = 3.5  
9  obj.integer = 4  
10 obj.z = 5  
11  
12 def incIntsI(obj):  
13     for name in obj.__dict__.keys():  
14         if name.startswith('i'):  
15             val = getattr(obj, name)  
16             if isinstance(val, int):  
17                 setattr(obj, name, val + 1)  
18  
19  
20 print(obj.__dict__)  
21 incIntsI(obj)  
22 print(obj.__dict__)
```

La fonction nommée `incIntsI()` obtient un objet de n'importe quelle classe, analyse son contenu afin de trouver tous les attributs entiers dont le nom commence par i et les incrémente d'un.

Impossible? Pas du tout!

Voici comment cela fonctionne :

- Ligne 1 : Définir une classe très simple...
- Lignes 3 à 10: ... et remplissez-le avec quelques attributs;
- Ligne 12 : C'est notre fonction !
- ligne 13 : scannez l'attribut `__dict__`, à la recherche de tous les noms d'attributs ;
- Ligne 14 : Si un nom commence par i...
- Ligne 15: ... Utilisez la fonction `getattr()` pour obtenir sa valeur actuelle ;
Remarque : `GetAttr()` prend deux arguments : un objet et son nom de propriété (sous forme de chaîne), et renvoie la valeur de l'attribut actuel ;
- ligne 16: Vérifiez si la valeur est de type integer, et utilisez la fonction `isinstance()` à cette fin;
- Ligne 17 : si la vérification se passe bien, incrémentez la valeur de l'attribut en utilisant la fonction `setattr()` ; la fonction prend trois arguments : un objet, le nom de la propriété (sous forme de chaîne) et la nouvelle valeur de l'attribut.

Le code produit :

```
{'a': 1, 'entier': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}  
{'a': 1, 'entier': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

4.7 Principaux points à retenir

1. Une méthode est une fonction intégrée dans une classe. Le premier (ou le seul) paramètre de chaque méthode est généralement nommé `self`, qui est conçu pour identifier l'objet pour lequel la méthode est appelée afin d'accéder aux propriétés de l'objet ou d'appeler ses méthodes.
2. Si une classe contient un **constructeur** (une méthode nommée `__init__`), elle ne peut renvoyer aucune valeur et ne peut pas être invoquée directement.
3. Toutes les classes (mais pas les objets) contiennent une propriété nommée `__name__`, qui stocke le nom de la classe. En outre, une propriété nommée `__module__` stocke le nom du module dans lequel la classe a été déclarée, tandis que la propriété nommée `__bases__` est un tuple contenant les superclasses d'une classe.

```
1 class Sample:  
2     def __init__(self):  
3         self.name = Sample.__name__  
4     def myself(self):  
5         print("My name is " + self.name + " living in a " + Sample.__module__)  
6  
7  
8 obj = Sample()  
9 obj.myself()
```

Sortie : My name is Sample living in a __main__

5 L'héritage, Pourquoi et comment ?

Avant de commencer à parler d'héritage, nous voulons présenter un nouveau mécanisme pratique utilisé par les classes et les objets de Python - **c'est la façon dont l'objet est capable de se présenter.**

Commençons par un exemple.

```
1 class Star:  
2     def __init__(self, name, galaxy):  
3         self.name = name  
4         self.galaxy = galaxy  
5  
6  
7 sun = Star("EtoileNoir", "FarFarAway")  
8 print(sun)
```

Le programme n'imprime qu'une seule ligne de texte, qui dans notre cas est la suivante:

```
<__main__.Objet étoile à 0x7f1074cc7c50>
```

Si vous exécutez le même code sur votre ordinateur, vous verrez quelque chose de très similaire, bien que le nombre hexadécimal (la sous-chaine commençant par 0x) soit différente, car il s'agit simplement d'un identifiant d'objet interne utilisé par Python, et il est peu probable qu'il apparaisse identique lorsque le même code est exécuté dans un environnement différent.

Comme vous pouvez le voir, l'impression à l'écran ici n'est pas vraiment utile, et quelque chose de plus spécifique, ou tout simplement plus joli, peut être préférable.

Heureusement, Python offre une telle fonction.

Lorsque Python a besoin qu'une classe/un objet soit présenté sous forme de chaîne de caractères (placer un objet comme argument dans l'appel de la fonction `print()` remplit cette condition), il essaie d'appeler une méthode nommée `__str__()` à partir de l'objet et d'utiliser la chaîne de caractères qu'il renvoie.

La méthode `__str__()` par défaut renvoie la chaîne ci-dessus – qu'on peut avouer est un peu moche et peu informative. Vous pouvez le changer simplement en **définissant votre propre méthode du même nom.**

Tada !

```
1 class Star:  
2     def __init__(self, name, galaxy):  
3         self.name = name  
4         self.galaxy = galaxy  
5  
6     def __str__(self):  
7         return self.name + ' in ' + self.galaxy  
8  
9  
0 sun = Star("EtoileNoir", "FarFarAway")  
1 print(sun)
```

Cette nouvelle méthode `str()` crée une chaîne composée des noms de l'étoile et de la galaxie - rien de spécial, mais les résultats d'impression sembleront meilleurs maintenant, n'est-ce pas?

Pouvez-vous deviner le résultat? Exécutez le code pour vérifier si vous aviez raison.

5.1 Contextualisation

Le terme héritage est bien plus ancien que la programmation informatique, et il décrit la pratique courante de transmettre différents biens d'une personne à une autre à la mort de celle-ci. Le terme, lorsqu'il est lié à la programmation informatique, a une signification **entièrement différente à celle-ci !!**

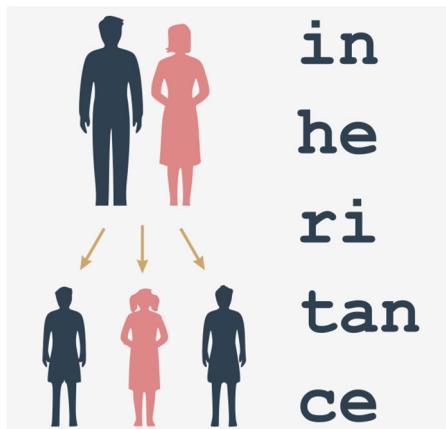


Figure 8 : L'héritage traditionnel

Définissons le terme pour nos besoins:

L'héritage est une pratique courante (en POO) consistant à **transmettre des attributs et des méthodes de la superclasse (définie et existante) à une classe nouvellement créée, appelée sous-classe**.

En d'autres termes, **l'héritage est un moyen de construire une nouvelle classe, non pas à partir de zéro, mais en utilisant un répertoire de traits déjà défini**. La nouvelle classe hérite (et c'est la clé) de toutes les caractéristiques déjà existantes, mais est capable d'en ajouter de nouvelles si nécessaire.

Grâce à cela, il est possible de **construire des classes plus spécialisées (plus concrètes)** en utilisant des ensembles de règles générales et de comportements prédéfinis.

Le facteur le plus important du processus est la relation entre la superclasse et toutes ses sous-classes (note: si `B` est une sous-classe de `A` et `C` est une sous-classe de `B`, cela signifie également que `C` est une sous-classe de `A`, car la relation est entièrement transitive).

Un exemple très simple **d'héritage à deux niveaux** est présenté ici :

```
class Vehicle:  
    pass  
  
class LandVehicle(Vehicle):  
    pass  
  
class TrackedVehicle(LandVehicle):  
    pass
```

Toutes les classes présentées sont vides pour l'instant, car nous allons vous montrer comment fonctionnent les relations mutuelles entre les super- et sous-classes. Nous les remplirons de contenu dans la suite du cours.

On peut dire que :

- La classe Vehicle est la superclasse pour les classes LandVehicle et TrackedVehicle;
- La classe LandVehicle est à la fois une sous-classe de Vehicle et une superclasse de TrackedVehicle ;
- La classe TrackedVehicle est une sous-classe des classes Vehicle et LandVehicle.

La connaissance que nous pouvons avoir de cette hiérarchisation provient de la lecture du code (en d'autres termes, nous le savons parce que nous pouvons le voir).

Python sait-il faire la même chose ? Est-il possible de demander à Python à connais-tu la sous-classe de ? **Oui**.

5.2 issubclass()

Python offre une fonction capable **d'identifier une relation entre deux classes**, et bien que son diagnostic ne soit pas complexe, il peut **vérifier si une classe particulière est une sous-classe d'une autre classe**.

Voici à quoi cela ressemble: `issubclass(ClassOne, ClassTwo)`

La fonction renvoie True si `ClassOne` est une sous-classe de `ClassTwo` et False dans le cas contraire.

```

1  class Vehicle:
2      pass
3
4
5  class LandVehicle(Vehicle):
6      pass
7
8
9  class TrackedVehicle(LandVehicle):
10     pass
11
12
13 for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
14     for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
15         print(issubclass(cls1, cls2), end="\t")
16     print()

```

Regardons le code ci-dessus :

Il y a deux boucles imbriquées qui ont pour but est **de vérifier toutes les paires de classes ordonnées possibles et d'imprimer les résultats de la vérification pour déterminer si la paire correspond à la relation sous-classe-superclasse.**

Exécutez le code. Le programme produit :

Vrai	Faux	Faux
Vrai	Vrai	Faux
Vrai	Vrai	Vrai

Rendons le résultat plus lisible :

↓ est une sous-classe de → Véhicule	Véhicule	Véhicule terrestre	Véhicule chenillé
Véhicule	Vrai	Faux	Faux
Véhicule terrestre	Vrai	Vrai	Faux
Véhicule chenillé	Vrai	Vrai	Vrai

Il y a une observation importante à faire : **chaque classe est considérée comme une sous-classe d'elle-même.**

5.3 isinstance()

Comme vous le savez déjà, **un objet est une incarnation d'une classe**. Cela signifie que l'objet est comme un gâteau cuit à l'aide d'une recette qui est incluse dans la classe. (Miam Miam)

Attention que cela peut générer des problèmes importants.

Supposons que vous ayez un gâteau (par exemple, en tant qu'argument passé à votre fonction). Vous voulez savoir quelle recette a été utilisée pour le faire.

Pourquoi? Parce que vous voulez savoir à quoi vous attendre, par exemple, s'il contient des noix ou non, ce qui est une information cruciale pour certaines personnes.

De même, cela peut être crucial si l'objet a (ou n'a pas) certaines caractéristiques. En d'autres termes, **s'il s'agit d'un objet d'une certaine classe ou non.**

Un tel fait pourrait être détecté par la fonction nommée `isinstance()` :

```
isinstance(objectName, ClassName)
```

La fonction renvoie la valeur `True` si l'objet est une instance de la classe, ou la valeur `False` dans le cas contraire.

Être une instance d'une classe signifie que l'objet (le gâteau) a été préparé en utilisant une recette contenue dans la classe ou l'une de ses superclasses.

N'oubliez pas: si une sous-classe contient au moins les mêmes caractéristiques que l'une de ses superclasses, cela signifie que les objets de la sous-classe peuvent faire la même chose que les objets dérivés de la superclasse. Cela signifie que c'est une instance de sa classe d'origine et de l'une de ses superclasses.

Prenons un exemple :

```
class Vehicle:  
    pass  
  
class LandVehicle(Vehicle):  
    pass  
  
class TrackedVehicle(LandVehicle):  
    pass  
  
my_vehicle = Vehicle()  
my_land_vehicle = LandVehicle()  
my_tracked_vehicle = TrackedVehicle()  
  
for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:  
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:  
        print(isinstance(obj, cls), end="\t")  
    print()
```

Nous avons créé trois objets, un pour chacune des classes. Ensuite, à l'aide de deux boucles imbriquées, nous vérifions toutes les paires objet-classe possibles **pour savoir si les objets sont des instances des classes.**

Voici ce que nous obtenons :

Vrai	Faux	Faux
Vrai	Vrai	Faux
Vrai	Vrai	Vrai

Rendons le résultat plus lisible une fois de plus:

↓ is an instance of →	Vehicle	LandVehicle	TrackedVehicle
my_vehicle	True	False	False
my_land_vehicle	True	True	False
my_tracked_vehicle	True	True	True

Le tableau confirme-t-il nos attentes?

5.4 L'opérateur is

Il existe également un opérateur Python qui mérite d'être mentionné, car il se réfère directement aux objets : `object_one is object_two`

L'opérateur `is` vérifie si deux variables (`object_one` et `object_two` ici) font référence au même objet.

N'oubliez pas que les **variables ne stockent pas les objets eux-mêmes, mais seulement des adresses pointant vers la mémoire interne de Python.**

L'affectation d'une valeur d'une variable objet à une autre variable ne copie pas l'objet, mais uniquement son descripteur. C'est pourquoi un opérateur comme celui-ci peut être très utile dans des circonstances particulières.

```
class SampleClass:  
    def __init__(self, val):  
        self.val = val  
  
object_1 = SampleClass(0)  
object_2 = SampleClass(2)  
object_3 = object_1  
object_3.val += 1  
  
print(object_1 is object_2)  
print(object_2 is object_3)  
print(object_3 is object_1)  
print(object_1.val, object_2.val, object_3.val)  
  
string_1 = "Mary had a little "  
string_2 = "Mary had a little lamb"  
string_1 += "lamb"  
  
print(string_1 == string_2, string_1 is string_2)
```

Analysons ce code :

- Il existe une classe très simple équipée d'un constructeur simple, créant une seule propriété. La classe est utilisée pour instancier deux objets. Le premier objet est ensuite affecté à une autre variable, et son attribut `val` est incrémentée de une unité.
- Ensuite, l'opérateur `is` est appliqué trois fois pour vérifier toutes les paires d'objets possibles, et toutes les valeurs de `val` sont également imprimées.
- La dernière partie du code effectue un autre test. Après trois affectations, les deux chaînes contiennent les mêmes textes, mais **ces textes sont stockés dans des objets différents**.

La sortie sera :

```
False
False
True
1 2 1
True False
```

Les résultats prouvent que `object_1` et `object_3` sont en fait les mêmes objets, alors que `string_1` et `string_2` ne le sont pas, bien que leur contenu soit le même.

5.5 Comment python utilise les méthodes et les attributs hérités

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

    def __str__(self):
        return "Je m'appelle " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)

obj = Sub("Dark Vador")
print(obj)
```

Analysons ce code :

- il existe une classe nommée `Super`, qui définit son propre constructeur utilisé pour affecter l'attribut de l'objet, nommée `name`.
- La classe définit également la méthode `__str__()`, ce qui permet à la classe de présenter son identité sous forme de texte clair.

- la classe est ensuite utilisée comme base pour créer une sous-classe nommée `Sub`. La sous-classe définit son propre constructeur, qui appelle celui de la superclasse. Notez comment nous l'avons fait: `Super.__init__(self, name)`.
- nous avons explicitement nommé la superclasse et indiqué la méthode pour invoquer `__init__()`, en fournissant tous les arguments nécessaires.
- nous avons instancié un objet de la classe `Sub` et l'avons imprimé.

Le code produit :

`Je m'appelle Dark Vador.`

→ Comme il n'y a pas de méthode `__str__()` dans la sous-classe, la chaîne imprimée doit être produite dans la super classe. Cela signifie que la méthode `__str__()` a été héritée par la sous-classe.

Modifions légèrement le code précédent

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

    def __str__(self):
        return "Je m'appelle " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        super().__init__(name)

obj = Sub("Dark Vador")

print(obj)
```

Nous l'avons modifié pour vous montrer une autre méthode d'accès à toute entité définie dans la superclasse.

Dans le dernier exemple, nous avons explicitement nommé la superclasse. Dans cet exemple, nous utilisons la fonction `super()`, qui **accède à la superclasse sans avoir besoin de connaître son nom** :

`super().__init__(nom)`

La fonction `super()` crée un contexte dans lequel vous n'avez pas à passer l'argument `self` à la méthode invoquée - c'est pourquoi il est possible d'activer le constructeur de superclasse en utilisant un seul argument.

Remarque : vous pouvez utiliser ce mécanisme non seulement pour **appeler le constructeur de superclasse, mais également pour accéder à toutes les ressources disponibles dans la superclasse**.

5.6 Accès aux variables de classes

Essayons de faire quelque chose de similaire, mais avec des attributs (plus précisément : avec **des variables de classe**).

Jetez un coup d'œil à l'exemple ci-dessous :

```
# Testing properties: class variables.  
class Super:  
    supVar = 1  
  
class Sub(Super):  
    subVar = 2  
  
obj = Sub()  
  
print(obj.subVar)  
print(obj.supVar)
```

Comme vous pouvez le voir, la classe `Super` définit une variable de classe nommée `supVar` et la classe `Sub` définit une variable nommée `subVar`. Ces deux variables sont visibles à l'intérieur de l'objet de la classe `Sub`, c'est pourquoi le code produit :

```
2  
1
```

5.7 Accès aux variables d'instances

Le même effet peut être observé avec les **variables d'instance**

```
# Testing properties: instance variables.  
class Super:  
    def __init__(self):  
        self.supVar = 11  
  
class Sub(Super):  
    def __init__(self):  
        super().__init__()  
        self.subVar = 12  
  
obj = Sub()  
  
print(obj.subVar)  
print(obj.supVar)
```

Le constructeur de classe `Sub` crée une variable d'occurrence nommée `subVar`, tandis que le constructeur `Super` fait de même avec une variable nommée `supVar`. Comme précédemment, les deux variables sont accessibles depuis l'objet de la classe `Sub`.

Les extrants du programme sont les suivants :

```
12  
11
```

Remarque : l'existence de la variable `supVar` est évidemment conditionnée par l'appel du constructeur de classe Super. L'omettre entraînerait l'absence de la variable dans l'objet créé (essayez-le vous-même).

5.7 Résumé de l'héritage jusqu'ici

Il est maintenant possible de formuler une déclaration générale décrivant le comportement de Python pour l'héritage.

Lorsque vous essayez d'accéder à l'entité d'un objet, Python essaie (dans cet ordre) :

- de le trouver **à l'intérieur de l'objet** lui-même;
- de le trouver dans **toutes les classes** impliquées dans la ligne d'héritage de l'objet de bas en haut ;

Si les deux éléments ci-dessus échouent, une **exception (`AttributeError`) est déclenchée**.

La première condition peut nécessiter une attention supplémentaire, pour s'assurer de votre bonne compréhension. Tous les objets dérivés d'une classe particulière peuvent avoir différents ensembles d'attributs, et certains attributs peuvent être ajoutés à l'objet longtemps après la création de l'objet.

L'exemple ci-dessous résume cela dans une **ligne d'héritage à trois niveaux**.

```
class Level1:  
    variable_1 = 100  
    def __init__(self):  
        self.var_1 = 101  
    def fun_1(self):  
        return 102  
  
class Level2(Level1):  
    variable_2 = 200  
    def __init__(self):  
        super().__init__()  
        self.var_2 = 201  
    def fun_2(self):  
        return 202  
  
class Level3(Level2):  
    variable_3 = 300  
    def __init__(self):  
        super().__init__()  
        self.var_3 = 301  
    def fun_3(self):  
        return 302  
  
obj = Level3()  
print(obj.variable_1, obj.var_1, obj.fun_1())  
print(obj.variable_2, obj.var_2, obj.fun_2())  
print(obj.variable_3, obj.var_3, obj.fun_3())
```

Tous les chapitres que nous avons faits jusqu'à présent sont liés à **l'héritage unique**, lorsqu'une sous-classe a exactement une superclasse. C'est la situation la plus courante (et la plus recommandée aussi mais on sait bien que fauter, vous aimez ça).

Python, cependant, offre la possibilité **d'héritage multiple**.

5.8 L'héritage multiple

L'héritage multiple se produit lorsqu'une classe a plus d'une superclasse.

Syntaxiquement, un tel héritage est présenté sous la forme d'une liste de superclasses séparées par des virgules placées entre parenthèses après le nouveau nom de classe - tout comme ici:

```
class SuperA:  
    var_a = 10  
    def fun_a(self):  
        return 11  
  
class SuperB:  
    var_b = 20  
    def fun_b(self):  
        return 21  
  
class Sub(SuperA, SuperB):  
    pass  
  
obj = Sub()  
  
print(obj.var_a, obj.fun_a())  
print(obj.var_b, obj.fun_b())
```

La **sous-classe** a deux superclasses: **SuperA** et **SuperB**. Cela signifie que la **sous-classe hérite de toutes les caractéristiques offertes par SuperA et SuperB**.

La sortie est :

10	11
20	21

Il est maintenant temps d'introduire un tout nouveau terme : **prépondérant**, en français ça ne veut rien dire, nous parlerons donc d'**overriding**

Que pensez-vous qu'il se passera si plus d'une des superclasses définit une entité d'un nom similaire ?

Pour Mieux se rendre donc de ce que cela implique, partons d'un exemple :

```
class Level1:  
    var = 100  
    def fun(self):  
        return 101  
  
class Level2(Level1):  
    var = 200  
    def fun(self):  
        return 201  
  
class Level3(Level2):  
    pass  
  
obj = Level3()  
  
print(obj.var, obj.fun())
```

Les classes `Level1` et `Level2` définissent une méthode nommée `fun()` et une propriété nommée `var`. Cela signifie-t-il que l'objet `de classe Level3` pourra accéder à deux copies de chaque entité ? **Pas du tout. NEVER**

L'entité définie ultérieurement (au sens de l'héritage) remplace la même entité définie précédemment. C'est pourquoi le code produit la sortie suivante :
`200 201`

Comme vous pouvez le voir, la variable de classe `var` et la méthode `fun()` de la classe `Level2` remplacent les entités des mêmes noms dérivées de la classe `Level1`.

Cette fonctionnalité peut être utilisée intentionnellement pour modifier les comportements de classe par défaut (ou précédemment définis) lorsque l'une de ses classes doit agir d'une manière différente de celle de son ancêtre.

Nous pouvons également dire que **Python recherche une entité du bas vers le haut** et est pleinement satisfait de la première entité du nom recherché.

Comment cela fonctionne-t-il lorsqu'une classe a deux ancêtres offrant la même entité et qu'ils se situent au même niveau? En d'autres termes, à quoi devez-vous vous attendre lorsqu'une classe émerge en utilisant l'héritage multiple? Une idée ??

Encore une fois, essayons d'expliquer à partir d'un exemple :

```

class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"

class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass

obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())

```

La sous-classe hérite de produits de deux superclasses, `Left` et `Right` (ces noms sont censés être significatifs).

Il ne fait aucun doute que la variable de classe `var_right` provient de la classe `Right` et `var_left` vient respectivement de `Left`.

C'est clair. Mais d'où vient `var`? Est-il possible de le deviner? Le même problème est rencontré avec la méthode `fun()` - sera-t-elle invoquée de `Left` ou de `Right`?

Exécutons le programme pour analyser la sortie et confronter les hypothèses:

`L LL RR Gauche`

Cela prouve que les deux cas qui sont peu clairs ont bien une solution à l'intérieur de la classe `Left`.

Est-ce une prémissse suffisante pour formuler une règle générale? Oui.

Nous pouvons dire que **Python recherche les composants** d'objet dans l'ordre suivant:

- **à l'intérieur de l'objet** lui-même;
- **dans ses superclasses**, de bas en haut;

s'il y a plus d'une classe sur un chemin d'héritage particulier, Python les analyse de gauche à droite.

Vous avez tout compris? Pour en être sûr, il suffit d'apporter une petite modification dans le code → remplacer: `class Sub(Left, Right):` par `class Sub(Right, Left):`, puis exécutez à nouveau le programme et voyez ce qui se passe.

Nous aurons :

`R LL RR Droit`

5.9 Comment construire une hiérarchie de classes

Construire une hiérarchie de classes n'est pas seulement de l'art pour l'art. Il y a réellement une raison à cela.

Si vous divisez un problème entre les classes et décidez laquelle d'entre elles doit être située en haut et laquelle doit être placée en bas de la hiérarchie, vous devez analyser attentivement la question de base, mais avant de vous montrer comment le faire (et comment ne pas le faire), nous voulons mettre en évidence un effet intéressant. Ce n'est rien d'extraordinaire (c'est juste une conséquence des règles générales présentées précédemment), mais s'en souvenir peut être essentiel pour comprendre comment certains codes fonctionnent et comment l'effet peut être utilisé pour construire un ensemble flexible de classes.

Comme à notre habitude, rien de tel qu'un exemple pour vous expliquer :

```
class One:
    def do_it(self):
        print("do_it from One")

    def doanything(self):
        self.do_it()

class Two(One):
    def do_it(self):
        print("do_it from Two")

one = One()
two = Two()

one.doanything()
two.doanything()
```

Il y a deux classes, nommées `One` et `Two`, tandis que `Two` est dérivée de `One`. Rien de spécial. Cependant, une chose semble suspect → la méthode `do_it()`.

La méthode `do_it()` est **définie deux fois**: à l'origine à l'intérieur de `One`, puis à l'intérieur de `Two`. Le but de l'exemple réside dans le fait qu'il n'est **invoqué qu'une seule fois** à l'intérieur de `One`.

La question du jour est de savoir laquelle des deux méthodes sera invoquée par les deux dernières lignes du code.

La première invocation semble être simple, et elle l'est, en fait, invoquer `doanything()` à partir de l'objet nommé `One` activera évidemment la première des méthodes.

La deuxième invocation a besoin de plus d'attention. C'est tout aussi simple si vous gardez à l'esprit comment Python trouve les composants de classe. Le deuxième appel lancera `do_it()` sous la forme existante dans la classe `Two`, indépendamment du fait que l'appel a lieu dans la classe `One`.

En effet, le code produit la sortie suivante :

```
do_it from One
do_it from Two
```

Remarque : la situation dans laquelle **la sous-classe est capable de modifier son comportement de superclasse (comme dans l'exemple) est appelée polymorphisme.**

Le mot vient du grec (polys: « beaucoup» et morphé, « forme»), ce qui signifie qu'une seule et même classe peut prendre différentes formes en fonction des redéfinitions effectuées par l'une de ses sous-classes.

La méthode, redéfinie dans l'une des superclasses, modifiant ainsi le comportement de la superclasse, est appelée **virtuelle**.

En d'autres termes, aucune classe n'est donnée une fois pour toute. Le comportement de chaque classe peut être modifié à tout moment par l'une de ses sous-classes.

Nous allons vous montrer **comment utiliser le polymorphisme pour étendre la flexibilité de classe**.

5.10 Le polymorphisme

```
import time

class TrackedVehicle:
    def control_track(left, stop):
        pass

    def turn(left):
        control_track(left, True)
        time.sleep(0.25)
        control_track(left, False)

class WheeledVehicle:
    def turn_front_wheels(left, on):
        pass

    def turn(left):
        turn_front_wheels(left, True)
        time.sleep(0.25)
        turn_front_wheels(left, False)
```

A quoi vous fait penser ce code ?

Il fait référence à l'exemple montré au début du module lorsque nous avons parlé des concepts généraux de la POO.

Cela peut vous sembler bizarre, mais nous n'avons pas utilisé l'héritage de quelque manière que ce soit, simplement pour vous montrer que cela ne nous limite pas.

Nous avons défini deux classes distinctes capables de produire deux types différents de véhicules terrestres. La principale différence entre eux réside dans la façon dont ils vont avancer (les roues). Un véhicule à roues ne fait que tourner les roues avant (généralement). Un véhicule à chenilles doit arrêter l'une de ses chenilles.

Pouvez-vous suivre le code?

- Un véhicule chenillé effectue un virage en s'arrêtant et en se déplaçant sur l'une de ses chenilles (ceci est fait par la méthode `control_track()`, qui sera mise en œuvre plus tard)
- Un véhicule à roues tourne lorsque ses roues avant tournent (ceci est fait par la méthode `turn_front_wheels()`)
- La méthode `turn()` utilise la méthode adaptée à chaque véhicule particulier.

Pouvez-vous voir **ce qui ne va pas dans ce le code?**

Les méthodes `turn()` semblent trop similaires pour les laisser sous cette forme.

Reconstruisons le code - nous allons introduire une superclasse pour rassembler tous les aspects similaires de la conduite des véhicules, en déplaçant tous les détails vers les sous-classes.

```
import time

class Vehicle:
    def change_direction(left, on):
        pass

    def turn(left):
        change_direction(left, True)
        time.sleep(0.25)
        change_direction(left, False)

class TrackedVehicle(Vehicle):
    def control_track(left, stop):
        pass

    def change_direction(left, on):
        control_track(left, on)

class WheeledVehicle(Vehicle):
    def turn_front_wheels(left, on):
        pass

    def change_direction(left, on):
        turn_front_wheels(left, on)
```

Voici ce que nous avons fait :

- nous avons défini une superclasse nommée `Vehicle`, qui utilise la méthode `turn()` pour implémenter un schéma général de virage, tandis que le virage lui-même est effectué par une méthode nommée `change_direction()`;
→ l'ancienne méthode est vide, car nous allons mettre tous les détails dans la sous-classe (une telle méthode est souvent appelée méthode **abstraite**)
- nous avons défini une sous-classe nommée `TrackedVehicle` (qui est dérivée de la classe `Vehicle`) qui instancie la méthode `change_direction()` en utilisant la méthode spécifique (concrète) nommée `control_track()`
- respectivement, la sous-classe nommée `WheeledVehicle` fait la même chose, mais utilise la méthode `turn_front_wheels()` pour forcer le véhicule à tourner.

L'avantage le plus important (en omettant les problèmes de lisibilité) est que cette forme de code vous permet d'implémenter un tout nouvel algorithme de virage simplement en modifiant la méthode `turn()`, ce qui peut être fait en un seul endroit, car tous les véhicules y obéiront.

C'est ainsi que **le polymorphisme aide le développeur à garder le code propre et cohérent**. Il est donc capital de bien le comprendre et de bien le mettre en place.

5.11 La composition

L'héritage n'est pas le seul moyen de construire des classes adaptables. Vous pouvez atteindre les mêmes objectifs (pas toujours, mais très souvent) en utilisant une technique appelée la composition.

La composition est le processus de composition d'un objet à l'aide d'autres objets différents. Les objets utilisés dans la composition fournissent un ensemble de propriétés souhaitées (attributs et / ou méthodes) de sorte que nous pouvons dire qu'ils agissent comme des blocs utilisés pour construire une structure plus compliquée.

On peut dire que :

- **L'héritage étend les capacités d'une classe en** ajoutant de nouveaux composants et en modifiant ceux qui existent déjà; en d'autres termes, la recette complète est contenue dans la classe elle-même et tous ses ancêtres; l'objet prend tous les biens de la classe et les utilise;
- **La composition projette une classe comme un conteneur** capable de stocker et d'utiliser d'autres objets (dérivés d'autres classes) où chacun des objets implémente une partie du comportement d'une classe souhaitée.

Illustrons la différence en utilisant les véhicules définis précédemment. L'approche précédente nous a conduits à une hiérarchie de classes dans laquelle la classe supérieure connaissait les règles générales utilisées pour tourner le véhicule, mais ne savait pas comment contrôler les composants appropriés (roues ou chenilles).

Les sous-classes ont mis en œuvre cette capacité en introduisant des mécanismes spécialisés. Faisons (presque) la même chose, mais en utilisant la composition. La classe mère (comme dans l'exemple précédent) sait comment tourner le véhicule, mais le virage réel est effectué par un objet spécialisé stocké dans un attribut nommé `controller`. Il est capable de contrôler le véhicule en manipulant les pièces du véhicule concerné.

Jetons un coup d'œil à un bout de code :

```

1 import time
2
3 class Tracks:
4     def change_direction(self, left, on):
5         print("tracks: ", left, on)
6
7
8 class Wheels:
9     def change_direction(self, left, on):
10        print("wheels: ", left, on)
11
12
13 class Vehicle:
14     def __init__(self, controller):
15         self.controller = controller
16
17     def turn(self, left):
18         self.controller.change_direction(left, True)
19         time.sleep(0.25)
20         self.controller.change_direction(left, False)
21
22
23 wheeled = Vehicle(Wheels())
24 tracked = Vehicle(Tracks())
25
26 wheeled.turn(True)
27 tracked.turn(False)

```

Il existe deux classes nommées `Tracks` et `Wheels` → elles savent comment contrôler la direction du véhicule. Il existe également une classe nommée `Vehicle` qui peut utiliser n'importe lequel des contrôleurs disponibles (les deux déjà définis, ou tout autre défini à l'avenir).

Le contrôleur lui-même est transmis à la classe lors de l'initialisation.

De cette façon, la capacité du véhicule à tourner est composée à l'aide d'un objet externe, non implémenté à l'intérieur de la classe `Vehicle`.

En d'autres termes, nous avons un véhicule universel et pouvons y installer des chenilles ou des roues. Vive la technologie, vive l'hybride !

Le code produit la sortie suivante :

```
wheels : True True
wheels : Vrai faux
tracks: False True
tracks: False False
```

5.12 L'héritage simple vs l'héritage multiple

Comme vous le savez déjà, il n'y a aucun obstacle à l'utilisation de l'héritage multiple en Python. Vous pouvez dériver n'importe quelle nouvelle classe de plusieurs classes précédemment définies.

Il n'y a qu'un seul « mais ». Le fait que vous puissiez le faire ne signifie pas que vous devez le faire. Un peu comme je peux rouler vite si je le souhaite mais... non !

N'oubliez pas que :

- Une seule classe d'héritage est toujours plus simple, plus sûre et plus facile à comprendre et à maintenir;
- L'héritage multiple est toujours risqué, car vous avez beaucoup plus d'occasions de faire une erreur en identifiant ces parties des superclasses qui influenceront efficacement la nouvelle classe;
- L'héritage multiple peut rendre le remplacement extrêmement délicat; De plus, l'utilisation de la fonction `super()` devient ambiguë ;

L'héritage multiple viole le **principe de responsabilité unique** car il crée une nouvelle classe de deux (ou plus) classes qui ne savent rien l'une de l'autre;

Nous ne pouvons que vous mettre en garde pour votre futur quant à l'utilisation de l'héritage multiple et de ne l'utiliser que comme la dernière de toutes les solutions possibles (si vous avez vraiment besoin des nombreuses fonctionnalités différentes offertes par différentes classes, la composition peut être une meilleure alternative). N'oubliez pas qu'un cours est fait pour voir les possibilités existantes mais aussi pour tenter de vous amenez vers de bonnes pratiques.

5.13 Principaux points à retenir

- Une méthode nommée `__str__()` est responsable de **la conversion du contenu d'un objet en une chaîne (plus ou moins) lisible**. Vous pouvez le redéfinir si vous voulez que votre objet puisse se présenter sous une forme plus élégante. Par exemple:

```
1 class Mouse:  
2     def __init__(self, name):  
3         self.my_name = name  
4  
5     def __str__(self):  
6         return self.my_name  
7  
8  
9  
10 the_mouse = Mouse('mickey')  
11 print(the_mouse) # Prints "mickey".
```

- Une fonction nommée `issubclass(Class_1, Class_2)` est capable de déterminer si `Class_1` est une **sous-classe** de `Class_2`. Par exemple:

```
1 class Mouse:  
2     pass  
3  
4  
5 class LabMouse(Mouse):  
6     pass  
7  
8  
9 print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse))
```

- Une fonction nommée `isinstance(Object, Class)` vérifie si un objet **provient d'une classe indiquée**. Par exemple:

```
1 class Mouse:  
2     pass  
3  
4  
5 class LabMouse(Mouse):  
6     pass  
7  
8  
9 mickey = Mouse()  
10 print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse))
```

- Un opérateur appelé `vérifie` si deux variables se réfèrent au **même objet**. Par exemple:

```

1 class Mouse:
2     pass
3
4
5 mickey = Mouse()
6 minnie = Mouse()
7 cloned_mickey = mickey
8 print(mickey is minnie, mickey is cloned_mickey)

```

- Une fonction sans paramètre nommée `super()` renvoie une **référence à la superclasse la plus proche de la classe**. Par exemple:

```

1 class Mouse:
2     def __str__(self):
3         return "Mouse"
4
5
6 class LabMouse(Mouse):
7     def __str__(self):
8         return "Laboratory " + super().__str__()
9
10
11 doctor_mouse = LabMouse();
12 print(doctor_mouse) # Prints "Laboratory Mouse".

```

- Les méthodes ainsi que les variables d'instance et de classe définies dans une superclasse sont **automatiquement héritées** par leurs sous-classes. Par exemple:

```

1 class Mouse:
2     Population = 0
3     def __init__(self, name):
4         Mouse.Population += 1
5         self.name = name
6
7     def __str__(self):
8         return "Hi, my name is " + self.name
9
10 class LabMouse(Mouse):
11     pass
12
13 professor_mouse = LabMouse("Professor Mouser")
14 print(professor_mouse, Mouse.Population) # Prints

```

- Afin de trouver n'importe quelle propriété d'objet / classe, Python la recherche à l'intérieur:
 - l'objet lui-même;
 - toutes les classes impliquées dans la ligne d'héritage de l'objet de bas en haut ;
 - s'il y a plus d'une classe sur un chemin d'héritage particulier, Python les analyse de gauche à droite ;
 - si les deux éléments ci-dessus échouent, l'exception `AttributeError` est déclenchée.
- Si l'une des sous-classes définit une variable méthode/classe/variable d'instance du même nom que celle existant dans la superclasse, le nouveau nom **remplace** toutes les instances précédentes du nom. Par exemple:

```

1 class Mouse:
2     def __init__(self, name):
3         self.name = name
4
5     def __str__(self):
6         return "My name is " + self.name
7
8 class AncientMouse(Mouse):
9     def __str__(self):
10        return "Meum nomen est " + self.name
11
12 mus = AncientMouse("Caesar") # Prints "Meum nomen est Caesar"
13 print(mus)

```

Programmations avancées - Pratique

Cours Pygame

Bachelier en informatique et systèmes – Finalité Télécommunications et réseaux –Bloc 1.

Chapelle Joakim – joakim.chapelle1@heh.be
Depreter Johan – johan.depreter@heh.be
Desmet Erwin – erwin.desmet@heh.be
Scopel Fabrice – fabrice.scopel@heh.be

TABLE DES MATIERES

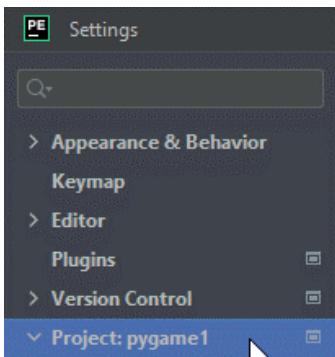
PYGAME.....	3
INSTALLER PYGAME DANS PYCHARM	3
INTRODUCTION - PYGAME	4
QU'EST-CE QUE PYGAME ?.....	4
CODE MINIMAL	5
CONFIGURER UN PROGRAMME PYGAME BASIQUE	6
« GAME LOOP » ET « GAME STATE »	7
LES SURFACES.....	8
LA SURFACE PRINCIPALE	8
LE PRINCIPE DE MODIFICATION D'UNE SURFACE	9
DÉFINIR UNE COULEUR – RGB MODEL.....	10
REmplir LA SURFACE AVEC UNE COULEUR UNIE.....	11
METTRE A JOUR L'AFFICHAGE	11
COORDONNÉES DES PIXELS.....	12
« DRAWING PRIMITIVES ».....	13
DESSINER UNE LIGNE.....	13
QU'EST-CE QUE L'ANTI-ALIASING.....	14
DESSINER UN RECTANGLE	15
DESSINER UN CERCLE	16
TRACER UN POLYGONE	17
LA CLASSE « RECT »	18
LES ATTRIBUTS DE LA CLASSE « RECT ».....	18
BOUGER UN RECTANGLE.....	19
AGRANDIR UN RECTANGLE	21
LES COLLISIONS.....	22
AFFICHER DES IMAGES	23
CHARGER UNE IMAGE	23
INTÉGRER NOTRE IMAGE	23
AFFICHER DU TEXTE.....	24
CHARGER UNE POLICE DE VOTRE OS	24
CHARGER UNE POLICE A PARTIR D'UN FICHIER	26
MÉTHODES UTILES POUR MANIPULER LES TEXTES	27

LES ÉVÈNEMENTS	27
ÉVÈNEMENT LIÉ AU REDIMENSIONNEMENT DE LA FENÊTRE.....	28
ÉVÈNEMENTS LIÉS AU CLAVIER	30
ÉVÈNEMENTS LIÉS A LA SOURIS	32
LES SPRITES.....	33
DÉFINIR UN JOUEUR BASIQUE	33
LES « SPRITES GROUP »	35
DÉTECTER LES COLLISIONS	36
MESURER LE TEMPS	36
METTRE LE PROCESSUS EN PAUSE	36
AJOUTER UN DÉLAI DANS LE CODE	38
LES FRAMES PER SECONDS.....	38
METTRE A JOUR SON AFFICHAGE VIA UN « USEREVENT »	40
JOUER DU SON	41
JOUER UN FICHIER AUDIO	41
STOPPER UN FICHIER AUDIO + FAEDOUT	42
RÉCUPÉRER ET MODIFIER LE VOLUME SONORE	42
RÉCUPÉRER LA DURÉE DU FICHIER.....	42
GÉRER L'ENSEMBLE DES FICHIER AUDIO.....	42

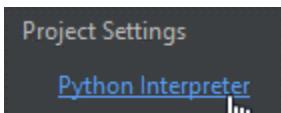
PYGAME

INSTALLER PYGAME DANS PYCHARM

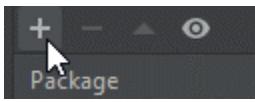
- Ouvrez un nouveau projet
- Allez dans « File > Settings > Project » et sélectionnez votre projet :



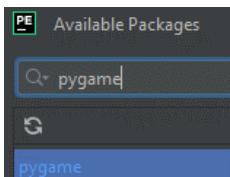
- Aller dans le menu « Python Interpreter » :



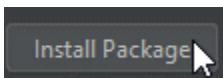
- Appuyez sur l'icône en forme de « + » pour ajouter une librairie à votre projet :



- Cherchez « pygame » dans la barre de recherche :



- Cliquez sur « Install Package » en bas à gauche de l'écran :



QU'EST-CE QUE PYGAME ?

Pygame est une bibliothèque Python basée sur SDL (Simple Direct media Layer) et destinée à la création de jeux vidéo. Elle a été conçue pour être facile à utiliser et permet aux développeurs de créer des jeux rapidement en utilisant les fonctionnalités de base fournies par la bibliothèque.

Elle offre une large gamme de fonctionnalités, telles que :

- **Gestion des graphiques** : Pygame offre un certain nombre de fonctions pour dessiner et afficher des images sur l'écran, y compris des fonctions de dessin de lignes, de formes et de textes. Il peut également afficher des images à partir de fichiers, ce qui est utile pour afficher des sprites (des images qui peuvent être utilisées dans les jeux vidéo pour représenter des objets ou des personnages en mouvement) ou des arrière-plans.
- **Gestion des sons et de la musique** : Lire et jouer des fichiers audio, ce qui est utile pour ajouter de la musique ou des effets sonores à un jeu.
- **Gestion des contrôleurs** : Pygame prend en charge une large gamme de contrôleurs, y compris les claviers, les souris et les manettes de jeu et fournit également des fonctions pour gérer les entrées de ces contrôleurs.
- **Gestion des événements** : Utilisation d'un système d'événements pour gérer les actions de l'utilisateur, telles que les clics de souris et les pressions de touche. Les développeurs peuvent créer des "boucles d'événements" qui surveillent en permanence les entrées de l'utilisateur et qui réagissent en conséquence.

Pygame est très apprécié par les débutants et les personnes souhaitant apprendre à développer des jeux vidéo, car il est relativement facile à utiliser et dispose de nombreux exemples et tutoriels en ligne.

CODE MINIMAL

Afin de vérifier que l'installation de Pygame s'est bien déroulée, nous allons créer notre toute première fenêtre.

Voici le code minimal qui vous servira de base pour l'ensemble de vos applications créées avec Pygame. Rassurez-vous l'ensemble de ce qui est écrit ci-dessous va être expliqué au fur et à mesure de ce chapitre :

```
# Importe et initialise la bibliothèque Pygame
import pygame
pygame.init()

# Configure la surface de dessin
surface = pygame.display.set_mode((300, 200))

# Donne un nom à la fenêtre du programme
pygame.display.set_caption('Hello World!')
```

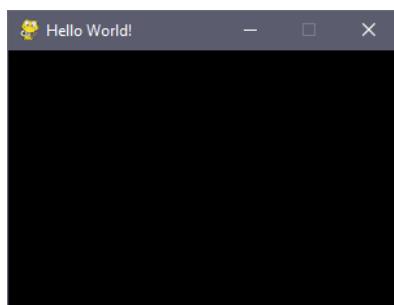
1.

```
# Game Loop - Le programme s'exécute jusqu'à ce
# que l'utilisateur quitte l'application
running = True
while running:

    #Est-ce que l'utilisateur a cliqué sur la croix ?
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

2.

Ce code produira une fenêtre vide (complètement noire) avec « Hello World » dans la barre de titre de la fenêtre :



Ce n'est pas très impressionnant, mais la création d'une fenêtre (que l'on appellera la **surface principale** de votre programme) est la toute première étape de la création de jeux !

Dans tout programme Pygame, le code peut être divisé en deux parties :

1. **Le bloc d'initialisation** : C'est là où on va initialiser les modules de Pygame, créer notre surface principale, définir des classes ou des fonctions et généralement initialiser l'ensemble des variables dont on aura besoin.
2. **La Game Loop** : C'est là où on va traiter les différentes actions du joueur et mettre à jour notre affichage. Nous reparlerons de la Game Loop plus loin dans ce cours.

Vous remarquerez que lorsque vous cliquez sur la croix en haut à droite de la fenêtre, votre programme se termine et la fenêtre disparaît. Cela est dû au code qui est placé dans la Game Loop. Nous en reparlerons lorsque nous aborderons **les évènements** dans Pygame.

Evidemment, les appels aux fonctions « print() » et « input() » ne serviront à rien ici puisque ces fonctions sont conçues pour être utilisées sur des programmes CLI. Nous verrons donc comment afficher du texte et comment récupérer les entrées clavier de l'utilisateur grâce à des fonctions qui sont propres à la bibliothèque Pygame.

Mais avant ça, nous allons analyser ce qu'il se passe dans notre code minimal en détail !

CONFIGURER UN PROGRAMME PYGAME BASIQUE

Les premières lignes du code minimal sont des lignes qui commenceront presque tous les programmes que vous écrirez en utilisant Pygame :

```
import pygame
```

Cette ligne permet simplement d'importer toutes les fonctions et modules de la bibliothèque Pygame. Ces modules fournissent une couche d'abstraction pour accéder au matériel spécifique de votre système, ainsi que des méthodes uniformes pour travailler avec ce matériel. Par exemple, le module « display » permet un accès uniforme à votre écran vidéo, tandis que le module « mixer » vous permettra de lire des fichiers audio !

Cependant, avant d'utiliser la moindre fonction Pygame, vous devrez initialiser les modules de la bibliothèque grâce à la ligne suivante :

```
pygame.init()
```

Sans ces deux premières lignes, vos applications Pygame ne pourront pas fonctionner !

Ensuite, nous devons configurer la fenêtre d'affichage de votre programme. Vous fournissez soit une liste, soit un tuple qui spécifie la largeur et la hauteur de la fenêtre à créer. Le code minimal utilise un tuple pour créer une fenêtre de 300 pixels de large pour 200 pixels de haut :

```
surface = pygame.display.set_mode((300, 200))
```

La fonction « set_mode() » du module « display » de Pygame renvoie un objet de type « Surface » que nous stockons dans une variable. Cela nous permettra par la suite d'influer en temps réel sur la fenêtre de notre programme (pour y dessiner des choses ou la remplir d'une couleur par exemple).

Enfin, la dernière ligne du bloc d'initialisation nous permet de donner un nom à la fenêtre de notre programme :

```
pygame.display.set_caption('Hello World!')
```

« GAME LOOP » ET « GAME STATE »

Passons désormais à la deuxième partie du code :

```
# Game Loop - Le programme s'exécute jusqu'à ce
# que l'utilisateur quitte l'application
running = True
while running:

    #Est-ce que l'utilisateur a cliqué sur la croix ?
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

Cette partie du code est placée dans un boucle « While » dont la condition est forcée à « True » via la variable « running ». Cela signifie que nous ne quitterons jamais la boucle tant que la variable « running » ne deviendra pas faux, ce qui arrive uniquement lorsque nous quittons le programme.

Ce boucle est donc destinée à boucler indéfiniment tant que votre jeu est en cours d'exécution. **C'est ce qu'on appelle la Game Loop.**

Tous les jeux, de Pong à Overwatch, utilisent une boucle de jeu pour contrôler le gameplay. La boucle de jeu fait quatre choses très importantes :

- Elle traite les entrées de l'utilisateur
- Met à jour l'état de tous les objets du jeu (on actualise le « Game State »)
- Met à jour l'affichage et la sortie audio (on fait un rendu du « Game State »)
- Elle maintient la vitesse du jeu (taux de rafraîchissement).

Chaque cycle de la boucle de jeu s'appelle une image, et plus votre code s'exécute vite, plus votre jeu sera rapide. Les frames se succèdent jusqu'à ce qu'une condition de sortie du jeu soit remplie.

Le « Game State » fait simplement référence à l'ensemble des valeurs que prennent toutes les variables d'un jeu. Par exemple, le « Game State » peut faire référence aux variables qui traquent la position du joueur ou son état de santé, son score, etc...

Lorsque quelque chose se produit au sein de votre programme, lorsque le joueur subit des dégâts ou lorsqu'un ennemi se déplace par exemple, on dit que le « Game State » a changé.



Étant donné que le « Game State » est généralement mis à jour en réponse à des événements (tels que des clics de souris ou des pressions sur le clavier) ou au passage du temps, la boucle de jeu vérifie et revérifie constamment, plusieurs fois par seconde, tout nouvel événement survenu. Cela est réalisé grâce au code qui se trouve dans la Game Loop :

```
#Est-ce que l'utilisateur a cliqué sur la croix ?
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        running = False
```

Ces lignes analysent et gèrent les événements dans la boucle de jeu. Dans ce cas, le seul événement géré est « pygame.QUIT », qui se produit lorsque l'utilisateur clique sur le bouton de fermeture de la fenêtre. Nous aurons l'occasion d'aborder les événements plus en détails plus loin dans ce syllabus.

LES SURFACES

La bibliothèque Pygame comprend également plusieurs classes qui peuvent être instanciées et utilisée pour réaliser tout un tas d'opérations. L'une de ces classes est la « Surface » qui, dans sa forme la plus simple, **définit une zone rectangulaire sur laquelle vous pouvez dessiner**. Un objet de type « Surface » représente donc une image 2D dont les pixels peuvent être modifiés !

Les objets de type « Surface » sont utilisés dans de nombreux contextes au sein d'un programme Pygame. Par exemple, dans la section sur les images, vous verrez comment charger une image à partir d'un fichier dans une surface et comment l'afficher à l'écran.

LA SURFACE PRINCIPALE

Avec Pygame nous ne créons donc pas de fenêtre (c'est plutôt votre système d'exploitation qui s'en charge). Ainsi, les **bordures de fenêtre, la barre de titre et les boutons** (mettre en arrière-plan, fermer la fenêtre...) **ne font pas partie de l'objet « Surface »**.

Tout est visualisé sur un seul affichage créé par l'utilisateur. Cet affichage est créé à l'aide de la méthode « set_mode ((x, y)) » du module « display » qui renvoie une surface représentant la partie visible de la fenêtre :

```
screen = pygame.display.set_mode((1280, 720))
```

Cette fonction prend en paramètre un tuple de deux éléments qui va définir la résolution de la surface que vous voulez créer. Le premier et le deuxième élément de ce tuple définissent respectivement le nombre de pixels de largeur et le nombre de pixels de hauteur que doit compter votre surface.

On aurait très bien pu écrire cela sous la forme :

```
resolution = (1280, 720)
screen = pygame.display.set_mode(resolution)
```

La variable « screen » contient donc ici un objet de type « Surface ».

Cette surface va nous servir de « tableau noir » sur laquelle nous pourrons écrire, dessiner, afficher des images ou jouer des animations (qui sont en réalité des successions d'images), etc...

Cette méthode « set_mode » contient des paramètres supplémentaires optionnels qui vont modifier la façon dont votre surface va se comporter à l'écran :

- **pygame.FULLSCREEN** : la surface sera rendue en plein écran
- **pygame.RESIZABLE** : rend la surface redimensionnable par l'utilisateur
- **pygame.NOFRAME** : masque les contrôles de fenêtres de votre OS
- **pygame.DOUBLEBUF** : double buffering
- **pygame.HWSURFACE** : accélération matérielle (déporte une partie des calculs sur certains composants dédiés dans votre ordinateur)

Voici comment ces différents arguments peuvent se combiner lors de l'appel à la méthode « set_mode » :

```
resolution = (1280, 720)
surface = pygame.display.set_mode(resolution, pygame.FULLSCREEN | pygame.NOFRAME)
```

Les différents arguments optionnels doivent être séparés par un « | » pour pouvoir être combinés !

Pour modifier le titre que prendra votre fenêtre, vous pouvez utiliser la méthode « set_caption(name) » du module « display » :

```
pygame.display.set_caption("Ma première fenêtre Pygame !")
```

LE PRINCIPE DE MODIFICATION D'UNE SURFACE

Il y'a une petite subtilité dans la façon de dessiner ou d'altérer les pixels d'une surface avec Pygame. Voyez-vous, dessiner un objet « Surface » sur l'écran de votre ordinateur n'est pas une opération qui est très rapide. Nous allons donc éviter de réafficher notre surface entre chaque modification de celle-ci.

L'objet de type « Surface » existe dans la mémoire de votre ordinateur. C'est un ensemble de données. Lorsque nous devons dessiner des images ou des formes sur une surface, nous altérons ces données en mémoire grâce aux différentes méthodes et fonctions que nous verrons dans ce chapitre.

Ainsi, à chaque itération de notre Game Loop, nous allons d'abord « dessiner » l'ensemble des éléments qui doivent apparaître à l'écran dans notre objet « Surface » (c'est-à-dire qu'on va simplement modifier les données stockées dans la mémoire de l'ordinateur) **avant de l'afficher à l'écran.**

En effet, aucune modification de votre surface ne sera perceptible à l'écran tant que nous n'appelons pas la fonction « flip() » (ou « update() ») du module « display ». **Ce rafraîchissement de l'affichage ne sera donc effectué qu'une seule fois par itération de la Game Loop.**

Cette manière de procéder est beaucoup plus performante car la mémoire d'un ordinateur est plus rapide à changer que les pixels d'un écran.

DÉFINIR UNE COULEUR – RGB MODEL

En informatique, les couleurs sont définies comme des tuples des couleurs de base rouge, vert et bleu. C'est ce qu'on appelle le modèle RGB (ou RVB en français).

Chaque couleur de base est représentée par un nombre compris entre 0 (minimum) et 255 (maximum) qui occupe 1 octet en mémoire. Une couleur RGB est donc représentée par une valeur de 3 octets.

Par exemple :

```
R = 50  
G = 90  
B = 255  
color = (R, G, B)
```

La première valeur du tuple est la quantité de rouge contenue dans la couleur. Une valeur entière de 0 signifie qu'il n'y a pas de rouge dans cette couleur, et une valeur de 255 signifie qu'il y a la quantité maximale de rouge dans la couleur. La deuxième valeur est pour le vert et la troisième valeur est pour le bleu.

En mélangeant deux couleurs ou plus, on obtient de nouvelles couleurs. Au total, 16 millions de couleurs différentes peuvent être représentées de cette manière. Voici les valeurs RGB pour des couleurs communes :

Color	RGB Values
Aqua	(0, 255, 255)
Black	(0, 0, 0)
Blue	(0, 0, 255)
Fuchsia	(255, 0, 255)
Gray	(128, 128, 128)
Green	(0, 128, 0)
Lime	(0, 255, 0)
Maroon	(128, 0, 0)
Navy Blue	(0, 0, 128)
Olive	(128, 128, 0)
Purple	(128, 0, 128)
Red	(255, 0, 0)
Silver	(192, 192, 192)
Teal	(0, 128, 128)
White	(255, 255, 255)
Yellow	(255, 255, 0)

REmplir la surface avec une couleur unie

Par défaut, nous avons vu que la surface que l'on crée avec la méthode « `set_mode` » du module « `display` » est entièrement noire.

Pour changer la couleur de notre surface, nous pouvons utiliser la méthode « `fill(color)` » de la classe « `Surface` » :

```
surface_res = (1280, 720)
surface = pygame.display.set_mode(surface_res)

R = 55
G = 90
B = 255
color = (R, G, B)

surface.fill(color)
```

Cependant, si on exécute le code ci-dessus, notre surface reste noire...

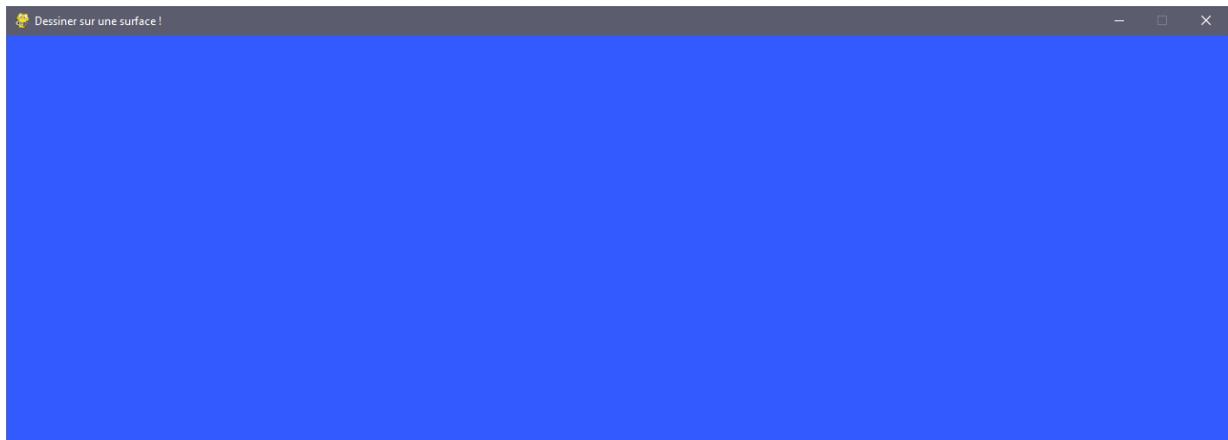
Ne vous inquiétez-pas, c'est normal ! En effet, avec Pygame, dès que vous allez modifier quelque chose au niveau de votre affichage, il faudra rafraîchir votre surface !

Mettre à jour l'affichage

Pour mettre à jour votre surface, il faudra utiliser la méthode « `flip()` » du module « `display` » :

```
pygame.display.flip()
```

Si on exécute maintenant le code, la fenêtre ressemble à ceci :

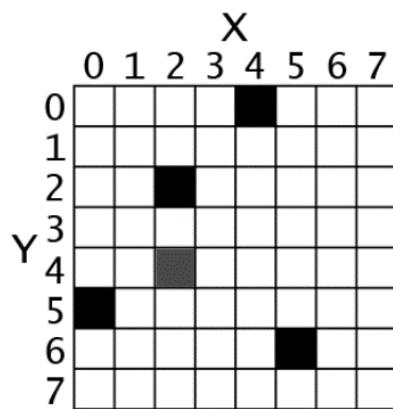


COORDONNÉES DES PIXELS

Avant de commencer à dessiner sur notre surface, faisons un point sur la notion de pixels. La surface que nous avons créée dans l'exemple précédent est **composée de petits points carrés sur votre écran appelés pixels.**

Chaque pixel est noir au départ, mais il peut être indépendamment réglé sur une couleur différente.

Imaginez que nous avons une petite surface carrée de 8 pixels de côté. Si nous agrandissons cette surface de manière à ce que chaque pixel ressemble à un carré dans une grille et que nous ajoutons des chiffres le long des axes X et Y, alors notre surface ressemblerait à ceci :



Nous pouvons nous référer à un pixel spécifique en utilisant un système de coordonnées cartésiennes. Chaque colonne de l'axe X et chaque ligne de l'axe Y aura une valeur qui est un entier de 0 à 7. Ainsi, nous pouvons localiser n'importe quel pixel en spécifiant les entiers des axes X et Y.

Attention, vous remarquerez que l'axe Y pointe vers le bas et que le point (0, 0) de votre surface se trouve en haut à gauche !

Par exemple, dans l'image 8x8 ci-dessus, nous pouvons voir que les pixels aux coordonnées (4, 0), (2, 2), (0, 5), et (5, 6) ont été peints en noir, le pixel à (2, 4) a été peint en gris, tandis que tous les autres pixels sont peints en blanc.

Pygame représente souvent les coordonnées cartésiennes sous la forme d'un tuple de deux entiers, comme (4, 0) ou (2, 2). Le premier entier est la coordonnée X et le second est la coordonnée Y.

« DRAWING PRIMITIVES »

Pygame fournit plusieurs fonctions différentes pour dessiner des formes sur un objet de type « Surface ».

Ces formes telles que les rectangles, les cercles, les ellipses, les lignes ou les pixels individuels sont souvent appelées « Drawing Primitives ». Attention, cela n'inclut pas les images comme celles des fichiers .png ou .jpg.

Pour pouvoir dessiner sur notre surface, nous allons avoir besoin des fonctions qui se trouvent dans le module « draw » de la bibliothèque Pygame.

Ces fonctions de dessin possèdent plusieurs points communs entre elles :

- Elles prennent un objet de type « Surface » comme premier argument
- Elles prennent une couleur comme deuxième argument
- Elles renvoient un objet de type « Rect » qui délimite la zone modifiée (nous aborderons les objets « Rect » un petit peu plus loin dans ce syllabus).

DESSINER UNE LIGNE

Pour pouvoir dessiner sur notre surface, nous allons avoir besoin du module « draw » de la bibliothèque Pygame.

Ce module « draw » possède une fonction « line(surface, color, start_pos, end_pos) » qui va nous permettre de tracer une ligne :

```
surface_res = (500, 500)

surface = pygame.display.set_mode(surface_res)
surface.fill(color)

black_color = (0, 0, 0)
pygame.draw.line(surface, black_color, [10, 10], [490, 490])

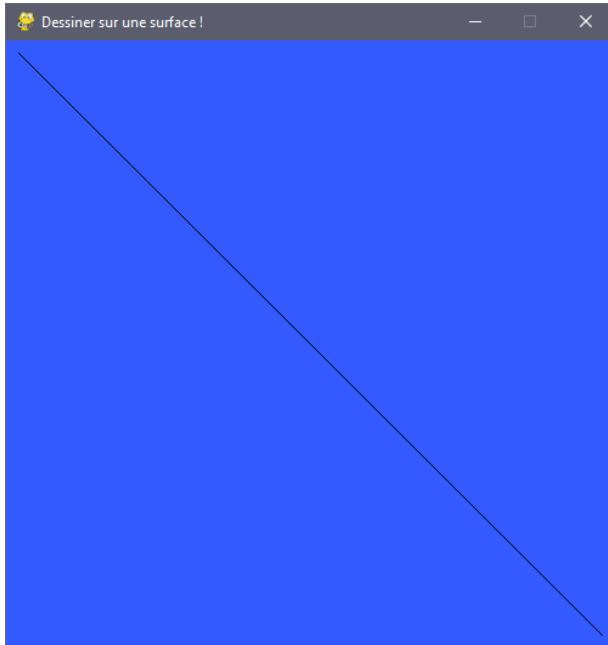
pygame.display.flip()
```

Le premier argument de la fonction « line » est une référence à la surface sur laquelle nous souhaitons dessiner et le deuxième argument est la couleur de la ligne.

Viennent ensuite les arguments 3 et 4 qui représentent respectivement les coordonnées en X et en Y de la position de départ et de la position de fin du tracé. Ces arguments peuvent être exprimés à l'aide de tuples ou, comme dans l'exemple ci-dessus, à l'aide de listes.

Note : Pour tracer un point, il suffit que les coordonnées de départ du tracé soient égales aux coordonnées de fin.

Voici ce qu'un tel code produit comme résultat :

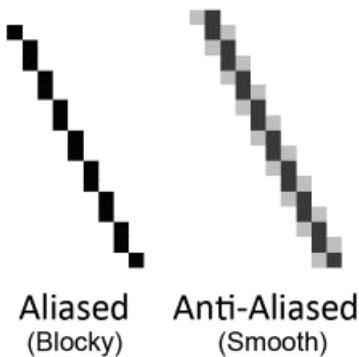


QU'EST-CE QUE L'ANTI-ALIASING

L'anti-aliasing ou l'anticrénelage en français est une technique graphique qui permet de rendre le texte et les formes moins grossiers en ajoutant un peu de flou à leurs bords.

Il faut un peu plus de temps de calcul pour dessiner avec l'anti-aliasing, donc bien les rendus visuels soient plus beaux, votre programme peut s'exécuter un petit peu plus lentement.

Si vous faites un zoom sur une ligne avec anti-aliasing et une ligne sans anti-aliasing, elles ressemblent à ceci :



Pour dessiner une ligne avec l'anti-aliasing activé, nous pouvons utiliser la fonction « `aaline(surface, color, start_pos, end_pos)` » du module « `draw` ». Cette fonction possède exactement les même paramètres que « `line(surface, color, start_pos, end_pos)` » vu précédemment.

La différence n'est pas spectaculaire si on reprend l'exemple précédent, mais cette technique aura toute son importance lorsque l'on parlera de l'affichage de texte !

DESSINER UN RECTANGLE

Pour pouvoir tracer un rectangle sur notre surface, nous allons utiliser la fonction « rect(surface, color, rect, width, border_radius) » du module « draw ».

Vous voyez que le troisième argument de cette fonction s'appelle également « rect ». En réalité, les rectangles dans Pygame sont considérés comme des objets (Nous verrons la classe « Rect » en détail un peu plus loin dans ce syllabus). La fonction s'attend donc à ce qu'on lui passe un objet de type « Rect » en paramètre.

Cet objet « Rect » peut-être créé dans une variable intermédiaire pour augmenter la lisibilité de votre code. Le constructeur de la classe « Rect » prends en paramètres les arguments suivants : le décalage en pixel à gauche de la surface, le décalage en pixel en haut de la surface, la largeur et la hauteur du rectangle.

Voici comment créer un objet rectangle dont le coin supérieur gauche se trouve aux coordonnées (10, 10) de notre surface et qui fait 150 pixels de large sur 65 pixels de haut.

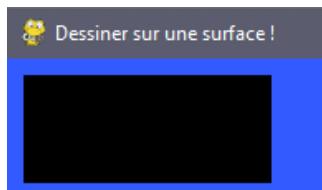
```
monRectangle = pygame.Rect(10, 10, 150, 65)
```

Pour dessiner ce rectangle sur notre surface, nous allons faire appel à la fonction « rect » vue précédemment :

```
monRectangle = pygame.Rect(10, 10, 150, 65)
pygame.draw.rect(surface, black_color, monRectangle)

pygame.display.flip()
```

Voici le résultat produit :

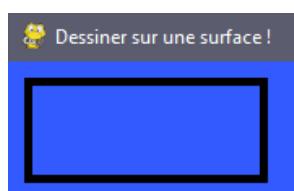


Vous remarquerez que, par défaut, le rectangle est rempli de la couleur spécifiée dans la fonction « rect ». Si l'on souhaite uniquement obtenir un contour de couleur et ne pas remplir l'intégralité du rectangle, nous pouvons renseigner un quatrième paramètre optionnel « width » à la fonction. Ce paramètre définira la largeur en pixel de votre contour de couleur :

```
monRectangle = pygame.Rect(10, 10, 150, 65)
pygame.draw.rect(surface, black_color, monRectangle, 5)

pygame.display.flip()
```

Voici le résultat produit :

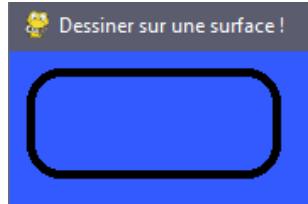


Un cinquième argument optionnel « border_radius » permet d'obtenir un contour arrondi :

```
monRectangle = pygame.Rect(10, 10, 150, 65)
pygame.draw.rect(surface, black_color, monRectangle, 5, 20)

pygame.display.flip()
```

Voici le résultat produit :



DESSINER UN CERCLE

Pour pouvoir tracer un cercle sur notre surface, nous allons utiliser la fonction « circle(surface, color, center, radius, width) » du module « draw » :

```
pygame.draw.circle(surface, black_color, [100, 100], 80)

pygame.display.flip()
```

Le troisième argument représente les coordonnées du centre de votre cercle sur la surface. Le quatrième argument représente le rayon de votre cercle en pixels.

Voici le résultat produit :

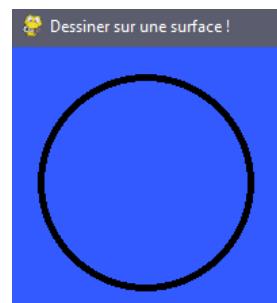


Comme pour le rectangle, un cinquième paramètre optionnel « width » vous permet de ne pas remplir l'entièreté du cercle mais plutôt d'obtenir un contour de couleur :

```
pygame.draw.circle(surface, black_color, [100, 100], 80, 5)

pygame.display.flip()
```

Voici le résultat produit :



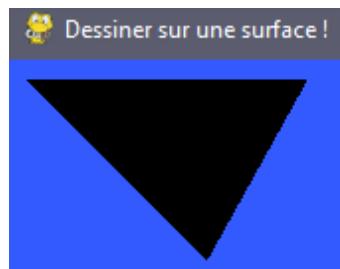
TRACER UN POLYGONE

Pour pouvoir tracer un polygone sur notre surface, nous allons utiliser la fonction « polygon(surface, color, points, width) » du module « draw ».

L'argument « points » est une liste de coordonnées qui représenteront les différents points constitutifs de votre polygone. Encore une fois, puisque nous travaillons en 2D, ces points ont une coordonnée en X et une coordonnée en Y. Nous obtiendrons donc une liste de tuples :

```
points = [(10,10), (150, 10), (100, 100)]  
pygame.draw.polygon(surface, black_color, points)  
  
pygame.display.flip()
```

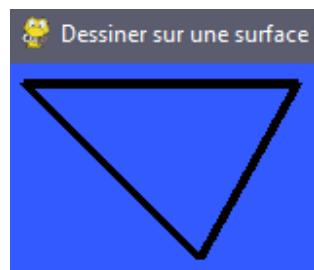
Voici le résultat produit :



Comme pour le rectangle et le cercle, le quatrième argument optionnel « width » vous permet d'obtenir un contour de couleur :

```
points = [(10,10), (150, 10), (100, 100)]  
pygame.draw.polygon(surface, black_color, points, 5)  
  
pygame.display.flip()
```

Voici le résultat produit :

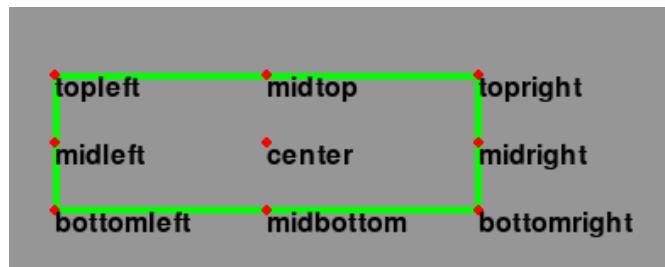


LA CLASSE « RECT »

Le rectangle est un objet très utile en programmation graphique. Il possède sa propre classe « Rect » dans Pygame et est utilisé pour stocker et manipuler une zone rectangulaire.

LES ATTRIBUTS DE LA CLASSE « RECT »

La classe Rect définit 4 points d'angle, 4 points médians et 1 point central.



Ces points, en plus des coordonnées en X et en Y du rectangle, sont en réalité des attributs de la classe « Rect » peuvent être utilisés pour déplacer le rectangle sans modifier la taille.

Par exemple, si on veut que notre rectangle soit positionné tout en bas à droite de notre fenêtre :

```
surface_res = (200, 200)
surface = pygame.display.set_mode(surface_res)

monRectangle = pygame.Rect(0, 0, 30, 30)
monRectangle.bottomright = surface_res

pygame.draw.rect(surface, red_color, monRectangle)
pygame.display.flip()
```

Voici le résultat :



Voici la liste des attributs de la classe « Rect » :

Attribute Name	Description
myRect.left	The int value of the X-coordinate of the left side of the rectangle.
myRect.right	The int value of the X-coordinate of the right side of the rectangle.
myRect.top	The int value of the Y-coordinate of the top side of the rectangle.
myRect.bottom	The int value of the Y-coordinate of the bottom side.
myRect.centerx	The int value of the X-coordinate of the center of the rectangle.
myRect.centery	The int value of the Y-coordinate of the center of the rectangle.
myRect.width	The int value of the width of the rectangle.
myRect.height	The int value of the height of the rectangle.
myRect.size	A tuple of two ints: (width, height)
myRect.topleft	A tuple of two ints: (left, top)
myRect.topright	A tuple of two ints: (right, top)
myRect.bottomleft	A tuple of two ints: (left, bottom)
myRect.bottomright	A tuple of two ints: (right, bottom)
myRect.midleft	A tuple of two ints: (left, centery)
myRect.midright	A tuple of two ints: (right, centery)
myRect.midtop	A tuple of two ints: (centerx, top)
myRect.midbottom	A tuple of two ints: (centerx, bottom)

BOUGER UN RECTANGLE

Bien que nous puissions directement modifier les attributs « x » et « y » d'un objet de type « Rect » comme nous venons de le voir, nous pouvons également utiliser des méthodes de cette classe pour bouger un rectangle.

La première méthode est « move(x, y) ». Les arguments « x » et « y » définissent respectivement le déplacement à effectuer le long des axes correspondants. **Ici on ne spécifie donc pas les coordonnées finales du rectangle sur l'écran mais bien un vecteur de déplacement !**

Cependant, la méthode « move(x, y) » ne déplace pas directement l'objet « Rect » sur lequel elle est appelée. Au lieu de cela, elle renvoie un nouvel objet « Rect » qui s'est déplacé le long du vecteur spécifié :

```
surface_res = (200, 200)
surface = pygame.display.set_mode(surface_res)

monRectangle = pygame.Rect(0, 0, 30, 30)
monNouveauRectangle = monRectangle.move(170, 170)

pygame.draw.rect(surface, red_color, monRectangle)
pygame.draw.rect(surface, red_color, monNouveauRectangle)

pygame.display.flip()
```

Voici le résultat :



Une deuxième méthode appelée « move_ip(x, y) » de la classe « Rect » permet quand à elle de bien déplacer le rectangle sur lequel elle est appelée au lieu de créer une copie. Par exemple, on pourrait imaginer créer un code qui déplacerait notre cube vers la droite de 1 pixel toutes les 0.01 secondes :

```
import pygame
import time

pygame.init()

surface_res = (200, 200)
surface = pygame.display.set_mode(surface_res)

i = 0
black_color = (0, 0, 0)
red_color = (255, 0, 0)
blue_color = (0, 0, 255)

monRectangle = pygame.Rect(10, 10, 30, 30)

pygame.draw.rect(surface, red_color, monRectangle)
pygame.display.flip()

running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    time.sleep(0.01)
    surface.fill(black_color)
    monRectangle.move_ip(1, 0)
    pygame.draw.rect(surface, red_color, monRectangle)
    pygame.display.flip()
```

Vous voyez que pour pouvoir donner l'illusion que le rectangle se déplace sur l'écran, nous devons redessiner l'entièreté de ce qui se trouve sur la surface ! (on remplit la surface en noir et on vient redessiner les éléments par-dessus à chaque itération de la Game Loop !)

AGRANDIR UN RECTANGLE

Il est évidemment possible de modifier la taille d'un rectangle grâce aux méthodes « `inflate(x, y)` » et « `inflate_ip(x, y)` ». Ces méthodes fonctionnent de la même manière que les méthodes que nous venons de voir pour déplacer un rectangle ; la première crée une copie et la seconde travaille directement sur l'objet « `Rect` » sur lequel elle est appelée :

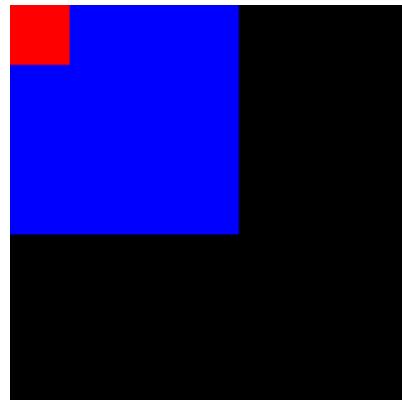
```
monRectangle = pygame.Rect(0, 0, 30, 30)

monNouveauRectangle = monRectangle.inflate(170, 170)

pygame.draw.rect(surface, blue_color, monNouveauRectangle)
pygame.draw.rect(surface, red_color, monRectangle)

pygame.display.flip()
```

Ce qui donne :



LES COLLISIONS

L'une des plus grande utilité de la classe « Rect » est qu'elle permet la détection des collisions entre plusieurs objets du type « Rect » entre eux ! Par exemple, cela nous sera particulièrement utile pour détecter la collision entre notre joueur et l'environnement !

Pour détecter une collision entre deux rectangles, on peut utiliser la méthode « collidrect(rect) » de la classe « Rect ». **Cette méthode renvoie un booléen s'il y'a ou non une collision qui est détectée sur le rectangle qu'on lui passe en paramètre :**

```
pygame.draw.rect(surface, red_color, monRectangle)
pygame.draw.rect(surface, blue_color, monMur)
pygame.display.flip()

collision = False
running = True

while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    time.sleep(0.01)
    surface.fill(black_color)

    if (monRectangle.collidrect(monMur)):
        collision = True

    if (collision == True):
        monRectangle.move_ip(-1, 0)
    else:
        monRectangle.move_ip(1, 0)

    pygame.draw.rect(surface, red_color, monRectangle)
    pygame.draw.rect(surface, blue_color, monMur)

    pygame.display.flip()
```

Le code ci-dessus va faire déplacer le rectangle rouge vers la droite jusqu'à ce qu'il collisionne le rectangle bleu. A partir de ce moment, le rectangle rouge va se déplacer dans l'autre sens et s'éloigner peu à peu du rectangle bleu.

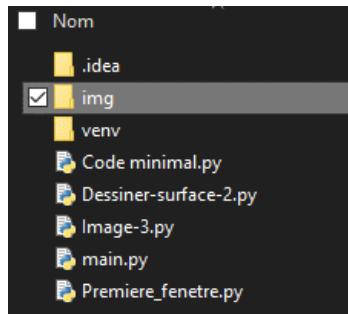
AFFICHER DES IMAGES

Jusqu'à présent, nous avions évoqués les différentes fonctions qui permettaient de dessiner sur notre surface. A titre explicatif, nous avions placés ces fonctions en dehors de notre Game Loop. Cependant, à partir de maintenant, tout ce qui sera lié à l'affichage sera exécuté dans la Game Loop.

Tout ce qui est lié à l'initialisation de la fenêtre ou au chargement des ressources restera en dehors de la Game Loop.

CHARGER UNE IMAGE

Tout d'abord, rendez-vous dans le dossier de votre projet Pycharm. Créez ensuite un dossier nommé « img » à la racine de ce dossier (au même niveau que vos scripts) :



C'est dans ce dossier que vous placerez les images auxquelles vous voudrez accéder.

De retour dans notre script, nous allons utiliser la fonction « load(path) » du module « image » de la bibliothèque Pygame :

```
myImage = pygame.image.load("img/super-mario.png")
```

Cette fonction « load » prendra en argument le chemin qui pointe vers l'image que vous souhaitez charger. **Attention, cette fonction vous retournera une surface. La variable « myImage » est donc bien de type « Surface ».**

INTÉGRER NOTRE IMAGE

Pour intégrer notre image à notre surface principale, nous allons utiliser la méthode « blit(source, dest) » de la classe « Surface » :

```
myImage = pygame.image.load("img/super-mario.png")

running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    surface.fill(color)
    surface.blit(myImage, [125, 50])
    pygame.display.flip()
```

Le premier argument de cette fonction représente la surface à appliquer (notre image). Le deuxième argument représente les coordonnées où on va placer le point qui se situe en haut à gauche de notre image.

Voici le résultat obtenu :



La fonction « load » gère la plupart des formats d'images, que ce soit du .png, du .jpg, du .TIFF, etc... Ces formats peuvent donc être des formats avec compression, avec canal alpha (pour la transparence)....

AFFICHER DU TEXTE

Pour afficher du texte sur une surface nous allons nous servir du module « font » de la bibliothèque Pygame. Ce module comporte deux constructeurs différents pour afficher du texte.

CHARGER UNE POLICE DE VOTRE OS

Le premier va utiliser les polices d'écriture qui sont installées sur votre système d'exploitation. Pour savoir quelles polices sont installées sur votre OS, vous pouvez simplement exécuter la ligne de code suivante qui vous retournera dans la console la liste de toutes les polices présentes sur le système :

```
print(pygame.font.get_fonts())
```

Pour charger une police présente sur le système vous pouvez utiliser la fonction « SysFont(name, size, bold, italic) » du module « font ».

Cette fonction prend en paramètre le nom de la police à utiliser et dans quelle taille cette police doit être affichée. Cette fonction va retourner un objet de la classe « Font » :

```
myFont = pygame.font.SysFont("arial", 25)
```

Deux paramètres booléen optionnels « bold » et « italic » permettent de mettre la police en gras et en italique :

```
myFont = pygame.font.SysFont("arial", 25, True, True)
```

Pour préparer le rendu du texte que nous souhaitons afficher à l'écran, nous allons utiliser la méthode « `render(text, antialias, color, background)` » de la classe « `Font` » :

```
black_color = (0,0,0)

myFont = pygame.font.SysFont("arial", 25, True, True)
myText = myFont.render("Hello World", False, black_color)
```

Le premier argument de cette fonction est la chaîne de caractère à afficher. Le deuxième est un booléen qui détermine si on doit appliquer de l'anti-aliasing sur le texte. Enfin, le troisième argument permet de définir la couleur du texte. Un quatrième argument optionnel « `background` » permet de donner une couleur de fond au texte.

La méthode « `render` » renvoie une surface. La variable « `myText` » est donc de type « `Surface` » !

Nous avons déjà vu comment appliquer une surface sur une autre lorsque nous avons parlé des images plus tôt dans ce syllabus. Pour ce faire, nous allons utiliser la méthode « `blit` » de la classe « `Surface` » :

```
import pygame

pygame.init()
pygame.display.set_caption("Afficher du texte !")

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res)

black_color = (0,0,0)

myFont = pygame.font.SysFont("arial", 40, True, True)
myText = myFont.render("Hello World", False, black_color)

running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    surface.fill((255, 80, 80))
    surface.blit(myText, [20, 20])
    pygame.display.flip()
```

Voici le résultat (sans anti-aliasing) :



Avec anti-aliasing :

```
myFont = pygame.font.SysFont("arial", 40, True, True)
myText = myFont.render("Hello World", True, black_color)
```



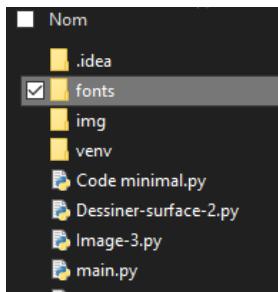
Evidemment, dès que l'on voudra changer le texte, il faudra faire un nouveau rendu et effacer l'ancien ! **De plus, Les rendus ne prennent pas en compte les caractères de retour à la ligne.** Pour écrire un texte sur deux lignes, il faudra faire 2 rendus !

Attention ! Charger des polices de caractères à partir de votre OS est une méthode qui n'est pas conseillée car elle n'est que peu portable (si votre programme doit tourner sur une autre machine qui ne possède pas cette police installée, vous rencontrerez des erreurs !)

CHARGER UNE POLICE A PARTIR D'UN FICHIER

Une méthode à privilégier pour afficher du texte dans vos programme est de charger la police à utiliser à partir d'un fichier qui se trouve dans le dossier de votre projet Pycharm.

Tout d'abord, téléchargez une police quelconque et ajoutez-là dans un dossier « fonts » à la racine de votre projet Pycharm (là où se trouvent vos différents scripts) :



Pour charger le fichier de police, nous utiliserons cette fois la fonction « `Font(path, size)` » du module « `font` ». Le premier argument est le chemin qui point vers le fichier de la police à charger. Le second est la taille de la police, comme vu précédemment :

```
myFont = pygame.font.Font("fonts/Nexa-Trial-Regular.ttf", 40)
myText = myFont.render("Hello World", True, black_color)
```

Attention, ici vous n'avez plus l'occasion de directement définir si la police doit être chargée en gras pu en italique. En effet, si vous voulez que votre police s'affiche en gras par exemple, il vous faudra charger le fichier de police correspondant !

Voici le résultat :



MÉTHODES UTILES POUR MANIPULER LES TEXTES

La classe « Font » possède plusieurs méthodes bien utiles pour manipuler du texte :

```
myFont = pygame.font.SysFont("arial", 40)

myFont.set_bold(True)      # Met la police en gras
myFont.set_italic(True)    # Met la police en italique
myFont.set_underline(True) # Souligne la police
```

LES ÉVÈNEMENTS

La pression des touches, les mouvements de la souris et même les mouvements du joystick sont quelques-uns des moyens par lesquels un utilisateur peut fournir des données.

Toute entrée utilisateur entraîne la génération de ce qu'on appelle un événement. Ils peuvent survenir à tout moment et proviennent généralement (mais pas toujours) de l'extérieur du programme.

Pygame enregistre les différents événements qui se produisent dans ce qu'on appelle une « file d'événements ». Ceux-ci sont donc stockés dans une file d'attente où ils seront ensuite traité (ou non) un à un.

Il est donc possible d'associer du code à exécuter dès qu'ils sont détectés (pour bouger un personnage au clavier par exemple). Réagir aux événements est donc essentiel pour permettre à l'utilisateur d'interagir avec votre application !

Jusqu'à présent, nous avions déjà du code dans notre Game Loop qui parcourait cette file d'événements :

```
running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

En effet, nous avons une boucle « for » qui parcourt la structure renvoyée par la fonction « `pygame.event.get()` ». C'est via cette fonction que vous accédez à la liste de tous les événements actifs dans la file d'attente.

Le principe général de la gestion d'évènement consiste donc à parcourir cette liste en boucle, inspecter chaque type d'événement et y répondre en conséquence.

On peut d'ailleurs voir que nous avons également une structure conditionnelle « `if` » qui teste le type d'évènement que nous rencontrons au sein de la boucle. Ici, nous regardons simplement si un évènement de type « `QUIT` » a eu lieu afin de fermer le programme. Ce type d'évènement se déclenche quand on appuie sur la petite « croix » pour fermer notre fenêtre de jeu par exemple.

Ainsi, chaque événement dans pygame est associé à **un type d'événement**. Les types d'événements sur lesquels vous allez principalement vous concentrer sont les pressions de touches et la fermeture de la fenêtre.

Les événements de pression de touches sont de type « KEYDOWN », et l'événement de fermeture de la fenêtre, comme vu dans l'exemple ci-dessus, est de type « QUIT ».

Les différents types d'événements peuvent également être associés à d'autres données. Par exemple, le type d'événement KEYDOWN possède également une variable appelée key pour indiquer quelle touche a été enfoncée.

Voici une liste des types d'événements que vous pouvez rencontrer, ainsi que les variables associées :

QUIT	none
ACTIVEEVENT	gain, state
KEYDOWN	unicode, key, mod
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy, axis, value
JOYBALLMOTION	joy, ball, rel
JOYHATMOTION	joy, hat, value
JOYBUTTONUP	joy, button
JOYBUTTONDOWN	joy, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	none
USEREVENT	code

ÉVÈNEMENT LIÉ AU REDIMENSIONNEMENT DE LA FENÊTRE

Dans cette section nous allons voir comment déclencher du code quand l'utilisateur redimensionne la fenêtre de jeu.

Pour rappel, pour permettre au joueur d'influer sur la taille de la fenêtre, il vous faut spécifier un paramètre optionnel « pygame.RESIZABLE » lorsque vous créer votre surface principale :

```
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)
```

Pour cet exemple, nous allons créer un texte qui va automatiquement se mettre à jour pour nous donner les dimensions exactes de notre fenêtre à n'importe quel moment de l'exécution de notre jeu.

En premier lieu, nous allons créer un code qui va afficher par défaut les dimensions de notre fenêtre au démarrage du jeu :

```
import pygame

pygame.init()
pygame.display.set_caption("Les événements !")

white_color = (255, 255, 255)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)

myFont = pygame.font.Font("fonts/Nexa-Trial-Regular.ttf", 30)
myText = myFont.render("{} x {}".format(surface_res[0], surface_res[1]), True, white_color)

surface.blit(myText, [20, 20])
pygame.display.flip()

running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

Nous allons rajouter un « if » dans la Game Loop afin de vérifier si un des événements de la file correspond au type « VIDEORESIZE ». Ce type d'événement se déclenche justement dès que la fenêtre de jeu subit un redimensionnement.

Voici le code de la Game Loop qui permet de changer dynamiquement l'affichage du texte en fonction des nouvelles valeurs de hauteurs et de largeur de la fenêtre :

```
running = True
while running: # main game loop
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            running = False

        elif event.type == pygame.VIDEORESIZE :
            surface.fill(black_color) # On efface tout
            myText = myFont.render("{} x {}".format(event.w, event.h), True, white_color)
            surface.blit(myText, [20, 20]) # On réaffiche le texte
            pygame.display.flip() # On actualise l'affichage
```

Comme vu dans le tableau des types d'événements, nous pouvons, dans le cas d'un évènement de type « VIDEORESIZE », récupérer la hauteur et la largeur de la fenêtre grâce aux attributs « .w » et « .h ».

Voici le résultat :



ÉVÈNEMENTS LIÉS AU CLAVIER

Le type d'évènement qui se déclenchera lorsque le joueur appuiera sur une touche du clavier est « KEYDOWN » (ou « KEYUP » lorsqu'il relâche la touche). Si vous regardez dans le tableau des types d'évènements, vous remarquerez que nous pouvons récupérer quelle touche a été appuyée grâce à l'attribut « key ».

Voici un tableau qui récapitule la plupart des noms que peuvent prendre cet attribut « key » lorsque la touche associée est utilisée sur le clavier :

KeyASCII	ASCII	Common Name	K_0	0	0
K_BACKSPACE	\b	backspace	K_1	1	1
K_TAB	\t	tab	K_2	2	2
K_CLEAR		clear	K_3	3	3
K_RETURN	\r	return	K_4	4	4
K_PAUSE		pause	K_5	5	5
K_ESCAPE	^[escape	K_6	6	6
K_SPACE		space	K_7	7	7
K_EXCLAIM	!	exclaim	K_8	8	8
K_QUOTEDBL	"	quotedbl	K_9	9	9
K_HASH	#	hash	K_COLON	:	colon
K_DOLLAR	\$	dollar	K_SEMICOLON	;	semicolon
K_AMPERSLASH	&	ampersand	K_LESS	<	less-than sign
K_QUOTE	"	quote	K_EQUALS	=	equals sign
K_LEFTPAREN	(left parenthesis	K_GREATER	>	greater-than sign
K_RIGHTPAREN)	right parenthesis	K_QUESTION	?	question mark
K_ASTERISK	*	asterisk	K_AT	@	at
K_PLUS	+	plus sign	K_LEFTBRACKET	[left bracket
K_COMMASLASH	,	comma	K_BACKSLASH	\	backslash
K_MINUS	-	minus sign	K_RIGHTBRACKET]	right bracket
K_PERIOD	.	period	K_CARET	^	caret
K_SLASH	/	forward slash	K_UNDERSCORE	_	underscore
			K_BACKQUOTE	`	grave

K_a	a	a	K_w	w	w
K_b	b	b	K_x	x	x
K_c	c	c	K_y	y	y
K_d	d	d	K_z	z	z
K_e	e	e	K_DELETE		delete
K_f	f	f	K_KP0		keypad 0
K_g	g	g	K_KP1		keypad 1
K_h	h	h	K_KP2		keypad 2
K_i	i	i	K_KP3		keypad 3
K_j	j	j	K_KP4		keypad 4
K_k	k	k	K_KP5		keypad 5
K_l	l	l	K_KP6		keypad 6
K_m	m	m	K_KP7		keypad 7
K_n	n	n	K_KP8		keypad 8
K_o	o	o	K_KP9		keypad 9
K_p	p	p	K_KP_PERIOD	.	keypad period
K_q	q	q	K_KP_DIVIDE	/	keypad divide
K_r	r	r	K_KP_MULTIPLY	*	keypad multiply
K_s	s	s	K_KP_MINUS	-	keypad minus
K_t	t	t	K_KP_PLUS	+	keypad plus
K_u	u	u	K_KP_ENTER	\r	keypad enter
K_v	v	v	K_KP_EQUALS	=	keypad equals

K_UP	up arrow
K_DOWN	down arrow
K_RIGHT	right arrow
K_LEFT	left arrow
K_INSERT	insert
K_HOME	home
K_END	end
K_PAGEUP	page up
K_PAGEDOWN	page down
K_F1	F1
K_F2	F2
K_F3	F3
K_F4	F4
K_F5	F5
K_F6	F6
K_F7	F7
K_F8	F8
K_F9	F9
K_F10	F10
K_F11	F11
K_F12	F12
K_F13	F13

Dans cet exemple, nous allons créer un rectangle rouge et nous allons le déplacer à l'aide des touches ZQSD. Commençons par créer le carré rouge :

```
red_color = (255, 80, 80)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)

monRectangle = pygame.Rect(235, 235, 30, 30)
pygame.draw.rect(surface, red_color, monRectangle)

pygame.display.flip()
```

Nous allons ensuite créer un dictionnaire qui va nous permettre d'associer une touche à une direction pour déplacer le cube :

```
dir = {pygame.K_q: (-15, 0), pygame.K_d: (15, 0), pygame.K_z: (0, -15),
pygame.K_s: (0, 15)}
```

Voici le code complet qui permet de déplacer le rectangle:

```
import pygame

pygame.init()
pygame.display.set_caption("Les évènements !")

black_color = (0, 0, 0)
red_color = (255, 80, 80)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)

monRectangle = pygame.Rect(235, 235, 30, 30)
pygame.draw.rect(surface, red_color, monRectangle)

dir = {pygame.K_q: (-15, 0), pygame.K_d: (15, 0), pygame.K_z: (0, -15),
pygame.K_s: (0, 15)}

pygame.display.flip()

running = True
while running: # main game loop
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.KEYDOWN:
            if event.key in dir:
                v = dir[event.key]
                monRectangle.move_ip(v)

    surface.fill(black_color)
    pygame.draw.rect(surface, red_color, monRectangle)

    pygame.display.flip()
```

Ici on vérifie que, lorsqu'une touche du clavier est pressée, cette touche fait bien partie de notre dictionnaire, c'est-à-dire que le joueur à appuyer sur Z, Q, S ou D. Si c'est le cas, on récupère la direction du déplacement que l'on a associée à cette touche dans le dictionnaire et on fait bouger le rectangle.

ÉVÈNEMENTS LIÉS A LA SOURIS

A partir de l'exemple précédent, imaginons que nous souhaitons désormais déplacer le rectangle à l'aide du curseur de notre souris.

Pour ce faire, nous allons utiliser le type d'évènement « `MOUSEMOTION` » qui se déclenche dès que le joueur déplace le curseur de la souris dans la fenêtre de jeu. Voici le code modifié :

```
import pygame

pygame.init()
pygame.display.set_caption("Les évènements !")

black_color = (0, 0, 0)
red_color = (255, 80, 80)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)

monRectangle = pygame.Rect(235, 235, 30, 30)
pygame.draw.rect(surface, red_color, monRectangle)

pygame.display.flip()

running = True
while running: # main game loop
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.MOUSEMOTION:
            monRectangle.x = event.pos[0]
            monRectangle.y = event.pos[1]

            surface.fill(black_color)
            pygame.draw.rect(surface, red_color, monRectangle)

            pygame.display.flip()
```

Ici on se sert de l'attribut « `pos` » qui renvoie un tuple de nombre entier qui représente les coordonnées en `x` et en `y` du curseur de la souris. On modifie donc les coordonnées de notre rectangle dynamiquement à partir de ce tuple.

LES SPRITES

En termes de programmation, un Sprite est une représentation en 2D d'un élément à l'écran. Pygame fournit une classe Sprite, conçue pour contenir une ou plusieurs représentations graphiques de tout objet de jeu que vous souhaitez afficher à l'écran. Pour l'utiliser, vous devez créer une nouvelle classe qui étend Sprite. Cela vous permet d'utiliser ses méthodes intégrées.

DÉFINIR UN JOUEUR BASIQUE

Pour définir une classe « Player » basique, vous pouvez hériter de la classe « pygame.sprite.Sprite » :

```
class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()

joueur = Player()
```

Ensuite, le constructeur « __init__() » utilise la méthode « super() » pour appeler le constructeur « __init__() » de la classe parente, « Sprite ».

Ensute, vous définissez et initialisez une variable « surf » de type « Surface » pour contenir l'image à afficher, qui est actuellement une boîte blanche (vous pouvez évidemment charger une image comme vu précédemment dans ce cours !). Vous définissez et initialisez également « rect », que vous utiliserez plus tard pour dessiner le joueur.

Pour utiliser cette nouvelle classe, vous devez évidemment l'instancier en créant un nouvel objet que l'on appellera « joueur » !

Nous allons maintenant voir comment faire bouger notre joueur. Pour ce faire, nous pouvons utiliser une fonction bien utile du module « key » appelée « get_pressed() ». Cette fonction renvoie l'ensemble des touches sur lesquelles le joueur a appuyé depuis le dernier appel de cet fonction sous la forme d'un **dictionnaire** :

```
# On récupère les touches appuyées en début de frame
pressed_keys = pygame.key.get_pressed()
```

Cette fonction doit être appelée dans la Game Loop !

Nous allons maintenant définir une méthode « move » dans notre classe « Player » qui va nous permettre de déplacer le personnage en fonction de ce que l'on récupère du dictionnaire précédent :

```
def update(self, pressed_keys):  
    # Permet de déplacer le joueur avec les flèches du clavier  
  
    if pressed_keys[pygame.K_UP]:  
        self.rect.move_ip(0, -1)  
  
    if pressed_keys[pygame.K_DOWN]:  
        self.rect.move_ip(0, 1)  
  
    if pressed_keys[pygame.K_LEFT]:  
        self.rect.move_ip(-1, 0)  
  
    if pressed_keys[pygame.K_RIGHT]:  
        self.rect.move_ip(1, 0)  
  
    # Garde le joueur à l'écran  
  
    if self.rect.left < 0:  
        self.rect.left = 0  
  
    if self.rect.right > surface_res[0]:  
        self.rect.right = surface_res[0]  
  
    if self.rect.top <= 0:  
        self.rect.top = 0  
  
    if self.rect.bottom >= surface_res[1]:  
        self.rect.bottom = surface_res[1]
```

Comme vu précédemment, un tel code nous permettra de bouger le joueur grâce aux flèches du clavier. La deuxième partie du code de cette méthode permet simplement de vérifier que le joueur ne sorte pas de l'écran !

Il nous faut donc passer tout cela dans notre Game Loop :

```
running = True  
while running: # main game loop  
    for event in pygame.event.get():  
        if event.type == pygame.QUIT:  
            running = False  
  
    # On efface la surface  
    surface.fill((0, 0, 0))  
  
    # On récupère les touches appuyées en début de frame  
    pressed_keys = pygame.key.get_pressed()  
  
    # On update le Sprite du joueur en fonction des touches  
    joueur.update(pressed_keys)  
  
    # On dessine le joueur à l'écran  
    surface.blit(joueur.surf, joueur.rect)  
  
    # On rafraîchit la surface  
    pygame.display.flip()
```

LES « SPRITES GROUP »

Une autre classe très utile fournie par Pygame est le groupe de Sprites. C'est un objet qui contient un groupe d'objets Sprite. Cet objet possède des méthodes très spécifiques qui peuvent être utiles pour manipuler un ensemble de Sprites.

Voyons comment créer un groupe de sprites :

```
all_sprites = pygame.sprite.Group()  
all_sprites.add(joueur)
```

La méthode « add(sprite) » de la classe « Sprites Group » permet d'ajouter des sprites à ce groupe. Pour retirer un Sprite du groupe, il suffit d'appeler la méthode « remove(sprite) » :

```
all_sprites.remove(joueur)
```

Lorsque vous appelez la méthode « kill() » de la classe « Sprite », le Sprite en question est retiré de chaque Groupe auquel il appartient. Cela supprime également les références au Sprite, ce qui permet au ramasse miettes de Python de récupérer la mémoire si nécessaire :

```
joueur.kill()
```

Maintenant que vous avez un groupe « all_sprites », vous pouvez modifier la façon dont les objets sont dessinés. Au lieu d'appeler la méthode « blit() » sur chaque élément que vous voulez dessiner, un à un, vous pouvez itérer sur tout ce qui se trouve dans « all_sprites » dans la Game Loop :

```
# On efface la surface  
surface.fill((0, 0, 0))  
  
# On parcours notre groupe pour dessiner les Sprites  
for entity in all_sprites:  
    surface.blit(entity.surf, entity.rect)
```

Maintenant, tout ce qui est mis dans « all_sprites » sera dessiné à chaque frame !

Comme raccourci, le groupe possède également une méthode « update() », qui appellera une méthode « update() » sur chaque sprite du groupe en passant les mêmes arguments à chacun d'eux.

Enfin, le groupe possède quelques autres méthodes qui vous permettent de l'utiliser avec la fonction intégrée « len() », pour obtenir le nombre de sprites qu'il contient

DÉTECTER LES COLLISIONS

La vérification des collisions est une technique de base de la programmation de jeux, et nécessite généralement des calculs assez complexes pour déterminer si deux sprites vont se chevaucher.

Heureusement, comme nous l'avons vu dans le chapitre sur les objets de type « Rect », Pygame dispose de méthodes de détection de collision que vous pouvez utiliser !

Pour détecter si des sprites sont entrés en collision entre eux, nous allons utiliser la méthode « `spritecollideany(sprite, group)` ». Cette méthode prend en paramètre un premier argument de type « `Sprite` » et un deuxième argument de type « `Sprites Group` » :

```
pygame.sprite.spritecollideany(joueur, all_sprites)
```

Cette fonction renvoie un booléen qui détermine si le Sprite en question a touché n'importe quel autre Sprite appartenant au groupe passé en paramètre. Cela est parfait pour, par exemple, vérifier si votre joueur a collisionné des ennemis que vous auriez ajouter à un groupe spécifique !

MESURER LE TEMPS

Dans Pygame, mesurer et contrôler le temps est essentiel pour, par exemple, gérer le taux de rafraîchissement des différents sprites de votre jeu afin d'avoir des animations qui soient synchronisées.

Bien que vous ayez déjà vu le module « `Time` » de Python pour manipuler le temps, Pygame possède son propre module « `time` » spécialement conçu pour la création de jeu.

La première fonction de ce module que l'on va voir est « `get_ticks()` » qui renvoie le nombre de millisecondes qui se sont écoulées depuis que le « `pygame.init()` » a été appelé :

```
print(pygame.time.get_ticks())
```

Dans la console, nous verrons donc :

```
pygame 2.1.2 (SDL 2.0.18, Python 3.8.2)
Hello from the pygame community. https://www.pygame.org/contribute.html
443
```

443 étant ici le nombre de milliseconde qui se sont écoulée entre l'appel de « `pygame.init()` » et l'appel de la fonction « `get_ticks()` ».

METTRE LE PROCESSUS EN PAUSE

Une manière d'arrêter le temps dans votre jeu est de placer le processus créé par votre application en pause au niveau de votre OS. Cette fonction met donc le processus en sommeil afin de partager le processeur avec d'autres programmes. Cette méthode est peu gourmande en ressource puisque votre machine n'allouera que peu ressource à votre application pendant ce temps de pause.

Pour ce faire, nous pouvons utiliser la fonction « `wait(time)` » du module « `time` ». L'argument à passer à cette fonction est le temps, en millisecondes, que votre application doit être mise en pause.

Par exemple, le code suivant va nous permettre de passer un texte qui était initialement en blanc à une couleur rouge après 3 secondes d'attente :

```
import pygame

pygame.init()
pygame.display.set_caption("Mesurer le temps")

white_color = (255, 255, 255)
black_color = (0, 0, 0)
red_color = (255, 80, 80)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res, pygame.RESIZABLE)

myFont = pygame.font.Font("fonts/Nexa-Trial-Regular.ttf", 40)

myText = myFont.render("Hello World", True, white_color)
surface.blit(myText, [20, 20])
pygame.display.flip()

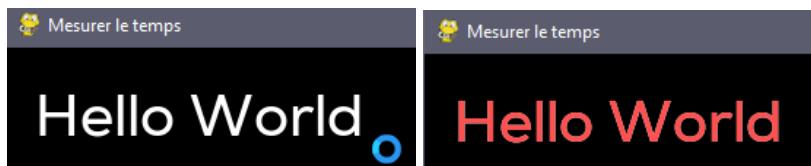
pygame.time.wait(3000)

myText = myFont.render("Hello World", True, red_color)
surface.blit(myText, [20, 20])
pygame.display.flip()

running = True
while running: # main game loop
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            running = False
```

Voici le résultat produit :



Vous remarquerez que pendant que le texte reste en blanc, votre curseur affiche une icône de chargement. Cela signifie que votre application ne répond plus à aucune de vos actions et que le processus lié à votre fenêtre de jeu est figé (plus aucun code ne s'exécute au sein de votre programme).

De plus, ce temps de pause peut parfois être imprécis !

A cause de cela, cette méthode n'est pas toujours adéquate.

AJOUTER UN DÉLAI DANS LE CODE

Si vous ne souhaitez pas figer l'entièreté de votre processus mais simplement ajouter un délai, vous pouvez utiliser la fonction « `delay(time)` » du module « `time` ».

Modifions l'exemple précédent en remplaçant la fonction « `wait` » par « `delay` » :

```
myText = myFont.render("Hello World", True, white_color)
surface.blit(myText, [20, 20])
pygame.display.flip()

pygame.time.delay(3000)

myText = myFont.render("Hello World", True, red_color)
surface.blit(myText, [20, 20])
pygame.display.flip()
```

Lors de l'exécution, vous remarquerez que le résultat final est similaire mais que votre curseur de souris n'affiche plus l'icône de chargement. Dans ce cas de figure, le processus lié à votre jeu n'est plus figé !

Cette méthode est évidemment plus gourmande en ressource que la précédente. En effet, cette fonction utilisera le processeur (plutôt que de placer le processus en sommeil pour partager les ressources CPU) afin de rendre le délai plus précis que la fonction « `wait` ».

LES FRAMES PER SECONDS

Pour mesurer le temps dans nos jeux, nous allons instancier un objet de la classe « `Clock` ». Cet objet va nous permettre de fixer le taux de rafraîchissement de notre application. Pour instancier un objet de la classe « `Clock` » :

```
pygame.init()
clock = pygame.time.Clock() # instantiation de l'objet Clock
```

Si vous jouez déjà un peu aux jeux-vidéo, vous devriez déjà être familier avec la notion de FPS (Frames Per Seconds). La fréquence d'images ou fréquence de rafraîchissement est le nombre d'images que le programme dessine par seconde et est mesurée en FPS ou images par seconde. (Sur les moniteurs, le nom commun pour FPS est « hertz »).

Une fréquence d'images faible dans jeux vidéo peut donner l'impression que le jeu est saccadé ou qu'il « lag ». En effet, si le programme a trop de code à exécuter pour dessiner à l'écran, alors le taux de rafraîchissement diminue et le joueur expérimente alors ce qu'on appelle une « chute de FPS ».

L'objet de type « `Clock` » assurera que notre programme fonctionne à un certain nombre de FPS grâce à sa méthode « `tick(fps)` ». Cette méthode veillera à ce que nos programmes de jeu ne tournent pas trop vite en mettant en place des petites pauses à chaque itération de la Game Loop et ce, quelle que soit la vitesse de l'ordinateur sur lequel notre jeu fonctionne.

Cette méthode doit d'ailleurs être appelée à la fin de chaque itération de la Game Loop :

```
running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    clock.tick(60) # 30 images par secondes
```

L'argument passé à la méthode « tick(fps) » établit la fréquence d'images désirée. Pour ce faire, elle calcule le nombre de millisecondes que chaque image doit prendre, en fonction de la fréquence d'images souhaitée.

Ensuite, elle compare ce nombre au nombre de millisecondes qui se sont écoulées depuis la dernière fois que « tick(fps) » a été appelée. S'il ne s'est pas écoulé suffisamment de temps, alors on retarde le traitement pour s'assurer qu'on ne dépasse jamais la fréquence d'images spécifiée.

Si nous n'avions pas ces petites pauses, notre programme de jeu programme de jeu s'exécuterait aussi vite que l'ordinateur peut le faire (souvent, les jeux s'exécuteraient à des vitesses trop élevées pour le joueur). De plus, puisque les ordinateurs deviennent de plus en plus rapides, de nouvelles machines exécuteraient le jeu plus rapidement que des machines moins performantes.

Dans cet exemple, nous allons créer un code qui va nous permettre d'afficher un compteur de FPS. Cela est rendu possible grâce à la méthode « get_fps() » de la classe « Clock » qui renvoie le nombre de FPS actuel du jeu :

```
import pygame

pygame.init()

clock = pygame.time.Clock() # instantiation de l'objet Clock

pygame.display.set_caption("Mesurer le temps")

white_color = (255, 255, 255)
black_color = (0, 0, 0)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res)

myFont = pygame.font.Font("fonts/Nexa-Trial-Regular.ttf", 40)

running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    surface.fill(black_color)
    myText = myFont.render("{fps:.0f} FPS".format(fps = clock.get_fps()), True,
white_color)
    surface.blit(myText, [20, 20])
    pygame.display.flip()
    clock.tick(60) # 60 images par secondes
```

Résultat :



METTRE A JOUR SON AFFICHAGE VIA UN « USEREVENT »

Le module « time » de Pygame nous permet également de créer des « timers » via la fonction « `set_timer(event, millis)` ». Cette fonction va permettre de créer un évènement de façon répétée et de la placer dans la file d'évènement après un certain délais en millisecondes (renseigné par l'argument « `millis` ») :

```
pygame.time.set_timer(pygame.USEREVENT, 2000)
```

Dans le code ci-dessus, un event de type « `USEREVENT` » (c'est un à dire un type d'évènement personnalisé) va se placer dans la file d'attente toutes les 2 secondes !

On va se servir de ce « `USERVENT` » pour mettre à jour notre affichage de façon régulière. Cette méthode est utile pour déclencher du code de manière répétée tout en s'assurant qu'il ne s'exécute pas trop souvent.

C'est une approche plus performante, car si on place simplement le code dans la Game Loop, celui-ci se déclenche à chaque frame (dans notre exemple cela veut dire 60 fois par seconde !) :

```
import pygame

pygame.init()
clock = pygame.time.Clock() # instantiation de l'objet Clock

pygame.time.set_timer(pygame.USEREVENT, 1000)

pygame.display.set_caption("Mesurer le temps")

white_color = (255, 255, 255)
black_color = (0, 0, 0)

surface_res = (500, 500)
surface = pygame.display.set_mode(surface_res)

myFont = pygame.font.Font("fonts/Nexa-Trial-Regular.ttf", 40)

myText = myFont.render("{fps:.0f} FPS".format(fps = clock.get_fps()), True,
white_color)
surface.blit(myText, [20, 20])
pygame.display.flip()

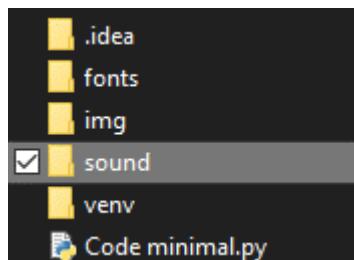
running = True
while running: # main game loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.USEREVENT:
            surface.fill(black_color)
            myText = myFont.render("{fps:.0f} FPS".format(fps =
clock.get_fps()), True, white_color)
            surface.blit(myText, [20, 20])
            pygame.display.flip()

    clock.tick(60) # 30 images par secondes
```

JOUER DU SON

Pour jouer du son dans vos applications Pygame, il vous faudra charger un fichier audio. Avant de voir les fonctions qui permettent de charger ce type de fichier au sein de votre programme, nous allons d'abord créer un dossier « sound » à la racine de votre projet Pycharm :



Vous placerez dans ce dossier l'ensemble des sons que vous souhaitez jouer dans votre programme.

Une fois ceci fait, nous allons utiliser le module « mixer » de la bibliothèque Pygame pour instancier un objet de type « Sound » :

```
mySong = pygame.mixer.Sound("sound/jdg-theme.mp3")
```

Le constructeur de la classe « Sound » prend en paramètre le chemin vers le fichier audio à charger.

JOUER UN FICHIER AUDIO

Pour jouer le son ainsi chargé, il nous suffit d'appeler la méthode « play(loop, time, fadein) » de la classe « Sound ». L'ensemble des arguments de cette fonction sont optionnels. Nous pouvons donc très bien jouer le son comme ceci :

```
mySong = pygame.mixer.Sound("sound/jdg-theme.mp3")
mySong.play()
```

Cependant, il peut être intéressant de jouer avec les arguments de cette fonction. Le premier paramètre « loop » est un nombre entier qui détermine le nombre de fois que le morceau doit être répété.

Ensuite, « time » permet de définir une limite en milliseconde de jusqu'où le fichier audio doit être lu (par exemple, si « time » prend 10000 comme valeur, on arrêtera de jouer le son après 10 secondes).

Enfin, « fadein » permet de créer un effet de transition d'ouverture et prend un nombre entier comme valeur. Par exemple, une valeur de 3000 signifiera que le son passera de 0% à 100% de volume sonore en 3 secondes.

```
mySong.play(0, 15150, 6000)
```

STOPPER UN FICHIER AUDIO + FAEDOUT

Pour stopper un fichier audio, vous pouvez appeler la méthode « `stop()` » de la classe « `Sound` » :

```
mySong.stop()
```

Si vous voulez définir un effet de transition de fermeture, vous pouvez utiliser la méthode « `fadeout(millis)` » de la classe « `Sound` » :

```
mySong.fadeout(3000)
```

Attention, une fois cette méthode appelée, le son sera entièrement coupé après le délai que vous passez en paramètre de la méthode (ici 3 secondes).

RÉCUPÉRER ET MODIFIER LE VOLUME SONORE

Si vous voulez modifiez l'intensité du volume sonore de votre fichier audio vous pouvez utiliser la méthode « `set_volume(value)` » où le paramètre « `value` » est un nombre flottant compris entre 0 et 1 (0 et 100% d'intensité) :

```
mySong.set_volume(0.5)
```

Dans cet exemple j'ai placé le volume de notre musique à 50% de son intensité normale.

Evidemment, nous pouvons aussi récupérer le volume actuel du fichier audio via la méthode « `get_volume()` » :

```
mySong.get_volume()
```

RÉCUPÉRER LA DURÉE DU FICHIER

Pour récupérer la durée de votre fichier audio en seconde, vous pouvez appeler la méthode « `get_length()` » :

```
mySong.get_length()
```

GÉRER L'ENSEMBLE DES FICHIERS AUDIO

Le module « `mixer` » possède une fonction « `stop()` » qui nous permet d'arrêter l'ensemble des fichiers audio qui sont en train d'être lus :

```
pygame.mixer.stop()
```