# Préparation projet 4

# Apprenez la programmation orientée objet avec Python

## Comprenez la programmation orientée objet

### Classe

Une classe contient un **état** et un **comportement**.

L’**état** fait référence à des données ou des variables. Une classe  Cake  (Gâteau), par exemple, peut avoir une variable  flavor  (saveur).

Le **comportement** d’une classe désigne simplement une série de choses que cette classe peut faire. Ce comportement est contenu dans les **méthodes**, qui sont identiques conceptuellement aux fonctions. Toujours pour reprendre notre exemple du gâteau, nous pourrions définir un comportement spécifique aux anniversaires (ajouter des bougies).

Une image contenant texte, capture d’écran, logo, nourriture

Description générée automatiquement

#### Alors, quelle est la différence entre une méthode et une fonction ?

Tout simplement, une méthode fait partie d’une classe, alors qu’une fonction est indépendante des classes. Les méthodes peuvent prendre des paramètres, modifier l’état interne de l’objet, appeler d’autres méthodes ou fonctions, et retourner des valeurs.

|  |
| --- |
| Client |
| **État (Variables)**  name  phone\_number  bank\_account |
| **Comportement (Méthodes)**  bill\_customer(cost)  refund\_customer(amount) |

**POO**: Techniquement, il s’agit de manipuler des sortes de conteneurs (les classes), qui stockent à la fois des valeurs brutes (appelées « propriétés ») et des fonctions (appelées « méthodes »).

|  |  |  |
| --- | --- | --- |
| **Boite à outils** | **Marteau** | **Tournevis** |
| **Variables**  tools | **Variables**  Color | **Variables**  size |
| **Méthode**  Add\_tool(tool)  Remove\_tool(tool) | **Méthodes**  Paint(color)  hammer\_in(nail)  remove(nail) | **Méthodes**  tighten(screw)  loosen(screw) |

### En résumé

* Les classes sont des **modèles** pour les objets.
* Les classes allient **état** (les données ou variables) et **comportement** (les méthodes) s’appliquant à ces données.
* Un objet peut être créé et utilisé dans un programme.

## Écrivez une classe Python

Les noms de classe, par convention, commencent par une majuscule. S’il y a plusieurs mots dans le nom d’une classe, on utilise généralement le CapitalizedCase, en mettant une majuscule à la première lettre de chaque mot.

class Rectangle:

width = 3

height = 2

def calculate\_area(self):

return self.width \* self.height

Note : en programmation orientée objet, il existe trois types d’attributs :

* les attributs **d’instance** (propres aux instances créées),
* les attributs **de classe** (propres à la classe, et partagés entre les instances),
* et les attributs **statiques** (qui sont presque indépendants de la classe).

Si chaque type d’attribut possède une utilité propre, essayez autant que possible de privilégier les attributs d’instance, qui permettent d’utiliser la programmation orientée objet à son plein potentiel.

### Déclarez des variables dynamiques avec le constructeur \_\_init\_\_

Il existe une catégorie spéciale de méthode nommée **constructeur**. Chaque classe en a un, et il est utilisé pour créer des objets à partir de cette classe. En Python, notre constructeur est une méthode magique nommée  \_\_init\_\_, que l’on peut utiliser un peu comme ceci :

De plus, les attributs définis dans la méthode magique \_\_init\_\_ ne seront pas partagés par les autres instances. Ce sont des **attributs d’instance**, qui seront propres à chaque objet créé.

class Rectangle:

def \_\_init\_\_(self, lenght, width, color=’red’):

self.width = width

self.height = height

self.color = color

#### Une méthode « magique » ?

Eh oui, elle est dite « magique » parce qu’elle modifie le comportement interne de la classe. En Python, il existe plusieurs méthodes magiques, qui commencent et se terminent par deux underscores (ou « *dunders* »).

### En résumé

* Créez une classe en utilisant le mot-clé  class.
* La méthode magique  \_\_init\_\_  est connue sous le nom de constructeur.
* Les constructeurs sont appelés lorsqu’un objet est créé.

## Créez et utilisez des objets Python

Il est important que les paramètres que vous fournissez correspondent aux paramètres du constructeur. Ceci inclut les positions des paramètres – dans notre constructeur ci-dessus, la « longueur » (length) est le premier paramètre donné, et la « largeur » (width) est le deuxième paramètre donné.

Notez que vous pouvez aussi nommer les paramètres obligatoires

( rectangle = Rectangle(length=2, width=3, color="white")). Le nommage des paramètres peut être une grande aide à la lecture du code, pensez-y ! 🧐

Le grand intérêt des objets est que leurs valeurs sont stockées en eux. Vous pouvez donc passer le rectangle de fonction en fonction en modifiant ses attributs, il gardera en mémoire ses modifications sur le temps.

Le « point » signifie simplement que vous accédez à un attribut. rectangle.lengthpeut se lire comme « length of rectangle » (« longueur du rectangle »).

Attention, assurez-vous que vous modifiez ou accédez à un attribut réellement présent sur l’objet ! Sinon, vous obtiendrez une erreurAttributeError, et votre programme va planter (nous y reviendrons plus tard).

En fait, lors de l’appel d’une méthode d’instance, Python passe l’instance  rectangle  en tant que premier argument “*self*” à la classe  Rectangle, sans crier garde ! Faire rectangle.calculate\_area() revient à faire Rectangle.calculate\_area(rectangle)! Vous avouerez que la première version est plus élégante et compréhensible.

#### Découvrez les différents attributs

Vous serez amené à lire trois types d’attributs de classe dans votre pratique du Python :

* les attributs d’instance ;
* les attributs de classe ;
* les attributs statiques.

Les **attributs d’instance** sont des variables définies à l’aide deself. Elles sont relatives à l’instance, et ne peuvent être accédées sans instanciation. Dans le cadre des méthodes, ce sont les méthodes classiques d'une classe, qui possèdent  self  **en premier paramètre**.

Les **attributs de classe** sont des variables définies directement dans le **corps de la classe**. Elles peuvent être accédées par la classe, sans passer par l’instanciation. Les attributs de classe peuvent se **référencer entre eux**, mais ne peuvent pas accéder aux attributs d’instance.

Dans le cadre des méthodes, elles seront précédées par un  @classmethod, et leur premier paramètre seracls(à la place deself) :

Les attributs de classe sont souvent utilisés pour créer des données ou des actions globales à la classe, qui **ne dépendent pas** d’une instance. Les instances peuvent cependant y accéder. Modifier un attribut de classe dans une instance le modifiera dans **toutes les autres**!

 Enfin, les **attributs statiques** sont des attributs qui n’ont pratiquement **aucun lien** avec la classe. Seules les **méthodes** peuvent être **statiques**, et l’ajout par rapport aux attributs de classe est **minime**: on n'a plus besoin de spécifier le paramètre  cls. Pour créer un attribut statique, il suffit de faire précéder la méthode par  @staticmethod :

class Bird:

"""Un oiseau. 🐦"""

@staticmethod

def get\_definition():

"""Donne la définition d'un oiseau."""

return (

"Animal (vertébré à sang chaud) au corps recouvert de plumes, "

"dont les membres antérieurs sont des ailes et qui a un bec."

)

print(Bird.get\_definition())

Je vous recommande de privilégier les **attributs d’instance** qui permettent d’utiliser la programmation orientée objet à son **plein potentiel**. Les attributs de classe peuvent être **très utiles**, mais sont censés être **plus rares**. Enfin, les attributs statiques sont à **l’opposé** du paradigme orienté objet : évitez de les utiliser, quitte à utiliser un attribut de classe à la place. 😉

### En résumé

* On peut instancier des objets en utilisant le nom de leur classe.
* Lorsqu’on instancie un objet, on appelle son constructeur, et il faut lui fournir les bons paramètres.
* Nous pouvons accéder aux attributs, les modifier, et appeler des méthodes en utilisant la syntaxe  objet.attribut.

*Toutes les classes ont un constructeur sans arguments, qui peut être utilisé même sans en écrire un sur mesure.*

## Appliquez l’héritage dans votre code Python

Une image contenant texte, capture d’écran, Police, logo

Description générée automatiquement

L’héritage vous permet de créer une **sous-classe** (ou classe **enfant**) d’une classe **parent**, qui contient les attributs du parent ainsi que d’autres attributs spécifiques à l’enfant.

Une image contenant texte, capture d’écran, Police

Description générée automatiquementL’héritage permet également l’**extensibilité** – c’est-à-dire la possibilité d’étendre la fonctionnalité d’un programme *sans* avoir à modifier le code existant.

Imaginons que vous proposiez deux options pour contacter vos utilisateurs – par SMS ou par hibou 🧙‍♀️ :

Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement

imaginons que nous voulions étendre la fonctionnalité de notre système en ajoutant un nouveau moyen de contacter nos utilisateurs : par la poste. Si nous créons une sous-classe nommée  PostContactSystem  et que nous l’implémentons, nous n’avons *rien* à changer dans notre classe Utilisateur, ni dans aucune partie du code impliquée dans l’envoi de messages !

#### Les modèles

Enfin, l’un des avantages de l’héritage concerne la façon dont nous concevons les systèmes. L’héritage, et les classes en général, nous permettent d’**exprimer des relations** entre différentes parties de notre code.

Tout comme nous regroupons des données et du code au sein d’un paquet en utilisant les classes, nous relions les classes qui sont conceptuellement apparentées grâce à l’héritage. Nous entrerons plus en détail dans les **hiérarchies d’héritage** et les multiples niveaux d’héritage au chapitre 4.

Les classes et l’héritage sont utiles, car ils rendent les systèmes plus faciles à comprendre. Cela permet aux développeurs et aux équipes de développement de construire les **modèles conceptuels** des comportements et des données dans le système – ce qui prend encore plus d’importance lors de la maintenance et de l’extension de projets vastes et complexes.

### En résumé

* Une classe peut avoir une **classe parent**, **classe mère** ou **superclasse**.
* La classe qui a un parent est nommée **sous-classe**, **classe fille** ou **classe enfant**.
* Cette classe **hérite d’attributs** de son parent.
* La classe enfant peut implémenter des **attributs supplémentaires**, en plus de ceux dont elle hérite.

## Écrivez une sous-classe en Python

### Définissez la syntaxe de l’héritage

L’élément principal de syntaxe utilisé pour définir les sous-classes se trouve dans la définition de classe elle-même : class  FilmCassette(Film). Le parent de la classe est placé entre parenthèses après le nom de la classe dans la toute première ligne de la classe !

En cas d’**héritage multiple** – sujet que nous aborderons brièvement plus loin, au chapitre 4 – les classes parents sont séparées par des parenthèses, de façon similaire à celle dont fonctionnent les paramètres de fonctions, comme ceci :  class SousClasse(ParentUn,ParentDeux, ...). L’héritage multiple est à utiliser avec précaution – pour la plupart des problèmes, il existe généralement des designs plus simples et plus faciles.

Si vous ne spécifiez pas de classe parent (ce que nous avions fait dans l’exercice de la partie précédente), vous dites en fait à Python de faire de la classe une sous-classe d’Object.  Object  constitue la base de toutes les classes, ce qui peut prêter à confusion. Vous vous souvenez que toutes les classes possèdent un constructeur sans argument par défaut ? C’est parce qu’elles héritent d’Object.

### En résumé

* On définit une classe enfant en utilisant  class Enfant(Parent).
* Par défaut, toutes les classes héritent **d’Objet**– un objet Python qui fournit une fonctionnalité basique.
* Les classes peuvent hériter de classes parents multiples simultanément – dans le cas de l’**héritage multiple**.

### Surchargez les méthodes en Python

La **surcharge** désigne le concept selon lequel une classe enfant crée une **nouvelle implémentation** d’une **méthode héritée**. Lorsqu’une méthode dans une classe enfant est créée avec le même nom et la même signature qu’une méthode dans le parent, la méthode enfant l’emporte.

La signature d’une méthode, c’est son nom et sa liste de paramètres.  watch(),  add(a, b), ou encore  send(message)  sont toutes des signatures de méthode.

En réalité, nous avons utilisé le concept de surcharge une fois ou deux dans les deux chapitres précédents, par exemple avec le  ContactSystem

### Utilisez des classes abstraites

Bon, je comprends que  area()  est surchargé, mais qu’est-ce que c’est que cet  ABC  ?

ABC signifie « Abstract Base Class », ou « classe de base abstraite ». Il s’agit du mécanisme utilisé par Python pour implémenter ce que l’on nomme une **classe abstraite**.

Une classe abstraite est une classe **qui ne peut pas être instanciée**– la seule façon de l’utiliser est de créer une sous-classe. Par exemple, prenons le cas de  Shape  ("Forme") – qu’est-ce que cela voudrait dire d’avoir un objet  Shape ? À quoi est-ce qu’il pourrait bien ressembler ? Nous pourrions faire de  Square, ou Carré, une sous-classe de  Shape, puis utiliser notre Carré. Cette approche nous permettrait d’avoir une classe parent commune pour Cercle, Carré, Triangle, etc., sans laisser personne instancier une  Shape.

Une classe abstraite peut aussi insister pour qu’une méthode soit implémentée par ses enfants. @abstractmethod : **Forçage de l'implémentation** : L'utilisation de méthodes abstraites garantit que les sous-classes doivent fournir une implémentation pour ces méthodes. Si une sous-classe ne fournit pas d'implémentation pour une méthode abstraite, une erreur sera levée au moment de l'instanciation de cette sous-classe.

### Accédez aux méthodes des parents

Bien que la surcharge nous permette de modifier entièrement des comportements hérités, il peut parfois être utile d’avoir accès au code des méthodes des classes parents, depuis les classes enfants.

Une image contenant texte, logiciel, Icône d’ordinateur, Logiciel multimédia

Description générée automatiquementL’un des emplacements les plus courants pour cette utilisation se trouve dans les constructeurs. Pour cela, nous utilisons**super()**– comme ceci :

L’approche avec  super()  vous permet de réutiliser le code plutôt que de le copier, et assure le regroupement des fonctionnalités de façon logique – soit **deux des plus grands avantages** de la programmation orientée objet !

### En résumé

* Une classe enfant peut fournir **sa propre implémentation** d’un élément hérité de sa classe parent.
* L’implémentation de la classe enfant est prioritaire sur celle du parent – elle **surcharge** l’implémentation de la classe parent.
* On peut utiliser la méthode **super** pour accéder à des méthodes dans la classe parent que nous avons surchargée.
* Une **classe abstraite** est une classe qui **ne peut pas être instanciée** – à la place, il faut en hériter.

### Utilisez l'héritage multiple

En quoi l’héritage multiple est-il différent d’une hiérarchie d’héritage ?

L’**héritage multiple** suppose qu’une classe ait de **multiples classes parents**. Dans une hiérarchie d’héritage, il y a plusieurs niveaux d’héritage – une classe **a un parent qui a un parent**.

L’héritage multiple a **mauvaise réputation** en programmation orientée objet – les systèmes qui utilisent l’héritage multiple peuvent être difficiles à comprendre. De plus, certains langages de programmation proposent des implémentations médiocres d’héritages multiples qui provoquent des problèmes.

class Cat:

"""Un chat."""

def meow(self):

"""Miaule."""

print("Meow!")

class Talker:

"""Interface qui définit la méthode "say" (dire)."""

def say(self, to\_say):

"""Affiche "to\_say" (à dire)."""

print(to\_say)

class TalkingCat(Cat, Talker):

"""Un chat qui parle ??"""

pass

salem = TalkingCat()

salem.meow()

salem.say("Hello!")

Par conséquent, unTalkingCat, comme notre Salem ici, a la capacité de miauler et de parler !

Le modèle que nous avons utilisé ici est souvent nommé un **Mixin** – nommé ainsi car vous ajoutez (« mix in ») une fonctionnalité nécessaire.

**Attendez, que se passe-t-il si deux classes parents implémentent la même méthode ? Laquelle va gagner ?**

C’est là l’un des problèmes principaux de l’héritage multiple – on l’appelle le problème du diamant (ou le « Deadly Diamond of Death », soit littéralement le « diamant de la mort qui tue » 💀). Chaque langage qui utilise l’héritage multiple a sa propre solution – celle de Python s’appelle le MRO, pour **Method Resolution Order**: l’**ordre de résolution de méthode**.

Expliqué simplement, le MRO d’une classe constitue l’ordre des emplacements où Python va chercher une définition de méthode. Ceci peut devenir assez compliqué – si vous souhaitez obtenir davantage d’informations à ce sujet, vous les trouverez dans la documentation Python.

De façon générale, néanmoins, Python cherchera **dans la classe parent la plus à gauche en premier**. Donc, dans notre exemple  TalkingCat  ci-dessus, Python cherchera d’abord dans Cat, puis dans Talker.

### En résumé

* Une classe parent peut elle-même avoir un parent.
* Lorsque nous avons de multiples niveaux d’héritage, nous avons une **hiérarchie d’héritage**.
* Une classe peut aussi hériter de multiples classes parents dans le cadre de ce que l’on nomme l’**héritage multiple**.

## Utilisez des objets dans des collections

Lorsque nous programmons, nous stockons fréquemment les données dans une **collection**. Les collections comprennent des **listes**– où les données ont une position et sont **indexables** – et des**dictionnaires** – où on attribue une **clé** aux données.

Vous vous en souvenez peut-être, on peut placer tous types de données – y compris un mélange de types – dans une liste ou un dictionnaire :

Une image contenant texte, logiciel, Site web, Page web

Description générée automatiquement

Les dictionnaires se doivent d’avoir des clés **immutables** (“immuables”). Les valeurs immuables sont par exemple des nombres et des chaînes de caractères – lorsque la valeur ne peut pas être changée après sa création. À part dans des cas très spéciaux, toutes les classes que vous écrirez seront **mutables** (“muables”) – elles peuvent être modifiées – et donc elles ne peuvent pas constituer des clés dans un dictionnaire. Créer des objets immuables est **possible**, mais c’est un peu complexe, et leur utilité est plus rare.

les objets de n’importe quelle classe sont comme n’importe quel autre élément ! Ils peuvent eux aussi être stockés dans des collections, un peu comme ici ; disons qu’il s’agit de notre liste de bénévoles pour une collecte de nourriture locale :

Une image contenant texte, logiciel, Icône d’ordinateur, Logiciel multimédia

Description générée automatiquement

### Utilisez le duck typing en toute sécurité

Python utilise ce que l’on appelle le **duck typing** (littéralement, typage de canard) : si ça a un bec et que ça cancane comme un canard, alors c’est probablement un canard. Autrement dit, les méthodes ou attributs d’un objet **sont plus importants** que son type ou sa classe.

Prenons un exemple. Et si nos bénévoles étaient un peu différents cette semaine ?

Une image contenant texte, logiciel, Logiciel multimédia, Icône d’ordinateur

Description générée automatiquement

Les poissons ne peuvent pas marcher, donc si nous exécutons ce code, nous obtiendrons ce genre de chose : AttributeError: ‘Fish’ object has no attribute ‘walk’

Alors, comment nous assurer que cela ne se produise pas ?

#### Documentez votre code

Vous pouvez documenter votre code à l’aide de **docstrings.** elles sont une forme de **documentation** réelle pour votre code (elles peuvent se récupérer par votre éditeur de code, ou générer des documentations sous forme de site, via des bibliothèques spécialisées pour).

#### Typez votre code

Python 3 vous fournit une bibliothèque detypes, qui vous permet d’écrire du code avec des annotations de type, de façon similaire à d’autres langages. Cette possibilité est de plus en plus utilisée – suivez le guide de styles du projet (nous y reviendrons dans la partie suivante) pour savoir si vous l’utiliserez ou non.

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Description générée automatiquement

Cette fonction retourne l’entier (int) le plus élevé d’une liste d’entiers. La signature de cette fonction dit que les nombres doivent constituer une  List[int]  – une liste d’entiers – et le **type de retour** de cette fonction est int.

Le typage peut être extrêmement utile pour vous assurer que les bonnes variables soient au bon endroit, mais il convient de souligner que Python ne garantit pas l’exactitude – par exemple, il vous serait toujours possible de mettre une liste de chaînes dans la fonction  highest  (le plus élevé).

Le typage Python se situe totalement dans le champ de compétences du développeur, et plus probablement, de l’IDE.

Les IDE prenant en compte des bibliothèques de vérification de types (telles que [mypy](http://mypy-lang.org/)) pourront vous avertir d’un typage non respecté. L’avantage du typage réside aussi dans **l’autocomplétion** fournie par votre IDE. En effet, typer les paramètres de fonction, par exemple, permettra à votre IDE de proposer tous les attributs disponibles pour ce paramètre. 🤗

#### Adoptez la « programmation défensive »

La base de l’approche de la programmation défensive, c’est de se prémunir contre la possibilité d’erreurs ou de fautes. Nous pouvons utiliser différents outils pour cela, et le plus simple d’entre eux est l’humble instruction if :

Une image contenant texte, logiciel, Page web, Icône d’ordinateur

Description générée automatiquement

Ici, nous nous prémunissons contre la possibilité de diviser par zéro. Python fournit quelques autres outils, comme la déclaration  assert  (affirmer) – souvent utilisée dans les tests – qui cause le déclenchement d’une Exception par un programme si quelque chose n’est pas vrai. Par exemple :

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Description générée automatiquement

Si nous donnons à cette fonction diviser (divide) un dénominateur de 0, le programme s’arrêtera avant d’essayer de résoudre la division. Attention cependant, le mot clé  assert  n’est pas conseillé pour un code de production

Python fournit également certaines fonctions pour garantir que les objets appartiennent à un type (isinstance) ou ont un attribut spécifique (hasattr), ce qui peut être utilisé ainsi :

Une image contenant texte, logiciel, capture d’écran, Page web

Description générée automatiquement

Attention, il est possible d’être excessivement défensif – après tout, chaque assertion, instruction if ou appel de fonction demande du temps informatique qui n’est pas dévolu à exécuter votre programme lui-même.

#### Gérez des exceptions

Certaines opérations non valides durant l’exécution d’un programme Python provoquent des exceptions – qui indiquent à l’utilisateur et à d’autres parties du programme qu’une erreur a eu lieu.

Nous pouvons écrire nos propres exceptions, faire en sorte qu’elles soient lancées en cas de problème, et gérer ces exceptions de différentes manières. Nous parlerons davantage des exceptions et de la gestion d’exception dans la troisième partie.

#### Une approche non interventionniste

Les scripts et les programmes plus petits ou moins importants ont fréquemment une approche beaucoup plus passive de ces erreurs. Autrement dit, se contenter de laisser le programme planter peut parfois être une option ! Bien qu’il faille l’éviter pour des applications plus grosses ou plus importantes, c’est souvent la bonne solution si une situation est irrécupérable.

### En résumé

* Les objets peuvent être stockés dans des collections – tout comme n’importe quoi d’autre.
* Python gère les types avec ce que l’on appelle le**duck typing** – si ça a un bec et cancane comme un canard, c’est probablement un canard.
* Nous devons être particulièrement attentifs, lorsque nous utilisons les collections, à ne pas essayer d’accéder à des attributs qu’un objet ne possède pas.

## Utilisez les modules et les classes pour découper votre code

### Pourquoi structurer son code ?

#### Principe n°1 : la découvrabilité

La découvrabilité désigne la facilité ou la difficulté à trouver l’emplacement de fonctionnalités précises. La découvrabilité peut aussi faire référence à la difficulté de **compréhension du code** lui-même

#### Principe n°2 : la simplicité

On peut également garantir que chaque section de code ait **aussi peu de responsabilités que possible** – de préférence une seule ! Les fonctions et méthodes qui **ne font qu’une seule chose** sont plus faciles à comprendre et à tester.

#### Principe n°3 : le style

Un code bien structuré **a un style cohérent**. Pour un projet professionnel ou open source, vous aurez un guide de style. Suivez-le pour garantir que votre code et celui de vos collègues et collaborateurs soient similaires.

### Structurez votre code

De façon générale, il convient de placer **chaque classe dans un fichier distinct**, et de donner un nom logique ou similaire entre le module et la classe qu’il contient. Si cela n’est pas possible ou pas souhaitable, on peut placer un groupe de classes liées logiquement dans le même fichier, avec un nom approprié.

### Modules et paquets Python

Un **module**, c’est simplement le code exécutable et les définitions de classe/fonction contenus dans un fichier Python unique, alors qu’un **paquet (ou package)** est une **collection de modules** regroupés logiquement dans un répertoire et partageant un **fichier de configuration** (\_\_init\_\_.py).

Cet  \_\_init\_\_.py  peut contenir n’importe quel code d’initialisation, et contient parfois une définition**\_\_all\_\_**, qui définit tous les modules du paquet, comme ceci :

\_\_all\_\_ = ["contact", "textcontact", "owlcontact"]

En plaçant ainsi les modules dans un paquet, nous obtenons des déclarations import plus logiques – ce qui est particulièrement utile si vous fournissez votre code comme bibliothèque à quelqu’un d’autre. Par exemple, si la déclaration « all » ci-dessus se trouve dans un dossier nommécontact, nous pouvons faire ceci :

Supposons que vous avez un package nommé monpackage, et que vous avez les modules module1.py, module2.py et module3.py à l'intérieur de ce package. Vous voulez seulement exporter module1 et module2 à partir de votre package. Voici comment vous pouvez le faire :

Contenu de \_\_init\_\_.py :

python

# Définition des modules à exporter

\_\_all\_\_ = ['module1', 'module2']

# Importation des modules à exporter

from . import module1

from . import module2

Dans cet exemple :

* \_\_all\_\_ est une liste de chaînes contenant les noms des modules que vous voulez exporter du package. Dans ce cas, seuls module1 et module2 seront disponibles lorsque vous importerez monpackage.
* Les directives from . import module1 et from . import module2 importent explicitement les modules que vous voulez exporter du package.
* \_\_init\_\_.py peut également contenir d'autres codes d'initialisation si nécessaire.

Cela signifie que si vous avez un script extérieur qui fait from monpackage import \*, seuls module1 et module2 seront importés et accessibles, tandis que module3 restera caché à l'utilisateur. C'est une façon de contrôler ce qui est exposé à l'utilisateur lorsqu'il importe votre package.

Lorsque vous importez un module, vous exécutez en réalité tout le code qu’il contient. Si un fichier ne contient que la définition de fonction et de classe, il ne se passe rien, mais d’autres éléments de code – les déclarations  import  étant les plus utiles – s’exécuteront.

Mais pourquoi ? Je ne nomme jamais mes fichiers \_\_main\_\_, moi !

En effet, ce nom n’est pas un nom que vous utiliserez pour nommer vos fichiers. Cependant, le fichier **principal** de votre code (celui que vous utilisez pour lancer le programme) verra automatiquement son nom changé en \_\_main\_\_  !

Cette condition permet donc d'exécuter du code **seulement si** le fichier courant est le **point d’entrée de l’application**. Le code ne s'exécutera donc pas si ce fichier a été importé depuis un autre module.

Il est particulièrement important de noter que vous avez très peu de contrôle sur l’**ordre** du code dans les différents paquets importés : ne vous reposez dessus que pour l’import de paquets et la configuration, et rien d’autre !

### En résumé

* Un fichier Python, contenant du code Python, est appelé un **module**.
* Un **paquet** est un répertoire de modules Python contenant un fichier\_\_init\_\_.py.
* Nous pouvons utiliser des déclarations **import** pour inclure des classes et des fonctions d’autres modules et paquets Python.

## Décomposez un problème de programmation orientée objet

#### Les fonctionnalités

Que doit faire votre programme ou fonctionnalité ? De quelles fonctionnalités additionnelles pourrait-on avoir besoin à l’avenir ? Comment vous assurer que votre code soit extensible, de façon à ce que ces modifications soient possibles ?

#### Les objets

Quels sont les objets qui existent dans l’espace du programme ? Quelles sont les responsabilités de chacun de ces objets ? À quoi servent-ils ? Quelles sont les relations entre ces objets ?

Nous avons parlé d’héritage pendant ce cours, lorsqu’une classe « is-a » (« est-une ») version plus spécialisée d’une autre classe. Il existe également des relations de **composition**, que nous avons évoquées sans les nommer, qui désignent des relations « has-a » (« a-un »). Par exemple, une voiture **a un** volant – le volant pourrait être stocké comme variable à l’intérieur de la voiture. C’est une façon d’imbriquer les objets entre eux, pour éviter d’avoir un seul gros objet qui ferait trop de **choses différentes**.

#### Les interfaces

Dans quelle mesure est-ce que ce code va interagir avec d’autres systèmes et d’autres parties de code ? À quoi ressembleront les interfaces – les méthodes, fonctions et autres fonctionnalités qui relient votre programme aux autres ?

## Gérez les exceptions

### Découvrez les exceptions

Python nous permet de définir nos propres exceptions personnalisées. Si nous gardons en tête que **les exceptions sont des objets**, nous pouvons définir nos propres classes d’Exceptions, exactement comme nous le ferions pour tout autre objet.

Une image contenant texte, capture d’écran, logiciel, Site web

Description générée automatiquement

Mais  InvalidAddressException  ne fait rien, alors pourquoi l’utiliser ?

De nombreuses exceptions personnalisées ressemblent à cela. Bien qu’il soit possible de surcharger des méthodes dans une exception personnalisée, le simple fait de disposer d’un type séparé nous est en réalité très utile.

Nous pouvons aussi mettre un message par défaut, ce qui est souvent plus pratique :

class InvalidAddressException(Exception):

"""Gère les exceptions liées aux mauvaises adresses."""

def \_\_init\_\_(self, address, base\_message="Adresse invalide !", \*args, \*\*kwargs):

"""Initialise le message.

L’utilisation de \*args et \*\*kwags permet de prendre un nombre

de paramètres dynamiques.

"""

msg = f"{base\_message} Adresse: {address}"

super().\_\_init\_\_(msg, \*args, \*\*kwargs)

Souvenez-vous de l’instruction except –  except InvalidAddressException, par exemple. Le bloc  except  **correspond au type** de l’exception. Autrement dit, le bloc  except  aura lieu si l’expression déclenchée est du typeInvalidAddressException. Le type de l’exception est suffisant à lui seul !

Une dernière chose : les exceptions personnalisées dans une grosse application n’héritent habituellement pas directement d’exception. Nous faisons alors généralement quelque chose qui ressemble à ceci :

class MyAppException(Exception):

pass

class MyAppInvalidAddressException(MyAppException):

pass

Nous définissons  MyAppException  comme classe de base pour notre hiérarchie d’exceptions personnalisées. Cela nous permet de savoir très rapidement si une exception est déclenchée depuis notre application ou pas ! Vous verrez que de nombreuses bibliothèques Python ont également leur propre classe de base d’exceptions personnalisées.

## Revenez sur vos acquis

Félicitations : vous êtes parvenu à la fin du cours ! Si vous débutez en programmation orientée objet, la meilleure manière de vous améliorer dans toutes les compétences liées à la programmation est de les mettre en pratique ! Concevez des systèmes, construisez-les, et apprenez de vos erreurs.

Gardez cela en tête pendant que nous récapitulons ce que nous avons appris dans ce cours...

### Les classes et les objets

La **programmation orientée objet** est un paradigme de programmation courant, dans lequel des **objets** – créés à partir de plans que l’on appelle des **classes** – interagissent pour produire des comportements.

Les **objets** ont un **état** et un **comportement**. L’état désigne les **variables** et les **données** qui correspondent à cet objet, alors que le comportement se trouve dans les **méthodes** de cet objet.

### L'héritage

Les classes et les objets peuvent avoir différentes relations entre eux, qui représentent différents concepts. Les relations **d’héritage** (« is-a », ou « est-un·e ») permettent aux classes enfants d’hériter de fonctionnalités de leurs classes parents et de les surcharger, tandis que les relations de **composition** (« has-a » ou « a-un·e ») concernent un objet contenant une référence à un ou des objet(s) autre(s). Ces dernières sont fréquemment utilisées lorsqu’un objet contient ou est composé d’autres objets.

### La décomposition

À mesure qu’un problème de programmation devient plus difficile, il en va de même pour le travail de conception nécessaire. Nous savons que nous pouvons **décomposer un problème** en parties plus petites qui interagissent, et découper notre programme en **modules** et en **paquets**.

### Les exceptions

Enfin, nous avons découvert – en acquérant des connaissances sur les **exceptions** et par un apprentissage de première main – que les choses peuvent mal se passer au sein des programmes. Il existe des**exceptions intégrées** qui nous avertissent quand nous avons fait quelque chose qui ne fonctionne pas, et des **exceptions personnalisées** que nous pouvons écrire nous-mêmes, pour qu’elles soient **déclenchées** en cas de problème.

Si tout cela vous semble familier, c’est formidable ! Sinon, il serait peut-être judicieux de jeter un coup d’œil aux chapitres correspondants avant de passer à la suite. En tous les cas, nous vous souhaitons une excellente continuation dans vos projets de développement et votre apprentissage continu de la programmation avec Python. 🎓

# [Écrivez du code Python maintenable](https://openclassrooms.com/fr/courses/7160741-ecrivez-du-code-python-maintenable)

### Découvrez le guide de style PEP 8

PEP est l'abréviation de Python Enhancement Proposal (proposition d'amélioration de Python).

[Les PEP](https://www.python.org/dev/peps/pep-0001/) sont des documents de conception pour la communauté Python. Elles décrivent des nouvelles fonctionnalités pour Python, ses processus ou son environnement. Par exemple, la **PEP 257**, que nous allons utiliser dans le prochain chapitre, est le document dédié aux conventions des chaînes de documentation.

La PEP qui sert de **guide de style** pour Python est la **PEP 8**

### En résumé

* D’autres développeurs auront fréquemment besoin d’utiliser votre code, et donc de le comprendre.
* Les conventions Python se trouvent sur le site de Python dans des documents qui s'appellent les PEP.
* La PEP 8 contient de nombreuses recommandations pour aider les développeurs Python à écrire du code compréhensible par les autres.

### Respectez les conventions de nommage

* Écrivez les noms de variables en minuscules, avec des *underscores* (tirets bas). C'est ce qu'on appelle la convention « *snake\_case*» :
* Écrivez les variables constantes en majuscules, avec des *underscores* :

En Python, on ne crée pas de variables dont la valeur ne peut pas être modifiée. Cependant, si vous voulez qu’une valeur demeure **constante**, le simple fait de l’écrire TOUT\_EN\_MAJUSCULES signale aux autres développeurs de ne pas écrire du code qui la modifie. Et, grâce à la PEP 8, ils devraient le comprendre. 🙂

* Écrivez les noms de classe avec une majuscule au début de chaque mot, sans ponctuation (la convention « *CapitalizedCase*») :
* Écrivez les noms de modules en minuscules, en évitant au maximum l’utilisation des *underscores*:
* Écrivez des phrases complètes en anglais (sauf s’il s’agit de code interne à une équipe qui préfère une autre langue).
* Évitez les commentaires qui contredisent le code. La PEP 8 est formelle : « Un commentaire qui contredit le code est pire que de ne pas en mettre ».

Ce qui nous amène à nos bonnes pratiques suivantes :

* Mettez les commentaires à jour lorsque le code change.
* Évitez les commentaires sur la même ligne que le code, ils font négligé.
* Mettez un seul espace entre le symbole dièse et le texte (pour différencier du code qui a été commenté).
* Utilisez le même niveau d’indentation que la ligne de code qui suit, pour une bonne lisibilité :

# Commentaire correctement indenté

def foo(arg):

# Commentaire correctement indenté

pass

# Commentaire mal indenté

def foo(arg):

# Commentaire mal indenté

pass

### Utilisez des chaînes de documentation (« docstrings »)

Les docstrings en début de fonction ou de classe sont spéciales, et on peut y accéder en utilisant l’attribut  \_\_doc\_\_  :

print(multiply.\_\_doc\_\_)

En Python, contrairement à d’autres langages, **rien n’est vraiment privé**. Mais vous pouvez préfixer vos méthodes d’un underscore pour *dire* qu’elles sont privées. Ce n’est cependant qu’une **convention de nommage**.

Pour finir, je vais vous exposer les deux manières de créer des docstrings :

* les docstrings sur **une ligne** (pour les cas les plus évidents) ;
* les docstrings sur **plusieurs lignes**(pour les cas plus complexes).

La première méthode va peut-être intéresser les plus pressés d’entre nous : il suffit d’une simple phrase monoligne, en mode impératif. Notez que la docstring **ne doit jamais décrire le fonctionnement interne** de sa fonction ou méthode, mais seulement son retour. Il ne s’agit pas ici d’expliquer le code, mais **d’expliquer son usage**. C’est tout son intérêt.

La deuxième méthode n’est qu’une **extension de la première**. Nous reprenons le principe de la phrase simple en première ligne. Puis nous ajoutons des **explications plus précises**, et documentons les paramètres, les retours et les exceptions.

### En résumé

* Les variables, classes et modules doivent porter des noms utiles et écrits selon le style suggéré.
* Les commentaires doivent être utilisés avec modération, maintenus à jour, et indentés de la même façon que la ligne de code suivante.
* Les docstrings sont des commentaires spéciaux, auxquels on peut accéder avec  \_\_doc\_\_  .
* À l’inverse des commentaires, usez et abusez des docstrings ! 😊

### Soyez cohérent avec l'indentation

En Python, vous pouvez utiliser soit des espaces, soit des tabulations, mais pas un mélange des deux. Du coup, il est **recommandé d’utiliser 4 espaces**, pour que les développeurs n’obtiennent pas d’erreurs d’« indentation inattendue » lorsqu’ils copient du code.

Voici quelques-unes des lignes directrices clés concernant les espaces. Vous trouverez des exemples pour chacune d’entre elles dans les exemples de code ci-dessus.

* Mettez **un seul espace** autour des opérateurs d’affectation (  is\_little = "little" in sanctuary  ) et des opérateurs logiques (  max\_places  < 5  ). La seule **exception** intervient lorsque l’on fixe des valeurs par défaut en paramètres de fonctions et méthodes, telles que  max\_places=4  .
* Ne laissez **jamais d’espaces** tout de suite à **l’intérieur de parenthèses ou de crochets**. Écrivez  (expression)  et  [0]  , pas  ( expression )   ou  [ 0 ]  .
* Ne laissez **pas de blanc** entre une fonction, comme  print()  , et ses arguments.
* Laissez **un espace** entre  if  et **toute parenthèse**. La même règle s’applique à  for  . Ceci vise à être cohérent avec les situations où il n’y a pas de parenthèses.

### Utilisez des sauts de ligne

De plus, la PEP 8 recommande ce qui suit :

* **Avant une définition de classe** (c’est-à-dire quand vous écrivez  class MyClass  ) ou **une définition de fonction** (c’est-à-dire quand vous écrivez  def my\_function(...):  ), sautez *deux lignes*.
* **Avant une définition de méthode** au sein d’une classe, ne sautez *qu’une seule ligne*.

### Limitez la longueur des lignes

C’est pourquoi la PEP 8 suggère de **limiter les lignes de code à 79 caractères**.

### Écrivez des expressions sur plusieurs lignes

Si je limite la longueur de mes lignes – comme on me le recommande – parfois, mes expressions sont trop longues et ne rentrent pas en une seule ligne ! Que dois-je faire dans ce cas ?

La PEP 8 recommande les alternatives suivantes :

# Aligne les paramètres sur la verticale

def function\_with\_a\_rather\_long\_name(parameter\_number\_1, parameter\_number\_2,

parameter\_number\_3):

my\_function(parameter\_number\_1)

return parameter\_number\_2

ou

# Même chose, mais avec un paramètre par ligne

def function\_with\_a\_rather\_long\_name(parameter\_number\_1,

parameter\_number\_2,

parameter\_number\_3):

my\_function(parameter\_number\_1)

return parameter\_number\_2

# Un paramètre par ligne, et la parenthèse au même niveau d’indentation que la fonction

def function\_with\_a\_rather\_long\_name(

parameter\_number\_1,

parameter\_number\_2,

Parameter\_number\_3

):

my\_function(parameter\_number\_1)

return parameter\_number\_2

Enfin, voici une astuce élégante pour découper vos chaînes de caractères trop longues : **utilisez une parenthèse**, et insérez à l’intérieur **plusieurs petites chaînes de caractères**, séparées seulement d’un **saut de ligne**.

super\_long\_password = (

"erfzfefrzvterbytnrezrtvbytyruetrgtrth"

"zeergvzreafz'((-'eg'((yhvgbrz'trvytrh"

"zerbetrtzbrtyezegbyebzrtbrebrtberbtrg"

"zevrebtniukoy;i;yu,yt,trntehtrgegretr"

)

Lors de l'exécution du code, elles vont se « ***recoller*** » pour n’en former qu’une grande.

### Organisez vos fichiers

Bien sûr, la PEP 8 a un ordre recommandé pour vos fichiers Python !

1. Les commentaires qui concernent la totalité du fichier vont en haut.
2. Les imports suivent cet ordre :
   1. Modules de la bibliothèque standard, par exemple  import random  .
   2. Modules de bibliothèques tierces, par exemple  import sklearn  .
   3. Modules locaux, par exemple  import mymodule  .
3. Les constantes, par exemple  MY\_CONSTANT = 77  .
4. Tout autre code.

### En résumé

* Pour que le code Python soit cohérent et facile à lire, la PEP a différentes recommandations sur la présentation du code.
* Le code doit être indenté avec 4 espaces (et non des tabulations).
* Le code doit utiliser des espaces et sauts de lignes adaptés pour aérer le code sans trop le gonfler.
* Les lignes de code ne doivent pas excéder 79 caractères, et il faut être prudent si l’on découpe de longues expressions en plusieurs lignes.
* Les fichiers doivent être écrits en commençant par les commentaires qui concernent le fichier, puis les imports, puis les constantes, et ensuite tout le reste du code.

## Écrivez du code Python antibug

Certains de ces modèles correspondent aux **patrons de conception** et aux **principes de conception SOLID** dont nous traiterons plus tard dans le cours, mais la PEP 8 possède également ses propres **recommandations de programmation,** qui vous aideront à écrire du code antibug.

### Générez des retours cohérents

* Soit toutes les instructions  return  retournent une valeur, soit aucune ne le fait.
* Tous les types de retours doivent être les mêmes (sauf s’il y a une très bonne raison de faire autrement !).
* Utilisez  return None  plutôt qu’un  return  nu.

S’il est important de retourner un message d’erreur spécifique (par exemple : « Ne divise pas correctement ! »), alors il vaut mieux lancer et attraper une exception.

### Simplifiez les exceptions

Votre objectif est que votre code ne contienne pas d’erreurs – et une façon facile de les prévenir est d’utiliser les **blocs try-except**.

* Le bloc try doit couvrir le moins de code possible, pour éviter de masquer d’autres bugs.
* N’utilisez jamais la clause   except  nue, car vous risquez de passer sous silence des erreurs critiques, et vous interférerez avec les exceptions KeyboardInterrupt  et  SystemExit  , qui servent à arrêter le programme. 😱

### En résumé

* La PEP 8 comprend différentes recommandations de programmation pour vous aider à écrire du code antibug.
* Les fonctions doivent retourner un seul type (entier, chaîne, booléen, etc.) et/ou None.
* On préfère  str.startswith  et  str.endswith  au tranchage de chaînes, car vous n’avez qu’un élément de votre code à changer lorsque vous les modifiez.
* Les blocs try doivent être placés de façon à couvrir le moins de code possible.
* Les clauses except doivent toujours attraper un type spécifique d’exception.

## Utilisez des linters pour que votre code reste propre

Un **linter** trouve les parties de votre code qui ne sont pas conformes à la PEP 8. Ensuite, il vous revient de résoudre ces problèmes.

### Utilisez un linter

Certains éditeurs ont des linters intégrés. Il existe également des linters en ligne gratuits, comme [PEP8 online](http://pep8online.com/).

Pour finir, sachez qu’il existe aussi des autoformateurs, qui vont faire un formatage automatique de votre code selon les conventions de la PEP 8. Un des plus populaires est [la bibliothèque Black](https://github.com/psf/black), qui va formater votre code d’une façon unique et “sans compromis”, selon ses auteurs. 😅

### En résumé

* Les linters sont des outils automatisés qui vous préviennent lorsque votre code n’est pas conforme à la PEP 8.
* Les linters ne trouveront pas toutes vos erreurs de programmation, mais uniquement les principales recommandations de style de la PEP 8.

## Construisez des systèmes complexes à l’aide de patterns

### Les patterns, une solution à un problème terrain

Une des roues que l’on a tendance à réinventer se situe au niveau de **l’architecture**. Nous partons souvent de zéro pour trouver des solutions à nos problèmes ; nous essayons de créer des communications d’une classe à l’autre, en tâtonnant. Il existe pourtant des programmeurs qui se sont rassemblés un jour, et qui se sont dit : « Mais attends, ton code ici utilise **un agencement que j’utilise aussi**! Et ça m’a souvent bien aidé. Et si on écrivait un livre en répertoriant tous ces mécanismes ? ».

Les design patterns étaient nés.

### Qu’est-ce qu’un design pattern ?

Un **design pattern**, aussi nommé *patron* ou *modèle de conception* en français, est une **solution éprouvée et réutilisable** à un **problème qui se produit fréquemment**. Il décrit la nature statique ou dynamique des classes et objets qui implémentent la solution. Pour tout design pattern, vous êtes libre de personnaliser la solution pour l’adapter à votre situation spécifique.

*En 1994, le* ***« Gang des Quatre » (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides)*** *sortait le livre* [*Design Patterns : Catalogue de modèles de conception réutilisables*](https://www.vuibert.fr/ouvrage/9782711786442-design-patterns)*. Ces quatre auteurs ont observé plus de 300 projets sur lesquels d’autres développeurs travaillaient. Ils ont constaté que de nombreux* ***problèmes*** *récurrents étaient* ***identiques****.*

Nous traiterons de trois patterns particulièrement adaptés à Python.

* **Le design pattern Constant :** ce pattern très simple facilite la mise à jour des valeurs dans le code pour les développeurs.
* **Le design pattern Décorateur (« Decorator ») :** ce pattern de complexité modérée facilite la création de nombreuses fonctions qui accomplissent des choses similaires.
* **Le pattern Modèle-Vue-Contrôleur (« Model-View-Controller ») :** ce pattern constitue une architecture que vous pouvez utiliser pour votre application dans son ensemble, facilitant la fiabilité des interactions des utilisateurs avec votre système.

### En résumé

* Les patterns sont des solutions éprouvées et fiables à des problèmes courants.
* Bien utilisés, les patterns conduisent à une meilleure compréhension et une meilleure maintenabilité de votre code.

## Évitez les confusions avec le design pattern Constant

### Comment utiliser le design pattern Constant

Le problème que nous avons rencontré, c’est que le code était **plein de nombres non expliqués**. Du coup, la façon dont un changement sur une valeur allait affecter les autres valeurs n’était **pas claire**.

Voici comment nous pourrions résoudre ce problème en utilisant le design pattern Constant :

* Identifiez tout nombre (ou autre variable) qui est utilisé à **plusieurs emplacements**, ou dont la signification n’est **pas claire**.
* Déclarez sa valeur en tant que variable ***globale*** (dans la portée du module), avec un nom clair – même si nous n’avons pas l’intention de la changer. Par exemple :  number\_of\_guests = 2  .
* Ces valeurs constantes seront souvent utilisées par **plusieurs fonctions, classes et fichiers** au sein d’un projet, donc d’après la PEP 8, elles devraient être définies vers le haut du fichier en majuscules, par exemple  NUMBER\_OF\_GUESTS = 2  .
* **Reformulez** toutes les déclarations qui reposent sur ce nombre en utilisant la nouvelle valeur constante.
* Ensuite, si un développeur a un jour besoin de modifier ce comportement, il n’a plus qu’à **changer la définition de la variable**, tout simplement ! Par exemple :  NUMBER\_OF\_GUESTS = 15  .

### Quand utiliser le design pattern Constant

Gardez en tête ce pattern quand vous écrivez un nombre ou une chaîne fixe dans votre code, car en cas de changement, un futur développeur rencontrerait des problèmes !

Et si je sais qu’une valeur ne changera pas ? Est-ce qu’il y a une raison d’utiliser le design pattern Constant dans ce cas ?

La fréquence à laquelle il vous faudra modifier des choses que vous pensiez gravées dans la roche éternellement vous surprendrait ! Mais il y a en réalité une autre raison d’utiliser le design pattern Constant.

Imaginez que mon équipe écrive un module Python qui fasse beaucoup de géométrie avec des cercles.

L’un d’entre nous utilise 3.14159265 comme valeur de π, un autre utilise 3,14, et j’utilise la fraction continue.

Nous allons commencer à voir des comportements étranges et incohérents à cause de ces différentes approximations.

Il serait préférable d’utiliser le design pattern Constant pour π, en écrivant par exemple :

MATH\_PI = 3.1416

Puis, chaque fois que quelqu’un aura besoin d’utiliser π dans une formule, il pourra simplement utiliser  MATH\_PI  . Problème résolu !

Attention cependant : une bonne constante est **immutable** (immuable). Elle ne doit pas être modifiée une fois le programme lancé. Cela permet **d’éviter les effets de bord** (comportements de code inattendus) propres aux variables globales.

Un objet **immutable** en Python est un objet qui ne peut être modifié. Un integer, une chaîne de caractères ou un tuple sont des valeurs immutables. Cependant, rien n’empêche de réassigner une nouvelle valeur aux variables contenant un objet immutable.

### En résumé

* Le design pattern Constant est un modèle simple qui n’affecte qu’une seule valeur.
* Les valeurs qui se répètent peuvent être définies une seule fois dans l’application.
* Les futurs développeurs pourront facilement comprendre la signification de la valeur.
* Les futurs développeurs pourront facilement modifier la valeur si les prérequis sont modifiés.
* De nombreux bugs étonnants peuvent être évités en utilisant le design pattern Constant.

## Créez des fonctions flexibles avec le design pattern Décorateur

### Comment utiliser le pattern Décorateur

#### Étape 1

Créez une **fonction décorateur** qui :

* attend une autre fonction en paramètre,
* et retourne une variation **décorée** de cette fonction en retour.

Le décorateur n’est qu’un **modificateur de fonction**. Il va la **transformer** pour rajouter des choses avant, et après.

Attendez une seconde – on peut utiliser des fonctions comme arguments d’autres fonctions, en Python ?!

Cela se fait rarement, mais oui ! En Python, tout est objet, et nous pouvons passer des fonctions, des classes et globalement tout ce qui peut être contenu dans une variable, en paramètre de fonction. Nous verrons cela en action sous peu.

Pour notre code, la fonction décorateur prendra nos fonctions de voyages pour les agrémenter automatiquement des procédures de décollage et d’atterrissage.

#### Étape 2

Chaque changement fait dans le décorateur se répercute donc directement dans les fonctions décorées !

Voici un exemple simple de décorateur écrit en Python :

*def decorate\_function(function):*

*"""Cette fonction va générer le décorateur."""*

*def wrapper():*

*"""Voici le "vrai" décorateur.*

*C'est ici que l'on change la fonction de base*

*en rajoutant des choses avant et après.*

*"""*

*print("Do something at the start")*

*result = function()*

*print("Do something at the end")*

*return result*

*return wrapper*

*def travelling\_through\_the\_stars():*

*"""Voyage à travers les étoiles."""*

*print("C'est parti pour un long voyage !")*

*# ici, nous allons récupérer le retour de "decorate\_function",*

*# qui n'est autre que la fonction "wrapper" !*

*# Notez que nous pouvons très bien renommer une fonction en*

*# l'assignant dans une nouvelle variable (ici "wrapper" devient "decorated").*

*decorated = decorate\_function(travelling\_through\_the\_stars)*

*decorated() # nous executons la fonction "wrapper"*

Donc, dans notre exemple, notre fonction de voyage ne contiendra plus que le code propre au voyage, mais les procédures de décollage et d’atterrissage seront reléguées dans la fonction “wrapper” du décorateur.

Essayons d’appliquer le pattern décorateur à nos voyages  !

### La syntaxe Décorateur en Python

L’une des conséquences regrettables des décorateurs, c’est que vous devez créer une fonction, puis la redéfinir avec le décorateur. Il serait préférable de pouvoir tout faire en une seule étape.

Heureusement, c’est possible, grâce à une syntaxe spécifique qui nous simplifie un peu la vie (on parle de sucre syntaxique) :

*def decorate\_function(function):*

*"""Cette fonction va générer le décorateur."""*

*def wrapper():*

*"""Le "vrai" décorateur."""*

*print("Do something at the start")*

*result = function()*

*print("Do something at the end")*

*return result*

*return wrapper*

*@decorate\_function # c'est ici que ça se passe !*

*def travelling\_through\_the\_stars():*

*"""Voyage à travers les étoiles."""*

*print("C'est parti pour un long voyage !")*

*# la fonction est directement décorée, et s'utilise comme telle, comme si rien*

*# comme si rien n'avait changé ! ;)*

*travelling\_through\_the\_stars()*

La syntaxe  @decorate\_function  dit à l’interpréteur Python que cette fonction doit être décorée par la fonction décorateur  decorate\_function  . Très concrètement, l’interpréteur Python va exécuter la fonction *décorateur* lors du lancement du code, et passer la fonction décorée en paramètre de son appel. Exactement comme pour la première méthode. Ici la fonction garde le même nom, mais représente en fait le “*wrapper*” retourné par le décorateur.

À quoi d’autre un décorateur pourrait-il me servir ?

Imaginons que votre code soit lent à s’exécuter, et que vous vouliez ajouter des **fonctionnalités de chronométrage et de logging** pour vous aider à trouver quelle partie s’exécute lentement.

Néanmoins, vous ne souhaitez pas modifier **directement la fonction** pour ce genre de test ! C’est là que les décorateurs entrent en jeu : ils permettent de garder le code de la fonction propre, et spécifique à une tâche seulement.

Vous pourriez donc écrire un décorateur qui démarre et arrête un chronomètre, et logue le résultat à l’écran ou dans un fichier.

*from time import time, sleep*

*def calculate\_time\_spent(function):*

*"""calcule le temps que met une fonction à s'executer."""*

*# notez \*args et \*\*kwargs. Ce sont des paramètres dynamiques*

*# qui permet au décorateur de s'adapter à tout type de fonction !*

*# N'hésitez pas à vous documenter sur l'unpacking pour en apprendre*

*# davantage.*

*def wrapper(\*args, \*\*kwargs):*

*"""Décore la fonction avec un calcul du temps."""*

*# retourne le temps en secondes depuis le 01/01/1970.*

*# On appelle cela le temps "epoch".*

*start = time()*

*result = function()*

*# mettez ici votre code. Il s'agit de faire la différence entre*

*# 2 temps "epochs", celui qui est gardé dans "start", et celui qui*

*# sera gardé dans votre variable 'end'. ;)*

*end = time()*

*time\_spent = end - start*

*print(f"secondes passées: {time\_spent:.2f}")*

*return result*

*return wrapper*

*@calculate\_time\_spent*

*def calculate\_the\_trajectory():*

*"""Calcule la trajectoire du vaisseau."""*

*print("Calcul en cours...")*

*sleep(3) # on met le programme en pause pendant 3 secondes !*

*print("Calcul terminé !")*

*calculate\_the\_trajectory()*

### En résumé

* En Python, les fonctions sont des objets comme les autres : elles peuvent donc être passées à d’autres fonctions et en sortir comme n’importe quelle autre valeur.
* Le design pattern Décorateur fournit une façon de modifier une fonction, souvent en ajoutant des fonctionnalités avant et après son exécution.
* Il peut être utile lorsque plusieurs fonctions similaires ont des fonctionnalités centrales différentes, mais des fonctionnalités partagées significatives.
* Il peut aussi être utile lorsque vous ne souhaitez pas modifier le code interne d’une fonction, pour pouvoir la réutiliser de différentes manières.
* La syntaxe  @decorate\_function  simplifie l’écriture de code impliquant des décorateurs.

## Structurez une application avec le pattern d’architecture MVC

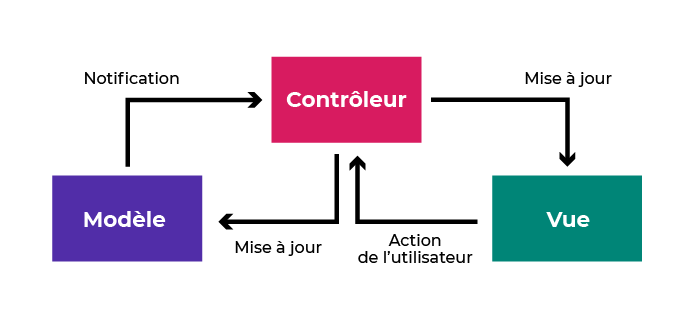
Notre dernier pattern, qui sera un pattern d’architecture, est le **MVC** – ou le pattern **Modèle-Vue-Contrôleur**. Comme nous le verrons, ce pattern constitue un excellent choix lorsque les utilisateurs doivent interagir fréquemment avec le système que vous codez !

Pour ce dernier exemple, je vais vous proposer un pattern d’architecture, qui à la différence des design patterns, **s’applique sur tout votre code**. Vous ne pourrez donc utiliser **qu’un seul** pattern d’architecture par projet, et ce projet pourra d’ailleurs inclure **plusieurs** design patterns.

### Qu’est-ce que le MVC ?

Le MVC est une approche d’architecture de logiciel. Il divise les responsabilités du système en 3 parties distinctes :

* **Modèle :** Le modèle contient les informations relatives à l’état du système. Ce sont les fonctionnalités brutes de l’application.
* **Vue :** La vue présente les informations du modèle à l’utilisateur. Elle sert d’interface visuelle et/ou sonore pour l’utilisateur.
* **Contrôleur :** Le contrôleur garantit que les commandes utilisateurs soient exécutées correctement, modifiant les objets du modèle appropriés, et mettant à jour l’application. C’est finalement les rouages de l’application, et c’est la couche qui apporte une interaction avec l’utilisateur.



Alors, que verrait-on si on pouvait implémenter le juke-box en Python ?Que contient le Modèle (M) ?

Les **informations relatives à l’état** sont conservées dans les classes du **modèle**. Ce sont les éléments qui sont vus et manipulés. De plus, s’il y a quelque chose que vous devrez stocker à long terme, ce sont bien les objets du modèle. Pour le juke-box, il pourrait s’agir de la bibliothèque de chansons et d’illustrations, de la liste des tarifs, et de la date du dernier passage du technicien.

*PRICE\_PER\_SONG = 1.20*

*class Song:*

*"""Modèle représentant un son."""*

*def \_\_init\_\_(self, name, artist, genre, artwork):*

*"""Initialise les détails relatifs au son."""*

*self.artist = artist*

*self.name = name*

*self.genre = genre*

*self.artwork = artwork*

*class Library:*

*"""Modèle qui stocke les sons."""*

*def \_\_init\_\_(self):*

*"""Initialise une liste de sons."""*

*self.songs = []*

*class ServiceInfo:*

*"""Modèle qui gère la maintenance de la jukebox."""*

*def \_\_init\_\_(self, status, engineer\_name):*

*"""Initialise les détails du service."""*

*self.service\_date = datetime.now()*

*self.status = status*

*self.engineer = engineer\_name*

### Que contient la Vue (V) ?

La vue correspond à la façon dont le modèle est **présenté** à l’utilisateur, et dont il **interagit** avec lui. C’est l’élément qui est le plus susceptible de changer. Il vous faut clairement distinguer la manière dont cette partie interagit avec le reste de votre système. Pour le juke-box, cela inclurait le panneau de boutons servant à sélectionner la musique, la fente où insérer la pièce pour payer, et les haut-parleurs qui passent vos classiques préférés.

*class Touchscreen:*

*"""Vue qui gère l'interface de la jukebox."""*

*def select\_song(self):*

*"""Sélectionne un son."""*

*pass*

*def prompt\_for\_next\_song(self, songs):*

*"""Demande un nouveau son."""*

*for song in songs:*

*# affiche les sons*

*pass*

*return "Dark Chest of Wonders"*

*class Speakers:*

*"""Vue qui gère le son."""*

*def \_\_init\_\_(self):*

*"""Initialise le volume."""*

*self.volume = 5*

*def get\_louder(self):*

*"""Augmente le volume."""*

*self.volume += 1*

*def get\_quieter(self):*

*"""Baisse le volume."""*

*self.volume -= 1*

*def play\_song(self, song):*

*"""Joue la musique."""*

*pass*

*class CoinSlot:*

*"""Vue qui gère la reception de l'argent."""*

*def \_\_init\_\_(self, amount):*

*"""Initialise le montant."""*

*self.amount = amount*

*def request\_money(self, amount):*

*"""Attend un montant de l'utilisateur."""*

*# attend l'argent*

*# donne le change*

*self.amount += amount*

*return True*

Comment la vue peut-elle contenir des haut-parleurs ? Ils ne produisent pas d’images !

C’est vrai – mais *Vue* est simplement le nom générique donné à toutes les composantes du système qui assurent l’interface avec l’utilisateur. Dans notre prochaine application, un jeu de cartes, la vue sera simplement constituée par les outputs de la console, mais pour d’autres applications, il pourrait plutôt s’agir d’une interface API, ou d’une page web.

### Que contient le Contrôleur (C) ?

C’est dans le contrôleur que le **flux de l’application** est géré. Tout le **séquençage des interactions** entre l’utilisateur et le système s’y trouve. Voyez le contrôleur comme une sorte de colle entre le modèle et la vue. L’utilisateur interagit avec la vue, qui interagit à son tour avec le contrôleur. Le contrôleur apporte ensuite les modifications appropriées aux objets du modèle, en crée de nouveaux, ou supprime ceux dont on n’a plus besoin.

Dans le juke-box, cela inclurait la logique de sélection des chansons d’après le catalogue que vous pouvez consulter, la logique qui attend que vous ayez payé avant que l’on ne passe la musique, et le système pour appeler le technicien. Notez la façon dont le contrôleur utilise et modifie le modèle, et celle dont il dit quoi faire à la vue.

class Controller:

"""Contrôleur principal."""

def \_\_init\_\_(self):

"""Initialise les modèles et les vues."""

self.bank = CoinBox()

self.library = Library()

self.service\_history = []

self.ui = Touchscreen()

self.audio\_output = Speakers()

def play\_next\_song(self):

"""Joue le prochain son."""

songs\_to\_suggest = []

for song in self.library:

# filter logic

songs\_to\_suggest.append(song)

chosen\_song = self.ui.prompt\_for\_next\_song(songs\_to\_suggest)

request\_money(PRICE\_PER\_SONG)

self.audio\_output.play\_song(chosen\_song)

# Beaucoup plus de méthodes ici...

### Quels sont les avantages du MVC ?

Imaginez que vous souhaitiez moderniser votre juke-box avec vos titres récents favoris. Vous n’avez qu’à mettre à jour le modèle ! La vue et le contrôleur travailleront sans accroc avec la bibliothèque mise à jour.

Ou alors, imaginez que vous souhaitiez ajouter des écouteurs sans fil, afin que vos amis et vous-même puissiez faire la fête sans déranger le voisinage. Ceci n’affecte que la vue, tandis que le modèle et le contrôleur restent inchangés.

Cela signifie aussi que vous pouvez tester la technologie des écouteurs avant de les acheter et de les installer – essayez-les simplement avec un autre appareil pour écouter de la musique au magasin !

Le fait de séparer ainsi des parties de l’architecture en morceaux différents, avec des responsabilités distinctes, rend le système beaucoup plus facile à :

* comprendre ;
* modifier ;
* tester ;
* réparer.

D’ailleurs, plus loin dans ce cours, nous traiterons des **principes SOLID**, qui constituent des règles que vous pouvez suivre pour vous assurer que votre code est maintenable pour ces raisons. Pas de surprise – nous verrons que le design pattern MVC suit très bien ces principes !

### En résumé

* Le MVC assure que votre code est facile à maintenir, en séparant les responsabilités :
  + Le modèle contient les informations sur l’état.
  + La vue contient les éléments qui interagissent avec l’utilisateur.
  + Le contrôleur s’assure que la séquence des étapes se déroule correctement.

## Implémentez le Modèle pour votre application

**Spécifications pour la jouabilité**

* Créez un jeu de cartes standard de 52 cartes.
* Entrez des noms de joueurs. Limitez le nombre de joueurs à cinq.
* Battez les cartes.
* Distribuez une carte à chaque joueur (face vers le bas).
* Retournez les cartes de tous les joueurs, montrant quelle carte ils ont.
* Vérifiez quel joueur a la carte de plus haut rang : As > Roi > Reine > Valet > 10… > 2.
* En cas d’égalité, on départage les joueurs grâce à la couleur :

Trèfle > Pique > Cœur > Carreau.

* Montrez le nom et la carte du gagnant.
* Remettez toutes les cartes dans la pile.
* Retournez à l’étape du battage des cartes.

Ce type de découpage s’appelle aussi un ***use case***, ou cas d’usage.

### [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from [list](https://docs.python.org/3/library/stdtypes.html#list); however, this class can be easier to work with because the underlying list is accessible as an attribute.

*class* collections.UserList([*list*])

Class that simulates a list. The instance’s contents are kept in a regular list, which is accessible via the [data](https://docs.python.org/3/library/collections.html#collections.UserList.data) attribute of [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) instances. The instance’s contents are initially set to a copy of list, defaulting to the empty list []. list can be any iterable, for example a real Python list or a [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) object.

In addition to supporting the methods and operations of mutable sequences, [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) instances provide the following attribute:

data

A real [list](https://docs.python.org/3/library/stdtypes.html#list) object used to store the contents of the [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) class.

**Subclassing requirements:** Subclasses of [UserList](https://docs.python.org/3/library/collections.html#collections.UserList) are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

### [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) objects

The class, [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from [str](https://docs.python.org/3/library/stdtypes.html#str); however, this class can be easier to work with because the underlying string is accessible as an attribute.

*class* collections.UserString(*seq*)

Class that simulates a string object. The instance’s content is kept in a regular string object, which is accessible via the [data](https://docs.python.org/3/library/collections.html#collections.UserString.data) attribute of [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) instances. The instance’s contents are initially set to a copy of seq. The seq argument can be any object which can be converted into a string using the built-in [str()](https://docs.python.org/3/library/stdtypes.html#str) function.

In addition to supporting the methods and operations of strings, [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) instances provide the following attribute:

data

A real [str](https://docs.python.org/3/library/stdtypes.html#str) object used to store the contents of the [UserString](https://docs.python.org/3/library/collections.html#collections.UserString) class.

Changed in version 3.5: New methods \_\_getnewargs\_\_, \_\_rmod\_\_, casefold, format\_map, isprintable, and maketrans.

### En résumé

* Le modèle est constitué des éléments avec lesquels vous avez des interactions. Ceux-ci contiennent les informations sur l’état du système.
* Pour trouver vos objets de modèle, consultez la liste des prérequis.
* Dans notre application, nous avons défini que :

○      Le modèle est constitué d’un joueur, d’une main, d’une carte à jouer, d’un jeu, d’un rang et d’une couleur.

○      Un joueur possède une main. Une main contient des cartes à jouer. Un jeu contient des cartes à jouer.

## Implémentez le Contrôleur et la Vue pour votre application

https://github.com/OpenClassrooms-Student-Center/ecrivez-du-code-python-maintenable-jeu-de-carte

### Comment planifier le Contrôleur ?

Le contrôleur est responsable du **séquençage** des **interactions utilisateur**. C’est un peu le médiateur de l’application, qui vérifie son fonctionnement global, et utilise vues et modèles quand il le faut. Si l’on observe le jeu, voici à quoi ressemble la séquence :

* Créez le jeu.
* Entrez les noms des joueurs.
* Battez les cartes.
* Distribuez une carte à chaque joueur.
* Retournez les cartes.
* Identifiez le vainqueur.
* Montrez le vainqueur.
* Reconstituez le jeu.

Comment le contrôleur sait-il quels inputs ont été donnés ?

La commande input vient de la vue. N’oubliez pas que la vue a deux buts  : elle présente les informations à l’utilisateur, mais elle lui demande aussi tout input nécessaire. Ce sera cependant au contrôleur de récupérer cet input pour le traiter !

Nous devons maintenant regarder la séquence des étapes ci-dessus et trouver les interactions entre le contrôleur et la vue :

#### **Démarrez le jeu**

La première étape de toute interaction est d’**instancier le modèle**. Le contrôleur reçoit ou construit les objets fondamentaux du jeu. Ce sont le jeu de 52 cartes, et une liste de joueurs vide.

Il doit également être informé de la vue. La vue doit être créée ailleurs, et passée au contrôleur, plutôt que de faire créer sa propre vue au contrôleur.

Pourquoi passer la vue au contrôleur, plutôt que de simplement laisser le contrôleur la créer ?

Eh bien, si vous deviez créer une version mobile de cette application, il lui faudrait une interface utilisateur (une vue) complètement différente. Vous devriez ensuite modifier le contrôleur. Dans l’idéal, le modèle, la vue et le contrôleur doivent chacun être remplaçables sans toucher aux autres composantes.

Dans la pratique, il n’existe pas de MVC parfait ! Le MVC peut se décliner en plusieurs formes, et chaque application est une implémentation spécifique. Certains MVC ont des controllers qui débordent sur les vues, et vice-versa. Il faut quand même tendre un maximum vers une séparation complète et garder cette rigueur pour profiter des avantages du pattern. Par contre, on évite toujours de coupler le modèle à n’importe quelle autre couche. 😉

#### **Entrez les noms de joueurs**

Pour chaque joueur qui entre son nom dans le contrôleur, ce dernier crée un objet Joueur et l’ajoute à sa liste de joueurs (dans le modèle).

#### **Indiquez que la saisie des noms est terminée**

Le contrôleur a également besoin de savoir quand tous les joueurs ont été ajoutés, pour qu’il puisse commencer à distribuer les cartes, etc.

#### **Distribuez les cartes**

Le contrôleur bat le jeu de cartes, puis pour chaque joueur il retire la carte du dessus du jeu pour la placer dans la main de ce joueur (tout cela dans le modèle).

#### **Révélez les cartes**

Le contrôleur retourne la carte de chaque joueur, puis identifie le vainqueur.

#### **Rejouez**

Les cartes sont reprises au joueur et remises dans le jeu. Le jeu peut alors recommencer depuis l’étape « Distribuez les cartes ».

Dans l’implémentation actuelle, la classe  Card  contient une méthode pour se comparer à d’autres cartes à jouer, afin de choisir la meilleure carte. Mais que se passerait-il si nous souhaitions utiliser ces objets  Card  dans un autre jeu avec des règles différentes ? Nous devrions alors les modifier ou les remplacer ! 😮

Lorsque nous nous pencherons sur les principes SOLID, nous verrons plus en détail pourquoi le fait de placer la logique de comparaison des cartes dans la classe  Card  est un mauvais choix de conception, et nous trouverons un meilleur moyen de faire les choses.

### Comment implémenter le Contrôleur ?

Le contrôleur doit avoir des méthodes qui correspondent aux différentes étapes du jeu. Elles peuvent utiliser n’importe quel objet du modèle, demander des informations à l’utilisateur par le biais de la vue, et présenter des informations à la vue.

Créons ces méthodes ensemble ! Reprenez le code des modèles [en suivant ce lien](https://github.com/OpenClassrooms-Student-Center/ecrivez-du-code-python-maintenable-jeu-de-carte/tree/31107a1e66bded52d5ed7cba8cd84fa286194e0e). Puis, suivez le fil avec la capture d’écran en vidéo ci-dessous :

Regardons ce que nous avons construit :

1. **Démarrez le jeu :** le constructeur de  Controller  s’attend à ce qu’on lui donne un Jeu et une Vue, et instancie une liste d’objets Joueurs.
2. **Entrez les noms de joueurs :** la méthode  add\_player  les récupère depuis la vue.
3. **Indiquez que la saisie des noms est terminée :** la méthode  run  interprète l’input None comme signifiant qu’il n’y a plus d’autres joueurs.
4. **Distribuez les cartes :** c’est géré par la méthode  start\_game  , car c’est la première chose qui se produit au début du jeu.
5. **Révélez les cartes :** la méthode  run  attend un signal de l’utilisateur, puis elle retourne les cartes et appelle la méthode  evaluate\_game  .
6. **Rejouez :** rebuild\_deck  récupère toutes les cartes des joueurs, les remet dans le jeu de cartes, puis remet l’état du jeu au début de l’étape 4.

### Comment implémenter la Vue ?

N’oubliez pas que la vue est à la fois l’**interface utilisateur** et le **générateur d’événements**. Notre contrôleur appellera les méthodes de la vue à la fois pour présenter des informations et pour récolter des informations de la part de l’utilisateur. La Vue ne doit par contre jamais traiter les entrées de l’utilisateur : c’est au contrôleur de s’en charger.

N’oubliez pas que si vous pouvez vous contenter de partager des valeurs “simples” (int, str, bool) entre vos différentes couches de l’application, vous pouvez aussi passer des objets complexes comme des listes, des dictionnaires ou encore des instances de classe. La plupart des design patterns reposent d’ailleurs sur les liens que peuvent faire les classes entre elles. 🧐

Alors, à quoi ressemblera la classe Vue (  View  ) ? Construisons-la ensemble !

Regardons ce que nous avons ajouté :

1. **Entrez les noms de joueurs :** la vue envoie la chaîne entrée au contrôleur, qui crée un objet Player et l’ajoute au modèle.
2. **Révélez les cartes :** la vue donne à l’utilisateur un moyen de faire retourner les cartes et évaluer le jeu.
3. **Démarrez le jeu :** la vue donne à l’utilisateur un moyen d’indiquer s’il veut rejouer ou quitter l’application.

Notre implémentation est simple et directe. Elle montre simplement les noms des joueurs, les cartes distribuées et les résultats sur la ligne de commande. Néanmoins, vu que nous avons séparé cette fonctionnalité, nous pourrions facilement remplacer ce mécanisme par une implémentation GUI, ou même des haut-parleurs et un micro !

Il ne reste plus qu’à ce que le code appelant crée le Jeu, la Vue et le Contrôleur, puis exécute l’application – regardons-les collaborer.

### En résumé

* Le contrôleur est responsable du séquençage du cas d’usage, et valide les événements envoyés par la vue.
* La vue est responsable de la présentation du modèle, de l’information de la séquence, et de la récolte des inputs de l’utilisateur.
* Les responsabilités du contrôleur sont définies en regardant les étapes de l’application. Les responsabilités de la vue sont définies par ce que le contrôleur doit montrer à l’utilisateur.

###  Question 6

**Voici une liste de prérequis** **:**

* Créez une paire de dés à 20 faces.
* Les joueurs entrent leur nom, ainsi que leur nom d’utilisateur.
* Un modérateur lance un autre dé à 20 faces.
* Tous les joueurs lancent leur paire de dés et notent leur score.
* Pour chaque score supérieur à celui du modérateur, le joueur marque un point.
* Le premier joueur à atteindre 10 points gagne.

**Lorsqu’on implémente ces prérequis avec le pattern MVC, quels éléments doivent être inclus dans le modèle ?**

* Dé, joueur, gagner un point par dé, 10 points pour gagner.
* Dé, joueur, modérateur, score du joueur.
* Créez le dé, entrez les noms, lancez le dé, déterminez le gagnant.
* Il faut discuter davantage avec le client.

Passons en revue les mauvaises réponses :

* Réponse 1 : 1 point par joueur et 10 points pour gagner sont deux éléments de logique business, qui doivent se trouver dans le contrôleur.
* Réponse 3 : Ce sont toutes des actions, qui doivent se trouver dans le contrôleur.
* Réponse 4 : Le fait de discuter avec le client pourrait apporter des éclaircissements, mais nous en savons déjà assez pour identifier quels éléments doivent se trouver dans le modèle, la vue et le contrôleur.

### Question 7

**En modèle-vue-contrôleur, quels fragments de code parmi les suivants appartiennent au modèle ?**

Attention, il y a plusieurs bonnes réponses.

Attention, plusieurs réponses sont possibles.

class Stopwatch:

def \_\_init\_\_(self):

self.start\_time = time.time()

def begin\_timing(self):

self.start\_time = time.time()

def get\_elapsed\_seconds(self):

return int(time.time() - self.start\_time)

class Stopwatch:

def display\_second\_hand(seconds):

# implementation

def reset\_second\_hand():

self.display\_second\_hand(0)

class RaceParticipant:

def \_\_init\_\_(self, name):

self.name = name

self.time\_taken = None

class Race:

def \_\_init\_\_(self):

self.racers = []

self.stopwatch = Stopwatch()

def add\_racer(self, name):

self.racers.append(RaceParticipant(name))

def start\_race(self):

self.stopwatch.begin\_timing()

for racer in self.racers:

racer.time\_taken = 0

Passons en revue les mauvaises réponses :

* Réponse 2 : Ce code semble se rapporter à l’affichage d’informations (vue).
* Réponse 4 : Ce code séquence les différents événements qui se produisent (contrôleur).

### Question 9

**En modèle-vue-contrôleur, quel fragment de code parmi les suivants appartient à la vue ?**

class PossibleView:

def \_\_init\_\_(self, name, username):

self.name = name

self.username = username

self.current\_score = 0

def add\_points(self, points):

self.current\_score += points

class PossibleView:

def \_\_init\_\_(self, name, username):

self.name = name

self.username = username

self.current\_score = 0

def set\_random\_username(self):

self.username = username\_manager.get\_random\_unique\_name()

class PossibleView:

def \_\_init\_\_(self):

self.text = TextBox()

self.player\_list = ListBox()

def handle\_user\_keypress(char):

self.text.append\_text(char)

def add\_new\_player(player):

player\_list.append(player)

class PossibleView:

def \_\_init\_\_(self, view, element):

self.view = view

self.model\_element = element

def process\_event(event) {

self.model\_element.set\_new\_value()

view.update(self.model\_element.value

Passons en revue les mauvaises réponses :

* Réponse 1 : Cette classe ne contient que des informations sur l’état (modèle).
* Réponse 2 : Cette classe contient des informations sur l’état et la logique requise (modèle).
* Réponse 4 : Ce code modifie des informations de départ sur la base d’un événement utilisateur entrant (contrôleur).

## Découvrez les bonnes pratiques de programmation avec les principes SOLID

### Comprenez les pièges des solutions compliquées

Pensez au principe [KISS](https://fr.wikipedia.org/wiki/Principe_KISS) (Keep It Simple, Stupid ou “garde ça simple, idiot”) ! 👶

Un avantage : votre solution sera **plus facile à comprendre**. Si elle est plus facile à comprendre, alors elle est **plus facile à modifier**. De plus, vous pouvez avoir davantage de certitude qu’une modification ne cassera rien.

Autre avantage, elle sera **plus facile à tester**. Mieux vos classes sont séparées, et plus il sera facile de les tester séparément.

Plus facile à dire qu’à faire ! Heureusement, vous pouvez capitaliser sur les connaissances de ceux qui vous ont précédé, et qui ont trouvé de meilleures façons de faire les choses ! Nous avons déjà vu cela avec les **design patterns** de la section précédente. Nous irons maintenant plus loin dans cette voie avec les **principes de conception SOLID**.

### Identifiez les principes SOLID

Chacune des lettres de l’acronyme SOLID représente une excellente idée à garder à l’esprit lorsque vous construisez l’architecture de votre système. Nous examinerons chacune d’entre elles en profondeur au fil du cours. Nous les mettrons également en pratique en améliorant notre application de jeu de cartes. Commençons par une vue d’ensemble de ces idées clés :

* **« S » désigne la responsabilité unique (« Single responsibility »).**

Chaque classe ou fonction doit faire une seule chose, et la faire bien. Elle ne doit avoir qu’une seule raison de changer.

* **« O » désigne le principe ouvert/fermé (« Open/Closed »).**

Les classes doivent être ouvertes à l’extension, mais fermées à la modification.

Qu’est-ce que cela peut bien vouloir dire ?

Eh bien, dans l’idéal, il doit être facile d’ajouter un nouveau concept au système en **étendant** la fonctionnalité d’origine, sans dupliquer tout un tas de code. De plus, dans l’idéal, vous ne devriez pas avoir à apporter de **modifications** au code existant, dans l’aventure.

* **« L » désigne la substitution de Liskov.**

Les sous-classes doivent pouvoir faire tout ce que font leurs classes parentes. Si vous remplacez une classe parente par l’une de ses sous-classes, cela ne doit pas casser votre système !

* **« I » désigne la ségrégation des interfaces (« Interface Segregation »).**

Cela correspond essentiellement au principe de responsabilité unique, appliqué aux interfaces.

* **« D » désigne l’inversion des dépendances (« Dependency Inversion »).**

Les classes parentes ne doivent pas avoir à changer lorsque l’une de leurs sous-classes est modifiée.

Comme nous le verrons, les design patterns que nous avons déjà abordés sont efficaces, car ils démontrent ces principes ! Nous les utiliserons pour améliorer encore davantage le jeu de cartes que nous avons écrit lorsque nous avons exploré le MVC.

### En résumé

* Les designs plus propres sont plus faciles à comprendre, maintenir, modifier, et tester.
* Si l’on suit les principes SOLID, on obtient un design plus propre.
* Les principes SOLID sont :

○      Responsabilité unique (« single responsibility »).

○      Principe ouvert/fermé (« open/closed »).

○      Substitution de Liskov.

○      Ségrégation des interfaces (« interface segregation »).

○      Inversion des dépendances (« dependency inversion »).

## « S » pour le principe de responsabilité unique (« single responsibility »)

### Pourquoi utiliser le principe de responsabilité unique ?

Chaque personne qui utilise votre code doit pouvoir comprendre ce qu’il se passe. Le fait de réduire au **minimum** les **responsabilités** d’une classe aide à atteindre cet objectif.

* Premièrement, le fait de trouver le code qui vous intéresse est facilité
* Deuxièmement, les **tests unitaires** sont beaucoup **plus faciles à écrire**

Si vous avez du mal à trouver un nom facile pour une classe ou une fonction, cela vous indique que vous violez ce principe. Si un élément accomplit de nombreuses choses différentes, il devient difficile de lui attribuer un nom.

Si vous êtes tenté d’inclure le mot « and » (« et ») dans son nom, voilà un autre indicateur de non-respect du principe de responsabilité unique !

### En résumé

* La responsabilité unique signifie que chaque classe ou fonction ne doit faire qu’une chose, c’est-à-dire qu’il ne doit y avoir qu’une seule raison de la modifier.
* C’est un principe facile à enfreindre quand vous ajoutez de nouvelles fonctionnalités au système. Lorsque vous ajoutez une nouvelle fonctionnalité, posez-vous ces questions :

○      Quels sont les changements à venir qui pourraient impacter cette classe ?

○      Qu’est-ce qui pourrait donner à la classe plus d’une raison d’être modifiée ?

* N’oubliez pas que, si une classe imite un concept de la vraie vie, elle doit implémenter uniquement la responsabilité unique qui correspond à ce concept du monde réel.

## « O » pour le principe ouvert/fermé (« open/closed »)

### Pourquoi utiliser le principe ouvert/fermé ?

Si vous **ne modifiez pas le code existant**, vous ne risquez pas de **casser ce qui marche**. L’ensemble de la nouvelle fonctionnalité est contenu dans la ou les fonction(s) et classe(s) nouvellement ajoutée(s).

Le plus difficile avec ce principe, c’est de reconnaître quand il peut être appliqué avant de commencer à coder. Voici deux **lignes directrices** pour vous aider à reconnaître quand vous pouvez envisager d’appliquer le principe ouvert/fermé.

* Lorsque vous avez des **algorithmes** qui accomplissent des calculs (coût, taxe, score dans un jeu, etc.) : il est probable que l’algorithme change au fil du temps.
* Lorsque vous avez des **données qui entrent ou sortent du système**: la destination finale (fichier, base de données, autre système) changera probablement, de même que le format concret des données.

### En résumé

* Le principe ouvert/fermé déclare que les classes doivent être ouvertes à l’extension, mais fermées à la modification.
* Dans les systèmes existants, vous pourrez avoir à retravailler le code pour lui permettre de bénéficier du principe ouvert/fermé.

## « L » pour le principe de substitution de Liskov

### Qu’est-ce que la substitution de Liskov ?

**L’héritage** commence par une bonne idée. Vous avez un concept existant, et vous voulez en ajouter un autre. Ce nouveau concept consiste simplement en une implémentation plus spécifique du concept original. Facile : **dérivez une nouvelle classe !**

Mais avec le temps, il est **facile d’abuser de l’héritage**. Lorsqu’il y a trop de couches d’héritage, c’est normal de perdre le fil de ce que toutes les classes spécifiques sont censées faire. À la fin, une classe incapable de faire quelque chose que ses classes parentes doivent pouvoir faire est ajoutée, et le système est cassé.

Il s’agit d’une violation de la substitution de Liskov, qui dit que :

Les objets des classes ***enfants*** doivent pouvoir remplacer les objets des classes ***parentes*** sans nuire à l’intégrité de l’application.

Autrement dit :

Tout code appelant des méthodes sur des objets d’un type spécifique doit continuer à fonctionner lorsque ces objets sont remplacés par des instances d’un sous-type.

Qu’est-ce qu’un sous-type ?

Un **sous-type** peut être soit une classe qui étend une autre classe, soit une classe qui implémente une interface.

Ce principe porte le nom de **Barbara Liskov**, l’une des premières femmes à avoir obtenu un doctorat en sciences informatiques aux États-Unis. Elle est également la créatrice des langages de programmation Argus et CLU.

Elle a présenté pour la première fois le principe de substitution de Liskov en 1987, lors de son discours d’ouverture de conférence intitulé « Data Abstraction » (« Abstraction des données »).

En tant qu’informaticienne pure et dure, elle l’a défini de façon formelle ainsi :

Si Φ(x) est une propriété démontrable pour tout objet x de type T, alors, Φ(y) doit être vrai pour les objets y de type S, lorsque S est un sous-type de T.

Mais cette définition semble très théorique, donc si vous êtes comme moi, il vous sera probablement plus facile de penser à ce principe en utilisant les définitions précédentes !

### Pourquoi utiliser le principe de substitution de Liskov ?

Disons que vous ayez une classe nommée félin. Elle possède une méthode nommée  manger()  . Vous pouvez passer une marque de nourriture pour chats standard – par exemple « Chaton Gourmet » – dans la méthode manger, et le chat (qui est un félin) ira très bien. 🐈 Si vous ajoutez une nouvelle classe, tigre, qui est aussi une sorte de félin, la méthode manger sera implémentée à nouveau. 🐅

En revanche, les tigres ne mangent pas de nourriture déshydratée en sachet. Ils mangent de la viande crue. Ce n’est pas le même comportement que celui que vous avez vu précédemment. Surprise !

Le problème survient lorsque vous pensez que le fait de passer la nourriture “Chaton Gourmet” dans la méthode  manger  fonctionne pour tous les félins. Un tigre est un félin, donc ça devrait marcher. Mais ça ne marche pas. Par conséquent, vous ne pouvez pas simplement mettre un tigre dans votre système où vous aviez un félin.

Cet exemple viole le **principe de substitution de Liskov**: vous ne pouvez pas remplacer la classe de base (félin) avec une classe dérivée (tigre) sans affecter le reste du système.

Le principe de substitution de Liskov vous aide en **limitant votre utilisation de l’héritage**. Bien qu’il soit facile de créer des classes de niveau bas et d’implémentation concrète (après tout, elles font ce que vous voulez qu’elles fassent), il est préférable de prendre un peu de recul et de réfléchir à une abstraction de haut niveau.

Par exemple, essayons de résoudre le problème du félin et du tigre. Vous avez là en réalité deux classes concrètes (c’est-à-dire, d’implémentation spécifique). Il vous faut introduire quelques interfaces/abstractions de plus haut niveau :

* Carnivore  , qui ne mange que de la viande.
* Omnivore  , qui mange de la viande et d’autres choses (comme du Chaton Gourmet).

Maintenant, quand vous ajoutez n’importe quel animal, vous pouvez lui faire implémenter l’une de ces deux **interfaces**.

Il semblait logique au départ de considérer le tigre comme un type de félin. Mais vous avez vu le problème qui est survenu au stade de l’implémentation. Il est souvent **difficile de reconnaître à l’avance** quand le principe de Liskov sera enfreint. Cependant, lorsque vous découvrez une violation, vous devez repenser votre utilisation de l’héritage. Par conséquent, lorsque vous voulez utiliser l’héritage, posez-vous les questions suivantes :

* Est-ce que la classe dérivée a une implémentation significative pour toutes les méthodes surchargées ? Si oui, c’est une bonne chose. ✅
* Est-ce que l’implémentation d’une méthode surchargée sortirait de l’ordinaire et risquerait d’avoir pour conséquence de lancer une exception ? Si oui, c’est mauvais signe. ❌
* Est-ce que l’implémentation d’une méthode surchargée ignorerait l’appel, et ne ferait rien ? Habituellement c’est mauvais signe, mais ça pourrait se justifier.

Observez si la classe dérivée agit ainsi pour :

○      Une seule méthode. ✅

○      De nombreuses méthodes. ❌

### En résumé

* Le principe de substitution de Liskov s’applique aux hiérarchies d’héritage. Il est enfreint lorsqu’une classe dérivée ne peut pas prendre la place d’une classe de base sans casser le système.
* Pour vous assurer de ne pas enfreindre cette règle, essayez de penser d’abord aux abstractions/interfaces de haut niveau, avant d’envisager les implémentations de bas niveau/concrètes.

## « I » pour le principe de ségrégation des interfaces (« Interface Segregation »)

Si vous vous sentez à l’aise avec le principe de responsabilité unique, le **principe de ségrégation des interfaces** vous semblera parfaitement logique. Il s’agit du même principe, mais appliqué aux interfaces :

*Une interface (représentée par une classe abstraite ne définissant que certains attributs – vides – en Python) doit décrire un ensemble de comportements.*

Vous pouvez rencontrer les mêmes problèmes avec les interfaces qu’avec les classes. Il vous faut ajouter une nouvelle capacité à une interface existante, plutôt que d’en créer une nouvelle. Maintenant, toutes les classes chargées d’implémenter doivent prendre en compte un élément supplémentaire. 😐

En vous assurant que vos **interfaces** restent **petites** et pertinentes, vous **diminuez le couplage**. Le couplage désigne à quel point deux éléments de logiciel sont liés. Plus une interface définit d’éléments, plus une classe qui implémente doit en faire à son tour. Cette situation rend cette classe **moins réutilisable**.

En raison de la célèbre fonctionnalité de **duck typing** (littéralement, typage de canard) de Python, les interfaces ne sont pas utilisées aussi largement que dans d’autres langages de programmation. Si un objet marche comme un canard et nage comme un canard, alors Python est satisfait de le traiter comme un canard, même s’il s’agit en réalité d’un cygne ! 🦢

Malgré tout, vous vous retrouverez fréquemment dans une situation où vous voudrez que deux classes (ou plus) agissent comme alternative l’une à l’autre, même si elles n’implémentent aucune interface explicite. Dans ce cas, le principe de **ségrégation des interfaces** s’applique toujours ! Il vous dit de ne pas exposer de méthodes inutiles dans ces classes. En effet, si vous le faites, vous devrez ajouter une méthode équivalente à toutes ses alternatives.

Une interface dans le contexte de la programmation orientée objet définit un ensemble de comportements que les classes qui l'implémentent doivent fournir. En Python, comme le langage ne fournit pas de support natif pour les interfaces comme certains autres langages (comme Java), les interfaces sont souvent représentées par des classes abstraites qui définissent des méthodes ou des attributs sans les implémenter.

Dans le cas de la définition que vous avez fournie, il est dit qu'une interface est représentée par une classe abstraite qui déclare certains attributs (ou méthodes) mais ne les implémente pas. Cela signifie que la classe abstraite définit simplement la signature (la structure) des attributs ou méthodes, mais elle ne fournit pas leur implémentation concrète.

Par exemple, considérons une interface Shape qui définit un comportement pour toutes les formes géométriques. Cette interface pourrait être représentée par une classe abstraite en Python de la manière suivante :

python

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

Dans cet exemple, la classe abstraite Shape définit deux méthodes abstraites area() et perimeter(), mais elle ne fournit pas d'implémentation pour ces méthodes. Les classes qui souhaitent implémenter l'interface Shape doivent fournir leurs propres implémentations pour ces méthodes. Cela garantit que toutes les formes géométriques implémentent une méthode area() et une méthode perimeter(), mais laisse aux classes concrètes la liberté d'implémenter ces méthodes de la manière la plus appropriée pour chaque forme spécifique.

[*https://github.com/OpenClassrooms-Student-Center/ecrivez-du-code-python-maintenable-jeu-de-carte/blob/471ff0cfeb32a311450683a2a022a81cd959f851/controllers/base.py*](https://github.com/OpenClassrooms-Student-Center/ecrivez-du-code-python-maintenable-jeu-de-carte/blob/471ff0cfeb32a311450683a2a022a81cd959f851/controllers/base.py)

Nous avons réussi à écrire une série d’objets de la vue alternatifs, qui ne sont pas encombrés par des méthodes inutiles, malgré la fonctionnalité supplémentaire qui a besoin d’être conservée dans  PlayerView  .

### En résumé

* La ségrégation des interfaces correspond au principe de responsabilité unique pour les interfaces.
* Ce principe est facile à enfreindre. Il peut être tentant d’ajouter une nouvelle méthode à une interface existante, vu qu’elle fait déjà quelque chose.
* En cas de doute, il vaut mieux avoir deux interfaces avec peu de méthodes à implémenter, qu’une seule interface qui ait trop de responsabilités.

## « D » pour le principe d’inversion des dépendances (« Dependency Inversion »)

<https://github.com/OpenClassrooms-Student-Center/ecrivez-du-code-python-maintenable-jeu-de-carte/tree/ebe1b0278830c3888092102a5931e0c5b996c57f>

Le principe d**’inversion des dépendances** a toujours l’air plus compliqué qu’il ne l’est réellement. Il dit que :

Les classes de haut niveau ne doivent pas avoir à changer juste parce que les classes de bas niveau changent.

Les classes de haut niveau sont les classes qui, habituellement, dirigent votre système. Elles doivent rester aussi stables que possible.

### Pourquoi utiliser l’inversion des dépendances ?

Prenons l’exemple des voitures. Les classes de haut niveau sont les éléments avec lesquels vous avez le plus d’interactions : le volant, l’accélérateur, et la pédale de frein. Ils disent aux classes d’implémentation de bas niveau (les pneus, le moteur, et les freins) ce qu’elles doivent faire.

Qu’arrive-t-il aux classes de haut niveau si vous transformez un moteur à essence en moteur électrique ? Rien ! Le conducteur continue à tourner, accélérer et freiner en utilisant les mêmes fonctionnalités. Si vous aviez enfreint ce principe, le passage au moteur électrique (une classe de bas niveau) vous forcerait à changer l’interface (une classe de haut niveau) pour le conducteur. C'est plutôt évident pour une voiture, mais c’est facile à rater dans du code.

Ce principe vous permet de vous assurer que ce sont les **classes de haut niveau** qui dirigent. Elles **définissent l’interface** à travers laquelle elles communiquent. Dans la voiture, vous faites accélérer le moteur en appuyant sur l’accélérateur. C’est au moteur de se conformer à ce standard.

Nous parlons de « niveaux » pour définir les « niveaux d’abstraction » dans le code. Pour faire simple, un niveau d’abstraction correspond à une **couche spécifique de détails**. Si par exemple une communication avec la base de données peut être une couche assez basse dans le code (puisqu’elle sera « cachée » par les modèles), l’interface utilisateur peut être considérée comme la couche la plus haute de l’application, puisqu’elle « cache » les autres couches en proposant une interface simple à utiliser.

### Comment appliquer l’inversion des dépendances à notre code ?

Dans notre jeu, la vue doit être « dirigée » par le contrôleur. C’est donc le contrôleur qui **définit** l’interface. Toute vue doit s’y conformer. Sinon, chaque fois que vous modifiez l’interface, le contrôleur devra être modifié pour se conformer à la vue (ce qui serait fait à l’envers).

Remarquez quand même qu’au chapitre précédent, nous avons dû modifier le contrôleur de façon significative pour pouvoir mettre à niveau la vue, afin qu’elle puisse gérer la diffusion ! Il semblerait donc que nous ne nous conformions pas encore au principe d’inversion des dépendances…

En réalité, la racine de notre problème vient du fait d’avoir donné trop d’informations au contrôleur sur la structure de la vue. Le déroulement du jeu n’est pas affecté par l’existence de différentes diffusions supplémentaires en cours, donc  Controller  ne doit pas contenir de code qui dépende de détails spécifiques de la vue.

Félicitations, vous avez un jeu de cartes qui fonctionne, et qui sera facile à maintenir toutes les fois où nous devrons ajouter de nouvelles fonctionnalités ! Nous pouvons simplement et directement changer tout aspect du jeu sans trop nous inquiéter de la façon dont nos changements risquent d’impacter d’autres parties de notre code. 😍

C’est le rêve pour écrire du code dans n’importe quel langage : d’avoir confiance en votre code, sachant qu’il est assez SOLID pour qu’un petit changement à un endroit ne provoque de cauchemars pour personne.

### En résumé

* L’inversion des dépendances dit que les concepts de haut niveau doivent communiquer à travers des abstractions de haut niveau. En d’autres termes : les classes de haut niveau ne doivent pas avoir à changer juste parce que les classes de bas niveau changent.
* Soyez attentif à ce que les classes n’en sachent pas trop sur les implémentations des autres classes.

## Évitez les pratiques de programmation STUPID

ous avez vu quelques bonnes approches d’implémentation de notre système. Intéressons-nous maintenant à l’autre côté du spectre : les idées qui **empirent les choses**. Vous devrez les **éviter** autant que possible. Heureusement, elles ont elles aussi leur acronyme, à l’instar de SOLID. Elles s’appellent **STUPID**. Vous entendrez peut-être des personnes les décrire comme des **anti-patterns** de design. Elles ressemblent aux design patterns, mais elles empirent votre code au lieu de l’améliorer !

La difficulté, lorsque l’on veut éviter ces approches, réside dans le fait qu’elles **semblent logiques sur le moment**. Vous avez besoin de faire fonctionner des éléments de code, et ces approches marchent. Au moins à court terme. Mais généralement, leur maintenabilité sur le long terme fait défaut.

Alors, évitez la tentation, et trouvez une façon d’appliquer plutôt les principes SOLID. La meilleure façon d’éviter les mauvaises habitudes, c’est d’en avoir conscience. Plongeons-nous dans ce sujet !

### Que désigne le « S » ? Le Singleton

Un Singleton est un objet qui garantit d’être la seule instance de son type ! Si vous en avez un, vous ne pouvez pas en créer un deuxième.

Le Singleton est considéré à la base comme un design pattern spécifique (encore un !). Cependant, étant très controversé, il finit par être considéré comme un “anti-pattern”, soit une erreur d’implémentation. 😯

Si le Singleton peut avoir son utilité, on l’évite généralement en programmation moderne.

Ah oui ? Pourquoi ?

* Il est difficile d’écrire des tests unitaires pour un Singleton (voir U pour Untestability – non-testabilité – ci-dessous).
* On ne peut pas sous-classer un Singleton.
* Il casse le O de SOLID – si les exigences changent et qu’il vous faut en fait une autre instance d’un Singleton, vous devrez modifier tout le code qui dépend du fait qu’il soit unique.
* Pour les situations où le Singleton est utile, il existe d’autres solutions plus SOLID.

Contrairement à d’autres langages de programmation, il est en fait un peu délicat d’implémenter cet antipattern en Python, donc au moins il est peu probable que vous tombiez dans ce piège par accident !

### Que désigne le « T » ? « Tight Coupling » (le couplage fort)

Le couplage fort se produit lorsque deux classes (ou modules) dépendent tellement l’une de l’autre que si vous apportez des modifications à l’une, vous devez souvent apporter des modifications à l’autre.

C'est ce qui rend votre code moins réutilisable (car vous devez également réutiliser tous les éléments qui accompagnent chaque classe fortement couplée), et en conséquence directe, plus difficile à tester.

### Que désigne le « U » ? « Untestability » (la non-testabilité)

Une classe peut être **difficile ou impossible à tester** pour de nombreuses raisons. Mais on en revient le plus souvent à un problème de **couplage fort** avec un autre composant. Si une classe a besoin de nombreuses dépendances pour fonctionner correctement, ça indique qu’il faut la réécrire. Le fait de tester un composant peut également être compliqué s’il **viole la responsabilité unique** et **fait trop de choses**.

### Que désigne le « P » ? « Premature Optimization » (l’optimisation prématurée)

L’optimisation prématurée désigne le fait de gérer un **problème que l’on anticipe** bien avant qu’il ne devienne **un problème**.

Mais… c’est plutôt une bonne chose, non ?

Pas toujours. Par exemple, à la fin de notre partie de cartes, nous devons remettre toutes nos cartes dans la pile puis battre le jeu.

Mais en réalité, nous n’avons aucune information sur l’ordre actuel des cartes (sauf pour celles que nous venons d’utiliser dans notre dernière partie) ! Nous n’avons donc pas besoin de battre l’ensemble du jeu – ce qui peut constituer un processus lent.

Nous pouvons plutôt, lorsque nous remettons les cartes dans la pile, nous contenter de les insérer à des emplacements aléatoires, grâce à un algorithme intelligent qui nous garantit un mélange aléatoire parfait (d’ailleurs, il est beaucoup plus difficile qu’il n’y paraît d’obtenir un mélange aléatoire parfait ici, mais je digresse !).

En réalité, on ne bat pas les cartes si fréquemment que ça. Par conséquent, l’optimisation de cet algorithme n’en vaut vraiment pas la peine – et, encore pire : nous avons écrit une quantité de code compliqué, que les futurs développeurs prendront plus longtemps à comprendre.

L’optimisation prématurée prend du temps et complexifie le code. Il se peut même que cette implémentation ne serve pas ! À l’inverse, l’acronyme [YAGNI](https://fr.wikipedia.org/wiki/YAGNI#:~:text=YAGNI%20(anglicisme%2C%20acronyme%20anglais%20de,n'est%20pas%20absolument%20n%C3%A9cessaire.) (you ain't gonna need it, ou “vous n’en aurez pas besoin”), conseille d’aller au plus simple, et de n’ajouter que les fonctionnalités qui sont absolument nécessaires. 😉

### Que désigne le « I » ? « Indescriptive Naming » (le nommage non descriptif)

Bien que cette pratique semble facile à éviter, elle se produit assez souvent. Pourquoi ça arrive ? Car, **au moment où** vous écrivez le code, le problème et la **solution sont logiques**. Imaginez que vous vous occupiez d’un rectangle. Vous nommez donc les variables de l’angle supérieur gauche x1 et y1. C’est logique.

Des mois plus tard, en regardant le code, vous (ou quelqu’un d’autre) voyez ces variables. Qu’est-ce que x1 ? Vous devez lire le code pour le découvrir. Si vous aviez nommé les variables  angle\_supérieur\_gauche\_x  et  angle\_supérieur\_gauche\_y  , vous auriez la réponse tout de suite.

En Python, vous pouvez encore simplifier les choses en utilisant les conventions de nommage de la PEP 8 dans votre code !

### Que désigne le « D » ? La duplication

La duplication constitue un piège dans lequel il est très facile de tomber. Vous avez besoin d’ajouter une nouvelle fonctionnalité. Elle doit fonctionner comme une fonctionnalité existante, mais un peu différemment. Que faites-vous ? Vous copiez, collez, et modifiez. Si vous adoptez cette approche de façon suivie, vous vous retrouverez avec du code dupliqué en de nombreux endroits. S’il faut modifier quelque chose de fondamental, tous ces emplacements copiés doivent être retrouvés et modifiés.

Est-ce que c’est toujours une mauvaise idée de copier-coller ?

Vous pouvez copier et coller lorsque vous devez mettre quelque chose en place rapidement. Malgré tout, il vous faudra y revenir et trouver une **meilleure solution à long terme**. Posez-vous les questions suivantes :

* Pourquoi y a-t-il autant de points communs entre ces deux éléments ?
* Est-ce que le code dupliqué peut être placé dans une fonction ou classe commune ?
* Puis-je extraire une interface et placer les éléments légèrement différents dans des implémentations différentes ?

Il n’y a pas de réponse unique qui fonctionne dans toutes les situations. L’idée principale, ici, est que toutes ces approches STUPID sont faciles à mettre en œuvre. Elles semblent logiques sur le moment. Les **problèmes qu’elles créent** ne **se manifestent** que **plus tard dans le projet**.

### En résumé

* L’acronyme STUPID désigne ces éléments :
* Singleton.
* Couplage fort (« Tight coupling »).
* Non-testabilité (« Untestability »).
* Optimisation prématurée (« Premature optimization »).
* Nommage non descriptif (« Indescriptive naming »).
* Duplication.
* Les approches STUPID conduisent à des conceptions de codage difficiles à maintenir et à tester.
* Il est facile de tomber dans des pièges STUPID, alors faites preuve de vigilance et posez-vous les bonnes questions SOLID.

# [Devenez un expert de Git et GitHub](https://openclassrooms.com/fr/courses/7688581-devenez-un-expert-de-git-et-github)