LES GUIDES DE GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°77
France METRO : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ cad — MAR : 130 MAD

ET MAINTENANT, J'APPRENDS LA ...

PROGRAMMATION ORIENTÉE OBJET

... EN 6 JOURS !

COMPATIBLE WINDOWS / MAC OS X / LINUX

zéro PRÉREQUIS

INTRODUCTION

Apprendre les bases de la programmation impérative

VOTRE SEMAINE

JOUR 1 : Créer une classe
JOUR 2 : Agréger plusieurs objets
JOUR 3 : Définir des objets à partir d'objets existants
JOUR 4 : Modifier un objet pour le rendre générique
JOUR 5 : Utiliser des objets pour créer un programme
JOUR 6 : Réfléchir aux évolutions possibles

INDEX

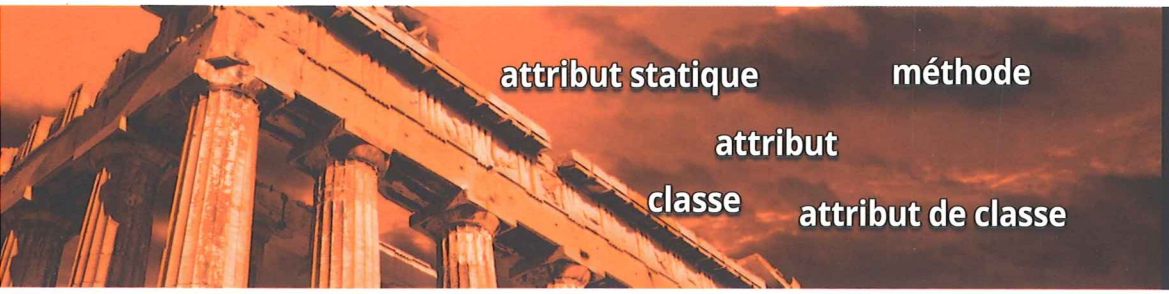
Le récapitulatif détaillé de toutes les notions et instructions

+ BONUS : Une annexe des erreurs les plus courantes

Édité par Les Éditions Diamond
L 15066 - 77 H - F: 12,90 € - RD
www.ed-diamond.com

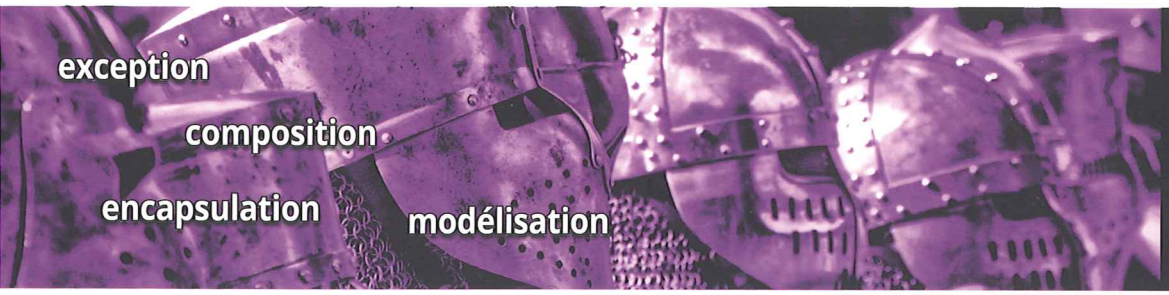
INTRODUCTION :

Apprendre les bases de la programmation impérative



JOUR 1 :

Créer une classe



JOUR 2 :

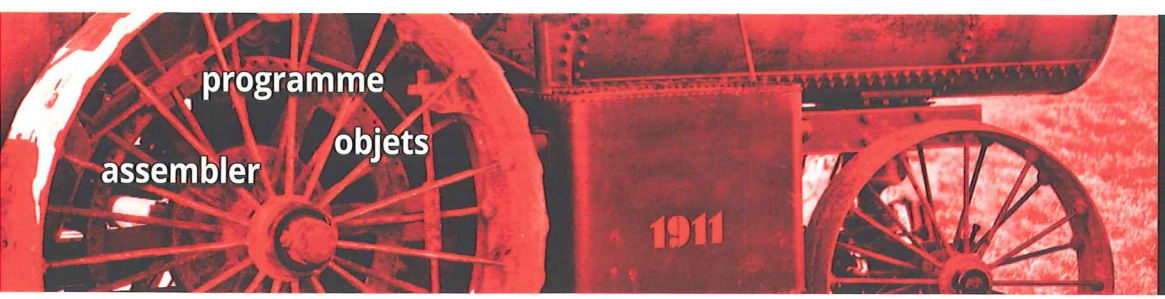
Agréger plusieurs objets

JOUR 3 :

Définir des objets à partir d'objets existants

JOUR 4 :

Modifier un objet pour le rendre générique



JOUR 5 :

Utiliser des objets pour créer un programme

JOUR 6 :

Réfléchir aux évolutions possibles

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

B.P. 20142 / 67603 Sélestat Cedex

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Illustrations : Maryan Sidorkiewicz

Conception graphique : Kathrin Scali, Caroline Massing

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

PRÉFACE

L'apprentissage de la programmation n'est pas chose aisée... d'autant plus lorsque l'on veut aborder la programmation orientée objet dans laquelle la façon de penser le code est un peu particulière. Mais vous avez décidé de vous lancer dans l'aventure et l'on ne peut que vous en féliciter !

Ce hors-série a été pensé pour des débutants et en ce sens de nombreuses notions, qui pourront paraître évidentes à certains lecteurs, seront expliquées tout au long de l'avancement d'un projet de programmation d'un petit jeu de cartes. Dans la même optique, deux annexes vous permettront de retrouver facilement les définitions des instructions et des notions essentielles. Enfin, une annexe contenant une définition des erreurs les plus courantes devrait vous être d'une grande aide pour résoudre seul(e) vos problèmes. En effet, le plus important n'est pas de savoir programmer sans erreur : si vous rencontrez un informaticien qui affirme en être capable, sachez qu'il ment. Nous faisons tous des erreurs pour diverses raisons : moment d'inattention, méconnaissance du langage, réflexion erronée, etc. Par contre, si vous savez identifier une erreur et la corriger rapidement, le monde de la programmation s'ouvrira à vous, et vous pourrez réaliser tout ce que vous souhaitez (en restant un tant soit peu raisonnable au début...).

La programmation orientée objet n'est pas une programmation facile et notre objectif sera donc mesuré : faire en sorte que vous puissiez acquérir en une semaine une « culture » de l'orienté objet, les termes et notions importants et que vous sachiez programmer de petits projets. Il faut être réaliste, vous devrez faire fructifier tout ce que vous pourrez apprendre dans ce hors-série en l'appliquant à la création de nouveaux projets pour maîtriser ce type de programmation. Bien entendu, il faudra rester raisonnable quant à la taille de ces projets, car les difficultés surgiront toujours là où on vous ne les attendez pas ! Mais c'est au prix de la résolution de tous ces petits problèmes que l'on progresse réellement.

Nous avons non seulement voulu que ce hors-série s'adresse à tous en partant du principe qu'aucun prérequis n'était fondamentalement indispensable, mais également que vous puissiez l'utiliser facilement quel que soit le système d'exploitation que vous employez. Ainsi, absolument tout ce qui est indiqué dans ce hors-série fonctionne sur les trois plateformes Windows, Linux et Mac OS X !

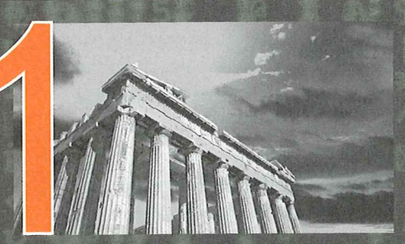
Je vous laisse maintenant vous plonger dans l'apprentissage des divers aspects de la programmation orientée objet. La programmation en elle-même peut-être vue comme un jeu... donc amusez-vous bien !

Tristan Colombo

Sommaire

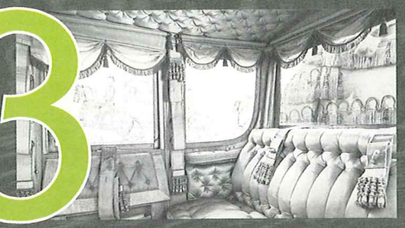
GNU/Linux Magazine Hors-Série N°77

INTRODUCTION



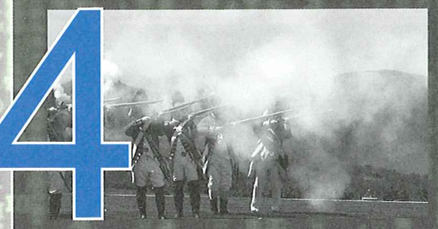
JOUR 1

JOUR 2

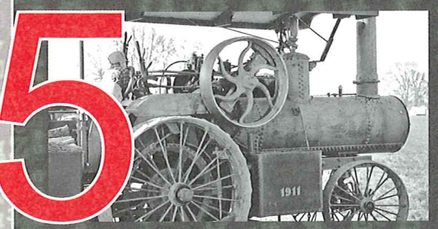


JOUR 3

PROGRAMMATION ORIENTÉE OBJET

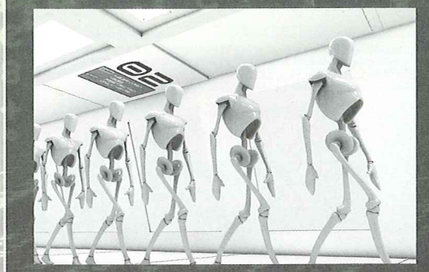


JOUR 4

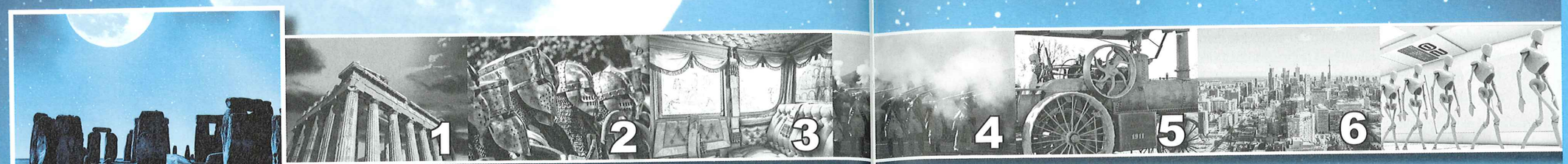


JOUR 5

JOUR 6



INDEX



INTRODUCTION

LES RÈGLES DU JEU

La Programmation Orientée Objet, ou POO, est une technique de programmation un peu particulière. Avant de pouvoir se lancer dans son apprentissage, il faut s'assurer qu'un certain nombre d'outils sont installés. C'est l'objectif principal de cette partie.

Vous avez décidé d'apprendre à programmer d'une manière un peu particulière : en « orienté objet ». Pourquoi utiliser la programmation orientée objet ? S'agit-il d'un choix délibéré ou simplement parce que l'on vous en a parlé ? Quoi qu'il en soit, nous verrons ensemble l'utilité de ce type de programmation.

En effet, il existe plusieurs « types » de programmation (on pourra parler d'architectures), mais les plus courants sont la programmation dite impérative et la programmation dite orientée objet. Certains langages utilisent exclusivement une architecture alors que d'autres acceptent les deux types de programmation. C'est le cas du langage Python où les deux formes seront acceptées. C'est d'ailleurs avec ce langage que nous travaillerons pour sa simplicité d'utilisation et son aspect multiplateforme : un programme écrit en Python fonctionnera aussi bien sous Windows que sous Mac OS X ou Linux. Dans cette partie, nous aborderons les concepts de base de ce langage qui nous permettront d'aborder la suite en toute sérénité.

Pour être certains de travailler dans les meilleures conditions, nous consacrerons également un peu de temps à l'installation des outils dont nous aurons besoin pour cet apprentissage, apprentissage que nous effectuerons en une semaine en réalisant un projet qui sera présenté au cours de cette partie introductive (mais on peut déjà dire qu'il s'agira d'un petit jeu). Tout au long de notre parcours dans le monde de la Programmation Orientée Objet, nous serons accompagnés par un petit pingouin qui a parfois quelques difficultés à comprendre certains concepts... ne suivez pas forcément toujours ces exemples et surtout ne vous découragez pas ! L'apprentissage de la programmation peut être réalisé rapidement, mais pour la maîtrise il faut du temps et des erreurs : ce sont vos erreurs qui vous apprendront le plus de choses. Vous passerez du temps à chercher pourquoi un programme ne fonctionne pas et le fait de trouver la solution par vous-même ancrera définitivement la solution dans votre cerveau. C'est pour cette raison qu'aucun des codes présentés dans ce mook n'est disponible en ligne : vous devez faire vos propres erreurs pour apprendre. Même une erreur de copie peut être intéressante et vous apprendre beaucoup ! Bien entendu, pour que cette méthode fonctionne, il faut que vous soyez certains que les programmes présentés dans ce mook soient exacts... et c'est le cas : ils ont tous été testés ! Si vous rencontrez une erreur, reportez-vous à l'annexe « Les erreurs courantes » et vous devriez comprendre rapidement pourquoi votre programme ne fonctionne pas !

Allez, il est temps de se préparer à une semaine d'apprentissage à la programmation orientée objet !

À retenir

Une langue est l'ensemble des règles (grammaticales et syntaxiques) qui permettent à deux individus de communiquer, de se comprendre l'un l'autre. Un langage de programmation est en quelque sorte la « langue » des ordinateurs : l'ensemble des règles qui permettent à un humain de communiquer avec un ordinateur.

1. LES DIFFÉRENTS LANGAGES DE PROGRAMMATION

Comme je le disais, il existe plusieurs types de programmation et de nombreux langages. La programmation impérative est une programmation qui fera intervenir des structures fondamentales :

- affectation d'une valeur à une variable : pour enregistrer une valeur (numérique ou textuelle) dans la mémoire d'un ordinateur, on utilise des **variables**. On associe à une valeur donnée un nom que l'on appelle `identifiant` et qui permettra de retrouver la valeur en mémoire sans avoir à connaître un code bizarroïde du type `0x7f315ce442a8`. Dans l'expression `a = 5`, `a` est le nom de la variable (l'identifiant) et `5` est sa valeur. Une fois qu'une variable a été définie, à chaque fois que son nom apparaît c'est comme si l'on avait indiqué directement sa valeur (contenue en mémoire).

- structures de test : ce sont des instructions qui vont permettre de créer un embranchement, d'exécuter telle partie du programme ou telle autre en fonction d'une condition. Ces structures permettront de coder des comportements du type : « si la variable a vaut 5, alors exécute la partie A du programme, sinon exécute la partie B ».

- structures de boucle : plutôt que d'écrire n fois les mêmes instructions, on utilise des structures de boucle qui permettent de préciser les instructions à répéter et le nombre de répétitions (ou la condition qui fait que la répétition doit cesser).

- fonctions : certaines instructions nécessitent d'être employées plusieurs fois en changeant seulement certaines valeurs et plutôt que de tout réécrire, on utilise des fonctions. Une fonction est un ensemble d'instructions dont le résultat à l'exécution va varier en fonction d'un ou de plusieurs paramètres. Par exemple, si l'on détermine la méthode qui permet de calculer le nom du énième jour suivant le jour courant : `n` sera le paramètre indiquant le nombre de jours à ajouter et nous utiliserons un seul et même code pour calculer le nom du jour suivant (`n = 1`), du jour précédent (`n = -1`), etc.

La programmation orientée objet est une évolution de la programmation impérative dans laquelle tous les éléments précédents seront assemblés sous forme d' « objets » avec pour objectif de réutiliser le plus possible un code ayant déjà été écrit et de créer des objets à partir d'autres objets. Ne prenez pas peur, nous reviendrons bien sûr petit à petit sur toutes ces notions.

Parmi les langages impératifs, on peut citer le langage C et parmi les langages orientés objet le Java. Le langage Python admet les deux types de programmation. Comme la programmation orientée objet va chercher ses sources dans la programmation impérative, nous verrons ces bases en Python de manière à pouvoir nous focaliser sur des concepts essentiels de programmation orientée objet.

Il faut enfin savoir que certains langages sont compilés et d'autres interprétés. Un langage compilé fera intervenir un code source (le code que vous aurez écrit) et passera par une étape de compilation transformant ce code en code binaire compris par votre machine. Avec ces langages, si vous effectuez la compilation sous un système d'exploitation Windows avec une architecture 32 bits, vous ne pourrez pas exécuter votre programme sur une autre architecture (il faudra réaliser une autre compilation en prenant en compte les caractéristiques de la machine sur laquelle on souhaite pouvoir exécuter le programme). Un langage interprété sera, lui, en quelque sorte compilé à la volée, au cours de son exécution. Donc les langages de cette famille sont multiplateformes : vous écrivez un code une fois et vous pouvez l'exécuter sur n'importe quel ordinateur, à condition que ce dernier possède bien l'interpréteur (le programme qui comprend le langage et le traduit en un code binaire compréhensible par l'ordinateur).

À retenir

Un code source est l'ensemble des instructions qui constitue un programme. Le code source est enregistré dans un ou plusieurs fichiers.

À retenir

Un `langage compilé` traduit le code source en code binaire compréhensible par l'ordinateur sur lequel la compilation a lieu. Un tel code dépend du système d'exploitation et de l'architecture du microprocesseur (32 bits, 64 bits, ARM, etc.) de l'ordinateur.

À retenir

Un langage interprété est « traduit » en code binaire au fil de son exécution. Les programmes écrits en utilisant de tels langages ne sont pas dépendants du système ou de l'architecture sur lesquels ils sont exécutés. On dit qu'ils sont multiplateformes : vous pouvez développer sous Windows et distribuer votre code à des gens travaillant sous Linux ou Mac OS X (et réciproquement).

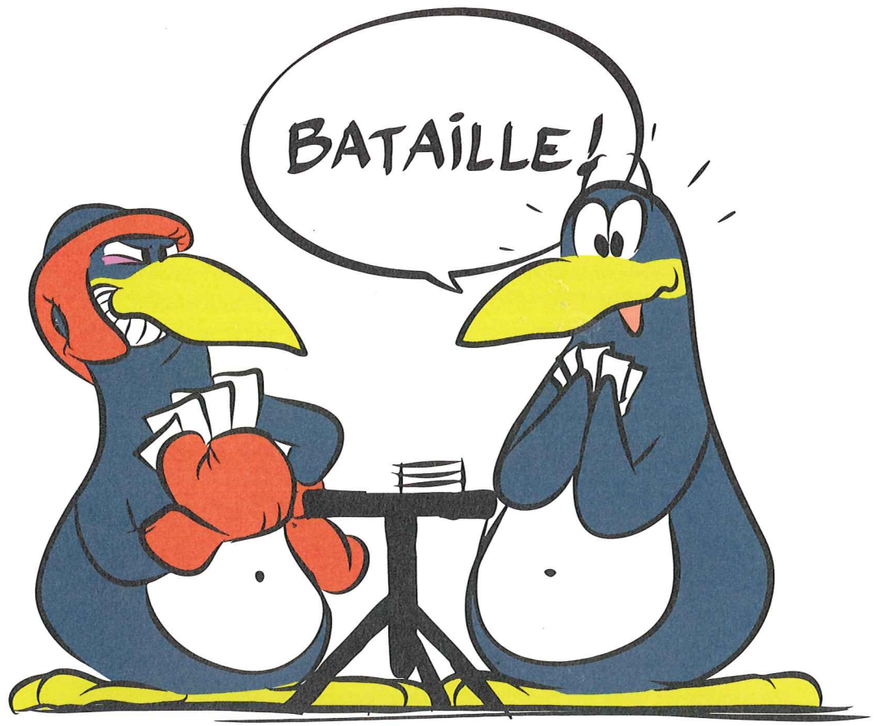
2. LE PROJET : UN JEU DE CARTES

En une semaine, nous allons concevoir un jeu de cartes. Nous partirons sur un jeu très simple : le jeu de bataille. Vous verrez que grâce à la programmation orientée objet vous pourrez faire évoluer le projet vers tout type de jeux de cartes.

Pour rappel, voici les règles du jeu de bataille : il s'agit d'un jeu à deux joueurs où l'on commence par distribuer l'ensemble d'un jeu de cartes (52 cartes) aux joueurs de manière à ce que chaque joueur possède le même nombre de cartes, c'est-à-dire 26. Les joueurs ne regardent pas les cartes et à chaque tour, ils retournent la carte se trouvant sur le dessus du paquet. Le joueur possédant la carte la plus forte (en fonction des points qui lui sont attribués dans l'ordre décroissant As, Roi, Dame, Valet, 10, ..., 2) récupère les deux cartes et les place sous son tas. Si les deux cartes sont de valeur égale, on dit qu'il y a bataille : chaque joueur place la carte du dessus de son tas à l'envers sur sa première carte, puis place une nouvelle carte par dessus. Le joueur ayant la carte la plus forte récupère l'ensemble des cartes (en cas de nouvelle égalité, on recommence). La partie se termine lorsque l'un des deux joueurs n'a plus de carte (et ce dernier a perdu).

Pour réaliser ce jeu, nous passerons par plusieurs étapes. Voici le planning de la semaine :

- Introduction : Nous y sommes, c'est l'étape de préparation qui nous permet de poser les bases de notre projet ;
- Jour 1 : Création d'une carte ;



À retenir

Python est régulièrement mis à jour, enrichi. Au moment où vous installerez Python, il est possible qu'une nouvelle version soit apparue. Assurez-vous simplement que vous installiez un Python 3.x.x : les deux derniers chiffres n'ont pas d'importance, sélectionnez la dernière version.

- Jour 2 : Assemblage de cartes pour constituer un jeu de cartes ;
- Jour 3 : Gestion d'un paquet de cartes ;
- Jour 4 : À partir des éléments précédents, déclinaison de plusieurs jeux de cartes différents ;
- Jour 5 : Jouer enfin contre l'ordinateur en assemblant les différents éléments créés précédemment ;
- Jour 6 : Réflexions diverses sur les améliorations possibles de notre jeu.

Les premiers jours seront les plus denses et au fil de l'avancée du projet vous verrez que tout le travail effectué précédemment s'assemblera naturellement, nécessitant moins d'efforts et de temps.

3. LES OUTILS

Pour pouvoir développer en Python nous aurons bien sûr besoin de Python, mais également d'un éditeur de texte permettant de taper le code source de nos programmes avec un maximum de confort.

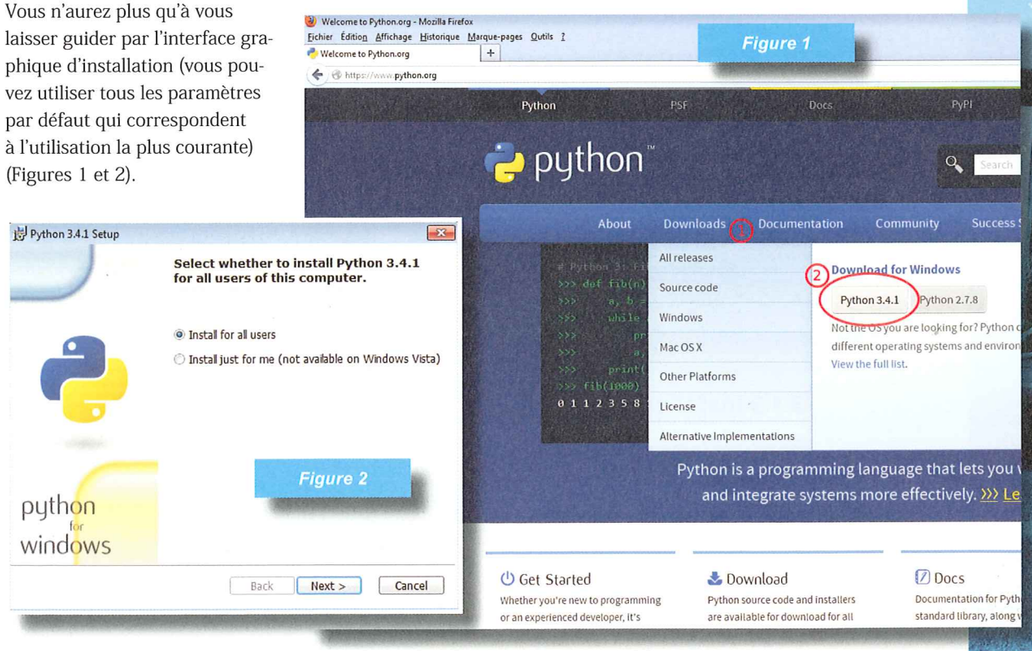
3.1 Installation de Python

Nous utiliserons la dernière version de Python, la version 3. En fonction de votre système d'exploitation, reportez-vous à la section qui vous intéresse.

3.1.1 Installer Python sous Windows

Rendez-vous sur la page **http://www.python.org** à l'aide de votre navigateur favori et sélectionnez ***Downloads***, puis cliquez sur le bouton ***Python 3.4.1***. Acceptez le téléchargement du fichier `python-3.4.1.msi`, puis rendez-vous dans le répertoire `Téléchargements` et double-cliquez sur ce fichier.

Vous n'aurez plus qu'à vous laisser guider par l'interface graphique d'installation (vous pouvez utiliser tous les paramètres par défaut qui correspondent à l'utilisation la plus courante) (Figures 1 et 2).



3.1.2 Installer Python sous Linux Ubuntu

Il existe de nombreuses versions (on parle de distributions) de Linux et de nombreux environnements graphiques. Je ne détaillerai ici l'installation de Python que sur la distribution la plus adaptée aux débutants : Ubuntu avec l'environnement graphique par défaut Unity.

Commencez par ouvrir la logithèque Ubuntu en cliquant sur l'icône en forme de sac sur le menu du bureau (par défaut à gauche de votre écran).

Utilisez ensuite la barre de recherche et tapez `python3` (sans espace). Vous sélectionnerez alors dans les résultats l'entrée ***Python (version 3.4)***. Si vous disposez d'Ubuntu 14.04, ce paquet est installé par défaut et vous verrez une petite coche blanche sur fond vert apparaître en bas à droite de l'icône du paquet (et si vous cliquez sur le lien, le système vous indiquera qu'il est installé). Dans le cas contraire, vous devrez cliquer sur le bouton ***Installer*** qui apparaîtra.

Comme nous utiliserons un éditeur un peu particulier, c'est tout ce qu'il y a à faire pour l'instant.

Figure 3

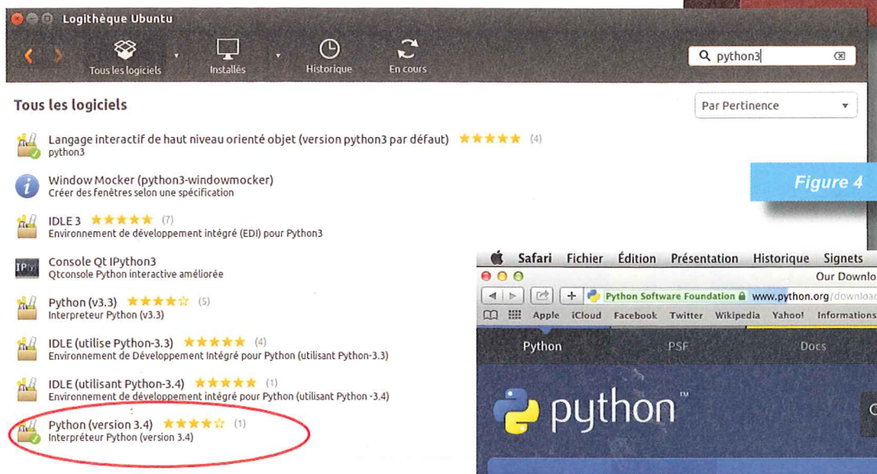


Figure 4

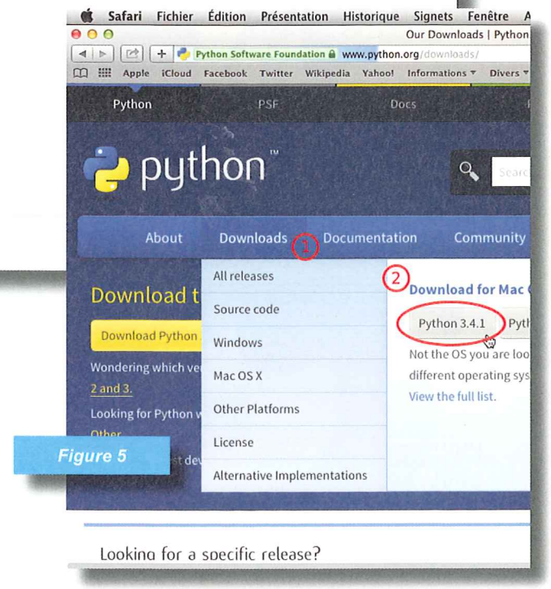


Figure 5

3.1.3 Installer Python sous Mac OS X

À l'aide d'un navigateur, rendez-vous sur la page **http://www.python.org** et sélectionnez ***Downloads***, puis cliquez sur le bouton ***Python 3.4.1***. Vous téléchargerez alors le fichier `python-3.4.1-macosx10.6.dmg`.

Figure 6

Double-cliquez ensuite sur l'icône de cette archive : une fenêtre apparaîtra montrant les différents fichiers qu'elle contient et notamment le fichier `Python.mpkg` sur lequel vous devez double-cliquer (il faudra accepter le fait d'exécuter une application d'un développeur non identifié). Suivez ensuite les instructions à l'écran pour finaliser votre installation.

3.2 Le choix de l'éditeur de code

Nous avons installé Python, donc nous pouvons maintenant exécuter des programmes écrits dans ce langage. Pour écrire un programme, un éditeur de texte suffit, mais nous allons utiliser ici un outil particulièrement adapté au développement de programmes : un éditeur de code. Et nous allons même faire mieux, car nous allons utiliser un Environnement de Développement Intégré !

Peut-être n'avez-vous jamais entendu parler d'Environnement de développement Intégré, que l'on abrège en IDE (sur la base de la dénomination anglaise) ? Il s'agit d'un logiciel permettant d'écrire du code source (comme un éditeur de texte), mais qui possède en plus des fonctionnalités de développement intégrées :

- la coloration syntaxique permettant de lire le code plus facilement : les variables apparaissent d'une certaine couleur, les mots-clés d'une autre couleur, etc.
- l'auto-complétion : commencez à taper le début d'une instruction et une liste apparaîtra vous permettant de ne pas avoir à taper toute l'instruction (ce qui peut être intéressant si vous ne vous rappelez plus exactement de son nom...) ;
- la documentation du langage : en sélectionnant une instruction, une bulle d'information apparaît ;
- la navigation parmi les fichiers du projet ;
- le débogage intégré ;
- et tout un tas d'outils relatifs au développement qui sont soit intégrés par défaut, soit qui peuvent être ajoutés par la suite.

Nous allons utiliser Eclipse comme IDE. Ce logiciel a l'avantage d'être codé en Java et donc d'être entièrement multiplateforme et, en lui ajoutant l'extension PyDev, il contient toutes les fonctionnalités utiles pour développer en Python. Dans cette partie, je ne ferai que décrire l'installation de cet IDE sur les différentes plateformes. Nous apprendrons à l'utiliser efficacement au fur et à mesure de nos progrès dans la POO avec Python.

ENVIRONNEMENT DE DÉVELOPPEMENT INTÉGRÉ

Un Environnement de Développement Intégré (EDI ou IDE pour *Integrated Development Environment*) est un logiciel regroupant un grand nombre de fonctionnalités nécessaires pour développer des programmes. Normalement, un IDE fournit tous les outils permettant de mener à terme un projet depuis la phase de conception jusqu'à la mise en production (moment où le logiciel est considéré comme pouvant être utilisé).

À retenir

Au moment où ces lignes sont écrites, la dernière version d'Eclipse disponible est Eclipse Luna (4.4.1). Vous pouvez installer toute version supérieure, le fonctionnement restera identique.

3.2.1 Installer Eclipse sous Windows

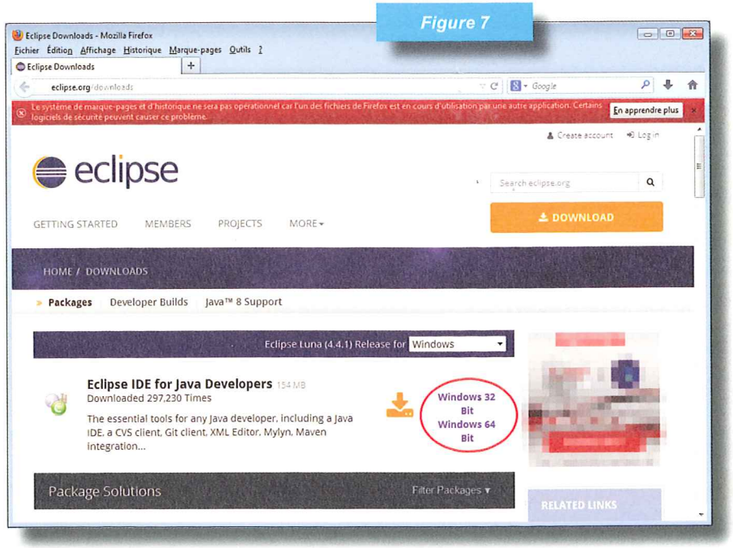
Rendez-vous sur la page **http://www.eclipse.org/downloads** à l'aide d'un navigateur Web et téléchargez le fichier `eclipse-java-luna-SR1-win32.zip` en cliquant sur le lien correspondant à votre type d'architecture (32 ou 64 bits).

Avant de poursuivre, nous devons installer Java 7 (si c'est déjà fait, vous pouvez sauter cette étape). Pour cette installation, rendez-vous sur la page **http://www.java.com/fr/download** et cliquez sur le bouton ***Téléchargement gratuit de Java***, puis sur le bouton ***Accepter et lancer le téléchargement gratuit***. Vous obtiendrez un fichier `jxpiinstall.exe` qu'il faut exécuter. Vous n'aurez plus qu'à suivre les instructions à l'écran.

Allez ensuite dans le répertoire `Téléchargements` où doit se trouver le fichier `eclipse-java-luna-SR1-win32.zip`, puis effectuez un clic droit sur ce dernier et sélectionnez ***Extraire tout...*** dans le menu contextuel. Choisissez l'emplacement d'extraction, puis cliquez sur le bouton ***Extraire***.

Pour lancer Eclipse, il suffit de cliquer sur l'icône comportant un croissant jaune masqué par un disque mauve avec trois rayures blanches et se situant

Figure 7



dans le répertoire d'extraction. Vous pouvez bien sûr créer un raccourci sur le bureau en effectuant un clic droit et en sélectionnant ***Nouveau raccourci*** dans le menu contextuel.

Pour finaliser l'installation d'Eclipse, rendez-vous en section 3.2.4 où vous trouverez les instructions qui sont désormais communes aux trois plateformes (Figure 7).

3.2.2 Installer Eclipse sous Linux Ubuntu

Il y a deux façons d'installer Eclipse sous Linux Ubuntu : depuis les dépôts, auquel cas vous ne disposerez pas de la dernière version, ou depuis le site officiel d'Eclipse.

3.2.2.1 Installation depuis les dépôts

C'est l'installation la plus simple et bien que la version d'Eclipse présente ne soit pas la dernière, elle est entièrement compatible avec tout ce qui sera dit dans ce mook.

Pour installer Eclipse (version 3.8.1), ouvrez la logithèque Ubuntu et tapez `eclipse` dans la barre de recherche, sélectionnez l'entrée ***Eclipse***, puis cliquez sur le bouton ***Installer***.

Une nouvelle icône (un croissant jaune masqué par un disque mauve avec trois rayures blanches) apparaîtra sur le menu du bureau à gauche et en cliquant dessus vous lancerez Eclipse.

La suite de la configuration est la même pour toutes les plateformes et vous pourrez trouver les instructions en section 3.2.4.

Figure 8

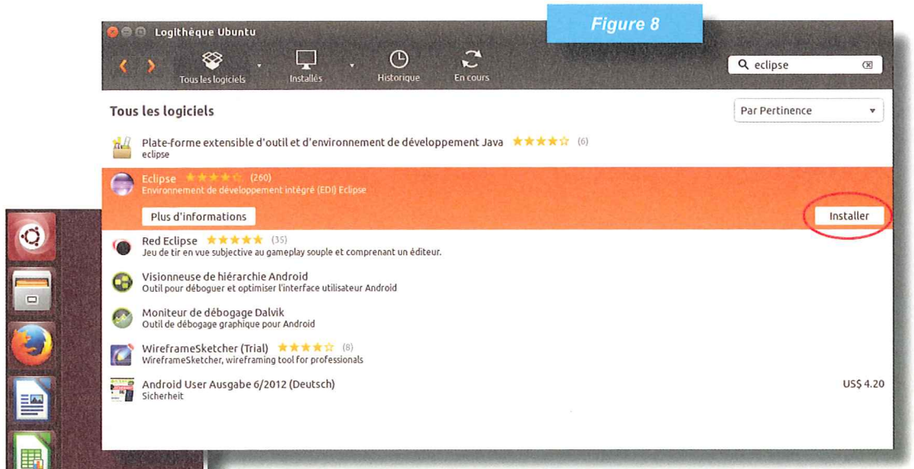
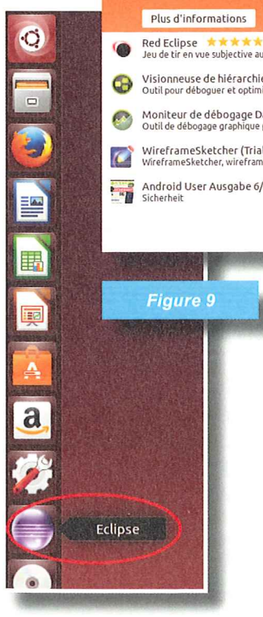


Figure 9



3.2.2.2 Installation manuelle depuis le site officiel

Pour obtenir la dernière version d'Eclipse, il faudra aller sur la page **http://www.eclipse.org/downloads** et sélectionner le lien correspondant à votre architecture (32 bits ou 64 bits) pour la version **Eclipse IDE for Java Developers**. Vous téléchargerez alors un fichier `eclipse-java-luna-SR1-linux-gtk.tar.gz`.

Il faut ensuite installer Java 7 pour pouvoir exécuter Eclipse avec PyDev. Ouvrez la logithèque Ubuntu et saisissez `Java` dans la zone de recherche. Sélectionnez ensuite ***Moteur OpenJDK Java 7*** et cliquez sur le bouton ***Installer***.

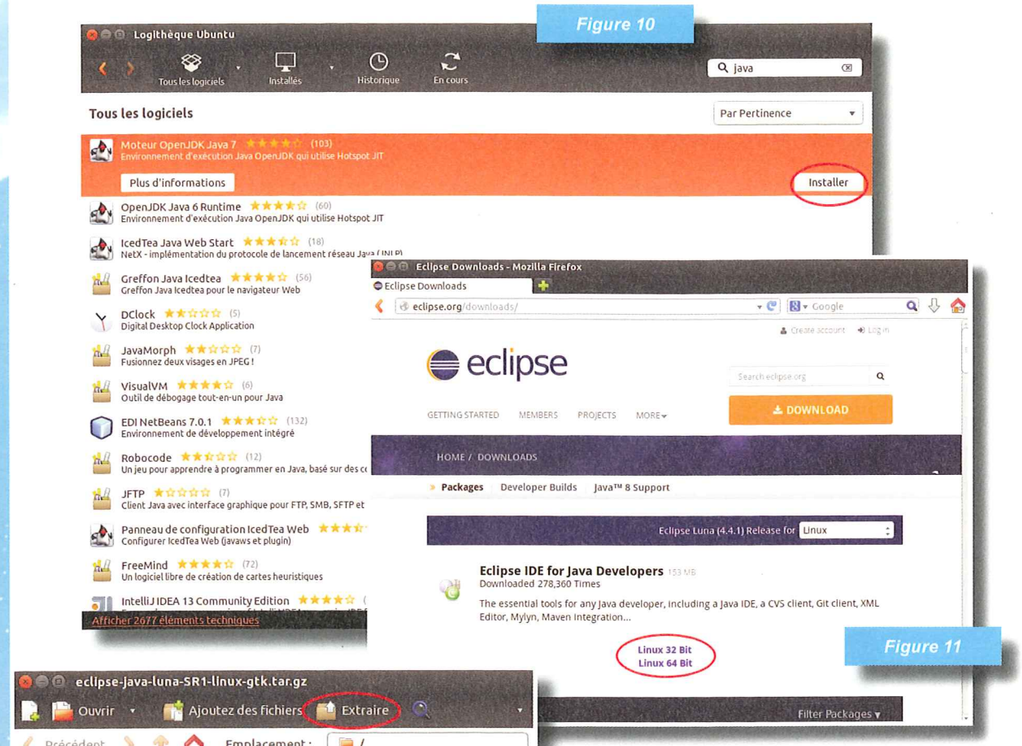


Figure 10

Figure 11

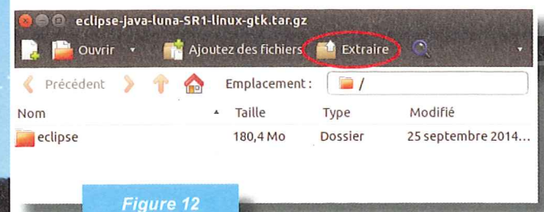


Figure 12

Rendez-vous dans le répertoire `Téléchargements` et double-cliquez sur le fichier `eclipse-java-luna-SR1-linux-gtk.tar.gz` pour le décompresser et cliquez sur le bouton ***Extraire*** pour enregistrer le répertoire `eclipse` sur votre disque dur. Sélectionnez ensuite un répertoire dans lequel vous souhaitez installer le programme (ça peut être votre répertoire personnel) et cliquez à nouveau sur ***Extraire***. Pour lancer Eclipse, vous n'aurez plus qu'à vous rendre dans ce répertoire avec votre navigateur de fichiers et à cliquer sur l'icône du programme `eclipse` (un carré avec des engrenages).

Pour la suite de la configuration, commune à toutes les plateformes, rendez-vous en section 3.2.4.

3.2.3 Installer Eclipse sous Mac OS X

Ouvrez un navigateur et rendez-vous sur la page **http://www.eclipse.org/downloads**. Téléchargez la version **Eclipse IDE for Java Developers** (la première sur la page) et cliquez, en fonction de l'architecture de votre microprocesseur, sur **Mac OS X 32 Bit** ou **Mac OS X 64 Bit** (attention, vous devez posséder au moins Mac OS X 10.5 pour installer Eclipse). Le fichier téléchargé sera `eclipse-java-luna-SR1-macosx-cocoa.tar.gz`.

Cliquez sur ce fichier pour décompresser l'archive. Dans le répertoire `Téléchargements`, vous verrez apparaître un nouveau répertoire nommé `eclipse`. Prenez-le et déplacez-le dans le répertoire de vos `Applications`.

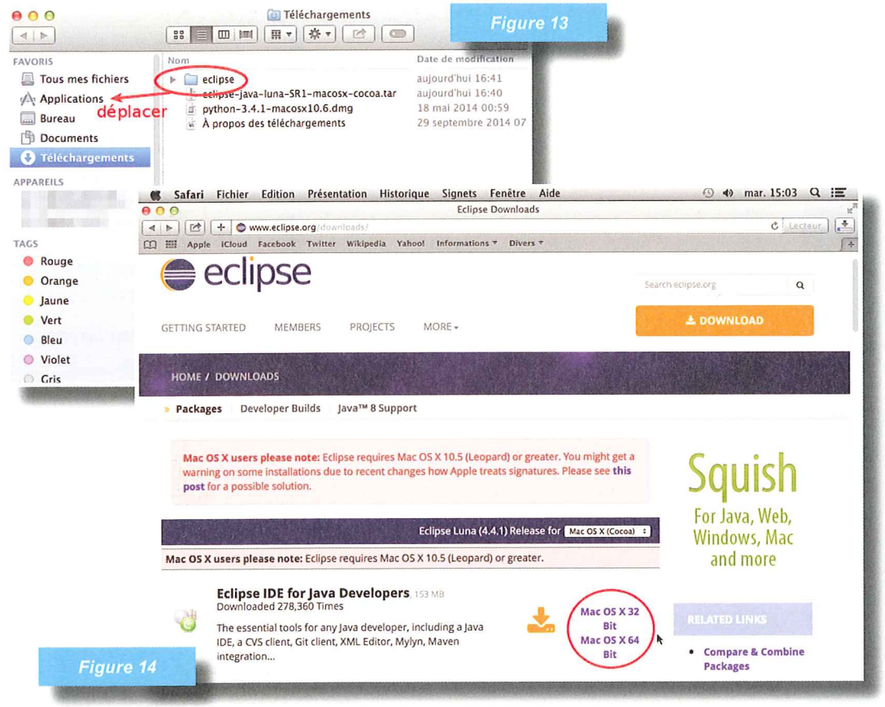


Figure 13

Figure 14

Pour lancer Eclipse, rendez-vous dans le répertoire `eclipse` et double-cliquez sur le programme Eclipse (avec l'icône représentant une éclipse : un croissant jaune masqué par un disque mauve avec trois rayures blanches). Si vous souhaitez pouvoir lancer ce programme simplement, faites un glisser/déposer de ce programme dans votre dock en bas d'écran. Après avoir confirmé que vous souhaitiez exécuter ce programme provenant d'internet, le système vous proposera d'installer Java SE 6 pour pouvoir utiliser Eclipse. Répondez bien sûr par l'affirmative en cliquant sur le bouton ***Installer***.

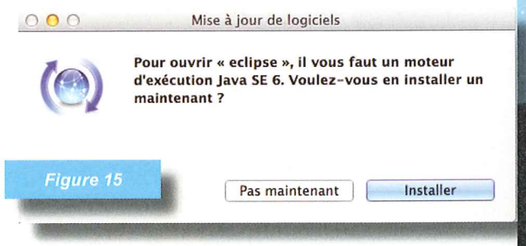


Figure 15

Comme l'extension PyDev nécessite au moins Java 7, allez dans ***Préférences Systèmes*** (dans le dock) et cliquez sur l'icône ***Java***, puis assurez-vous que vous avez bien effectué la mise à jour.

À la fin de l'installation de Java, Eclipse démarrera. La fin de la procédure étant commune aux trois plate-formes, elle est décrite dans la section suivante.

3.2.4 Fin d'installation d'Eclipse

Vous devrez créer un espace de travail (*workspace*) qui sera un répertoire qui contiendra l'ensemble des projets. Choisissez ce que le logiciel vous propose par défaut ou sélectionnez le répertoire qui vous convient.

Un écran apparaît : il s'agit de l'écran d'accueil qui peut être un peu différent en fonction des plateformes. Cliquez sur le bouton ***Workbench*** pour accéder à l'espace de travail.

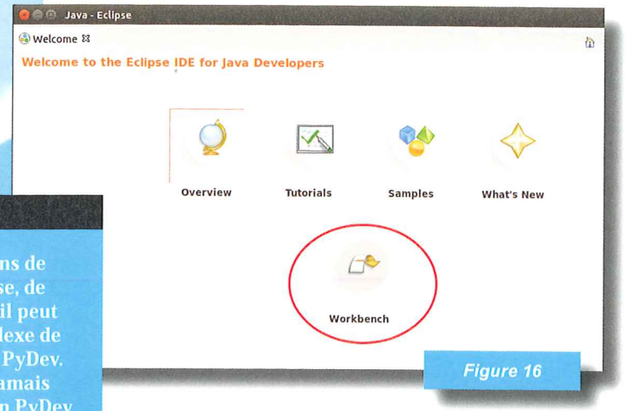


Figure 16

Nous allons maintenant installer l'extension PyDev. Pour cela, dans la barre de menu supérieure, sélectionnez ***Help > Install New Software...*** puis, sur l'écran suivant, cliquez sur le bouton ***Add...*** et ajoutez le site :

- Name : `Pydev and Pydev Extensions`
- Location : **http://pydev.org/updates**.

Après validation, une nouvelle fenêtre apparaîtra indiquant les extensions fournies par le site que vous venez d'ajouter. Sélectionnez PyDev en cochant la case appropriée, puis cliquez sur le bouton ***Next***. Suivez ensuite les directives à l'écran en acceptant les diverses licences et autres certificats.

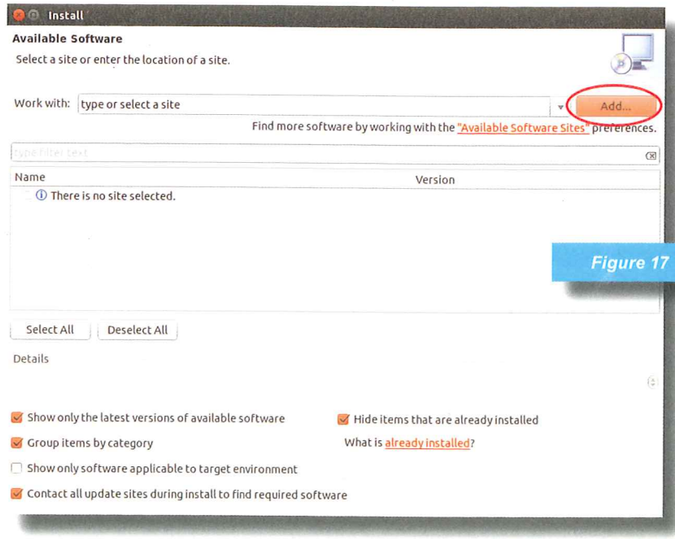


Figure 17

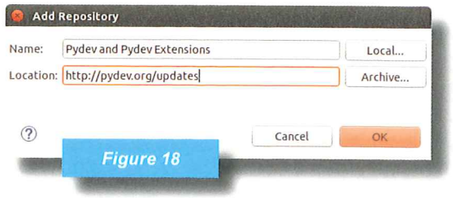


Figure 18

Pour achever l'installation de PyDev, Eclipse vous proposera un redémarrage qu'il faudra bien évidemment accepter.

Pour l'instant Eclipse est configuré pour des développements en Java comme vous pouvez le remarquer en haut et à droite de la fenêtre avec la présence du nom Java (dans le jargon des IDE, on parle de la « perspective » Java : l'ensemble des outils associés au langage

À retenir

Suivant les versions de Mac OS X, d'Eclipse, de PyDev et de Java, il peut être un peu complexe de parvenir à lancer PyDev. Si vous ne voyez jamais apparaître l'option PyDev dans les perspectives, avant d'abandonner, installez la dernière version de PyDev en version 2.x :

- Allez dans ***Help > Install New Software...*** et cliquez sur le lien ***already installed***, puis sélectionnez ***PyDev*** et cliquez sur le bouton ***Uninstall...*** ;
- Après redémarrage d'Eclipse, retournez dans ***Help > Install New Software...*** et décochez ***Show only the latest versions of available software***. Sélectionnez ensuite PyDev 2.8.2 et effectuez l'installation comme décrite précédemment.

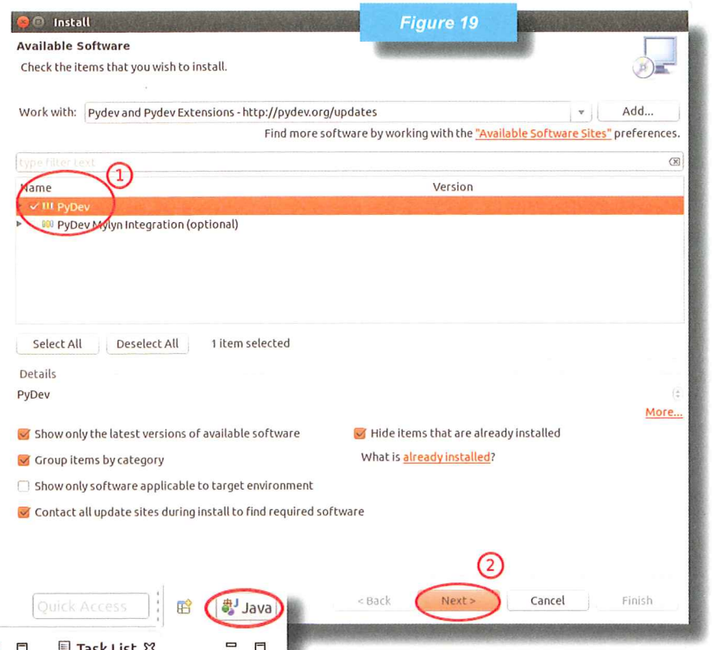


Figure 19

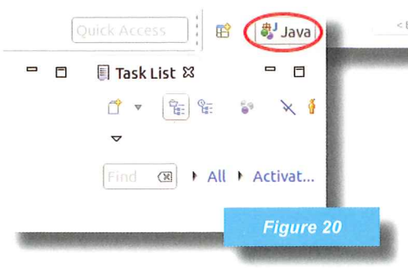


Figure 20

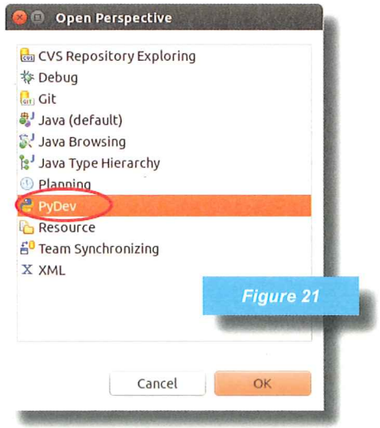


Figure 21

Java est accessible simplement). Pour passer en mode de développement Python, il faut activer la perspective PyDev. Pour cela, cliquez sur le petit bouton situé sur la gauche de Java et dans la nouvelle fenêtre qui vient de s'ouvrir, sélectionnez PyDev. Vous constaterez une modification de l'interface : vous voilà prêt à développer en Python !

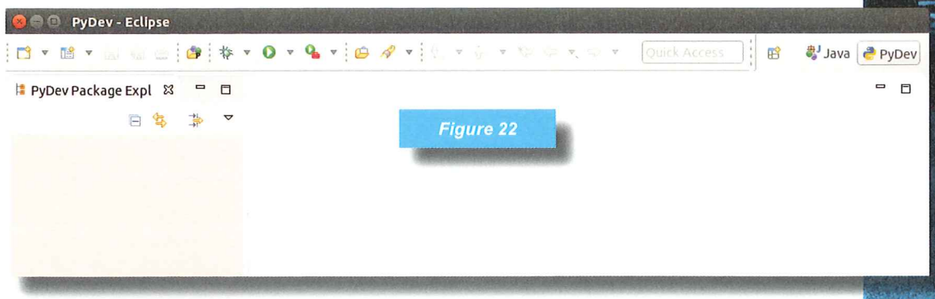


Figure 22

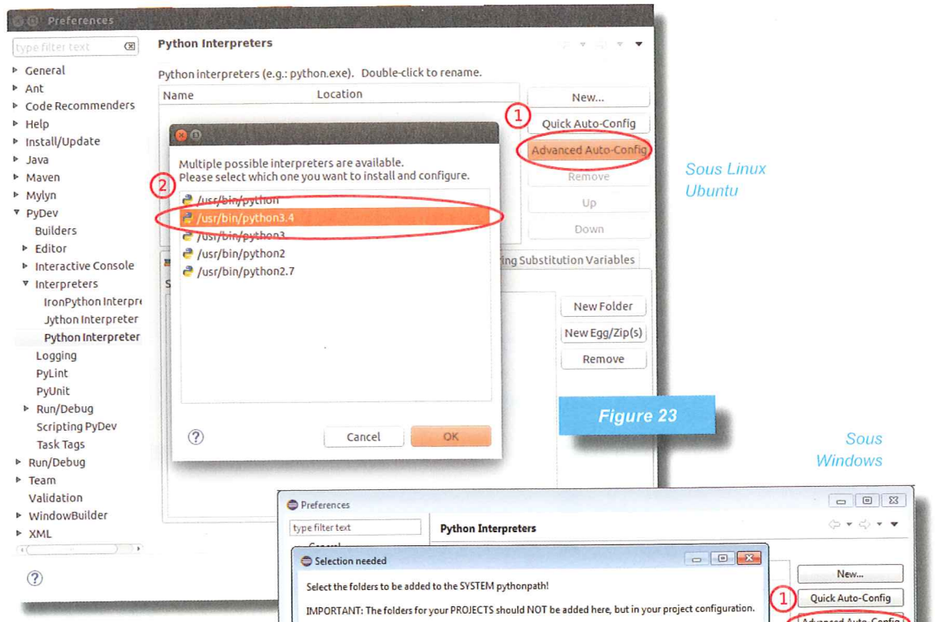
À retenir

Sous Mac OS X, si la configuration automatique de l'interpréteur Python ne fonctionne pas, cliquez sur le bouton *New...* et saisissez :

- Interpreter Name : `Python 3`
- Interpreter Executable : `/Library/Frameworks/Python.framework/Versions/3.4/bin/python3`

4. LES BASES DE PYTHON

Python peut être utilisé de manière purement impérative. Ce sont ces notions de base que nous allons voir ici et qui nous serviront par la suite. Nous irons un peu vite, mais ne vous inquiétez pas, les notions seront réexpliquées lorsque nous en aurons besoin dans notre projet. Si vous souhaitez vraiment les approfondir, vous pouvez consulter avec profit *GNU/Linux Magazine HS n°71* « C'est décidé, aujourd'hui je m'y mets ! Je programme ».



Sous Linux Ubuntu

Figure 23

Sous Windows

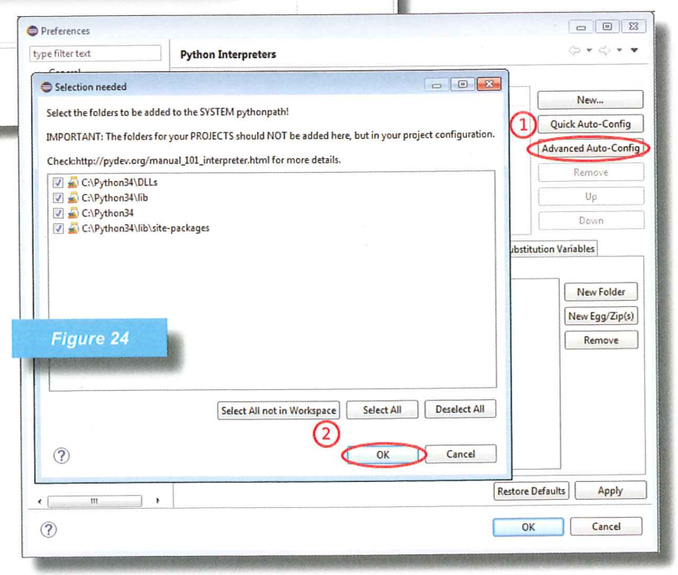
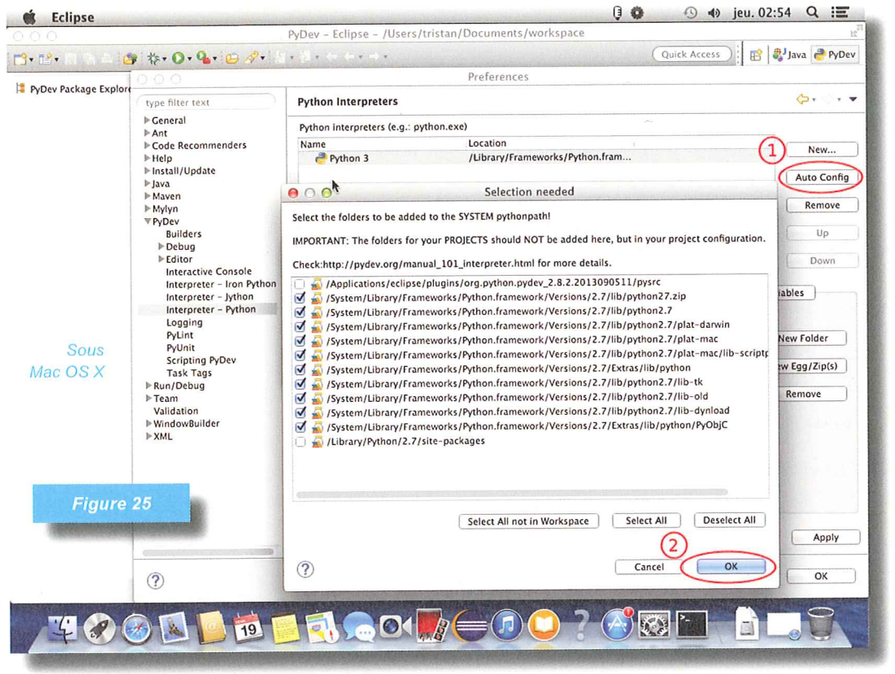


Figure 24

Les exemples seront donnés en utilisant l'interpréteur interactif qui est fourni avec le langage. Ce dernier permet de taper des instructions en Python et d'obtenir immédiatement le résultat. Mais nous n'avons pas encore indiqué à PyDev quel interpréteur Python nous souhaitons utiliser... C'est ce que nous allons faire pour pouvoir utiliser l'interpréteur interactif :

⇨ Sous Windows et Linux, sélectionnez le menu ***Windows > Preferences > PyDev > Interpreters > Python Interpreter***. Sous Mac OS X, le menu se trouve dans ***Eclipse > Préférences... > PyDev > Interpreter - Python*** ;



Sous Mac OS X

Figure 25

⇨ Cliquez sur le bouton ***Advanced Auto-Config*** et sélectionnez ***Python3.4***. Les chemins vers Python seront bien sûr différents que vous utilisiez Linux Ubuntu, Windows ou Mac OS X ;

⇨ Cliquez ensuite sur le menu ***Window > Show view > Console***. Cela fera apparaître un petit onglet en bas de page. Il faut modifier cette console pour utiliser la console PyDev en cliquant sur l'icône en forme de fenêtre (en haut et à droite de l'onglet), puis en sélectionnant l'entrée ***PyDev Console*** et en cliquant dans la fenêtre suivante sur ***Python Console***.

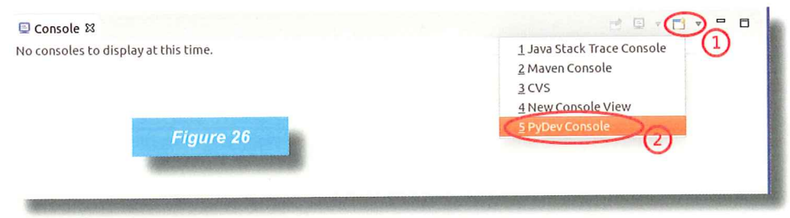


Figure 26

Vous obtiendrez alors la console interactive Python dans laquelle vous pourrez effectuer les tests.

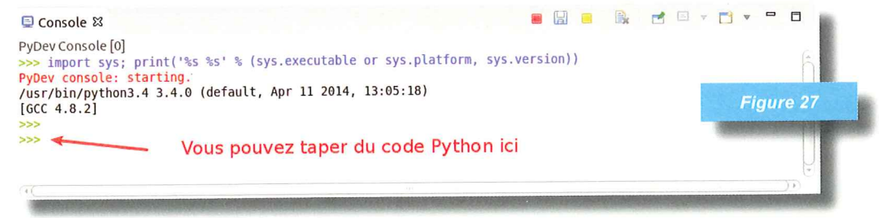


Figure 27

4.1 Les variables

Python est un langage à **typage dynamique fort** : une variable prend le type de la valeur qu'elle contient. Par exemple, si l'on définit une variable `a` en y plaçant la valeur `10`, alors `a` sera un entier :

Fichier
```
>>> a = 10
>>> a
10
>>> type(a)
<class 'int'>
```

Il existe de nombreux types en Python, parmi lesquels :

⇨ les entiers, comme nous l'avons déjà vu ;

⇨ les nombres réels (encore appelés flottants) :

Fichier
```
>>> b = 3.14
>>> type(b)
<class 'float'>
```

⇨ les booléens pouvant valoir vrai (`True`) ou faux (`False`) :

Fichier
```
>>> c = True
>>> type(c)
<class 'bool'>
```

⇨ les chaînes de caractères, encadrées par des guillemets ou des apostrophes :

Fichier
```
>>> d = "c'est une chaîne"
>>> e = 'c\'est une autre chaîne'
>>> type(e)
<class 'str'>
```

⇨ les listes qui permettent de stocker plusieurs valeurs (on peut les voir comme des tableaux de valeurs) :

Fichier
```
>>> f = [1, 2, 3, "abc", 5.65, True]
>>> f[0]
1
>>> f[2]
3
>>> f[2] = False
>>> f[2]
False
>>> type(f)
<class 'list'>
```

⇨ les tuples qui sont des listes non modifiables (on ne peut pas changer la valeur d'un élément après création du tuple) :

Fichier
```
>>> g = (1, 2, 3, "abc", True)
>>> g[0] = 1
>>> g[1] = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> type(g)
<class 'tuple'>
```

4.2 Les structures de test

Les structures de test permettent d'exécuter un bloc de code si une condition est vérifiée (renvoie un booléen). En Python, les blocs sont définis par une instruction se terminant par le caractère deux-points et une indentation sur la ou les lignes suivantes appartenant au bloc :

Fichier
```
>>> a = 1
>>> if a == 1:
...     print("a vaut 1")
...     print("Fin du bloc")
... else:
...     print("a est different de 1")
...
a vaut 1
Fin du bloc
```

Ces deux instructions constituent un bloc, car elles possèdent le même niveau d'indentation.

Ici, on peut voir une première fonction : `print()` permet d'afficher un message à l'écran (le message est passé en paramètre, entre les parenthèses).

4.3 Les structures de boucle

Une structure de boucle va permettre de répéter une opération plusieurs fois. Il existe deux structures de boucles en Python.

La première consiste à parcourir tous les éléments d'une liste et s'arrêter une fois que la liste est vide :

Fichier
```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
```

La fonction range(n) renvoie une liste de n valeurs de 0 à n-1.

Comme on parcourt la liste `[0, 1, 2, 3, 4]`, on affiche un à un chacun de ces éléments.

La deuxième structure effectue une boucle tant qu'une condition est vraie :

Fichier

```
>>> i = 0
```
Définition d'une variable « compteur » ou « variable de boucle ».

```
>>> while i < 5:
```
Condition de fin : dès que i est supérieur ou égal à 5 on arrête.

```
...     print(i)
...     i = i + 1
...
0
1
2
3
4
```
Ne pas oublier de modifier la valeur du compteur sinon la boucle est infinie !

Ce code effectue le même traitement que le précédent est affiche les entiers de 0 à 4.

4.4 Les fonctions

Une fonction permet de définir un comportement qui pourra être paramétré. Par exemple :

Fichier

```
>>> def mois(n):
...     m = ("janvier", "fevrier", "mars", "avril", "mai", "juin",
"juillet", "aout", "septembre", "octobre", "novembre", "decembre")
```
Stockage des mois dans un tuple.

```
...     if n < 1 or n > 12:
```
Test permettant de savoir si le nombre n passé en paramètre peut désigner un élément du tuple m.

```
...         print("Mois inconnu !")
...     else:
...         return m[n-1]
...
>>> mois(1)
'janvier'
```
Valeur de retour de la fonction. Lors de l'appel, on remplacera celui-ci par la valeur retournée.

Lors de l'appel de `mois(1)`, `n` prend la valeur de 1, et la fonction renvoie la valeur `m[1-1]`, soit `m[0]` qui correspond à la chaîne de caractères `janvier`.

4.5 Les modules

Il est possible de charger des informations (fonctions par exemple) depuis un autre fichier Python. On parle de modules. Il y a les modules standards, installés en même temps que le langage :

Fichier

```
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> from math import pi
>>> pi
3.141592653589793
```


Et il y a les modules définis par l'utilisateur et qui seront appelés par leur nom de fichier sans l'extension `.py`. Par exemple, pour un module `monModule.py`, il faudrait importer son contenu par :

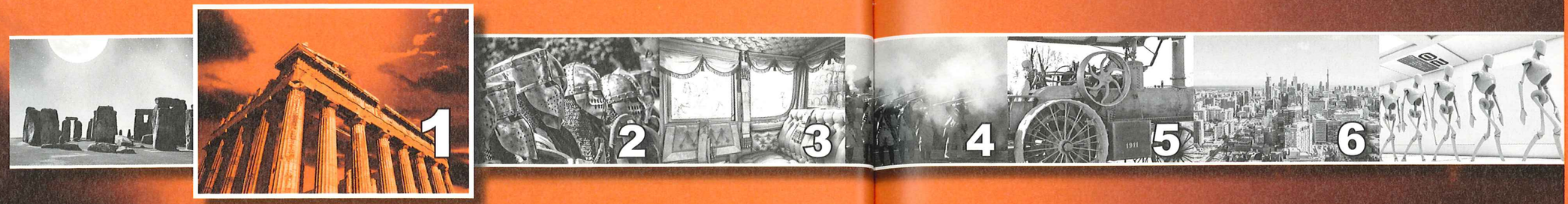
Fichier

```
>>> fom monModule import nom_fonction_1, nom_fonction_2
```

Après cette ligne, nous aurons accès aux fonctions `nom_fonction_1` et `nom_fonction_2`.

Pour récapituler :

- Pour écrire notre code Python, nous utiliserons l'Environnement de Développement Intégré Eclipse avec l'extension PyDev ;
- Il est possible de tester et d'exécuter directement du code Python dans l'interpréteur interactif ;
- On ne déclare pas le type d'une variable, on lui affecte directement une valeur ;
- Le premier élément des listes et des tuples se trouve à l'indice 0 ;
- Les blocs (suites d'instructions à exécuter « ensemble ») sont définis par des indentations (espacement supplémentaire de 4 caractères espaces ou de tabulations).



JOUR 1

LA CARTE, UN OBJET SIMPLE

Pour démarrer notre projet, nous allons commencer par l'élément le plus simple : la carte. Nous allons définir ce qu'est une carte et comment elle doit être utilisée, puis nous écrirons son code de manière à pouvoir la manipuler.

JOUR 1

La Programmation Orientée Objet se base sur une observation de notre environnement : nous sommes entourés d'objets. Le mook que vous tenez dans les mains est un objet, les pages qui le composent sont aussi des objets et sa version PDF également. Chacun de ces objets est défini à l'aide d'un certain nombre d'attributs. Pour le mook, nous pourrons le caractériser par son titre, le nombre de pages, etc.

Voyons ce qu'il en est pour une carte.

1. LE CAS DE LA CARTE

Une carte possède une valeur et une couleur. Par exemple, pour l'as de cœur, la valeur est 14 (pour la bataille) et la couleur est cœur. Pour représenter l'ensemble des 52 cartes, nous aurons donc des combinaisons des valeurs 2 à 14 et des couleurs cœur, carreau, pique et trèfle.

Pour représenter une carte dans un programme, nous aurons besoin de conserver deux valeurs dans des variables comme le montre le schéma suivant :

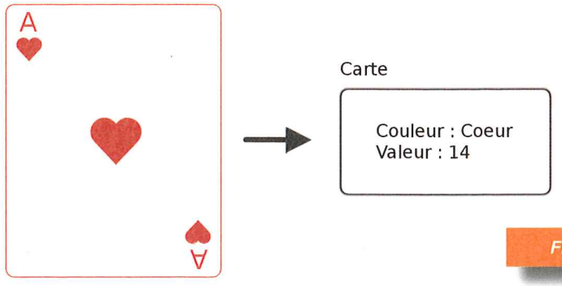


Figure 1

Les variables qui permettent de définir une carte et plus généralement un objet sont appelées **attributs**. En modifiant la valeur de ces attributs, on modifie l'objet (un as de cœur est différent d'un valet de trèfle).

En informatique, pour créer un objet il va d'abord falloir expliquer comment le créer, quel est le schéma à suivre. C'est le rôle de la **classe** qui indique quel est le nom de l'objet (par exemple une carte) et quels sont ces attributs (par exemple, couleur et valeur). Vous ne pourrez rien faire du schéma seul. Prenez un plan de Lego : sans les briques vous ne parviendrez pas à réaliser le modèle. En POO, les briques peuvent être comparées à la mémoire de l'ordinateur dans laquelle seront stockées les valeurs des attributs de l'objet. Cet aspect de « construction » de l'objet est tenu par une fonction particulière appelée justement **constructeur**. L'appel de cette fonction permettra de créer l'objet et de l'utiliser ensuite.

Il peut exister de nombreux objets construits à partir d'un même schéma, d'une même classe. Chacun de ces objets est une **instance** de la classe ayant servi à les créer.

Voyons ce que cela donne en Python. Pour commencer, il faut créer la classe `Carte` qui sera le schéma, la recette permettant de créer autant de cartes que l'on souhaite. Lancez donc l'éditeur Eclipse et sélectionnez dans le menu ***File > New > PyDev Project***. Une nouvelle

fenêtre va apparaître dans laquelle vous pourrez indiquer le nom du projet. Si plusieurs versions de Python sont disponibles sur votre machine, n'oubliez pas de vérifier que le champ ***Grammar Version*** est bien réglé sur 3.0.

Une fois le projet créé, effectuez un clic droit sur ce dernier et sélectionnez ***New > File***.

Créez alors un fichier `Carte.py` qui contiendra le code indiquant comment créer une carte (si des messages apparaissent, n'en tenez pas compte et acceptez tous les choix par défaut).

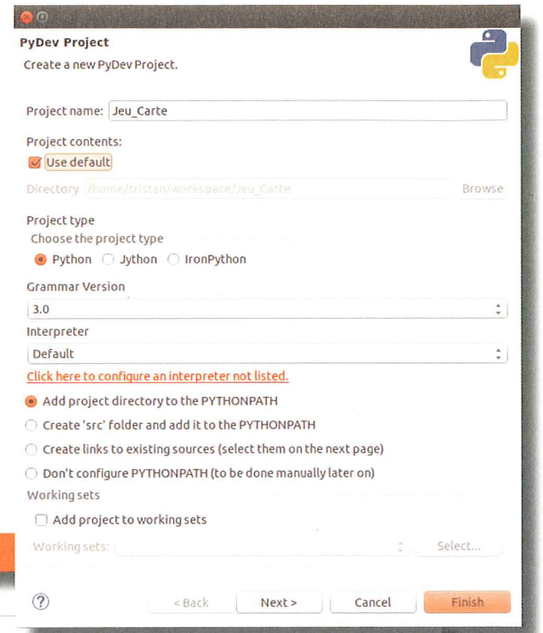


Figure 2

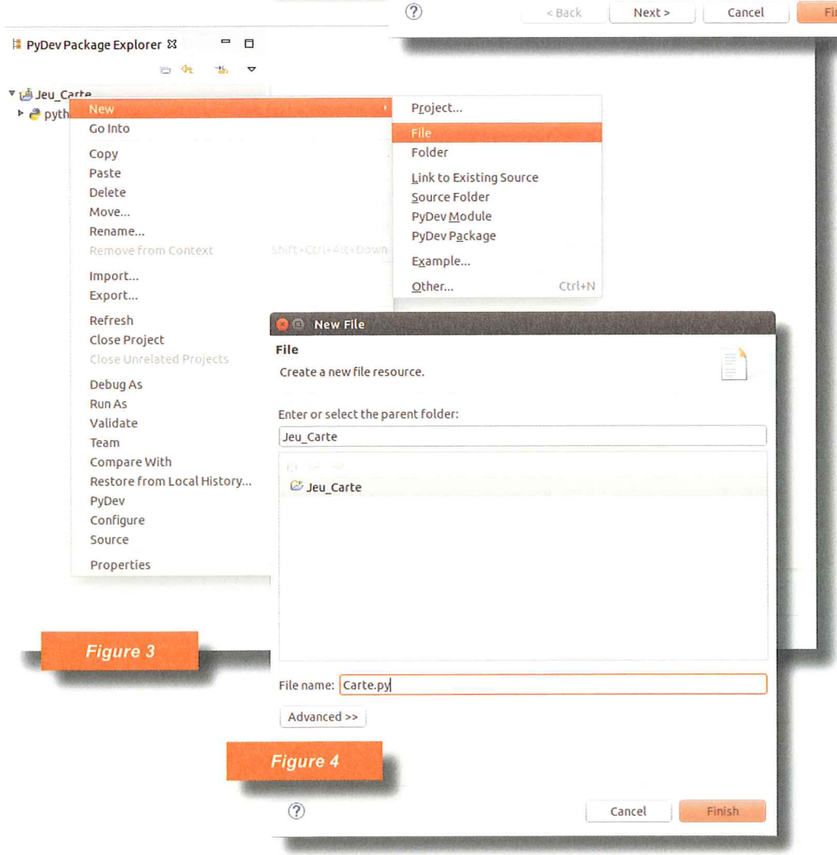


Figure 3

Figure 4

JOUR 1

Vous voilà prêt à saisir le code de votre premier objet dans l'onglet **Carte**. L'icône en forme de feuille comportant un **P** indique qu'il s'agit d'un fichier de code Python (à cause de l'extension **.py** saisie tout à l'heure). Pour l'instant, le code sera très court :

Fichier

```
class Carte:
    def __init__(self, val, coul):
        self.valeur = val
        self.couleur = coul
```

Dans Eclipse, vous remarquerez que la coloration syntaxique vous indique les mots-clés du langage et suivant où se trouve votre curseur, les occurrences des différentes variables.

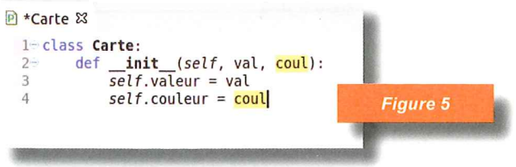


Figure 5

Si vous exécutez ce code en cliquant sur l'icône en forme de flèche blanche dans un disque vert, il ne se passera absolument rien...

Essayons donc de comprendre ces quatre lignes avant de voir ce que l'on peut en faire.

Figure 6

La première ligne, **class Carte:**, indique que toutes les lignes qui suivront avec au moins un niveau d'indentation (espaces ou tabulation) feront partie de la définition de ce qu'est un objet de type **Carte**.

Sur la ligne suivante, **def __init__(self, val, coul):**, on peut voir qu'il s'agit d'une définition de fonction (mot-clé **def**). Cette fonction porte un nom étrange : **__init__** (avec deux caractères underscore au début et à la fin). Ce mot est un nom réservé (comme tous les noms encadrés par des doubles underscores en Python) et il indique que la fonction est le constructeur de la classe. Cette fonction admet trois paramètres : **self**, **val** et **coul**.

Les deux derniers ne posent pas trop de problèmes, ils serviront à indiquer la valeur (**val**) et la couleur (**coul**) de la carte que nous souhaitons créer. Par contre, le premier paramètre, **self**, peut paraître un peu étrange : nous n'avons besoin que de deux valeurs pour définir une carte. En fait, **self** fait référence à l'objet courant, la carte que nous sommes en train de manipuler. Cet élément permet de faire la distinction entre une simple variable et les attributs de la classe que l'on retrouve dans les deux dernières lignes sous la forme **self.valeur** et **self.couleur**. Ces quatre lignes permettent donc d'indiquer que pour créer une carte il faut indiquer deux valeurs qui seront conservées en tant qu'attributs dans **valeur** et **couleur**.

Nous avons écrit un petit code et compris ce qu'il faisait... mais le voir fonctionner pourrait nous aider un peu ! Pour cela, il faut utiliser la classe que nous venons de définir pour créer un objet de type **Carte**, autrement dit une instance de **Carte**. Pour commencer, nous allons utiliser l'interpréteur interactif : dans l'onglet ***Console***, cliquez sur le bouton ***Nouvelle fenêtre*** et sélectionnez ***PyDev Console*** comme nous l'avons vu dans la partie d'introduction. Pour pouvoir utiliser la classe, il faut la charger en mémoire grâce à une instruction **import** :

Fichier

```
>>> from Carte import Carte
```

Cette ligne peut se lire comme : « dans le fichier **Carte.py** récupère la définition de la classe **Carte** ». Une fois cette opération réalisée, nous pouvons créer des cartes :

Fichier

```
>>> c1 = Carte(14, "coeur")
```

JOUR 1

Ici, nous construisons une carte : nous faisons référence au constructeur de la classe `Carte` et donc nous appelons implicitement la fonction `__init__()`. Comme vous pouvez le voir, nous ne passons que deux paramètres à cette fonction alors que dans la définition il y en avait trois avec le paramètre `self`. Ce paramètre qui décrit l'objet courant est en fait géré directement par Python et dans le cas du constructeur il fait référence à l'objet qui va être créé... donc ne pouvons pas le passer en paramètre puisque nous voulons justement le créer !

Une fois l'instance de `Carte` créée, nous pouvons accéder à ses attributs `valeur` et `couleur` :

Fichier

```
>>> c1.couleur
'coeur'
>>> c1.valeur
14
```

Si l'on crée une autre instance de `Carte`, nous aurons forcément d'autres valeurs pour les attributs :

Fichier

```
>>> c2 = Carte(9, "pique")
>>> c2.couleur
'pique'
>>> c2.valeur
9
```

Ces différentes valeurs peuvent être utilisées pour afficher la carte de manière intelligible :

Fichier

```
>>> print(c2.valeur, "de", c2.couleur)
9 de pique
```

Par contre, dans le cas de la carte `c1`, il faudra passer par une étape de traduction pour ne pas afficher « 14 de cœur », mais « as de cœur ». Du coup, il serait intéressant de disposer d'une fonction qui permettrait d'afficher le contenu d'une carte. Nous allons pouvoir apercevoir ici une partie de la puissance de la POO : cette fonction va être rattachée à la définition d'une carte et pourra ainsi être utilisée pour n'importe quelle carte.

2. UNE FONCTION POUR AFFICHER UNE CARTE

Nous avons vu que pour afficher une carte, il fallait utiliser la fonction `print()` et lui passer en paramètre la valeur et la couleur d'une carte. Si nous réalisons cette opération au sein de la classe définissant une carte, ces valeurs sont déjà connues et il est inutile de les passer en paramètre ! C'est l'utilité du `self` désignant l'objet courant : on peut y faire référence n'importe où dans la classe et il permet d'accéder aux valeurs stockées sous forme d'attributs (ceux qui ont été définis en précédant leur nom du mot-clé `self`).

Ajoutons cette fonction à la définition de notre carte :

Fichier

```
class Carte:
    def __init__(self, val, coul):
        self.valeur = val
        self.couleur = coul

    def affiche(self):
        print(self.valeur, "de", self.couleur)
```

Comme vous pouvez le voir, cette fonction prend en paramètre `self` : ceci permet d'indiquer qu'elle est « liée » à un objet de type `Carte`. L'appel à cette fonction se fera en l' « appliquant » à un objet `Carte`. Testons cela dans l'interpréteur interactif. Comme nous avons modifié le code de notre classe, il faut fermer l'interpréteur précédent (cliquez sur le carré rouge ***Terminate current console***) et en ouvrir un nouveau en suivant la même procédure que précédemment. Si vous ne faites pas cela, le nouveau code ne sera pas reconnu et vous obtiendrez un message d'erreur lorsque vous tenterez d'afficher la carte.

Fichier

```
>>> from Carte import Carte
>>> c1 = Carte(7, "trefle")
>>> c1.affiche()
7 de trefle
```

Après avoir créé une carte, nous l'affichons en appliquant la fonction `affiche()` à cette dernière. Ici, le mot « appliquer » prend tout son sens puisque l'on prend `c1`, l'objet créé avec la classe `Carte` (ou l'instance de `Carte` en langage POO), et grâce à l'opérateur « point », on lui applique la fonction `affiche()` (en langage POO, on ne parle plus alors de fonction, mais de `méthode`). Lors de l'appel de la méthode `affiche()`, `self` fera référence à l'objet courant, c'est-à-dire `c1`. Les deux schémas suivants illustrent le fonctionnement des variables dans le code :

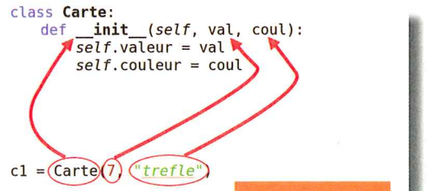


Figure 7

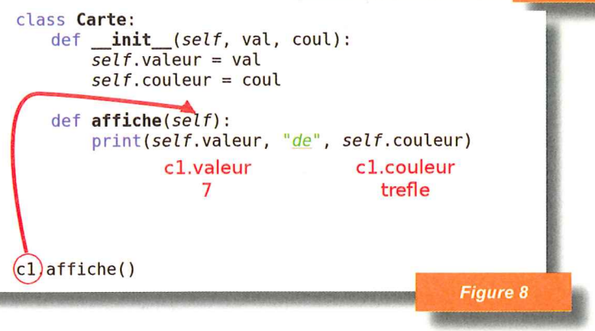


Figure 8

À retenir

Lors de l'utilisation de l'interpréteur interactif (PyDev console), une fois qu'une définition de classe est chargée, vous ne pouvez plus la modifier (dans l'interpréteur). Il vous faut alors refermer cet interpréteur, en démarrer un nouveau et charger la classe modifiée pour pouvoir l'utiliser.

Maintenant que ces lignes sont comprises, nous allons pouvoir améliorer un petit peu notre code. Un ordinateur sait très bien manipuler les nombres par contre, pour lui, « roi », « as », « cœur », etc. ne signifie rien. Nous allons donc introduire un codage pour représenter chacune de nos 52 cartes. Nous avions déjà associé la valeur 14 pour l'as, nous aurons donc : 11 pour le valet, 12 pour la dame et 13 pour le roi. Pour les couleurs, nous procéderons de manière arbitraire avec 1 pour cœur, 2 pour carreau, 3 pour trèfle et 4 pour pique. Tout cela est résumé dans les deux tableaux suivants :

Code	Valeur
0	-
1	-
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	Valet
12	Dame
13	Roi
14	As

Code	Couleur
0	Cœur
1	Carreau
2	Trèfle
3	Pique

Pour créer la carte « 8 de carreau », nous devrons ainsi exécuter :

Fichier

```
>>> c = Carte(8, 1)
```

Pour l'instant, cela rend la tâche plus compliquée, mais vous verrez par la suite qu'au contraire, ce principe nous fera gagner du temps. Par contre, pour l'affichage de la carte, il faudra bien sûr passer par une étape de décryptage. Nous allons donc modifier la méthode `affiche()` pour que celle-ci soit capable de traduire ce code en français. Auparavant, nous allons nous assurer qu'il est impossible de créer une carte avec un code indéchiffrable (par exemple, avec une valeur de 21 et une couleur de 6) :

Fichier

```
class Carte:
    def __init__(self, val, coul):
        if val < 2 or val > 14:
            print("Erreur :
La valeur d'une carte est comprise entre 2 et 14")
            exit(1)
        if coul < 0 or coul > 3:
            print("Erreur : Le code couleur d'une carte est compris entre
0 et 3")
            exit(1)
        self.valeur = val
        self.couleur = coul
```

On teste si val contient une valeur inférieure à 2 ou supérieure à 14.

On teste si coul contient une valeur inférieure à 0 ou supérieure à 3.

Nous testons les deux cas d'erreur possibles : soit le code de la valeur est erroné, soit le code de la couleur est faux. Dans chacun de ces cas, s'il s'avère que l'on cherche à créer une carte qui ne peut pas exister, nous arrêtons l'exécution du programme en faisant appel à la fonction `exit()`. Cette fonction admet pour paramètre un code d'erreur compris entre `1` et `255` (le code `0` sert à indiquer qu'il n'y a pas eu d'erreur).

Modifions maintenant la méthode `affiche()` de manière à déchiffrer le code représentant une carte :

Fichier

```
def affiche(self):
    affiche_valeur = None
    affiche_couleur = None

    if self.valeur <= 10:
        affiche_valeur = self.valeur
    elif self.valeur == 11:
        affiche_valeur = "Valet"
    elif self.valeur == 12:
        affiche_valeur = "Dame"
    elif self.valeur == 13:
        affiche_valeur = "Roi"
    else:
        affiche_valeur = "As"

    if self.couleur == 0:
        affiche_couleur = "Coeur"
    elif self.couleur == 1:
        affiche_couleur = "Carreau"
    elif self.couleur == 2:
        affiche_couleur = "Trefle"
    else:
        affiche_couleur = "Pique"

    print(affiche_valeur, "de", affiche_couleur)
```

variables locales n'existant que dans cette méthode.

Nous utilisons ici deux variables `affiche_couleur` et `affiche_valeur` dites `locales` : elles n'existent que « localement » dans le bloc dans lequel elles ont été définies. Ces variables permettent de stocker les valeurs et couleurs (traductions des codes) qui seront affichées à l'écran. Pour traduire les codes dans les valeurs qui leur sont associées, nous utilisons des tests imbriqués (suites de `if` et de `elif`). L'instruction `elif` est une contraction de `else if` et se lit donc « sinon, si ... ».

L'utilisation de tous ces tests donne un code simple à écrire, mais un peu long. Nous pouvons utiliser une astuce en créant des tableaux de conversion. En Python, les tableaux sont appelés **liste** et pour accéder à un élément on doit spécifier sa position (en partant de 0). On peut donc utiliser ce nombre, appelé **index**, pour décoder des valeurs. Voici un exemple d'utilisation de ce principe dans l'interpréteur interactif. Nous n'utiliserons pas une liste, mais un tuple (liste non modifiable), car les éléments ne devront jamais changer :

À retenir

La valeur **None** est une valeur spéciale qui indique un élément indéfini. Une variable qui vaut **None** existe, mais n'a pas de valeur.

Fichier

```
>>> couleurs = ("Coeur", "Carreau", "Trefle", "Pique")
>>> couleurs[0]
'Coeur'
>>> couleurs[2]
'Trefle'
```

Pour savoir à quelle couleur correspond le code **0**, il suffit donc de regarder le contenu de la variable **couleurs[0]** et de la même manière le code **n** correspondra à la couleur **couleurs[n]**. En utilisant cette technique, nous pouvons considérablement réduire la taille du code de la méthode **affiche()** :

Fichier

```
    def affiche(self):
        valeurs = (None, None,  <- aucune carte ne peut avoir la valeur 0 ou 1.
2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet", "Dame", "Roi", "As")
        couleurs = ("Coeur", "Carreau", "Trefle", "Pique")

        print(valeurs[self.valeur], "de", couleurs[self.couleur])
```

Nous avons grandement simplifié le code (même s'il faut un peu plus de réflexion et de connaissances en Python pour le comprendre). Nous pouvons aller encore plus loin dans l'optimisation : à chaque affichage d'une carte, nous allons créer les deux variables **valeurs** et **couleurs** et occuper de l'espace en mémoire. Pourtant ces deux variables sont les mêmes pour toutes les cartes et il suffit donc de les créer une seule fois ! Ces variables devront donc devenir des attributs spéciaux de la classe **Carte** : des attributs identiques et partagés par l'ensemble des objets de type **Carte**. Ces attributs sont des **attributs de classe** ou **attributs statiques**. À la différence des attributs « classiques », ces derniers devront être déclarés à l'extérieur du constructeur. De plus, comme ils sont liés à la classe et non plus à l'objet courant, pour les utiliser il faudra faire précéder leur nom du nom de la classe (et non plus de **self**). Voici ce que cela donne pour la classe **Carte** :

Fichier

```
class Carte:
    valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet", "Dame",
"Roi", "As")
    couleurs = ("Coeur", "Carreau", "Trefle", "Pique")

    def __init__(self, val, coul):
        if val < 2 or val > 14:
            print("Erreur : La valeur d'une carte est comprise entre
2 et 14")
            exit(1)
        if coul < 0 or coul > 3:
            print("Erreur : Le code couleur d'une carte est compris entre
0 et 3")
            exit(1)
        self.valeur = val
        self.couleur = coul

    def affiche(self):
        print(Carte.valeurs[self.valeur], "de", Carte.couleurs[self.
couleur])
```


Ce code fonctionne de la même manière que le code précédent... il est simplement plus court et plus efficace. Prenez le temps de comprendre ce qui se passe réellement en machine en modifiant des valeurs et en voyant la répercussion de ces modifications sur l'exécution du code.

3. PLUSIEURS MÉTHODES D'AFFICHAGE

Comme nous avons écrit une méthode **affiche()** permettant d'afficher une carte, nous pouvons écrire une multitude de méthodes différentes ayant chacune un objectif précis. Par exemple, nous pouvons écrire une méthode qui affiche la carte en mode pseudo-graphique :

Fichier

```
    def affiche_ascii(self):
        nom = str(Carte.valeurs[self.valeur]) +
            " de " + Carte.couleurs[self.couleur]
        Concaténation de chaînes de caractères (on « colle » les
        chaînes entre elles à l'aide de l'opérateur +).
        taille = len(nom) + 2
            On récupère la taille de la chaîne de caractères
            nom et on y ajoute 2.
        print("/", "-" * taille, "\\", sep="")
    Affichage de caractères : l'opérateur * affiche n fois la chaîne et
    le paramètre sep permet d'indiquer que les chaînes doivent être
    collées sans ajout de caractère (normalement il y a un espace).
        print("|", "-" * taille, "|", sep="")
        print("|", nom, "|")
        print("|", "-" * taille, "|", sep="")
        print("\\", "-" * taille, "/", sep="")
```

Nous avons maintenant la possibilité d'afficher une carte de deux façons :

Fichier

```
>>> c = Carte(11, 1)
>>> c1.affiche()
Valet de Carreau
>>> c1.affiche_ascii()
/------------------\
|                  |
| Valet de Carreau |
|                  |
\------------------/
```

Pour afficher une carte de différentes façons, nous pourrions donc créer de nombreuses méthodes. Mais il y a une méthode particulière qui est très intéressante : la méthode **__str__()**. C'est à vous de définir cette méthode et comme vous pouvez le voir, celle-ci doit avoir un nom particulier, un nom réservé, puisqu'encadré par des doubles underscores. Cette méthode sera automatiquement appelée lorsque

À retenir

L'opérateur + appliqué à des chaînes de caractères permet de réaliser une opération de concaténation, c'est-à-dire que l'on prend deux chaînes et qu'on les « colle » ensemble de manière à n'en avoir plus qu'une seule.

À retenir

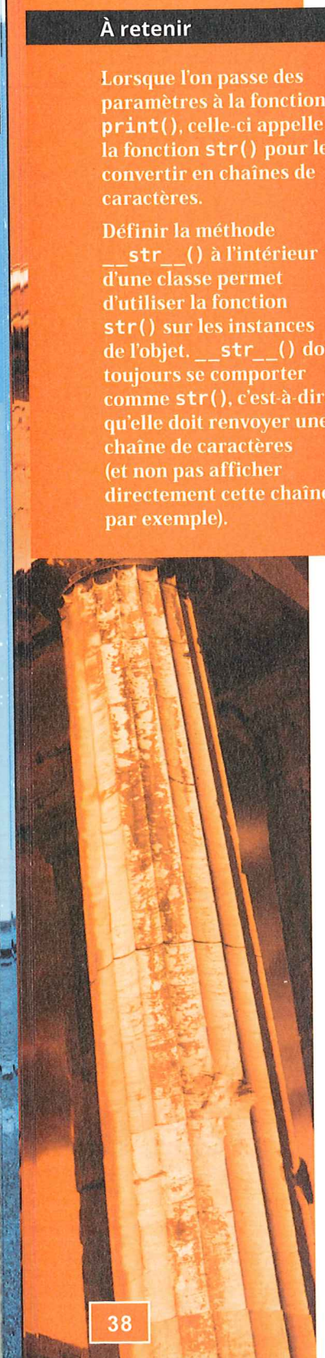
L'opérateur + appliqué à des chaînes de caractères permet de réaliser une opération de concaténation, c'est-à-dire que l'on prend deux chaînes et qu'on les « colle » ensemble de manière à n'en avoir plus qu'une seule.

JOUR 1

À retenir

Lorsque l'on passe des paramètres à la fonction `print()`, celle-ci appelle la fonction `str()` pour les convertir en chaînes de caractères.

Définir la méthode `__str__()` à l'intérieur d'une classe permet d'utiliser la fonction `str()` sur les instances de l'objet. `__str__()` doit toujours se comporter comme `str()`, c'est-à-dire qu'elle doit renvoyer une chaîne de caractères (et non pas afficher directement cette chaîne par exemple).



l'on essaiera de convertir notre objet en chaîne de caractères... et c'est justement ce que fait la fonction `print()` ! Donc avec une telle méthode, nous pourrons afficher une carte beaucoup plus simplement :

Fichier

```
>>> c = Carte(11, 1)
>>> c1.affiche()
>>> print(c1)
Valet de Carreau
```

Nous allons modifier le code de la méthode `affiche()` de manière à obtenir la méthode `__str__()`. La particularité de cette méthode est qu'elle doit toujours renvoyer une chaîne de caractères :

Fichier

```
    def __str__(self):
        return str(Carte.valeurs[self.valeur]) + " de " +
Carte.couleurs[self.couleur]
```

4. UN PROGRAMME QUI AFFICHE DES CARTES

Nous avons créé une classe qui définit le comportement d'une carte mais, jusqu'à présent, pour l'utiliser nous étions obligés de passer par l'interpréteur interactif. Nous allons maintenant créer un programme qui utilisera cette classe pour créer quelques cartes et les afficher.

Comme nous avons fait beaucoup de tests sur la classe `Carte`, reprenons le fichier `Carte.py` pour mettre notre code au propre :

Fichier

```
class Carte:
    valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10,
"Valet", "Dame", "Roi", "As")
    couleurs = ("Coeur", "Carreau", "Trefle", "Pique")

    def __init__(self, val, coul):
        if val < 2 or val > 14:
            print("Erreur : La valeur d'une carte est
comprise entre 2 et 14")
            exit(1)
        if coul < 0 or coul > 3:
            print("Erreur : Le code couleur d'une carte est
compris entre 0 et 3")
            exit(1)
        self.valeur = val
        self.couleur = coul
```


Fichier

```
    def __str__(self):
        return str(Carte.valeurs[self.valeur]) + " de " +
            Carte.couleurs[self.couleur]

    def affiche_ascii(self):
        nom = str(Carte.valeurs[self.valeur]) + " de " +
            Carte.couleurs[self.couleur]
        taille = len(nom) + 2
        print("/", "-" * taille, "\\", sep="")
        print("|", "-" * taille, "|", sep="")
        print("|", nom, "|")
        print("|", "-" * taille, "|", sep="")
        print("\\", "-" * taille, "/", sep="")
```

Il n'y a là rien de nouveau. Nous utiliserons ensuite cette classe depuis un autre fichier que nous appellerons `start.py`. Créez donc un nouveau fichier de la même manière que nous avions créé le fichier `Carte.py`. Dans ce programme nous allons indiquer quelles sont les actions à effectuer pour utiliser notre classe :

Fichier

```
from Carte import Carte                    <- On charge la définition de la classe Carte.

if __name__ == "__main__":                 <- Ligne permettant de savoir si le programme a été exécuté ou chargé depuis un autre programme.

    c = Carte(12, 2)
    c2 = Carte(14, 3)                      <- Création de deux instances de Carte.

    print("Premiere carte :")
    print(c)                               <- Affichage de la première carte.

    print("\nDeuxieme carte :")
    c2.affiche_ascii()                     <- Affichage de la deuxième carte.
```

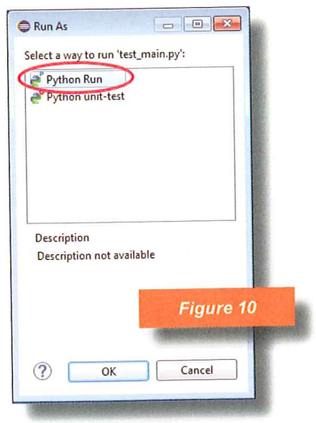


Figure 10

Dans ce code, nous définissons le programme à exécuter. Pour voir le résultat qu'il produit, cliquez sur l'icône ***Run As...*** en forme de disque vert sur lequel se trouve une flèche de lecture blanche (vu précédemment). Lors de la première exécution, vous devrez indiquer que vous souhaitez exécuter le code Python : ***Python Run***.

Le résultat de l'exécution du programme s'affichera dans l'onglet Console en bas de la fenêtre d'Eclipse (voir figure 11, page suivante).

JOUR 1

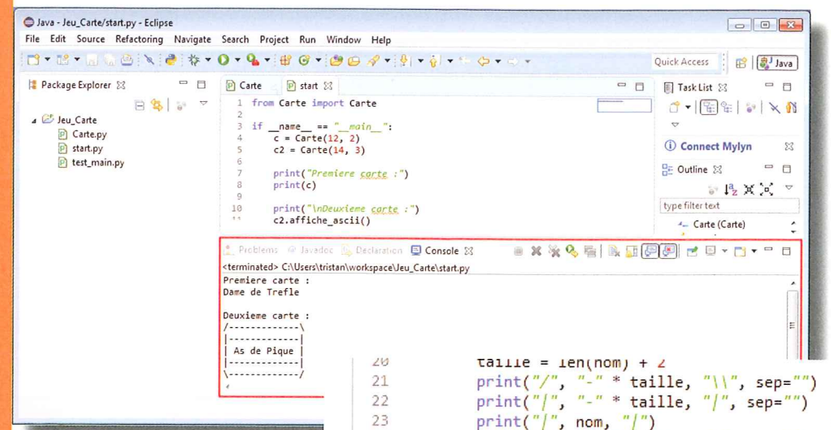


Figure 11

```
20          taille = len(nom) + 2
21          print("/", "-" * taille, "\\", sep="")
22          print("|", "-" * taille, "|", sep="")
23          print("|", nom, "|")
24          print("|", "-" * taille, "|", sep="")
25          print("\\", "-" * taille, "/", sep="")
```

Console

```
<terminated> C:\Users\tristan\workspace\Jeu_Carte\start.py
Premiere carte :
Dame de Trefle

Deuxieme carte :
/-------------\
|-------------|
| As de Pique |
|-------------|
\-------------/
```

Writable | Insert | 4

À retenir

En POO, une bonne pratique veut que les fichiers portent le même nom que la classe qu'ils contiennent et que les noms de classes commencent toujours par une majuscule. En Python ce n'est pas une obligation, mais en appliquant cette nomenclature vous pourrez naviguer beaucoup plus facilement dans votre code.

Ce programme fonctionnerait exactement de la même manière si nous n'avions pas ajouté le test conditionnel `if __name__ == "__main__"`. Ce test permet simplement de distinguer proprement les parties de code qui pourront être exécutées lorsque le programme sera chargé en tant que module (tout fichier de code Python est un module) ou exécuté directement. Pour vous en convaincre, vous pouvez réaliser l'expérience décrite dans la suite. Créez un nouveau fichier `test_main.py` dans lequel vous placez la ligne :

Fichier

```
print(__name__)
```

Si vous exécutez ce code en cliquant sur l'icône ***Run As...***, vous obtiendrez :

Fichier

```
__main__
```

Par contre, si vous allez dans l'interpréteur interactif et que vous chargez (importez) ce module, vous obtiendrez :

Fichier

```
>>> from test_main import *
test_main
```

Cette fois-ci, le nom du module est affiché. Donc la variable réservée `__name__` a pour valeur `__main__` lorsque le code du fichier est exécuté directement et contient le nom du fichier lorsque celui-ci est importé. C'est ce qui permet à notre test de fonctionner... ▮

Pour récapituler :

Nous avons abordé dès ce premier jour beaucoup de notions qui, sans être complexes prises individuellement, peuvent paraître obscures si elles ne sont pas étudiées avec méthode.

- Une **classe** permet de définir un objet (sa recette ou encore son manuel de montage), mais il faut créer une **instance** de la classe pour pouvoir interagir avec ledit objet.
- Les **attributs** sont des variables associées à une classe et accessibles depuis les **méthodes** de la classe.
- Les **méthodes** sont des fonctions associées à une classe et qui ont accès aux **attributs**.
- Pour pouvoir créer une instance d'une classe, il faut qu'une méthode spéciale ait été définie : le constructeur. En Python, le constructeur se note `__init__()`.
- La méthode spéciale `__str__()` permet de définir la chaîne de caractères qui sera renvoyée lorsque la fonction `str()` sera appliquée à une instance de l'objet.
- Un **attribut de classe** ou **attribut statique** est un attribut qui est partagé par l'ensemble des instances d'une classe : tous les objets créés à partir d'une même classe peuvent lire et écrire dans une même variable.
- La variable spéciale `__name__` permet de savoir si l'on a chargé le fichier Python en tant que module, ou s'il a été exécuté directement.

JOUR 2

AVEC PLUSIEURS CARTES, ON CRÉE UN JEU DE CARTES

Nous avons défini ce qu'était une carte. Pour pouvoir jouer à bataille, nous aurons besoin d'un jeu de 52 cartes et pour créer ce jeu, nous avons deux possibilités : soit une création manuelle de chaque carte, soit la définition d'un nouvel objet qui contiendra l'ensemble des cartes. C'est bien sûr la deuxième solution qui est la plus intéressante...

JOUR 2

Lors du premier jour, nous avons pu créer des cartes et les afficher. Mais il est bien évident que la création d'un jeu de cartes en utilisant uniquement cet objet n'est pas pensable : les risques d'erreurs de manipulation seraient beaucoup trop importants ! Il nous faut réfléchir à une structure qui engloberait l'ensemble de ces cartes.

1. UTILISER UN OBJET POUR CRÉER UN NOUVEL OBJET

Si nous essayons de définir ce qu'est un jeu de cartes, nous pourrions dire que c'est un ensemble de cartes toutes différentes sur lesquelles certaines actions sont possibles : mélanger le jeu et tirer une carte sur le sommet du jeu par exemple. Les dessins aident souvent à réfléchir à des idées, à des concepts que l'on souhaite mettre en œuvre. En POO, on utilise une notation particulière avec la **modélisation UML** (*Unified Modeling Language*). Dans cette modélisation, et plus précisément les **diagrammes de classes**, les classes sont représentées par des rectangles découpés en trois zones : le nom de la classe, les attributs de la classe et les méthodes de la classe.

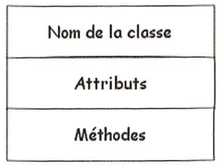


Figure 1

Avec cette notation (qui n'est pas encore véritablement de l'UML), la classe `Carte` que nous avons définie hier est représentée de la manière suivante : voir la figure ci-contre.

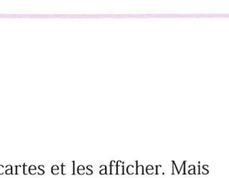


Figure 2

Si nous formalisons de la même manière un jeu de cartes, d'après la définition que nous en avons donnée précédemment, nous obtenons :

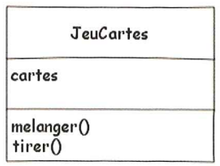


Figure 3

Nous voyons bien qu'il s'agit d'un nouvel objet, d'une nouvelle classe à écrire. Par contre, on peut encore se poser la question du contenu de l'attribut `cartes`. Comme nous allons manipuler un ensemble de 52 cartes, le plus simple est d'utiliser une liste de 52 instances de `Carte`. Pour l'instant, nous pouvons représenter cela de la manière suivante : voir Figure 4.

Ce que nous venons de faire s'appelle la `composition` : nous avons utilisé un objet en tant qu'attribut d'un autre objet. D'un point de vue technique, il n'y a rien à faire si ce n'est créer les instances de `Carte` et les stocker dans l'attribut `cartes` de la classe `JeuCartes`.

À retenir

La `composition` permet d'utiliser des objets en tant qu'attributs pour créer de nouveaux objets.

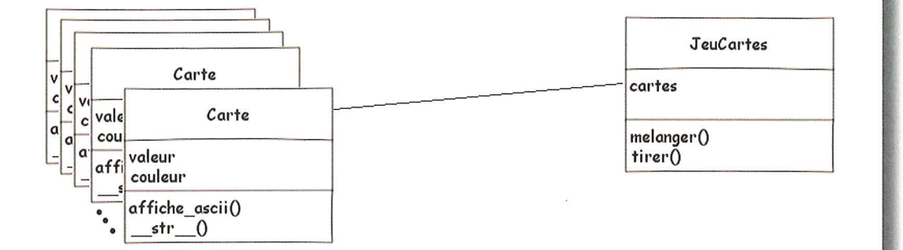


Figure 4

2. LE JEU DE CARTES

Commençons la définition de notre jeu. Pour cela, comme il s'agit d'une nouvelle classe, il faut créer un nouveau fichier `JeuCartes.py` :

Fichier

```
from Carte import Carte          <- Chargement de la définition de la classe Carte.

class JeuCartes:
    def __init__ (self):         <- Attribut cartes initialisé avec
        self.cartes = []             une liste vide.

        for val in range(2, 15):
            for coul in range(4):
                self.cartes.append(Carte(val, coul))
```

Boucles permettant de générer toutes les combinaisons correspondant aux 52 cartes d'un jeu.

Pour tester cette classe, comme nous n'avons pas encore écrit de programme, il faudra passer par l'interpréteur interactif :

Fichier

```
>>> from JeuCartes import JeuCartes
>>> j = JeuCartes()
```

Il n'y a pas d'erreur donc apparemment `j` est une instance de `JeuCartes` et doit contenir les 52 cartes... mais comment le vérifier ? Affichons le contenu de l'attribut `cartes` :

Fichier

```
>>> print(j.cartes)
[<Carte.Carte object at 0x7f26a968f7f0>, <Carte.Carte object at
0x7f26a96b28d0>, ..., <Carte.Carte object at 0x7f26a58422b0>, <Carte.Carte
object at 0x7f26a58422e8>]
```

Aïe ! Mais qu'est-ce que c'est que ce charabia ? On peut voir que l'attribut `cartes` de l'instance `j` contient bien une liste d'éléments puisque l'affichage de son contenu contient un crochet ouvrant et un crochet fermant. Par contre, les éléments de la liste sont de la forme `<Carte.Carte object at 0x...>`. En fait, cela nous indique qu'il s'agit d'un objet de type `Carte` (d'une instance

de **Carte**) et que nous pourrons l'utiliser en tant que tel. **j.cartes[0]** est une carte, **j.cartes[1]** est une carte, etc. Pour afficher le contenu d'une carte, il suffit de faire appel à la fonction **print()** puisque nous avons défini la méthode **__str__()** dans la classe **Carte** :

Fichier

```
>>> print(j.cartes[0])
2 de Coeur
```

En effet, ça fonctionne ! Vérifions que nous avons bien créé un jeu de 52 cartes en affichant le nombre d'éléments de la liste contenue dans l'attribut **cartes** :

Fichier

```
>>> len(j.cartes)
52
```

Visiblement, nous avons réussi à faire ce que nous souhaitions. Pour en être totalement certain, il faudrait afficher les 52 cartes du jeu. Ce n'est guère compliqué puisqu'il suffit d'appeler la fonction **print()** sur l'ensemble des éléments de l'attribut **cartes**. Comme pour l'objet **Carte**, nous placerons cet affichage dans la méthode spéciale **__str__()** de la classe **JeuCartes** :

Fichier

```
    def __str__(self):
        cartes_du_jeu = ""          # Variable locale dans laquelle sera stockée la chaîne de caractères renvoyée.
        for carte in self.cartes:   # Parcours de l'ensemble des cartes de la liste self.cartes.
            if cartes_du_jeu == "":
                cartes_du_jeu = str(carte)
            else:
                cartes_du_jeu += ", " + str(carte)   # Construction de la chaîne de retour.
        return cartes_du_jeu        # Retour de la chaîne contenant tous les noms de cartes.
```

Pour tester cette méthode, nous passerons par le traditionnel interpréteur interactif :

Fichier

```
>>> from JeuCartes import JeuCartes
>>> j = JeuCartes()
>>> print(j)
2 de Coeur, 2 de Carreau, 2 de Trefle, 2 de Pique, 3 de Coeur, ..., As de
Trefle, As de Pique
```

La première étape fonctionne parfaitement et nous disposons d'un jeu de 52 cartes complet. Il faut maintenant pouvoir mélanger le jeu et tirer une carte.

2.1 Mélanger le jeu

En informatique, il n'y a jamais une seule solution valable. Dans le cas du mélange, on peut soit créer de A à Z une méthode qui réalisera ce mélange, soit rechercher dans la documentation de Python si une fonction ne peut pas faire le travail à notre place. Ce que nous cherchons à faire ici est de prendre une liste d'éléments et de les mélanger complètement au hasard. La façon la plus simple de réaliser cette opération est de rechercher la fonction capable d'effectuer cette tâche... mais pour cela, il faut savoir où trouver l'information et comment y accéder.

La documentation de Python est disponible en ligne à l'adresse **https://docs.python.org/3**. Une autre façon d'accéder à la documentation est de passer par le système de documentation intégré au langage et qui se nomme **pydoc**. Nous allons utiliser ce système en passant par l'interpréteur interactif. Nous voulons mélanger les éléments d'une liste, donc nous allons commencer par créer une petite liste de test :

Fichier

```
>>> ma_liste = list(range(10))
>>> ma_liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nous aurions pu écrire directement **ma_liste = [0, 1, ..., 9]**, mais il est plus simple d'utiliser les fonctions qui permettent d'obtenir ce résultat : **range(10)** est un intervalle de **0** à **9** et **list()** convertit cet intervalle sous forme de liste.

Nous pouvons débuter notre recherche : nous voulons mélanger aléatoirement les éléments de la liste. En anglais, « aléatoire " se dit *random*, et il existe un **module** standard qui porte ce nom-là ! Pour rappel, tous les fichiers Python sont des modules, mais leur fonction est, normalement, de fournir des outils aux développeurs (fonctions, objets, etc.). Un module standard est un module installé par défaut avec Python.

À retenir

La fonction `help()` permet d'afficher les pages de documentation associées à l'objet qui lui est passé en paramètre (cet objet peut être un module, une fonction, etc.). Dans le cas d'un module, la documentation risque d'être longue à lire aussi, pour restreindre le champ de recherche, on peut utiliser la fonction `dir()` permettant d'afficher la liste des méthodes et attributs associés à l'objet passé en paramètre. Il faut alors sélectionner un ou plusieurs noms qui semblent pouvoir désigner l'action attendue et lire la documentation à l'aide de `help()`.

JOUR 2

Chargeons le module **random** :

Fichier
```
>>> import random
```

Avec cette écriture, chaque fois que nous voudrons utiliser un élément de **random**, nous devrons le préfixer par le nom du module (donc **random**). Pour voir rapidement la liste des fonctions proposées par le module, nous pouvons utiliser la fonction **dir()** :

Fichier
```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', ..., 'shuffle', 'triangular',
'uniform', 'vonmisesvariate', 'weibullvariate']
```

Perdu au milieu des différents noms, un terme devrait vous interpeller : *shuffle*. En effet, il s'agit du mot anglais pour mélanger/brouiller. Nous allons maintenant utiliser la fonction **help()** qui prend en paramètre une fonction ou un objet et nous affiche la documentation qui y est associée :

Fichier
```
>>> help(random.shuffle)
Help on method shuffle in module random:

shuffle(x, random=None) method of random.Random instance
    Shuffle list x in place, and return None.

    Optional argument random is a 0-argument function returning a
    random float in [0.0, 1.0); if it is the default None, the
    standard random.random will be used.
```

On voit aussitôt quels sont les paramètres acceptés par la fonction et ce qu'elle fait : la liste sera remplacée par la liste mélangée. Nous pouvons donc conclure notre test :

Fichier
```
>>> random.shuffle(ma_liste)
>>> ma_liste
[1, 7, 8, 2, 3, 9, 5, 0, 6, 4]
```

Il ne nous reste plus qu'à appliquer ce mécanisme au sein de la classe **JeuCartes** en ajoutant la méthode **melanger()**. Je ne redonnerai pas ici le code du constructeur et de la méthode **__str__()** qui restent inchangés :

Fichier
```
from Carte import Carte
import random                    <-- Chargement du module random.

class JeuCartes:

    ...

    def melanger(self):          <-- Définition de la méthode
        random.shuffle(self.cartes)   melanger() qui mélange les
                                      éléments de self.cartes.
```


Le test dans l'interpréteur interactif nous montrera que le mélange est effectué correctement :

Fichier
```
>>> from JeuCartes import JeuCartes
>>> j = JeuCartes()
>>> j.melanger()
>>> print(j)
2 de Trefle, 9 de Trefle, 6 de Pique, 10 de Coeur, ..., Valet
de Trefle, As de Carreau, Roi de Pique, 7 de Pique, Dame de
Coeur, 4 de Carreau, Dame de Pique, 7 de Trefle
```

2.2 Tirer une carte

Pour tirer une carte (prendre la carte se trouvant à l'index 0 dans la liste **cartes** et la supprimer de la liste), nous allons adopter la même stratégie que précédemment. En Python tout ce que l'on manipule est un objet : une liste est donc également un objet. Pour afficher la liste des méthodes qui peuvent être appliquées à cet objet, on fait appel à la fonction **dir()** (toujours dans l'interpréteur interactif) :

Fichier
```
>>> ma_liste = [0, 1, 2, 3]
>>>  dir(ma_liste)
['__add__', '__class__', ..., 'pop', 'remove', 'reverse', 'sort']
```

Ici, on peut hésiter entre *pop* (jeter) et *remove* (enlever). Au pire, nous lirons deux pages de documentation à l'aide de la fonction **help()** :

Fichier
```
>>> help(ma_liste.pop)
Help on built-in function pop: pop(...) method of builtins.list
instance
    L.pop([index]) -> item -- remove and return item at index
(default last).
    Raises IndexError if list is empty or index is out of range.
```

Finalement, nous n'aurons pas besoin de chercher plus longtemps : c'est bien la méthode **pop()** qu'il faut appliquer en indiquant en paramètre que nous souhaitons dépiler le premier élément (index **0**). La dernière ligne de la documentation nous indique que si la liste est vide, une erreur de type **IndexError** sera générée (nous reviendrons plus tard sur le traitement des erreurs). Pour l'instant, ajoutons la méthode **tirer()** dans la classe **JeuCartes** :

Fichier
```
    ...

    def tirer(self):
        return self.cartes.pop(0)   <-- Utilisation de pop(0) pour
                                        récupérer et supprimer
                                        de la liste self.cartes le
                                        premier élément.
```

À retenir

À chaque fois qu'une erreur interrompt le déroulement d'un programme, une exception est générée. Une exception décrit l'erreur et par défaut elle provoque l'affichage d'un message et l'arrêt du programme. Il est possible de personnaliser le comportement du programme en fonction des exceptions en les interceptant : un bloc `try` évalue le code et, en cas d'erreur, provoque un branchement dans un bloc `except` portant le nom de l'exception.

L'aspect objet va vraiment apparaître ici : lorsque l'on tire une carte du jeu, on obtient... une carte.

Fichier

```
>>> from JeuCartes import JeuCartes
>>> j = JeuCartes()
>>> j.melanger()
>>> carte = j.tirer()
>>> print(carte)
Valet de Pique
```

Mais que se passe-t-il lorsqu'il n'y a plus de carte dans le jeu ?

Fichier

```
>>> carte = j.tirer()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "JeuCartes.py", line 25, in tirer
    return self.cartes.pop(0)
IndexError: pop from empty list
```

Python nous indique qu'une erreur est apparue en ligne 25 du fichier **JeuCartes.py** et que celle-ci a provoqué une erreur du type **IndexError**. C'est le système de gestion des erreurs qui envoie ce message. Il s'agit d'une **exception** : une erreur générique qui peut être interceptée et traitée par le développeur. **IndexError** est en fait le nom d'une classe d'erreur et l'interception de cette erreur permet de récupérer l'objet, instance de **IndexError**, qui contient les informations sur l'erreur. La récupération d'une erreur se fait à l'aide d'une structure **try/except** : dans le bloc **try** on scrute les erreurs et le bloc **except** est exécuté si une erreur survient dans le bloc précédent. Effectuons un test dans l'interpréteur interactif :

Fichier

```
>>> ma_liste = []
>>> try:
...     elt = ma_liste.pop()
... except IndexError as e:
...     print("Erreur")
...     dir(e)
...
Erreur
['__cause__', '__class__', '__context__', '__delattr__', '__dict__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__',
'__sizeof__', '__str__', '__subclasshook__', '__suppress_context__', '__
traceback__', 'args', 'with_traceback']
```

L'instance e n'existe que dans ce bloc.

Tous les noms encadrés par des doubles underscores sont des noms réservés, mais nous pouvons par exemple afficher le contenu de la liste **args** pour obtenir le message d'erreur associé à l'exception.

Fichier

```
>>> try:
...     elt = ma_liste.pop()
... except IndexError as e:
...     print(e.args[0])
...
pop from empty list
```


Il ne nous reste plus qu'à appliquer ce mécanisme dans la méthode **tirer()** :

Fichier

```
    ...

    def tirer(self):
        try:
            return self.cartes.pop(0)
        except IndexError as erreur:
            print("Il n'y a plus de carte dans le jeu !")
            return None
```

Désormais, s'il n'y a plus de carte dans le jeu, au lieu d'obtenir un message d'erreur, nous verrons apparaître une phrase nous en informant. De plus, la variable censée contenir la carte contiendra la valeur **None** indiquant qu'elle ne contient rien :

Fichier

```
>>> from JeuCartes import JeuCartes
>>> j = JeuCartes()
...
>>> carte = j.tirer()
Il n'y a plus de carte dans le paquet !
```

Notre jeu de cartes est maintenant fonctionnel.

3. ET LA TRICHE ?

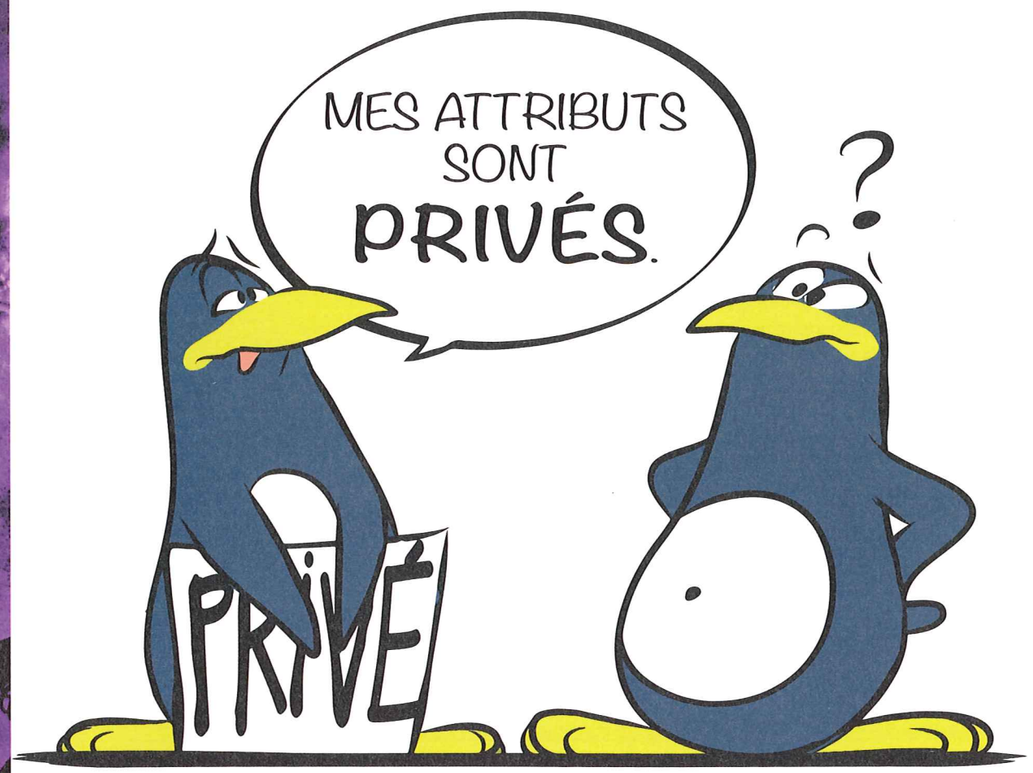
Il est normal que les méthodes du jeu de cartes aient accès aux cartes (pour pouvoir les afficher par exemple). Mais peut-on laisser l'utilisateur faire ce qu'il veut avec les différents objets utilisés ? Si aucun contrôle n'est effectué, on peut tout à fait passer outre les vérifications du constructeur d'une carte et créer une carte sans signification. Testons cela dans un interpréteur interactif :

Fichier

```
>>> from Carte import Carte
>>> c = Carte(2, 3)
>>> print(c)
2 de Pique
>>> c.valeur = 32
>>> c.couleur = "turquoise"
>>> print(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Carte.py", line 16, in __str__
    return str(Carte.valeurs[self.valeur]) + " de " + Carte.couleurs[self.
couleur]
IndexError: tuple index out of range
```

Eh oui, nous obtenons un message d'erreur puisque les valeurs des attributs **valeur** et **couleur** ne peuvent pas être traduites. Ce qui pose donc problème c'est la possibilité d'accéder librement aux attributs. En Python, la philosophie consiste à faire confiance au développeur, mais la théorie de la POO veut que la protection des attributs et des accès aux méthodes soit un élément fondamental. Ici, les attributs sont en

JOUR 2



accès **public** et il est possible de restreindre leur accès au code situé dans la classe **Carte** par une visibilité **privée**. Le schéma suivant illustre les accès possibles en fonction de la position du code et de la visibilité des attributs :

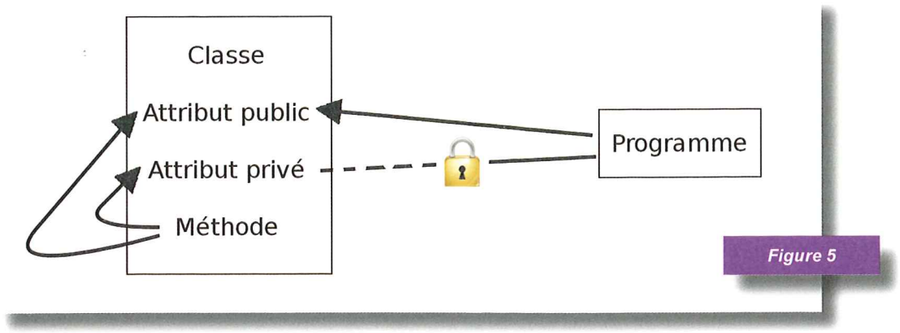


Figure 5

En Python, la visibilité publique est celle qui est appliquée par défaut. Pour donner une visibilité privée à un attribut ou une méthode, il faudra préfixer son nom par un double underscore. Pour bien étudier ce fonctionnement, nous allons définir l'attribut **valeur**

en tant qu'attribut privé et l'attribut **couleur** en tant qu'attribut public (par la suite les deux seront en visibilité privée). Modifiez donc le code de la classe **Carte.py** de la manière suivante en remplaçant toutes les occurrences de **self.valeur** par **self.__valeur** :

Fichier

```
class Carte:
    valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet", "Dame",
"Roi", "As")
    couleurs = ("Coeur", "Carreau", "Trefle", "Pique")


    def __init__(self, val, coul):
        ...
        self.__valeur = val
        self.couleur = coul

    def __str__(self):
        return str(Carte.valeurs[self.__valeur]) + " de " +
            Carte.couleurs[self.couleur]

    def affiche_ascii(self):
        nom = str(Carte.valeurs[self.__valeur]) + " de " +
            Carte.couleurs[self.couleur]
        ...
```

Et maintenant, qui dit test dit... interpréteur interactif bien sûr :

Fichier

```
>>> from Carte import Carte
>>> c = Carte(5, 2)
>>> print(c)
5 de Trefle
>>> print(c.couleur)
2
```

L'attribut **couleur** qui est public est toujours accessible. Jusqu'ici tout est normal.

Fichier

```
>>> print(c.valeur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carte' object has no attribute 'valeur'
```

L'attribut **valeur** n'est plus accessible... Mais quand on y réfléchit bien, cela ne prouve nullement que l'attribut est en visibilité privée : il n'y a plus d'attribut **valeur** puisque nous l'avons remplacé par **__valeur**. Effectuons donc le test sur cet attribut :

Fichier

```
>>> print(c.__valeur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carte' object has no attribute '__valeur'
```

Il n'est toujours pas accessible donc tout va bien.

JOUR 2

En POO, on doit normalement définir l'ensemble des attributs avec une visibilité privée puis, éventuellement, relâcher les contraintes par la suite. Ainsi, pour pouvoir accéder en lecture ou en écriture à un attribut privé, il faudra définir une méthode publique qui permettra de rentrer dans la classe et d'utiliser « depuis l'intérieur » l'attribut privé. Si l'accès à l'attribut ne se fait que depuis la classe, il faudra définir des méthodes d'accès privées.

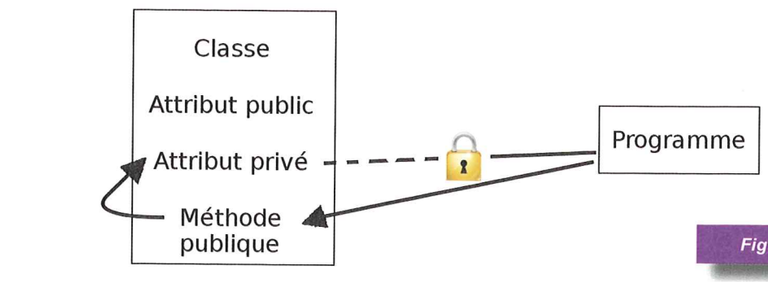


Figure 6

Pour pouvoir accéder en lecture et en écriture à l'attribut privé **`__valeur`**, il va falloir ajouter les deux méthodes suivantes :

Fichier

```
class Carte:
    ...
    def getValeur(self):
        return self.__valeur

    def setValeur(self, val):
        self.__valeur = val
```

Il est maintenant possible d'accéder à cet attribut depuis l'extérieur de la classe... à condition de passer par les méthodes publiques **`getValeur()`** et **`setValeur()`** :

Fichier

```
>>> from Carte import Carte
>>> c = Carte(5, 2)
>>> print(c.getValeur())
5
>>> c.setValeur(14)
>>> print(c.getValeur())
14
```

Une fois que ces méthodes sont définies, il faut les utiliser de partout, y compris dans les méthodes de la classe **`Carte`** qui utilisent l'attribut **`__valeur`**. C'est le principe de l'**`encapsulation`**. Imaginons que vous développiez un programme et qu'une fois terminé, vous distribuiez le code de votre objet à d'autres développeurs qui l'utiliseront dans leurs propres programmes. Vous pouvez continuer à améliorer votre objet, mais il faut que cela soit « transparent » pour les développeurs, il ne faut pas qu'ils aient à réécrire l'ensemble de leur programme. Si vos modifications touchent les attributs, vous ne pourrez pas faire en sorte que les développeurs n'aient pas à modifier eux aussi leur code... à moins d'avoir implémenté l'encapsulation. Les méthodes d'accès et de modification des attributs jouent le rôle d'un tampon entre les appels extérieurs et le mécanisme interne de la classe. Vous pouvez même changer le nom des attributs sans que cela n'ait d'impact sur un programme extérieur.

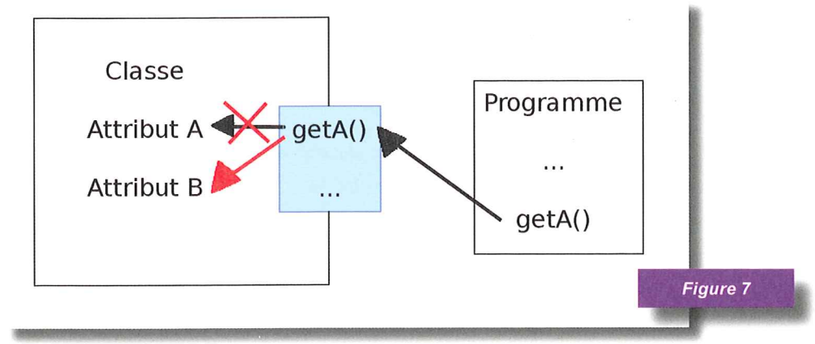


Figure 7

Pour finir, voici une petite astuce Python qui permet d'appeler les méthodes d'accès et de modification de manière « masquée », sans que le développeur n'ait à faire référence de manière implicite à une méthode. C'est le mécanisme des propriétés qui joue le rôle d'un aiguillage :

- ⇨ si j'essaye d'accéder à la propriété en lecture, alors je suis redirigé vers la méthode de lecture ;
- ⇨ si j'essaye d'accéder à la propriété en écriture, alors je suis redirigé vers la méthode d'écriture.

En reprenant le code de l'attribut **`__valeur`**, l'ajout d'une propriété se fait par :

Fichier

```
...
 def __getValeur(self):
    print("Passage dans getValeur !")
    return self.__valeur

def __setValeur(self, val):
    print("Passage dans setValeur !")
    self.__valeur = val

valeur = property(__getValeur, __setValeur)
```

Ces deux méthodes sont privées : on ne peut y accéder qu'en passant par la propriété valeur.

On peut ensuite utiliser **`valeur`** comme s'il s'agissait d'un attribut public avec **`self.valeur`** dans la classe et en utilisant l'attribut **`valeur`** d'une instance à l'extérieur :

Fichier

```
>>> from Carte import Carte
>>> c = Carte(5, 2)
>>> print(c.valeur)
Passage dans getValeur !
5
>>> c.valeur = 12
Passage dans setValeur !
```

Il ne vous reste plus qu'à implémenter l'encapsulation pour tous les attributs des classes **`Carte`** de **`JeuCartes`**... et à vous de voir si les attributs doivent rester privés ou si on doit relâcher les contraintes sur certains d'entre eux. Si vous ne parvenez pas à écrire ce code, vous pourrez trouver la solution dans le résumé à la fin de cette partie. ▮

JOUR 2

Pour récapituler :

- La **modélisation** permet de réfléchir à la conception d'un objet et aux interactions avec d'autres objets.
- La **composition** permet d'utiliser des objets pour en créer de nouveaux.
- Un affichage du type **0x...** indique une adresse mémoire.
- Les fonctions **dir()** et **help()** permettent d'effectuer efficacement des recherches dans la documentation d'un objet.
- Le mécanisme des **exceptions** permet de personnaliser le traitement d'une erreur.
- Tout attribut ou méthode dont le nom commence par un double underscore a une visibilité privée : on ne peut y accéder que depuis l'intérieur de la classe où il est défini. Ne pas confondre avec les noms réservés qui sont eux encadrés par des doubles underscores.
- L'**encapsulation** consiste à n'utiliser que des attributs privés et à en autoriser éventuellement l'accès par la suite à l'aide de méthodes publiques.
- Les **propriétés** permettent de faire un lien élégant entre des méthodes d'accès aux attributs et des attributs privés.

Résumé code :

Voici tout d'abord le code de la classe **Carte** contenu dans le fichier **Carte.py**. Tous les attributs sont privés et nous ne donnons aucun accès depuis l'extérieur. La programmation en Python étant moins stricte au niveau de l'encapsulation, nous n'ajouterons pas de propriété privée pour accéder aux attributs :

```
01: class Carte:
02:     valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet",
"Dame", "Roi", "As")
03:     couleurs = ("Coeur", "Carreau", "Trefle", "Pique")
04:
05:
06:     def __init__(self, val, coul):
07:         if val < 2 or val > 14:
08:             print("Erreur : La valeur d'une carte est comprise
entre 2 et 14")
09:             exit(1)
10:         if coul < 0 or coul > 3:
11:             print("Erreur : Le code couleur d'une carte est
compris entre 0 et 3")
12:             exit(1)
13:         self.__valeur = val
14:         self.__couleur = coul
15:
16:     def __str__(self):
```


Résumé code :

```
17:         return str(Carte.valeurs[self.__valeur]) + " de " +
Carte.couleurs[self.__couleur]
18:
19:     def affiche_ascii(self):
20:         nom = str(Carte.valeurs[self.__valeur]) + " de " +
Carte.couleurs[self.__couleur]
21:         taille = len(nom) + 2
22:         print("/", "-" * taille, "\\", sep="")
23:         print("|", "-" * taille, "|", sep="")
24:         print("|", nom, "|")
25:         print("|", "-" * taille, "|", sep="")
26:         print("\\", "-" * taille, "/", sep="")
```

Ensuite, voici le code de la classe **JeuCartes** du fichier **JeuCartes.py**. Là encore l'attribut sera privé et il est inutile d'implémenter une propriété :

```
01: from Carte import Carte
02: import random
03:
04: class JeuCartes:
05:     def __init__ (self):
06:         self.__cartes = []
07:
08:         for val in range(2, 15):
09:             for coul in range(4):
10:                 self.__cartes.append(Carte(val, coul))
11:
12:     def __str__(self):
13:         cartes_du_jeu = ""
14:         for carte in self.__cartes:
15:             if cartes_du_jeu == "":
16:                 cartes_du_jeu = str(carte)
17:             else:
18:                 cartes_du_jeu += ", " + str(carte)
19:         return cartes_du_jeu
20:
21:     def melanger(self):
22:         random.shuffle(self.__cartes)
23:
24:     def tirer(self):
25:         try:
26:             return self.__cartes.pop(0)
27:         except IndexError as erreur:
28:             print("Il n'y a plus de carte dans le jeu !")
29:             return None
```

Enfin, pour pouvoir tester l'ensemble du code, il faut modifier le fichier **start.py** :

```
01: from JeuCartes import JeuCartes
02:
03: if __name__ == "__main__":
04:     j = JeuCartes()
05:     j.melanger()
06:     print("On tire la première carte :")
07:     print(j.tirer())
```

JOUR 3

QUELLE DIFFÉRENCE ENTRE UN JEU ET UN PAQUET DE CARTES ?

Nous avons créé un jeu de cartes, mais pour que deux joueurs puissent s'affronter, il faut que chacun d'eux dispose d'un paquet de cartes issues du jeu... et non pas de deux jeux de cartes !

JOUR 3

Lorsque nous avons créé nos objets, nous avons peut-être un peu négligé une partie fondamentale : la réflexion. Avant de se lancer dans un projet informatique, il faut un minimum de planification et c'est d'autant plus vrai avec la programmation orientée objet. Comment peut-on imaginer créer ne serait-ce qu'une dizaine d'objets sans savoir comment ils vont interagir entre eux ?

Aujourd'hui, nous allons revenir sur l'étape de modélisation et réfléchir à notre programme dans son intégralité. Cette réflexion fera apparaître des besoins techniques que nous aborderons de manière à résoudre certains problèmes.

1. LA MODÉLISATION, UNE ÉTAPE ESSENTIELLE

Nous avons déjà vu un exemple de pseudo modélisation UML avec le diagramme de classes réalisé pour les classes `Carte` et `JeuCartes`. Pour commencer notre réflexion, nous allons utiliser la même représentation que précédemment pour laquelle les seuls outils indispensables sont un crayon et du papier. Même si par la suite vous souhaitez rendre votre diagramme plus présentable en utilisant un logiciel, la réalisation d'un schéma sur papier a l'avantage de pouvoir être réalisée et corrigée très rapidement.

1.1 En utilisant un crayon et du papier

Pour pouvoir identifier les différents objets de notre projet, voici un rappel de notre objectif. Nous voulons créer un jeu de bataille à deux **joueurs**. Il nous faut un **jeu de cartes** (52 cartes) qui seront distribuées sous forme de **paquets** aux **joueurs**. Les **joueurs** ne regardent pas les **cartes** et à chaque tour ils **retournent** la **carte** se trouvant sur le dessus du **paquet**. Le **joueur** possédant la carte la plus forte **récupère** les deux **cartes** et les place sous son **paquet**. Si les deux **cartes** sont de valeur égale, on dit qu'il y a **bataille** : chaque **joueur** place la **carte** du dessus de son paquet à l'envers sur sa première carte puis place une nouvelle carte par dessus. Le **joueur** ayant la **carte** la plus forte récupère l'ensemble des cartes (en cas de nouvelle égalité, on recommence). La **partie se termine** lorsque l'un des deux joueurs n'a plus de carte.

Dans le paragraphe ci-dessus, j'ai indiqué en gras les termes importants, ceux qui doivent nous aider dans notre modélisation en nous indiquant soit des objets à créer, soit des méthodes à ajouter aux objets. Nous voyons bien que nous aurons besoin d'un objet `Joueur`, mais rien ne nous dit de manière précise quels sont les attributs et les méthodes qui constitueront cet objet. Il y aura des éléments fondamentalement nécessaires (si un joueur n'a pas de paquet de cartes, il ne peut pas jouer) et puis des éléments optionnels qui dépendront de nos choix d'implémentation. On peut se dire qu'un `Joueur` est caractérisé par son nom, son prénom, son palmarès (nombre de victoires et nombre de défaites) et le contenu de son paquet de cartes. Les actions associées à un joueur sont peu nombreuses : un joueur doit être capable de tirer une carte. Possédant ces informations, nous pouvant modéliser la classe `Joueur`. Comme depuis notre dernière modélisation nous avons utilisé l'encapsulation avec les visibilités publique et privée, il faut savoir qu'un attribut ou une méthode sera précédé du signe `-` en notation UML s'il est privé et du signe `+` s'il est public. Pour ne pas encombrer inutilement le diagramme, je ne noterai pas les méthodes d'accès à chaque attribut (`getNomAttribut()` et `setNomAttribut()`) .

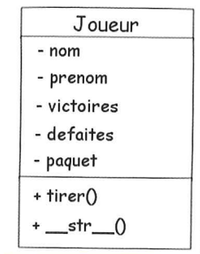


Figure 1

La méthode `__str__()` permettra d'afficher le nom du joueur ainsi que son palmarès complet. Le type de chaque attribut n'est pas précisé, mais un `paquet` sera une instance de la classe `Paquet`, classe qu'il faut maintenant créer de manière à pouvoir utiliser la composition (on utilise un `Paquet` comme attribut de `Joueur`).

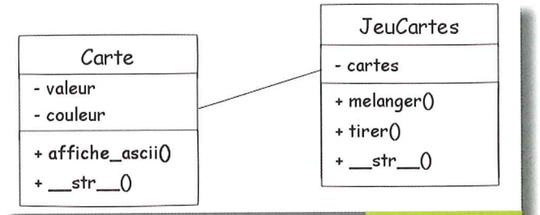


Figure 2

Pour la modélisation de `Paquet`, il faut se rappeler que nous disposons déjà de `JeuCartes` qui utilise la classe `Carte`.

Mais finalement, quelle est la différence fondamentale entre un paquet et un jeu de cartes ? Un paquet est bien composé d'un ensemble de cartes que l'on peut mélanger et on peut également tirer une carte d'un paquet... En fait, la seule différence entre un paquet et un jeu de cartes est qu'un paquet est tout d'abord vide puis qu'on y ajoute des cartes au cours de la distribution des cartes. Un paquet est donc un jeu de cartes particulier, une spécialisation du jeu de cartes. Dans ce cas, on parle d'héritage : nous allons utiliser le code écrit pour `JeuCartes` et modifier les parties différentes et éventuellement ajouter des attributs ou des méthodes. Pour représenter une relation d'héritage en UML, on utilise une flèche qui part de la classe la plus spécialisée vers la classe la plus générale.

Figure 3

La classe `Paquet` hérite donc de `JeuCartes` à laquelle elle rajoute la méthode `ajouter()` permettant d'ajouter une carte au paquet.

L'HÉRITAGE

L'héritage permet de réutiliser du code existant dans le cadre d'une spécialisation de la fonction d'un objet. Il faut faire très attention à ne pas confondre héritage et composition :

- ⇨ un triangle est un polygone particulier (seulement trois sommets), donc la classe Triangle héritera de la classe Polygone ;
- ⇨ une roue permet de construire une moto, donc il s'agit d'un cas de composition où `Roue` sera un attribut de `Moto`.

De manière générale, si lorsque vous dites la phrase « classeA est un(e) classeB particulier(ère) », si celle-ci est correcte c'est qu'il s'agit d'un cas d'héritage et sinon c'est qu'il s'agit d'un cas de composition. Exemple : « Une roue est une moto particulière »... ??? Non, donc il s'agit bien d'une composition. Par contre : « Un triangle est un polygone particulier » : héritage !

À retenir

En notation UML une relation d'héritage entre deux classes est symbolisée par une flèche partant de la classe la plus spécialisée (ou classe fille) et allant vers la classe la plus générale (ou classe mère).

Nous avons schématisé les éléments principaux de notre programme, mais il manque encore la partie centrale, le chef d'orchestre qui fera intervenir tous les éléments pour que la partie de bataille se déroule correctement. Le cœur de notre projet sera donc la classe `PartieBataille` qui gérera les deux joueurs et le jeu de cartes : mélange du jeu, distribution des cartes aux joueurs, puis contrôle de la partie en demandant aux joueurs de mettre leurs cartes en jeu. Cette classe dialoguant avec les autres classes, nous obtenons le diagramme de classes complet du projet.

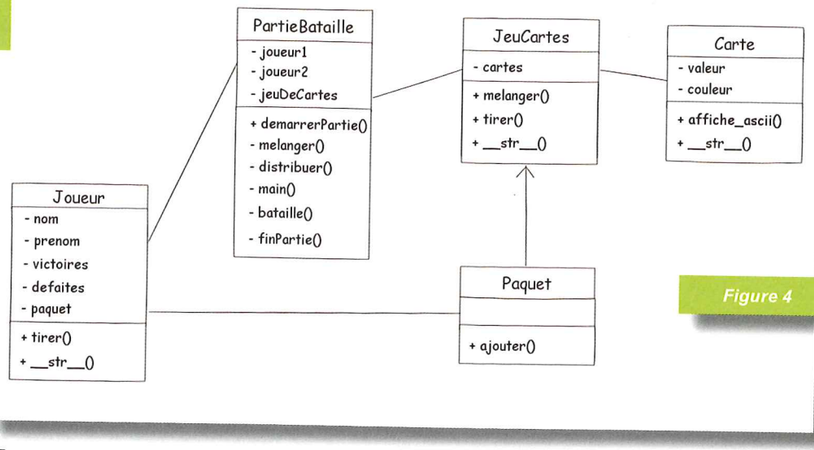


Figure 4

À retenir

Les attributs et méthodes privés ne sont pas hérités.

Vous constaterez sur notre schéma que les compositions ne sont pas simples à lire, il faudrait savoir quelle classe utilise l'autre classe. Pour cela, en UML on note un petit losange noir sur le trait touchant la classe « utilisatrice ». Nous allons voir comment cela est représenté par un logiciel dédié à la création de diagrammes UML.

1.2 En utilisant un logiciel de modélisation

Il existe de très nombreux logiciels de modélisation UML. Parmi ces logiciels certains sont indépendants et d'autres sont liés à Eclipse sous forme d'extension. Si vous tenez à rester dans Eclipse, vous pourrez utiliser par exemple Papyrus. Pour l'installer, rendez-vous dans Eclipse et dans la page du menu ***Help > Install New Software...*** sélectionnez comme source (*Work with*) l'entrée Luna (***Luna - http://download.eclipse.org/releases/luna***) en utilisant la flèche de la liste de sélection. Après mise à jour des logiciels disponibles, cochez la case ***Modeling > Papyrus UML***, puis cliquez sur le bouton ***Next*** (deux fois), acceptez la licence et cliquez sur le bouton ***Finish***.

L'installation est un peu longue, donc vous pourrez en profiter pour faire une pause le temps que tout soit fini, puis il faudra redémarrer Eclipse et vous pourrez alors utiliser l'interface graphique en faisant glisser les différents éléments de votre diagramme.

Comme exemple de logiciel externe, il y a Modelio que vous pourrez télécharger sur **http://www.modelio.org/downloads/download-modelio.html** en sélectionnant votre système d'exploitation et l'architecture de votre machine. Je ne décrirai pas l'installation ni l'utilisation de ce logiciel qui seraient un peu longues dans le cadre de ce hors-série, mais vous pouvez constater que l'on obtient de très jolis diagrammes.

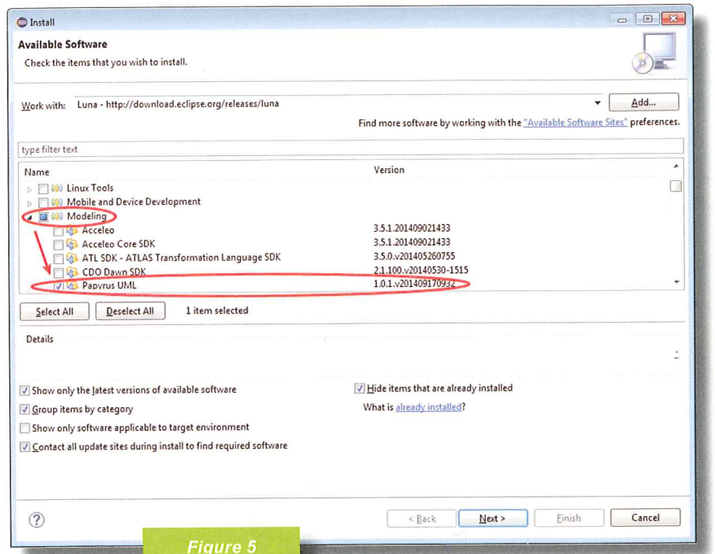


Figure 5

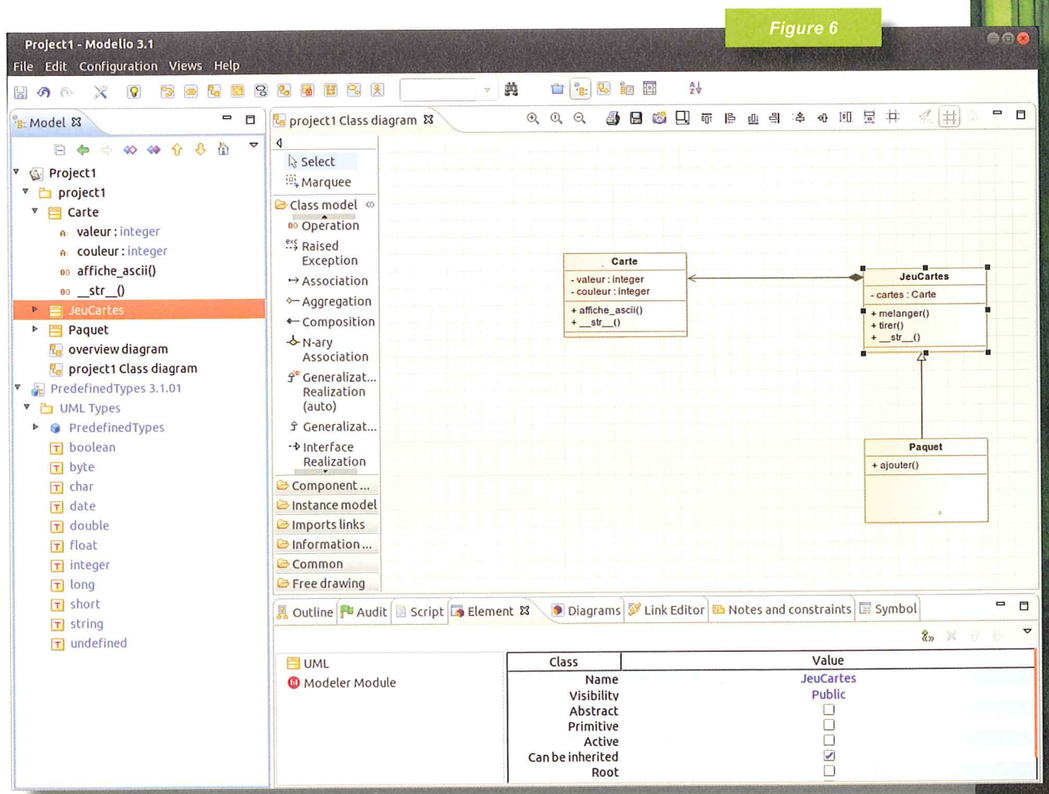
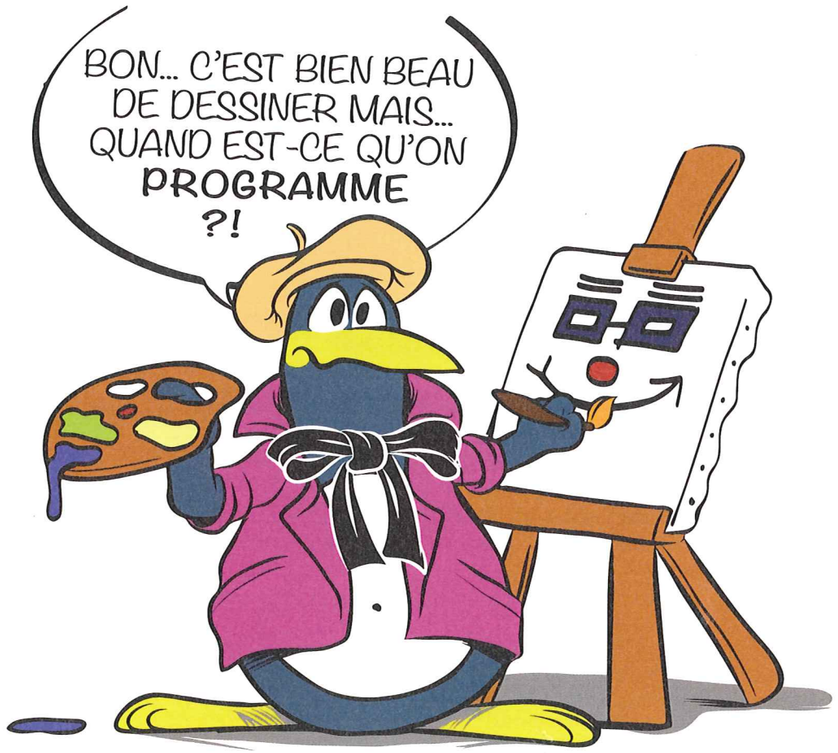


Figure 6

JOUR 3

Dans d'autres langages, le fait d'effectuer la modélisation avec de tels logiciels est très intéressant, car il est possible de générer automatiquement le squelette du code et les méthodes d'accès et de modifications. En Python, soit les logiciels ne sont pas capables d'effectuer une telle génération, soit le code généré est vraiment minimal. L'étape de modélisation est indispensable, mais dans un premier temps vous pouvez amplement vous satisfaire de la méthode la plus simple en prenant votre crayon et votre papier, quitte à scanner le résultat obtenu pour en garder une trace numérique dans votre projet.



2. UTILISATION DE L'HÉRITAGE POUR CRÉER UN PAQUET DE CARTES

Nous allons maintenant voir l'héritage d'un point de vue technique. En Python, pour indiquer qu'une classe est une spécialisation d'une autre classe, qu'elle hérite de ses caractéristiques, on doit le signaler lors de la définition de la classe en indiquant le nom de la classe mère entre parenthèses :

Fichier

```
class MaClasse(ClasseMere):
```


Bien sûr, pour pouvoir utiliser `ClasseMere`, il faudra avoir importé sa définition au préalable :

Fichier

```
from ClasseMere import ClasseMere
```

De plus, dans le constructeur de la classe fille, il faudra faire appel au constructeur de la classe mère. Il existe deux syntaxes pour cela :

- `ClasseMere.__init__(self, parametre_1, ..., parametre_n)` ;
- `super().__init__(parametre_1, ..., parametre_n)`.

Dans la deuxième syntaxe, `super()` fait référence à l'objet « classe mère » et c'est pour cela qu'il est inutile d'ajouter `self` dans les paramètres. Voyons comment utiliser cette syntaxe pour écrire la classe `Paquet` :

Fichier

```
from JeuCartes import JeuCartes          <- Chargement de la classe mère.

class Paquet(JeuCartes):                 <- La classe Paquet hérite de la classe JeuCartes.

    def __init__(self):
        super().__init__(True)           <- Appel du constructeur de la classe mère JeuCartes.

    def ajouter(self, carte):            <- Méthode venant s'ajouter aux méthodes
        self.cartes.append(carte)           héritées de la classe JeuCartes.
```

Lors de l'appel du constructeur, vous remarquerez que nous avons ajouté un paramètre `True` et que dans la méthode nous utilisons un attribut `cartes` alors qu'il n'y a qu'un attribut privé `__cartes`. Cela vient du fait que nous avions écrit le code de la classe `JeuCartes` avant de réfléchir aux différentes classes dont nous aurions besoin dans notre projet et aux interactions entre celles-ci. Du coup, nous sommes obligés de modifier la classe `JeuCartes` :

1. Pour pouvoir créer un jeu de cartes complet ou vide, il faut modifier le constructeur :

Fichier

```
...
05:     def __init__ (self, vide=False):
06:         self.__cartes = []
07:
08:         if not vide:
09:             for val in range(2, 15):
10:                 for coul in range(4):
11:                     self.__cartes.append(Carte(val, coul))
...
```

Nous avons simplement ajouté un paramètre `vide` qui est initialisé à `False` par défaut (si le développeur ne spécifie pas de paramètre lors de la création d'une instance de `JeuCartes`, `vide` aura pour valeur `False`). Par défaut, le jeu sera construit avec les 52 cartes, mais si `vide` vaut `True` alors il ne sera pas rempli.

2 Les attributs privés ne sont pas hérités. Il faut donc créer les méthodes d'accès à `__cartes` et une propriété.

Fichier

```
...
13:     def __getCartes(self):
14:         return self.__cartes
15:     def __setCartes(self, carte):
16:         if len(self.__cartes) > 52:
17:             raise Exception("Jeu complet")
18:         self.__cartes.append(carte)
19:     cartes = property(__getCartes, __setCartes)
...
```

Création d'une exception.

Ici, nous avons ajouté un contrôle lors de l'ajout d'une carte : s'il y a déjà 52 cartes, nous créons une erreur. Plutôt que d'afficher un message et de sortir du programme avec un appel à `exit()`, il vaut mieux utiliser le système des exceptions que nous avons vu hier : un bloc `try`/`except` permet de « capter » une exception pour la traiter de manière personnalisée et là nous voyons la façon de créer une exception à l'aide de l'instruction `raise` suivie de la création d'une instance de la classe `Exception` (c'est l'objet que l'on peut récupérer avec `except`). Si vous souhaitez mettre en place un contrôle plus strict, vous pouvez ajouter la vérification du fait que la carte passée en paramètre n'est pas déjà présente dans le jeu.

Une fois que la propriété `cartes` est définie, il faut bien sûr l'utiliser dans l'ensemble des méthodes de la classe (le code complet est donné à la fin de cette partie).

Nous pouvons dès à présent tester le comportement de notre nouvelle classe à l'aide de l'interpréteur interactif :

Terminal

```
>>> from Paquet import Paquet
>>> p = Paquet()
>>> print(p)

>>> from Carte import Carte
>>> c = Carte(12, 2)
>>> print(c)
Dame de Trefle
>>> p.ajouter(c)
>>> print(p)
Dame de Trefle
```

Vous pouvez même vérifier que l'héritage a bien fonctionné en utilisant une méthode de `JeuCartes` sur l'instance de `Paquet` :

Terminal

```
>>> print(p.tirer())
Dame de Trefle
>>> print(p)
```

À savoir

Plutôt que d'arrêter abruptement le programme en cas d'erreur, il est préférable d'utiliser le mécanisme des exceptions en « levant » une erreur à l'aide de l'instruction `raise` et de laisser le développeur gérer cette erreur en la récupérant avec un bloc `try/catch`.

3. LA SURCHARGE D'OPÉRATEURS

Pour terminer cette journée, nous allons voir un mécanisme amusant appelé « surcharge d'opérateurs ». Vous connaissez les opérateurs `+`, `-`, `<`, `>=`, `==`, etc. Eh bien, il est possible de les utiliser avec des objets que nous créons. En Python, des méthodes spéciales sont associées à ces opérateurs : `__add__()` pour `+`, `__lt__()` pour `<`, etc. Il suffit donc d'écrire le code de ces fonctions pour pouvoir utiliser les opérateurs associés. Prenons l'exemple de l'ajout d'une carte dans un paquet. Pour l'instant, pour réaliser cette opération, nous utilisons la méthode `ajouter()` : `paquet.ajouter(carte)`. Nous souhaitons utiliser une écriture plus élégante qui sera : `paquet + carte`. Il faut donc écrire le code de la méthode `__add__()` :

Fichier

```
...
11:     def __add__(self, carte):
12:         self.ajouter(carte)
```

Ainsi, pour ajouter une carte nous pouvons désormais faire :

Terminal

```
>>> from Paquet import Paquet
>>> from Carte import Carte
>>> dame_trefle = Carte(12, 2)
>>> paquet = Paquet()
>>> paquet + dame_trefle
>>> print(paquet)
Dame de Trefle
```

Vous avez peut-être encore un peu de mal à voir le rapport entre la ligne `paquet + dame_trefle` et la méthode `__add__()`. En fait, ce n'est pas très compliqué : lorsque vous écrivez `paquet + dame_trefle`, l'ordinateur voit en fait `paquet.__add__(dame_trefle)`. Donc dans la définition de la méthode `__add__()`, `self` fait référence à `paquet` et `carte` à `dame_trefle`.

Normalement, pour que le code reste compréhensible on utilise les opérateurs en conservant leur structure initiale. Par exemple, l'addition ne modifie pas ces éléments, mais renvoie un résultat : exactement le contraire de ce que nous avons fait... mais sur ce code nous pouvons nous le permettre, car il est assez court et la lecture reste simple. Par contre, vous ne pourrez pas modifier complètement le comportement d'un opérateur. Pour une addition, il faut deux éléments, vous ne pourrez pas définir une addition ne faisant intervenir qu'un seul élément. ■

À retenir

Lorsque l'on surcharge un opérateur, il est conseillé (mais non obligatoire) de lui garder son « sens ». Par exemple, est un test qui renvoie une valeur booléenne ; le code sera plus simple à lire si cet opérateur est utilisé dans le même cadre logique.

JOUR 3

Pour récapituler :

- Il est essentiel de passer par une phase de **modélisation** pour bien voir quels sont les objets qui vont intervenir dans le projet et comment ils vont interagir entre eux. Pour cela, on réalise un **diagramme de classes** sur papier ou à l'aide d'un logiciel adapté.
- L'**héritage** permet de réutiliser un objet dans une optique de spécialisation de sa fonction.
- Il faut faire très attention de ne pas confondre **composition** et **héritage**.
- Plutôt que de traiter arbitrairement une erreur en interrompant le programme, il faut utiliser le mécanisme des exceptions permettant au développeur utilisant la classe de personnaliser le traitement des erreurs. Une exception est lancée ou « levée » en utilisant l'instruction `raise`.
- Il est possible de paramétrer le comportement des opérateurs appliqués sur un objet grâce à la **surcharge des opérateurs**. Des méthodes spéciales, encadrées par des doubles underscores, doivent être définies (voir **https://docs.python.org/3/library/operator.html**).

Résumé code :

Le fichier `Carte.py` n'a pas été modifié.

```
01: class Carte:
02:     valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet",
"Dame", "Roi", "As")
03:     couleurs = ("Coeur", "Carreau", "Trefle", "Pique")
04:
05:
06:     def __init__(self, val, coul):
07:         if val < 2 or val > 14:
08:             print("Erreur : La valeur d'une carte est comprise entre
2 et 14")
09:             exit(1)
10:         if coul < 0 or coul > 3:
11:             print("Erreur : Le code couleur d'une carte est compris
entre 0 et 3")
12:             exit(1)
13:         self.__valeur = val
14:         self.__couleur = coul
15:
16:     def __str__(self):
17:         return str(Carte.valeurs[self.__valeur]) + " de " +
Carte.couleurs[self.__couleur]
18:
19:     def affiche_ascii(self):
20:         nom = str(Carte.valeurs[self.__valeur]) + " de " +
Carte.couleurs[self.__couleur]
21:         taille = len(nom) + 2
```



```
22:         print("/", "-" * taille, "\\", sep="")
23:         print("|", "-" * taille, "|", sep="")
24:         print("|", nom, "|")
25:         print("|", "-" * taille, "|", sep="")
26:         print("\\", "-" * taille, "/", sep="")
```

Dans le fichier `JeuCartes.py`, nous avons modifié le constructeur et ajouté l'encapsulation :

```
01: from Carte import Carte
02: import random
03:
04: class JeuCartes:
05:     def __init__ (self, vide=False):
06:         self.__cartes = []
07:
08:         if not vide:
09:             for val in range(2, 15):
10:                 for coul in range(4):
11:                     self.__cartes.append(Carte(val, coul))
12:
13:     def __getCartes(self):
14:         return self.__cartes
15:     def __setCartes(self, carte):
16:         if len(self.__cartes) > 52:
17:             raise Exception("Jeu complet")
18:         self.__cartes.append(carte)
19:     cartes = property(__getCartes, __setCartes)
20:
21:     def __str__(self):
22:         cartes_du_jeu = ""
23:         for carte in self.cartes:
24:             if cartes_du_jeu == "":
25:                 cartes_du_jeu = str(carte)
26:             else:
27:                 cartes_du_jeu += ", " + str(carte)
28:         return cartes_du_jeu
29:
30:     def melanger(self):
31:         random.shuffle(self.cartes)
32:
33:     def tirer(self):
34:         try:
35:             return self.cartes.pop(0)
36:         except IndexError as erreur:
37:             print("Il n'y a plus de carte dans le jeu !")
38:             return None
```

Le fichier `Paquet.py` est lui tout récent puisqu'il s'agit de l'implémentation de la classe `Paquet` :

```
01: from JeuCartes import JeuCartes
02:
03: class Paquet(JeuCartes):
04:
05:     def __init__(self):
06:         super().__init__(True)
07:
08:     def ajouter(self, carte):
09:         self.cartes.append(carte)
10:
11:     def __add__(self, carte):
12:         self.ajouter(carte)
```

JOUR 4

UNE MÊME OSSATURE, MAIS DES JEUX DIFFÉRENTS

Aujourd'hui, nous allons nous éloigner de notre objectif de création d'un jeu de bataille. Nous avons pu découvrir l'héritage qui permet de spécialiser des objets tout en récupérant leur fonctionnement de base. Pourquoi ne pas utiliser ce mécanisme pour créer différents jeux de cartes ?

JOUR 4

Lorsque nous avons créé notre jeu de cartes, nous nous sommes focalisés sur notre objectif et donc sur le jeu de bataille. En ce sens, notre jeu contient 52 cartes et nos cartes sont composées d'une couleur et d'une valeur. La classe `Carte` est suffisamment générique pour pouvoir être spécialisée (on peut aussi parfois rencontrer le terme **dérivé**) en une classe décrivant un autre type de carte. La classe `JeuCartes` par contre n'est pas suffisamment générique puisqu'elle initialise le paquet avec 52 cartes d'un type particulier (trèfle, pique, cœur, carreau pour les valeurs allant de 2 à l'As). Nous revoyons encore ici l'importance de la modélisation et de l'implémentation : il faut réfléchir à des objets permettant de répondre à l'objectif que l'on se fixe dans l'instant, mais qui soient en même temps suffisamment génériques pour pouvoir s'adapter à d'autres situations.

Dans un premier temps, nous allons reprendre la classe `JeuCartes` pour la rendre plus générique. Dans un deuxième temps, nous utiliserons cette classe pour créer un jeu « classique » de 52 cartes avant de l'adapter pour pouvoir jouer à un jeu d'héroïc-fantasy de type *Magic The GatheringTM©®*.

1. UN JEU DE CARTES PLUS GÉNÉRIQUE

Si l'on regarde notre modélisation de la classe `JeuCartes`, de prime abord on peut se demander comment la rendre plus générique (Figure 1).

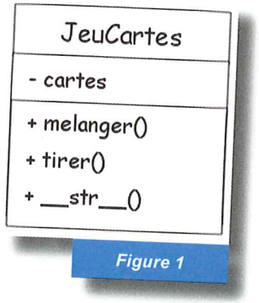


Figure 1

En effet, rien n'indique ici qu'il s'agit d'un jeu de 52 cartes et on retrouve des actions qui doivent correspondre à n'importe quel jeu de cartes : mélanger le jeu, tirer une carte et afficher le contenu du paquet. Ce n'est pas vraiment la modélisation qui est ici fautive, mais plutôt l'implémentation que nous avons réalisée à partir de cette dernière : le constructeur initialise le jeu avec 52 cartes d'un format bien défini et la seule option que nous avons ajoutée est la possibilité d'obtenir un jeu vide.

Comme toujours en informatique il n'y a pas une unique solution :

1. On peut conserver notre implémentation en considérant que la classe `JeuCartes` est par défaut le jeu « classique » et que tout jeu de cartes héritant de cette classe utilisera le constructeur avec le paramètre `vide` à `True` de manière à pouvoir initialiser par lui-même les cartes présentes dans le jeu. Dans cette solution, le remplissage du jeu s'effectue dans le constructeur. Par rapport à notre implémentation, il n'y aurait donc rien à modifier, tout fonctionne correctement.

2. On peut envisager une solution un peu plus élégante en « sortant » du constructeur les instructions permettant de remplir le jeu de cartes. On ferait ainsi appel à une méthode qui pourrait être surchargée dans les classes filles. Mais cette méthode, que l'on pourrait appeler `initialiser()`, ne serait pas définie directement pour la classe `JeuCartes` et du coup cette classe serait inutilisable tant que la méthode `initialiser()` ne serait pas écrite. Dans ce cas, on parle de `classe abstraite` : une classe qui n'est pas complètement définie et qui ne peut pas être utilisée pour créer une instance. Pour pouvoir l'utiliser, il faut forcément créer une classe fille qui définit la ou les méthodes manquantes.

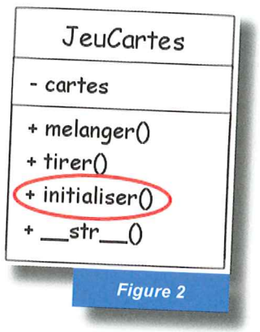


Figure 2

Comme la première solution fonctionne déjà, nous allons nous pencher sur l'implémentation de la seconde. Tout d'abord, il faut savoir que la notion de classe abstraite n'existe pas vraiment en Python et que nous allons donc utiliser une astuce pour l'obtenir. Au niveau de la modélisation, nous n'ajoutons qu'une seule méthode (`initialiser()` qui ne sera pas définie dans la classe mère) (Figure 2).

Vous vous demandez sans doute comment nous allons réussir à empêcher la création d'instances d'une classe... Voyons cela sur une petite classe de test dans l'interpréteur interactif :

Terminal
```
>>> class MaClasse:
...     def __init__(self):
...         print("constructeur")
...
>>> objet = MaClasse()
constructeur
>>> objet.__class__
<class '__main__.MaClasse'>
```

L'attribut réservé `__class__` contient le type de la classe (et donc son nom). Si nous effectuons un test dans le constructeur pour savoir si ce dernier est appelé depuis la classe mère ou une classe fille nous pouvons interdire son accès :

Terminal
```
>>> class MaClasse:
...     def __init__(self):
...         if self.__class__ is MaClasse:
...             raise Exception("Construction directe interdite!")
...         else:
...             print("constructeur")
...
>>> objet = MaClasse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
Exception: Construction directe interdite!
```

Nous ne pouvons pas créer d'instance de `MaClasse`. Voyons si une classe fille peut être utilisée :

Terminal
```
>>> class ClasseFille(MaClasse):
...     def __init__(self):
...         print("Appel depuis la classe fille")
...         super().__init__()
...
>>> objet = ClasseFille()
Appel depuis la classe fille
constructeur
```

Ça marche ! Nous pouvons utiliser cette structure pour modifier le code de la classe **JeuCartes** :

Fichier

```
01: import random
02:
03:
04: class JeuCartes:
05:     def __init__(self, vide=False):
06:         if self.__class__ is JeuCartes:
07:             raise Exception("Creation interdite!")
08:         else:
09:             self.__cartes = []
10:             self.initialiser()
11:
12:     def __getCartes(self):
13:         return self.__cartes
14:     def __setCartes(self, carte):
15:         self.__cartes.append(carte)
16:     cartes = property(__getCartes, __setCartes)
17:
18:     def __str__(self):
19:         cartes_du_jeu = ""
20:         for carte in self.cartes:
21:             if cartes_du_jeu == "":
22:                 cartes_du_jeu = str(carte)
23:             else:
24:                 cartes_du_jeu += ", " + str(carte)
25:         return cartes_du_jeu
26:
27:     def melanger(self):
28:         random.shuffle(self.cartes)
29:
30:     def tirer(self):
31:         try:
32:             return self.cartes.pop(0)
33:         except IndexError:
34:             print("Il n'y a plus de carte dans le jeu !")
35:             return None
36:
37:     def initialiser(self):
38:         pass
```

Création d'une instance de JeuCartes interdite.

Appel de la méthode abstraite (non définie) permettant d'initialiser le jeu.

Méthode définie, mais sans contenu.

Pour définir la méthode abstraite, il faut employer l'instruction **pass** : nous indiquons que cette fonction ne fait rien. Bien sûr, si dans la classe fille nous ne redéfinissons pas le comportement de la méthode **initialiser()**, nous pourrons quand même créer une instance de la classe (dans d'autres langages ça serait impossible). En Python, même en simulant le comportement d'une classe abstraite, nous n'obtenons pas un fonctionnement très strict. Mais cela va nous servir : la classe **Paquet** n'a pas besoin de phase d'initialisation. Il faudra seulement modifier le constructeur de manière à ne plus lui transmettre de paramètre :

Fichier

```
...
06:     def __init__(self):
07:         super().__init__()
...
```

Comme d'habitude nous pouvons tester notre code dans l'interpréteur interactif :

Terminal

```
>>> from Paquet import Paquet
>>> from Carte import Carte
>>> p = Paquet()
>>> print(p)

>>> c = Carte(11, 1)
>>> p.ajouter(c)
>>> print(p)
Valet de Carreau
```

Nous avons réussi à obtenir un jeu complètement générique. Au passage, j'en profite pour vous faire remarquer que Python est un langage très permissif qui se base énormément sur l' « intelligence » des développeurs : on vous dit qu'un jeu de cartes contient des cartes, mais rien ne vous empêche d'insérer n'importe quel type d'objet dans un paquet (techniquement, on pourrait réaliser un contrôle, mais ce serait contraire à la philosophie de Python).

Terminal

```
>>> p.ajouter(24)
>>> p.ajouter("hello!")
>>> print(p)
Valet de Carreau, 24, hello!
```

C'est donc à vous de veiller à bien insérer des cartes dans le paquet.

2. UN JEU CLASSIQUE

Pour jouer à bataille, nous aurons besoin d'un jeu « classique » de 52 cartes. Pour cela nous allons devoir créer une nouvelle classe qui va hériter de **JeuCartes** et définir la méthode **initialiser()** en réutilisant le code que nous avions écrit dans le constructeur de l'ancienne version de la classe mère. La classe sera stockée dans un fichier **JeuClassique.py** :

Fichier

```
01: from JeuCartes import JeuCartes
02: from Carte import Carte
03:
```

À retenir

Une classe abstraite est une classe qui n'est pas complètement définie : le code de certaines méthodes est « mis en attente » pour être écrit dans les classes filles. On ne pourra pas créer d'instance de la classe mère (puisqu'incomplète).

JOUR 4

Fichier

```
04:
05: class JeuClassique(JeuCartes):
06:     def __init__(self):
07:         super().__init__()
08:
09:     def initialiser(self):
10:         for val in range(2, 15):
11:             for coul in range(4):
12:                 self.cartes.append(Carte(val, coul))
```

La classe JeuClassique hérite de JeuCartes.

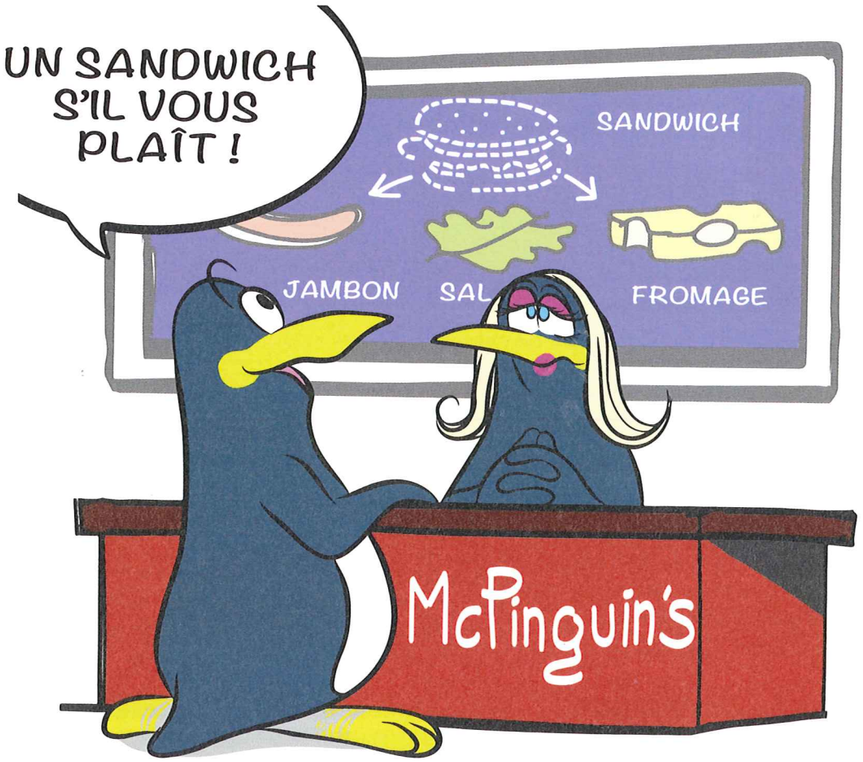
Appel du constructeur de la classe mère.

Surcharge (ou redéfinition) de la méthode initialiser().

Le test montre que nous obtenons le même fonctionnement qu'au départ (mais avec un code plus évolutif tout de même) :

Terminal

```
>>> from JeuClassique import JeuClassique
>>> j = JeuClassique()
>>> j.melanger()
>>> print(j)
3 de Pique, As de Carreau, ..., 4 de Trefle
```



C'est un mécanisme qui paraît un peu magique quand on le voit fonctionner pour la première fois. Si la structure de la classe mère a été correctement pensée, la programmation des autres classes se fait pratiquement toute seule.

3. UN JEU PLUS EXOTIQUE

Passons maintenant à notre jeu de cartes de type *Magic The GatheringTM©®* et contenant : une illustration, une couleur (mana), une valeur (coût de lancement), une description et éventuellement des points d'attaque et des points de vie. Dans ce cadre, notre objet `CarteMagic` va hériter de `Carte`, utiliser les attributs `valeur` et `couleur` et ajouter d'autres attributs. `JeuCartes` est ensuite suffisamment générique pour permettre de créer un jeu de cartes *Magic*.

Nous avons donc exactement le même problème que précédemment pour le jeu de cartes : `Carte` n'est pas assez générique ! Les deux attributs de classe `valeurs` et `couleurs` ainsi que le constructeur sont adaptés à des cartes « classiques ». La solution consistera là encore à rendre la classe plus générique et à écrire une classe fille qui reprendra le comportement de la classe initiale. Pourtant, qu'est-ce que l' « essence » même d'une carte ? Quel que soit le jeu, il faudra bien lui attribuer une valeur et une couleur. En effet, si l'on y réfléchit bien, même dans un jeu des sept familles, il y a une notion de valeur (père, fils, etc.) et de couleur (la famille). L'aspect générique de la classe viendra donc essentiellement des phases d'initialisation qui ne seront plus contenues que dans les classes filles, la classe mère devenant abstraite.

Le code suivant utilise le même principe que celui employé avec la classe `JeuCartes` pour empêcher la création directe d'une instance :

JOUR 4

Fichier

```
01: class Carte:
02:     valeurs = None
03:     couleurs = None
04:
05:
06:     def __init__(self, val, coul):
07:         if self.__class__ is Carte:
08:             raise Exception("Création interdite!")
09:
10:         self.validation(val, coul)
11:         self.__valeur = val
12:         self.__couleur = coul
13:
14:     def getValeur(self):
15:         return self.__valeur
16:     def setValeur(self, val):
17:         self.__valeur = val
18:     valeur = property(getValeur, setValeur)
19:
20:     def getCouleur(self):
21:         return self.__couleur
22:     def setCouleur(self, coul):
23:         self.__couleur = coul
24:     couleur = property(getCouleur, setCouleur)
25:
26:     def validation(self, val, coul):
27:         pass
28:
29:     def __str__(self):
30:         return str(Carte.valeurs[self.valeur]) + " de " + Carte.couleurs[self.couleur]
31:
32:     def affiche_ascii(self):
33:         nom = str(Carte.valeurs[self.valeur]) + " de " + Carte.couleurs[self.couleur]
34:         taille = len(nom) + 2
35:         print("/", "-" * taille, "\\", sep="")
36:         print("|", "-" * taille, "|", sep="")
37:         print("|", nom, "|")
38:         print("|", "-" * taille, "|", sep="")
39:         print("\\", "-" * taille, "/", sep="")
```

Les attributs de classe seront initialisés dans les classes filles.

La classe Carte ne peut plus être utilisée directement.

Appel de la méthode validation()

Méthodes d'accès et de modification des attributs.

qui sera définie dans la classe fille.

Utilisation des propriétés pour accéder aux attributs.

Maintenant que nous possédons une classe **Carte** plus générique, nous devons créer une nouvelle classe **CarteClassique** qui reprendra le fonctionnement de l'ancienne version de la classe **Carte**. Pour cela, cette classe héritera de la classe **Carte** et permettra d'initialiser les attributs de classe **valeurs** et **couleurs** ainsi que de définir la méthode **validation()** :

Fichier

```
01: from Carte import Carte
02:
03:
04: class CarteClassique(Carte):
05:
06:     def __init__(self, val, coul):
07:         Carte.valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10, "Valet", "Dame", "Roi", "As")
08:         Carte.couleurs = ("Coeur", "Carreau", "Trefle", "Pique")
09:         super().__init__(val, coul)
10:
11:     def validation(self, val, coul):
12:         if val < 2 or val > 14:
13:             print("Erreur : La valeur d'une carte est comprise entre 2 et 14")
14:             exit(1)
15:         if coul < 0 or coul > 3:
16:             print("Erreur : Le code couleur d'une carte est compris entre 0 et 3")
17:             exit(1)
```

Import de la classe mère Carte.

Héritage de la classe Carte.

Définition des attributs de classe.

Appel du constructeur de la classe mère.

Définition de la fonction de validation.

Lorsque l'on va créer une instance de la classe **CarteClassique**, les attributs de classe **valeurs** et **couleurs** seront initialisés, ce qui permettra de les utiliser lors de l'appel au constructeur de la classe mère **Carte** qui appellera lui-même la méthode **validation()** définie dans **CarteClassique**.

Nous pourrions améliorer encore un peu notre code : si l'on regarde le contenu de la méthode **validation**(), on s'aperçoit que l'on n'utilise aucun attribut et aucun appel à méthode. Quel intérêt donc de passer en paramètre **self** ? Nous n'avons ici besoin que d'une « simple » fonction, un élément de code qui effectuera exactement la même tâche quelle que soit l'instance qui l'appelle, quitte à ce qu'il puisse être appelé sans instance du tout. On appelle ce type de méthodes des **méthodes de classes** ou **méthodes statiques**.

Pour les déclarer, il faut utiliser un mécanisme particulier : les décorateurs. Nous ne rentrerons pas ici dans le détail de ce mécanisme, mais sachez que vous devrez ajouter une instruction avant la signature de la méthode qui indiquera qu'il s'agit d'une méthode de classe. Ensuite, il sera possible d'appeler cette méthode en la préfixant par le nom de la classe, tout comme pour les attributs de classe :

Fichier

```
01: class Carte:
02:     valeurs = None
03:     couleurs = None
04:
05:
06:     def __init__(self, val, coul):
07:         if self.__class__ is Carte:
08:             raise Exception("Création interdite!")
09:
```

Fichier

```
10:         self.__class__.validation(val, coul)
```

L'appel de la méthode de classe se fait en préfixant son nom par le nom de la classe « appelante » (si on crée une instance de CarteClassique, self.__class__ représentera CarteClassique).

```
11:         self.__valeur = val
12:         self.__couleur = coul
13:
...
26:     @staticmethod
27:     def validation(val, coul):
```

Décorateur indiquant que la méthode est une méthode de classe.

Déclaration de la méthode de classe (absence de self dans la liste des paramètres).

```
28:         pass
...
```

Cette modification entraîne bien sûr la modification de la classe **`CarteClassique`** :

Fichier

```
01: from Carte import Carte
02:
03:
04: class CarteClassique(Carte):
...
11:     @staticmethod
12:     def validation(val, coul):
...
```

Nous pouvons effectuer des tests pour bien voir que la méthode de classe peut être appelée sans instance :

Terminal

```
>>> from CarteClassique import CarteClassique
>>> CarteClassique.validation(2, 6)
Erreur : Le code couleur d'une carte est compris entre 0 et 3
```

Bien sûr, l'appel à cette méthode depuis une instance fonctionne également (même si aucune information de l'instance ne sera employée pour le traitement) :

Terminal

```
>>> from CarteClassique import CarteClassique
>>> c = CarteClassique(2, 1)
>>> c.validation(2, 6)
Erreur : Le code couleur d'une carte est compris entre 0 et 3
```

À retenir

Une méthode de classe (ou méthode statique) est une méthode qui n'est pas liée à une instance : on peut donc l'appeler avant même d'avoir créé une instance de classe. Comme l'existence de l'instance n'est pas obligatoire, la méthode ne peut accéder aux attributs autres que les attributs de classe. Ceci se voit très simplement d'un point de vue syntaxique : puisque `self` n'est pas présent dans la liste des paramètres, il est impossible d'accéder à un paramètre `parametre` à l'aide de `self.parametre`.

Maintenant que nous avons modifié la classe **`Carte`** et ajouté la classe **`CarteClassique`**, il faudra bien sûr modifier la classe **`JeuClassique`** pour utiliser **`CarteClassique`** à la place de **`Carte`**.

Tout ce travail nous permet d'écrire la classe **`CarteMagic`** qui, pour rappel, doit contenir les informations suivantes : une illustration, une couleur (mana), une valeur (coût de lancement), une description et éventuellement des points d'attaque et des points de vie.

Fichier

```
01: from Carte import Carte
02:
03:
04: class CarteMagic(Carte):
05:     def __init__(self, cout, mana, illustration, description, attaque=None, pv=None):
06:         Carte.valeurs = (0, 1, 2, 3, 4, 5, 6)
07:         Carte.couleurs = ("plaine", "ile", "montagne", "forêt", "marais")
```

Redéfinition des valeurs et couleurs.

```
08:         super().__init__(cout, mana)
09:         self.__illustration = illustration
10:         self.__description = description
11:         self.__attaque = attaque
12:         self.__pv = pv
13:
14:     def getIllustration(self):
15:         return self.__illustration
16:     def setIllustration(self, illustration):
17:         self.__illustration = illustration
18:     illustration = property(getIllustration, setIllustration)
19:
...
24:     description = property(getDescription, setDescription)
25:
...
30:     attaque = property(getAttaque, setAttaque)
31:
...
36:     pv = property(getPv, setPv)
37:
38:     @staticmethod
39:     def validation(cout, mana):
40:         if mana not in Carte.couleurs:
41:             print("Erreur : {} n'est pas un type de mana valide".format(mana))
42:             exit(1)
43:         if cout < 0 or cout > 6:
44:             print("Erreur : Les valeurs doivent être comprises entre 0 et 6")
45:             exit(1)
46:
```

Ajout des nouveaux attributs.

Encapsulation des attributs.

Redéfinition de la méthode validation.

Fichier

```
47:     def __str__(self):
48:         return "Carte {} : {}".format(self.couleur, self.description)
```

Redéfinition de la méthode __str__ pour un affichage correspondant aux informations contenues par la carte.

Nous avons maintenant la possibilité d'utiliser ce nouveau type de carte :

Terminal

```
>>> from CarteMagic import CarteMagic
>>> c = CarteMagic(5, "plaine", "db.png", "Dragon des plaines : blabla",
5, 10)
>>> print(c)
Carte plaine : Dragon des plaines : blabla
```

Vous l'aurez sans doute noté, l'attribut **mana** s'appelle **couleur** et l'attribut **cout** s'appelle **valeur**. Il pourrait être intéressant de renommer les anciens attributs pour une meilleure compréhension du code. Si vous adoptez cette démarche, il faudra toutefois être prudent : toutes les méthodes héritées qui utilisaient au moins l'un des attributs **couleur** ou **valeur** ne fonctionneront plus (puisqu'ils n'existent plus). Je donne seulement des pistes pour ceux d'entre vous qui souhaiteraient explorer cette voie :

- l'attribut **__dict__** d'un objet est un dictionnaire qui contient les références à tous les attributs de l'objet ;
- l'instruction **del** permet de détruire un objet (et donc un attribut).

Ne perdez quand même pas trop de temps sur cet exercice si vous voulez finir ce hors-série en une semaine...

Pour conclure cette journée encore chargée, voici le nouveau diagramme de classes (simplifié) correspondant à toutes les modifications que nous avons effectuées. Cette fois-ci, j'ai utilisé le logiciel Modelio pour le réaliser (Figure 3).

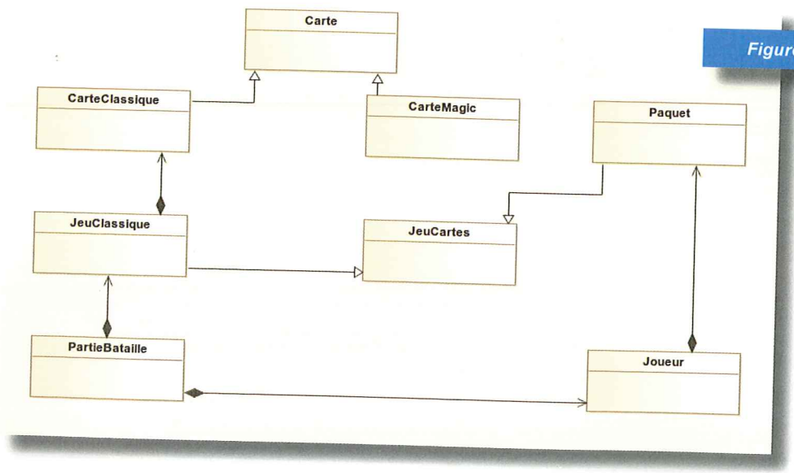
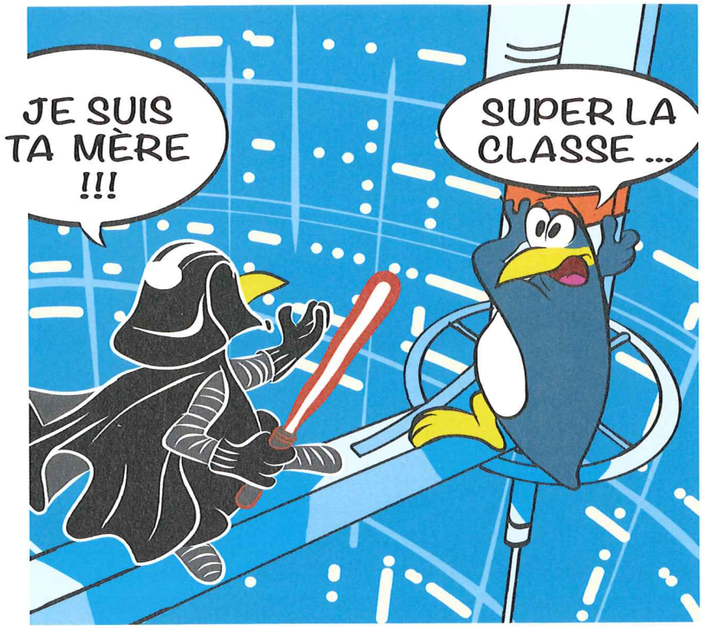


Figure 3

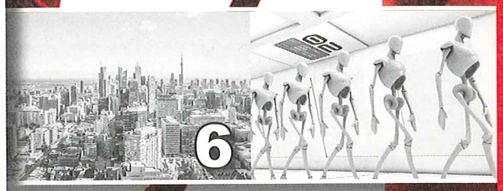


Pour récapituler :

- Pensez à concevoir vos classes de manière à ce qu'elles soient les plus **génériques** possible. Cela vous permettra par exemple de mettre en place simplement un héritage en cas de besoin.
- Les **classes abstraites** sont des classes qui ne peuvent pas être instanciées. En Python, les classes abstraites n'existent pas, mais il est possible de simuler leur comportement.
- Les **méthodes de classe** sont des méthodes qui peuvent être appelées même si aucune instance de la classe où elles sont définies n'existe. Seuls les attributs de classe peuvent être utilisés par une méthode de classe (une méthode « traditionnelle » peut employer les attributs de classe et les attributs « standards »).

Résumé code :

Pas de résumé pour ce jour-ci, demain nous allons mettre en commun toutes les parties déjà codées et nous aurons donc une vision globale du projet.



JOUR 5

JOUER CONTRE L'ORDINATEUR

Nous avons créé différents objets qui permettent de manipuler des cartes. Il reste maintenant à faire en sorte que l'on puisse utiliser ces objets pour jouer.

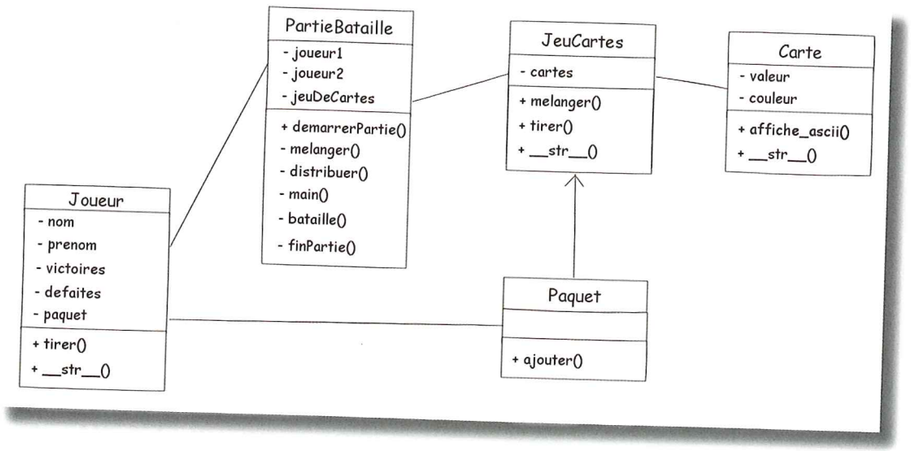
1911

JOUR 5

Dans le diagramme que nous avons créé dans les jours précédents, nous avons imaginé un objet **Joueur** permettant de manipuler un paquet de cartes et un objet **JeuBataille** qui représente l'« intelligence » du jeu, l'arbitre qui permet à deux joueurs de s'affronter. Nous allons créer ces deux classes qui vont nous permettre de jouer contre l'ordinateur.

1. LE JOUEUR

Pour pouvoir faire une partie de bataille, nous allons avoir besoin de deux joueurs. Lors de la quatrième journée, nous avions déjà établi le diagramme de cette classe qui doit comporter les cinq attributs **nom**, **prenom**, **victoires** (nombre de victoires), **defaites** (nombre de défaites) et **paquet** ainsi que les méthodes **tirer()** et **__str__()**. Nul besoin de connaissances supplémentaires pour créer cette classe (que je vous recommande d'ailleurs d'essayer de coder avant de lire la suite). Pour rappel, voici le diagramme que nous avions créé :



Avant de nous lancer dans le codage, réfléchissons à nouveau sur les actions que va pouvoir effectuer le joueur :

- indiquer son palmarès (ce sera effectué dans **__str__()**) ;
- tirer une carte de son paquet pour pouvoir jouer (c'est le rôle de la méthode **tirer()** qui fera appel à la méthode **tirer()** de **JeuCartes** dont hérite **Paquet**). Appliquer une méthode « tirer » sur un joueur, ça fait un peu bizarre quand même, non ? Je vous propose donc de renommer cette méthode en **tirerCarte()** ;
- en cas de victoire, ajouter une carte au paquet... Aïe ! Voilà un oubli ! Mais comme vous le voyez, en réfléchissant sur chaque objet au fur et à mesure de l'avancement du projet, vous pouvez préciser leur fonctionnement et donc savoir plus précisément comment les implémenter. J'ai ajouté ce cas pour vous montrer qu'il y a moyen de réparer une erreur, mais ne perdez pas de vue qu'il est quand même plus confortable d'avoir réalisé correctement le diagramme du premier coup... Nous devons implémenter ici une méthode supplémentaire **ajouterCarte()** (il faut l'ajouter au diagramme de classes qui vous servira de documentation pour faire évoluer le code par la suite ou tout simplement pour le maintenir).

Nous voilà prêts pour créer un **Joueur** :

Fichier

```
01: from Paquet import Paquet
02:
03: class Joueur:
04:
05:     def __init__(self, nom, prenom):
06:         self.__nom = nom
07:         self.__prenom = prenom
08:         self.__victoires = 0
09:         self.__defaites = 0
10:         self.__paquet = Paquet()
11:
12:     def getNom(self):
13:         return self.__nom
14:     nom = property(getNom)
15:
16:     def getPrenom(self):
17:         return self.__prenom
18:     prenom = property(getPrenom)
19:
20:     def getVictoires(self):
21:         return self.__victoires
22:     def setVictoires(self, n):
23:         self.__victoires = n
24:     victoires = property(getVictoires, setVictoires)
25:
26:     def getDefaites(self):
27:         return self.__defaites
28:     def setDefaites(self, n):
29:         self.__defaites = n
30:     defaites = property(getDefaites, setDefaites)
31:
32:     def getPaquet(self):
33:         return self.__paquet
34:     paquet = property(getPaquet)
35:
36:     def tirerCarte(self):
37:         return self.paquet.tirer()
38:
39:     def ajouterCarte(self, carte):
40:         self.paquet.ajouter(carte)
41:
42:     def __str__(self):
43:         return "{} {}\nPalmarès: {} défaite(s) et {} victoire(s)\n{}".format(
44:             self.prenom, self.nom, self.defaites, self.victoires, str(self.paquet))
```

Définition des attributs.

Encapsulation des attributs. Notez que certains attributs ne sont accessibles qu'en lecture.

Appel de la méthode tirer() de la classe Paquet.

Appel de la méthode ajouter() de la classe Paquet.

Construction de la chaîne indiquant les informations du joueur.

Comme toujours, nous pouvons tester cet objet indépendamment pour nous assurer que tout fonctionne correctement :

Fichier

```
>>> from Joueur import Joueur
>>> from CarteClassique import CarteClassique
>>> c = CarteClassique(2, 2)
>>> j1 = Joueur("Lagaffe", "Gaston")
>>> print(j1)
Gaston Lagaffe
Palmarès: 0 défaite(s) et 0 victoire(s)
```

Fichier

```
>>> j1.ajouterCarte(c)
>>> print(j1)
Gaston Lagaffe
Palmarès: 0 défaite(s) et 0 victoire(s)
2 de Trefle
```

2. LE JEU

La classe **PartieBataille** va permettre de lancer la partie. Nous considérerons qu'il faut transmettre deux joueurs en paramètre de la méthode **démarrerPartie()**. Si un seul joueur est indiqué, c'est que l'adversaire est joué par l'ordinateur. Seule la partie contre l'ordinateur, complètement automatisée, sera présentée ici. Je vous laisse le loisir d'implémenter une partie à deux joueurs humains où une intervention manuelle pourra être incorporée (appuyer sur une touche pour tirer une carte par exemple).

Dans notre modélisation, la méthode **demarrerPartie()** est forcément lancée pour commencer une partie puisque cette dernière est la seule méthode publique. La méthode **main()** est à prendre au sens d'une main dans un jeu de cartes, c'est-à-dire des cartes jouées par l'ensemble des joueurs lors d'un tour. Le nom des autres méthodes indique clairement leur fonction. La présentation du code du fichier **PartieBataille.py** sera découpée en deux parties de manière à pouvoir en décrire plus précisément le fonctionnement :

Fichier

```
01: from Joueur import Joueur
02: from JeuClassique import JeuClassique
03:
04:
05: class PartieBataille:
06:
07:     def __init__(self, joueur):
08:         self.__joueur = joueur
09:         self.__ordinateur = Joueur('9000', 'HAL')
10:         self.__jeu = JeuClassique()
11:
12:     def demarrerPartie(self):
13:         self.__melanger()
14:         self.__distribuer()
15:
16:         poursuivre = True
17:         while poursuivre:
18:             poursuivre = self.__main()
19:
20:     def __melanger(self):
21:         self.__jeu.melanger()
22:
23:     def __distribuer(self):
24:         for i in range(len(self.__jeu.cartes)):
25:             carte = self.__jeu.tirer()
26:             if i % 2 == 0:
27:                 self.__joueur.ajouterCarte(carte)
28:             else:
29:                 self.__ordinateur.ajouterCarte(carte)
```

Création des joueurs et du jeu de cartes (on choisit les cartes « classiques »).

Dans une partie, on commence par mélanger le paquet de cartes.

Puis on distribue les cartes aux joueurs.

Enfin, les deux joueurs jouent tant qu'il leur reste une carte.

Le mélange du jeu de cartes se fait en appelant la méthode melanger() de la classe JeuCartes.

On distribue 26 cartes à chaque joueur (voir explications ci-dessous).

Il n'y a rien de bien compliqué dans ce code, seule la méthode **__distribuer()** peut nécessiter quelques explications supplémentaires. J'utilise ici une variable **i** dans une boucle. **i** prendra pour valeur **0**, puis **1**, et ainsi de suite jusqu'à **51** (la fonction **len()** nous renvoie le nombre de cartes du jeu et **range()** crée une liste d'entiers contenant autant d'éléments et commençant par **0**). Il n'y a plus alors qu'à regarder si **i** est pair ou impair (**i % 2** est le reste de **i** divisé par **2**, le modulo) et ensuite à attribuer la carte à l'un ou l'autre des joueurs.

Voyons la suite du code :

Fichier

```
31:     def __main(self, reste=[]):
32:         carte_joueur = self.__joueur.tirerCarte()
33:         carte_ordinateur = self.__ordinateur.tirerCarte()
34:
35:         if carte_joueur is None:
36:             self.finPartie(self.__joueur, self.__ordinateur)
37:             return False
38:         elif carte_ordinateur is None:
39:             self.finPartie(self.__ordinateur, self.__joueur)
40:             return False
41:
42:         print('Main :')
43:         print('    - {} {} : {}'.format(self.__joueur.prenom,
44:             self.__joueur.nom, str(carte_joueur)))
45:         print('    - {} {} : {}'.format(self.__ordinateur.prenom,
46:             self.__ordinateur.nom, str(carte_ordinateur)))
47:
48:         if carte_joueur.valeur == carte_ordinateur.valeur:
49:             reste.append(carte_joueur)
50:             reste.append(carte_ordinateur)
51:             return self.__bataille(reste)
52:         elif carte_joueur.valeur > carte_ordinateur.valeur:
53:             self.__joueur.ajouterCarte(carte_joueur)
54:             self.__joueur.ajouterCarte(carte_ordinateur)
55:             print('    {} {} gagne la main'.format(self.__joueur.prenom,
56:                 self.__joueur.nom))
57:         else:
58:             self.__ordinateur.ajouterCarte(carte_ordinateur)
59:             self.__ordinateur.ajouterCarte(carte_joueur)
60:             print('    {} {} gagne la main'.format(self.__ordinateur.prenom,
61:                 self.__ordinateur.nom))
62:
63:         return True
64:
65:     def __bataille(self, reste):
66:         print('*** BATAILLE ***')
67:         return self.__main(reste)
68:
69:     def finPartie(self, perdant, gagnant):
70:         perdant.defaites += 1
71:         gagnant.victoires += 1
72:         print('{} {} a gagné!!'.format(gagnant.prenom, gagnant.nom))
```

En cas de bataille(s), le paramètre reste contiendra une liste de cartes qui seront gagnées (ce mécanisme permet de réutiliser le code de main() en cas de bataille).

Chaque joueur tire une carte.

Si l'un des deux joueurs n'a plus de carte, il a perdu.

Affichage des cartes mises en jeu.

On teste qui a gagné et récupère les cartes. En cas d'égalité, on place les cartes mises en jeu dans la liste reste et on appelle la méthode bataille().

Si les deux joueurs ont pu jouer, on renvoie la valeur True.

En cas de bataille, on rappelle la méthode main() en lui indiquant qu'il y a déjà des cartes en jeu. Ce mécanisme permet de faire des batailles successives en augmentant simplement le nombre de cartes en jeu

Quand un joueur a gagné, on met à jour le palmarès des deux joueurs et on affiche un petit message.

JOUR 5

Comme vous avez pu le constater, il n'y avait plus qu'à lier tous les objets que nous avons créés. Bien sûr, cette implémentation n'est qu'un exemple parmi d'autres et il aurait été possible d'écrire un code réalisant les mêmes opérations d'une façon tout à fait différente.

Il ne reste plus qu'à écrire le code du programme principal qui permettra de lancer une partie. J'ai placé ces lignes dans un fichier **bataille.py** :

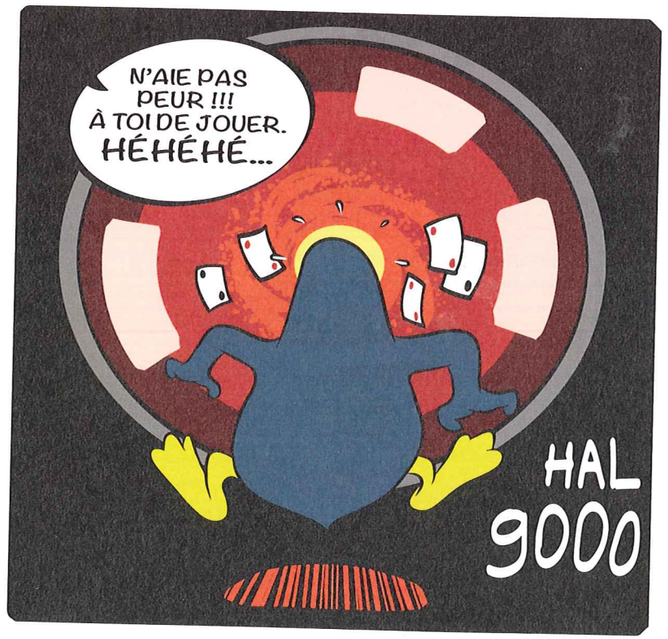
Fichier

```
01: from PartieBataille import PartieBataille
02: from Joueur import Joueur
03:
04:
05: if __name__ == '__main__':
06:     joueur_1 = Joueur('Lagaffe', 'Gaston')
07:     jeu = PartieBataille(joueur_1)
08:     jeu.demarrerPartie()
```

Comme tout est automatique, à moins d'insérer une pause à l'aide de l'instruction **input()**, vous verrez défiler les messages jusqu'à la fin de la partie :

Terminal

```
$ python3 bataille.py
...
Main :
    - Gaston Lagaffe : 5 de Coeur
    - HAL 9000 : 10 de Pique
    HAL 9000 gagne la main
Il n'y a plus de carte dans le jeu !
HAL 9000 a gagné!!
```



Pour la dernière journée, nous réfléchirons à des améliorations possibles...

Pour récapituler :

- Une fois que les objets sont créés, il faut les assembler pour créer le programme final. La segmentation en objets a permis de diviser le problème général en problèmes plus petits que l'on peut résoudre plus facilement.
- Il n'y a pas une unique façon de résoudre un problème, plusieurs approches sont possibles et les codes associés à ces solutions seront forcément différents... mais pas obligatoirement faux.
- Le diagramme de classes permet de savoir comment est découpé un problème et ce qu'il faut coder... mais si vous vous rendez compte en cours d'implémentation qu'il y a des corrections à apporter, c'est tout à fait possible (mais faites-le quand même le plus rapidement possible).

Résumé code :

Pour obtenir le programme complet, il manque le code de six classes : **Carte**, **CarteClassique**, **Paquet**, **JeuCartes**, et **JeuClassique**. Commençons par le fichier **Carte.py** :

```
01: class Carte:
02:     valeurs = None
03:     couleurs = None
04:
05:
06:     def __init__(self, val, coul):
07:         if self.__class__ is Carte:
08:             raise Exception("Création interdite!")
09:
10:         self.__class__.validation(val, coul)
11:         self.__valeur = val
12:         self.__couleur = coul
13:
14:     def getValeur(self):
15:         return self.__valeur
16:     def setValeur(self, val):
17:         self.__valeur = val
18:     valeur = property(getValeur, setValeur)
19:
20:     def getCouleur(self):
21:         return self.__couleur
22:     def setCouleur(self, coul):
23:         self.__couleur = coul
24:     couleur = property(getCouleur, setCouleur)
25:
26:
```

```
27:     @staticmethod
28:     def validation(val, coul):
29:         pass
30:
31:     def __str__(self):
32:         return str(Carte.valeurs[self.valeur]) + " de " + Carte.
couleurs[self.couleur]
33:
34:     def affiche_ascii(self):
35:         nom = str(Carte.valeurs[self.valeur]) + " de " + Carte.
couleurs[self.couleur]
36:         taille = len(nom) + 2
37:         print("/", "-" * taille, "\\", sep="")
38:         print("|", "-" * taille, "|", sep="")
39:         print("|", nom, "|")
40:         print("|", "-" * taille, "|", sep="")
41:         print("\\", "-" * taille, "/", sep="")
```

Voici ensuite **CarteClassique.py** :

```
01: from Carte import Carte
02:
03:
04: class CarteClassique(Carte):
05:
06:     def __init__(self, val, coul):
07:         Carte.valeurs = (None, None, 2, 3, 4, 5, 6, 7, 8, 9, 10,
"Valet", "Dame", "Roi", "As")
08:         Carte.couleurs = ("Coeur", "Carreau", "Trefle", "Pique")
09:         super().__init__(val, coul)
10:
11:     @staticmethod
12:     def validation(val, coul):
13:         if val < 2 or val > 14:
14:             print("Erreur : La valeur d'une carte est comprise
entre 2 et 14")
15:             exit(1)
16:         if coul < 0 or coul > 3:
17:             print("Erreur : Le code couleur d'une carte est
compris entre 0 et 3")
18:             exit(1)
```

Le fichier **Paquet.py** :

```
01: from JeuCartes import JeuCartes
02:
03:
04: class Paquet(JeuCartes):
05:
06:     def __init__(self):
07:         super().__init__()
08:
09:     def ajouter(self, carte):
10:         self.cartes.append(carte)
11:
12:     def __add__(self, carte):
13:         self.ajouter(carte)
```


Le fichier **JeuCartes.py** :

```
01: import random
02:
03:
04: class JeuCartes:
05:     def __init__(self, vide=False):
06:         if self.__class__ is JeuCartes:
07:             raise Exception("Creation interdite!")
08:         else:
09:             self.__cartes = []
10:             self.initialiser()
11:
12:     def __getCartes(self):
13:         return self.__cartes
14:     def __setCartes(self, carte):
15:         self.__cartes.append(carte)
16:     cartes = property(__getCartes, __setCartes)
17:
18:     def __str__(self):
19:         cartes_du_jeu = ""
20:         for carte in self.cartes:
21:             if cartes_du_jeu == "":
22:                 cartes_du_jeu = str(carte)
23:             else:
24:                 cartes_du_jeu += ", " + str(carte)
25:         return cartes_du_jeu
26:
27:     def melanger(self):
28:         random.shuffle(self.cartes)
29:
30:     def tirer(self):
31:         try:
32:             return self.cartes.pop(0)
33:         except IndexError:
34:             print("Il n'y a plus de carte dans le jeu !")
35:             return None
36:
37:     def initialiser(self):
38:         pass
```

Et enfin le fichier **JeuClassique.py** :

```
01: from JeuCartes import JeuCartes
02: from CarteClassique import CarteClassique
03:
04:
05: class JeuClassique(JeuCartes):
06:     def __init__(self):
07:         super().__init__()
08:
09:     def initialiser(self):
10:         for val in range(2, 15):
11:             for coul in range(4):
12:                 self.cartes.append(CarteClassique(val, coul))
```

JOUR 6

AMÉLIORATIONS

Notre jeu fonctionne, nous sommes parvenus à notre but ! Mais ne pourrions-nous pas faire mieux, n'y a-t-il rien qui puisse être amélioré ? Consacrons cette dernière journée à une réflexion sur les différentes pistes qui s'offrent à nous.

JOUR 6

On peut toujours « mieux faire » et souvent le problème principal dans un projet personnel est de savoir quand s'arrêter, quand le projet est arrivé à son terme et que les développements restants ne sont pas essentiels, mais représentent des améliorations pour les versions futures. Nous nous étions fixé un objectif et nous l'avons atteint, mais que cela ne nous empêche pas d'être lucide et de voir qu'il y aurait encore bien des choses que l'on pourrait améliorer. Je vais dresser ici une liste de pistes de développements possibles pour lesquels vous serez totalement libres : les connaissances acquises dans les jours précédents doivent vous permettre de les réaliser seul. Certaines améliorations pourront être codées rapidement et feront en quelque sorte partie de cette journée.

1. JEU À 2 JOUEURS

Le jeu de bataille est facilement automatisable puisque la seule action de chaque joueur est de tirer la carte se situant sur le dessus de son paquet. Dans notre implémentation du jeu contre l'ordinateur, le joueur humain n'intervient pas, c'est un peu comme si deux ordinateurs s'affrontaient... Il faut donc commencer par ajouter des interactions entre le programme et le joueur puis, ajouter un second joueur. Les modifications vont porter sur la classe `PartieBataille` puisque c'est cette dernière qui gère toute la partie. Vous aurez besoin d'utiliser la fonction `input()` pour poser des questions au joueur : demander ses nom et prénom lors de la création du joueur puis éventuellement demander d'appuyer sur une touche pour tirer une carte. Pour modifier la façon de tirer une carte avec un tirage qui peut être soit manuel, soit automatique, il faut modifier la méthode `tirer()` de la classe `JeuCartes`. Voici un exemple de ce à quoi pourrait ressembler la méthode :

Fichier

```
def tirer(self, manuel=False):
    try:
        if manuel:
            input('Appuyez sur <Return> pour tirer une carte')
        return self.cartes.pop(0)
    except IndexError:
        print("Il n'y a plus de carte dans le jeu !")
        return None
```

2. AUTRE JEU

Le jeu de Bataille est très limité en termes d'intelligence de jeu et de choix stratégiques pour le joueur. C'est d'ailleurs pour cela que nous l'avons utilisé de manière à pouvoir nous focaliser sur l'architecture orientée objet du code. Rien ne vous empêche maintenant de continuer le développement associé à la classe `CarteMagic` ou bien d'utiliser encore `CarteClassique`, mais avec un jeu aux règles un petit peu plus complexes.

Si vous partez sur un jeu complètement différent, utilisant d'autres cartes, n'oubliez pas de mettre en place une relation d'héritage et de créer une nouvelle classe de cartes et de jeu de cartes.

3 INTERFACE GRAPHIQUE

L'ajout d'une interface graphique constitue une grande avancée en terme d'utilisabilité pour le joueur... mais pour le développeur il s'agit d'une étape éprouvante : il faut bien réfléchir à la disposition des différents éléments, trouver ou

créer les graphismes et enfin, bien sûr, modifier le code de manière à afficher l'interface et à interagir avec les actions de l'utilisateur. Voici dans la suite quelques-uns des modules Python permettant d'implémenter une interface graphique.

3.1 Pygame

Pygame (**http://www.pygame.org/docs/**) est un module dédié à la création de jeux avec la possibilité de contrôler la souris, un joystick, le clavier (appui sur une touche), le son, etc. Il est particulièrement complet, mais a le défaut de ne pas posséder de version pour Python 3 (la version de Python que nous avons utilisé jusqu'à maintenant). Si vous vous sentez l'âme d'un(e) aventurier(ère), vous pouvez le tester, mais en utilisant la syntaxe de la version précédente de Python, la version 2.7 (et en installant également cette version sur votre machine).

Pour voir un petit peu à quoi peut ressembler un code utilisant Pygame, nous allons faire rebondir une balle à l'écran. Pour cela, il faut récupérer une image sur Internet. J'utilise souvent le site **http://www.iconfinder.com** qui propose un bon moteur de recherche et des images libres de droits. Dans le cas présent, j'ai récupéré une balle de tennis que j'ai enregistré sous le nom `balle.png` (Figure 1).

Figure 1

Nous avons notre image, il ne nous reste plus qu'à écrire le code... sauf que Pygame n'est pas un module installé par défaut !

3.1.1 Installation

Suivant le système d'exploitation que vous utilisez, l'installation ne se fera pas de la même manière.

3.1.1.1 Windows

Rendez-vous sur la page **http://www.pygame.org/download.shtml** et téléchargez le fichier `pygame-1.9.1.win32-py2.7.msi`. Il s'agit d'un fichier exécutable qui déclenchera le programme d'installation de Pygame. Vous devez indiquer où seront copiés les fichiers du module en sélectionnant l'entrée ***Will be installed on local hard drive*** (Figure 2).

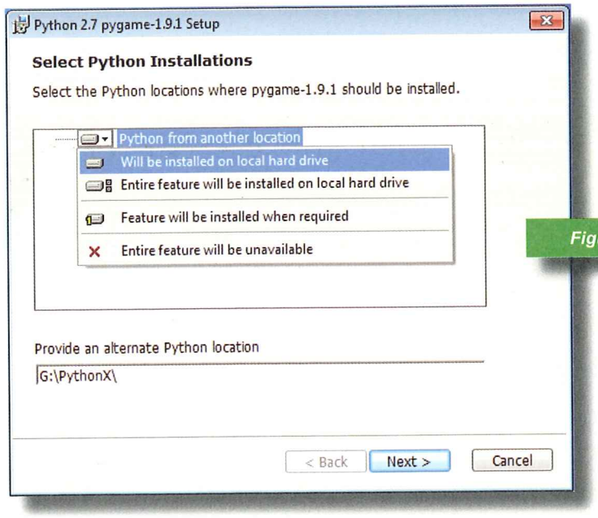


Figure 2

3.1.1.2 Linux Ubuntu

Rendez-vous dans le logiciel de gestion des logiciels et tapez `pygame` dans la barre de recherche et cliquez simplement sur le bouton ***Installer***.

3.1.1.3 Mac OS X

Rendez-vous sur la page **http://www.pygame.org/download.shtml** et téléchargez le fichier `pygame-1.9.1release-python.org-32bit-py2.7-macosx10.3.dmg`. Attention, l'application ne voudra pas se lancer, car provenant d'un développeur non identifié. Pour passer outre, appuyez sur la touche [Ctrl] et cliquez sur l'icône de l'application puis choisissez l'entrée ***Ouvrir*** du menu contextuel (puis confirmez votre choix en cliquant sur le bouton ***Ouvrir*** dans la fenêtre suivante).

Vous n'aurez ensuite plus qu'à suivre les indications à l'écran (Figure 3).

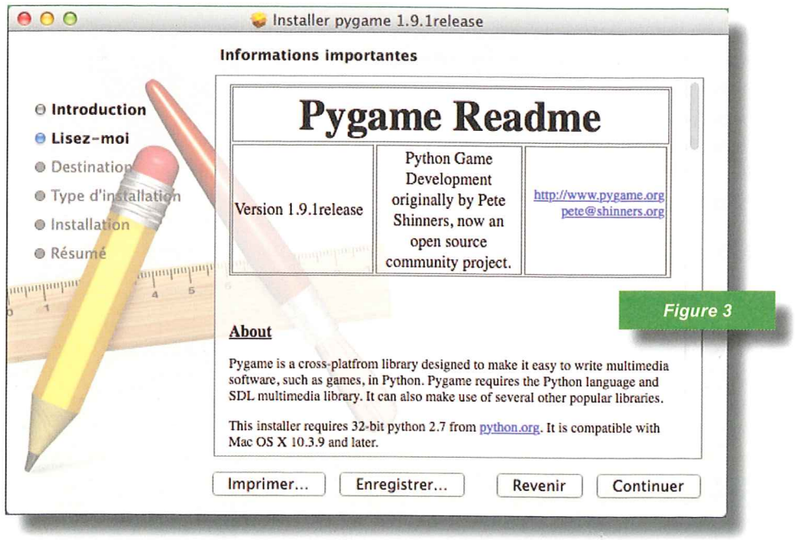


Figure 3

3.1.2 Exemple

Voici donc maintenant le petit exemple de code, adapté de la documentation de Pygame :

Fichier

```
01: import pygame ← Chargement du module Pygame.
02:
03: pygame.init() ← Initialisation de l'environnement graphique.
04:
05: size = width, height = 640, 480 ← size est un tuple contenant les deux
06: speed = [1, 1]                      valeurs width et height soit 640 et 480.
07: black = 0, 0, 0
08:
```


Fichier

```
09: screen = pygame.display.set_mode(size) ← Définition de la taille de la fenêtre.
10:
11: ball = pygame.image.load("balle.png") ← Chargement de l'image de la balle.
12: ballrect = ballrect.get_rect()
13:
14: while True: ← Boucle infinie.
15:     for event in pygame.event.get():
16:         if event.type == pygame.QUIT: ← Définition du comportement
17:             exit()                        lors de la fermeture de la
18:                                           fenêtre graphique.
19:     ballrect = ballrect.move(speed)
20:     if ballrect.left < 0 or ballrect.right > width:
21:         speed[0] = -speed[0]
22:     if ballrect.top < 0 or ballrect.bottom > height: ← Calcul du
23:         speed[1] = -speed[1]                             déplacement de
24:                                                          la balle.
25:     screen.fill(black)
26:     screen.blit(ball, ballrect) ← Affichage du contenu de la fenêtre.
27:     pygame.display.flip()
```

Au lancement, on obtient bien notre petite balle verte qui rebondit sur les coins de la fenêtre (Figure 4).

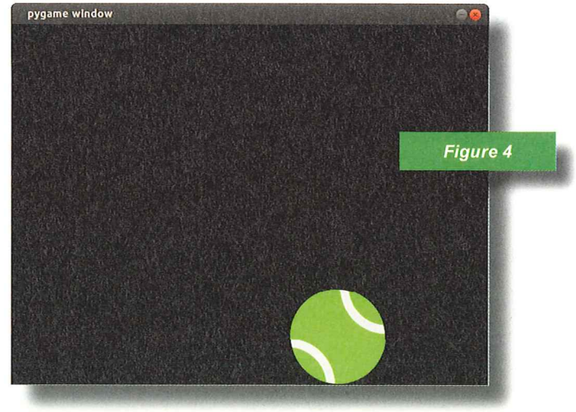


Figure 4

3.2 Cocos2d

Cocos2d (**http://cocos2d.org/**) est un autre module dédié au développement de jeux. Pour l'installer, vous devrez utiliser le programme d'installation des modules Python nommé `pip`. Reportez-vous à l'index « Les outils autour de Python » pour voir comment l'installer. Ceci fait, vous n'aurez qu'à taper la commande suivante pour télécharger et installer le module :

Terminal

```
pip3 install cocos2d
```

Voici un exemple d'utilisation de ce module. Cette fois-ci, nous allons faire « rebondir » la balle en lui appliquant une réduction puis une augmentation de taille. Pour faire plus joli, la balle tournera sur elle-même au sommet du rebond :

Fichier

```
01: import cocos
02: from cocos.actions import * ← Chargement du module.
03:
```

Fichier

```
04: class Balle(cocos.layer.Layer):
05:     def __init__(self):
06:         super().__init__()
07:         self.sprite = cocos.sprite.Sprite('balle.png')
08:         self.sprite.position = 320,240
09:         self.sprite.scale = 1
10:         self.add(self.sprite)
11:
12:         animation = ScaleBy(3, duration=2) + RotateBy(45, duration=2)
13:         self.sprite.do(Repeat(Reverse(animation) + animation))
```

On retrouve ici une structuration objet connue avec la création d'une classe Balle héritant de cocos.layer.Layers. C'est dans le constructeur que l'on charge le sprite et que l'on définit l'animation.

```
14:
15: if __name__ == '__main__':
16:     cocos.director.director.init()
17:     balle_layer = Balle()
18:     main_scene = cocos.scene.Scene(balle_layer)
19:     cocos.director.director.run(main_scene)
```

Initialisation de l'interface graphique.

Création d'une instance de Balle.

Affichage de la fenêtre graphique.

3.3 tkinter

tkinter (**https://wiki.python.org/moin/TkInter**) est un module dédié au développement d'applications utilisant Tk (un gestionnaire graphique multiplateforme). L'affichage est un peu « vieillot » avec certains éléments, mais l'avantage de ce module est d'être présent par défaut avec toutes les installations de Python. De plus, ce module sera certainement le plus simple à utiliser pour créer une première interface (qui sera plus proche de ce que l'on trouve en bureautique).

Je vous propose cette fois un petit programme affichant un texte et un bouton.

Fichier

```
01: import tkinter
02:
03: class App:
04:     def __init__(self, master_ui):
05:         frame = tkinter.Frame(master_ui)
06:         frame.pack()
07:
08:         self.label = tkinter.Label(frame, text="Ceci est un texte !")
09:         self.label.pack()
10:
11:         self.btn_OK = tkinter.Button(frame, text="OK", command=frame.quit)
12:         self.btn_OK.pack()
13:
```

Chargement du module tkinter.

La classe App définit le comportement de la fenêtre graphique.

Ajout d'une frame (cadre vide) à la fenêtre.

Ajout d'un label (texte).

Ajout d'un bouton.

Fichier

```
14: if __name__ == '__main__':
15:     root = tkinter.Tk()
16:     app = App(root)
17:     root.mainloop()
```

Lancement de l'application graphique.

Comme vous pouvez le constater, nous obtenons une interface pratique, mais pas forcément très élégante... (Figure 5)

Figure 5

Le domaine des interfaces graphiques représente un très vaste sujet que nous ne pourrons malheureusement pas traiter ici. Avec ces quelques pistes, vous devriez pouvoir vous faire une idée de ce qui est facilement réalisable et dans quelle direction commencer vos recherches.

4. DOCUMENTATION

Prenez le temps d'ajouter des commentaires à votre code, cela vous servira quand vous voudrez y faire des modifications après l'avoir mis de côté pendant quelque temps : vous gagnerez un temps précieux. Pour le diagramme de classes, vous pouvez par exemple placer celui-ci dans un répertoire **doc** et il doit être présent dans un format lisible rapidement (du PDF ou une image par exemple), mais également dans le format d'origine du logiciel avec lequel vous l'avez créé (pour pouvoir le modifier).

CONCLUSION

C'est fini, vous devriez être maintenant capable de programmer vos propres (petits) projets en utilisant une architecture orientée objet. Bien sûr, au début cela vous paraîtra compliqué (très compliqué même), mais si vous prenez vraiment le temps de réfléchir, de concevoir proprement votre diagramme de classes au lieu de vous jeter sur l'écriture du code, vous devriez vous rendre compte que c'est tout à fait à votre portée... Seules la pratique et la correction des erreurs vous permettront de comprendre réellement ce que vous faites. Alors programmez ! ▮

INDEX

INDEX

Les problèmes surviennent pendant la phase de développement et il faut alors être capable de les comprendre pour pouvoir les corriger. Nous vous proposons des index qui vous permettront de retrouver rapidement toutes les informations dont vous pourriez avoir besoin pour corriger vos programmes de manière efficace.

INDEX

LES OUTILS AUTOUR DE PYTHON

Dans l'environnement Python, certains programmes sont incontournables. Vous trouverez ici des explications sur l'utilisation de ces programmes.

1. PIP

pip est le logiciel chargé de gérer les modules Python de votre système. Une large partie des modules Python est hébergée sur PyPi, le Python Package Index. Lorsqu'il vous manque un module, le réflexe est donc de vous tourner vers cet outil pour pouvoir télécharger ledit module et l'installer.

Il ne s'agit pas ici d'une application graphique et vous allez donc devoir utiliser la ligne de commandes pour appeler ce programme et lui indiquer quoi faire grâce à des paramètres. L'installation va varier suivant le système d'exploitation que vous utilisez.

1.1 Installation

1.1.1 Windows

Vous avez de la chance ! `pip` est désormais inclus dans les installations de Python... vous n'avez rien à faire !

1.1.2 Linux Ubuntu

Il faudra à nouveau passer par le gestionnaire de programmes et rechercher cette fois-ci `python3-pip`. Le reste de la procédure est classique.

1.1.3 Mac OS X

Comme pour Windows vous n'aurez rien à faire, les dernières versions de Python incluent le programme `pip` !

1.2 Utilisation

Si vous n'en avez pas l'habitude, le plus difficile va être de trouver le programme permettant de saisir les commandes. Sans rentrer dans les détails, ce programme se nomme **terminal** ou **invite de commandes** suivant les systèmes.

1.2.1 Lancer le terminal

1.2.1.1 Windows

Dans le menu de démarrage, effectuez une recherche sur le terme `cmd` et cliquez sur le seul résultat renvoyé (Figure 1).

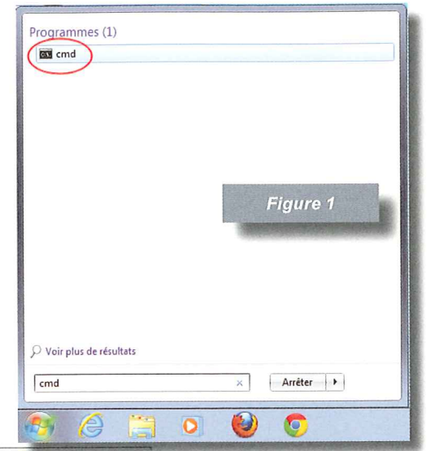


Figure 1

1.2.1.2 Linux Ubuntu

Cliquez dans le coin supérieur gauche du bureau sur l'icône du tableau de bord et effectuez une recherche sur le terme `terminal`. Cliquez ensuite sur l'icône de l'application **Terminal**.

1.2.1.3 Mac OS X

Vous trouverez l'icône du terminal dans le menu ***Application*** puis ***Utilitaires*** (Figure 2).

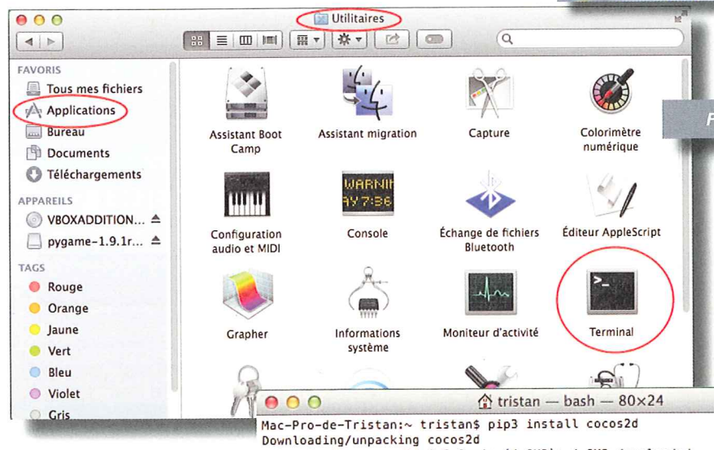


Figure 2

Le curseur gris vous indique que vous pouvez taper des caractères. C'est ici qu'il faudra indiquer les commandes de `pip` et que le programme vous répondra... (Figure 3).

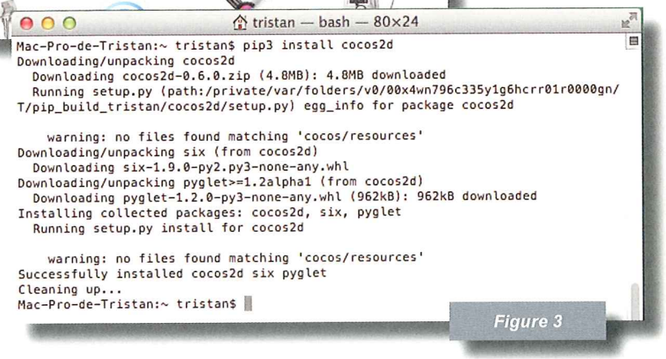


Figure 3

1.2.2 Les commandes de pip

Le nom de la commande `pip` pour la version de Python que nous utilisons, Python 3, est `pip3`. Cette commande doit être suivie d'un nom d'action et d'un paramètre associé à cette action. Les actions que vous utiliserez le plus sont :

⇨ `install` : installe un nouveau module et l'ensemble de ses dépendances (les autres programmes dont il a besoin pour fonctionner). Par exemple, pour installer le module `cocos2d` :

Terminal
```
pip3 install cocos2d
```

L'option `-r` permet de passer en paramètre le nom d'un fichier contenant une liste de noms de modules. Pip3 installera alors l'ensemble de ces modules. Ce système est très pratique pour indiquer les dépendances d'un programme et faciliter son installation. Le format du fichier transmis sera du même type que le résultat obtenu en utilisant la commande `list` (voir le point suivant). Voici un exemple d'appel sur un fichier `requirements.txt` :

Terminal
```
pip3 install -r requirements.txt
```

L'option `-u` permet de mettre à jour un module déjà installé.

⇨ `list` : affiche la liste des modules installés sur le système. Exemple :

Terminal
```
pip3 list
argparse (1.2.2)
cocos2d (0.6.0)
colorama (0.2.5)
Cython (0.21.1)
ipython (1.2.1)
lxml (3.3.3)
numba (0.15.1)
numpy (1.8.2)
...
```

⇨ `uninstall` : désinstalle un module. Par exemple pour retirer le module `cocos2d` du système :

Terminal
```
pip3 uninstall cocos2d
```

⇨ `search` : cette commande va vous permettre d'effectuer une recherche dans la description des modules hébergés sur PyPi. Supposons que vous recherchiez un module permettant de réaliser une interface graphique. Vous allez effectuer une recherche sur le terme GUI (*Graphical User Interface* en anglais).

Terminal
```
pip3 search GUI
PicoGUI                   - PicoGUI python client library
...
tkform                    - a tkinter form-based GUI that wraps python scripts
...
WiPy                      - GUI Wired client using wxPython
...
pyglet-gui                - An extension of pyglet for GUIs
...
```

2. LES ENVIRONNEMENTS VIRTUELS

Lorsque l'on programme en Python, il est fréquent d'utiliser de nombreux modules... mais ces modules peuvent changer de version et les nouvelles versions , si elles comportent des changements majeurs, risquent de faire planter vos programmes. De plus, certains modules sont utilisés directement par votre système d'exploitation et lorsque vous voudrez voir quels sont les modules nécessaires pour faire fonctionner votre code, vous obtiendrez une liste de l'ensemble des modules du système. Pour éviter cela, on utilise les environnements virtuels...

Un environnement virtuel est un espace dans lequel on développe un programme et qui est isolé du reste du système. Par exemple, si sur votre système le module `cocos2d` est installé, mais que vous ne l'avez pas installé dans l'environnement virtuel alors vous n'y aurez pas accès. Du coup, lorsque vous listerez les modules de votre environnement virtuel vous obtiendrez la liste des modules requis uniquement pour votre programme.

Pour pouvoir utiliser les environnements virtuels, vous devez avoir installé `virtualenv`... et comme vous avez tous installé `pip` sur vos machines, il n'y a plus de distinction à faire entre Windows, Linux et Mac OS X... ou presque !

2.1 Installation

Dans un terminal, tapez :

Terminal
```
pip install virtualenv
pip install virtualenvwrapper
```

Attention ! Pour Windows, le nom du dernier paquet est légèrement différent. Il s'agit de `virtualenvwrapper-win` :

Terminal
```
pip install virtualenvwrapper-win
```

2.2 Utilisation

Vous disposez maintenant de plusieurs commandes (à utiliser toujours dans le terminal) :

⇨ `mkvirtualenv` : permet de créer un nouvel environnement virtuel. L'option `--python` permet éventuellement de spécifier la version de Python à utiliser en indiquant le chemin complet vers l'interpréteur (cette version devra forcément avoir été installée sur votre système). Voici un exemple créant un environnement virtuel `jeu_cartes` :

Terminal
```
mkvirtualenv --python=/usr/bin/python3.4 jeu_cartes
```

⇨ `rmvirtualenv` : supprime un environnement existant. Par exemple :

Terminal
```
rmvirtualenv jeu_cartes
```

⇨ `workon` : sélectionne un environnement virtuel pour y travailler. Le prompt de votre ligne de commandes sera modifié pour vous indiquer que vous êtes rentré dans un environnement virtuel et que vous n'aurez donc accès plus qu'à ses modules :

Terminal
```
workon jeu_cartes
(jeu_cartes)
```

⇨ `deactivate` : permet de sortir d'un environnement virtuel.

Vous pouvez maintenant utiliser la commande `freeze` de `pip` pour lister les modules installés dans un environnement virtuel et en faire un fichier de prérequis :

Terminal
```
(jeu_cartes) pip3 freeze > requirements.txt
```

INDEX DES INSTRUCTIONS

Vous retrouverez ici des explications et des exemples sur les différentes instructions utilisées pour réaliser notre projet.

__CLASS__

Attribut contenant le type d'une classe (et donc son nom).

Fichier
```
>>> a = 2
>>> print(a.__class__)
<class 'int'>
```

__DICT__

Dictionnaire-attribut qui contient les références à tous les attributs de l'objet.

__NAME__

Variable contenant le nom d'un module si celui-ci est importé ou la chaîne de caractères `'__main__'` s'il est exécuté directement.

__STR__

Nom de la méthode qui s'exécute lorsque l'on tente de convertir un objet en chaîne de caractères (appel à str()). Cette méthode ne prend pas de paramètre et doit forcément retourner une chaîne de caractères :

Fichier
```
class Test:
    ...
    def __str__(self):
        return 'Chaîne de caractères'
```


@STATICMETHOD

Décorateur utilisé pour déclarer une méthode de classe (ou méthode statique) :

Fichier
```
class Test:
    ...
    @staticmethod
    def methodeStatique():
        print('Coucou !')
```

Les méthodes statiques peuvent être appelées même si aucune instance de la classe n'a été créée :

Fichier
```
>>> from Test import Test
>>> Test.methodeStatique()
Coucou !
```

DEF

Pour créer une fonction (ou une méthode), il faut employer l'instruction `def` suivie du nom de la fonction (ou de la méthode) et ses paramètres.

Fichier
```
def ma_fonction():
    print("Ceci est une fonction")
```

DEL

Supprime un objet (en Python tout est objet, donc vous pouvez supprimer n'importe quel élément) :

Fichier
```
>>> a = 1
>>> print(a)
1
>>> del a
>>> print(a)
NameError: name 'a' is not defined
```

DIR

Liste les méthodes et les attributs d'un objet :

Fichier
```
>>> from Carte import Carte
>>> c = Carte(2, 1)
>>> dir(c)
['_Carte__couleur', '_Carte__valeur', ..., 'affiche_ascii', ..., 'validation']
```

ELIF

Instruction permettant d'effectuer des tests en les chaînant.

Fichier
```
if val == 1:
    print("Choix 1")
elif val == 2:
    print("Choix 2")
elif val == 3:
    print("Choix 3")
else:
    print("Choix par défaut")
```

ELSE

Utilisé dans les structures conditionnelles (`if`), permet de proposer un traitement correspondant à la non-vérification de la condition.

Fichier
```
a = 5

if a == 2:
    print("Valeur 2 détectée")
else:
    print("Valeur différente de 2")
```

EXCEPT

Dans le mécanisme de gestion des exceptions, `except` permet de faire un branchement conditionnel sur une erreur de manière à ne pas afficher le message défini par défaut et sortir du programme, mais à exécuter un bloc de code personnalisé. S'utilise avec `try` qui « analyse » un bloc de code de manière à intercepter les erreurs :

Fichier
```
try:
    n = int(input("Saisir un entier :"))
except ValueError:
    print("Vous n'avez pas saisi un entier !")
```

EXIT

Permet d'interrompre l'exécution d'un programme (de manière définitive). Cette fonction attend un entier compris entre 0 et 255 en paramètre : il s'agit d'un code d'erreur défini de manière arbitraire. Seule la valeur 0 est utilisée pour indiquer que le programme s'est terminé correctement (sans erreur).

Fichier
```
>>> exit(1)
```

Sous Mac OS X et Linux, cette valeur peut être lue depuis le shell en utilisant la variable `$?` :

Terminal
```
$ echo $ ?
1
```

Sous Windows, cette valeur peut être lue depuis l'invite de commande MS-DOS en utilisant la variable `%errorlevel%` :

Terminal
```
C:\WINDOWS> echo %errorlevel%
1
```

FOR

Instruction permettant de parcourir l'ensemble des éléments d'une liste et de créer ainsi une boucle :

Fichier
```
liste = [1, 2, 3, 4, 5]

for elt in liste :
    print("Valeur :", elt)
```

L'exécution de ces lignes provoque l'affichage suivant :

Terminal
```
Valeur: 1
Valeur: 2
Valeur: 3
Valeur: 4
Valeur: 5
```

FORMAT

Permet de formater une chaîne de caractères en utilisant une syntaxe de formatage :

- `{}` pour insérer une valeur dont le type sera détecté automatiquement ;
- `{:d}` pour un entier ;
- `{:.nf}` pour afficher seulement n chiffres de la partie décimale d'un réel ;
- `{:s}` pour une chaîne de caractères ;
- etc.

Fichier
```
>>> print('{:.2f}'.format(5.3243245))
5.32
```

FROM

Associé à import permet de charger un module en spécifiant ou non les éléments à charger.

Fichier
```
from Carte import Carte
from random import *

c = Carte(1, 2)
print(randint(1, 10))
```

HELP

Affiche la page d'aide associée à une instruction :

Fichier
```
>>> help(print)
```

Attention : les noms de commandes doivent être passés sous forme de chaîne de caractères. Une commande est une instruction qui ne comporte pas de parenthèses pour spécifier des paramètres (`return` par exemple) :

Fichier
```
>>> help('return')
```

IF

Structure de test permettant de créer un embranchement.

Fichier
```
a = 5

if a == 2:
    print("Valeur 2 détectée")
```

IMPORT

Instruction permettant de charger un module. Pour utiliser les éléments proposés par le module ainsi chargé il faudra les faire précéder du nom du module (à moins d'utiliser `from`).

Fichier
```
import Carte

c = Carte.Carte(2, 2)
```

LEN

Fonction permettant de connaître la taille d'une liste, d'un tuple, d'un dictionnaire ou d'un ensemble :

Fichier
```
>>> liste = [14, 5, 1]
>>> print(len(liste))
3
```

LIST

Fonction de conversion sous forme de liste.

Terminal
```
>>> list("abcdef")
["a", "b", "c", "d", "e", "f"]
```

NONE

Valeur indiquant... l'absence de valeur. Permet de définir une variable sans pour autant lui attribuer une valeur :

Fichier
```
>>> a = None
```

PASS

Instruction permettant de créer un bloc qui n'effectue aucun traitement :

Fichier
```
def vide():
    pass
```

PRINT

Fonction d'affichage de texte. Les paramètres non nommés séparés par des virgules seront concaténés (attachés les uns aux autres et séparés par un caractère qui peut être défini par le paramètre nommé `sep`). Notez que les variables seront automatiquement converties en chaîne de caractères.

Le paramètre `end` permet de redéfinir les caractères de fin de ligne (le retour à la ligne).

Fichier
```
>>> a = 5
>>> print("Valeur", a, sep = " : ", end = "!\n")
Valeur : 5
```

PROPERTY

Instruction utilisée dans la mise en place de l'encapsulation et permettant de créer un « lien » vers des méthodes en fonction du type d'accès à une variable (en lecture ou en écriture). Exemple :

Fichier
```
class Test:
    def __init__(self):
        self.__attribut = 10

    def getAttribut(self):
        return self.__attribut
    def setAttribut(self, valeur):
        self.__attribut = valeur
    nouveau_lien = property(getAttribut, setAttribut)
```

À l'utilisation, `nouveau_lien` nous donnera accès à `__attribut` :

Fichier
```
>>> from Test import Test
>>> obj = Test()
>>> obj.nouveau_lien = 12
```

RAISE

Permet de lancer (on dit « lever ») une exception. On envoie un message pour indiquer qu'il y a eu une erreur, libre ensuite au développeur qui utilisera notre code de traiter cette erreur en la récupérant dans un bloc `try / except` :

Fichier
```
raise Exception('Erreur !')
```

RANDOM.SHUFFLE

Méthode `shuffle()` du module random permettant de mélanger les éléments d'une liste :

Fichier
```
>>> import random
>>> l = [1, 2, 3, 4, 5, 6]
>>> random.shuffle(l)
>>> print(l)
[5, 6, 3, 2, 1, 4]
```

RANGE

Fonction générant une « liste » d'entiers à partir d'un, deux ou trois paramètres :

⇨ `range(fin)` : liste de `0` à `fin - 1`

⇨ `range(debut, fin)` : liste de `debut` à `fin - 1`

⇨ `range(debut, fin, pas)` : liste de `debut` à `fin - 1`

La valeur par défaut du pas est 1. Si vous souhaitez obtenir une liste, il faudra utiliser la fonction de conversion list (sinon vous obtiendrez un objet sur lequel vous pourrez itérer, mais que vous ne pourrez pas manipuler comme une liste) :

Fichier
```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On trouve souvent cette fonction utilisée dans des boucles :

Fichier
```
for i in range(5):
    print(i)
```

RETURN

Instruction permettant de déterminer la valeur de retour d'une fonction ou d'une méthode.

Fichier
```
def carre(x):
    """
        Retourne le carré de la valeur passée en paramètre
    """
    return x ** 2
```

SELF

Élément désignant l'instance courante (l'objet créé) à l'intérieur d'une classe. Correspond au `this` d'autres langages.

Fichier
```
def methode(self):
    x = 0
    self.x = 0
```

x est une variable qui sera détruite à la fin du bloc de `methode()` alors que `self.x` est « attaché » à l'objet courant.

SUPER()

Fonction permettant de faire référence à la classe mère dans le cadre d'une relation d'héritage. L'utilisation de `super()` permet de s'affranchir du passage de `self` en paramètre :

Fichier
```
super().__init__(variable)
```

Cette ligne est équivalente à :

Fichier
```
Nom_Classe_Mere.__init__(self, variable)
```

STR

Fonction permettant de convertir un objet en chaîne de caractères :

Fichier
```
>>> str(12)
'12'
```

TRY

Voir `except`.

WHILE

Instruction de boucle signifiant « tant que » : tant que la condition qui suit le `while` est valide, le bloc de code suivant sera exécuté. Faites attention que la condition testée soit bien modifiée lors des itérations successives jusqu'à être invalidée à un moment donné. Sinon votre boucle ne se finira jamais...

Fichier
```
i = 0
while i < 10:
    print(i)
    i += 1
```


INDEX DES NOTIONS

Vous retrouverez ici une définition des différentes notions abordées tout au long de la construction de notre projet.

ATTRIBUT

Un attribut est une variable attachée à une classe. Les attributs d'une classe sont visibles et accessibles par l'ensemble des méthodes de la classe. Il est possible de leur donner une visibilité publique ou privée.

Fichier
```
class Test:
    def __init__(self):
        self.attribut = 12

    def methode(self):
        print(self.attribut)
```

ATTRIBUT DE CLASSE

Un attribut de classe est un attribut qui est partagé par l'ensemble des instances d'une classe : si une instance modifie sa valeur alors la valeur visible par l'ensemble des autres instances est modifiée.

D'un point de vue syntaxique, un attribut de classe est déclaré sans `self` et pour l'utiliser il faut préfixer son nom par le nom de la classe :

Fichier
```
class Test:
    attribut_de_classe = 1

    def __init__(self) :
        print(Test.attribut_de_classe)
```

ATTRIBUT STATIQUE

Voir attribut de classe.

BLOC

Un bloc de code est un ensemble d'instructions se trouvant au même niveau d'indentation et qui constitue en quelque sorte une seule instruction : dans le cas d'un branchement conditionnel par exemple, toutes les instructions du bloc seront exécutées.

Fichier

```
if variable == 10:
    print('Début bloc')
    print('Bloc exécuté')
    ...
    print('Fin du bloc')
```

BOUCLE

Une boucle permet de répéter un bloc d'instructions un certain nombre de fois. Pour effectuer cette tâche, on peut utiliser les instructions `for` ou `while`.

CLASSE ABSTRAITE

Une classe abstraite est une classe qui possède au moins une méthode qui n'est pas définie (elle sert de « cadre » à la création d'autres classes par héritage). Il n'est pas possible de créer une instance d'une classe abstraite.

CLASSE DÉRIVÉE

Voir classe fille.

CLASSE FILLE

Dans une relation d'héritage, classe construite à partir d'une autre classe (sa classe mère). Il existe un lien de spécialisation entre les deux classes : on peut dire que la « classe fille » est une « classe mère » particulière. Par exemple, un triangle est un polygone particulier.

CLASSE MÈRE

Dans une relation d'héritage, classe servant de « modèle » pour en créer une autre.

CONSTRUCTEUR

Méthode particulière qui réserve l'espace mémoire permettant la création d'une instance de la classe.

DIAGRAMME DE CLASSES

Un diagramme de classes est un schéma indiquant la constitution des classes (attributs et méthodes) ainsi que les relations entre classes. Il existe de nombreux programmes permettant de réaliser de tels schémas... ou simplement un crayon et une feuille de papier.

ENCAPSULATION

Méthode consistant à protéger l'accès à tous les attributs d'une classe en les définissant en visibilité privée puis en « ouvrant » les accès en lecture et en écriture au cas par cas en utilisant des méthodes publiques.

FONCTION

Bloc d'instructions exécutable d'après son nom et paramétrable. Le principe est le même que pour les fonctions mathématiques.

Fichier

```
>>> def f(x):
...     return 2 * x + 3
...
>>> f(2)
7
```

HÉRITAGE

Fait d'utiliser les attributs et méthodes d'une classe pour en créer une autre plus spécialisée. La classe « modèle » est appelée classe mère et la classe résultante est appelée classe dérivée ou classe fille.

INSTANCE

Une instance d'une classe est un élément qui a été créé en s'appuyant sur le schéma donné par une classe. À partir d'une classe, on peut créer plusieurs instances : des éléments semblables, ayant les mêmes propriétés, mais différant par les valeurs de leurs attributs.

INSTRUCTION

Une instruction est une commande informatique qui peut être interprétée par la machine en vue de fournir un résultat. L'instruction est un terme générique recouvrant l'ensemble des commandes.

MÉTHODE

Une méthode est une fonction « attachée » à une classe et qui a accès aux attributs de cette classe.

MÉTHODE DE CLASSE

Une méthode de classe est une méthode qui n'a accès qu'aux attributs de classe et qui peut être appelée même si aucune instance de la classe n'a été créée.

MÉTHODE STATIQUE

Voir méthode de classe.

MODULE

Un module est un fichier Python contenant une suite d'instructions. On le charge en mémoire à l'aide des instructions `import` ou `from ... import`. À ce moment-là, les instructions du module sont exécutées et les fonctions ou les objets qu'il propose deviennent accessibles.

Fichier

```
>>> c = Carte(1, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Carte' is not defined
>>> from Carte import Carte
>>> c = Carte(1, 3)
```

SURCHARGE DES OPÉRATEURS

Il est possible de modifier le comportement d'un opérateur appliqué sur un objet. Par exemple, la méthode `__add__()` permet de définir le comportement d'un objet lorsque l'on applique le signe `+` à une de ces instances.

On peut surcharger n'importe quel opérateur.

TEST CONDITIONNEL

Un test conditionnel permet de créer un embranchement dans l'exécution d'instructions : si la valeur d'une condition booléenne est vraie, on va exécuter un bloc de code, sinon, si elle est fausse, on va exécuter un autre bloc de code.

Fichier

```
>>> a = 10
>>> if a == 10:
...     print("Vrai")
... else:
...     print("Faux")
...
Vrai
```

VARIABLE

Une variable est un espace mémoire contenant une valeur. On y accède à l'aide d'une étiquette ou identifiant. En Python, les variables ne sont pas typées et on peut donc y stocker n'importe quelle sorte de valeur : entier, chaîne de caractères, etc.

Fichier

```
>>> ma_variable = 'coucou'
```

VARIABLE LOCALE

Une variable locale est une variable qui n'existe que dans un bloc donné :

Fichier

```
def coucou():
    msg = "coucou"
    print (msg)

coucou()

print("A l'exterieur de la fct:")
print(msg)
```

À l'exécution, la variable `msg` ne sera visible que dans la fonction `coucou()` :

Terminal

```
coucou
A l'exterieur de la fct:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print(msg)
NameError: name 'msg' is not defined
```

VISIBILITÉ PRIVÉE

Des attributs ou méthodes ayant une visibilité privée ne sont accessibles en lecture et en écriture que depuis l'extérieur de la classe de définition.

VISIBILITÉ PUBLIQUE

Des attributs ou méthodes ayant une visibilité publique sont accessibles en lecture et en écriture depuis l'extérieur de la classe de définition.

D'un point de vue syntaxique, la visibilité privée est donnée par une convention de nommage : le nom de l'attribut ou de la méthode doit commencer par deux caractères underscores.

BONUS

ANNEXE - LES ERREURS COURANTES

Vous retrouverez ici une liste des erreurs les plus courantes. Servez-vous de cette annexe pour résoudre plus rapidement vos erreurs.

ImportError: No module named '<nom>'

Vous essayez de charger un module qui n'existe pas :

1. Vous avez fait une faute de frappe dans le nom du module ;
2. Le fichier du module n'est pas accessible : pour être chargé, le module doit se trouver dans le même répertoire que le fichier de votre programme.

Une fonction d'un module standard est inaccessible

Vérifiez que l'un de vos fichiers se trouvant dans le même répertoire que votre programme ne porte pas le nom d'un module standard. Par exemple, si vous avez nommé un fichier `math.py`, c'est ce module qui sera chargé par l'instruction `from math import <nom_fct>` et non le module standard !

J'ai modifié une classe, mais lorsque je la teste la modification n'apparaît pas...

Plusieurs cas sont possibles :

1. Êtes-vous certain(e) d'utiliser le fichier que vous avez modifié ?
2. Si vous travaillez dans un interpréteur interactif (PyDev console), avez-vous pensé à refermer l'interpréteur puis à le rouvrir et à importer votre classe ?

NameError: name '<nom>' is not defined

Vous essayez d'accéder à une variable qui n'a pas encore été définie (attention à la casse : `nom` est différent de `Nom`, `Nom`, `NOM` ou encore `noM`).

Ce message apparaît également lorsque l'on essaye de créer une instance d'une classe alors que la définition de la classe n'a pas été importée. Par exemple, si vous essayez de créer une carte sans avoir chargé la classe Carte :

Fichier

```
>>> c = Carte(12, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Carte' is not defined
>>> from Carte import Carte
>>> c = Carte(12, 2)
```


Si vous importez un module à l'aide de l'instruction `import` seule, vous devez préfixer le nom de chaque élément du module par le nom du module lui-même :

Fichier

```
>>> import Carte
>>> c = Carte(2, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'module' object is not callable
>>> c = Carte.Carte(2, 1)
```

AttributeError: '<nom_objet>' object has no attribute '<nom_attribut>'

Vous essayez d'accéder à un attribut privé ou à un attribut qui n'existe pas. Vérifiez que vous n'avez pas fait de faute de frappe en tapant le nom de l'attribut.

<bound method <nom_classe>.<nom_méthode> of <<nom_classe> object at 0x...>>

Vous avez appelé une méthode en oubliant les parenthèses... Vous obtenez donc l'adresse mémoire de la méthode au lieu de l'exécuter. Ajoutez des parenthèses (et éventuellement des paramètres) pour corriger l'erreur.

<built-in function <nom_fonction>>

Même principe que précédemment : vous appelez une fonction en ayant omis les parenthèses. Par exemple :

Fichier

```
>>> print
<built-in function print>
```

TypeError: unsupported operand type(s) for <opérateur>: '<classe_1>' and '<classe_2>'

Vous n'avez pas surchargé l'opérateur `<opérateur>` que vous essayez d'appliquer entre une instance de la classe `<classe_1>` et une instance de la classe `<classe_2>` :

- vous avez oublié de définir la méthode associée à `<opérateur>` ;
- ou vous avez mal écrit le nom de la méthode (avez-vous mis deux caractères underscore au début et à la fin du nom ?) ;
- ou vous n'avez pas défini la méthode dans la bonne classe. Pour `<classe_1> <opérateur> <classe_2>` la méthode doit être définie dans `<classe_1>`.

IndexError: tuple index out of range

Vous essayez d'accéder à un élément d'un tuple qui n'est pas défini. Ceci peut arriver sur un simple tuple :

Fichier

```
>>> t = (1, 2, 3)
>>> print(t[3])
```

`t[3]` n'existe pas : il n'y a que trois éléments indexés de `0` à `2`.

Vous pouvez également rencontrer cette erreur en appelant certaines fonctions pour lesquelles vous n'aurez pas fourni suffisamment de paramètres :

Fichier
```
>>> print("{} + {} = {}".format(12, 5))
```

Le formatage attend trois données à insérer dans la chaîne de caractères.

IndexError: list index out of range

Vous essayez d'accéder à un élément d'une liste qui n'est pas définie :

Fichier
```
>>> t = [1, 2, 3]
>>> print(t[3])
```

`t[3]` n'existe pas : il n'y a que trois éléments indexés de `0` à `2`.

TypeError: Can't convert '<type>' object to str implicitly

Vous essayez d'utiliser un objet quelconque dans un contexte de chaîne de caractères alors que sa méthode `__str__()` n'est pas définie. Cette erreur apparaît aussi sur des types de base :

Fichier
```
>>> a = 'Valeur : ' + 12
```

IndentationError: expected an indented block

Vous avez utilisé une instruction ouvrant un bloc (`if`, `for`, etc), mais vous n'avez pas ajouté d'indentation sur la ligne suivante. Par exemple, ces deux lignes provoquent une erreur :

Fichier
```
if a == 1:
print('ok')
```

Il aurait fallu écrire :

Fichier
```
if a == 1:
    print('ok')
```

IndentationError: unexpected indent

Vous avez ajouté une indentation à un endroit où il n'y en avait pas besoin. Par exemple, si l'on ajoute une indentation sur une ligne d'affectation n'appartenant pas à un bloc, nous obtiendrons ce message d'erreur.

Fichier
```
    a = 1
```

Pour corriger l'erreur, il suffit de retirer l'indentation.

SyntaxError: invalid syntax

C'est l'erreur la plus générale... Les causes peuvent être multiples !

1 Peut-être avez-vous utilisé une instruction ouvrant un bloc sans la terminer par deux-points comme dans :

Fichier
```
if a == 1
    print('ok')
```

Il faut alors ajouter les deux-points à la fin de la première ligne :

Fichier
```
if a == 1:
    print('ok')
```

2 Avez-vous utilisé le bon opérateur pour effectuer un test ? Il ne faut pas confondre `=` qui réalise une affectation avec `==` qui teste l'égalité.

3 Vous essayez d'utiliser un mot-clé réservé de Python en tant qu'identifiant de variable. Par exemple, en essayant d'appeler une variable `pass` :

Fichier
```
>>> pass = 1
```

4 Si vous avez déjà développé dans d'autres langages auparavant, peut-être avez-vous essayé d'utiliser l'opérateur d'incrémentation `++` ou l'opérateur de décrémentation `--` qui n'existent pas en Python...

TypeError: 'list' object cannot be interpreted as an integer

Vous utilisez un objet de type liste comme s'il s'agissait d'un entier. Ceci peut arriver si vous oubliez de considérer la taille d'une liste dans une boucle `for` :

Fichier
```
>>> tab = [1, 2, 3]
>>> for i in range(tab):
...     print(tab[i])
```

Ici, il fallait utiliser `range(len(tab))`.

TypeError: 'str' object does not support item assignment

Vous essayez de modifier la valeur d'une chaîne de caractères (qui est un élément non modifiable) ! Voici un exemple provoquant cette erreur :

Fichier
```
>>> a = 'hello'
>>> a[1] = 'a'
```

SyntaxError: EOL while scanning string literal

Vous avez oublié de fermer une chaîne de caractères avec un guillemet ou une apostrophe :

Fichier
```
>>> print('Hello)
```

Il manque une apostrophe après le `o`.

KeyError: '<nom>'

Vous essayez d'accéder à une clé d'un dictionnaire qui n'a pas été définie :

Fichier

```
>>> d = {}
>>> d = {'a' : 1, 'b' : 2}
>>> d['c']
```

La clé `c` n'existe pas !

TypeError: 'range' object does not support item assignment

Vous essayez d'utiliser la fonction `range()` comme si elle générait une liste d'entiers alors qu'il s'agit d'un itérateur. Pour résoudre le problème, vous devez convertir le résultat de `range()` en liste avec la fonction `liste()`.

Fichier

```
>>> tab = list(range(10))
>>> print(tab[1])
```

TypeError: <nom_methode>() takes 0 positional arguments but 1 was given

Vous avez oublié d'indiquer le paramètre `self` dans une méthode ou alors vous vouliez définir une méthode de classe et vous avez oublié le décorateur `@staticmethod`. Voici un exemple ou la méthode `affiche()` provoque l'affichage d'un message d'erreur :

Fichier

```
class Test:
    def __init__(self, x):
        self.x = x
    def affiche():
        print('coucou')
...
t = Test(1)
t.methode()
```

UnboundLocalError: local variable '<nom_variable>' referenced before assignment

Vous avez utilisé un même nom de variable en tant que variable globale et en tant que variable locale. Il vaut mieux éviter d'utiliser les variables globales qui posent de nombreux problèmes dans les programmes !

Voici un exemple où une variable a est utilisée une première fois en tant que variable globale (dans le `print`), puis en tant que variable locale (dans l'assignation). Du coup, l'interpréteur pense que la première occurrence de la variable est locale et il ne la trouve pas :

Fichier

```
>>> a = 5
>>> def fct():
...     print(a)
...     a = 50
```