LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE
N°71
France METRO : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ cad — MAR : 130 MAD

C'EST DÉCIDÉ, AUJOURD'HUI JE M'Y METS !

JE PROGRAMME

zéro PRÉREQUIS

EN 7 JOURS SEULEMENT ET SANS (TROP) D'EFFORTS !

INTRODUCTION : Préparer ses armes

Choisir les bons outils pour s'initier à la programmation

VOTRE SEMAINE

JOUR 1 : Afficher des caractères
JOUR 2 : Saisir des données
JOUR 3 : Mettre en place les tests et les conditions
JOUR 4 : Gérer les fichiers et modules
JOUR 5 : Traiter les erreurs et les exceptions
JOUR 6 : Améliorer l'interface en mode console
JOUR 7 : Créer une interface graphique

INDEX

Le récapitulatif détaillé de toutes les notions et instructions à connaître

Édité par Les Éditions Diamond
L 15066 - 71 H - F: 12,90 € - RD
boutique.ed-diamond.com

Retrouvez toutes nos publications

LES ÉDITIONS DIAMOND

sur **boutique.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
boutique.ed-diamond.com

Directeur de publication : Arnaud Metzler

Rédacteur en chef : Denis Bodor

Remerciements à Tristan Colombo

Illustrations : Pierre Wisson

Conception graphique : Kathrin Scali

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

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

Membre April
www.april.org

PRÉFACE

Vous voulez apprendre à programmer, mais vous ne savez pas vraiment comment vous y prendre pour que cela soit à la fois instructif et amusant ? Nous avons réalisé ce hors-série pour vous !

Tout au long des pages de ce magazine, vous découvrirez de nouvelles notions qui vous seront expliquées simplement et de manière ludique avant d'être mises en pratique dans la réalisation d'un petit jeu. Nous utiliserons pour cela le « petit langage qui monte », Python, qui suivant les classements, apparaît en troisième ou quatrième position des langages les plus utilisés. Autre avantage : ce langage est particulièrement adapté pour l'initiation à la programmation avec une écriture assez simple et structurée.

Ce magazine ne fera pas de vous un crack de Python, son objectif est de vous donner envie de programmer, ou en tout cas, de ne pas vous en dégoûter !! De nombreuses notions, un peu complexes, seront passées sous silence... Vous serez toujours à temps de les aborder plus tard, lorsque vous aurez déjà acquis une petite expérience en programmation.

Au détour d'un forum, vous êtes déjà tombé sur des lignes du type `(cos(a++)-sin(--b)==0)?return a;return b;` et vous n'avez rien compris ? C'est plutôt rassurant... Vous êtes tout à fait normal ! Pas de signes cabalistiques dans ce numéro, pas d'incantations étranges à prononcer devant votre machine. Notez toutefois que les jurons seront admis en cas d'erreur et parfois même recommandés pour sauver le matériel d'une destruction brutale et aveugle que vous pourriez regretter par la suite. Nous avons tout fait pour éviter que vous en arriviez à de telles extrémités, mais si tel était le cas, arrêtez tout et allez prendre l'air. Vous verrez qu'en revenant vos idées seront plus claires et peut-être résoudrez-vous votre problème en quelques secondes (ça arrive très souvent...).

Une autre solution consiste à utiliser la technique du *Teddy Bear* (ours en peluche en anglais) : vous exposez votre problème à haute voix à un animal en peluche, votre animal de compagnie, votre voisine, bref... à ce qui vous passe sous la main. L'interlocuteur n'a pas besoin de comprendre ce que vous racontez (d'où l'intérêt d'utiliser un objet inerte pour ne pas paraître complètement fou) : le simple fait d'énoncer votre problème vous permet de cerner plus précisément ce qui doit être corrigé. La plupart du temps, vous n'irez pas jusqu'à la fin de la description de votre problème, vous trouverez la solution avant ! Attention toutefois à ne pas créer de nouveaux problèmes d'un autre ordre en discutant trop souvent avec la voisine ou le voisin...

Malgré tout ce qui a pu être dit précédemment, ne soyez pas trop anxieux. Ce n'est pas parce que vous allez programmer que vous allez vous retrouver instantanément transformé en *geek* barbu aux cheveux longs et gras, et à la tenue douteuse. Ce que l'on oublie souvent, c'est qu'il n'y a pas que des geeks, les geekettes aussi ça existe :-)

Tristan Colombo

Sommaire

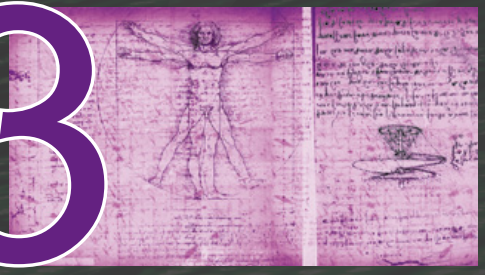
GNU/Linux Magazine Hors-Série N°71

JE PROGRAMME

1

2

3



JOUR 4

JOUR 5

JOUR 6

JOUR 7

INDEX

INTRODUCTION : PRÉPARER SES ARMES

Avant de construire une maison, il faut creuser des fondations. De ces fondations va dépendre la solidité et la longévité de la construction. En informatique, il en va de même. Nous allons installer nos outils et apprivoiser notre environnement de travail pour disposer d'une base suffisamment stable pour ne pas être source d'erreurs.

Si vous avez acheté ce *mook* (livre-magazine), c'est que vous souhaitez apprendre à programmer sans aucune connaissance particulière en informatique (savoir allumer un ordinateur serait un plus !). Cette démarche soulève deux questions principales, qui en appelleront d'autres par la suite :

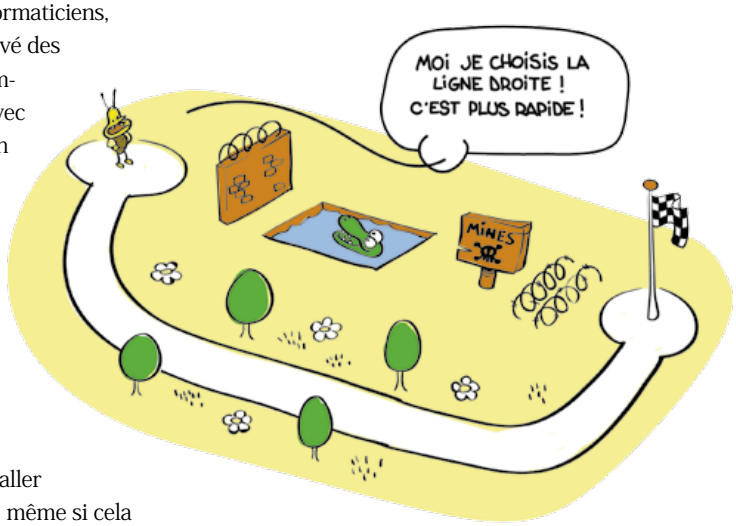
Pourquoi apprendre à programmer ?

Qu'est-ce que la programmation ?

Pour répondre à ces questions et vous permettre d'être complètement autonome pour mener à bien vos différents projets, nous vous proposons d'apprendre à programmer en une semaine. Pourquoi une semaine ? C'est le temps qu'il nous faudra pour mener à bien un petit projet de jeu au cours duquel vous rencontrerez les notions principales de programmation. Pour nous aider dans cette tâche, nous pourrons suivre les conseils plus ou moins avisés de Buggy le petit cafard. Il nous secondera et nous indiquera bien souvent ce qu'il vaut mieux éviter de faire...

1. POURQUOI APPRENDRE À PROGRAMMER ?

Programmer n'est pas réserver aux seuls informaticiens, tout comme écrire n'est pas le domaine réservé des écrivains. L'ordinateur est un outil, certes complexe, mais un simple outil. Pour dialoguer avec cet outil, on passe par des applications qui, en fonction de nos choix, vont déclencher des actions (calculer un résultat, déplacer un personnage à l'écran, imprimer, etc.). Le comportement de ces applications a été déterminé par la ou les personne(s) qui (a) ont écrit le programme. On ne peut pas obtenir un comportement différent de ce qui a été pensé préalablement. Par analogie, s'il existe une route pour aller d'une ville A à une ville B et d'une ville B à une ville C, pour aller de A à C vous devrez forcément passer par B, même si cela constitue un détour : vous n'êtes pas maître du tracé routier.



En sachant programmer, vous pourrez prendre le contrôle complet des applications que vous développerez : votre imagination sera le seul facteur limitant ! De cet apprentissage :

- Vous pourrez bien sûr créer vos propres applications ou vos propres jeux, que ce soit parce que vous ne trouvez pas ce que vous recherchez parmi les applications existantes ou simplement en tant que loisir, pour vous amuser. Vous prendrez du plaisir (et quelques crises de nerfs) à partir de rien pour parvenir à l'application imaginée ;
- Vous comprendrez comment fonctionne un programme et pourquoi les applications que vous utilisez réagissent de telle ou telle façon ;
- Vous pourrez améliorer des programmes libres existants en leur ajoutant les fonctionnalités qui leur font défaut.

La différence entre le fait de programmer et la programmation est ténue, pour ne pas dire inexistante. Je dirai ici que programmer correspond à l'aspect théorique et que la programmation est l'application pratique, technique, du fait de programmer.

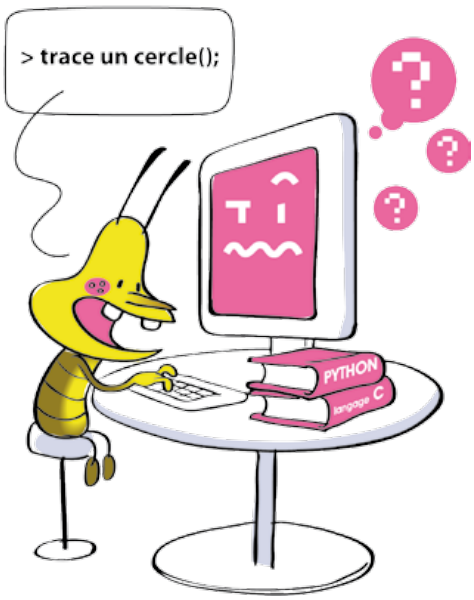
2. QU'EST-CE QUE LA PROGRAMMATION ?

La programmation consiste à dialoguer avec un ordinateur dans un langage qu'il comprend (et que nous comprenons normalement également). Si une personne parle suédois et norvégien et que vous essayez de communiquer avec elle en parlant japonais, vous ne parviendrez pas à vous comprendre. Il faut donc dans un premier temps que vous parliez la même langue (on dit « langage ») que la machine.

Il existe de très nombreux langages de programmation et vous avez sans doute déjà entendu parler du C ou de PHP. Ces langages sont définis, comme pour les langues que nous parlons, par un ensemble de règles grammaticales et syntaxiques. Les langages de programmation ont été inventés pour éviter de communiquer avec les machines dans le seul langage qu'elles comprennent vraiment : le binaire. Les ordres seront traduits en suites de 0 et de 1 avant d'être exécutés. Il existe ainsi deux grandes catégories de langages : les **langages compilés** où les ordres sont traduits en binaire lors d'une étape appelée « compilation » et les **langages interprétés**, qui sont traduits en binaire au cours de leur exécution.

Le dialogue entre un humain et un ordinateur par le biais d'un langage de programmation et de type « maître - esclave » : vous ordonnez et la machine exécute. Si vos ordres, que l'on appelle également des « instructions », ne sont pas compris, l'ordinateur vous l'indiquera par un message d'erreur.

Pour que l'ordinateur puisse comprendre un langage donné, il faut que lui aussi « apprenne » ce langage. Nous utiliserons le langage Python et il faudra donc l'installer de manière à ce que vous puissiez écrire en Python et que l'ordinateur puisse exécuter vos ordres.



À retenir

Un langage de programmation est un ensemble de règles syntaxiques et grammaticales permettant de donner des ordres à un ordinateur par le biais d'instructions.

À retenir

Une instruction est un mot-clé défini dans un langage de programmation et indiquant à la machine une tâche à effectuer.

3. POURQUOI CHOISIR PYTHON ?

Python est un langage relativement récent puisqu'apparu en 1991 et qui possède un grand nombre de qualités :

- Python a été créé dans l'optique d'être le plus simple possible à apprendre : il est simple à lire, à écrire et à comprendre ;
- Il s'agit d'un langage dit de « haut niveau » : vous n'aurez pas à vous occuper de parties techniques et pointues comme l'occupation de la mémoire par exemple ;
- Python est distribué sous licence libre : n'importe quel utilisateur peut étendre les fonctionnalités du langage... et beaucoup l'ont déjà fait et ont partagé leur travail !
- Python est un langage multi-plateforme : vous écrirez le même programme que vous soyez sous Linux, Windows, ou Mac OS X et il pourra être exécuté indifféremment sur l'un de ces systèmes d'exploitation ;
- Python est un langage opérationnel complet : il ne sert pas qu'à apprendre à programmer. De nombreuses applications scientifiques complexes sont développées en Python. À la NASA, au CNRS, chez Google, chez Yahoo, etc., de nombreux projets sont développés en Python.

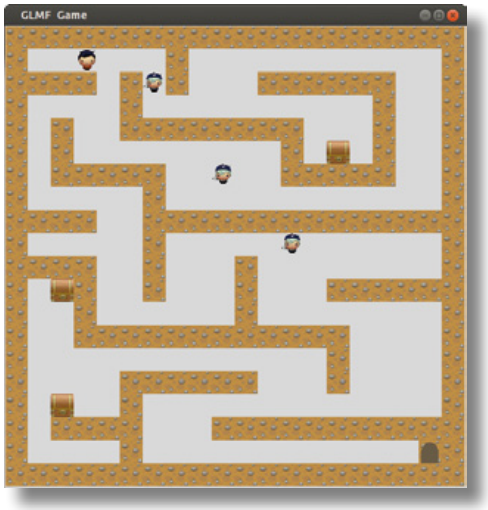
Bien que nous utilisions Python dans ce mook, la majorité des notions qui seront abordées pourront être utilisées avec n'importe quel autre langage, seule la syntaxe sera différente.

4. LE PROJET

Pour apprendre à programmer en Python, nous allons créer en sept jours un petit jeu, étape par étape, en faisant intervenir des notions de plus en plus avancées. Notre objectif est de concevoir un petit jeu dans lequel il faudra déplacer à l'aide du clavier un petit personnage dans un labyrinthe tout en évitant les ennemis, les pièges et en collectant un maximum de pièces d'or.

Voici les étapes que nous allons suivre :

- **Jour 1** : Affichage de la bordure du labyrinthe sous forme de caractères ;
- **Jour 2** : Saisie de caractères et déplacement du personnage (un 'X') ;
- **Jour 3** : Ajout de l'« intelligence » du jeu avec la détection des collisions avec les murs ;
- **Jour 4** : Chargement de différents labyrinthes ou niveaux depuis des fichiers ;
- **Jour 5** : Ajout des ennemis, des pièges et des trésors. Gestion des combats ;
- **Jour 6** : Amélioration de l'interface en mode texte ;
- **Jour 7** : Interface graphique.



Pour vous aider, à tout moment vous pourrez vous reporter aux index à la fin du mook pour trouver des rappels sur des instructions ou des notions déjà vues.

La première des choses à faire maintenant est d'installer Python sur votre ordinateur. Nous avons vu que ce langage est multi-plateforme et nous allons donc voir comment réaliser cette installation sur Linux (distribution Ubuntu ou Linux Mint), Windows (version 7) ou Mac OS X. Vous pouvez vous reporter directement à la section qui vous intéresse.

5. INSTALLER PYTHON SOUS LINUX

Sous Linux, il y a de fortes chances pour que Python soit déjà installé... mais pas forcément la dernière version. De plus, le programme `idle` que nous allons utiliser par la suite n'est certainement pas installé. Nous allons donc voir comment installer tous les programmes dont nous aurons besoin.
Par souci de simplification, je ne détaillerai les installations que sous Ubuntu 13.04 Raring Ringtail (avec les environnements Unity et GNOME) et sous Linux Mint 15 Olivia avec l'environnement Cinnamon.

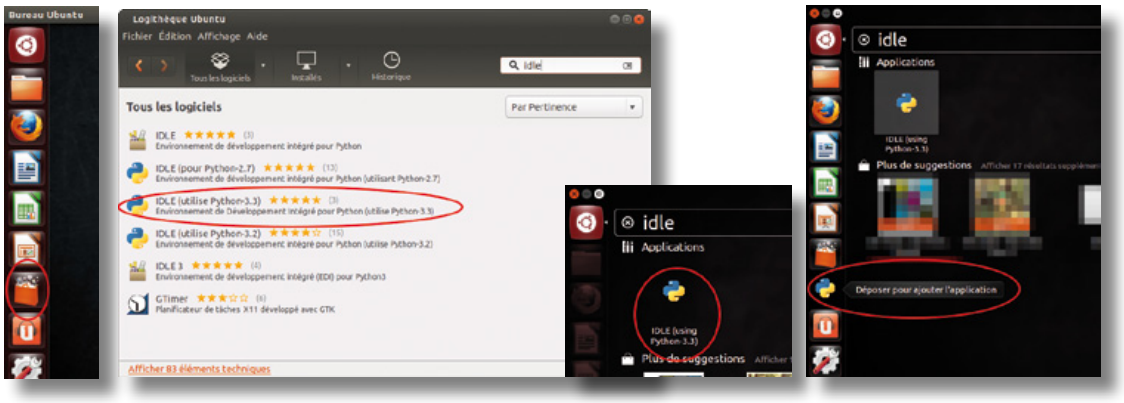
5.1 Linux Ubuntu avec l'environnement Unity

Ouvrez la logithèque Ubuntu en cliquant sur l'icône ***Logithèque Ubuntu*** sur le menu du bureau.

Tapez ensuite « idle » dans la barre de recherche et sélectionnez ***IDLE (utilise Python-3.3)*** dans les résultats.

Le système vous indiquera que le paquetage n'est pas installé et vous n'aurez qu'à cliquer sur le bouton ***Installer***. L'accès au programme se fera ensuite par le tableau de bord en tapant « idle » dans la barre de recherche.

Si vous souhaitez ajouter un raccourci dans la barre de menu du bureau, après avoir recherché « idle » dans le tableau de bord, cliquez sur l'icône et déplacez-la vers le menu.

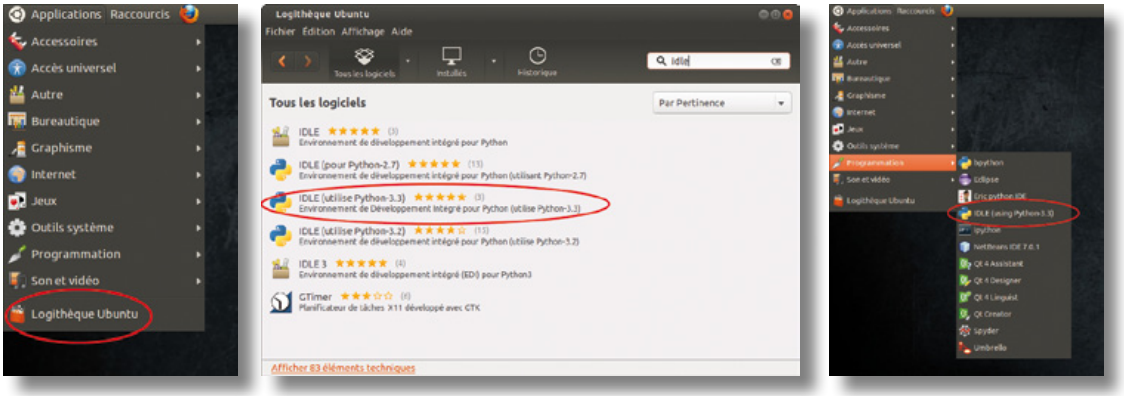


5.2 Linux Ubuntu avec l'environnement GNOME

Ouvrez la logithèque Ubuntu qui se trouve dans ***Applications > Logithèque Ubuntu***.

Tapez ensuite « idle » dans la barre de recherche et sélectionnez ***IDLE (utilise Python-3.3)*** dans les résultats.

Le système vous indiquera que le paquetage n'est pas installé et vous n'aurez qu'à cliquer sur le bouton ***Installer***. L'accès au programme se fera ensuite par le menu ***Applications > Programmation***.



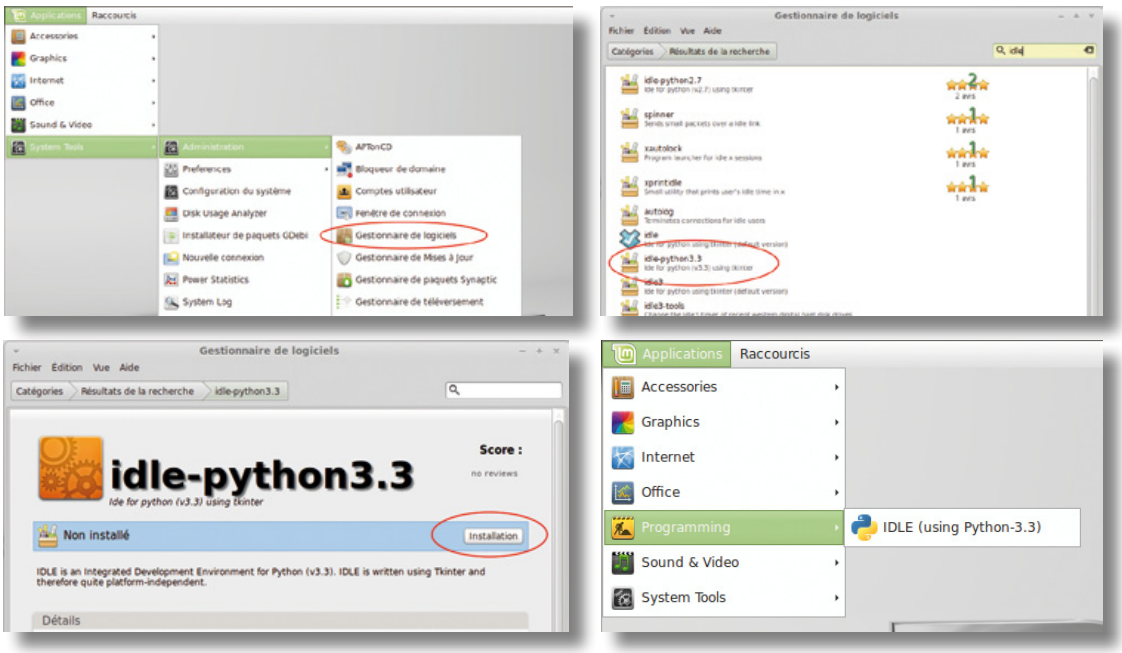
5.3 Linux Mint avec l'environnement Cinnamon

Ouvrez le gestionnaire de logiciels qui se trouve dans ***Applications > System Tools > Administration***.

Tapez ensuite « idle » dans la barre de recherche et sélectionnez ***idle-python3.3*** dans les résultats.

Le système vous indiquera que le paquetage n'est pas installé et vous n'aurez qu'à cliquer sur le bouton ***Installation***.

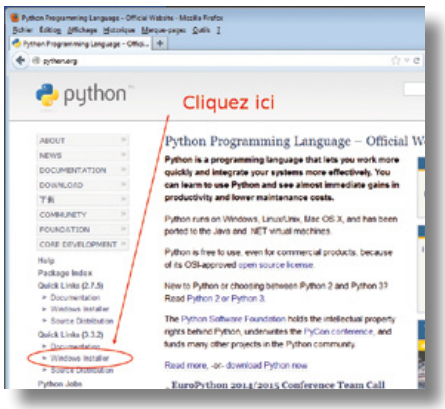
L'accès au programme se fera par le menu ***Applications > Programming***.



6. INSTALLER PYTHON SOUS WINDOWS 7

À l'aide d'un navigateur, rendez-vous sur la page **http://www.python.org** et dans le menu de gauche intitulé ***Quick Links (3.3.2)***, cliquez sur le lien ***Windows Installer*** qui vous proposera de télécharger le fichier `python-3.3.2.msi` comme le montre la capture à gauche.

Une fois le fichier téléchargé, dans votre répertoire de téléchargements, double-cliquez sur `python-3.3.2.msi` et suivez les indications à l'écran.



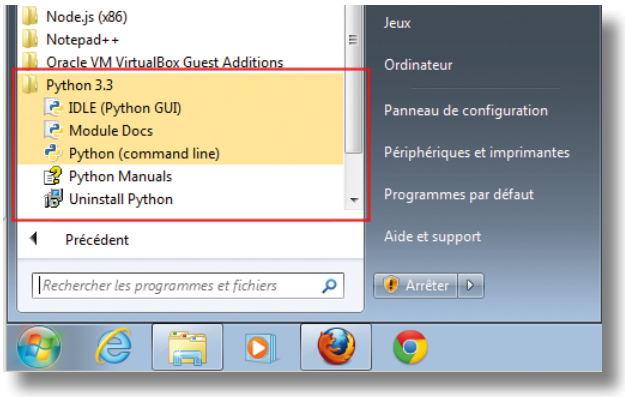
NOTE

Il est possible que la version que vous téléchargiez soit supérieure à 3.3.2. Cela est sans importance ! Assurez-vous simplement que votre numéro de version commence par un 3 et non pas par un 2.

Voici les réponses à apporter aux différents écrans :

1. Laissez le choix par défaut ***Install for all users*** et cliquez sur le bouton ***Next*** ;
2. Laissez le répertoire d'installation par défaut (vous pouvez le modifier, mais notez le chemin complet d'installation). En laissant la valeur par défaut, vous devriez avoir un chemin qui ressemble à `C:\Python33\`. Cliquez ensuite sur le bouton ***Next*** ;
3. Vous arrivez au dernier écran permettant de personnaliser votre installation de Python (***Customize Python 3.3.2***). Laissez tous les paramètres par défaut et cliquez simplement sur le bouton ***Next*** ;

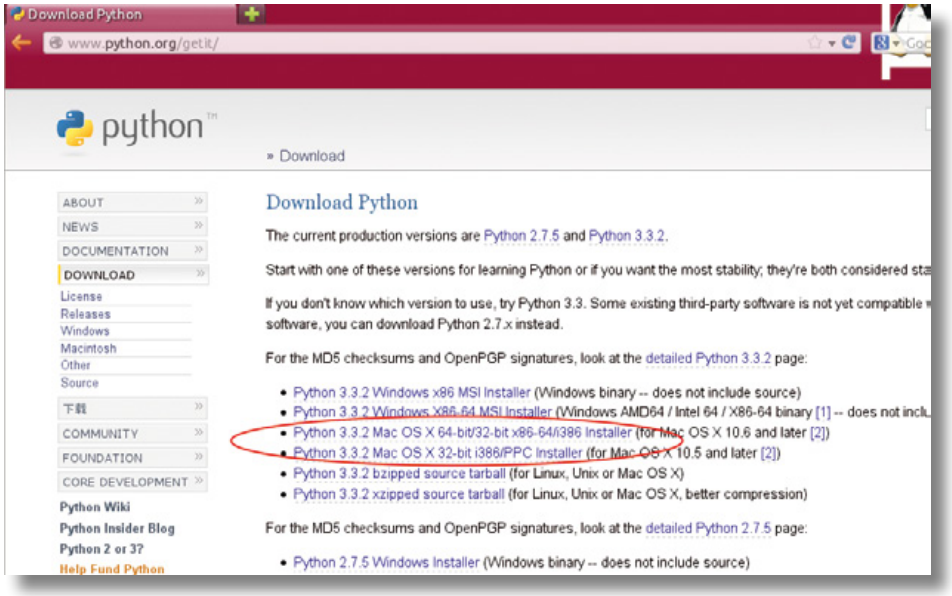
Une barre de progression apparaît, il ne vous reste plus qu'à attendre puis à cliquer sur le bouton ***Finish*** pour quitter le programme d'installation. Dans votre menu d'applications, de nouvelles entrées sont apparues :

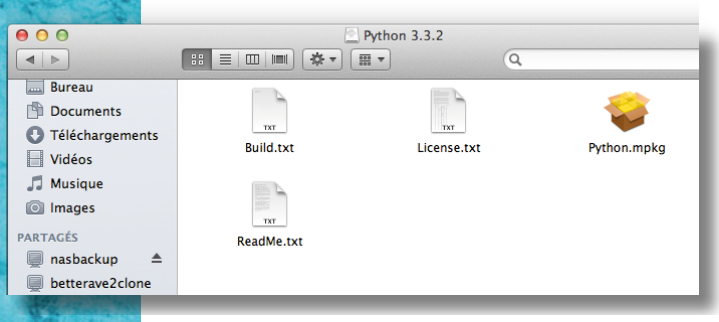


Nous nous servirons de l'entrée ***IDLE***.
Si vous souhaitez ajouter un raccourci vers ce programme, effectuez un clic droit sur votre bureau, sélectionnez l'entrée ***Nouveau***, puis ***Raccourci*** et indiquez le chemin : `C:\Python33\Lib\idlelib\idle.pyw -n`. Donnez un nom à votre raccourci et validez-le. Vous pouvez maintenant vous rendre à la section « Et maintenant ? » pour comprendre ce qu'est ce programme que nous venons d'installer et comment l'utiliser.

7. INSTALLER PYTHON SOUS MAC OS X

Pour être sûr d'obtenir la dernière version de Python sur Mac OS X, rendez-vous à l'aide d'un navigateur à l'adresse **http://www.python.org/getit**. Dans la liste des premiers liens vous trouverez les programmes d'installation pour Mac OS X 10.5 ou supérieur à 10.6 :





Le fichier téléchargé est un fichier d'extension `.dmg` du type `python-3.3.2-macosx10.6.dmg`. Double-cliquez sur l'icône de cette archive pour ouvrir une fenêtre montrant les fichiers qu'elle contient.

Double cliquez sur l'icône `Python.mpkg` et suivez les instructions à l'écran.

Si vous le souhaitez, vous pouvez maintenant créer un raccourci sur le bureau vers le programme que nous venons d'installer.

Voyons à présent quoi faire de ce programme.

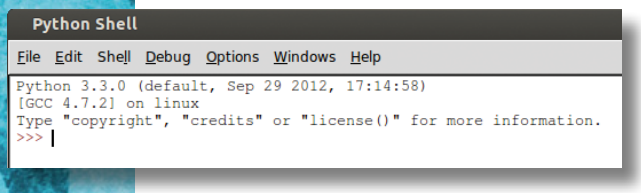
8. ET MAINTENANT ?

Nous avons installé Python et un mystérieux programme nommé `idle`. Qu'allons nous faire de cela ?

Comme vous l'avez sans doute remarqué lors de l'installation, il existe deux versions de Python : une version 2.x.x et une version 3.x.x. Ces deux versions ne sont pas compatibles, le code que nous allons apprendre ne pourra pas être exécuter avec Python 2.x.x. La compatibilité entre les deux versions (appelée rétro-compatibilité) a été supprimée de manière à permettre au langage de continuer à évoluer et de corriger ses erreurs de jeunesse. Ne vous affolez pas, ce genre de décision intervient rarement dans la vie d'un langage et c'est plutôt un bon signe prouvant que le langage « vit », continue à évoluer. Nous développerons notre projet avec la dernière version du langage, à savoir Python 3.

À savoir

Il existe deux versions de Python, Python 2 et Python 3, qui ne sont pas compatibles entre elles.



Revenons maintenant sur ce mystérieux programme `idle`. Lancez-le en suivant la procédure relative à votre système d'exploitation. Une nouvelle fenêtre intitulée « Python shell » va apparaître. Il s'agit d'une application permettant d'écrire et d'exécuter des instructions en Python.

Les trois petits signes « supérieur à » que vous voyez sur la dernière ligne indiquent que le programme attend que l'on saisisse une information. On appelle ces symboles **le prompt**. Si vous tapez un calcul quelconque, vous obtiendrez le résultat de ce calcul comme si vous étiez face à une calculatrice géante :

Terminal

```
>>> 2 + 2
4
```

L'avantage par rapport à une simple calculatrice, c'est que vous pouvez réaliser de très gros calculs :

Terminal
```
>>> 123456789101112131415 16 * 9876543210
121932631212499102497702618906360
```

Le résultat apparaît en bleu alors que l'opération est affichée en noir. Essayons de taper une autre instruction :

Terminal
```
>>> print(2 + 2)
4
```

Ici, le mot « print » apparaît en rose. Et vous aurez peut-être remarqué l'apparition d'un petit cadre alors que vous tapiez ce mot :

```
>>> print(
      print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Il s'agit d'une bulle d'information indiquant comment s'utilise l'instruction **print**. Nous analyserons ce message en détail plus tard.

print est un mot réservé du langage qui signifie « écrit ». Nous ordonnons à la machine d'afficher le résultat de l'opération indiquée entre parenthèses. Pour l'instant, il n'y a pas de grandes différences entre cette écriture et un simple « 2 + 2 », mais nous n'allons pas tarder à voir que ces deux lignes ne sont pas équivalentes.

Nous pourrions également demander d'afficher un texte :

Terminal
```
>>> print("J'apprends à programmer en Python")
J'apprends à programmer en Python
```

Le texte est noté entre guillemets : on ouvre les guillemets pour indiquer que les caractères qui vont suivre font partie du texte et on ferme les guillemets pour indiquer que notre texte est terminé. Un texte étant constitué par une suite de lettres, ou caractères, en informatique on parle de **chaîne de caractères**.

Si nous tapons un ordre que Python ne comprend pas, nous obtiendrons un message d'erreur rouge vif :

Terminal
```
>>> trace un cercle
SyntaxError: invalid syntax
```

La chaîne de caractères apparaît en vert… Mais d'où viennent donc toutes ces couleurs ? Il s'agit de la **coloration syntaxique** : pour aider à la lecture des programmes (on parle du **code source**, ou par raccourci du **code**), on associe à chaque catégorie d'éléments composant une instruction une couleur. Nous avons pu voir que les chaînes de caractères apparaissent en vert, les mots-clés correspondant à des noms d'instruction Python en rose et les résultats d'instructions en bleu.

Si vous fermez cette fenêtre en cliquant sur ***File > Exit***, puis que vous relancez `idle`, vous vous apercevrez que toutes vos lignes de code ont disparu… Au lancement, `idle` fonctionne en mode interactif : vous tapez du code Python, `idle` l'exécute et vous affiche le résultat. Si vous souhaitez conserver votre code, il faut cliquer sur ***File > New Window***. Une nouvelle fenêtre apparaîtra dans laquelle vous ne verrez plus le prompt. Il s'agit d'un éditeur de code : comme avec un traitement de texte, vous pouvez taper du texte (sans aucun style), puis l'enregistrer en cliquant sur ***File > Save*** ou en appuyant simultanément sur les touches [Ctrl] et [s]. On vous demandera de donner un nom à votre programme : choisissez ce que vous voulez, mais donnez-lui l'extension `.py`, ceci indique qu'il s'agit d'un fichier Python.

Par exemple, nous pouvons reprendre l'instruction permettant d'afficher du texte et l'enregistrer dans un fichier `premier_programme.py` :

Fichier
```
print("J'apprends à programmer en Python")
```

Vous pouvez voir que la coloration syntaxique s'applique également à cette fenêtre. Pour exécuter votre programme, cliquez maintenant sur ***Run*** puis ***Run Module***, ou alors appuyez sur la touche [F5]. Dans la fenêtre précédente, vous verrez apparaître une ligne indiquant le début du programme, puis le résultat de votre instruction :

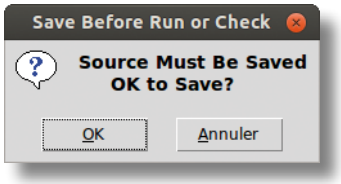
Terminal
```
>>> ========================= RESTART ============================
>>>
J'apprends à programmer en Python
>>>
```

Si vous souhaitez exécuter plusieurs instructions les unes à la suite des autres, il suffit de les écrire les unes en-dessous des autres. Ajoutons notre calcul `2 + 2` :

Fichier
```
print("J'apprends à programmer en Python")
2 + 2
```

À savoir

Le fichier contenant l'ensemble des instructions permettant à un programme de fonctionner est appelé code source.



Exécutez le programme. Mais... nous avons oublié de sauvegarder le code ! Une petite fenêtre nous rappelle à l'ordre et nous demande de confirmer l'enregistrement des modifications avant exécution.

Le résultat obtenu... est le même que le précédent ! Nous ne voyons pas de **4** en plus. Ceci est tout à fait normal : lorsque l'on tape 2 + 2 dans la fenêtre interactive (après le prompt), Python réagit comme une calculatrice et nous affiche tous les résultats qu'il est en mesure de calculer. Par contre, lorsque nous écrivons un programme, Python calcule **2 + 2**, mais comme nous ne lui avons pas demandé explicitement d'afficher le résultat il ne se passe rien. Modifiez votre code de la manière suivante :

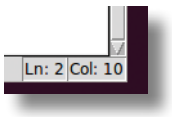
Fichier

```
print("J'apprends à programmer en Python")
print(2 + 2)
```

Cette fois l'instruction **print** demande d'afficher le résultat de **2 + 2** et nous obtenons :

Terminal

```
>>> ========================== RESTART ==============================
>>>
J'apprends à programmer en Python
4
>>>
```



Vous voilà presque prêt. Un dernier détail avant que vous ne passiez au premier jour de notre projet : au bas de la fenêtre de code, vous pouvez voir deux informations : le numéro de ligne et le numéro de colonne indiquant la position du curseur.

N'hésitez pas à tester et refaire les manipulations que nous venons d'effectuer jusqu'à les maîtriser parfaitement. ▉

Pour récapituler :

- Lorsque l'on veut exécuter directement des lignes de code en Python, on lance **idle** et on écrit les commandes après le prompt **>>>** ;
- Lorsque l'on veut écrire un programme, on lance **idle** puis dans le menu on sélectionne ***File > New Window*** qui ouvre une nouvelle fenêtre dans laquelle nous allons écrire nos lignes de code. Pour pouvoir exécuter le programme, il faut le sauvegarder en cliquant sur ***File > Save*** ou en appuyant sur les touches [Ctrl] et [s], indiquer son nom (extension **.py**) et cliquer sur ***Run > Run Module*** ou appuyer sur la touche [F5]. Le résultat du programme apparaît ensuite dans la fenêtre interactive (première fenêtre ouverte).

JOUR 1

JOUR 1

AFFICHER DES CARACTÈRES

Pour notre jeu, nous allons commencer par afficher la bordure du labyrinthe sous la forme de caractères. Cette tâche est simple, mais elle va faire surgir de nombreux problèmes que nous résoudrons en faisant intervenir des notions de programmation particulières.

1. Les chaînes de caractères
2. Des données dans un tableau
3. print pour afficher des données
4. Boucler pour ne pas répéter la même chose

Nous avons vu précédemment comment afficher une chaîne de caractères à l'écran à l'aide de l'instruction **print** tapée dans le shell Python disponible dans l'application **idle**. Si nous choisissons le caractère **#** pour indiquer un mur, pour afficher un mur horizontal nous pouvons taper :

Terminal

```
>>> print("####################")
####################
```

Finalement, ce n'est pas bien compliqué... Et si nous demandions directement à ce que tout le labyrinthe soit affiché en retournant à la ligne à la fin de chaque portion de mur ?

Terminal

```
>>> print("####################
SyntaxError: EOL while scanning string literal
```

Un message apparaît en rouge... La machine ne nous comprend plus ! Pour que nous puissions corriger notre code, Python nous indique le type d'erreur qu'il a détecté. Toutes les erreurs se présenteront sous la même forme : la catégorie de l'erreur suivie du caractère « : », puis une phrase de description de cette erreur.

Ici, il s'agit d'une erreur de syntaxe (**SyntaxError**), c'est-à-dire que vous n'avez pas respecté la grammaire du langage. Le message nous indique que la fin de ligne (**EOL** pour *End Of Line*) a été atteinte alors que la chaîne de caractères (**string**) n'était pas terminée. Pour ne pas se retrouver constamment confrontés à ce type d'erreur sur les chaînes de caractères, nous devons comprendre comment elles doivent être écrites.

À savoir

Une erreur intervient lorsque la machine n'est plus capable de comprendre le code que vous avez écrit. L'exécution du programme est alors interrompue et un message d'erreur est affiché pour indiquer pourquoi le programme s'est arrêté.

1. LES CHAÎNES DE CARACTÈRES

Jusqu'ici nous avons encadré nos chaînes de caractères par des guillemets (et l'absence de guillemet pour refermer la chaîne a provoqué une erreur de syntaxe). Ce caractère étant utilisé d'une manière spéciale pour indiquer le début et la fin d'une chaîne, comment faire pour l'insérer à l'intérieur d'une chaîne de caractères ? En effet, si nous l'intégrons dans une chaîne, nous obtenons un message d'erreur :

Terminal
```
>>> print("Il dit : "Coucou, c'est moi!"")
SyntaxError: invalid syntax
```

Cette erreur est tout à fait prévisible, car elle nous a été signalée avant que nous ne validions notre code ! Comment ? Grâce aux couleurs qui sont employées pour afficher les caractères que nous tapons. Regardez bien les couleurs de votre ligne de code : il y a du violet pour l'instruction `print`, puis du vert et du noir. Le vert désigne en fait tout ce qui a été reconnu comme une chaîne de caractères. Nous voyons ainsi que `"Il dit :"` est une chaîne, que `Coucou, c` est analysé comme une instruction (qui ne sera bien sûr pas comprise) et que `'est moi!""")` est une chaîne qui n'est visiblement pas fermée... Étrange, car elle ne commence pas par des guillemets. Faisons donc un petit test :

Terminal
```
>>> print('Ceci est une chaîne de caractères')
Ceci est une chaîne de caractères
```

Nous n'obtenons pas d'erreur ! Une chaîne de caractères peut donc être encadrée soit par des guillemets, soit par des apostrophes. Donc, si nous voulons afficher le caractère guillemet, le plus simple est d'encadrer notre chaîne de caractères par des apostrophes :

Terminal
```
>>> print('Il dit : "Coucou!"')
Il dit : "Coucou!"
```

Mais que se passe-t-il si nous souhaitons intégrer à une même chaîne des guillemets et des apostrophes ?

Terminal
```
>>> print('Il dit : "Coucou, c'est moi!"')
SyntaxError: invalid syntax
```

Ici, nous avons défini une première chaîne de caractères par `'Il dit : "Coucou, c'` et une seconde chaîne par `"')`, chaîne pour laquelle il manque les guillemets de fermeture. Pour empêcher qu'un caractère guillemet ou apostrophe soit interprété en tant qu'ouverture ou fermeture d'une chaîne de caractères, il faut le faire précéder par un caractère `\`. On dit que l'on « protège » le caractère. Les deux écritures suivantes sont alors équivalentes :

À retenir

Une chaîne de caractères est une suite de caractères encadrés par des guillemets "..." ou par des apostrophes '...'.

À retenir

Le caractère \ est utilisé pour protéger un caractère – ne pas l'interpréter, le conserver tel quel – ou introduire un caractère spécial comme la tabulation \t ou le retour à la ligne \n.

Terminal

```
>>> print('Il dit : "Coucou, c\'est moi!"')
Il dit : "Coucou, c'est moi!"
>>> print("Il dit : \"Coucou, c'est moi!\"")
Il dit : "Coucou, c'est moi!"
```

En utilisant ce type d'écriture, il existe des caractères spéciaux qui désignent par exemple la tabulation (`\t`) ou le retour à la ligne (`\n`). De ce fait, si vous voulez insérer un anti-slash dans une chaîne de caractères, il faudra lui même le protéger : `\\`.

1.1 Chaînes de caractères sur plusieurs lignes

Nous savons maintenant comment correctement débuter une chaîne de caractères, mais nous n'avons pas la solution pour définir une chaîne sur plusieurs lignes, quoique... En utilisant le retour à la ligne `\n`, nous pourrions définir le labyrinthe :

Terminal

```
>>> print("####################\n#    #       #     #\n###
#######   #")
####################
#    #       #     #
###      #######   #
```

Ça fonctionne... Mais ce n'est pas très lisible. Pour vraiment pouvoir écrire une chaîne de caractères sur plusieurs lignes, il faut tripler les caractères d'ouverture et de fermeture que sont les guillemets ou les apostrophes. Si nous choisissons les guillemets comme caractères d'encadrement et que nous reprenons l'exemple précédent, cela donne :

Terminal

```
>>> print("""####################
#    #       #     #
###      #######   #""")
####################
#    #       #     #
###      #######   #
```

C'est déjà mieux, même si les caractères de la première ligne ne sont pas alignés avec les autres.

1.2 Stockage dans une variable

Le décalage entre la première ligne et les autres est induit par l'appel à l'instruction **`print`** et les trois guillemets ouvrant la chaîne de caractères. Mais nous n'avons pas forcément besoin de cette instruction pour manipuler une chaîne de caractères, elle peut avoir une existence propre complètement indépendante d'un appel à une quelconque instruction, comme le montre le code suivant :

Terminal

```
>>> "Une chaîne de caractères"
'Une chaîne de caractères'
```

L'interpréteur Python nous répond qu'il s'agit bien d'une chaîne de caractères contenant « Une chaîne de caractères ». Il n'y a pas d'erreur : cette expression est donc valide. Par contre, elle ne fait absolument rien ! Si nous lançons le même code depuis l'éditeur en le sauvegardant puis en l'exécutant, il ne se passera rien du tout.

Pour vous le prouver, dans le menu du shell Python, cliquez sur ***File*** puis sur ***New Window*** et tapez votre code dans la fenêtre venant d'apparaître (vous remarquerez l'absence de prompt, c'est-à-dire les trois signes **>>>**). Enregistrez le code sous la forme d'un fichier en cliquant sur ***File > Save*** (ou [Ctrl]+[s]), donnez un nom se terminant par **.py** pour retrouver plus simplement vos scripts Python, puis lancez l'exécution de votre programme en cliquant sur ***Run > Run module*** (ou [F5]).

Pour pouvoir réutiliser cette chaîne, il faut l'enregistrer dans la mémoire de l'ordinateur et pour pouvoir la retrouver il faut lui associer une étiquette, un nom compréhensible par un être humain qui soit différent des noms utilisés par la machine et que l'on appelle « des adresses mémoire ». Lorsque l'on utilise le nom de notre étiquette, nous retrouvons la valeur qui lui est associée. C'est ce que l'on appelle **une variable**.

Imaginez une énorme bibliothèque contenant des milliers et des milliers d'ouvrages. À chacun de ces livres nous associons une étiquette contenant un nom unique (il est impossible de trouver deux fois le même nom sur l'ensemble des étiquettes utilisées dans cette bibliothèque). Pour retrouver un livre, il nous suffit de connaître le texte contenu par l'étiquette qui, en général, nous indiquera en plus la position du livre. Nous aurons ainsi accès très simplement à l'information stockée à cet emplacement, à savoir le contenu du livre.

Pour aller plus loin, nous pouvons même considérer qu'une nouvelle édition du livre paraît : nous remplaçons l'ancien livre par le nouveau sans changement d'étiquette. Nous avons ainsi modifié le contenu, mais pas la méthode d'accès.

Pour les variables, tout fonctionne exactement comme avec la bibliothèque :

⇨ On associe une étiquette à une information :

Terminal

```
>>> ma_variable = "Une chaîne de caractères"
```

À retenir

Une variable est une étiquette associée à une information stockée dans la mémoire de l'ordinateur. Le nom d'une variable doit respecter certaines règles :

- ⇨ Il ne peut pas s'agir d'un nom déjà utilisé pour désigner une instruction Python (par exemple `print`) ;
- ⇨ Il doit forcément commencer par une lettre (majuscule ou minuscule) ou par le caractère *underscore* `_` ;
- ⇨ La suite du nom peut comporter des lettres, des chiffres et des caractères *underscore*.

Une variable ne contient pas nécessairement une chaîne de caractères ; il peut s'agir d'un entier, d'un nombre réel, etc.

NOTE

Attention : lorsque vous choisissez un nom de variable, Python distingue les lettres minuscules des lettres majuscules. On dit qu'il est sensible à la casse. Les variables `ma_variable`, `Ma_variable` et `MA_VARIABLE` sont ainsi trois variables distinctes.

⇨ On récupère l'information à partir de l'étiquette :

Terminal

```
>>> print(ma_variable)
Une chaîne de caractères
```

⇨ On peut mettre à jour les informations :

Terminal

```
>>> ma_variable = "Des caractères"
>>> print(ma_variable)
Des caractères
```

NOTE

Si à la suite d'une erreur de manipulation vous voyez apparaître un message similaire à `<... at 0x251ba68>`. Il ne s'agit pas d'un message d'erreur, simplement d'un avertissement indiquant que vous manipulez l'adresse mémoire `0x251ba68`.

Pour notre labyrinthe, nous pouvons ainsi définir une variable qui contiendra le premier niveau :

Terminal

```
>>> level_1 = """####################
#    #       #     #
###      #######   #"""
```

Nous pouvons stocker des niveaux aisément, mais le problème de lecture du niveau est toujours présent. Essayons une autre écriture :

Terminal

```
>>> level_1 = """
####################
#    #       #     #
###      #######   #"""
```

Cela paraît plus satisfaisant. Mais si nous affichons le contenu de cette variable à l'aide d'un `print`, nous allons nous apercevoir que le premier saut à la ligne, après les `"""` du début de la chaîne, est conservé :

Terminal

```
>>> print(level_1)

####################
#    #       #     #
###      #######   #
```

En Python, pour indiquer que nous souhaitons aller à la ligne et qu'une instruction n'est pas terminée, il faut utiliser le caractère anti-slash. Si nous l'ajoutons avant notre premier passage à la ligne, le problème est réglé :

Terminal

```
>>> level_1 = """\
##########T###########
#    #       #     #
###      #######    #"""
>>> print(level_1)
####################
#    #       #     #
###      #######    #
```

1.3 Multiplication de chaînes de caractères

Pour de petits labyrinthes de 20 x 20 cases, la méthode précédente est déjà fastidieuse... Imaginez s'il faut créer des labyrinthes de 100 x 100 ! Une des solutions que l'on peut appliquer en Python est de multiplier une chaîne de caractères... Oui, vous avez bien lu : « mul-ti-pli-er ». La chaîne de caractères sera répétée **n** fois :

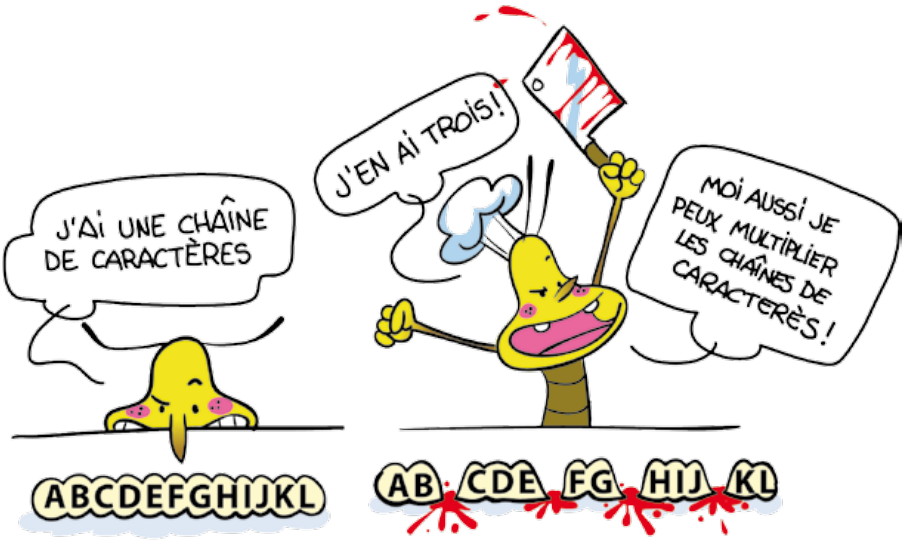
Terminal

```
>>> print("coucou " * 3)
coucou coucou coucou
```

Ce mécanisme peut bien sûr être employé avec les variables :

Terminal

```
>>> mur_complet = "#" * 100
>>> print(mur_complet)
#####...####
```



Et si nous souhaitons insérer le contenu d'une variable à l'intérieur d'une chaîne ? Par exemple, nous devrons déplacer un personnage que nous symboliserons par le caractère **X** :

Terminal

```
>>> perso = "X"
```

Comment faire pour qu'il apparaisse à l'intérieur d'une autre chaîne représentant une partie du labyrinthe ?

À retenir

La concaténation est une opération qui s'effectue entre deux chaînes de caractères et consiste à les assembler pour n'en former plus qu'une seule. On peut appliquer plusieurs concaténations consécutives pour relier plusieurs chaînes de caractères.

1.4 Affichage formaté

La première solution est de coller plusieurs chaînes de caractères entre elles pour n'en former plus qu'une. Ce mécanisme est appelé **concaténation** et s'effectue à l'aide du signe **+** :

Terminal
```
>>> "Bonjour, " + "c'est " + "moi"
"Bonjour, c'est moi"
```

L'opérateur « plus » agit donc comme une colle entre les chaînes de caractères. Attention toutefois, Python sait additionner deux entiers et « coller » des chaînes de caractères... Par contre, il ne sait pas additionner un entier et une chaîne, ni même les « coller » :

Terminal
```
>>> 1000 + 230
1230
>>> "Votre score : " + 1230
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    "Votre score : " + 1230
TypeError: Can't convert 'int' object to str implicitly
```

Nous obtenons une erreur `TypeError` nous indiquant qu'il faudrait convertir l'entier en chaîne de caractères pour que cela fonctionne. En Python, l'instruction `str` permet justement d'effectuer cette conversion :

Terminal
```
>>> str(1230)
'1230'
```

Notez la présence des apostrophes qui indiquent qu'il s'agit bien d'une chaîne de caractères. Donc, pour concaténer un entier et une chaîne, il faut faire appel à `str` :

Terminal
```
>>> "Votre score : " + str(1230)
'Votre score : 1230'
```

Pour insérer le personnage dans le labyrinthe, comme il ne s'agit que de chaînes de caractères, nous n'aurons aucune conversion à effectuer et la concaténation fonctionne de la même manière avec des variables :

Terminal
```
>>> print("#       " + perso + "          #")
#       X          #
```

Il existe une autre méthode plus élégante : **le formatage**. Cette technique va consister à créer une chaîne de caractères dans laquelle nous indiquerons à l'aide de symboles spéciaux qu'il y a des « trous », des zones d'insertion qui seront comblées par des valeurs :

Fichier
```
Je __(1)__ Linux __(2)__
```

Si nous disons que 1 vaut « lis » et 2 vaut « Pratique », nous obtenons :

Fichier

```
Je lis Linux Pratique
```

Python permet de mettre en place exactement cette méthode. Les « trous » du texte seront représentés par `{}` et pour indiquer les valeurs de remplacement nous utiliserons l'instruction `format` :

Terminal

```
>>> print("#       {}          #".format(perso))
#       X          #
```

Le formatage a deux intérêts : il peut servir de modèle et la conversion sous forme de chaîne de caractères des valeurs à insérer se fait de manière complètement automatique.

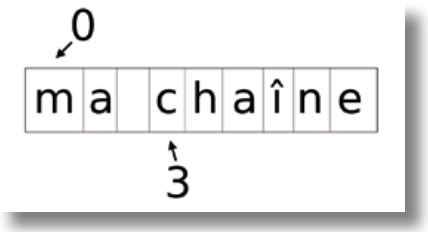
Terminal

```
>>> print("Votre {} : {}".format("score", 1230))
Votre score : 1230
>>> modele = "Votre {} : {}"
>>> print(modele.format("score", 1230))
Votre score : 1230
>>> print(modele.format("nombre de vie(s) : ", "deux"))
Votre nombre de vie(s) : deux
```

Nous sommes maintenant capables d'afficher le labyrinthe de différentes manières. Pour pouvoir suivre la position du personnage, il serait plus simple de parler de coordonnées `(x, y)` où `x` désigne le numéro de ligne et `y` le numéro de colonne sur la grille du labyrinthe. Ça tombe bien, les chaînes de caractères sont déjà adaptées à ce type de traitement !

1.5 Accès par index

Une chaîne de caractères est une suite de caractères : il y a le premier caractère, puis le deuxième, le troisième, etc., jusqu'au dernier caractère. Nous pouvons alors représenter notre chaîne comme un tableau où la première case contient la première lettre et ainsi de suite. Pour repérer plus facilement chaque case, nous allons les numéroter. Un être humain commencerait par 1… Un ordinateur commence par 0 (pour différentes raisons très valables). La figure 1 montre donc une chaîne de caractères représentée sous la forme d'un tableau où les cases sont numérotées. On dit qu'il est **indexé**.



► *Fig. 1 : Une chaîne de caractères sous la forme d'un tableau indexé de 0 à 8.*

À partir de ces index, on peut indiquer quel caractère on souhaite afficher :

Terminal

```
>>> variable = "ma chaîne"
>>> print(variable[0])
m
>>> print(variable[3])
c
```

Dans le cas du labyrinthe, nous avons ainsi accès aux colonnes… Mais comment faire pour les lignes ?

2. DES DONNÉES DANS UN TABLEAU

Python nous permet de créer nos propres tableaux, appelés **listes**. Pour définir un tableau, il suffit d'indiquer ses différents éléments entre crochets :

Terminal

```
>>> tab = ["########", "#      #"]
>>> print(tab[0])
########
```

Le premier élément de notre tableau, `tab[0]`, que nous avons affiché étant une chaîne de caractères, on peut le considérer lui aussi comme un tableau pour n'afficher que le deuxième caractère :

Terminal

```
>>> print tab[0][1]
#
```

Nous pouvons ainsi créer des listes (ou tableaux) à 2, 3 ou n dimensions. Pour l'utilisation que nous allons en faire pour stocker notre labyrinthe de 20 x 20, nous pouvons définir la liste suivante en raffinant un peu l'affichage : un `-` représente un mur horizontal, un `|` représente un mur vertical et un `+` représente un angle :

Terminal

```
>>> level_1 = [
  "+------------------+",
  "|                  |",
  ...
  "|                  |",
  "+------------------+"
]
```

Ces points de suspension indiquent une répétition de la ligne précédente (pour ne pas la répéter 18 fois). Il ne s'agit pas d'un code Python !

La variable `level_1` contient le labyrinthe du premier niveau, il reste maintenant à l'afficher.

À retenir

Une liste est une structure pouvant contenir des éléments de différents types (chaîne de caractères, entiers, etc.) organisés sous la forme d'un tableau et accessibles grâce à un index.

3. PRINT POUR AFFICHER DES DONNÉES

Lorsque l'on utilise l'instruction `print`, elle affiche les données qui lui sont passées en paramètre (ce que l'on indique entre parenthèses). Jusqu'à présent, nous avons utilisé cette instruction de façon très basique, mais il va maintenant falloir comprendre comment elle fonctionne car, visiblement, l'affichage d'une liste ne fonctionne pas comme nous l'aurions souhaité :

Terminal

```
>>> print(level_1)
['+------------------+', '|                  |', ..., '|                  |', '+------------------+']
```

Nous obtenons l'affichage de la liste sous la forme... d'une liste. Cela semble logique. Mais comment fonctionne donc cette instruction ?

`print` concatène et affiche les données qui lui sont transmises en paramètres entre parenthèses et séparées par des virgules :

Terminal
```
>>> print("Ma", "chaîne")
Ma chaîne
```

Vous remarquerez que le caractère espace a été ajouté automatiquement entre les deux chaînes de caractères : il s'agit du caractère de séparation par défaut. Pour modifier ce caractère, il faut ajouter un paramètre sous la forme `sep="..."` (comme pour la création d'une variable) :

Terminal
```
>>> print("Ma", "chaîne", sep="___")
Ma___chaîne
```

De la même manière, il est possible de modifier le caractère de fin de ligne qui est par défaut un retour à la ligne `\n`. Le paramètre à ajouter est alors `end="..."` :

Terminal
```
>>> print("Ma", "chaîne", sep="___", end="!")
Ma___chaîne!
```

En mode interactif dans le shell Python cela n'est pas visible, mais dans un programme, un deuxième `print` afficherait ces données juste après le point d'exclamation. Enregistrez et exécutez le code suivant pour vous en convaincre :

Fichier
```
score = 1230
vies = 3
print("Votre", "score :", score, end="\t")
print("Vie(s) :", vies)
```

Vous obtiendrez un affichage sur une seule ligne :

Terminal
```
Votre score : 1230          Vie(s) : 3
```

Tout cela ne résout pas notre problème de liste… En fait, ce qu'il faudrait faire c'est parcourir toutes les lignes du tableau pour les afficher :

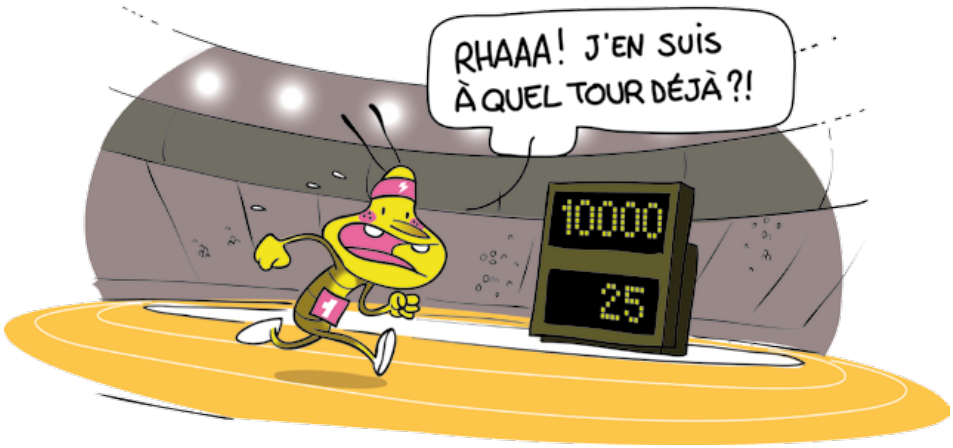
Fichier
```
print(level_1[0])
print(level_1[1])
print(level_1[2])
...
print(level_1[19])
```

Cette fois-ci, nous obtenons un affichage correct. Mais là encore, si le labyrinthe tient sur une grille de 100 x 100, il va falloir écrire cent lignes de code où une seule valeur changera : le numéro de ligne.

4. BOUCLER POUR NE PAS RÉPÉTER LA MÊME CHOSE

L'instruction `for` permet de parcourir tous les éléments d'une liste : elle lit le premier élément et le place dans une variable que l'on peut ensuite utiliser, puis elle lit le deuxième élément qu'elle place dans la variable et ainsi de suite tant qu'il y a des éléments dans la liste.

Il s'agit d'**une boucle**. En athlétisme, un coureur participant au 10000m doit parcourir 25 tours de piste. À chaque tour, il peut indiquer quel est le numéro de son tour : le premier, le deuxième, etc. Au vingt-cinquième tour, il s'arrête car c'est la condition qui marque la fin de la course. Une boucle informatique fonctionne exactement de la même manière.



Revenons au parcours d'une liste en Python. La syntaxe sera la suivante :

Terminal

```
>>> for tour in [1, 2, 3, 4]:
 print(tour)


1
2
3
4
```

`tour` est la variable qui prendra ses valeurs dans la liste suivant le mot-clé `in`. La présence du caractère `:` à la fin de la ligne est très importante. C'est ce caractère qui indique le début d'un bloc : une suite d'instructions qui sont exécutées dans un certain contexte (ici, tant qu'il y a des éléments dans la liste).

Pour bien voir ces instructions, il faut ajouter une indentation – des espaces ou une tabulation – en début de ligne. Si vous oubliez ces espaces, votre code ne fonctionnera pas ou aura une toute autre signification. Le shell Python ajoute automatiquement une indentation après une ligne se terminant par `:`. Pour indiquer que vous avez terminé votre bloc, il faut simplement appuyer sur [Entrée] sur une ligne vierge.

Voyons deux exemples de programmes où seule l'indentation diffère :

Fichier

```
01: for tour in [1, 2, 3, 4]:
02:     print(tour)

03: print("C'est fini !")
```

boucle

L'exécution de ce code provoque l'affichage suivant :

Terminal

```
1
2
3
4
C'est fini !
```

Seule la ligne 2 fait partie de la boucle. Modifions maintenant l'indentation du code :

Fichier

```
01: for tour in [1, 2, 3, 4]:
02:     print(tour)
03:     print("C'est fini !")
```

boucle

Cette fois-ci, le résultat n'est plus très logique :

Terminal

```
1
C'est fini !
2
C'est fini !
3
C'est fini !
4
C'est fini !
```

Les lignes 2 et 3 sont dans la boucle et sont donc exécutées quatre fois !

Pour afficher notre labyrinthe, il faudra donc parcourir la variable `level_1`. Pour conserver le travail effectué, enregistrez sous forme de programme la définition de la variable `level_1` à laquelle vous ajouterez les lignes suivantes :

Fichier

```
for ligne in level_1:
    print(ligne)
```

Vous obtiendrez ainsi l'affichage du labyrinthe de manière correcte :

Terminal

```
+------------------+
|                  |
...
|                  |
+------------------+
```

Pour récapituler :

- Une chaîne de caractères se comporte, en première approximation, comme une liste de caractères ;
- Une liste permet de stocker des éléments qui seront accessibles à partir d'un index ;
- Une variable enregistre en mémoire une information en vue de la ré-utiliser plus tard ;
- L'instruction `print` concatène et traduit automatiquement en chaînes de caractères les éléments qui lui sont transmis sous forme de paramètres ;
- Une boucle répète automatiquement des lignes d'instructions structurées en bloc (décalées par rapport à la ligne débutant le bloc). Cette répétition s'achève par la modification d'une condition telle que « y a-t-il encore des éléments dans la liste ? ».

À retenir

Une boucle est une structure qui parcourt les mêmes instructions jusqu'à ce qu'une condition d'arrêt soit remplie (plus d'éléments dans une liste par exemple).

À retenir

Un bloc est un ensemble de lignes de code qui sont toutes exécutées à une certaine condition. C'est un peu comme si toutes les lignes ne représentaient plus qu'une seule instruction, plus complexe.

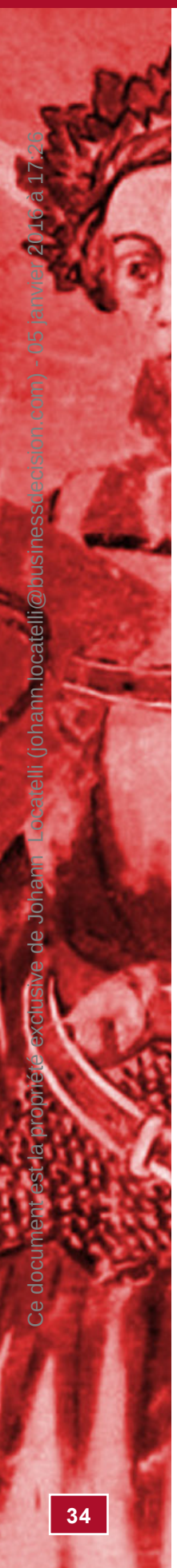
JOUR 2

JOUR 2

SAISIR DES DONNÉES

Le personnage doit pouvoir être déplacé dans le labyrinthe. Pour cela, le joueur appuiera sur des touches qui permettront de commander les déplacements... Mais comment récupérer l'information transmise par l'utilisateur ?

1 Les fonctions
2 Des notes en guise d'aide



L'instruction **print** permet d'afficher des données. La commande que l'on pourrait qualifier de réciproque, permettant d'obtenir des données de l'utilisateur, est l'instruction **input**. Cette commande va afficher un message (que nous définirons), puis attendra que l'utilisateur saisisse des caractères. La fin de la saisie se fera par un appui sur la touche [Entrée].

Nous demanderons à l'utilisateur de choisir son déplacement à l'aide d'une instruction semblable à la suivante (la saisie utilisateur est notée en jaune) :

Terminal

```
>>> input("Quelle direction ? ")
Quelle direction ? Haut
'Haut'
```

Vous pouvez voir qu'après avoir appelé l'instruction, la chaîne de caractères est affichée et le curseur vient se placer après elle, en attente d'une saisie de l'utilisateur. Lors de l'appui sur la touche [Entrée], la saisie est transmise à la machine (qui nous répond ici qu'il s'agit de la chaîne de caractères **'Haut'**). En l'état, nous ne pouvons rien faire de cette information : il faudrait l'enregistrer pour pouvoir y accéder par la suite. Comment ? Grâce aux variables que nous avons vues hier : comme l'instruction **input** nous répond qu'elle a bien lu une chaîne de caractères, nous pouvons stocker cette dernière dans une variable :

Terminal

```
>>> deplacement = input("Quelle direction ? ")
Quelle direction ? Haut
>>> print(deplacement)
Haut
```

Mais que se passe-t-il si nous saisissons un entier ?

Terminal

```
>>> input("Entrez un nombre : ")
Entrez un nombre : 12
'12'
```

L'instruction nous répond qu'elle a lu une chaîne de caractères... Pour pouvoir faire des calculs avec cette valeur, il va falloir la convertir avec une instruction équivalente à **str** (qui traduit un élément en chaîne de caractères) : l'instruction **int**. Cette conversion peut être effectuée de deux manières différentes :

⇨ Soit en stockant la saisie de l'utilisateur dans une variable, puis en la convertissant :

Terminal

```
>>> val = input("Entrez un nombre : ")
Entrez un nombre : 12
>>> val_entier = int(val)
>>> val _entier
12
```

Ici, **val** est une chaîne de caractères alors que **val_entier** est un entier. Nous aurions également pu remplacer le contenu de **val** (chaîne de caractères) par sa conversion en entier en écrivant :

Terminal

```
>>> val = int(val)
```

⇨ Soit en convertissant la saisie de l'utilisateur avant de l'enregistrer dans une variable :

Terminal

```
>>> val = int(input("Entrez un nombre : "))
Entrez un nombre : 12
>>> val
12
```

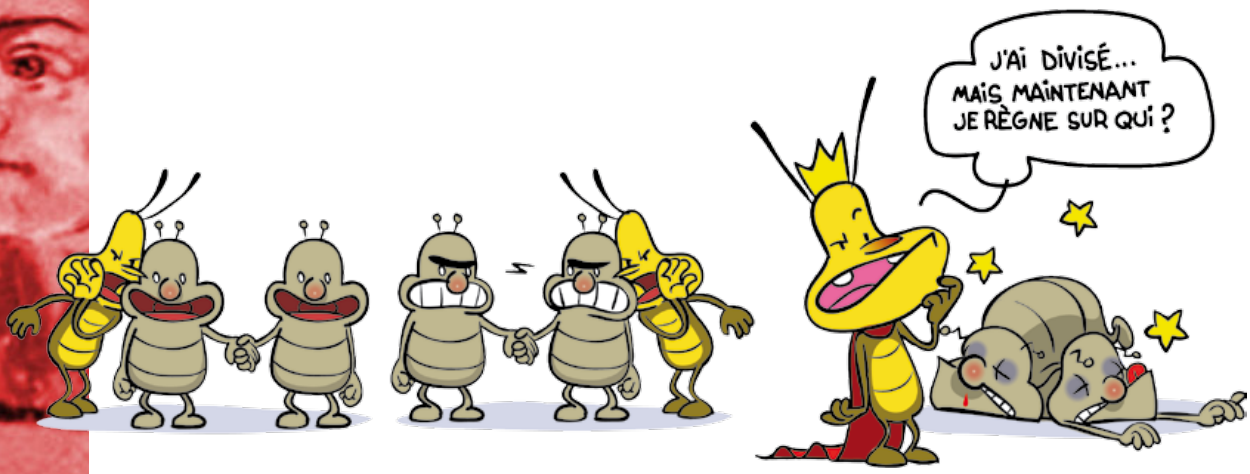
Ici, lorsque l'utilisateur saisit sa réponse, l'instruction **`input("Entrez un nombre : ")`** est remplacée par la saisie, c'est-à-dire la chaîne de caractères **`'12'`**. On applique alors sur cette chaîne la conversion à l'aide de l'instruction **`int`** et la variable **`val`** contient donc un entier.

Nous sommes maintenant capables d'interroger l'utilisateur sur le déplacement qu'il souhaite effectuer. À chaque déplacement, il faudra afficher à nouveau le labyrinthe dans lequel la position du personnage aura été modifiée. Pour pouvoir travailler proprement, il serait intéressant d'être capable de séparer les différentes parties de notre code : affichage du labyrinthe et du personnage, saisie du déplacement, etc. Tout ceci se fait en créant des fonctions.

JOUR 2

1. LES FONCTIONS

Une fonction permet de mettre en place une technique de contrôle ancestrale : « Diviser pour régner ». Le code est écrit par petits blocs, ce qui permet d'isoler correctement les différentes parties et de les modifier simplement.



Une fonction en informatique reprend le principe des fonctions mathématiques : on donne (éventuellement) une ou des valeur(s) en entrée et on obtient une valeur en sortie. Une fonction qui calcule le double d'un nombre donné s'écrira : `f(x) = 2 x`. Si on veut savoir quel est le double de `3`, nous appellerons la fonction `f` en lui indiquant que `x` vaut `3` : `f(3)`. Le résultat, la valeur de retour de la fonction sera `2 x 3` soit `6`. Commençons donc par écrire cette fonction en Python :

Terminal

```
>>> def f(x):
  return 2 * x
```

La partie de définition du nom de la fonction et des paramètres (valeurs) qui lui seront transmis se fait sur la première ligne par `def f(x):` qui correspond à `f(x)=`. Notez que, comme pour les boucles, nous débutons un nouveau bloc : toutes les instructions qui seront indentées (décalées par rapport à la marge de gauche), feront partie de la fonction.

La valeur de retour de la fonction est définie par l'instruction `return` (ici, ce sera `2 * x` où `*` est l'opérateur correspondant à la multiplication... pour ne pas le confondre avec la lettre « x »). Quand nous appellerons cette fonction, la valeur de retour sera calculée, renvoyée à la ligne de code ayant effectué l'appel, puis utilisée. Prenons l'exemple d'un simple affichage :

Terminal

```
>>> print("Résultat : ", f(3))
Résultat : 6
```

Avant que l'instruction `print` puisse être exécutée, elle a besoin de savoir à quoi correspond `f(3)`. La fonction `f` est donc appelée et la valeur de retour est utilisée dans le `print`. C'est comme si nous avions directement écrit :

À retenir

Une fonction est un bloc d'instructions qui s'exécutent en utilisant des valeurs transmises sous forme de paramètres puis renvoient un résultat.

Terminal

```
>>> print("Résultat : ", 6)
Résultat : 6
```

Utilisons maintenant ce mécanisme pour afficher un labyrinthe. Hier, nous avons défini une variable **level_1** contenant uniquement les bordures du labyrinthe (reprenez le code que nous avions enregistré hier). Une fonction permettant d'afficher n'importe quel labyrinthe acceptera en paramètre une variable contenant le labyrinthe et effectuera la boucle permettant de parcourir tous les éléments :

Fichier

```
def affiche_labyrinthe(lab):
    for ligne in lab:
        print(ligne)
```

Boucle parcourant la liste lab passée en paramètre

lab représente une liste. Nous appellerons la fonction en utilisant **level_1** en paramètre et ce sera donc la valeur de **lab** :

Fichier

```
affiche_labyrinthe(level_1)
```

Nous obtiendrons bien l'affichage du labyrinthe. Vous aurez sans doute remarqué que la fonction **affiche_labyrinthe** ne contenait pas d'instruction **return**... Donc, elle ne renvoie pas de valeur ? En fait si ! En Python, toute fonction doit renvoyer une valeur (c'est quand même la définition d'une fonction) ! Si, arrivée à la fin d'une fonction, Python ne rencontre pas d'instruction **return**, il l'ajoutera automatiquement et renverra une valeur spéciale indiquant qu'il n'y a rien à renvoyer. Cette valeur, c'est **None** et c'est un peu comme si Python ajoutait en dernière ligne de la fonction **return None** :

Fichier

```
def affiche_labyrinthe(lab):
    for ligne in lab:
        print(ligne)
    return None
```

Valeur None

Si nous utilisons une variable pour stocker la valeur de retour de **affiche_labyrinthe**, nous obtiendrons l'affichage du labyrinthe, puis notre variable contiendra la valeur **None** :

Fichier

```
val_retour = affiche_labyrinthe(level_1)
```

S'il ne s'agit que de dessiner les contours d'un labyrinthe, nous pouvons exploiter pleinement la puissance des fonctions en paramétrant la taille du labyrinthe :

Fichier

```
def affiche_bordures_labyrinthe(taille):
    print("+{}+".format("-" * (taille - 2)))
    for i in range(taille - 2):
        print("|{}|".format(" " * (taille - 2)))
    print("+{}+".format("-" * (taille - 2)))
```

Utilisation de l'affichage formaté et de la multiplication de chaînes de caractères

Répétition de l'affichage de la ligne (taille - 2 fois)

Ici, j'ai créé une fonction à laquelle on peut transmettre une valeur, le paramètre **taille**. Cette valeur est utilisée ensuite pour déterminer le nombre de caractères à afficher sur chaque ligne.

Dans la boucle, une nouvelle instruction est apparue : la fonction **range**. Lorsque nous avons abordé les boucles, nous avions vu que l'instruction **for** permet de parcourir l'ensemble des éléments d'une liste. Mais ici, nous n'avons pas une liste, nous avons l'instruction **range(taille – 2)**... Que fait donc cette instruction ? Dans le shell Python, si nous tapons la ligne suivante nous obtiendrons un début de réponse :

Terminal

```
>>> range(10)
range(0, 10)
```

La réponse de Python à notre instruction est simplement une autre instruction... Il ne s'agit donc pas d'une liste. Essayons cette instruction au sein d'une boucle **for** pour voir si cela marche et ce que l'on peut afficher :

Terminal

```
>>> for i in range(10):
 print(i)

0
1
...
8
9
```

La variable **i** prend des valeurs allant de 0 jusqu'à 9. C'est donc un peu comme si la fonction **range(10)** nous avait renvoyé la liste **[0, 1, 2, 3, ..., 9]**.
D'ailleurs, vous pouvez le vérifier en convertissant **range(10)** en liste :

Terminal

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

range(10) ne renvoie pas directement une liste pour des raisons d'optimisation de la place occupée en mémoire (si vous créez une liste d'un million d'éléments seulement pour répéter un million de fois un bloc vous utilisez beaucoup de mémoire pour rien !).
La fonction **range** n'admet pas forcément qu'un seul paramètre, on peut définir une plage de nombres depuis un nombre de départ jusqu'à un nombre d'arrivée, en sautant éventuellement des nombres :

- ⇨ **range(fin)** : compte de **0** à **fin - 1** ;
- ⇨ **range(debut, fin)** : compte de **debut** à **fin - 1** ;
- ⇨ **range(debut, fin, saut)** : compte de **debut** à **fin - 1** en sautant entre chaque nombre **saut** nombres. Voici un exemple permettant d'obtenir la liste des entiers entre 0 et 10, en comptant de 2 en 2 :

Terminal

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Si nous reprenons notre programme depuis le début, en y ajoutant une fonction permettant de récupérer le choix de déplacement d'un joueur, voici ce que nous obtenons :

Fichier

```
01: level_1 = [
02:   "+------------------+",
03:   "|                  |",
```

Fichier

```
...
20:    "|                   |",
21:    "+-------------------+"
22: ]
23:
24: def affiche_labyrinthe(lab):
25:     for ligne in lab:
26:         print(ligne)
27:
28: def affiche_bordures_labyrinthe(taille):
29:     print("+{}+".format("-" * (taille - 2)))
30:     for i in range(taille - 2):
31:         print("|{}|".format(" " * (taille - 2)))
32:     print("+{}+".format("-" * (taille - 2)))
33:
34: def choix_joueur():
35:     return input("Votre déplacement (Haut/Bas/Droite/Gauche) ? ")
36:
37: affiche_bordures_labyrinthe(20)
38: dep = choix_joueur()
39: print(dep)
```

Notre programme commence à être conséquent avec plus de quarante lignes et il est bien sûr hors de question d'utiliser le shell Python qui ne nous permettrait pas de modifier proprement notre code.

Mais nous avons déjà un peu de mal à nous relire. Il faudrait que nous puissions ajouter des notes qui nous permettraient de comprendre rapidement à quoi correspond chaque fonction ou chaque partie du programme.

2. DES NOTES EN GUISE D'AIDE

Il est possible d'ajouter des lignes dans le programme qui ne seront pas interprétées, des sortes de *post-it* invisibles. Un peu comme si l'ordinateur ne pouvait pas les voir mais le développeur si. Pour cela, il existe plusieurs techniques.

Tout d'abord, le commentaire dit « en ligne » : tout ce qui suit le caractère **#** sera considéré comme un commentaire et ne sera pas exécuté :

Terminal

```
>>> print("Coucou") # Ceci ne sera pas lu
Coucou
>>> # print("Coucou")
>>>
```

Ce type de commentaires permet d'indiquer ce que fait une ligne. Il peut également être utilisé pour segmenter le code en grande parties. Reprenons le début de notre programme de jeu :

Fichier

```
01: ##########
02: # Définition des différents niveaux
03: ## Niveau 1
04: level_1 = [
```

Commentaires

Fichier

```
05:   "+------------------+",
06:   "|                  |",
...
23:   "|                  |",
24:   "+------------------+"
25: ]
```

La coloration syntaxique va mettre en exergue les lignes 1 à 3 et nous permettra de naviguer plus simplement dans le code.

Une autre méthode permettant d'ajouter des commentaires est l'utilisation des chaînes de caractères. Hier, nous avions vu qu'une valeur non affectée à une variable ne produisait aucune action particulière en dehors du shell Python. Nous allons utiliser ce mécanisme pour commenter le code. Nous pourrions ainsi remplacer les commentaires précédents par :

Fichier

```
01: "**********"
02: "Définition des différents niveaux"
03: "Niveau 1"
```

Ceci est valable d'un point de vue syntaxique, vous n'obtiendrez pas de message d'erreur... Par contre, ce n'est pas la technique préconisée en Python. Dans ce cas, on préférera utiliser le commentaire en ligne. Les commentaires écrits à l'aide de chaînes de caractères sont appelés des **docstrings** et, positionnés après la première ligne de définition d'une fonction, ils peuvent être lus par une instruction d'aide qui les affichera avec un formatage spécial. On utilisera les chaînes de caractères multilignes pour obtenir un texte plus lisible :

Fichier

```
29: def affiche_labyrinthe(lab):
30:     """
31:         Affichage d'un labyrinthe
32:
33:         lab : Variable contenant le labyrinthe
34:     """
35:     for ligne in lab:
36:         print(ligne)
```

Explication concise de ce que fait la fonction et quels sont les paramètres attendus

Lorsque des fonctions sont commentées de cette sorte, l'instruction **help** permet d'afficher le commentaire lorsque l'on se trouve dans le shell Python. Voici un petit exemple effectué justement dans le shell Python :

Terminal

```
>>> def f(x):
 """
  Calcule le double de la valeur passée en paramètre

  x : Valeur dont on souhaite calculer le double
 """
 return 2 * x

>>> help(f)
Help on function f in module __main__:

f(x)
    Calcule le double de la valeur passée en paramètre

    x : Valeur dont on souhaite calculer le double
```

Pour être utile, un commentaire doit être pertinent et concis : commentez vos lignes de code… mais pas trop ! Le mieux est l'ennemi du bien. À vouloir donner trop d'informations vous perdrez le lecteur qui n'y trouvera pas les explications qu'il recherche. De plus, certaines lignes de code parlent d'elles-mêmes : l'instruction `print` affiche un message, inutile donc d'ajouter un message pseudo informatif du style « Affichage du message d'erreur ». Par contre, si ce même `print` constitue la seule ligne d'une fonction, il faudra quand même commenter la fonction en indiquant qu'elle affiche un message…

Aujourd'hui, nous n'avons pas énormément avancé sur le jeu, nous avons surtout appris à structurer le code de manière à ce que nous puissions développer par la suite notre programme le plus rapidement et surtout, avec le moins d'erreurs possible. ▮

Pour récapituler :

- Une **fonction** est un bloc de code qui s'exécute lorsque l'on fait appel à lui par son nom en lui fournissant éventuellement des paramètres qui modifieront son comportement ;
- Un **commentaire** est une chaîne de caractères qui ne représente aucune instruction et qui ne sera pas interprétée lors de l'exécution du programme. Les commentaires ne sont utiles que pour les développeurs, les utilisateurs n'y ayant pas accès.

À savoir

Un commentaire est un court texte indiquant au développeur ce que fait une partie précise du code. Les commentaires sont utiles pour vous rappeler pourquoi vous avez choisi d'écrire une partie du programme de telle ou telle façon et pour aider éventuellement d'autres personnes à comprendre plus rapidement votre code. Sur un programme un peu complexe, si vous délaissez votre code une ou deux semaines, il y a fort à parier que sans commentaires vous ne puissiez vous remettre au travail sans passer du temps à analyser et comprendre ce que vous aviez écrit précédemment.

JOUR 3

JOUR 3

LES TESTS DE DÉPLACEMENT

Une fois que l'utilisateur a saisi un déplacement, il faut le répercuter à l'écran. Et que se passe-t-il si le joueur désire déplacer son personnage dans un mur ? Il va falloir que nous mettions en place des tests pour nous assurer que le déplacement est autorisé.

1. Un peu de logique booléenne pour pouvoir tester des valeurs
2. Tester des valeurs et effectuer des actions en fonction du résultat obtenu
3. Découpage en tranches
4. Une première version de notre jeu

JOUR 3

Hier, nous sommes parvenus à afficher le labyrinthe et à demander à l'utilisateur de saisir le déplacement qu'il souhaite appliquer à son personnage... Mais nous n'avons pas affiché le personnage dans le labyrinthe ! En effet, pour cela il nous manquait des connaissances : la possibilité d'effectuer un test entre deux valeurs.

Pour positionner notre personnage dans le labyrinthe, nous allons numéroter chaque case en fonction de sa colonne et de sa ligne. La première case qui se trouve en haut et à gauche (celle qui correspond à un caractère **+** dans **`level_1`**) se verra donc affecter la valeur **`(0, 0)`**. La case suivante, à droite de **`(0, 0)`**, se trouve sur la colonne suivante de la même ligne, donc sur **`(1, 0)`**. Donc, pour savoir où se trouve le personnage du joueur, nous aurons besoin d'un couple de données (colonne, ligne). Pour stocker ces valeurs qui évolueront au cours de la partie, nous allons utiliser une liste.

Au départ, le personnage se trouvera sur la première case libre du labyrinthe, c'est-à-dire la case **`(1, 1)`** :

Fichier

```
perso = "X"
pos_perso = [1, 1]
```

Pour savoir sur quelle colonne se trouve le personnage, il faudra consulter **`pos_perso[0]`** et pour connaître la ligne se sera **`pos_perso[1]`**.

Comme le labyrinthe est affiché ligne par ligne, il va nous falloir commencer par détecter si nous nous trouvons sur la ligne sur laquelle le personnage est positionné. Pour rappel, la fonction permettant d'afficher le labyrinthe est la suivante :

Fichier

```
def affiche_labyrinthe(lab):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
    """
    for ligne in lab:
        print(ligne)
```

Si nous voulons savoir sur quelle ligne nous nous trouvons, il faut ajouter **un compteur**. Il s'agira d'une simple variable qui aura pour valeur de départ **`0`** et à laquelle nous ajouterons une unité à chaque passage dans la boucle :

Fichier

```
def affiche_labyrinthe(lab):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
    """
    n_ligne = 0
    for ligne in lab:
        print(ligne)
        n_ligne = n_ligne + 1
```

Mise en place du compteur

Il faut maintenant être capable de comparer le numéro de ligne courant contenu dans la variable **`n_ligne`** avec le numéro de ligne du personnage **`pos_perso[1]`**. Pour cela, nous allons avoir besoin des tests qui font intervenir la logique booléenne.

1. UN PEU DE LOGIQUE BOOLÉENNE POUR POUVOIR TESTER DES VALEURS

En informatique, tous les mécanismes de tests sont basés sur la logique booléenne : une expression est soit vraie (`True`) soit fausse (`False`). Les opérateurs de comparaison permettent de comparer deux valeurs et d'indiquer si la condition énoncée est vérifiée (`a` est plus grand que `b`, `a` et `b` ont la même valeur, etc.). Faisons un essai dans le shell Python :

Terminal

```
>>> a = 1
>>> b = 2
>>> a < b
True
>>> a > b
False
```

Il existe sept opérateurs de comparaison :

- ⇨ `>` : supérieur à ;
- ⇨ `<` : inférieur à ;
- ⇨ `>=` : supérieur ou égal à ;
- ⇨ `<=` : inférieur ou égal à ;
- ⇨ `!=` : différent de ;
- ⇨ `==` : égal à ;
- ⇨ `is` : identique à.

Comme vous pouvez le constater, l'égalité se teste à l'aide d'un opérateur contenant deux fois le caractère `=`. Pourquoi cette redondance ? Tout simplement pour ne pas confondre un test avec une affectation de variable. Imaginons que nous reprenions notre variable `a` et que nous voulions tester si la valeur qu'elle contient est bien `5` :

Terminal

```
>>> a == 5
False
>>> print(a)
1
```

La ligne `a == 5` peut se lire « Est-ce que `a` a pour valeur 5 ? ». La réponse est bien sûr non. Par contre, si nous nous trompons et que nous oublions un signe égal :

Terminal

```
>>> a = 5
>>> print(a)
5
```

Nous avons modifié la valeur de `a` ! Il faut donc toujours être très prudent et penser à bien mettre l'opérateur de comparaison en doublant le signe `=` lors d'une comparaison.

Vous vous demandez sans doute pourquoi s'il existe un opérateur d'égalité, un opérateur d'identité a été ajouté... Je ne répondrai que brièvement pour votre culture générale, cette notion n'étant pas indispensable pour créer notre jeu. L'identité est plus forte que l'égalité. Certaines valeurs peuvent être considérées comme équivalentes à d'autres. Par exemple, `1` est équivalent à `True` et `0` est équivalent à `False` dans un contexte de test. Prenons le cas de la valeur `0` :

Terminal

```
>>> 0 == False
True
```

Par contre, l'identité va chercher si, en plus, les deux valeurs comparées sont de même type. Or, ici, nous avons un entier et un booléen :

Terminal

```
>>> 0 is False
False
```

Si vous souhaitez effectuer des tests un peu plus complexes, vous aurez besoin de « combiner » des tests. Cette succession de tests se fait à l'aide de trois opérateurs booléens :

- ⇨ `and` : deux conditions doivent être vérifiées. Par exemple, `a == 1 and b < 3` signifie que pour que l'expression soit vraie, `a` doit valoir `1` et `b` doit contenir un nombre inférieur à `3` ;
- ⇨ `or` : au moins une des deux conditions doit être vérifiée. Par exemple, `a == 1 or b < 3` signifie que pour que l'expression soit vraie, soit `a` doit valoir `1`, soit `b` doit contenir un nombre inférieur à `3`, soit `a` vaut `1` et `b` est inférieur à `3` ;
- ⇨ `not` : la condition ne doit pas être vérifiée ; si elle est fausse, on la considère comme vraie et réciproquement.

De ces opérateurs on tire les tables de vérités, tableaux indiquant la valeur de retour des tests auxquels on applique des opérateurs booléens. Ici, `1` représente `True` et `0` représente `False`.

and

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

or

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

not

A	not A
0	1
1	0

Nous savons tester des conditions, mais il faut également être capable d'exécuter certaines instructions si ces conditions sont vraies et d'autres si elles sont fausses.

À retenir

Une valeur booléenne est une valeur qui ne peut être que vraie ou fausse (`True` ou `False`). Tous les opérateurs de test retournent une valeur booléenne.

2. TESTER DES VALEURS ET EFFECTUER DES ACTIONS EN FONCTION DU RÉSULTAT OBTENU

Pour créer un « aiguillage » permettant d'exécuter un bloc d'instructions **A** si une condition est respectée et un bloc d'instructions **B** sinon, nous allons utiliser la commande **if**. Si la condition qui suit cette instruction est vraie, alors le bloc suivant est exécuté :

Terminal

```
>>> a = 1
>>> if a == 1:
 print("La valeur est bien 1")

La valeur est bien 1
```

La ligne de test peut se lire : « Si **a** est égal à **1** alors exécute le bloc ». Si la variable **a** est différente de **1**, il ne se passera rien. On peut ajouter un traitement pour ce cas grâce à l'instruction **else** :

Terminal

```
>>> if a == 2:
 print("La valeur est bien 2")
else:
 print("Valeur différente de 2 :", a)

Valeur différente de 2 : 1
```

Ici, **a** est différent de **2** donc c'est le deuxième bloc qui est affiché. On peut lire ce code de la manière suivante : « Si **a** est égal à **2** alors exécute le premier bloc, sinon exécute le second bloc ».

Notez l'indentation des lignes (décalage par rapport à la marge) : l'instruction **else** est collée contre la marge et ne se trouve pas dans l'alignement du **if**. Ceci est dû au prompt **>>>** du shell Python. Dans un programme, nous aurions effectué un alignement parfait :

Fichier

```
if a == 2:
    print("La valeur est bien 2")
else:
    print("Valeur différente de 2 :", a)
```

Nous sommes maintenant capables de détecter la ligne sur laquelle afficher le personnage dans notre labyrinthe. Pour rappel, la position du personnage est stockée dans la liste **pos_perso**. **pos_perso[0]** donne le numéro de colonne et **pos_perso[1]** donne le numéro de ligne :

Fichier

```
def affiche_labyrinthe(lab, pos_perso):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
    """
    n_ligne = 0
    for ligne in lab:
        if pos_perso[1] == n_ligne :
            print(ligne, "<- ligne du personnage")
        else :
            print(ligne)
        n_ligne = n_ligne + 1
```

Nous avons besoin d'un paramètre supplémentaire pos_perso qui contiendra la position du personnage sous forme de liste

Ne s'exécute que si le numéro de ligne n_ligne est le même que celui du personnage pos_perso[1]

S'exécute lorsque n_ligne est différent de pos_perso[1]

S'exécute toujours (une seule indentation, donc appartient au bloc de la boucle et non au bloc du if)

À l'exécution, nous obtenons quelque chose comme :

Terminal
```
+------------------+
|                  | <- ligne du personnage
...
|                  |
+------------------+
```

Pour pouvoir vraiment afficher le personnage, il faut lire la ligne caractère à caractère de manière à remplacer le caractère en position **`pos_perso[0]`** (la colonne où se trouve le personnage) par le caractère représentant le personnage. Il ne s'agit en fait que d'une boucle sur l'ensemble des caractères de la chaîne. Pour pouvoir la réaliser, il faut seulement savoir que l'instruction **`len`** nous donne le nombre de caractères que contient une chaîne :

Terminal
```
>>> chaine = "|                  |"
>>> perso = "X"
>>> pos_perso = [1, 1]
>>> for i in range(len(chaine)):
 if i == pos_perso[0]:
  print(perso, end = "")
 else:
  print(chaine[i], end = "")

|X                 |
```

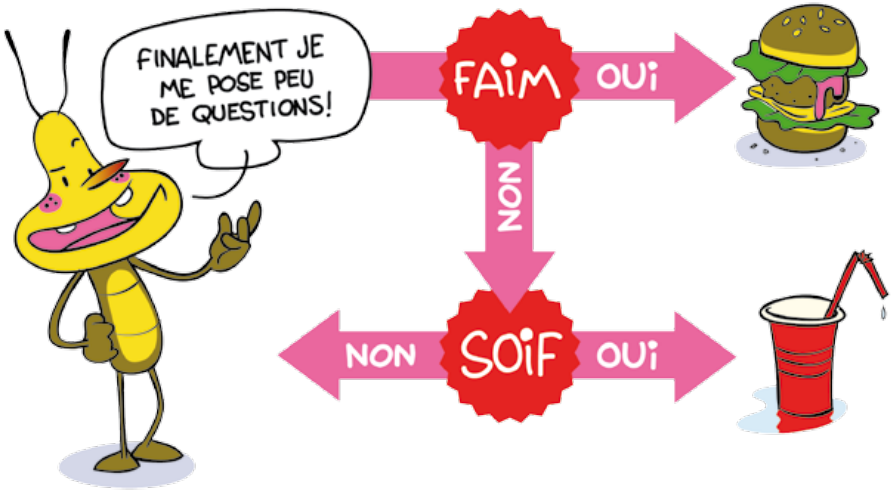
L'instruction la plus difficile à lire ici est **`range(len(chaine))`**. Pour mieux la comprendre il faut la décortiquer étape par étape. Tout d'abord, **`len(chaine)`** nous donne la taille de la chaîne soit **`20`**. L'instruction globale est donc équivalente à **`range(20)`** et nous savons que cette commande produit une sorte de liste allant de **`0`** à **`19`**. Donc, nous passerons vingt fois dans la boucle où la variable **`i`** variera de **`0`** à **`19`**.

Si **`i`** correspond à la position du personnage nous l'affichons et sinon, nous affichons le ième caractère de la chaîne de caractères, c'est-à-dire **`chaine[i]`**. Notez l'utilisation du paramètre **`end`** dans l'instruction **`print`** pour ne pas avoir de retour à la ligne intempestif (sinon nous afficherions notre ligne sous la forme d'une colonne...).

Nous pourrions incorporer ce morceau de code dans notre fonction **`affiche_labyrinthe`**, mais il existe une manière plus simple d'afficher le personnage.

À retenir

Un test conditionnel est une structure permettant de déterminer quel bloc de code exécuter en fonction de la valeur de retour d'une expression booléenne (`True` ou `False`) éventuellement fournie par le résultat de l'évaluation d'une condition.

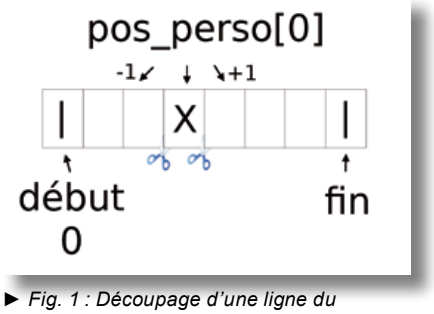


3. DÉCOUPAGE EN TRANCHES

Si nous pouvions découper une chaîne de caractères en « tranches », comme du saucisson, comment ferions-nous pour y placer notre personnage ? Il faudrait conserver tous les caractères avant la position `pos_perso[0]`, insérer le caractère du personnage et afficher tous les caractères de la ligne du labyrinthe se trouvant après la position `pos_perso[0]` (voir figure 1). En Python, un mécanisme appelé « découpage en tranches » – ou **slicing** – permet de sélectionner des portions à l'intérieur d'une chaîne de caractères.

Pour cela, au lieu d'indiquer entre crochets seulement l'indice du caractère que l'on souhaite récupérer, nous indiquerons la position du caractère de départ, celle du caractère de fin et éventuellement un pas (en fait, exactement comme les paramètres de la fonction `range`, mais au lieu de séparer les paramètres par des virgules, nous utiliserons le caractère `:`). Tous ces paramètres ne sont pas obligatoires, on peut indiquer :

- Seulement la position du caractère de départ : la portion se terminera alors à la fin de la chaîne de caractères initiale ;
- La position du caractère de départ et de fin : la portion contiendra donc tous les caractères désignés ;
- La position du caractère de départ, de fin et un pas : la portion contiendra alors les caractères de la chaîne initiale obtenus en la parcourant par sauts d'un nombre de caractères indiqués par le pas.



► *Fig. 1 : Découpage d'une ligne du labyrinthe en deux parties et insertion du personnage.*

Prenons quelques exemples :

Terminal
```
>>> chaine = "ABCDEFGHIJKLM"
>>> print("Taille de la chaîne :", len(chaine))
Taille de la chaîne : 13
>>> chaine[0:5]
'ABCDE'
```

Ici, nous sélectionnons la portion de chaîne commençant en position `0` jusqu'à la position `5` non comprise.

Terminal
```
>>> chaine[0:5:2]
'ACE'
```

En prenant la même chaîne que précédemment mais en modifiant le pas, nous ne récupérons plus qu'une lettre sur deux.

Terminal
```
>>> chaine[:5]
'ABCDE'
```

Si nous omettons le premier paramètre, la valeur par défaut `0` est utilisée. L'écriture précédente est donc équivalente à `chaine[0:5]`. Pour être plus précis, on utilise également ici le pas par défaut qui a pour valeur `1`. L'écriture équivalente complète de notre instruction est donc `chaine[0:5:1]`.

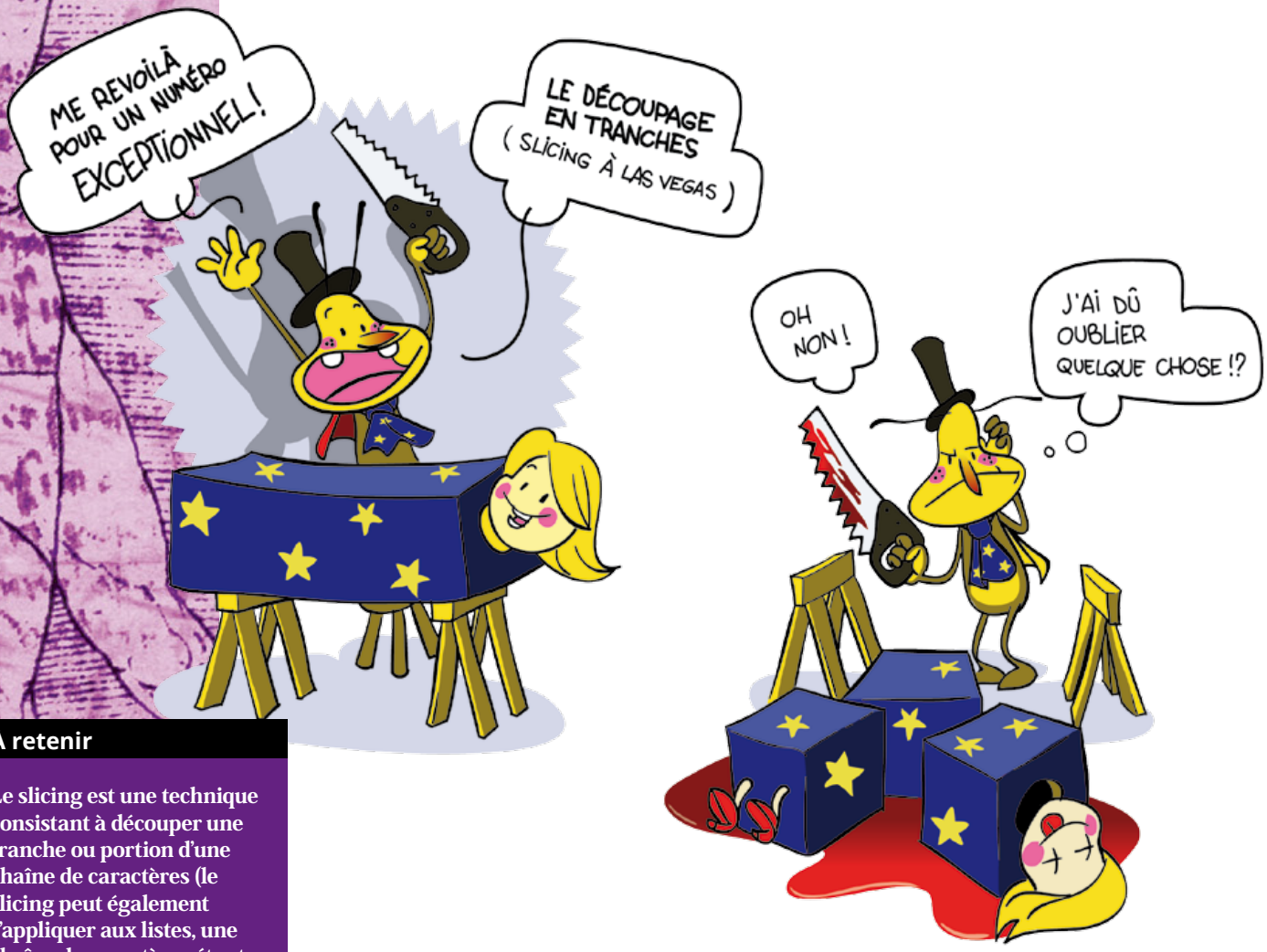
Terminal
```
>>> chaine[6:]
'GHIJKLM'
```

Cette fois, c'est la position finale qui est la valeur par défaut. Notez qu'une autre écriture permet d'accéder au dernier caractère avec `chaine[6:len(chaine)]` (`len(chaine)` renvoie le nombre total de caractères, soit la position du dernier caractère + 1). L'instruction `chaine[6:]` est équivalente à `chaine[6::1]` puisque le pas de `1` est la valeur par défaut.

Terminal

```
>>> chaine[::2]
'ACEGIKM'
```

Pour finir, nous récupérons une lettre sur deux sur l'ensemble de la chaîne (les valeurs par défaut de début et de fin de chaîne nous amènent à considérer la chaîne dans son intégralité).



À retenir

Le slicing est une technique consistant à découper une tranche ou portion d'une chaîne de caractères (le slicing peut également s'appliquer aux listes, une chaîne de caractères étant une liste particulière). Pour déterminer une tranche, il faut indiquer la position du premier caractère, du dernier caractère (non compris) et du pas.

Si nous adaptons le slicing sur notre labyrinthe pour placer le personnage, en fonction du numéro de colonne où se trouve le personnage, la ligne commencera par son contenu normal jusqu'à la colonne du personnage (`ligne[0:pos_perso[0]]`), nous ajouterons ensuite le personnage (`perso`), puis les caractères suivants du labyrinthe (`ligne[pos_perso[0] + 1:]`).

4. UNE PREMIÈRE VERSION DE NOTRE JEU

Avec toutes les connaissances que nous avons acquises jusqu'à maintenant, nous pouvons produire une première version de notre jeu où il sera possible de voir le personnage se déplacer en fonction des choix du joueur. Pour cela, nous allons un peu modifier la fonction de saisie de l'action du joueur :

Fichier

```
001: ##########
002: # Définition des différents niveaux
003: ## Niveau 1
004: level_1 = [
005:   "+------------------+",
006:   "|                  |",
...
023:   "|                  |",
024:   "+------------------+"
025: ]
026:
027:
028:
029: def affiche_labyrinthe(lab, perso, pos_perso):
030:     """
031:         Affichage d'un labyrinthe.
032:
033:         lab : Variable contenant le labyrinthe
034:         perso : caractère représentant le personnage
035:         pos_perso : liste contenant la position du personnage [ligne, colonne]
036:
037:         Pas de valeur de retour
038:     """
039:     n_ligne = 0
040:     for ligne in lab:
041:         if n_ligne == pos_perso[1]:
042:           print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0] + 1:])
043:         else:
044:             print(ligne)
045:         n_ligne += 1
```

Jusqu'ici rien de nouveau, j'ai seulement utilisé le slicing en ligne 42 pour placer le personnage dans le labyrinthe en fonction de la position indiquée par la liste **`pos_perso`** qui est passée en paramètre de la fonction en même temps que le labyrinthe et le caractère représentant le personnage.

Nous allons modifier notre fonction de saisie des déplacements de manière à ce qu'elle interroge une autre fonction pour savoir si le déplacement est valide et éviter que le personnage ne rentre dans un mur par exemple). Nous appellerons cette fonction **`verification_deplacement`** :

Fichier

```
048: def verification_deplacement(lab, pos_col, pos_ligne):
049:     """
050:         Indique si le déplacement du personnage est autorisé ou pas.
051:
052:         lab : Labyrinthe
```

Fichier

```
053:         pos_ligne : position du personnage sur les lignes
054:         pos_col : position du personnage sur les colonnes
055:
056:         Valeurs de retour :
057:           None : déplacement interdit
058:           [col, ligne] : déplacement autorisé sur la case indiquée par la liste
059:     """
060:     # Calcul de la taille du labyrinthe
061:     n_cols   = len(lab[0])
062:     n_lignes = len(lab)
063:     # Teste si le déplacement conduit le personnage en dehors de l'aire
064:     # de jeu
065:     if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
066:        pos_col > (n_cols - 1):
```

Le symbole \ permet d'indiquer que nous n'avons pas terminé notre ligne. Nous avons ici une illustration de condition longue composée à l'aide de trois opérateurs booléens

```
067:         return None
068:     elif lab[pos_ligne][pos_col] != " ":
069:         return None
070:     else:
071:         return [pos_col, pos_ligne]
```

L'instruction elif est une contraction de else if (sinon, si ...)

Si le déplacement n'est pas autorisé (en dehors de l'aire de jeu – lignes 65 à 67 – ou sur un mur – lignes 68 et 69), la fonction renvoie la valeur spéciale **None** (rien). Sinon, elle renvoie une liste contenant les nouvelles coordonnées du personnage dans le labyrinthe.

La fonction qui permet au joueur de saisir son choix de déplacement va maintenant faire appel à la fonction **verification_deplacement** pour s'assurer que le déplacement est autorisé. Un choix supplémentaire sera proposé : « Quitter » pour arrêter le programme et sortir du jeu.

Fichier

```
074: def choix_joueur(lab, pos_perso):
075:     """
076:         Demande au joueur de saisir son déplacement et vérifie s'il est
077:         possible. Si ce n'est pas le cas affiche un message, sinon
078:         modifie la position du perso dans la liste pos_perso
079:
080:         lab: Labyrinthe
081:         pos_perso : liste contenant la position du personnage [colonne, ligne]
082:
083:         Pas de valeur de retour
084:     """
085:     choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
086:     if choix == "H" or choix == "Haut":
087:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1)
088:     elif choix == "B" or choix == "Bas":
089:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] + 1)
090:     elif choix == "G" or choix == "Gauche":
091:         dep = verification_deplacement(lab, pos_perso[0] - 1, pos_perso[1])
092:     elif choix == "D" or choix == "Droite":
093:         dep = verification_deplacement(lab, pos_perso[0] + 1, pos_perso[1])
094:     elif choix == "Q" or choix == "Quitter":
```

Fichier

```
095:         exit(0)
096:     if dep == None:
097:         print("Déplacement impossible")
098:     else:
099:         pos_perso[0] = dep[0]
100:         pos_perso[1] = dep[1]
```

La fonction exit permet d'interrompre l'exécution d'un programme. La valeur passée en paramètre doit être un entier compris entre 0 et 255 : 0 indique que le programme s'est terminé normalement et une valeur supérieure et considérée comme un code représentant une erreur

Modification du contenu de la liste pos_perso

Ici, quelques explications supplémentaires sont nécessaires à propos du paramètre **pos_perso**. En effet, la variable qui est transmise à la fonction **choix_joueur** va voir sa valeur modifiée, or ce n'est pas le comportement normal comme le montre l'exemple suivant :

Terminal

```
>>> a = 1
>>> def modifier(param):
 param = param + 1
 print("Nouvelle valeur :", param)


>>> print(a)
1
>>> modifier(a)
Nouvelle valeur : 2
>>> print(a)
1
```

Comme vous le voyez, la valeur de la variable **a** est transmise à la fonction **modifier** et copiée dans la variable **param**. Nous avons donc deux variables distinctes. On parle de passage de paramètres par valeur. En Python, ce mécanisme s'applique à tous les types de base : entier, réel, booléen, etc.

En ce qui concerne les listes, le passage de paramètres se fait par adresse : il s'agit du même emplacement mémoire qui est utilisé, il n'y a donc pas copie de la valeur. C'est comme si l'on travaillait sur la même variable :

Terminal

```
>>> l = [1, 2, 3]
>>> def modifier(liste):
 liste[0] = 0
 print("Nouvelle liste :", liste)


>>> print(l)
[1, 2, 3]
>>> modifier(l)
Nouvelle liste : [0, 2, 3]
>>> print(l)
[0, 2, 3]
```

Tout ceci nous amène à aborder la visibilité des variables : une variable **locale** est une variable définie dans un bloc et détruite en sortie de bloc.

Terminal

```
>>> def ma_fonction():
 variable = 2
 print(variable)


>>> ma_fonction()
2
>>> print(variable)
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    print(variable)
NameError: name 'variable' is not defined
```

Ici, **variable** n'existe que dans le bloc de définition de la fonction **ma_fonction**.

Réciproquement, une variable **globale** est une variable définie dans le bloc principal du programme (contre la marge de gauche) et qui est visible dans n'importe quel bloc :

Terminal

```
>>> ma_variable = "Coucou"
>>> def ma_fonction():
 print(ma_variable)


>>> ma_fonction()
Coucou
```

Comme vous le voyez, la variable **ma_variable** n'est pas passée en paramètre de **ma_fonction**... et pourtant elle est accessible. Il est préférable d'éviter d'utiliser les variables globales pour ne pas subir des effets de bord (variables modifiées on ne sait où dans le programme). Ces erreurs sont très difficiles à retrouver et à corriger et donc particulièrement chronophages.

Pour revenir à notre programme, il nous manque une fonction qui va gérer le jeu : afficher le labyrinthe, demander au joueur où il souhaite déplacer son personnage et recommencer.

Fichier

```
103: def jeu(level, perso, pos_perso):
104:     """
105:         Boucle principale du jeu. Affiche le labyrinthe dans ses différents
106:         états après les déplacements du joueur.
107:
108:         level : Labyrinthe
109:         perso : caractère représentatnt le personnage
110:         pos_perso : liste contenant la position du personnage [colonne, ligne]
111:     """
112:     while True:
113:         affiche_labyrinthe(level, perso, pos_perso)
114:         choix_joueur(level, pos_perso)
```

J'ai utilisé une nouvelle forme de boucle : la boucle **while** qui signifie « tant que ». Tant que la condition est vraie, nous exécutons le code du bloc qui est associé à la boucle.

À retenir

Une variable locale est une variable qui n'existe qu'à l'intérieur d'un bloc.

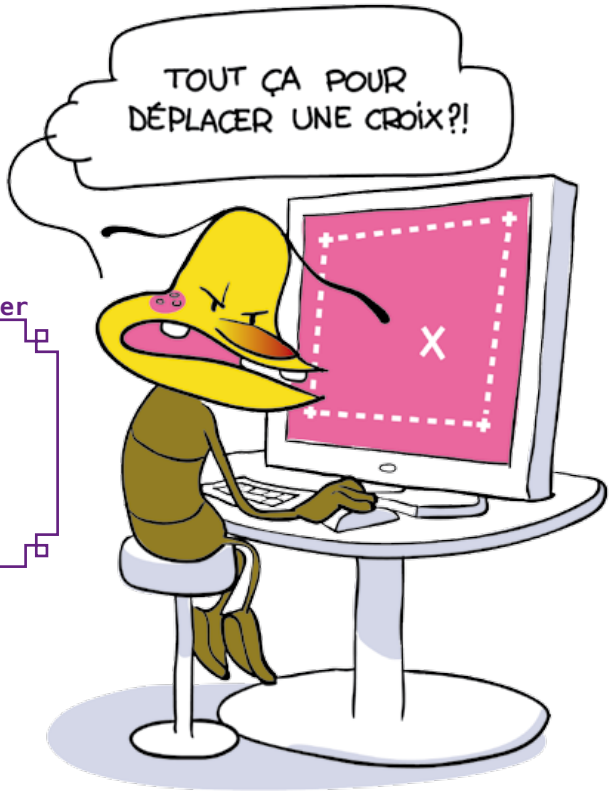
Une variable globale est une variable qui est accessible depuis n'importe quel bloc. Les variables globales sont dangereuses, car on ne maîtrise pas le moment où elles sont modifiées (une faute de frappe par exemple peut conduire à modifier une variable globale au lieu de produire un message d'erreur... Et c'est beaucoup plus difficile à déboguer !).

À savoir

La boucle `while` est une boucle qui s'exécute tant que la condition qui lui est transmise n'est pas fausse. Il faut donc s'assurer que cette condition évolue au fil du temps pour que la boucle ne soit pas infinie.

Ici, la boucle est infinie puisque **`True`** ne changera jamais de valeur. La seule façon de quitter le programme est de taper « Quitter » lors du choix de déplacement.

Il ne nous reste plus qu'à utiliser les fonctions que nous avons définies au sein d'un programme. C'est ce que font les dernières lignes du fichier :



Fichier

```
117: ##########
118: # Programme principal
119: # Initialisation du personnage
120: perso = "X"
121: pos_perso = [1, 1]
122: # Lancement de la partie
123: jeu(level_1, perso, pos_perso)
```

Vous pouvez maintenant « jouer » à déplacer le personnage dans le labyrinthe. ▉

Pour récapituler :

- Une **condition** est une expression dont l'évaluation donne une valeur booléenne qui peut valoir **`True`** ou **`False`** ;
- Pour effectuer un branchement logique en fonction d'une condition, on utilise l'instruction **`if`** suivie éventuellement d'un traitement **`else`**. Pour enchaîner les tests, on peut utiliser **`elif`** qui est une forme contractée de **`else if`** ;
- Le **slicing** permet de découper une portion d'une chaîne de caractères en spécifiant son début, sa fin et le pas de déplacement ;
- Les listes sont passées par adresse dans les fonctions : une modification de leur valeur à l'intérieur de la fonction modifie la valeur de la variable passée en paramètre ;
- Une **variable locale** est une variable qui n'existe qu'à l'intérieur d'un bloc défini ;
- L'instruction **`while`** permet de créer une boucle où un bloc sera exécuté tant qu'une condition ne sera pas invalidée.

JOUR 4

JOUR 4

DE NOUVEAUX LABYRINTHES

Nous pouvons déplacer un personnage dans un grand cadre vide. Il est temps d'ajouter des niveaux à notre jeu !

1. Un peu de logique booléenne pour pouvoir tester des valeurs
2. Tester des valeurs et effectuer des actions en fonction du résultat obtenu
3. Découpage en tranches
4. Une première version de notre jeu

À retenir

Un `tuple` est une liste optimisée pour la lecture et qui n'est pas accessible en écriture.

Nous avons choisi de stocker les labyrinthes sous forme de listes, comme le montre l'exemple du premier niveau :

Fichier

```
level_1 = [
  "+------------------+",
  "|                  |",
...
  "|                  |",
  "+------------------+"
]
```

Pour commencer, nous allons optimiser notre code. Une liste est une structure à laquelle on peut accéder en lecture et en écriture (on peut modifier un élément). Dans le cas du labyrinthe, les murs ne vont certainement pas bouger ! Nous pouvons alors utiliser une autre structure, s'utilisant exactement comme une liste : **le tuple**. Un tuple est une liste optimisée pour l'accès en lecture et qui n'est pas modifiable... En fait, c'est le principe de fonctionnement des chaînes de caractères. Pour définir un tuple, on n'utilise plus les crochets mais des parenthèses :

Terminal

```
>>> t = (1, 2, 3)
>>> print(t[0])
1
```

Bien sûr, à partir du moment où vous tenterez de modifier le contenu d'un tuple vous obtiendrez un message d'erreur :

Terminal

```
>>> t[0] = 5
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    t[0] = 5
TypeError: 'tuple' object does not support item assignment
```

La modification de notre premier niveau sera donc minime : remplacer les crochets par des parenthèses. Tout le reste du code continuera de fonctionner comme avant puisque nous ne modifions pas cette liste.

Fichier

```
level_1 = (
  "+------------------+",
  "|                  |",
...
  "|                  |",
  "+------------------+"
)
```

Dans notre première version du jeu, il manque beaucoup de choses pour pouvoir réellement jouer, mais un élément essentiel est absent : la sortie du labyrinthe ! Nous allons la symboliser par le caractère **O** :

Fichier

```
level_1 = (
  "+------------------+",
  "|                  |",
...
  "|                 O|",
  "+------------------+"
)
```

Pour pouvoir détecter la présence du personnage sur la sortie, nous devons modifier la fonction de vérification des déplacements :

Fichier

```
def verification_deplacement(lab, pos_col, pos_ligne):
    """
        ...
    """
    # Calcul de la taille du labyrinthe
    n_cols   = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
       pos_col > (n_cols - 1):
        return None
    elif lab[pos_ligne][pos_col] == "O":
        # Une position hors labyrinthe indique la victoire
        return [-1, -1]
    elif lab[pos_ligne][pos_col] != " ":
        return None
    else:
        return [pos_col, pos_ligne]
```

Il faut également modifier la boucle du jeu de manière à détecter la victoire dans un niveau :

Fichier

```
def jeu(level, perso, pos_perso):
    """
        ...
    """
    while True:
        affiche_labyrinthe(level, perso, pos_perso)
        choix_joueur(level, pos_perso)
        if pos_perso == [-1, -1]:
            print("Vous avez passé le niveau !")
            break
```

Nous introduisons ici deux nouveautés :

- Un test conditionnel entre une variable est une liste. Plutôt que de tester chaque élément de la liste par `pos_perso[0] == -1 and pos_perso[1] == -1`, il est possible de tester les éléments par identification (est-ce que le premier élément de la liste de gauche est égal au premier élément de la liste de droite, etc.) ;
- L'instruction `break`, qui permet de sortir de la boucle. Quelle que soit l'instruction de boucle (`for` ou `while`), l'instruction `break` indiquera au programme qu'il faut interrompre le traitement du bloc courant (bloc relatif à la boucle) et exécuter les commandes suivantes.

Nous pouvons maintenant sortir d'un labyrinthe... Mais si nous passions à un mécanisme un peu plus sophistiqué de gestion des niveaux ? Un mécanisme qui permettrait de créer simplement un nouveau labyrinthe dans un fichier qui serait ensuite chargé par le jeu. Pour cela, il va falloir être capable de lire un fichier texte.

JOUR 4

1. CHARGEMENT DES LABYRINTHES DEPUIS DES FICHIERS EXTERNES

Commençons par le plus simple : le format des labyrinthes contenus dans les fichiers. Nous utilisions jusqu'alors des listes de chaînes de caractères, donc nous allons rester dans le même format c'est-à-dire des chaînes de caractères dans un fichier texte. Nous nommerons nos fichiers `level_numero_du_niveau.txt` où `numero_du_niveau` sera un entier identifiant le niveau de manière unique. Le fichier `level_1.txt` pourra par exemple contenir :

Fichier

```
+------------------+
|      |           |
|--- + |   -----+  |
|    |           |  |
| +  +-------+   |  |
| |          |   |  |
| +---+     +---+  |
|     |            |
|--+  +------------|
|     |            |
|--+  |   +        |
|  |  +   |   +----|
|  |      |        |
|  +------+---+    |
|             |    |
|    +-----+  +    |
|    |              |
| +--|   +----------|
|    |            O|
+------------------+
```

Pour afficher ce niveau, il faut lire le fichier, placer les données dans un tuple et exécuter le code que nous avons déjà écrit. La manipulation d'un fichier passe par trois étapes :

1. Ouverture du fichier : création d'une variable qui nous permettra d'accéder au fichier. Un fichier peut être ouvert dans différents modes en fonction de ce que l'on souhaite réaliser :
   - ⇨ en lecture (*read* en anglais) : pour lire des données. Si le fichier que l'on désire ouvrir n'existe pas, un message d'erreur apparaîtra ;
   - ⇨ en écriture (*write* en anglais) : pour écrire des données. Si le fichier que l'on veut créer existe déjà, il est détruit et remplacé par le nouveau (on dit qu'il est écrasé) ;
   - ⇨ en ajout (*append* en anglais) : pour ajouter des données à un fichier existant. Si le fichier à ouvrir existe, l'écriture des données commence à la fin du fichier, il n'y a donc pas perte des informations qui étaient précédemment stockées à l'intérieur. Si le fichier n'existe pas, il est créé.
2. Action sur le fichier : en fonction des cas, il s'agira de la lecture ou de l'écriture.
3. Fermeture du fichier : après avoir travaillé sur le fichier, nous indiquons au système que nous n'avons plus besoin d'y accéder.

D'un point de vue syntaxique, l'ouverture d'un fichier se fait par la fonction `open` :

Terminal

```
>>> fic = open("level_1.txt", "r")
```

La variable que nous avons appelée `fic` est un descripteur de fichier. Dorénavant, c'est grâce à elle que nous pourrons accéder au fichier dont nous avons passé le nom en paramètre, c'est-à-dire `level_1.txt`.

Le second paramètre, la chaîne de caractères ne contenant qu'une seule lettre, est le mode d'ouverture du fichier. Nous retrouvons les trois modes décrits précédemment : `"r"` pour *read* – la lecture, `"w"` pour *write* – l'écriture, et `"a"` pour *append* – l'ajout. Il faut savoir que, par défaut, le fichier sera recherché dans le répertoire courant du programme (vous pouvez bien sûr également indiquer un chemin absolu, c'est-à-dire un chemin complet partant de la racine de l'arborescence des fichiers représentée par `/`).

Il existe ensuite plusieurs fonctions permettant de lire des données dans un fichier. Nous considérerons qu'une ligne est une suite de caractères se terminant par un saut à la ligne :

- ⇨ `readlines()` : lit toutes les lignes du fichier et les place dans une liste ;
- ⇨ `readline()` : lit une liste et la place dans une chaîne de caractères ;
- ⇨ `read()` : lit toutes les données du fichier et les place dans une seule chaîne de caractères ;
- ⇨ `read(n)` : lit `n` caractères et les place dans une chaîne de caractères.

Dans tous les cas, le saut de ligne qui aura été lu dans le fichier sera conservé dans la chaîne de caractères stockée en mémoire. Donc, si vous l'affichez à l'aide d'un `print` sans désactiver le retour à la ligne automatique du paramètre `end`, vous obtiendrez une ligne vide superflue.

Pour l'écriture il n'y a qu'une seule fonction, `write`, qui prend en paramètre la chaîne de caractères à écrire dans le fichier.

Ces fonctions s'exécutent d'une manière un peu spéciale : on ne leur transmet pas le descripteur de fichier en paramètre, mais on dit qu'elles s'appliquent au descripteur de fichier. Cette nuance fait que l'on ne doit pas écrire :

Fichier
```
level = readlines(fic)
```

Mais plutôt :

Fichier
```
level = fic.readlines()
```

Vous l'aurez compris, dans le cas de notre jeu c'est la fonction `readlines` qui est la plus adaptée. Il faudra seulement convertir la liste de résultat en tuple grâce à la fonction de conversion du même nom :

Fichier
```
level = tuple(fic.readlines())
```

Une fois que nous avons fini de travailler sur le fichier, nous l'indiquons en fermant le descripteur de fichier :

Fichier
```
fic.close()
```

Avant de mettre en place cette technique dans notre jeu, effectuons quelques tests dans le shell Python. Nous allons créer un fichier, y écrire quelques lignes, le refermer puis l'ouvrir pour lire son contenu et s'assurer qu'il correspond bien à ce que nous y avons enregistré. Première étape, la création du fichier :

Terminal
```
>>> fic.close()
>>> fic = open("fichier_test.txt", "w")
>>> fic.write("Une ligne...\n")
13
>>> fic.write("Encore une ligne !\n")
19
>>> fic.close()
```

À chaque fois que nous écrivons dans le fichier, la fonction **write** nous indique le nombre de caractères qui ont été écrits. Notez la présence de **\n** à la fin des chaînes de caractères pour passer à la ligne (sinon nos deux chaînes se trouveront l'une à la suite de l'autre dans le fichier).

La deuxième étape est la lecture. Nous allons réaliser le même type de lecture que ce dont nous allons avoir besoin dans notre jeu :

Terminal
```
>>> fic = open("fichier_test.txt", "r")
>>> data = tuple(fic.readlines())
>>> fic.close()
>>> data
('Une ligne...\n', 'Encore une ligne !\n')
```

La variable **data** est un tuple de chaînes de caractères... Mais nous allons être ennuyés par la présence du caractère de retour à la ligne si nous ne voulons pas trop modifier le code que nous avons déjà écrit pour afficher les labyrinthes :

Terminal
```
>>> print(data[0])
Une ligne...

>>>
```

Nous affichons bien deux sauts à la ligne ! Une solution pourrait être d'employer le slicing pour supprimer le dernier caractère :

Terminal
```
>>> print(data[0][:len(data[0]) - 1])
Une ligne...
>>>
```

Mais il faudrait quand même modifier le code d'affichage... En fait, il nous faudrait une instruction capable d'enlever les caractères superflus en fin de chaîne (espace, tabulation, saut à la ligne). Cette instruction existe, c'est **strip**, qui permettra de retirer tous les caractères invisibles du début et de la fin d'une chaîne (**lstrip** pour seulement le début et **rstrip** pour seulement la fin). Là encore, cette instruction s'utilise d'une manière un peu spéciale en l'appliquant directement à une chaîne de caractères :

Terminal
```
>>> print(data[0].strip())
Une ligne...
```

Nous pouvons maintenant écrire la fonction de lecture d'un labyrinthe dans un fichier extérieur et supprimer la définition de la variable **level_1** qui nous est désormais inutile :

Fichier

```
001: def charge_labyrinthe(nom):
002:     """
003:         Charge le labyrinthe depuis le fichier nom.txt
004:
005:         nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)
006:
007:         Valeur de retour :
008:           Tuple contenant les données du labyrinthe
009:     """
010:     fic = open(nom + ".txt", "r")
011:     data = fic.readlines()
012:     fic.close()
013:
014:     for i in range(len(data)):
015:         data[i] = data[i].strip()
016:
017:     return tuple(data)
```

Lecture des données dans le fichier

Parcours de la liste data pour supprimer les caractères invisibles de début et de fin de toutes les lignes

Conversion de la liste data en tuple

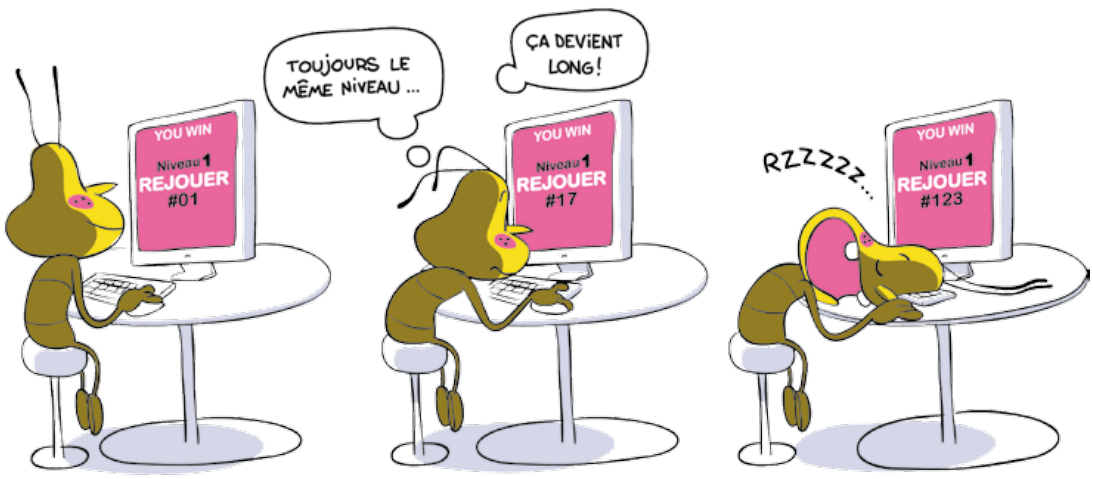
La seule autre modification se situe alors dans le programme principal, où il faut lire le niveau avant de lancer le jeu :

Fichier

```
118: ##########
119: # Programme principal
120: # Initialisation du personnage
121: perso = "X"
122: pos_perso = [1, 1]
123: # Lancement de la partie
124: level_1 = charge_labyrinthe("level_1")
125: jeu(level_1, perso, pos_perso)
```

Lecture du fichier level_1.txt et stockage du labyrinthe sous forme de tuple dans la variable level_1

Nous pouvons maintenant jouer… à un niveau ! Il est temps d'ajouter des labyrinthes.



2. CHANGEMENTS DE NIVEAUX

Nous avons mis en place un mécanisme permettant de définir les labyrinthes dans des fichiers distincts. Il est donc simple de changer de niveau en fonction du nombre de niveaux disponibles. Nous avons décidé de nommer nos fichiers `level_num.txt` et donc, s'il y a vingt niveaux de disponibles, il faudra charger les fichiers `level_1.txt` à `level_20.txt`…

Une simple boucle dans le programme principal peut faire cela :

Fichier

```
118: ##########
119: # Programme principal
120: # Initialisation du personnage
121: perso = "X"
122: pos_perso = [1, 1]
123: n_levels = 20
124: # Lancement de la partie
125: for i in range(1, n_levels + 1):
126:     level = charge_labyrinthe("level_" + str(i))
127:     jeu(level, perso, pos_perso)
```

Variable contenant le nombre total de niveaux

Boucle où i prendra ses valeurs entre 1 et n_levels

Pour créer le nom du fichier, nous concaténons la chaîne «level_» et le contenu de la variable i convertie en chaîne de caractères

Il est ainsi possible de changer de niveau… Mais le joueur n'a aucune information sur le niveau où il se trouve. Nous allons ajouter une barre de score qui indiquera le niveau. Cette barre sera gérée par une fonction indépendante :

Fichier

```
20: def barre_score(n_level):
21:     """
22:         Barre de score affichant les données du jeu
23:
24:         n_level : niveau courant
25:
26:         Pas de valeur de retour
27:     """
28:     print("Level : {:3d}".format(n_level))
```

Cette fonction est très simple puisqu'elle ne contient en fait qu'une ligne d'affichage. Il faut toutefois noter que nous employons un affichage formaté un peu particulier, puisqu'au lieu de **{}** nous avons employé **{:3d}**. Le code **:3d** donné entre les accolades permet d'indiquer que nous souhaitons afficher un entier (la lettre **d**) sur trois chiffres (le chiffre **3**) alignés à droite. Cette astuce nous assure qu'il n'y aura pas de décalage d'affichage au passage des dizaines ou des centaines.

Comme c'est la fonction **jeu** qui est chargée de l'affichage du labyrinthe, c'est elle aussi qui va afficher la barre de score et il va falloir lui transmettre un paramètre supplémentaire : le numéro du niveau courant. L'appel de la fonction va donc être modifié :

Fichier

```
137: # Lancement de la partie
138: for n_level in range(1, n_levels_total + 1):
139:     level = charge_labyrinthe("level_" + str(n_level))
140:     jeu(level, n_level, perso, pos_perso)
```

Et bien sûr, la fonction **jeu** elle-même devra être revue :

Fichier

```
112: def jeu(level, n_level, perso, pos_perso):
113:     """
114:         Boucle principale du jeu. Affiche le labyrinthe dans ses différents
115:         états après les déplacements du joueur.
116:
117:         level : Labyrinthe
118:         n_level : numéro du niveau courant
```

Fichier

```
119:         perso : caractère représentant le personnage
120:         pos_perso : liste contenant la position du personnage [colonne, ligne]
121:     """
122:     while True:
123:         affiche_labyrinthe(level, perso, pos_perso)
124:         barre_score(n_level)
125:         choix_joueur(level, pos_perso)
126:         if pos_perso == [-1, -1]:
127:             print("Vous avez passé le niveau !")
128:             break
```

Nous n'avons pas vu ici de nouvelle instruction ou de nouvelle technique de programmation, nous avons simplement amélioré le jeu. Pour continuer les améliorations, nous allons maintenant avoir besoin d'une nouvelle notion permettant d'ajouter des fonctionnalités au langage.

3. AJOUT DE FONCTIONNALITÉS À PYTHON ET STRUCTURATION DU CODE

Il est possible d'enrichir le langage Python avec de nouvelles fonctionnalités. Cela passe par des sortes de bibliothèques de fonctions, que l'on peut charger et utiliser à loisir : **les modules**. Et Python en compte des milliers, dont certains sont installés par défaut avec toute distribution !

Les instructions fournies par un module sont inconnues tant que celui-ci n'a pas été chargé (c'est lui qui contient le code qui définit ces instructions). Pour charger un module, on utilise l'instruction **`import`** suivie du nom du module. Toutes les fonctions fournies par le module sont alors accessibles en préfixant leur nom par le nom du module. Prenons l'exemple du module **`sys`** donnant accès à des fonctions et des variables système et notamment à la variable **`platform`** qui indique le nom du système d'exploitation sur lequel le programme est exécuté. Si nous tentons de l'afficher sans avoir chargé le module **`sys`**, voici ce qui se passe :

Terminal

```
>>> print(platform)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(platform)
NameError: name 'platform' is not defined
```

Après chargement du module, il ne faut pas oublier de préfixer la variable par le nom du module :

Terminal

```
>>> import sys
>>> print(platform)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(platform)
NameError: name 'platform' is not defined
>>> print(sys.platform)
linux2
```

Nous sommes donc en mesure de déterminer sous quel système d'exploitation notre jeu est exécuté. Cela va nous permettre de réaliser un appel système grâce à la fonction **system** du module **os** et d'effacer l'écran avant chaque affichage pour avoir l'impression que le personnage se déplace dans le labyrinthe. Ce n'est pas une obligation, mais pour plus de lisibilité, les modules sont généralement chargés en début de programme :

Fichier

```
001: import sys
002: import os
```

La fonction d'effacement de l'écran ne contiendra qu'un test pour savoir quel appel système exécuter :

Fichier

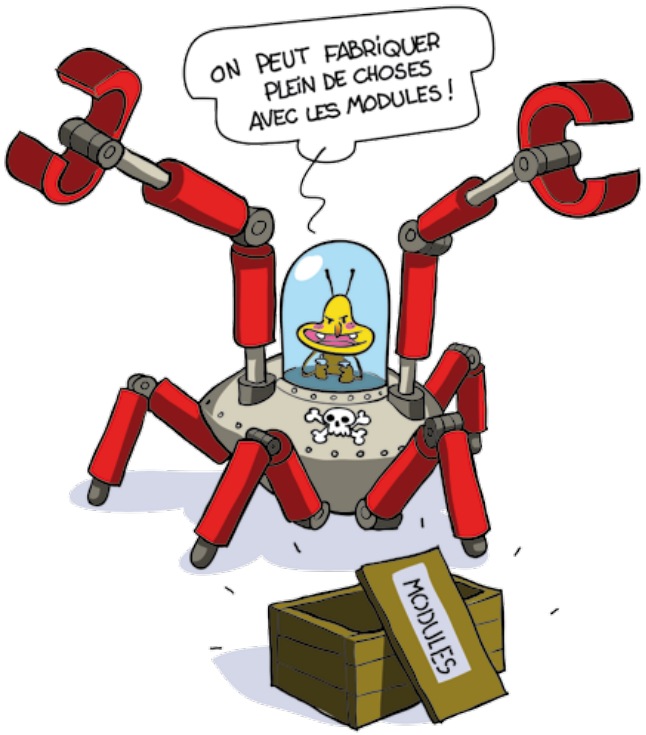
```
058: def efface_ecran():
059:     """
060:         Efface l'écran de la console
061:     """
062:     if sys.platform.startswith("win"):
063:         # Si système Windows
064:         os.system("cls")
065:     else:
066:         # Si système Linux ou OS X
067:         os.system("clear")
```

Il suffit de lire l'anglais pour comprendre cette ligne : la fonction startswith s'applique à une chaîne de caractères et teste si elle commence par la chaîne qui lui est transmise en paramètre. Il existe bien sûr également la fonction endswith

Cette fonction est appelée dans la fonction **jeu** juste avant d'afficher le labyrinthe :

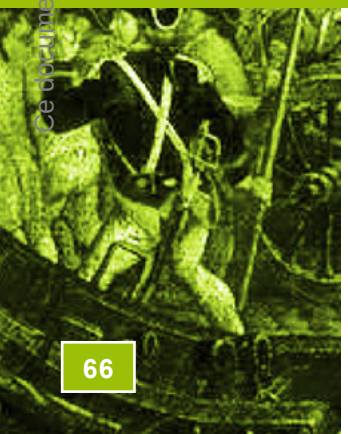
Fichier

```
128: def jeu(level, n_level, perso, pos_perso):
...
138:     while True:
139:         efface_ecran()
140:         affiche_labyrinthe(level, perso, pos_perso)
...
```

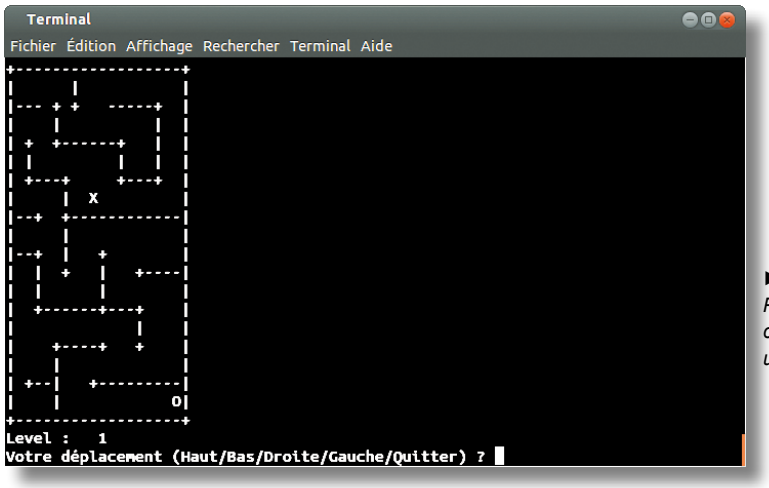


À retenir

Un module est un ensemble de fonctions Python permettant d'ajouter des fonctionnalités au langage. D'un point de vue purement technique, tout fichier Python est un module...

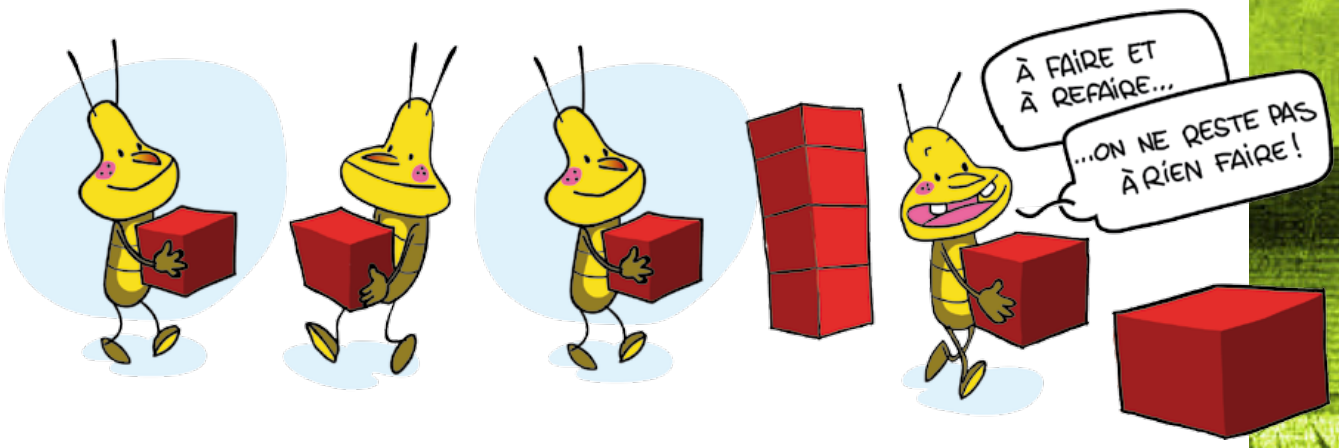


Notre jeu commence à ressembler à un vrai jeu (en mode console, certes). La figure 1 montre une capture d'écran d'une partie lancée dans une console.



► *Fig. 1 : Partie en cours dans une console.*

Arrivés à ce point, nous allons une nouvelle fois restructurer notre code à l'aide des connaissances acquises aujourd'hui. Comme il va y avoir de nouveaux fichiers et que cela risque d'être compliqué à suivre, je vais détailler les mécanismes de la restructuration sur des exemples simples et vous pourrez retrouver l'intégralité du code du jeu en fin d'article.



La restructuration va consister à créer un module contenant les fonctions et ce module sera chargé par le programme principal. Un module, nous l'avons vu, est un simple fichier Python. Ainsi, nous pouvons créer le fichier **mon_module.py** :

Fichier

```
def ma_fonction():
    print("OK")

print("Je suis dans mon_module")
```

Si vous vous placez dans le shell Python pour charger ce module, vous aurez une petite surprise :

Terminal

```
>>> import mon_module
Je suis dans mon_module
```

Nous ne voulions pas du `print`, nous voulions simplement avoir accès à la fonction que nous proposait le module ! Un test permet alors de savoir si le fichier Python a été appelé de manière directe (comme exécutable par exemple) ou chargé en tant que module :

Fichier

```
def ma_fonction():
    print("OK")

if __name__ == "__main__" :
    print("Je suis dans mon_module")
    print(__name__)
```

La variable `__name__` est une variable spéciale qui prend pour valeur `"__main__"` si le module est exécuté directement, ou le nom du module si celui-ci est chargé. Exécutons notre code :

Terminal

```
Je suis dans mon_module
__main__
```

Et maintenant chargeons-le dans le shell Python :

Terminal

```
>>> import mon_module
>>> mon_module.ma_fonction()
OK
```

Le test conditionnel sur la variable `__name__` permet donc de déterminer quelle est la portion de code qui correspond au code principal. ▉

À retenir

Le chargement d'un module se fait en indiquant son nom sans l'extension `.py`.

Prenez garde de ne pas nommer vos scripts du même nom qu'un module existant : Python chargerait alors votre module et non le module standard (par défaut Python recherche d'abord les modules dans le répertoire courant). Ainsi, si le fichier `os.py` existe dans votre répertoire courant, l'instruction `os.system` n'existera pas puisque c'est votre fichier qui aura été chargé !

Pour récapituler :

- Un tuple est une liste protégée contre l'écriture et qui est plus rapidement accessible ;
- Le traitement d'un fichier passe par trois étapes : ouverture (création du descripteur de fichier), action et fermeture ;
- Il existe trois modes d'ouverture d'un fichier : en écriture `"w"`, en lecture `"r"` et en ajout `"a"` ;
- Un module est un fichier qui fournit de nouvelles fonctionnalités ;
- La variable spéciale `__name__` permet de savoir si l'on a chargé un fichier Python en tant que module, ou s'il a été directement exécuté.

Résumé code :

Le fichier **Lab.py** contient les fonctions principales du jeu :

```
001: import sys
002: import os
003:
004:
005: def charge_labyrinthe(nom):
006:     """
007:         Charge le labyrinthe depuis le fichier nom.txt
008:
009:         nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)
010:
011:         Valeur de retour :
012:           Tuple contenant les données du labyrinthe
013:     """
014:     fic = open(nom + ".txt", "r")
015:     data = fic.readlines()
016:     fic.close()
017:
018:     for i in range(len(data)):
019:         data[i] = data[i].strip()
020:
021:     return tuple(data)
022:
023:
024: def barre_score(n_level):
025:     """
026:         Barre de score affichant les données du jeu
027:
028:         n_level : niveau courant
029:
030:         Pas de valeur de retour
031:     """
032:     print("Level : {:3d}".format(n_level))
033:
034:
035: def affiche_labyrinthe(lab, perso, pos_perso):
036:     """
037:         Affichage d'un labyrinthe.
038:
039:         lab : Variable contenant le labyrinthe
040:         perso : caractère représentant le personnage
041:         pos_perso : liste contenant la position du personnage [ligne, colonne]
042:
043:         Pas de valeur de retour
044:     """
045:     n_ligne = 0
046:     for ligne in lab:
047:         if n_ligne == pos_perso[1]:
048:           print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0] + 1:])
049:         else:
050:             print(ligne)
051:         n_ligne += 1
052:
053:
```



```
054: def efface_ecran():
055:     """
056:         Efface l'écran de la console
057:     """
058:     if sys.platform.startswith("win"):
059:         # Si système Windows
060:         os.system("cls")
061:     else:
062:         # Si système Linux ou OS X
063:         os.system("clear")
064:
065:
066: def verification_deplacement(lab, pos_col, pos_ligne):
067:     """
068:         Indique si le déplacement du personnage est autorisé ou pas.
069:
070:         lab : Labyrinthe
071:         pos_ligne : position du personnage sur les lignes
072:         pos_col : position du personnage sur les colonnes
073:
074:         Valeurs de retour :
075:           None : déplacement interdit
076:           [col, ligne] : déplacement autorisé sur la case indiquée par la liste
077:     """
078:     # Calcul de la taille du labyrinthe
079:     n_cols   = len(lab[0])
080:     n_lignes = len(lab)
081:     # Teste si le déplacement conduit le personnage en dehors de l'aire
082:     # de jeu
083:     if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
084:        pos_col > (n_cols - 1):
085:         return None
086:     elif lab[pos_ligne][pos_col] == "O":
087:         # Une position hors labyrinthe indique la victoire
088:         return [-1, -1]
089:     elif lab[pos_ligne][pos_col] != " ":
090:         return None
091:     else:
092:         return [pos_col, pos_ligne]
093:
094:
095: def choix_joueur(lab, pos_perso):
096:     """
097:         Demande au joueur de saisir son déplacement et vérifie s'il est
098:         possible. Si ce n'est pas le cas affiche un message, sinon
099:         modifie la position du perso dans la liste pos_perso
100:
101:         lab: Labyrinthe
102:         pos_perso : liste contenant la position du personnage [colonne, ligne]
103:
104:         Pas de valeur de retour
105:     """
106:     choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
107:     if choix == "H" or choix == "Haut":
108:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1)
109:     elif choix == "B" or choix == "Bas":
```

```
110:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] + 1)
111:     elif choix == "G" or choix == "Gauche":
112:         dep = verification_deplacement(lab, pos_perso[0] - 1, pos_perso[1])
113:     elif choix == "D" or choix == "Droite":
114:         dep = verification_deplacement(lab, pos_perso[0] + 1, pos_perso[1])
115:     elif choix == "Q" or choix == "Quitter":
116:         exit(0)
117:     if dep == None:
118:         print("Déplacement impossible")
119:         input("Appuyez sur <Return> pour continuer")
120:     else:
121:         pos_perso[0] = dep[0]
122:         pos_perso[1] = dep[1]
123:
124:
125: def jeu(level, n_level, perso, pos_perso):
126:     """
127:         Boucle principale du jeu. Affiche le labyrinthe dans ses différents
128:         états après les déplacements du joueur.
129:
130:         level : Labyrinthe
131:         n_level : numéro du niveau courant
132:         perso : caractère représentant le personnage
133:         pos_perso : liste contenant la position du personnage [colonne, ligne]
134:     """
135:     while True:
136:         efface_ecran()
137:         affiche_labyrinthe(level, perso, pos_perso)
138:         barre_score(n_level)
139:         choix_joueur(level, pos_perso)
140:         if pos_perso == [-1, -1]:
141:             print("Vous avez passé le niveau !")
142:             input("Appuyez sur <Return> pour continuer")
143:             break
```

Le fichier **GLMF_Game.py** est le fichier principal qui va utiliser le module **Lab** pour lancer le jeu. La première ligne permet d'indiquer sous Linux quel programme doit être utilisé pour exécuté le code. Il suffit ainsi de rendre le fichier exécutable pour lancer le jeu.

```
01: #!/usr/bin/python3
02:
03: import Lab
04:
05: if __name__ == "__main__":
06:     # Initialisation du personnage
07:     perso = "X"
08:     pos_perso = [1, 1]
09:     n_levels_total = 20
10:     # Lancement de la partie
11:     for n_level in range(1, n_levels_total + 1):
12:         level = Lab.charge_labyrinthe("level_" + str(n_level))
13:         Lab.jeu(level, n_level, perso, pos_perso)
14:     print("Vous avez gagné !")
```


JOUR 5

JOUR 5

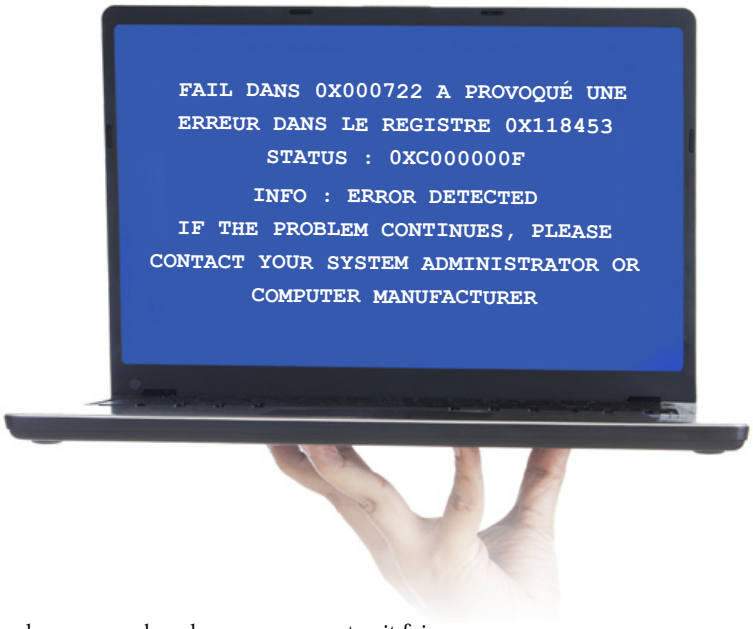
ENNEMIS, COMBATS ET TRÉSORS

Les bases de notre jeu sont posées et tout semble fonctionner correctement. Mais il reste bien sûr encore des choses à améliorer d'un point de vue technique et du point de vue du jeu, car trouver seulement la sortie d'un labyrinthe manque cruellement de piquant...

1. La gestion des erreurs
2. Les trésors
3. Les ennemis et les combats

JOUR 5

Lorsque l'on développe un programme, il y a une partie essentielle à ne pas négliger : la gestion des erreurs.
On ne peut pas laisser l'utilisateur complètement perdu face à un écran affichant un message incompréhensible. Imaginez-vous en train de jouer et puis, brusquement, le jeu s'arrête et l'écran prend une couleur uniforme (bleu par exemple, sans aucune arrière-pensée) et vous pouvez y lire le message ci-contre :



Vous conviendrez qu'à part éventuellement le développeur (et encore), personne ne peut comprendre pourquoi le programme s'est arrêté ! L'utilisateur ne sait donc pas si c'est une fausse manipulation de sa part qui a conduit à l'arrêt du jeu, ou un problème plus profond. Quoi qu'il en soit, ce genre de cas ne doit pas apparaître, il faut analyser les erreurs dans le programme et soit faire en sorte que le programme soit fermé proprement avec un message clair en indiquant la raison, soit permettre à l'utilisateur de corriger l'action ayant conduit à l'erreur.

1. LA GESTION DES ERREURS

Dans notre jeu, nous avons un exemple flagrant d'erreur non traitée : la saisie du joueur pour déplacer son personnage. En effet, nous ne traitons que les cas où la saisie est correcte (« Haut », « H », « Droite », etc.). Que se passe-t-il si le joueur indique un déplacement qui n'est pas traité ?

Terminal

```
+------------------+
...
+------------------+
Level :   1
Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? a
Traceback (most recent call last):
  File "./GLMF_Game.py", line 13, in <module>
    Lab.jeu(level, n_level, perso, pos_perso)
  File "/home/login/Jeu/Lab.py", line 139, in jeu
    choix_joueur(level, pos_perso)
  File "/home/login/Jeu/Lab.py", line 117, in choix_joueur
    if dep == None:
UnboundLocalError: local variable 'dep' referenced before assignment
```

Ici, nous avons beaucoup de détails sur la cause de l'erreur, des informations qui peuvent nous servir à corriger le problème... Ce n'est certainement pas à l'utilisateur de le faire ! Notre erreur se situe dans la fonction **`choix_joueur`** du fichier **`Lab.py`** en ligne 117 : nous testons le contenu d'une variable qui n'existe pas. Le message d'erreur se lit du bas vers le haut : en bas, la ligne ayant provoqué l'erreur et au-dessus les appels ayant conduit à cette erreur (fonction **`jeu`** dans **`GLMF_Game.py`** qui appelle **`choix_joueur`** dans **`Lab.py`**). Retrouvons donc le code ayant provoqué l'erreur :

Fichier

```
095: def choix_joueur(lab, pos_perso):
096:     """
...
105:     """
106:     choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
107:     if choix == "H" or choix == "Haut":
108:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1)
109:     elif choix == "B" or choix == "Bas":
110:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] + 1)
111:     elif choix == "G" or choix == "Gauche":
112:         dep = verification_deplacement(lab, pos_perso[0] - 1, pos_perso[1])
113:     elif choix == "D" or choix == "Droite":
114:         dep = verification_deplacement(lab, pos_perso[0] + 1, pos_perso[1])
115:     elif choix == "Q" or choix == "Quitter":
116:         exit(0)
117:     if dep == None:
118:         print("Déplacement impossible")
119:         input("Appuyez sur <Return> pour continuer")
120:     else:
121:         pos_perso[0] = dep[0]
122:         pos_perso[1] = dep[1]
```

Effectivement, arrivé en ligne 117, si le choix de l'utilisateur est différent de l'un des choix prédéfinis, la variable **dep** n'existe pas puisque nous ne sommes rentrés dans aucun choix conditionnel. Si cette variable existe et contient la valeur **None**, alors nous affichons que le déplacement est impossible (lignes 117 à 119). Il suffirait donc que la variable **dep** soit égale à **None** pour que, lors d'un choix non traité, nous affichions un message indiquant que l'action ne peut être réalisée et que nous restions dans le programme. Ajoutons donc une initialisation de cette variable en ligne 106 :

Fichier

```
095: def choix_joueur(lab, pos_perso):
096:     """
...
105:     """
106:     dep = None
107:     choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
108:     if choix == "H" or choix == "Haut":
109:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1)
...
```

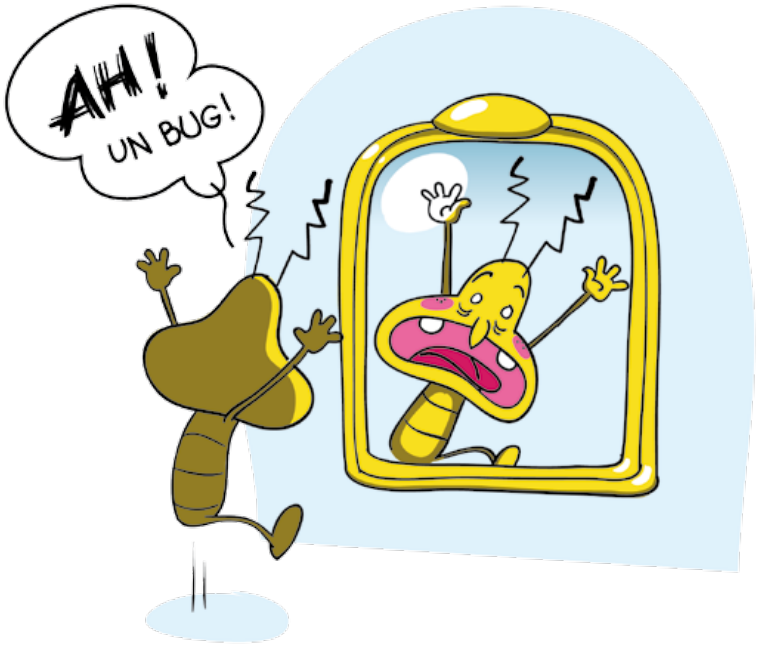
Cette fois, si nous saisissons une valeur non admise, nous obtenons un message nous indiquant que le déplacement est impossible et nous pouvons poursuivre notre partie :

Terminal

```
+------------------+
...
+------------------+
Level :   1
Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? a
Déplacement impossible
Appuyez sur <Return> pour continuer
```

L'analyse que nous venons d'effectuer est une séquence de débogage : nous sommes partis d'un problème précis, nous avons cherché à le comprendre pour le résoudre. Dans tout développement informatique vous rencontrerez de nombreuses erreurs (les fameux bugs)

qu'il faudra résoudre. Ici, en plus, l'erreur est provoquée par une saisie particulière de l'utilisateur et sans cette saisie, le bug n'est pas détectable... D'où l'intérêt de gérer tous les cas possibles lors d'une saisie : les choix corrects et ceux qui ne le sont pas.



Il existe un autre type d'erreur, qu'il faut traiter différemment : **les exceptions**. Lorsque vous essayez de convertir sous forme d'entier une chaîne de caractères ne contenant pas un entier, ou lorsque vous cherchez à ouvrir un fichier qui n'existe pas, vous déclenchez un mécanisme d'erreur particulier, une exception :

Terminal
```
>>> int("GLMF")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int("GLMF")
ValueError: invalid literal for int() with base 10: 'GLMF'
```

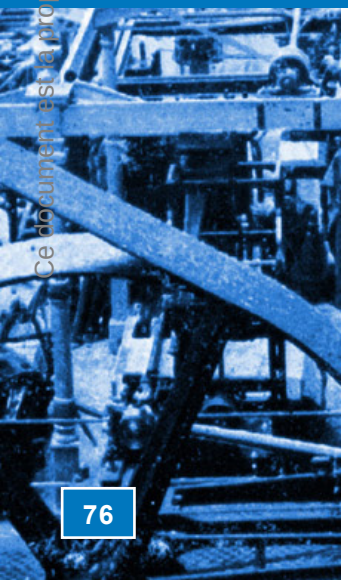
Ce que nous appelions jusqu'alors le nom de l'erreur est en fait le nom de l'exception et c'est grâce à ce nom que nous allons pouvoir traiter l'erreur. Dans le cadre de notre jeu, c'est l'ouverture des fichiers de niveaux qui risque de poser problème. Nous allons donc utiliser cet exemple pour mettre en place la gestion des exceptions. La première étape consiste à connaître le nom de l'exception à traiter. Le plus simple est de provoquer une erreur de ce type dans le shell Python :

Terminal
```
>>> fic = open("mon_fichier", "r")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    fic = open("mon_fichier", "r")
IOError: [Errno 2] No such file or directory: 'mon_fichier'
```

À savoir

Le débogage consiste à repérer et corriger les erreurs ou bugs dans le code.

Pourquoi dit-on « débogage » et non pas « debugage » puisque nous enlevons des bugs ? Tout simplement parce qu'en français nous sommes censés appeler une erreur un « bogue » (oui, comme pour la châtaigne). Pour la petite histoire, le premier bug détecté dans un ordinateur en était réellement un : en anglais « bug » signifie insecte et c'est la présence de ces insectes qui provoquaient des erreurs matérielles. Dire « bogue » au lieu de « bug », c'est comme dire « cédérom » au lieu de « CD-ROM » : c'est écrire en français au détriment du sens du mot. Nous emploierons donc ici le terme anglais *bug* et le terme français débogage, qui peut être prononcé plus ou moins à l'anglaise à l'oral, permettant ainsi de rester proche du sens du mot : retirer un insecte.



Il s'agit ici d'une exception **`IOError`**. Le traitement des exceptions se fait sous la forme d'une structure composée de plusieurs blocs :

- Le premier bloc contient le code à scruter : si une exception est déclenchée dans ce bloc, le programme ne sera pas directement interrompu, mais l'on cherchera dans les blocs suivants s'il y a des instructions permettant de gérer ce cas ;
- Les blocs suivants indiquent comment traiter les différentes exceptions.

D'un point de vue syntaxique, le premier bloc est ouvert par l'instruction **`try`** (« essaye » en anglais) et les autres blocs par **`exception`** suivi du nom de l'exception à traiter. Reprenons l'exemple de l'ouverture d'un fichier :

Fichier

```
01: try:
02:     fic = open("mon_fichier", "r")
03: except IOError:
04:     print("Impossible d'ouvrir le fichier !")
05:     exit(1)
06:  ...
```

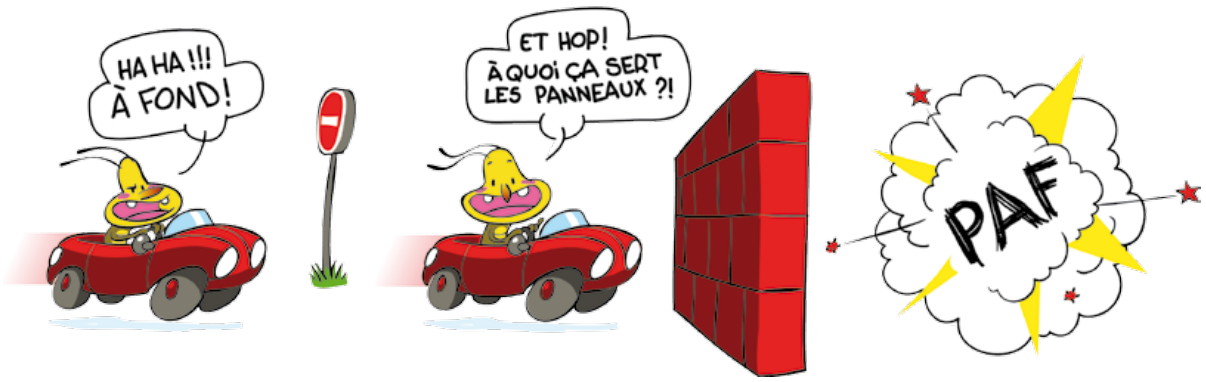
Si le fichier n'est pas accessible, alors les lignes 4 et 5 seront exécutées. Sinon, si tout se déroule correctement, alors le programme continuera son exécution en ligne 6. Si d'autres exceptions peuvent apparaître dans le bloc « scruté », il suffit d'ajouter des blocs de traitement :

Fichier

```
01: erreur = 0
02:
03: try:
04:     fic = open("mon_fichier", "r")
05:     print("conversion en entier :", int("une chaîne"))
06: except IOError:
07:     print("Impossible d'ouvrir le fichier !")
08:     exit(1)
09: except ValueError:
10:     print("Erreur lors de la conversion en entier")
11:     erreur = erreur + 1
12:
13: print("On continue l'exécution !")
14: ...
```

L'erreur sur l'ouverture du fichier est bloquante (puisque l'on sort du programme), alors que l'erreur sur la conversion en entier incrémente une variable et poursuit le traitement en ligne 13.

Vous pourriez être tenté d'utiliser ce mécanisme pour empêcher une erreur d'interrompre le programme sans pour autant la traiter. N'en faites surtout rien : vous pourriez engendrer des erreurs plus importantes et votre programme deviendrait très compliqué à déboguer.



2. LES TRÉSORS

Nous allons ajouter des trésors dans le jeu. Ces trésors seront répartis en trois catégories et seront représentés par un caractère distinct sur la carte chargée en mémoire (l'utilisateur ne verra lui que le même symbole) :

- ⇨ **1** : rapporte entre 1 et 5 pièces d'or ;
- ⇨ **2** : rapporte entre 5 et 10 pièces d'or ;
- ⇨ **3** : rapporte entre 0 et 25 pièces d'or.

Modifions donc notre carte **level_1.txt** :

Terminal

```
+------------------+
|      |           |
|--- + +   -----+  |
|    |          |  |
| +  +------+   |  |
| |         | 1 |  |
| +---+     +---+  |
|     |            |
|--+  +------------|
|     |            |
|--+  |   +        |
| 2|  +   |   +----|
|  |      |        |
|  +------+---+    |
|             |    |
|    +----+   +    |
| 3  |             |
| +--|   +---------|
|    |           o|
+------------------+
```

Quelles vont être les conséquences de ces modifications ?

- ⇨ Il va falloir tenir un compte du nombre de pièces d'or du joueur et les afficher dans la barre des scores ;
- ⇨ Le personnage devra pouvoir se déplacer sur une case trésor et il faudra ensuite supprimer ce trésor ;
- ⇨ Il faudra être capable de tirer au hasard un entier entre deux entiers donnés pour déterminer la quantité d'or gagnée.

Commençons par le plus simple : le tirage aléatoire. Python dispose d'un module baptisé **random**, qui contient de nombreuses fonctions en relation avec les nombres aléatoires. Pour obtenir une description de ce que font ces fonctions, vous pouvez utiliser le système d'aide pydoc dans le shell Python :

Terminal

```
>>> import random
>>> help(random)
Help on module random:

NAME
    random - Random variable generators.
...
```

Bien sûr le descriptif est un peu long et si vous ne savez pas vraiment ce que vous cherchez cela risque de prendre un peu de temps. La fonction **dir** viendra peut-être à votre secours : elle affichera toutes les fonctions liées à un module. En fonction du nom, qui est bien souvent explicite, vous pourrez focaliser ensuite la demande d'aide sur une fonction particulière :

Terminal

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_
Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__name__', '__package__', '_acos', '_ceil', '_cos', '_e',
'_exp', '_inst', '_log', '_pi', '_random', '_sha512', '_sin', '_sqrt',
'_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice',
'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate',
'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random',
'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular',
'uniform', 'vonmisesvariate', 'weibullvariate']
>>> help(random.randint)
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Les fonctions dont le nom est précédé de __ sont des fonctions internes au module et qui ne nous concernent pas. En regardant dans la liste suivante, la fonction **randint** devrait faire ce que nous cherchons : **rand** – aléatoire et **int** – entier. En appelant la fonction **help**, nous en obtenons la confirmation.

Donc, pour un trésor de première catégorie délivrant de une à cinq pièces d'or, nous aurons besoin de l'instruction suivante :

Terminal

```
>>> random.randint(1, 5)
3
>>> random.randint(1, 5)
1
```

Le principe d'un générateur aléatoire est de délivrer un nombre aléatoire. Donc, si vous quittez le shell Python et que vous le relancez, **randint** vous fournira d'autres suites d'entiers. Sur un ordinateur, les nombres ne sont pas générés réellement de façon aléatoire, un calcul est effectué pour obtenir ces nombres et il se base sur une valeur de départ. Si l'on fixe cette valeur de départ, les séries de nombres aléatoires seront toujours les mêmes… Ce qui est très utile pendant la phase de mise au point d'un programme. Pour initialiser cette valeur, appelée graine, on utilise la fonction **seed** à laquelle on transmet un paramètre :

Terminal

```
>>> import random
>>> random.seed(1)
>>> random.randint(1, 5)
2
>>> random.randint(1,5)
5
>>> ...
```

Si l'on relance le shell Python et que l'on exécute à nouveau ces lignes, nous obtiendrons les mêmes valeurs.

À retenir

Un dictionnaire est une liste de données indexées par un mot-clé : à chaque valeur est associée une autre valeur unique qui permet de l'identifier. Au contraire des listes, les dictionnaires ne sont pas des structures ordonnées. Si vous parcourez un dictionnaire, vous n'obtiendrez pas les éléments dans l'ordre dans lequel vous les avez insérés (pour des raisons de représentation en mémoire).

Nous pouvons déterminer le contenu d'un trésor, il faut maintenant que le joueur puisse stocker cet or. Pour cela, nous allons utiliser une variable et nous afficherons la valeur grâce à la fonction **`barre_score`**. Cette fonction devra donc avoir accès au numéro du niveau courant (**`n_level`**), à la quantité de pièces d'or du joueur, et par la suite au nombre de points de vie du joueur.

Pour simplifier le traitement des données, nous allons les regrouper au sein d'une structure de liste un peu particulière, dans laquelle on n'accède pas aux éléments à l'aide de leur position, mais à l'aide d'un mot-clé. Cette structure s'appelle **un dictionnaire** et on utilise des accolades pour le créer :

Terminal

```
>>> d = { "po" : 0, "pv" : "aucun" }
>>> d
{'pv': 'aucun', 'po': 0}
>>> print(d["pv"])
aucun
>>> d["test"] = True
>>> d
{'test': True, 'pv': 'aucun', 'po': 0}
```

Dans le fichier **`GLMF_Game.py`**, il faudra donc initialiser un dictionnaire contenant les données du jeu et le transmettre à la fonction **`jeu`** qui le transmettra elle-même à la fonction **`barre_score`** (voir le code complet en fin d'article).

Fichier

```
import Lab

if __name__ == "__main__":
    # Initialisation du personnage
    perso = "X"
    pos_perso = [1, 1]
    n_levels_total = 20
    data = {
        "or"    : 0,
        "pv"    : 25,
        "level" : None
    }
    # Lancement de la partie
    for n_level in range(1, n_levels_total + 1):
        level = Lab.charge_labyrinthe("level_" + str(n_level))
        data["level"] = n_level
        Lab.jeu(level, data, perso, pos_perso)
    print("Vous avez gagné !")
```

En ce qui concerne la gestion des trésors, il faut les afficher tous de la même manière. L'affichage du labyrinthe a lieu dans la fonction **`affiche_labyrinthe`** et c'est donc dans celle-ci que nous allons remplacer toutes les occurrences de **`1`**, **`2`** ou **`3`** par **`#`** (caractère choisi pour représenter un trésor). Pour cela, nous allons utiliser la fonction **`replace`** qui s'applique à une chaîne de caractères et remplace toutes les occurrences de son premier paramètre par le second :

Fichier

```
def affiche_labyrinthe(lab, perso, pos_perso, tresor):
    """
        Affichage d'un labyrinthe.

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [ligne, colonne]
        tresor : caractère représentant le trésor

        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        for i in range(1, 4):
            ligne = ligne.replace(str(i), tresor)
        if n_ligne == pos_perso[1]:
          print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0] + 1:])
        else:
            print(ligne)
        n_ligne += 1
```

i va valoir 1, puis 2 et enfin 3, ce qui correspond aux trois chiffres représentant des trésors

Il faut ensuite que le personnage puisse se retrouver sur une case de trésor et que dans ce cas la quantité d'or soit augmentée du nombre de pièces du trésor. De plus, il faut supprimer l'emplacement du trésor qui a été trouvé... Et là, nous allons avoir un problème : nous avons décidé d'utiliser un tuple pour stocker le labyrinthe et un tuple est une structure non modifiable... Il y a plusieurs solutions pour résoudre le problème :

1. Nous nous sommes trompés dans la structure de notre programme et il faut repasser le stockage des labyrinthes sous forme de listes ;
2. Il faut modifier le stockage de la position des trésors en les enregistrant, par exemple, sous forme de liste de coordonnées. Ce stockage sera alors décorrélé du labyrinthe ;
3. Utiliser la solution 2 tout en conservant les informations de positionnement des trésors dans le labyrinthe. C'est lors de la lecture que nous extrairons les positions des trésors et que nous les stockerons dans une liste ;
4. Utiliser une bidouille en convertissant le tuple en liste, en modifiant la liste, puis en reconvertissant la liste en tuple et en transmettant cet élément en paramètre à toutes les fonctions ;
5. Arrêter tout et pleurer comme Buggy.

Dans le développement d'un programme, on essaye de penser à tout... Mais on n'y arrive pas toujours. Il faut alors accepter de modifier le code pour que sa structure soit stable et qu'il puisse continuer à évoluer.

Ici, il faudrait appliquer la solution 3, qui est la plus pérenne et évolutive, mais la solution 1 est la plus simple à mettre en œuvre et c'est celle-ci que nous allons appliquer (la structuration sous forme de tuple du labyrinthe n'était qu'un prétexte pour aborder les tuples...).

Il suffit alors de modifier la fonction **`charge_labyrinthe`** pour qu'elle renvoie une liste et non plus un tuple et surtout de modifier la fonction **`verification_deplacement`** :

Fichier

```
def verification_deplacement(lab, pos_col, pos_ligne, data):
    """
        ...
    """
    # Calcul de la taille du labyrinthe
    n_cols   = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
       pos_col > (n_cols - 1):
        return None
    elif lab[pos_ligne][pos_col] == "O":
        # Une position hors labyrinthe indique la victoire
        return [-1, -1]
    elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
         lab[pos_col][pos_ligne] == "3":
        # Découverte d'un trésor
        decouverte_tresor(lab[pos_ligne][pos_col], data)

        lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
                         lab[pos_ligne][pos_col + 1:]

        return [pos_col, pos_ligne]
    elif lab[pos_ligne][pos_col] != " ":
        return None
    else:
        return [pos_col, pos_ligne]
```

Teste si le personnage se déplace sur l'une des trois catégories de trésor

Appel de la fonction chargée de calculer le montant du butin (voir le code suivant)

Une chaîne de caractères n'est pas modifiable et il faut donc créer une nouvelle chaîne à partir de l'ancienne en supprimant le trésor découvert

Le calcul du montant du butin est délégué à une fonction qui modifiera la variable contenant toutes les informations de jeu :

Fichier

```
def decouverte_tresor(categorie, data):
    """
        Incrémente le nombre de pièces d'or du joueur en fonction du trésor

        categorie : type de trésor
                    - 1 : entre 1 et 5 po
                    - 2 : entre 5 et 10 po
                    - 3 : entre 0 et 25 po
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    if categorie == "1":
        data["po"] = data["po"] + random.randint(1, 5)
    elif categorie == "2":
        data["po"] = data["po"] + random.randint(5, 10)
    else:
        data["po"] = data["po"] + random.randint(0, 25)
```

Calcul du nombre de pièces gagnées par tirage aléatoire en tenant compte de la catégorie du trésor

3. LES ENNEMIS ET LES COMBATS

La gestion des ennemis sera très similaire à celle des trésors (pour ne pas alourdir notre code, les ennemis resteront statiques). La gestion des combats se fera par tirage aléatoire d'un nombre entre 1 et 10 :

- 1 : l'ennemi est tué, mais perte de 5 à 10 points de vie ;
- 2 à 4 : l'ennemi est tué, mais perte de 1 à 5 points de vie ;
- 5 à 10 : l'ennemi est tué.

Les ennemis seront représentés par un caractère **$**. Nous ne verrons ici que la fonction réglant le combat, le reste des modifications étant identiques à celles effectuées pour ajouter les trésors (voir code complet en fin d'article).

Fichier

```
def combat(data):
    """
        Détermine le nombre de points de vie perdus lors d'un combat

        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    de = random.randint(1, 10)
    if de == 1:
        data["pv"] = data["pv"] - random.randint(5, 10)
    elif de >= 2 and de <= 4:
        data["pv"] = data["pv"] - random.randint(1, 5)
```

Tirage du dé entre 1 et 10

Perte de 5 à 10 points si le dé donne 1

Perte de 1 à 5 points si le dé donne 2, 3, ou 4

Le jeu dispose de règles très simples, mais il est maintenant parfaitement jouable ! ▮

Pour récapituler :

- Il faut toujours gérer toutes les erreurs ;
- Le traitement **try**/**except** permet d'intercepter et de traiter des exceptions ;
- Pour rechercher de l'aide dans le shell Python on peut utiliser les commandes **help** et **dir** ;
- Un dictionnaire est une liste dans laquelle les éléments ne sont pas indexés par une valeur numérique, mais par une clé qui peut être de n'importe quel type.

Résumé code :

Le fichier **Lab.py** contient les fonctions principales du jeu. C'est lui qui a été le plus modifié par rapport à hier :

```
001: import sys
002: import os
003: import random
004:
005:
006: def charge_labyrinthe(nom):
007:     """
008:         Charge le labyrinthe depuis le fichier nom.txt
009:
010:         nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)
011:
012:         Valeur de retour :
013:             - une liste avec les données du labyrinthe
014:     """
015:     try:
016:         fic = open(nom + ".txt", "r")
017:         data = fic.readlines()
018:         fic.close()
019:     except IOError:
020:         print("Impossible de lire le fichier {}.txt".format(nom))
021:         exit(1)
022:
023:     for i in range(len(data)):
024:         data[i] = data[i].strip()
025:
026:     return data
027:
028:
029: def barre_score(data):
030:     """
031:         Barre de score affichant les données du jeu
032:
033:         data : dictionnaire de données de la barre de score
034:
035:         Pas de valeur de retour
036:     """
037:     print("PV : {:2d}      PO : {:4d}      Level : {:3d}".format(data["pv"],
038:       data["po"], data["level"]))
039:
040:
041: def affiche_labyrinthe(lab, perso, pos_perso, tresor):
042:     """
043:         Affichage d'un labyrinthe.
044:
045:         lab : Variable contenant le labyrinthe
046:         perso : caractère représentant le personnage
047:         pos_perso : liste contenant la position du personnage [ligne, colonne]
048:         tresor : caractère représentant le trésor
049:
050:         Pas de valeur de retour
051:     """
052:     n_ligne = 0
053:      for ligne in lab:
054:          for i in range(1, 4):
```

```
055:             ligne = ligne.replace(str(i), tresor)
056:         if n_ligne == pos_perso[1]:
057:           print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0] + 1:])
058:         else:
059:             print(ligne)
060:         n_ligne += 1
061:
062:
063: def efface_ecran():
064:     """
065:         Efface l'écran de la console
066:     """
067:     if sys.platform.startswith("win"):
068:         # Si système Windows
069:         os.system("cls")
070:     else:
071:         # Si système Linux ou OS X
072:         os.system("clear")
073:
074:
075: def decouverte_tresor(categorie, data):
076:     """
077:         Incrémente le nombre de pièces d'or du joueur en fonction du trésor
078:
079:         categorie : type de trésor
080:                     - 1 : entre 1 et 5 po
081:                     - 2 : entre 5 et 10 po
082:                     - 3 : entre 0 et 25 po
083:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
084:     """
085:     if categorie == "1":
086:         data["po"] = data["po"] + random.randint(1, 5)
087:     elif categorie == "2":
088:         data["po"] = data["po"] + random.randint(5, 10)
089:     else:
090:         data["po"] = data["po"] + random.randint(0, 25)
091:
092:
093: def combat(data):
094:     """
095:         Détermine le nombre de points de vie perdus lors d'un combat
096:
097:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
098:     """
099:     de = random.randint(1, 10)
100:     if de == 1:
101:         data["pv"] = data["pv"] - random.randint(5, 10)
102:     elif de >= 2 and de <= 4:
103:         data["pv"] = data["pv"] - random.randint(1, 5)
104:
105:
106: def verification_deplacement(lab, pos_col, pos_ligne, data):
107:     """
108:         Indique si le déplacement du personnage est autorisé ou pas.
109:
110:         lab : Labyrinthe
111:         pos_ligne : position du personnage sur les lignes
112:         pos_col : position du personnage sur les colonnes
113:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
114:
```


```
115:         Valeurs de retour :
116:           None : déplacement interdit
117:           [col, ligne] : déplacement autorisé sur la case indiquée par la liste
118:     """
119:     # Calcul de la taille du labyrinthe
120:     n_cols   = len(lab[0])
121:     n_lignes = len(lab)
122:     # Teste si le déplacement conduit le personnage en dehors de l'aire
123:     # de jeu
124:     if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
125:        pos_col > (n_cols - 1):
126:         return None
127:     elif lab[pos_ligne][pos_col] == "O":
128:         # Une position hors labyrinthe indique la victoire
129:         return [-1, -1]
130:     elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
131:          lab[pos_col][pos_ligne] == "3":
132:         # Découverte d'un trésor
133:         decouverte_tresor(lab[pos_ligne][pos_col], data)
134:         lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
135:                          lab[pos_ligne][pos_col + 1:]
136:         return [pos_col, pos_ligne]
137:     elif lab[pos_ligne][pos_col] == "$":
138:         # Rencontre d'un ennemi
139:         combat(data)
140:         lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
141:                          lab[pos_ligne][pos_col + 1:]
142:         return [pos_col, pos_ligne]
143:     elif lab[pos_ligne][pos_col] != " ":
144:         return None
145:     else:
146:         return [pos_col, pos_ligne]
147:
148:
149: def choix_joueur(lab, pos_perso, data):
150:     """
151:         Demande au joueur de saisir son déplacement et vérifie s'il est
152:         possible. Si ce n'est pas le cas affiche un message, sinon
153:         modifie la position du perso dans la liste pos_perso
154:
155:         lab: Labyrinthe
156:         pos_perso : liste contenant la position du personnage [colonne, ligne]
157:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
158:
159:         Pas de valeur de retour
160:     """
161:     dep = None
162:     choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
163:     if choix == "H" or choix == "Haut":
164:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1,
165:             data)
166:     elif choix == "B" or choix == "Bas":
167:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] + 1,
168:             data)
169:     elif choix == "G" or choix == "Gauche":
170:         dep = verification_deplacement(lab, pos_perso[0] - 1, pos_perso[1],
171:             data)
172:     elif choix == "D" or choix == "Droite":
173:         dep = verification_deplacement(lab, pos_perso[0] + 1, pos_perso[1],
174:             data)
175:     elif choix == "Q" or choix == "Quitter":
```

```
176:         exit(0)
177:     if dep == None:
178:         print("Déplacement impossible")
179:         input("Appuyez sur <Return> pour continuer")
180:     else:
181:         pos_perso[0] = dep[0]
182:         pos_perso[1] = dep[1]
183:
184:
185: def jeu(level, data, perso, pos_perso, tresor):
186:     """
187:         Boucle principale du jeu. Affiche le labyrinthe dans ses différents
188:         états après les déplacements du joueur.
189:
190:         level : Labyrinthe
191:         data  : dictionnaire contenant
192:                 - level : le numéro de niveau
193:                 - po    : le nombre de pièces d'or
194:                 - pv    : le nombre de points de vie
195:         perso : caractère représentant le personnage
196:         pos_perso : liste contenant la position du personnage [colonne, ligne]
197:         tresor : caractère représentant le trésor
198:     """
199:     while True:
200:         efface_ecran()
201:         affiche_labyrinthe(level, perso, pos_perso, tresor)
202:         barre_score(data)
203:         if data["pv"] <= 0:
204:            print("Vous avez PERDU...")
205:            exit(0)
206:         choix_joueur(level, pos_perso, data)
207:         if pos_perso == [-1, -1]:
208:             print("Vous avez passé le niveau !")
209:             input("Appuyez sur <Return> pour continuer")
210:             break
```

Le fichier **GLMF_Game.py** n'a été que très peu modifié :

```
01: !/usr/bin/python3
02:
03: import Lab
04:
05: if __name__ == "__main__":
06:     # Initialisation du personnage
07:     perso          = "X"
08:     pos_perso      = [1, 1]
09:     tresor         = "#"
10:     n_levels_total = 20
11:     data = {
12:         "po"    : 0,
13:         "pv"    : 25,
14:         "level" : None
15:     }
16:     # Lancement de la partie
17:     for n_level in range(1, n_levels_total + 1):
18:         level = Lab.charge_labyrinthe("level_" + str(n_level))
19:         data["level"] = n_level
20:         Lab.jeu(level, data, perso, pos_perso, tresor)
21:     print("Vous avez gagné !")
```

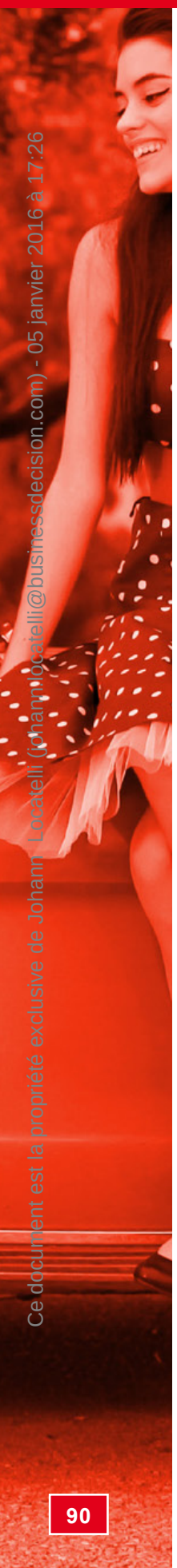
JOUR 6

JOUR 6

INTERFACE CONSOLE AMÉLIORÉE

Notre jeu est terminé, mais la saisie des déplacements est plutôt laborieuse : chaque saisie doit être validée par un appui sur la touche [Return]. Nous allons donc modifier notre programme pour obtenir une interface plus fluide.

1 Utilisation du module curses
2 Le module curses dans notre jeu



Le module **curses** permet de gérer l'affichage dans un terminal et l'interception des événements clavier (dès que vous appuyez sur une touche une information est envoyée au programme). Avant de pouvoir utiliser ce module pour améliorer notre jeu, il faut comprendre comment il fonctionne.

1. UTILISATION DU MODULE CURSES

L'utilisation du module **curses** impose de passer dans un mode « graphique » particulier. Nous devons donc passer par deux étapes obligatoires : au début du programme il faudra initialiser les paramètres graphiques et à la fin du programme il faudra restaurer les anciennes valeurs. Attention : pour utiliser le module **curses** vous devez forcément exécuter le code depuis un terminal. Sous le shell Python, vous obtiendrez un message d'erreur :

Terminal

```
>>> import curses
>>> curses.initscr()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    curses.initscr()
  File "/usr/lib/python3.2/curses/__init__.py", line 31, in initscr
    fd=_sys.__stdout__.fileno())
_curses.error: setupterm: could not find terminal
```

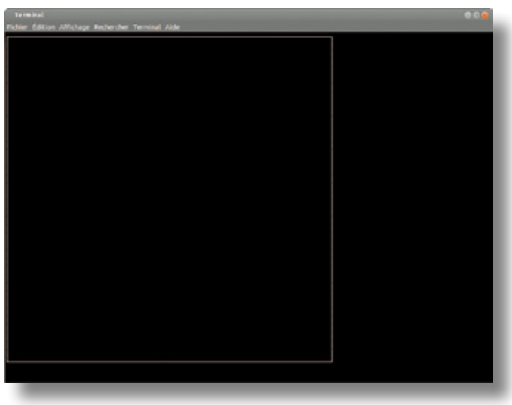
Comme nous allons devoir utiliser de nombreuses commandes, il est préférable de débuter directement par un petit programme de test **test_curses.py** :

Fichier

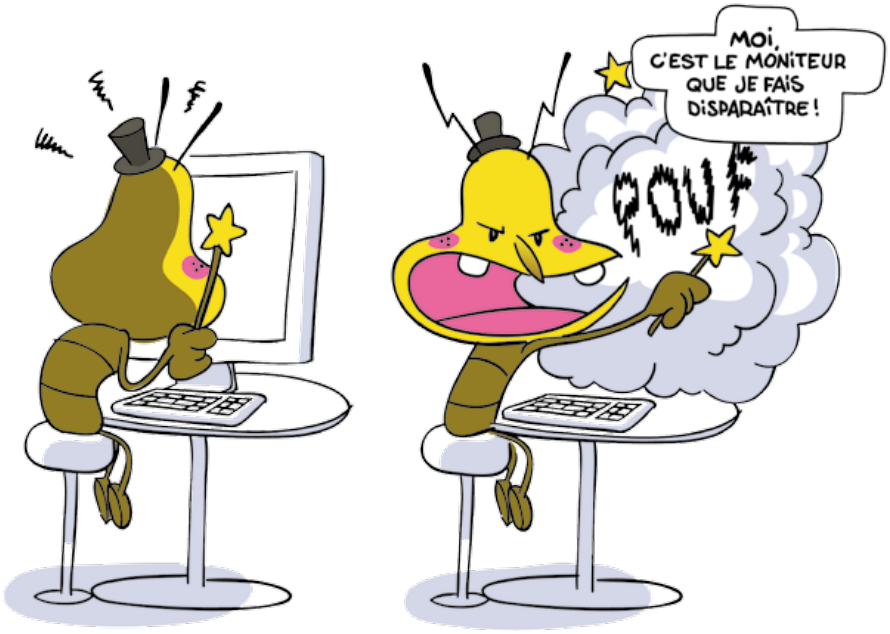
```
01: import curses
02:
03: if __name__ == "__main__":
04:     curses.initscr()        # Initialisation du mode 'graphique'
05:     curses.noecho()         # Désactivation de l'affichage des touches tapées au clavier
06:     curses.cbreak()         # Interception des touches tapées au clavier (sans appui sur [Return])
07:     curses.curs_set(0)      # Désactivation de l'affichage du curseur
08:
09:     window = curses.newwin(40, 79, 0, 0)
        # Création d'une fenêtre graphique dans laquelle nous écrirons. Sa taille est de
        # 40 caractères de haut et 79 de large et elle est positionnée en (0, 0), soit
        # en haut et à gauche du terminal
10:     window.border(0)        # Activation du tracé de la bordure de la fenêtre
11:     window.keypad(1)        # Activation du nommage des touches spéciales : KEY_UP
                                # désignera la flèche vers le haut, etc
12:
13:     c = window.getch()      # Attente de l'appui sur une touche du clavier
```

Cet exemple est un peu long pour débuter, mais il faut bien que nous puissions voir quelque chose... Ici, comme le montre la figure 1, un cadre va apparaître à l'écran et restera affiché jusqu'à ce que l'on appuie sur une touche (n'importe laquelle).

Notez qu'une fois le programme achevé, vous ne voyez plus ce que vous tapez dans le terminal... C'est normal, puisque nous n'avons pas restauré les paramètres d'affichage.



► *Fig. 1 : Fenêtre basique obtenue grâce au module curses*

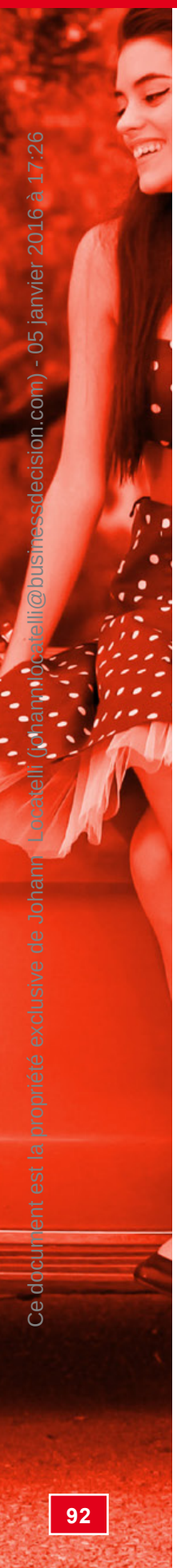


Nous allons donc reprendre notre programme en le structurant à l'aide d'une fonction d'initialisation et une fonction de fermeture. Nous en profiterons également pour ajouter un traitement d'erreur si l'initialisation de la fenêtre graphique est impossible :

Fichier

```
01: import curses
02:
03: def init_curses(lignes, cols, pos):
04:     """
05:         Initialisation des paramètres graphiques
06:
07:         lignes : nombre de lignes (en caractères)
08:         cols   : nombre de colonnes (en caractères)
09:         pos    : tuple contenant la position du coin supérieur gauche
10:                  de la fenêtre graphique
11:
12:         Valeur de retour :
13:             La fenêtre curses ayant été créée
14:     """
15:     curses.initscr()
16:     curses.noecho()
17:     curses.cbreak()
18:     curses.curs_set(0)
19:
20:     window = curses.newwin(lignes, cols, pos[0], pos[1])
21:     window.border(0)
22:     window.keypad(1)
23:     return window
24:
25:
26: def close_curses():
27:     """
28:         Restauration des paramètres graphiques
29:     """
30:     curses.echo()
31:     curses.nocbreak()
32:     curses.curs_set(1)
33:     curses.endwin()
```

Appel des fonctions inverses de celles invoquées dans init_curses



Fichier

```
34:
35:
36: if __name__ == "__main__":
37:     try:
38:         win = init_curses(40, 79, (0, 0))
39:         # Attente de l'appui sur une touche
40:         c = win.getch()
41:     except curses.error:
42:         print("Erreur graphique : interruption du programme")
43:     finally:
44:         close_curses()
```

Dans cet exemple, nous avons introduit un nouveau bloc lié au traitement de l'exception : le bloc **finally**. Ce bloc est exécuté quoi qu'il se passe, qu'il y ait eu une erreur interceptée ou non. Grâce à ce mécanisme, on s'assure de bien exécuter la fonction de restauration des paramètres initiaux du terminal, même si une erreur est rencontrée.

Pour écrire en couleur, il va falloir activer l'utilisation des couleurs et définir des associations entre un code (que nous utiliserons par la suite) et une paire de couleurs représentant les couleurs de premier plan et d'arrière-plan. Les couleurs que l'on peut utiliser sont stockées sous la forme **curses.COLOR_NAME**, où **NAME** est le nom d'une couleur en anglais (**RED**, **GREEN**, etc.).

Nous allons définir une fonction d'initialisation des couleurs, ce qui nous permettra ensuite d'afficher des messages colorés. Attention : l'affichage n'est plus exécuté à l'aide de la fonction **print**, mais d'une fonction **addstr** à laquelle on indique en quelle position (x, y) nous souhaitons positionner notre chaîne de caractères.

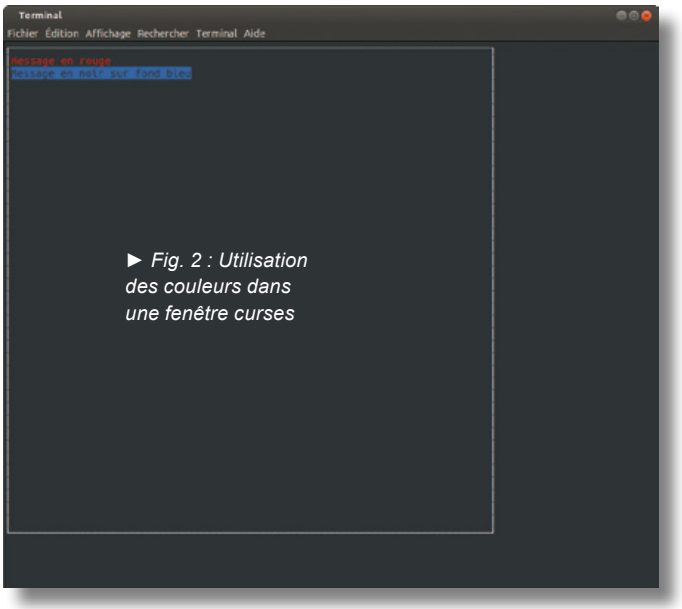
Fichier

```
01: import curses
02:
03: def init_curses(lignes, cols, pos):
...
24:
25:
26: def close_curses():
...
34:
35:
36: def init_colors():
37:     """
38:         Initialisation des couleurs
39:
40:         Valeur de retour :
41:             liste contenant le nom des couleurs (index = code couleur)
42:     """
43:     curses.start_color()
44:     curses.init_pair(1, curses.COLOR_RED, curses.COLOR_BLACK)
45:     curses.init_pair(2, curses.COLOR_GREEN, curses.COLOR_BLACK)
46:     curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_BLUE)
47:     return ["RED", "GREEN", "BLUE"]
48:
49:
50: def color(code, l_color):
51:     """
52:         Sélectionne une couleur
53:
54:         code : nom de la couleur
55:        l_color : liste des couleurs
```

Fichier

```
56:
57:         Valeur de retour :
58:             code de couleur curses
59:     """
60:     return curses.color_pair(l_color.index(code) + 1)
61:
62:
63: if __name__ == "__main__":
64:     try:
65:         win = init_curses(40, 79, (0, 0))
66:
67:         # Initialisation des couleurs
68:         coul = init_colors()
69:
70:         # Messages
71:         win.addstr(1, 1, "Message en rouge", color("RED", coul))
72:         win.addstr(2, 1, "Message en noir sur fond bleu", color("BLUE", coul))
73:
74:         # Attente de l'appui sur une touche
75:         c = win.getch()
76:     except curses.error:
77:         print("Erreur graphique : interruption du programme")
78:     finally:
79:         close_curses()
```

La figure 2 montre le résultat obtenu avec ce code. En ligne 60, nous avons utilisé une nouvelle fonction : **index**. Cette fonction, appliquée à une chaîne de caractères, permet de retrouver l'index correspondant à la valeur qui lui est passée en paramètre (si **RED** se trouve à l'index **0**, alors la fonction nous renverra **0**). Si la liste contient plusieurs fois la même valeur, c'est l'index de la première valeur rencontrée qui sera renvoyé.



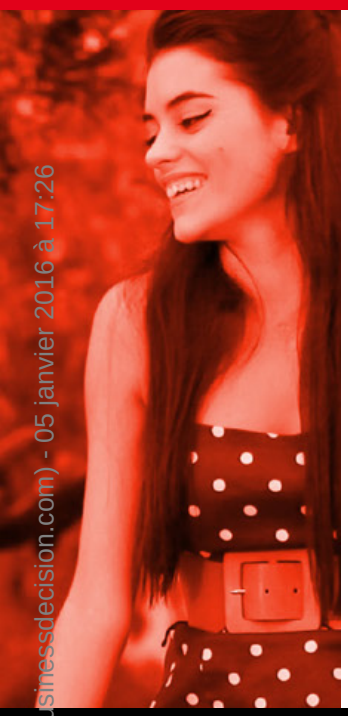
► *Fig. 2 : Utilisation des couleurs dans une fenêtre curses*

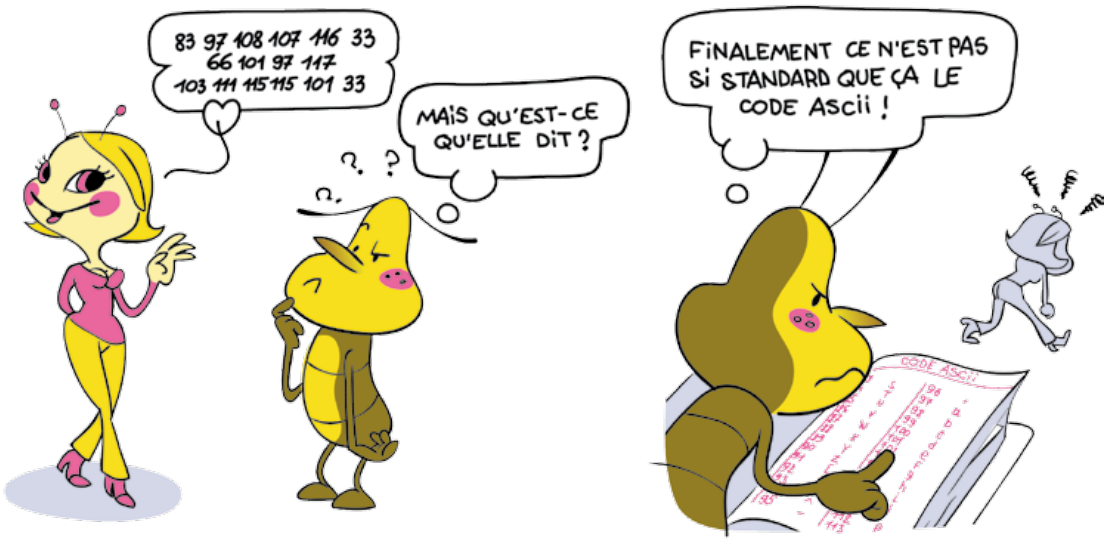
Enfin, pour terminer notre découverte de ce module, voyons comment intercepter des caractères au clavier. Nous avons vu la fonction **getch**... Il suffit donc d'effectuer une boucle sur cette instruction pour récupérer toutes les saisies au clavier et les traiter :

Fichier

```
c = None
while c != 27:
    c = win.getch()
    if c == curses.KEY_UP:
        win.addstr(10, 1, "Flèche vers le haut  ")
    elif c == curses.KEY_DOWN:
        win.addstr(10, 1, "Flèche vers le bas   ")
    elif c == curses.KEY_LEFT:
        win.addstr(10, 1, "Flèche vers la gauche")
    elif c == curses.KEY_RIGHT:
        win.addstr(10, 1, "Flèche vers la droite")
```

getch renvoie le code ASCII associé au caractère. 27 correspond à la touche [Echap]





À chaque touche on associe un code particulier. Ainsi, à la lettre **A** correspond le code **65**, à **B** correspond le code **66**, etc. Ce code est appelé **code ASCII** et permet d'identifier toutes les touches du clavier, y compris les touches spéciales. C'est pourquoi nous pouvons tester en condition de fin de boucle si le code de la touche est **27** : en appuyant sur la touche [Échap] on sort de la boucle. Pour les autres touches spéciales, nous utilisons les raccourcis fournis par le module **curses**. Pour une liste complète des raccourcis, vous pourrez consulter la page **http://docs.python.org/3/library/curses.html** en section 16.11.3 intitulée « Constants ».

2. LE MODULE CURSES DANS NOTRE JEU

Pour utiliser **curses** dans notre jeu, nous allons pouvoir récupérer les fonctions que nous avons créées pour tester le module : **init_curses**, **close_curses** et **color**. Comme nous avons segmenté notre code, seules les fonctions d'affichage devront être modifiées. La plus touchée sera la fonction **affiche_labyrinthe** :

Fichier

```
def affiche_labyrinthe(lab, perso, pos_perso, tresor, win, coul):
    """
        Affichage d'un labyrinthe.

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
        [ligne, colonne]
        tresor : caractère représentant le trésor
        win : fenêtre du mode graphique
        coul : liste de couleurs pour le mode graphique

        Pas de valeur de retour
    """
```

À savoir

Le code ASCII (pour *American Standard Code for Information Interchange*, soit code américain normalisé pour l'échange d'informations) est un code associé aux caractères affichables ou non. Par exemple, les codes 0 à 31 correspondent à des caractères accessibles depuis le clavier, mais non affichables tels que [Backspace], ou encore [Fin].

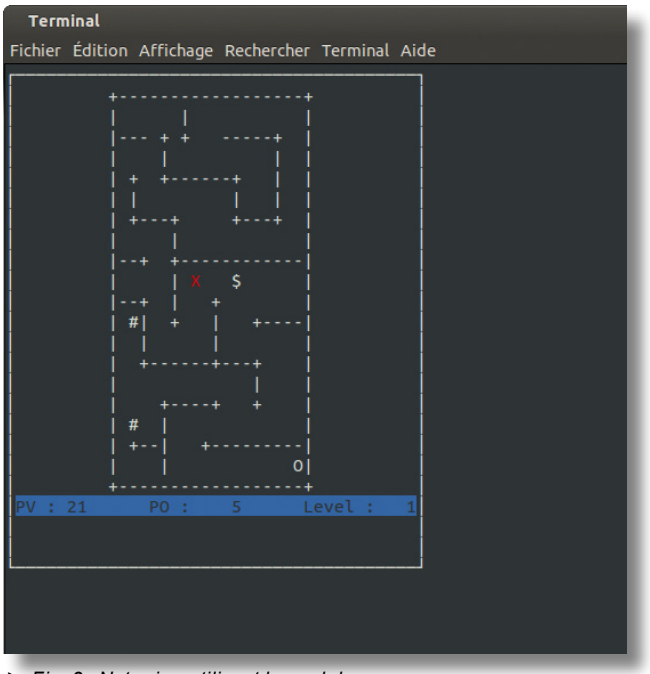
Le code ASCII ne code pas les caractères accentués (les codes vont de 0 à 127) ; le code ASCII étendu, qui lui va de 0 à 255, prend en compte les caractères accentués. La plupart du temps, lorsque l'on parle de code ASCII, c'est en fait le code ASCII étendu que l'on utilise. De nombreux sites Internet proposent des tables ASCII qui représentent la correspondance entre le code ASCII en décimal, hexadécimal, octal et sous forme de caractère.

Fichier

```
        n_ligne = 0
        for ligne in lab:
            for i in range(1, 4):
                ligne = ligne.replace(str(i), tresor)
            if n_ligne == pos_perso[1]:
                win.addstr(n_ligne + 1, 10, ligne[0:pos_perso[0]] + perso + \
                           ligne[pos_perso[0] + 1:])
                # Coloration du personnage
                win.addstr(n_ligne + 1, 10 + pos_perso[0], perso,
                    color("RED", coul))
            else:
                win.addstr(n_ligne + 1, 10, ligne)
            n_ligne += 1
```

Pour ne pas toucher à la structure de notre programme, nous continuons à afficher tous les caractères de toutes les lignes et il faut donc afficher une deuxième fois le personnage pour le colorer en rouge. En s'autorisant plus de modifications, comme le module **curses** permet d'afficher des caractères à des coordonnées précises dans la fenêtre, il suffirait d'afficher le labyrinthe une seule fois (et non plus à chaque fois que l'utilisateur appuie sur une touche) et il faudrait alors seulement effacer le personnage de son ancien emplacement et l'afficher à sa nouvelle position.

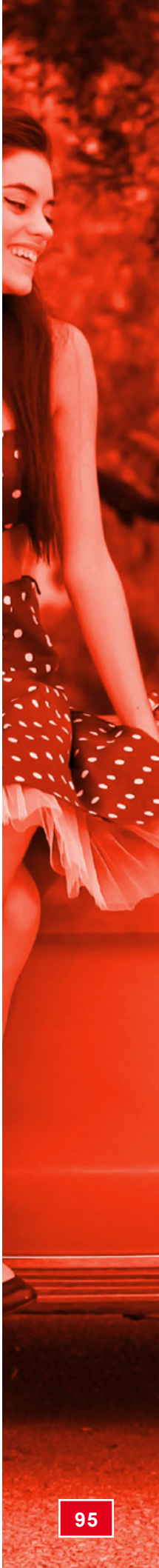
Cela peut constituer un bon exercice : partez du programme complet d'aujourd'hui et modifiez-le de manière à n'afficher les murs du labyrinthe qu'une seule fois et à colorer les ennemis en vert. ■

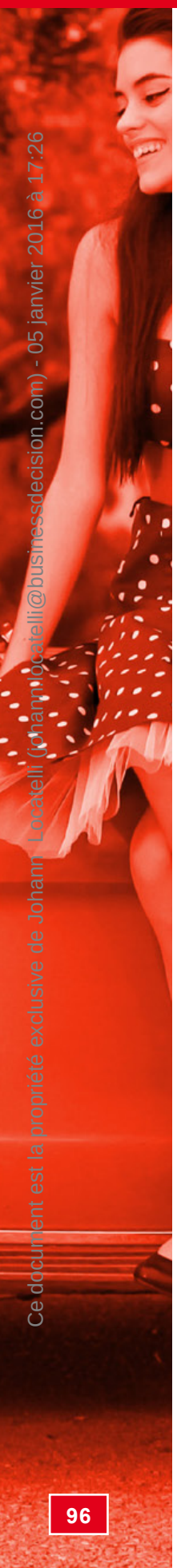


► *Fig. 3 : Notre jeu utilisant le module curses*

Pour récapituler :

- Le module **curses** permet d'améliorer les interfaces en mode console ;
- Lorsque l'on intercepte une exception avec un bloc **try**/**except**, l'ajout d'un bloc **finally** permet de s'assurer qu'un bloc de code soit exécuté, qu'il y ait ou non une erreur.



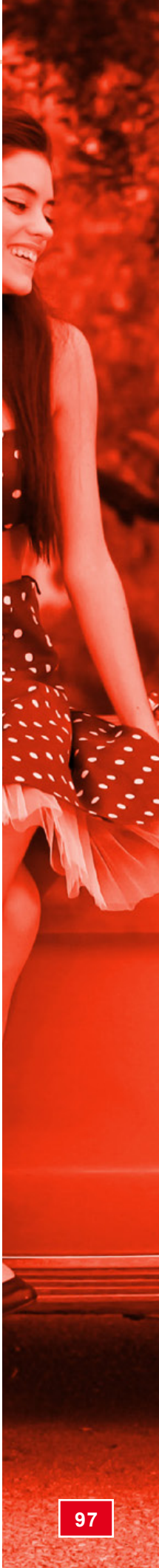


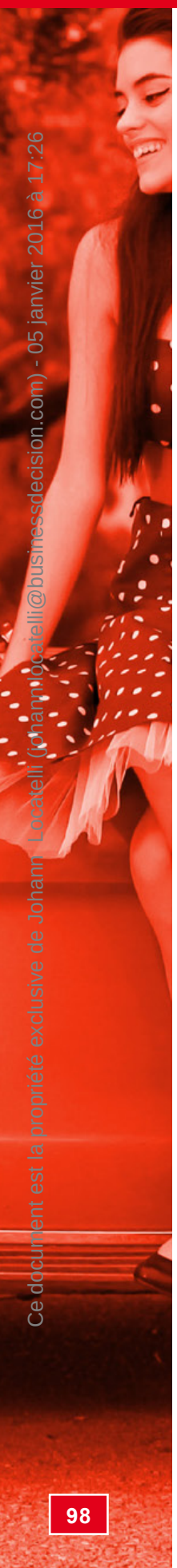
Résumé code :

Comme le fichier **Lab.py** était bien structuré, une fois que les fonctions spécifiques au mode « graphique » console ont été ajoutées, peu de lignes sont modifiées (voir lignes en jaune).

```
001: import random
002: import curses
003:
004: def init_curses(lignes, cols, pos):
005:     """
006:         Initialisation des paramètres graphiques
007:
008:         lignes : nombre de lignes (en caractères)
009:         cols   : nombre de colonnes (en caractères)
010:         pos    : tuple contenant la position du coin supérieur gauche
011:                  de la fenêtre graphique
012:
013:         Valeur de retour :
014:             La fenêtre curses ayant été créée
015:     """
016:     curses.initscr()
017:     curses.noecho()
018:     curses.cbreak()
019:     curses.curs_set(0)
020:
021:     window = curses.newwin(lignes, cols, pos[0], pos[1])
022:     window.border(0)
023:     window.keypad(1)
024:     return window
025:
026:
027: def close_curses():
028:     """
029:         Restauration des paramètres graphiques
030:     """
031:     curses.echo()
032:     curses.nocbreak()
033:     curses.curs_set(1)
034:     curses.endwin()
035:
036:
037: def init_colors():
038:     """
039:         Initialisation des couleurs
040:
041:         Valeur de retour :
042:             liste contenant le nom des couleurs (index = code couleur)
043:     """
044:     curses.start_color()
045:     curses.init_pair(1, curses.COLOR_RED, curses.COLOR_BLACK)
046:     curses.init_pair(2, curses.COLOR_GREEN, curses.COLOR_BLACK)
047:     curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_BLUE)
```

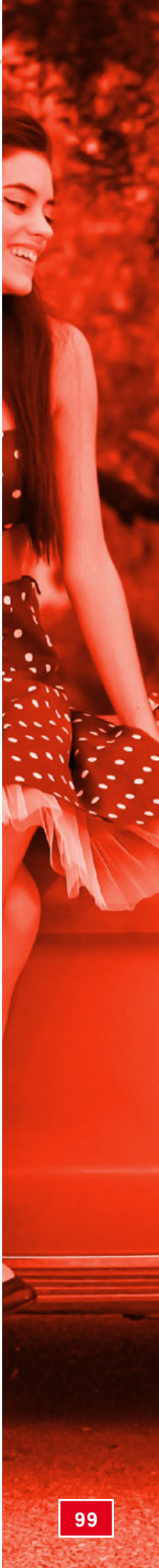
```
048:     return ["RED", "GREEN", "BLUE"]
049:
050:
051: def color(code, l_color):
052:     """
053:         Sélectionne une couleur
054:
055:         code : nom de la couleur
056:         l_color : liste des couleurs
057:
058:         Valeur de retour :
059:             code de couleur curses
060:     """
061:     return curses.color_pair(l_color.index(code) + 1)
062:
063:
064: def charge_labyrinthe(nom):
065:     """
066:         Charge le labyrinthe depuis le fichier nom.txt
067:
068:         nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)
069:
070:         Valeur de retour :
071:             - une liste avec les données du labyrinthe
072:     """
073:     try:
074:         fic = open(nom + ".txt", "r")
075:         data = fic.readlines()
076:         fic.close()
077:     except IOError:
078:         print("Impossible de lire le fichier {}.txt".format(nom))
079:         exit(1)
080:
081:     for i in range(len(data)):
082:         data[i] = data[i].strip()
083:
084:     return data
085:
086:
087: def barre_score(data, win, coul):
088:     """
089:         Barre de score affichant les données du jeu
090:
091:         data : dictionnaire de données de la barre de score
092:         win : fenêtre graphique
093:         coul : liste de couleurs pour le mode graphique
094:
095:         Pas de valeur de retour
096:     """
097:     barre = "PV : {:2d}      PO : {:4d}      Level : {:3d}"
098:     win.addstr(21, 1, barre.format(data["pv"],
```

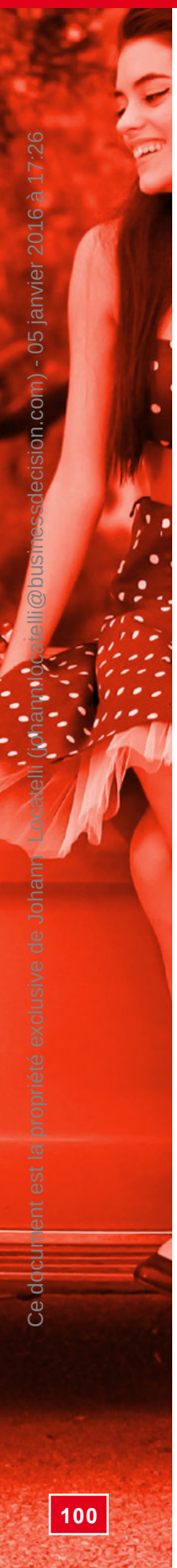






```
099:                 data["po"], data["level"]), color("BLUE", coul))
100:
101:
102: def affiche_labyrinthe(lab, perso, pos_perso, tresor, win, coul):
103:     """
104:         Affichage d'un labyrinthe.
105:
106:         lab : Variable contenant le labyrinthe
107:         perso : caractère représentant le personnage
108:         pos_perso : liste contenant la position du personnage [ligne, colonne]
109:         tresor : caractère représentant le trésor
110:         win : fenêtre du mode graphique
111:         coul : liste de couleurs pour le mode graphique
112:
113:         Pas de valeur de retour
114:     """
115:     n_ligne = 0
116:     for ligne in lab:
117:         for i in range(1, 4):
118:             ligne = ligne.replace(str(i), tresor)
119:         if n_ligne == pos_perso[1]:
120:             win.addstr(n_ligne + 1, 10, ligne[0:pos_perso[0]] + perso + \
121:                        ligne[pos_perso[0] + 1:])
122:             # Coloration du personnage
123:             win.addstr(n_ligne + 1, 10 + pos_perso[0], perso,
124:                 color("RED", coul))
125:         else:
126:             win.addstr(n_ligne + 1, 10, ligne)
127:         n_ligne += 1
128:
129:
130: def decouverte_tresor(categorie, data):
131:     """
132:         Incrémente le nombre de pièces d'or du joueur en fonction du trésor
133:
134:         categorie : type de trésor
135:                     - 1 : entre 1 et 5 po
136:                     - 2 : entre 5 et 10 po
137:                     - 3 : entre 0 et 25 po
138:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
139:     """
140:     if categorie == "1":
141:         data["po"] = data["po"] + random.randint(1, 5)
142:     elif categorie == "2":
143:         data["po"] = data["po"] + random.randint(5, 10)
144:     else:
145:         data["po"] = data["po"] + random.randint(0, 25)
146:
147:
148: def combat(data):
149:     """
```

```
150:         Détermine le nombre de points de vie perdus lors d'un combat
151:
152:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
153:     """
154:     de = random.randint(1, 10)
155:     if de == 1:
156:         data["pv"] = data["pv"] - random.randint(5, 10)
157:     elif de >= 2 and de <= 4:
158:         data["pv"] = data["pv"] - random.randint(1, 5)
159:
160:
161: def verification_deplacement(lab, pos_col, pos_ligne, data):
162:     """
163:         Indique si le déplacement du personnage est autorisé ou pas.
164:
165:         lab : Labyrinthe
166:         pos_ligne : position du personnage sur les lignes
167:         pos_col : position du personnage sur les colonnes
168:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
169:
170:         Valeurs de retour :
171:           None : déplacement interdit
172:           [col, ligne] : déplacement autorisé sur la case indiquée par la liste
173:     """
174:     # Calcul de la taille du labyrinthe
175:     n_cols   = len(lab[0])
176:     n_lignes = len(lab)
177:     # Teste si le déplacement conduit le personnage en dehors de l'aire
178:     # de jeu
179:     if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
180:        pos_col > (n_cols - 1):
181:         return None
182:     elif lab[pos_ligne][pos_col] == "O":
183:         # Une position hors labyrinthe indique la victoire
184:         return [-1, -1]
185:     elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
186:          lab[pos_ligne][pos_col] == "3":
187:         # Découverte d'un trésor
188:         decouverte_tresor(lab[pos_ligne][pos_col], data)
189:         lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
190:                          lab[pos_ligne][pos_col + 1:]
191:         return [pos_col, pos_ligne]
192:     elif lab[pos_ligne][pos_col] == "$":
193:         # Rencontre d'un ennemi
194:         combat(data)
195:         lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
196:                          lab[pos_ligne][pos_col + 1:]
197:         return [pos_col, pos_ligne]
198:     elif lab[pos_ligne][pos_col] != " ":
199:         return None
200:     else:
```

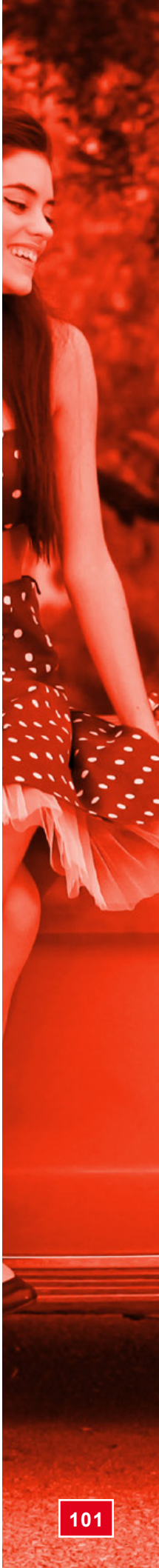



```
201:         return [pos_col, pos_ligne]
202:
203:
204: def choix_joueur(lab, pos_perso, data, win):
205:     """
206:         Demande au joueur de saisir son déplacement et vérifie s'il est
207:         possible. Si ce n'est pas le cas affiche un message, sinon
208:         modifie la position du perso dans la liste pos_perso
209:
210:         lab: Labyrinthe
211:         pos_perso : liste contenant la position du personnage [colonne, ligne]
212:         data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
213:         win : fenêtre graphique
214:
215:         Pas de valeur de retour
216:     """
217:     dep = None
218:     choix = win.getch()
219:     if choix == curses.KEY_UP:
220:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] - 1,
221:             data)
222:     elif choix == curses.KEY_DOWN:
223:         dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] + 1,
224:             data)
225:     elif choix == curses.KEY_LEFT:
226:         dep = verification_deplacement(lab, pos_perso[0] - 1, pos_perso[1],
227:             data)
228:     elif choix == curses.KEY_RIGHT:
229:         dep = verification_deplacement(lab, pos_perso[0] + 1, pos_perso[1],
230:             data)
231:     elif choix == 27:
232:         close_curses()
233:         exit(0)
234:     if dep != None:
235:         pos_perso[0] = dep[0]
236:         pos_perso[1] = dep[1]
237:
239:
239: def jeu(level, data, perso, pos_perso, tresor, win, coul):
240:     """
241:         Boucle principale du jeu. Affiche le labyrinthe dans ses différents
242:         états après les déplacements du joueur.
243:
244:         level : Labyrinthe
245:         data  : dictionnaire contenant
246:                 - level : le numéro de niveau
247:                 - po    : le nombre de pièces d'or
248:                 - pv    : le nombre de points de vie
249:         perso : caractère représentant le personnage
250:         pos_perso : liste contenant la position du personnage [colonne, ligne]
251:         tresor : caractère représentant le trésor
252:         win : fenêtre graphique
```

```
253:         coul : liste de couleurs pour la fenêtre graphique
254:     """
255:     while True:
256:         affiche_labyrinthe(level, perso, pos_perso, tresor, win, coul)
257:         barre_score(data, win, coul)
258:         if data["pv"] <= 0:
259:             win.addstr(1, 20, "Vous avez PERDU...", color("RED", coul))
260:             win.getch()
261:             close_curses()
262:             exit(0)
263:         choix_joueur(level, pos_perso, data, win)
264:         if pos_perso == [-1, -1]:
265:             win.addstr(22, 1, "Vous avez passé le niveau !", color("RED", coul))
266:             win.addstr(23, 1, "Appuyez sur une touche pour continuer",
267:                 color("RED", coul))
268:             win.getch()
269:             win.addstr(1, 20, " " * 50)
270:             win.addstr(1, 21, " " * 50)
271:             break
```

Même constat pour le fichier **GLMF_Game.py**, qui délègue l'essentiel du travail au module **Lab**.

```
01: #!/usr/bin/python3
02:
03: import Lab
04:
05: if __name__ == "__main__":
06:     # Initialisation du personnage
07:     perso          = "X"
08:     pos_perso      = [1, 1]
09:     tresor         = "#"
10:     n_levels_total = 20
11:     data = {
12:         "po"    : 0,
13:         "pv"    : 25,
14:         "level" : None
15:     }
16:
17:     # Initialisation de l'affichage graphique
18:     win = Lab.init_curses(25, 41, (0, 0))
19:     # Initialisation des couleurs
20:     coul = Lab.init_colors()
21:
22:     # Lancement de la partie
23:     for n_level in range(1, n_levels_total + 1):
24:         level = Lab.charge_labyrinthe("level_" + str(n_level))
25:         data["level"] = n_level
26:         Lab.jeu(level, data, perso, pos_perso, tresor, win, coul)
27:     win.addstr(1, 22, "Vous avez gagné !!", Lab.color("RED", coul))
28:     win.getch()
29:     Lab.close_curses()
```

JOUR 7

JOUR 7

PASSAGE EN MODE GRAPHIQUE

Notre jeu est terminé, mais il s'exécute dans une console et il n'est pas vraiment joli. Nous allons donc créer pour lui une interface graphique... Ce qui va complètement modifier notre code !

1. Une fenêtre graphique
2. Des objets
3. Dessiner dans une fenêtre
4. Gestion des événements

JOUR 7

Il existe de nombreux modules Python permettant de créer des interfaces graphiques. Certains sont même spécialement conçus pour le développement de jeux. Nous utiliserons ici un environnement graphique très simple et installé par défaut sur tous les systèmes : **Tk**.

Le problème avec le module **tkinter** que nous allons utiliser, c'est qu'il n'est pas écrit de la même manière que notre code, une architecture particulière a été utilisée : la programmation orientée objet. Ce style de programmation est un peu plus complexe que ce que nous avons fait jusqu'alors, mais il possède une grande qualité : le code est bien structuré et facilement réutilisable. Nous ne pourrons pas aborder en un jour tous les principes de la programmation orientée objet. Nous profiterons simplement du fait d'utiliser **tkinter** pour découvrir certaines notions clés.

1. UNE FENÊTRE GRAPHIQUE

L'ouverture d'une fenêtre graphique se fait en quelques lignes depuis le shell Python :

Terminal

```
>>> from tkinter import *
>>> fenetre = Tk()
```

Dès que vous aurez saisi la deuxième ligne, une petite fenêtre apparaîtra, identique à celle présentée en figure 1. Dans un script indépendant, une dernière ligne est nécessaire pour lancer l'affichage de la fenêtre graphique :

Fichier

```
fenetre.mainloop()
```

► *Fig. 1 : Fenêtre graphique vide*

Il s'agit de ce que l'on appelle la boucle événementielle : une boucle infinie qui s'exécute en attendant de récupérer les événements de l'utilisateur, tels que l'appui sur une touche du clavier, le déplacement de la souris, etc. Dans le shell Python, cette boucle n'est pas nécessaire et vous pourrez donc voir après chaque instruction tapée l'impact sur l'apparence de la fenêtre.

Nous avons chargé le module **tkinter** à l'aide de la syntaxe **from module import *** et non **import module** : c'est une autre façon de charger un module qu'il vaut mieux réserver aux modules qui fournissent des objets. En effet, avec cette écriture, vous n'avez plus besoin de nommer le module devant le nom d'une fonction. Vous pensez que c'est plus pratique ? Que se passe-t-il si deux modules **A** et **B** contiennent chacun une fonction **fct** qui ne réalise pas la même tâche ?

Fichier **A.py** :

Fichier

```
def fct():
    print("Je suis dans A !")
```

Fichier **B.py** :

Fichier

```
def fct():
    print("Je suis dans B !")
```

Test depuis le shell Python :

Terminal

```
>>> from A import *
>>> from B import *
>>> fct()
Je suis dans B !
```

La première version de la fonction, présente dans le module **A**, a donc été écrasée par la deuxième version et nous n'y avons plus accès. En utilisant la syntaxe **import** ce n'est pas le cas :

Terminal

```
>>> import A
>>> import B
>>> A.fct()
Je suis dans A !
>>> B.fct()
Je suis dans B !
```

Comme les objets sont définis dans un bloc qui leur est propre et porte leur nom, ce risque d'erreur n'est pas possible. Mais au fait, qu'est-ce qu'un objet ?

2. DES OBJETS

Dans la vie, nous sommes entourés d'objets : ce livre est un objet, la page que vous lisez est un objet... Bref, tout est objet ! Et comme vous l'aurez remarqué, un livre étant composé de pages, des objets peuvent être construits en utilisant d'autres objets.

En informatique, un objet est une entité représentant un élément à partir de ces caractéristiques fondamentales. Un livre a forcément un titre, un ou des auteur(s) et un éditeur. Ces informations seront stockées dans des variables attachées à l'objet livre : pour un livre donné nous aurons un titre, etc. Ces variables portent un nom spécifique pour bien montrer qu'elles définissent un objet. On les appelle **des attributs** de l'objet.

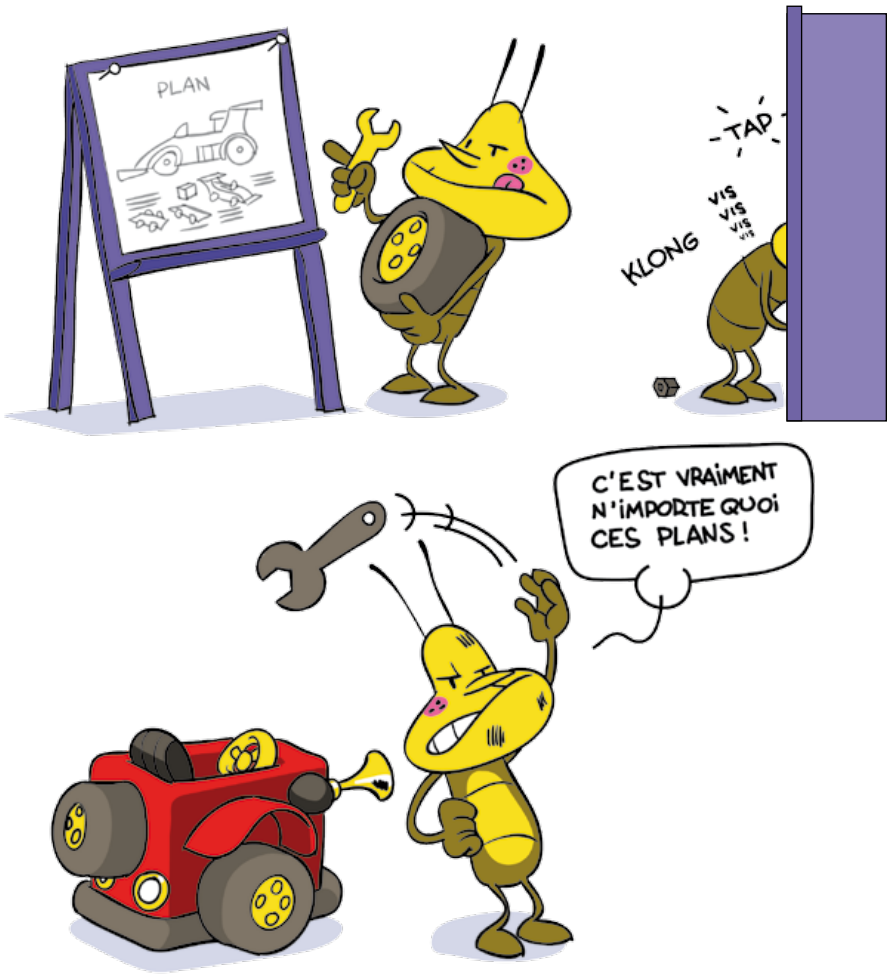
Il n'existe pas « un » livre, mais « des » livres. Lorsque nous définissons un objet, nous ne parlons pas d'un objet particulier, mais de l'ensemble des objets qu'il représente. On parle alors d'**une classe**. La classe « livre » permet de définir comment un livre doit être fabriqué et, plus tard, on utilisera ce modèle pour créer des livres qui seront alors **des instances** de la classe « livre » (des livres réels, qui ne sont pas seulement des définitions).

À savoir

Il existe plusieurs méthodes pour importer un module nommé A :

- ⇨ **`import A` : les fonctions issues de A doivent être préfixées par le nom du module ;**
- ⇨ **`import A as B` : les fonctions issues de A doivent être préfixées par B (on a renommé le module A dans notre code) ;**
- ⇨ **`from A import *` : toutes les fonctions de A sont importées directement ;**
- ⇨ **`from A import fct1, fct2` : seules les fonctions `fct1` et `fct2` du module A sont importées, il est impossible d'accéder aux autres.**

Prenons l'exemple des Lego. Une boîte de Lego contient un plan et des briques : le plan seul, ou une brique seule n'ont pas grand intérêt. Par contre, si l'on se sert du plan (la classe) pour assembler les briques, on peut créer un objet qui sera une instance du plan de départ. Si la boîte correspond à une voiture, nous obtiendrons une voiture et si nous avons d'autres briques, de couleurs différentes, en suivant le plan nous obtiendrons un autre modèle de voiture.



D'un point de vue informatique, des fonctions vont pouvoir être liées à un objet. Ces fonctions particulières, appelées **méthodes**, connaissent toutes la structure de l'objet et ont un accès direct à ses attributs. Voyons un exemple en Python, avec la classe **`Livre`** que nous stockerons dans un fichier du même nom **`Livre.py`** :

Fichier

```
class Livre(object):
```
Définition de la façon de construire un objet « Livre »

```
    def __init__(self, auteur, titre, editeur):
        self.auteur  = auteur
        self.titre   = titre
        self.editeur = editeur
```

Définition de la méthode permettant de construire l'objet.
Trois attributs définissent ce qu'est un livre

```
    def couverture(self):
        print(self.titre)
        print("de", self.auteur)
        print("édité par", self.editeur)
```
Méthode affichant la couverture d'un livre

Le mot-clé **`self`**, présent dans de nombreuses lignes, désigne l'objet courant et permet de faire la distinction entre les variables et les attributs. Toutes les méthodes prennent pour premier paramètre l'objet courant. La méthode **`__init__`** est une méthode particulière appelée **constructeur**. C'est elle qui sera appelée quand nous créerons des instances de la classe :

Terminal

```
>>> from Livre import *
>>> l1 = Livre("David Gemmell", "Druss la legende", "Milady")
>>> l2 = Livre("Richard Castle", "Vague de chaleur", "Gina Cowell")
>>> l1.couverture()
Druss la legende
de David Gemmell
édité par Milady
>>> l2.couverture()
Vague de chaleur
de Richard Castle
édité par Gina Cowell
```

l1 et **l2** sont des instances de la classe **Livre**. Ce sont donc des objets Livre créés grâce au constructeur **__init__**, qui est appelé indirectement dans les lignes **l1 = Livre(....)** et **l2 = Livre(...)** (pour vous en convaincre, vous pouvez ajouter une instruction **print** dans le constructeur). On peut lire le contenu de la couverture d'un livre grâce à la méthode **couverture**. Vous noterez que l'on ne transmet aucun paramètre à **couverture**... C'est l'opérateur **.** qui fait le lien et « transforme » l'instance **l1** ou **l2** en **self** lorsque l'on se retrouve dans la classe. Nous avons déjà utilisé de nombreuses méthodes sans les nommer, car en Python, tout ce que l'on manipule est un objet (entier, chaîne de caractères, etc.).

Pour revenir à l'interface graphique, le code **fenetre = Tk()** crée une instance de la classe **Tk**, qui est une fenêtre graphique. Nous allons maintenant pouvoir ajouter des éléments graphiques en comprenant ce que nous faisons.

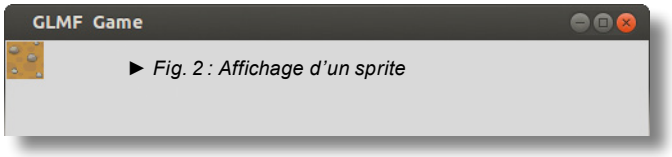
3. DESSINER DANS UNE FENÊTRE

Pour notre jeu, nous avons simplement besoin d'afficher de petites images (*sprites*) pour créer le labyrinthe et représenter les ennemis, les trésors, la sortie et le joueur. Il faudra 5 sprites différents, que l'on peut créer à l'aide de GIMP, ou en se basant sur des images existantes (attention à la licence !). Les images que j'utilise ici ont une taille de 30x30 px.

Regardons dans un premier temps comment afficher un seul sprite :

Fichier

```
from tkinter import *

fenetre = Tk()
fenetre.title("GLMF  Game")  <-- Ajout d'un titre en haut de la fenêtre graphique

can = Canvas(fenetre, width = 500, height = 500)
 Création d'un élément graphique permettant de dessiner à l'intérieur

photo_wall = PhotoImage(file = "sprites/wall.gif")
sprite_wall = can.create_image(0, 0, anchor = NW, image = photo_wall)
              Chargement de l'image wall.gif et affichage en position (0, 0)

can.pack()  <-- Positionnement effectif du canevas dans la fenêtre principale

fenetre.mainloop()  <-- Boucle événementielle
```



► *Fig. 2 : Affichage d'un sprite*

Lors de la création d'une instance de la classe **Canvas**, nous donnons en premier paramètre la fenêtre : il s'agit de l'objet dans lequel nous allons placer le canevas. La construction d'une interface graphique se fait sur la base d'empilements successifs : un bouton dans une fenêtre, une image dans le bouton, etc. Une fois la liaison faite entre le contenant et le contenu, il faut en quelque sorte la valider, la rendre effective. C'est ce que fait la méthode **pack**.
La fenêtre obtenue à l'aide de ce code est visible en figure 2.

Pour afficher le labyrinthe, nous utiliserons la fonction de lecture que nous avions définie et en parcourant les caractères de la liste obtenue, nous pourrons déterminer quel sprite afficher :

Fichier

```
def affiche_labyrinthe(lab, fenetre, size_sprite, pos_perso):
    """
        Affichage d'un labyrinthe.

        lab : Variable contenant le labyrinthe
        fenetre : Fenêtre graphique
        size_sprite : Taille des sprites en pixels
        pos_perso : Liste contenant la position du personnage

        Valeur de retour:
            Tuple contenant le canevas, le sprite du personnage et un
            dictionnaire des images utilisées pour les sprites
    """
    can = Canvas(fenetre, width = 600, height = 600)

    photo_wall     = PhotoImage(file = "sprites/wall.gif")
    photo_treasure = PhotoImage(file = "sprites/treasure.gif")
    photo_ennemy   = PhotoImage(file = "sprites/ennemy.gif")
    photo_exit     = PhotoImage(file = "sprites/exit.gif")
    photo_hero     = PhotoImage(file = "sprites/hero.gif")

    n_ligne = 0
    for ligne in lab:
        n_col = 0
        for car in ligne:
            # Murs
            if car == "+" or car == "-" or car == "|":
                can.create_image(n_col + n_col * size_sprite,
                    n_ligne + n_ligne * size_sprite, anchor = NW,
                    image = photo_wall)
            # Trésors
            if car == "1" or car == "2" or car == "3":
                can.create_image(n_col + n_col * size_sprite,
                    n_ligne + n_ligne * size_sprite, anchor = NW,
                    image = photo_treasure)
            # Ennemis
            if car == "$":
                can.create_image(n_col + n_col * size_sprite,
                    n_ligne + n_ligne * size_sprite, anchor = NW,
                    image = photo_ennemy)
            # Sortie
            if car == "O":
                can.create_image(n_col + n_col * size_sprite,
                    n_ligne + n_ligne * size_sprite, anchor = NW,
                    image = photo_exit)
            n_col += 1
        n_ligne += 1

    # Affichage du personnage
    sprite_hero = can.create_image(pos_perso[0] + pos_perso[0] * size_sprite,
```

Fichier

```
            pos_perso[1] + pos_perso[1] * size_sprite, anchor = NW,
            image = photo_hero)

        can.pack()

        return (can, sprite_hero, {
          "hero"     : photo_hero,
          "wall"     : photo_wall,
          "treasure" : photo_treasure,
          "ennemy"   : photo_ennemy,
          "exit"     : photo_exit})
```

Ce code est une réécriture de notre fonction **affiche_labyrinthe** avec les sprites : la structure reste pratiquement inchangée. Notez toutefois qu'il y a ici beaucoup de valeurs qui sont retournées par la fonction : nous devons garder accès au canevas pour le modifier, au sprite du personnage pour le déplacer et aux différentes photos pour les afficher (si elles ne sont plus en mémoire, rien n'apparaîtra à l'écran).

Pour appeler cette fonction, après avoir créé la fenêtre graphique, il faut charger un niveau et penser à récupérer les valeurs de retour (le code complet est disponible en fin d'article) :

Fichier

```
level = Lab.charge_labyrinthe("level_1")

(canvas, sprite_perso, photos) = Lab.affiche_labyrinthe(level, fenetre,
    size_sprite, pos_perso)
```

Passons maintenant à la saisie des déplacements du personnage.

4. GESTION DES ÉVÉNEMENTS

En mode graphique, toutes les actions de l'utilisateur sont codées sous forme d'événements. Un clic de souris, un appui sur une touche sont des événements. Chaque élément graphique de l'interface peut être associé à un ou plusieurs événement(s). Par exemple, si nous voulons afficher un message lorsque l'utilisateur clique dans la fenêtre (sur le canevas), nous ferons :

Fichier

```
canvas.bind("<Button-1>", click)
```

Nous « lions » l'événement **<Button-1>**, c'est-à-dire un clic gauche, sur l'objet graphique **canvas**. Lorsque l'événement se produit, la fonction **click** sera exécutée. Cette fonction doit être écrite de la manière suivante :

Fichier

```
def click(event):
  print("Vu ! Vous avez cliqué sur le jeu !")
```

Comme vous pouvez le constater, elle possède un paramètre... Alors que nous n'en avons transmis aucun. C'est normal, le système transmet automatiquement un paramètre d'informations sur le type d'événement intercepté. Ici, lors du clic de l'utilisateur, une variable est créée et transmise à la fonction **click** dont le nom était passé en paramètre à la méthode **bind**. Il s'agit bien du nom de la fonction et non d'un appel, comme vous pouvez le vérifier dans le shell Python avec un test sur la fonction **print** :

Terminal

```
>>> print("Hello")
Hello
>>> print
<built-in function print>
```

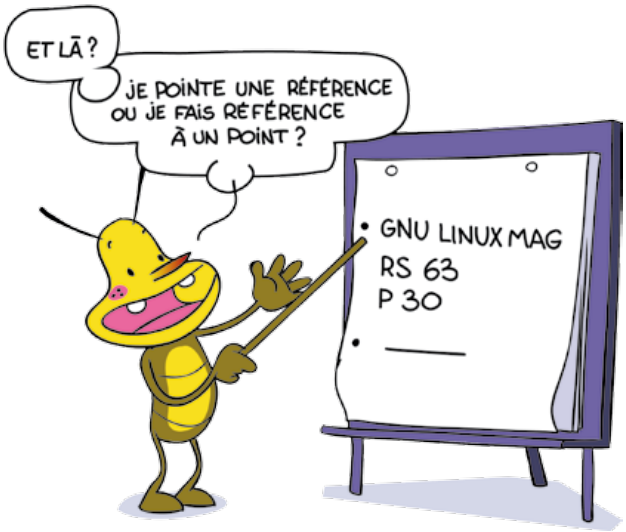
La deuxième syntaxe est une référence au code de la fonction **print**. Tant qu'il n'y a pas de parenthèses, l'appel n'est pas exécuté. On pourrait même stocker cette référence dans une variable pour y faire appel plus tard :

Terminal

```
>>> ecrire = print
>>> ecrire("Hello")
Hello
```

La variable **ecrire** est une référence à la fonction **print**. C'est ce que l'on appelle **un pointeur** : la variable pointe vers une zone de la mémoire qui contient un certain code (ici une fonction).

Le problème de ce mécanisme, lorsqu'on l'utilise pour intercepter des événements, est que l'on ne peut pas transmettre de paramètres à la fonction puisqu'on ne doit donner que sa référence et non son appel direct...



Pour contourner ce problème, nous allons utiliser des fonctions anonymes ou lambda fonctions. Le principe consiste à créer une référence à une fonction construite par appel à une fonction contenant des paramètres. Exemple :

Terminal

```
>>> f = lambda txt = "Hello" : print(txt)
>>> f
<function <lambda> at 0x7f64dcfb7160>
>>> f()
Hello
```

f est une référence vers une fonction créée en appelant la fonction **print** avec le paramètre **txt**. Lorsque nous exécutons **f** à l'aide des parenthèses, nous obtenons l'affichage du contenu de la variable **txt**.

À retenir

Une fonction anonyme ou lambda fonction est une référence à une fonction, mais qui ne porte pas de nom. Ce mécanisme permet de définir et de transmettre sous forme de paramètre une fonction à une autre fonction, le tout en une seule ligne.

Concrètement, dans la fonction **init_touches**, qui va définir les actions à effectuer en fonction des touches sur lesquelles l'utilisateur va appuyer, nous allons utiliser des fonctions anonymes pour déplacer le personnage et fermer la fenêtre graphique :

Fichier

```
def init_touches(fenetre, canvas, lab, pos_perso, perso):
    """
        Initialisation du comportement des touches du clavier

        canvas    : canevas où afficher les sprites
        lab       : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso     : sprite représentant le personnage

        Pas de valeur de retour
    """
    fenetre.bind("<Right>", lambda event, can = canvas, l = lab,
        pos = pos_perso,
        p = perso: deplacement(event, can, "right", l, pos, p))
    # Autres cas
    # ...
    fenetre.bind("<Escape>", lambda event, fen = fenetre : destroy(event, fen))
```

Appels de fonctions anonymes

Les fonctions **deplacement** et **destroy** vont être appelées lors de l'appui sur la flèche droite ou la touche [Échap]. Le premier paramètre est l'objet événement transmis automatiquement à la fonction.

Fichier

```
def deplacement(event, can, dep, lab, pos_perso, perso):
    """
        Déplacement du personnage

        event     : objet décrivant l'événement ayant déclenché l'appel à cette fonction
        can       : canevas où afficher les sprites
        dep       : type de déplacement("up", "down", "left", ou "right")
        lab       : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso     : sprite représentant le personnage

        Pas de valeur de retour
    """
    # Calcul de la taille du labyrinthe
    n_cols   = len(lab[0])
    n_lignes = len(lab)
    pos_col, pos_ligne = [pos_perso[0], pos_perso[1]]

    # Déplacement vers la droite
    if dep == "right":
        pos_col += 1

    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
       pos_col > (n_cols - 1):
        return None

    # Si le déplacement est possible sur une case vide
    if lab[pos_ligne][pos_col] == " ":
        can.coords(perso, pos_col + pos_col * 30, pos_ligne + pos_ligne * 30)
        del pos_perso[0]
        del pos_perso[0]
        pos_perso.append(pos_col)
        pos_perso.append(pos_ligne)

def destroy(event, fenetre):
    """
        Fermeture de la fenêtre graphique
```

Modification de la liste pos_perso pour que les nouvelles valeurs soient accessibles depuis les autres fonctions

Fichier

```
        event   : objet décrivant l'événement ayant déclenché l'appel à cette
                  fonction
        fenetre : fenêtre graphique

        Pas de valeur de retour
    """
    fenetre.destroy()
```

Appel de la méthode destroy de la fenêtre graphique (à ne pas confondre avec la fonction dans laquelle on se trouve)

Ici, il a fallu modifier la structure de la fonction chargée du déplacement du personnage : les fonctions sont appelées suite à des événements liés à des fonctions et nous ne pouvons pas récupérer de valeurs en retour. Nous modifions donc directement la liste **pos_perso**, de manière à ce que les modifications soient accessibles depuis n'importe quelle fonction, qu'elles restent persistantes.

L'instruction **del** permet de supprimer des éléments d'une liste (en les décalant). Donc si, sur une liste de deux éléments, on supprime le premier élément, il ne reste plus qu'une liste d'un seul élément et, pour qu'elle soit vide, il faut de nouveau supprimer le premier élément.

L'appel de la fonction **init_touches** se fait dans le programme principal en lui passant en paramètre la fenêtre graphique, le canevas, la liste contenant le labyrinthe, la position et le sprite du personnage :

Fichier

```
Lab.init_touches(fenetre, canvas, level, pos_perso, sprite_perso)
```

CONCLUSION

Je n'ai volontairement pas codé en mode graphique le jeu tel que nous l'avions terminé hier. À partir de tous les éléments que nous avons pu voir cette semaine, vous devriez être capable d'écrire le code complet de notre jeu en mode graphique. N'oubliez pas de récupérer les variables associées aux sprites qui peuvent être déplacés (ou enlevés) et d'afficher la barre des scores en utilisant la méthode **create_text** qui, comme **create_image**, prend en premier paramètre la position du texte suivie du texte à afficher.

N'hésitez pas à utiliser l'aide en ligne ou les ressources du site **python.org** pour vous aider et surtout ne vous découragez pas : il faut environ trois mois à un informaticien pour apprendre un nouveau langage. Seules la pratique et la correction des erreurs vous permettront de comprendre réellement ce que vous faites. ▉

Pour récapituler :

- Un **objet** est une structure permettant une programmation modulaire et réutilisable. La définition d'un objet se fait dans une **classe** ;
- Un **attribut** est une variable attachée à un objet et permettant de le définir ;
- Une **méthode** est une fonction attachée à un objet et qui a accès à ses attributs ;
- Une **instance** est une variable créée en se servant d'une classe pour modèle ;
- **del** est une instruction permettant de supprimer des éléments dans une liste.

Résumé code :

Le fichier **Lab.py** contient les fonctions principales du jeu :

```
001: import random
002: from tkinter import *
003:
004:
005: def charge_labyrinthe(nom):
006:     """
007:         Charge le labyrinthe depuis le fichier nom.txt
008:
009:         nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)
010:
011:         Valeur de retour :
012:             - une liste avec les données du labyrinthe
013:     """
014:     try:
015:         fic = open(nom + ".txt", "r")
016:         data = fic.readlines()
017:         fic.close()
018:     except IOError:
019:         print("Impossible de lire le fichier {}.txt".format(nom))
020:         exit(1)
021:
022:     for i in range(len(data)):
023:         data[i] = data[i].strip()
024:
025:     return data
026:
027:
028: def affiche_labyrinthe(lab, fenetre, size_sprite, pos_perso):
029:     """
030:         Affichage d'un labyrinthe.
031:
032:         lab : Variable contenant le labyrinthe
033:         fenetre : Fenêtre graphique
034:         size_sprite : Taille des sprites en pixels
035:         pos_perso : Liste contenant la position du personnage
036:
037:         Valeur de retour:
038:             Tuple contenant le canevas, le sprite du personnage et un
039:             dictionnaire des images utilisées pour les sprites
040:     """
041:     can = Canvas(fenetre, width = 600, height = 600)
042:
043:     photo_wall     = PhotoImage(file = "sprites/wall.gif")
044:     photo_treasure = PhotoImage(file = "sprites/treasure.gif")
045:     photo_ennemy   = PhotoImage(file = "sprites/ennemy.gif")
046:     photo_exit     = PhotoImage(file = "sprites/exit.gif")
047:     photo_hero     = PhotoImage(file = "sprites/hero.gif")
048:
049:     n_ligne = 0
050:     for ligne in lab:
051:         n_col = 0
052:         for car in ligne:
053:             # Murs
054:             if car == "+" or car == "-" or car == "|":
055:                 can.create_image(n_col + n_col * size_sprite,
056:                     n_ligne + n_ligne * size_sprite, anchor = NW,
057:                     image = photo_wall)
058:             # Trésors
059:             if car == "1" or car == "2" or car == "3":
060:                 can.create_image(n_col + n_col * size_sprite,
```

```
061:                         n_ligne + n_ligne * size_sprite, anchor = NW,
062:                         image = photo_treasure)
063:             # Ennemis
064:             if car == "$":
065:                 can.create_image(n_col + n_col * size_sprite,
066:                     n_ligne + n_ligne * size_sprite, anchor = NW,
067:                     image = photo_ennemy)
068:             # Sortie
069:             if car == "O":
070:                 can.create_image(n_col + n_col * size_sprite,
071:                     n_ligne + n_ligne * size_sprite, anchor = NW,
072:                     image = photo_exit)
073:             n_col += 1
074:         n_ligne += 1
075:
076:     # Affichage du personnage
077:     sprite_hero = can.create_image(pos_perso[0] + pos_perso[0] * size_sprite,
078:         pos_perso[1] + pos_perso[1] * size_sprite, anchor = NW,
079:         image = photo_hero)
080:
081:     can.pack()
082:
083:     return (can, sprite_hero, {
084:       "hero"     : photo_hero,
085:       "wall"     : photo_wall,
086:       "treasure" : photo_treasure,
087:       "ennemy"   : photo_ennemy,
088:       "exit"     : photo_exit})
089:
090:
091: def deplacement(event, can, dep, lab, pos_perso, perso):
092:     """
093:         Déplacement du personnage
094:
095:         event     : objet décrivant l'événement ayant déclenché l'appel à cette
096:                     fonction
097:         can       : canevas où afficher les sprites
098:         dep       : type de déplacement("up", "down", "left", ou "right")
099:         lab       : liste contenant le labyrinthe
100:         pos_perso : position courante du personnage
101:         perso     : sprite représentant le personnage
102:
103:         Pas de valeur de retour
104:     """
105:     # Calcul de la taille du labyrinthe
106:     n_cols   = len(lab[0])
107:     n_lignes = len(lab)
108:     pos_col, pos_ligne = [pos_perso[0], pos_perso[1]]
109:
110:     # Déplacement vers la droite
111:     if dep == "right":
112:         pos_col += 1
113:
114:     # Teste si le déplacement conduit le personnage en dehors de l'aire
115:     # de jeu
116:     if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes - 1) or \
117:        pos_col > (n_cols - 1):
118:         return None
119:
120:     # Si le déplacement est possible sur une case vide
121:     if lab[pos_ligne][pos_col] == " ":
122:         can.coords(perso, pos_col + pos_col * 30, pos_ligne + pos_ligne * 30)
123:         del pos_perso[0]
124:         del pos_perso[0]
125:         pos_perso.append(pos_col)
126:         pos_perso.append(pos_ligne)
```

```
127:
128:
129: def destroy(event, fenetre):
130:     """
131:         Fermeture de la fenêtre graphique
132:
133:         event   : objet décrivant l'événement ayant déclenché l'appel à cette
134:                   fonction
135:         fenetre : fenêtre graphique
136:
137:         Pas de valeur de retour
138:     """
139:     fenetre.destroy()
140:
141:
142: def init_touches(fenetre, canvas, lab, pos_perso, perso):
143:     """
144:         Initialisation du comportement des touches du clavier
145:
146:         canvas    : canevas où afficher les sprites
147:         lab       : liste contenant le labyrinthe
148:         pos_perso : position courante du personnage
149:         perso     : sprite représentant le personnage
150:
151:         Pas de valeur de retour
152:     """
153:     fenetre.bind("<Right>", lambda event, can = canvas, l = lab,
154:         pos = pos_perso,
155:         p = perso: deplacement(event, can, "right", l, pos, p))
156:     # ...
157:     fenetre.bind("<Escape>", lambda event, fen = fenetre : destroy(event, fen))
```

Le fichier **GLMF_Game.py** est le fichier principal qui va utiliser le module **Lab** pour lancer le jeu :

```
01: #!/usr/bin/python3
02:
03: import Lab
04: from tkinter import *
05:
06: if __name__ == "__main__":
07:     # Initialisation du personnage
08:     perso          = "X"
09:     pos_perso      = [1, 1]
10:     tresor         = "#"
11:     n_levels_total = 20
12:     data = {
13:         "po"    : 0,
14:         "pv"    : 25,
15:         "level" : 1
16:     }
17:     size_sprite = 29
18:
19:     # Initialisation de l'affichage graphique
20:     fenetre = Tk()
21:     fenetre.title("GLMF  Game")
22:
23:     # Lancement de la partie
24:     level = Lab.charge_labyrinthe("level_1")
25:
26:     (canvas, sprite_perso, photos) = Lab.affiche_labyrinthe(level, fenetre,
27:         size_sprite, pos_perso)
28:     Lab.init_touches(fenetre, canvas, level, pos_perso, sprite_perso)
29:
30:     # Boucle événementielle
31:     fenetre.mainloop()
```

INDEX DES INSTRUCTIONS

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

__NAME__

Variable spéciale permettant de détecter si un fichier est exécuté directement ou chargé en tant que module. Typiquement, on utilise cette variable pour déterminer le bloc de programme principal.

Fichier

```
if __name__ == "__main__":
    …
```

AND

Opérateur logique permettant de tester sur deux conditions si l'une et l'autre sont vraies.

Condition 1	Condition 2	Condition 1 and Condition 2
False	False	False
False	True	False
True	False	False
True	True	True

Fichier

```
if a % 2 == 0 and a < 10:
    print("a est un entier pair inférieur à 10")
```

BREAK

Pour sortir d'une structure de boucle avant la fin normale de la boucle, il faut employer l'instruction `break`.

Fichier

```
for i in range(10):
    print(i)
    if i == 4:
        break
```

L'exécution de ce code donnera :

Terminal

```
0
1
2
3
4
```

CLASS

Mot-clé permettant de définir une classe, la manière de construire un objet.

Fichier

```
class Personne(object):
    def __init__(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom

    def __str__(self):
        return self.prenom + " " + self.nom
```

CLOSE

Après avoir travaillé sur un fichier (en lecture ou en écriture), il faut refermer le descripteur de fichier à l'aide de `close`. Pour un exemple, vous pourrez vous reporter à la définition des fonctions `open`, `write`, `read` et `readline`.

DEF

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

Fichier

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

DEL

Instruction permettant de supprimer un élément.

Terminal

```
>>> a = 1
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
>>> del l[1]
>>> l
[1, 3]
```

DIR

Fonction renvoyant une liste de tous les attributs et méthodes de l'objet passé en paramètre.

Terminal

```
>>> import math
>>> dir(math)
[ …, 'pi', 'pow', 'radians', 'sin', … ]
```

ELIF

Instruction permettant d'effectuer des tests à la chaîne.

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`) pour proposer une alternative au traitement correspondant à la vérification de la condition.

Fichier

```
a = 5

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

EXCEPT

Dans une structure de traitement des exceptions `try`/`except`, définit les exceptions à intercepter et le code à exécuter.

Fichier

```
try:
    n = int(input("Donnez un entier : "))
except ValueError:
    print("Ceci n'est pas un entier")
```

EXIT

Fonction permettant d'interrompre l'exécution d'un script en renvoyant un code indiquant la raison de l'arrêt : `0` pour un arrêt normal, et un code compris entre `1` et `255` pour une erreur (la signification du code est déterminée par le développeur).

Fichier

```
exit(0)
```

FINALLY

Dans le cadre d'un traitement des exceptions par `try`/`except`, un bloc `finally` sera exécuté qu'il y ait eu ou non une erreur.

Fichier

```
try:
    n = int(input("Donnez un entier : "))
```

Fichier

```
except ValueError:
    print("Ceci n'est pas un entier")
finally:
    print("Fin du programme")
```

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

Méthode permettant de formater une chaîne de caractères : les caractères **{}** définissent des zones d'insertion à l'intérieur d'une chaîne de caractères, et c'est la méthode **format** qui indique les valeurs à insérer.

Fichier

```
a = 5
chaine = "Valeur de a : {}".format(a)
```

FROM

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

Fichier

```
from math import cos, pi
from random import *

print(cos(pi/2))
print(randint(1, 10))
```

HELP

Fonction permettant d'obtenir de l'aide dans le shell Python. On obtiendra des informations sur l'instruction passée en paramètre.

Terminal

```
>>> help(print)
Help on built-in function print  in module bultins:
print(...)
    print(value, …, sep=' ', end='\n', file=sys.stdout)
    Prints the values  to a stream,  or to sys.stdout by default.
    ...
```

INDEX

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.

Fichier

```
import math
print(math.pi)
```

IN

Mot-clé permettant de savoir si un élément est présent dans une liste.

Fichier

```
if "linux" in ["linux", "windows", "mac"]:
    print("Trouvé !!")
```

Associé à l'instruction `for`, permet de parcourir tous les éléments d'une liste.

Fichier

```
for elt in ["linux", "windows", "mac"]:
    print(elt)
```

Ce code produit l'affichage suivant :

Terminal

```
linux
windows
mac
```

INPUT

Fonction permettant de stocker une saisie effectuée au clavier par un utilisateur.

Fichier

```
saisie_utilisateur = input("Veuillez saisir des données : ")
```

INT

Conversion d'une valeur sous forme d'un entier.

Fichier

```
chaine = "123"
entier = int(chaine)
```

LAMBDA

Mot-clé utilisé pour créer une fonction anonyme, ou « lambda fonction ». Si elle n'est pas stockée dans une variable, la fonction anonyme est détruite après utilisation.

Terminal

```
>>> carre = lambda c, c ** 2
>>> carre(3)
9
```

LIST

Fonction de conversion sous forme de liste.

Terminal

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

NONE

Mot-clé particulier indiquant une absence de valeur. En cas d'absence d'appel à `return` dans une fonction, c'est ce mot-clé qui sera renvoyé.

NOT

Opérateur logique inversant une condition. Si une condition est vraie (`True`) alors `not` renverra le résultat inverse soit faux (`False`) et réciproquement.

Terminal

```
>>> val = True
>>> print(not val)
False
```

OPEN

Ouverture d'un fichier texte pour lire ou écrire des données. La fonction `open` permet de créer un descripteur de fichier qui permettra de le manipuler.

Fichier

```
fic = open("mon_fichier.txt", "r")
print("Contenu du fichier :")
print(fic.read())
fic.close()
```

OR

Opérateur logique permettant de tester sur deux conditions si l'une ou l'autre est vraie.

Condition 1	Condition 2	Condition 1 or Condition 2
False	False	False
False	True	True
True	False	True
True	True	True

Fichier

```
if a == 0 or a > 100:
    print("a est égal à zéro ou alors est strictement supérieur à 100")
```

PRINT

Fonction d'affichage de texte. Les paramètres **sep** et **end** permettent de définir les caractères de séparation des différentes valeurs passées en paramètre et de redéfinir les caractères de fin de ligne. Par défaut, le caractère de séparation (**sep**) est un espace, et le caractère de fin de ligne (**end**) est le retour à la ligne.

Fichier

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

RANGE

Fonction générant une liste d'entiers à partir d'un, deux ou trois paramètre(s) :

⇨ **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**

Lorsque la valeur du **pas** n'est pas définie, elle vaut **1** (valeur par défaut). En association avec la fonction de conversion **list**, on peut obtenir une véritable liste :

Terminal

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

Classiquement, on utilise **range** dans des boucles.

Fichier

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

READ

Fonction permettant de lire des données dans un fichier. Sans paramètre, lit toutes les données du fichier et les renvoie sous la forme d'une chaîne de caractères (voir la fonction **open**). Si on spécifie un paramètre, lit le nombre de caractères spécifiés.

Fichier

```
fic = open("mon_fichier.txt", "r")
print("10 Premiers caractères du fichier :")
print(fic.read(10))
fic.close()
```

READLINE

Fonction permettant de lire une ligne dans un fichier.

Fichier

```
fic = open("mon_fichier.txt", "r")
print("Première ligne du fichier :")
print(fic.readline)
fic.close()
```

READLINES

Fonction permettant de lire des données dans un fichier et qui les renvoie sous la forme d'une liste de chaînes de caractères (chaque élément de la liste correspond à une ligne du fichier se terminant par un retour à la ligne).

RETURN

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

Fichier

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

STR

Conversion d'une valeur sous forme de chaîne de caractères.

Fichier

```
a = 1
chaine = str(a)
```

TRY

Permet de commencer un bloc où les exceptions seront interceptées pour être traitées dans un ou des bloc(s) **except** (voir l'instruction **except**).

TUPLE

Fonction de conversion d'un élément sous forme de tuple.

Terminal

```
>>> tuple("abcd")
("a", "b", "c", "d")
```

WHILE

Instruction permettant de créer une boucle : tant que la condition qui suit le **while** est valide, le bloc de code suivant sera exécuté.

Fichier

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

WRITE

Instruction permettant d'écrire dans un fichier.

Fichier

```
fic = open("mon_fichier.txt", "w")
fic.write("Texte à écrire dans le fichier")
fic.close()
```

INDEX DES NOTIONS

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

ATTRIBUT

En programmation orientée objet (POO), un attribut est une variable attachée à la structure permettant de définir un objet. En Python, les attributs sont préfixés par le mot-clé `self` pour bien montrer qu'ils appartiennent à l'objet courant.

Fichier

```
class Livre(object):
    def __init__(self, titre):
        # Attribut d'un livre
        self.titre = titre
```

BLOC

Un bloc de code est un ensemble d'instructions exécutées les unes à la suite des autres et groupées (un peu comme s'il s'agissait d'une seule instruction). En Python, les blocs sont reconnaissables à leur indentation : toutes les instructions d'un bloc sont alignées.

Fichier

```
for i in range(10):
    if i % 2 == 0:
        # Début bloc
        print("Valeur de i :", i)
        # Fin du bloc
```

BOUCLE

Une boucle permet de répéter plusieurs fois un traitement, en faisant éventuellement varier certains paramètres. En Python, on utilise les instructions `for` et `while`.

CLASSE

En POO, une classe est la définition d'un objet, le modèle à partir duquel tous les objets du même type pourront être créés.

ERREUR

Une erreur apparaît lorsque le code ne peut pas être analysé, exécuté correctement.
Il existe de nombreux types d'erreurs, les plus courantes étant les erreurs de syntaxe : mauvaise indentation, oubli d'un paramètre lors de l'appel d'une fonction, etc.

EXCEPTION

En POO, le mécanisme de gestion des erreurs est basé sur les exceptions : des objets que l'on crée pour donner des informations sur l'erreur, et qui sont récupérés dans une structure `try` / `except`.

FONCTION

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

Terminal

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

INSTANCE

Une instance est la réalisation concrète d'un objet défini par une classe. C'est avec l'instance que l'on peut interagir pour modifier des valeurs.

Terminal

```
>>> from Livre import *
>>> l = Livre("GNU Linux Magazine")
```

`l` est une instance de la classe `Livre`.

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

En POO, une méthode est une fonction particulière qui est rattachée à un objet et a accès à ses attributs.

Fichier

```
class Livre(object):
    def __init__(self, titre):
        self.titre = titre

    # Méthode de la classe Livre
    def getTitre(self):
        return self.titre
```

La méthode `__init__` est une méthode particulière appelée « constructeur ».

MODULE

Un module est un fichier Python contenant des instructions. Lors de son chargement à l'aide des instructions `import` ou `from ... import`, les instructions du module sont exécutées et les fonctions ou les objets qu'il propose deviennent accessibles.

Terminal

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

OBJET

Un objet est une structure définissant un élément de programmation à l'aide de valeurs (les attributs) et d'actions (les méthodes).

TEST CONDITIONNEL

Un test conditionnel permet de créer un embranchement dans l'exécution d'instructions. En fonction de la valeur d'une condition booléenne (vraie ou fausse), on va exécuter tel ou tel bloc de code.

Terminal

```
>>> a = 2
>>> if a == 2:
...     print("Vrai")
... else:
...     print("Faux")
...
Vrai
```

VARIABLE

Une variable est un espace mémoire contenant une valeur et une étiquette pour pouvoir y accéder. En Python les variables ne sont pas typées, c'est-à-dire que l'on peut y stocker n'importe quelle sorte de valeur : entier, chaîne de caractères, etc.

Terminal

```
>>> ma_variable = 10
```

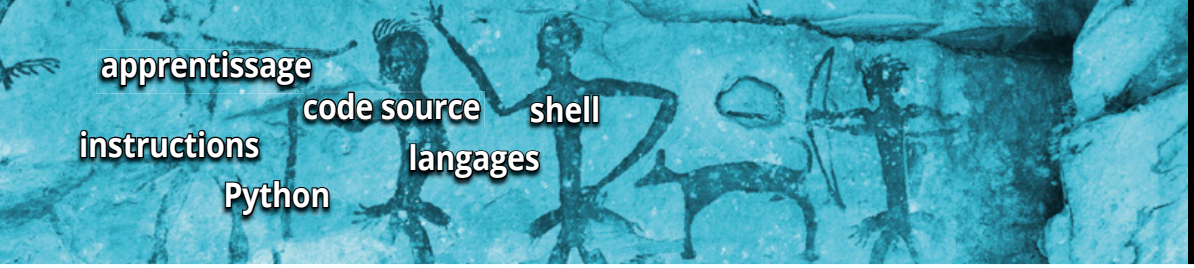
LINUX MAGAZINE / FRANCE · LINUX PRATIQUE · LINUX ESSENTIEL · MISC Multi-System & Internet Security Cookbook · Open Silicium

VENEZ DÉCOUVRIR NOS GUIDES !

DÉJÀ PARUS !

DISPONIBLES CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

boutique.ed-diamond.com



INTRODUCTION : Préparer ses armes

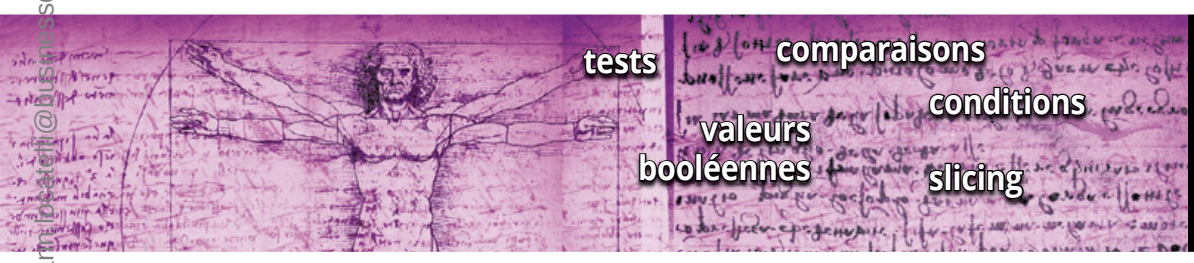
Choisir les bons outils pour s'initier à la programmation

JOUR 1 :

Afficher des caractères

JOUR 2 :

Saisir des données



JOUR 3 :

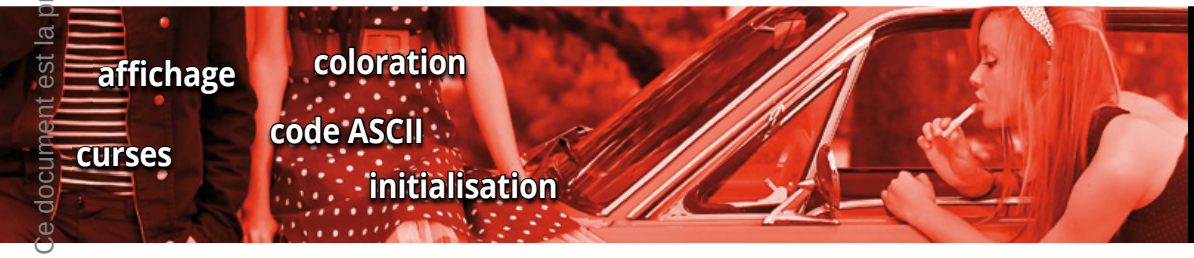
Mettre en place les tests et les conditions

JOUR 4 :

Gérer les fichiers et modules

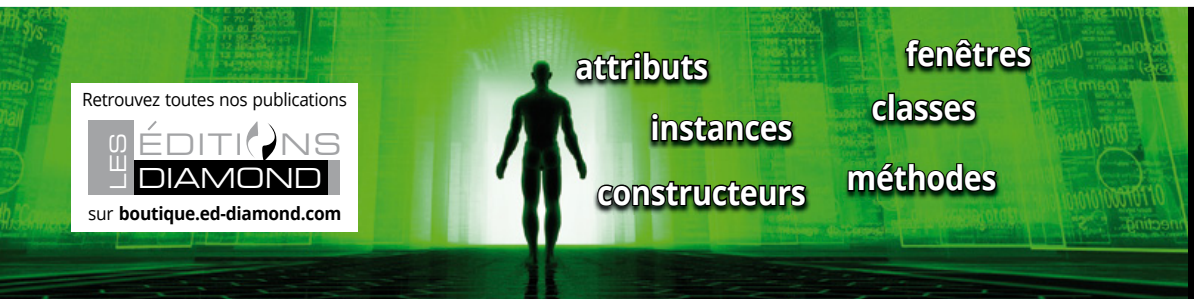
JOUR 5 :

Traiter les erreurs et les exceptions



JOUR 6 :

Améliorer l'interface en mode console



JOUR 7 :

Créer une interface graphique