Initiation à la programmation

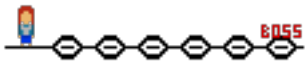
Bastien Gorissen & Thomas Stassin

Année 2016

Table des matières

Chapitre 1

Level 1



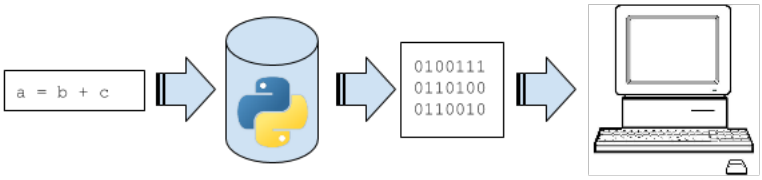
1.1 Qu'est-ce qu'un langage de programmation ?

> Un langage de programmation est une convention pour donner des ordres à un ordinateur. Ce n'est pas censé être obscur, bizarre et plein de pièges subtils. Ca, ce sont les caractéristiques de la magie.
>
> *Dave Small*

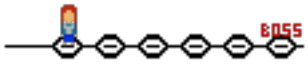
Un langage de programmation est un moyen d'intéragir avec l'ordinateur afin de lui donner des instructions, le langage qui est *intelligible*, sera *interprété* afin d'être compris par la machine.

1.1.1 Commentaires

Si vous ouvrez le script `run_game.py`, vous pourrez voir que certaines lignes commencent par `#`. Celles-ci seront ignorées par l'ordinateur. Ce sont des *commentaires* destinés à clarifier le code.



1.1.2 Stage 1-1

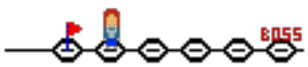


Exécuter le script se trouvant dans le dossier `lv1/run_game.py`

1.2 Commander à l'ordinateur

Pour "commander" l'ordinateur, on lui donne des *instructions*, le plus souvent, ces instructions seront regroupées dans un *script*.

1.2.1 Stage 1-2



Dans l'exercice précédent, nous avons exécuté le script `run_game.py`, ce script est rempli d'instructions, dont celles-ci :

```
dungeon_size = (5, 5)
game = world.Game(dungeon_size)
game.run()
```

Que ce passerait-il si on modifiait l'instruction qui définit la grandeur du donjon dans le script ? Changez les données de dimension du donjon dans le script et observez ce qu'il se passe lors de l'exécution du script.

1.2.2 À retenir

Python est un *langage sensible à la casse*[1], ce qui veut dire qu'il fait la différence entre les majuscules et les minuscules. Autrement dit `A` sera différent de `a` et `world` différent de `World`.

Donc si je remplace `world` par `World` dans le script précédent, il générera une erreur.

```
dungeon_size = (5, 5)
game = World.Game(dungeon_size)
game.run()
```

Vous devriez obtenir un message ressemblant à "`NameError: name 'World' is not defined`"

1.3 Les instructions de sortie

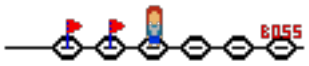
Il existe plusieurs sortes d'instructions, l'une d'elles sont les instructions de sortie. Une instruction de sortie envoie vers une "sortie" ce qu'on lui donne. Une des sorties les plus couramment utilisée est la console et avec Python, l'instruction de sortie vers la console est `print`.

Et donc voici le classique, mais indémodable "Hello World" en Python :

```
print("Hello World!")
```

1. *Case sensitive* en anglais.

1.3.1 Stage 1-3



Lorsque le donjon est créé, signalez-le par un message dans la *console*.

1.4 Les variables

Souvent, il sera utile de stocker certaines valeurs tout au long de l'exécution de votre code. Par exemple pour stocker la valeur d'un calcul, ou même afin de réutiliser ses valeurs plusieurs fois. Pour stocker des valeurs, on utilise des *variables*.

La *variable* est un moyen de stocker une valeur quelconque (un nombre, du texte, voire même des *objets* plus complexes) dans la mémoire du programme.

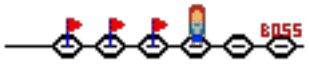
Dans le cas de notre jeu, le donjon, ainsi que le héro sont contenus chacun dans une *variable*.

```
dungeon_size = (5, 5)
game = world.Game(dungeon_size)
hero = Game.hero
```

En Python, l'*affectation* d'une valeur se fait avec l'opérateur =.

Dans `a=3`, on affecte `3` à la variable `a` [2]

1.4.1 Stage 1-4



Faites en sorte de stocker les dimensions du donjon dans des variables (par exemple `length` et `width` [3]).

2. En Python il suffit d'affecter une valeur à une variable pour qu'elle commence à exister, ce n'est pas vrai pour la plupart des langages (comme le C# par exemple).

3. Il est courant d'utiliser des noms de variables en anglais. La programmation est un monde très anglophone.

1.5 Opérations sur une chaîne de caractères

Il vous sera parfois utile de savoir manipuler des chaînes de caractères, l'exemple classique est dans une ligne de dialogue, où l'on voudrait dire au héros le nombre d'ennemis qu'il lui reste à tuer avant de finir la quête. Or, lorsque que vous codez, vous ne connaissez pas le nombre d'ennemis. Vous avez sûrement stocké cette information dans une variable et donc vous allez devoir intégrer cette variable à votre ligne de dialogue.

1.5.1 Conversion

Il y un moyen facile de convertir une variable en chaine de caractères[4], il suffit d'utiliser la fonction `str`.

Si je voulais convertir la variable `nbr_enemies` qui contient le chiffre `3` je procéderais comme suit :

```
nbr_enemies = 3
nbr_enemies = str(nbr_enemies)
```

A la fin de l'exécution de ce petit script, `nbr_enemies` ne vaut plus `3` mais `"3"`.

1.5.2 Concaténation

Le fait de coller deux chaînes de caractères l'une derrière l'autre porte un nom : *la concaténation*. En Python, l'opérateur pour *concaténer* deux variables ensemble est l'opérateur `+`.

Par exemple, si je voulais concaténer la variable contenant le nom du héro avec un message cela donnerait :

```
message = ", il te reste des ennemis à tuer."
message = hero_name + message
```

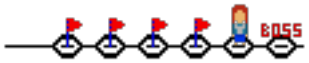
Donc si le héro se nomme Brutor, la chaîne de caractères en fin de script sera égale à `"Brutor, il te reste des ennemis à tuer."`

4. En Python les chaines de caractères ont le *type* `str` pour le mot anglais *string*, qui veut dire chaîne

On peut même aller plus loin en mêlant la *conversion* et la *concaténation* en indiquant aussi dans le message le nombre d'ennemis qu'il reste.

```
    message = ", il te reste " + str(nbr_enemies)"
    message = message + " ennemis à tuer."
    message = hero_name + message
```

1.5.3 Stage 1-5



Lorsque le donjon est créé, indiquer dans un message à la console la longueur et la largeur de celui-ci.

1.6 Les fonctions et les méthodes, première approche

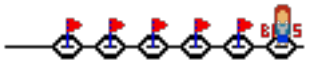
Pour terminer ce premier level, on va mettre des mots sur deux choses que l'on a vues précédemment. Les *fonctions globales* et les *méthodes*. Nous avons déjà vu plusieurs fonctions dans les parties précédentes :

- `print`
- `str`

Ces fonctions sont dîtes "globales". Ce type de fonctions n'est pas lié à un type d'*objet*.

Il existe aussi une série de fonctions liées à des *objets*, par exemple la fonction *run* de la variable *game*. On appelle ces fonctions des *méthodes* et elles sont liées à un type d'objets en particulier.

1.6.1 Stage 1-6



La variable du héros, sobrement appelée `hero` est du type `Hero`, c'est un type de variable que nous avons construit pour le jeu [5]. Nous avons donné trois *méthodes* à cette variable :

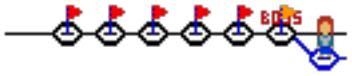
`turn_left :` ce qui fait tourner le héros à gauche.

`turn_right :` ce qui fait tourner le héros à droite.

`move :` ce qui fait avancer le héros d'une "case".

Essayez de faire bouger le héros grâce à ces trois méthodes.

1.7 Bonus Stage



Pour compléter le stage bonus il faut que vous arriviez à amener le héros sur la sortie du donjon. La sortie est représentée par la case bleue en haut à gauche de la pièce. Le donjon doit au moins avoir une longueur et une largeur de 7.

5. Nous verrons plus tard qu'il est possible de définir ses propres types et de leur donner les caractéristiques que l'on désire.

Chapitre 2

Level 2 - Le labyrinthe

Savoir commander l'ordinateur est une chose, encore faut-il arriver à lui faire faire des choses intelligentes. Heureusement, tous les languages de programmation proposent une série d'outils qui vont permettre de donner des instructions complexes à la machine.

2.1 Comparaison n'est pas raison

L'un des éléments récurrents en programmation est le besoin d'effectuer des comparaisons, et plus globalement d'évaluer si certains *conditions* sont vérifiées ou pas. Le sujet est vaste, mais tout bon programmeur se doit de savoir écrire les bonnes conditions s'il veut que son code se comporte comme il le veut.

2.1.1 Les `booléens`

Outre manipuler des nombres, des chaînes de caractères et autres listes, il y a un autre type de données omniprésent en Python : les *booléens*.

La définition d'une variable booléenne est une valeur qui peut être soit *vraie*, soit *fausse*. En Python, on représente la valeur vraie par `True`, et la valeur fausse par `False`.

Comme les booléens sont un type de variables comme les autres, vous avez le droit de manipuler `True` et `False` comme n'impore quelle autre valeur :

```
a = True
b = False
print(a)
print(b)
```

Tout l'intérêt de ce type de variables va résider dans les opérations spécifiques que l'on peut leur appliquer, et qui permettent d'effectuer des choix par rapport à certains éléments du code.

2.1.2 Les comparaisons

Une façon d'obtenir des valeurs booléennes est d'utiliser les opérateurs dit de comparaison. Vous les connaissez probablement pratiquement tous :

`<` **Plus petit que** Teste si la valeur à gauche de `<` est plus petite que celle à droite.

`<=` **Plus petit ou égal à** Teste si la valeur à gauche de `<=` est plus petite OU égale à celle à droite.

`>` **Plus grand que** Teste si la valeur à gauche de `>` est plus grande que celle à droite.

`>=` **Plus grand ou égal à** Teste si la valeur à gauche de `>=` est plus grande OU égale à celle à droite.

`==` **Egal à** Teste si la valeur à gauche de `==` est égale à celle à droite.

`!=` **Différent de** Teste si la valeur à gauche de `!=` est différente de celle à droite.

Rien de bien sorcier, et la comparaison renvoie `True` si la condition est remplie, et `False` sinon.

A noter, l'opérateur "égal" (`==`) n'est pas simplement `=`, puisque `=` est déjà "pris" par l'affectation d'une variable.

Voici quelques exemples de comparaisons, avec en commentaire la valeur renvoyée :

```
a = 3
b = 5
c = 3

a > 5      # False
```

```
a < b      # True
c == a     # True
a <= b-2   # True
a != c     # False
4 >= 1     # False
```

Petite astuce : on peut comparer autre chose que des nombres, bien sûr. Python définit un *ordre* sur beaucoup de types d'objets, par exemple les *string*. `"message" < "texte"` renverra `True`, car l'ordre est "lexicographique" pour le texte (plus ou moins équivalent à l'ordre alphabétique).

On peut évidemment affecter le résultat d'une comparaison à une variable :

```
a = 3
b = 5

condition = a < b
print(condition)   # Imprime True
```

Voyons maintenant ce qu'on peut faire de beau avec ces fameux booléens, avant de voir les applications vraiment pratiques !

2.1.3 L'algèbre booléenne pour les nuls

Les comparaisons et les valeurs booléennes, aussi intéressantes qu'elles soient (et elles le sont dans de nombreux cas), ne révèlent leur vraie puissance que lorsqu'elle sont combinées via une série de règles inventées par Mr Boole [1]. On appelle ces règles l'algèbre booléenne.

Ce cours n'a pas pour vocation de faire de vous des pros de la logique, mais vous rencontrerez très souvent les opérations suivantes...

La négation

L'opération de négation, consiste simplement à inverser une valeur booléenne. C'est probablement l'opération la plus courante, et en Python, on la note `not`.

```
not True     # == False
not False    # == True
```

1. Véridique, un nom pareil, ça ne s'invente pas.

La conjonction (ET)

L'opération de conjonction, consiste à vérifier si deux conditions sont toutes les deux vraies. Egalement un cas très courant, et notée en Python `and`. Son comportement est très similaire à ce dont on pourrait s'attendre :

```
True and True      # == True
True and False     # == False
False and True     # == False
False and False    # == False
```

En résumé, `a and b` ne renvoie `True` que si et seulement si `a` est vrai, ET `b` est également vrai. Dans tous les autres cas, le résultat du `and` vaut `False`.

La disjonction (OU)

L'opération de disjonction est un peu le pendant du ET, et représente un OU. C'est-à-dire, on va vérifier si soit la première condition, soit la seconde, soit les deux, sont vraies. En Python, on la note `or`. Son comportement est :

```
True or True      # == True
True or False     # == True
False or True     # == True
False or False    # == False
```

En résumé, `a or b` renvoie `True` que si au moins `a` est vrai, OU `b` est vrai. Le seul cas où le résultat du `or` vaut `False` est si les deux conditions sont fausses.

L'utilité de tout ça...

Avec ces quelques règles, qui finalement sont assez logiques, on est capable de réaliser des opérations très complexes, qui vont nous permettre de résoudre énormément de cas. Par exemple, on pourrait écrire une condition du style : "Le héros à moins de la moitié de ses points de vie et il n'a pas de potion, ou il a un allié qui est en train de le soigner."

Bien entendu en code ça sera un tout petit peu différent visuellement, mais le principe est là.

Mettons en pratique tout ceci...

2.2 Prendre des décisions

Nous voilà enfin arrivé dans le vif du sujet. Pour que le code écrit par un programmeur soit réactif par rapport aux actions de l'utilisateur, ou même qu'il puisse traiter des cas demandant des réponses différentes, il existe un outil véritablement omniprésent dans tous les languages : les instructions de *contrôle de flux*, plus communément appellées par leur petit nom : `if`.

2.2.1 `if`, version simple

`if` en anglais, se traduit par "si". De là, on peut deviner le but de cette instruction : faire quelque chose *si* une condition est remplie.

Et c'est exactement ce qu'il se passe.

Voyons un exemple élémentaire :

```
a = 3
if a < 10:
    print(str(a) + " est plus petit que 10 !")
```

A votre avis, que se passe-t-il si on change la première ligne par `a = 245` ?

Il ne se passe rien. De façon un peu plus précise, l'instruction `if` est une instruction de branchement. Si la condition indiquée entre le `if` et le `:` est remplie (et donc égale à `True`), le code indenté[2] qui suit est exécuté, et seulement dans ce cas là.

Comme le `if` demande une condition, et donc une valeur booléenne, nous pouvons mettre en pratique ce que nous avons vu dans les sections précédentes.

```
a = 3
b = 5
if a < 10 and b > 3:
    print("Conditions remplies !")
```

Qu'est-ce qu'il se passe si `b` vaut 2 ? Si `a` vaut 12 ?

Et qu'en est-il de cet exemple encore plus compliqué ?

2. Nous allons y revenir tout de suite.

```
a = 3
b = 5
if (a < 10 and b > 3) or (a >= b):
    print("Conditions remplies !")
```

Quid si `b` vaut -12 et `a` vaut 42512 ?

On voit donc que rapidement, l'utilisation de quelques opérations de comparaison et d'algèbre de Boole permettent de définir des cas déjà assez complexes. Mais on peut encore faire mieux ! Avant de voir ça, cependant, passons quelques minutes à parler d'un sujet important, particulièrement en Python.

2.2.2 De l'importance de l'indentation

L'indentation, ou alignement du code, est capital en Python. En effet, à la différence de la plupart des langages, Python n'utilise pas d'accolades pour délimiter les différents blocs de code. A la place, l'indentation va jouer le rôle de différenciateur.

Pour comprendre, voici un petit script :

```
a = 3
if a > 10:
    print("Conditions remplies !")
    print("a est plus grand que 10 !")
print("a vaut " + str(a))
```

Les lignes 3 et 4, qui sont alignées, mais un niveau en plus que le reste des lignes, représente un bloc de code séparé. Si vous lancez le script tel quel, seule la dernière ligne sera imprimée à l'écran. Si vous changez le `a` pour qu'il vaille 15, le bloc qui suit le `if` sera bien exécuté.

Les problèmes d'indentation sont véritablement une source d'erreur courante. Il faut donc vraiment y payer une attention toute particulière.

2.2.3 `if`, version *full options*

Quand on utilise un *si*, on a naturellement envie d'utiliser un *sinon*. Ca tombe bien, Python, comme beaucoup de langages de programmation, vous offre cette possibilité. Pour reprendre un exemple vu plus haut et l'étoffer un petit peu :

```
a = 3
b = 5
if a < 10 and b > 3:
    print("Conditions remplies !")
else:
    print("Les conditions ne sont pas remplies !")
```

Le `else` dans l'exemple ci-dessus représente le *sinon*. Il indique que le bloc de code qui le suit directement doit être exécuté si et seulement si la condition du `if` n'est pas remplie.

Notez qu'il s'agit ici d'un choix exclusif : les deux blocs de code ne peuvent en aucun cas être tous les deux exécutés. Soit la condition du `if` est remplie, soit elle ne l'est pas, mais on ne peut pas avoir les deux en même temps.

Dans certains cas, vous aurez plus d'un cas à tester. Python offre une dernière option pour effectuer ça, `elif` :

```
a = 4
if a < 3:
    print("Condition 1 remplie !")
elif a < 5:
    print("Condition 2 remplie !")
elif a < 7:
    print("Condition 3 remplie !")
else:
    print("Les conditions ne sont pas remplies !")
```

Python va évaluer les différentes conditions indiquées dans le `if` et `elif` les unes après les autres, et exécuter le premier bloc de code pour lequel la condition est vérifiée. Et uniquement celui-là ! Même si les conditions suivantes sont remplies, après avoir exécuter le premier bloc, Python va passer le reste de l'instruction.

Le `else` ne sera exécuté que si toutes les conditions sont `False`, et uniquement ce cas-là.

Voilà de quoi déjà faire pas mal de choses un peu plus intéressantes. Une fois la notion de choix bien en place, nous pouvons passer à la suite, et apprendre à faire des boucles...

2.3 La boucle `for`, premier acte

S'il y a bien une chose pour laquelle les ordinateurs sont doués, c'est la répétitions. Contrairement aux humains, un ordinateur ne se fatigue pas, ni ne se lasse des tâches répétitives. Au contraire, la possibilité d'effectuer rapidement des opérations similaires sur un grand nombre d'objets ou de données constitue l'une des plus grande force de l'informatique.

D'ailleurs, au coeur de tout jeu vidéo, on trouve une boucle. En simplifiant, l'ordinateur est pris dans un cycle comme suit :

```
Tant que le joueur ne quitte pas le jeu:
    Vérifier si un bouton a été pressé
        Si oui, calculer la réaction
    Afficher le jeu mis à jour
    Recommencer
```

Les boucles sont donc un élément extrêmement important dans la boite à outils d'un programmeur. Mais avant d'écrire notre première boucle, faisons un petit détour pour apprendre un élément qui nous aidera.

2.3.1 `range()` ou comment générer une séquence

Python propose une série de fonctions permettant de gagner un temps précieux dans l'exécution de tâches courantes. L'une des plus courante est `range()`, qui permet de générer une série de nombres entiers.

La façon la plus simple de l'utiliser est simplement :

```
range(10)
```

Ce qui génère la série suivante :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ou, en généralisant, si `N` est un nombre entier :

```
range(N)
```

Le code renverra la série suivante :

```
[0, 1, 2, ... , N-1]
```

Le `N-1` est important ! La liste contient bien `N` éléments, mais ceux-ci commencent à `0`, et s'arrêtent à `N-1`. C'est une subtilité, mais à laquelle on s'habitue assez vite. En informatique, on commence souvent à compter à partir de 0 et non de 1.

Pour visualiser le résultat de `range()`, on peut utiliser la fonction `list()` qui créeera une liste intelligible pour nous à partir de la séquence renvoyée par `range()`. Mais un exemple vaut mieux qu'un long discours :

```
my_sequence = range(10)
my_list = list(my_sequence)
print(my_sequence)
```

A propos, on peut enchaîner les commandes pour condenser un peu le code :

```
print(list(range(10)))
```

Mais attention à ne pas se perdre dans les parenthèses !

Avant de voir comment utiliser `range()` pour contrôler une boucle, voyons encore rapidement 2 autres éléments.

Il est possible de générer une séquence qui va de `A` à `N-1` en écrivant

```
range(A, N)
```

Et si on voulait générer une séquence qui progresse par pas de 5 par exemple ?

```
range(0, N, 5)
```

Pour résumer :

```
range(start, end, step)
```

Où `start` est le point de départ de la séquence, `end` la fin de la séquence +1, et `step` l'incrément entre deux éléments de la séquence.

Bonus : comment faire une séquence qui compte de 10 à 1 ? Une idée ?

2.3.2 Le réveil de la `for`(ce)

Passons sur le titre grammaticalement incorrect de cette section, et intéressons-nous au véritable sujet qui nous intéresse : Comment créer une boucle en Python ?

Il existe plusieurs instructions de boucle en Python, la première que nous allons voir est une boucle dite *arithémtique*, aussi appellée boucle `for`.

La boucle `for` va parcourir une *séquence* d'éléments, comme une liste ou un tuple, et va effectuer le code qu'elle contient pour chaque élément de la séquence. Un exemple vaut mieux qu'un long discours :

```
for i in range(10):
    print('Compteur : ' + str(i))
```

Le résultat sera l'impression dans la console des nombres de 0 à 9 (souvenez-vous, Python commence à compter à 0, et va jusqu'au nombre qu'on lui a donné, -1).

Et c'est l'entierté de ce qu'il faut savoir pour utiliser une boucle `for` ! Comme le montre l'exemple, la boucle est dirigée par une variable spéciale, ici `i`, qui est appelée "compteur". On peut l'utiliser, à l'intérieur de la boucle, comme une variable normale.

Il existe un autre type de boucle, sans compteur, mais nous en parlerons plus tard dans le cours.

Chapitre 3

Level 3

3.1 Les boucles logiques

On a vu, dans le chapitre précédent, la boucle dite arithmétique. Avec cette boucle, la partie de code *contrôlée* par celle-ci s'exécute un nombre de fois précis.

Parfois, on sera intéressé par faire en sorte que le programme s'arrête non pas après un certain nombre de fois, mais plutôt lorsqu'une certaine condition est remplie. Pour ce faire, on fait appel aux *boucles logiques*.

En Python, c'est l'instruction `while` [1] qui est utilisée.

Cette instruction doit être suivie d'une *condition*, celle-ci va déterminer à chaque *itération*, si le programme doit continuer ou non d'exécuter la boucle.

Dans le script suivant, la boucle sera exécutée tant qu'il reste des monstres.

```
while monsters_nbr >= 0:
    hero_kills = hero.attacks()
    monster_nbr = monster_nbr - hero_kills
    game.run()
```

La méthode `attacks` de la variable `hero` renvoie 1 ou 0, donc il est impossible de savoir combien d'itérations seront nécessaires pour réduire la variable `monsters_nbr` à 0. C'est donc un *boucle logique* qui est utilisée dans ce genre de cas.

1. "Tant que" en anglais

La condition qui contrôle la boucle est similaire à la condition utilisée avec l'instruction `if`. La boucle s'arrête lorsque la condition est *faux*.

Dans le cas de notre script, c'est lorsque le nombre de monstres passe en-dessus de 0 que la condition devient *fausse*.

3.1.1 À retenir

Très, très, très important : il est impératif qu'à l'intérieur de votre *boucle logique*, il y ait un moyen de rendre la condition de continuation fausse. Sinon vous allez vous retrouver face à une boucle infinie.

3.1.2 Stage 3-1

Reprenez le *bonus stage* du *level 2*[2]. Arrangez ce code avec une *boucle logique* à la place de la *boucle arithmétique* pour que le héros arrête de bouger dès qu'il est arrivé à destination et non après un nombre d'itération arbitraire. Lorsque le héros a atteint sa cible, la méthode `hero.is_at_goal()` renvoie `True`, dans les autres cas elle renvoie `False`.

3.2 imports

En Python, il est courant de devoir *importer* le contenu d'un fichier dans un autre fichier. À plusieurs reprises, on déjà vu cette instruction dans les scripts que l'on a utilisés :

```
import world
```

Cette commande, demande à Python d'*importer* le contenu du fichier `world.py` en vue de l'utiliser plus tard. Dans la suite, si l'on veut faire appel au contenu de `world`, on doit utiliser son nom suivit d'un `"."`

Par exemple, si je veux utiliser la variable `GROUND_IMAGE_PATH` dans mon script, je vais procéder comme suit :

```
import world

scree_size = world.GROUND_IMAGE_PATH
```

2. au besoin le code corrigé de cet exercice se trouve dans le script `bonus_stage_2.py`

On peut aussi n'*importer* qu'une partie d'un *module*[3].

```
from world import GROUND_IMAGE_PATH
```

En procédant de cette manière, on n'*importe* que la variable `GROUND_IMAGE_PATH`. Cela présente l'avantage de ne pas devoir précéder la variable `GROUND_IMAGE_PATH` de `world.`, on peut se contenter du nom de la variable seul.

```
from world import GROUND_IMAGE_PATH

screen_size = GROUND_IMAGE_PATH
```

Et enfin, on peut *importer* plusieurs composants d'un *module* sur la même ligne. Si je voulais importer `GROUND_IMAGE_PATH` et la variable `OBSTACLE_IMAGE_PATH`, je procéderait ainsi :

```
from world import GROUND_IMAGE_PATH, OBSTACLE_IMAGE_PATH
```

3.2.1 Stage 3-2

Dans le fichier `data.py` se trouvent toutes les données utilisées pour le jeu. Parmis ces données se trouve la donnée de la *taille de la fenêtre du jeu* : `SCREEN_SIZE` (c'est un `tuple` correspondant à `(largeur,hauteur)` et la donnée du *facteur de grossissement* : `SCALE_FACTOR`. La grandeur réelle de l'écran est calculée en multipliant la *largeur* et la *hauteur* de l'écran par le *facteur de grossissement*. *Importez* ces données dans le `run_game` et affichez dans la console la taille réelle de la fenêtre de jeu.

3.2.2 N'Import quoi !

Il n'y a pas que les *modules* que vous avez faits vous-même que vous pouvez *importer*. Python est livré avec un série de *modules* prêts à être utilisés par vos mains avides de coder. Pour importer ces *modules*, il n'y a pas de différences par rapport à pour vos *modules* personnels. Il suffit juste de connaitre le nom du *module* que vous voulez importer[4]. L'un de ces modules qui nous sera très utile dans le cadre de la réalisation de jeux vidéos, c'est `random`.

Pour l'importer donc pas de mystère :

3. C'est le nom que l'on donne au fichier que l'on importe
4. La liste des modules disponibles dans Python et leur contenu est disponible sur le site de Python (`www.python.org/doc`)

```
import random
```

3.2.3 Stage 3-3

Ce module contient la fonction `(randint)`. Cette fonction prend deux arguments qui doivent être de type `int`. En important cette fonction et uniquement cette fonction (Rappelez-vous, les méthodes d'*import* de *modules* Python ne sont pas différentes de l'*import* de modules personnels), essayez de déterminez ce que fait la fonction (même si le nom du *module* et de la fonction sont de gros indices). Vous pouvez utiliser le script `run_game`, ou bien la console Python pour faire ces tests.

3.3 La génération de nombres aléatoires

Vous avez pû le constater, la fonction `randint` génère des nombres aléatoires compris entre le premier argument de la fonction et le deuxième argument [5] Générer des nombres aléatoires est extrêmement pratique, car cela permet de simuler le caractère hasardeux d'une situation, un peu à l'image d'un dé.

3.3.1 Stage 3-4

Utilisez la fonction random sur le jeu. Simulez un dé à 6 faces :

1 : le héros tourne à droite

2 : le héros tourne à gauche

3-5 : le héros avance

6 : le héros arrête de bouger définitivement

En plus de cela, à chaque lancer de dé : affichez dans la console le résultat de celui-ci.

5. Pour autant que les arguments soient en ordre croissant, sinon ça génère une erreur.

Chapitre 4

Level 4

4.1 Les fonctions

Une *fonction* est un outils qui permet d'abstraire le code afin de garder un maximum de structure et de lisibilité dans le code.

En gros, une *fonction* peut être vue comme un bout de code sur lequel on aurait mis un nom.

Nous avons déjà utilisé plusieurs fonctions : `print`, `input`, `len`, ...

Nous avons même déjà vu sans savoir exactement ce que c'était comment définir une *fonction* dans le code de notre jeu :

```
import world


def main():
    dungeon_size = (10, 10)
    game = world.Game(dungeon_size)
    hero = game.hero
```

Dans ce morceau de code la *fonction* se nomme `main`.

Pour définir une fonction, il faut simplement écrire `def` suivit de parenthèse [1] et d'un `":` Le code de la fonction comme pour les boucles ou la condition, doit être indenté.

Pour appeler une fonction, il suffit d'utiliser son nom, comme suit :

1. Nous verrons dans la suite du cours que ces parenthèses ne seront pas toujours vide.

```
main()
```

Dans ce cas, c'est la fonction `main` qui est appelée.

4.1.1 Important

Pour la claretée de votre code et pour facilité sa relecture, il est très important que le nom de votre fonction soit explicite. Ne nommez pas une variable qui simule le lancement d'un dé `a` ou même `d`, mais plutôt `dice` ou `dice_simulation`

4.1.2 Stage 4-1

Dans le jeu du bingo (Stage 3-4) Ajouter une *fonction* `bingo` qui affiche "Bingo !" dans la console. Dans le code principale du jeu, remplacé la partie qui s'occupe d'écrire "Bingo !" par l'appel de votre fonction

4.2 Le retour

L'une des caractéristique d'une *fonction* c'est qu'elle peut avoir un *retour*. Par exemple, le nombre aléatoire donné par la focntion `randint` est son *retour*.

Un *retour* est une valeur donnée par une *fonction*. On dit d'ailleurs que la fonction *retourne* une valeur.

Lorsqu'il y a besoin de retourner une valeur on utilise le mot clé `return`.

Le *retour* d'une fonction doit bien évidemment être recueillit par le code appelant, sinon il est perdu.

```
import random


def dice_simulation():
    dice = random.randint(1, 6)
    return dice

result = dice()
print("Résultat du dé: " + result)
```

4.3 Stage 4-2

Faite en sorte de bouger dans une *fonction* la partie du code qui demande à l'utilisateur de choisir un chiffre dans une *fonction* et faite en sorte que cette *fonction renvoie* le nombre choisis par l'utilisateur.

4.4 Paramètres

Une fonction peut aussi avoir un ou plusieurs *arguments*. Les *arguments* sont des valeurs qui sont transféree du code appelant à la *fonction* via des variables appellées *paramêtres*

Dans le code suivant :

```
from random import randint


dice = randint(1, 6)
```

1 et 6 sont les *paramêtres* de `randint`.

On définit les paramêtres d'une fonction en donnant des nom de variable (qui seront ensuite utilisées dans le code de la fonction.

```
def double(number):
    return number * 2
```

Dans cette exemple un peu bateau, `number` est l'argument de la *fonction* `double`.

```
def list_multiplication(array, number):
    clone = []

    for n in array:
        clone.append(n * number)

    return clone
```

Dans cette autre exemple, `array` et `number` sont les arguments de la fonction `list_multiplication`

4.4.1 Stage 4-3

Dans le bingo, faites en sorte de sortir le code qui vérifie le nombre rentré par le joueur dans une *fonction*. Le code affichera si le nombre est plus petit ou plus grand, et renverra `True` si le joueur à deviner juste et `False` dans les autre cas. La *fonction* prendra en *paramêtres* le nombre donné par le joueur ainsi que le nombre à deviner.

4.5 Valeur par défaut

Il y a un moyen de donner une *valeur par défaut* à un argument. Cette valeur lui sera donné si le code appelant ne le fait pas.

```
import random


def dice_simulation(faces=6):
    dice = random.randint(1, faces)
    return dice
```

Dans le script ci-dessus, l'*argument* à une *valeur par défaut* de 6. Si le code appelant est :

```
result = dice_simulation(10)
```

le *paramêtre* `faces` vaut 10, mais si on ne l'avait pas préciser, comme dans le code suivant :

```
result = dice_simulation()
```

le *paramêtre* `faces` vaut 6 dans ce cas là.

Il est possible de mélanger des *arguments* sans *valeur par défaut* et des argument avec *valeur par défaut* dans une fonction, mais il faut impérativement que les *paramêtres* avec *valeur par défaut* soit en dernier dans la liste d'*arguments*

```
import random


def dices_simulation(dice_number, faces=6):
    result = 0
```

```
    for n in range(dice_number)
        dice = random.randint(1, faces)
        result = result + dice
    return result
```

Dans le code du bingo, modifier la détermination des bornes du jeu à l'aide d'une *fonction* avec une *valeur par défaut*. La première fois, le nombre est choisis de manière normale, mais à chaque fois que le joueur recommence de jouer, la borne maximum est modifiée de manière à valoir `born_max + (7 - tentatives`. La toute première fois où elle est appellée, la *fonction* n'aura pas de *paramêtre*, et les autres fois elle aura le nombre de tentatives comme *paramêtre*.