Etoile Filante
TAlone
Bibou
Amaury

Zeste de savoir

Zeste de Code – Billet participant

4 mai 2019

Table des matières

1. Introduction

Ce billet vise à être le corrigé complet de la partie Python de l'atelier Zeste de Code ; il donnera aussi aux participant · e · s des éléments de réflexion et des pistes d'amélioration complémentaires.

En premier lieu, il sera possible de trouver un tutoriel d'installation des différents éléments nécessaires pour utiliser Python à la maison, puis une synthèse de la présentation des fonctions principales sera mise à disposition, et les dernières parties, spécifiques à l'atelier, reprendrons la structure du Snake, avec la mise à disposition de l'aide-mémoire des fonctions de la bibliothèque de l'atelier, et un corrigé point-par-point du jeu.

i Ce billet est écrit pour les participants de *Zeste de Code*

Zeste de Code est un atelier d'initiation à la programmation organisé par Zeste de Savoir. Ce billet est à destination des participantes de cet atelier afin qu'ils et elles puissent continuer et terminer à la maison.

Si vous n'avez pas participé à l'atelier et que vous voulez juste apprendre Python, préférez consulter directement **le cours de Python pour débutants** . Et si ces ateliers vous intéressent, surveillez le site, on en organisera d'autres !

2. Installation de Python et des logiciels utiles

2.1. Présentation de Python et Spyder

Comme nous l'avons vu pendant l'atelier, un ordinateur est fondamentalement idiot ; il ne sait rien faire de lui-même, on rappellera donc que toutes les fonctions qui doivent être exécutées dans votre jeu seront commandées par vous-même, sans quoi l'ordinateur n'en fera rien.

Afin de parler avec l'ordinateur, nous mentionnions l'existence de langages, analogues dans le monde physique au langues humaines, et qui sont en fait simplement des outils permettant de communiquer facilement avec l'ordinateur.

Les langages de programmation sont de différents « niveaux » ; Python, que nous avons choisi est dit de haut-niveau car il masque la plupart des opérations élémentaires effectuées par l'ordinateur. Par opposition, les langages de bas-niveau gèrent les opérations de l'ordinateur à un niveau plus proche de ce qu'il se passe réellement dans l'ordinateur.

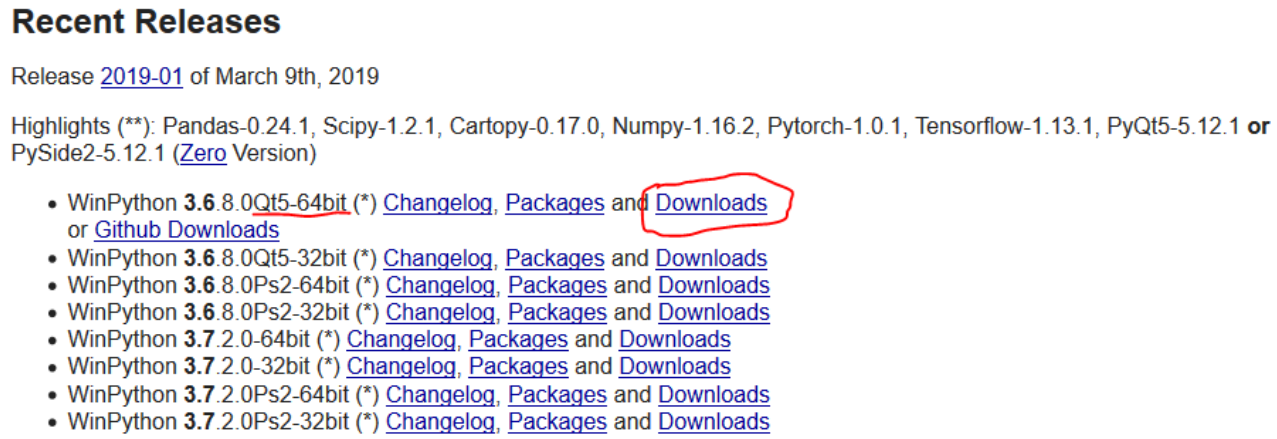
Python est un langage simple et moderne, utilisé sur des applications allant de l'éditeur de texte au serveur web ; pour citer quelques services tournant sous Python, afin d'illustrer la puissance du langage :

— le réseau social Instagram est écrit en Python ;
— EDF a recours à Python pour ses calculs mécaniques et d'interférences électromagnétiques ;
— Google utilise le langage pour sa plateforme de vidéos YouTube, entre autres ;
— la NASA, l'ESA, le CNES, utilisent Python pour divers logiciels de traitement des observations spatiales ;
— LibreOffice, suite bureautique libre, utilise Python comme langage de commande ;
— le site sur lequel vous êtes en train de lire ce billet est aussi en Python ;
— cette liste n'est bien entendu pas exhaustive.

Afin d'utiliser le langage Python, il est possible d'avoir recours à diverses méthodes ; nous vous proposons aujourd'hui une méthode simple d'installation de Spyder, qui est un IDE – c'est-à-dire un logiciel regroupant éditeur de texte, compilateur et gestion des erreurs. Ce logiciel est le même que celui utilisé lors de l'atelier.

2.2. Installation des outils nécessaires

Pour installer Spyder, nous allons utiliser la méthode de WinPython, plus simple que la méthode habituelle par Anaconda ; commençons par aller sur le site de WinPython . Il suffit ensuite de cliquer sur « Downloads » à côté de la version dite « Qt5-64bit », puis une fois sur une nouvelle page, de cliquer sur cette même version ; pour référence, voici les captures d'écran de téléchargement de la version disponible à l'heure où ces lignes sont écrites :



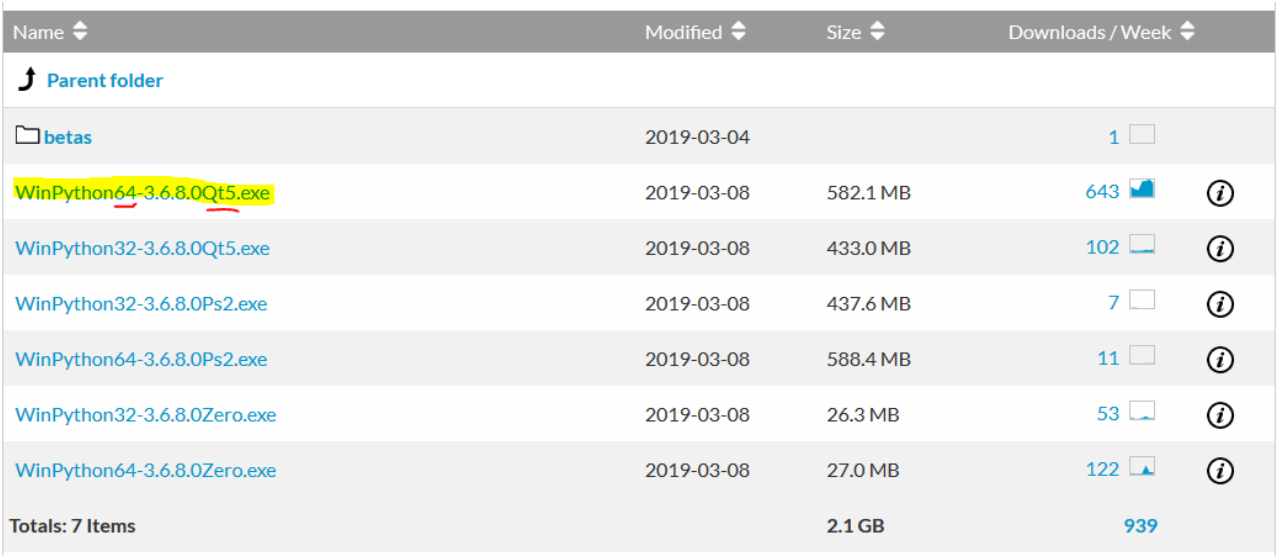
Recent Releases

Release 2019-01 of March 9th, 2019

Highlights (**): Pandas-0.24.1, Scipy-1.2.1, Cartopy-0.17.0, Numpy-1.16.2, Pytorch-1.0.1, Tensorflow-1.13.1, PyQt5-5.12.1 **or** PySide2-5.12.1 (Zero Version)

- WinPython **3.6**.8.0Qt5-64bit (*) Changelog, Packages and Downloads or Github Downloads
- WinPython **3.6**.8.0Qt5-32bit (*) Changelog, Packages and Downloads
- WinPython **3.6**.8.0Ps2-64bit (*) Changelog, Packages and Downloads
- WinPython **3.6**.8.0Ps2-32bit (*) Changelog, Packages and Downloads
- WinPython **3.7**.2.0-64bit (*) Changelog, Packages and Downloads
- WinPython **3.7**.2.0-32bit (*) Changelog, Packages and Downloads
- WinPython **3.7**.2.0Ps2-64bit (*) Changelog, Packages and Downloads
- WinPython **3.7**.2.0Ps2-32bit (*) Changelog, Packages and Downloads

FIGURE 2. – Téléchargement depuis le site officiel



Name	Modified	Size	Downloads / Week
Parent folder			
betas	2019-03-04		1
WinPython64-3.6.8.0Qt5.exe	2019-03-08	582.1 MB	643
WinPython32-3.6.8.0Qt5.exe	2019-03-08	433.0 MB	102
WinPython32-3.6.8.0Ps2.exe	2019-03-08	437.6 MB	7
WinPython64-3.6.8.0Ps2.exe	2019-03-08	588.4 MB	11
WinPython32-3.6.8.0Zero.exe	2019-03-08	26.3 MB	53
WinPython64-3.6.8.0Zero.exe	2019-03-08	27.0 MB	122
Totals: 7 Items		2.1 GB	939

FIGURE 2. – Téléchargement depuis Sourceforge

Il vous suffit ensuite d'attendre quelques secondes et le téléchargement va commencer ; sur certaines connexions, le téléchargement peut prendre un certain temps. Une fois l'installateur téléchargé, il suffit de l'exécuter, puis d'accepter les termes de la licence (« I accept the agreement »), et de cliquer deux fois sur « Next » ; aucun privilège administrateur n'est demandé. Une fois l'installation prête, cliquez sur « Install », puis patientez quelques temps pour que le logiciel s'installe.

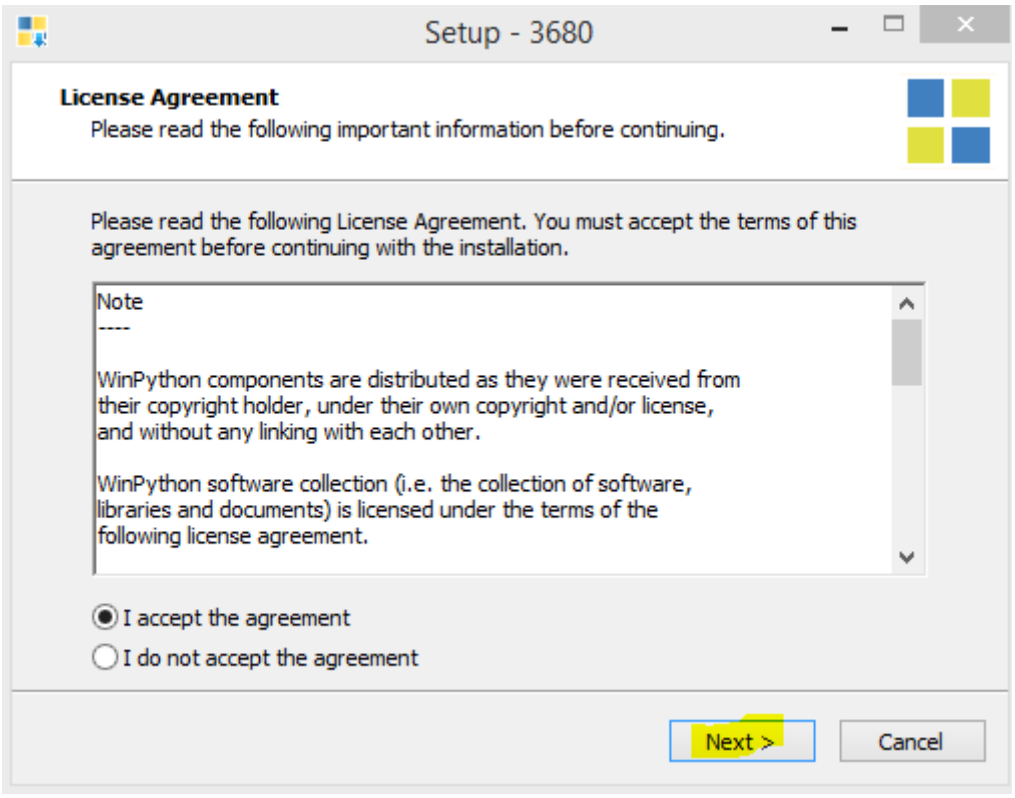


FIGURE 2. – Installation du logiciel

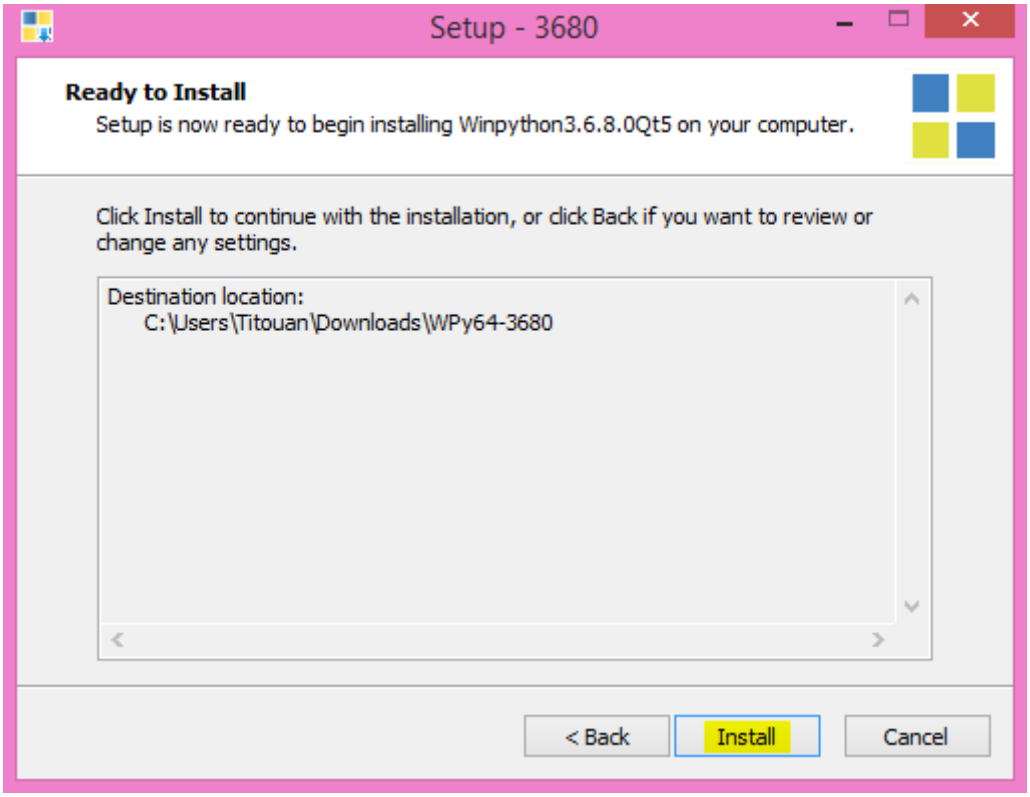


FIGURE 2. – Installation du logiciel 2

Une fois le logiciel installé, rendez-vous dans le dossier d'installation (normalement il se situe à l'endroit où vous avez exécuté l'installateur, sauf si vous avez modifié le chemin), puis lancez

le logiciel « Spyder.exe ». Si Python vous demande d'accéder au réseau, vous pouvez refuser si vous n'avez pas les droits administrateurs, nous n'aurons pas besoin d'un accès réseau dans Python.

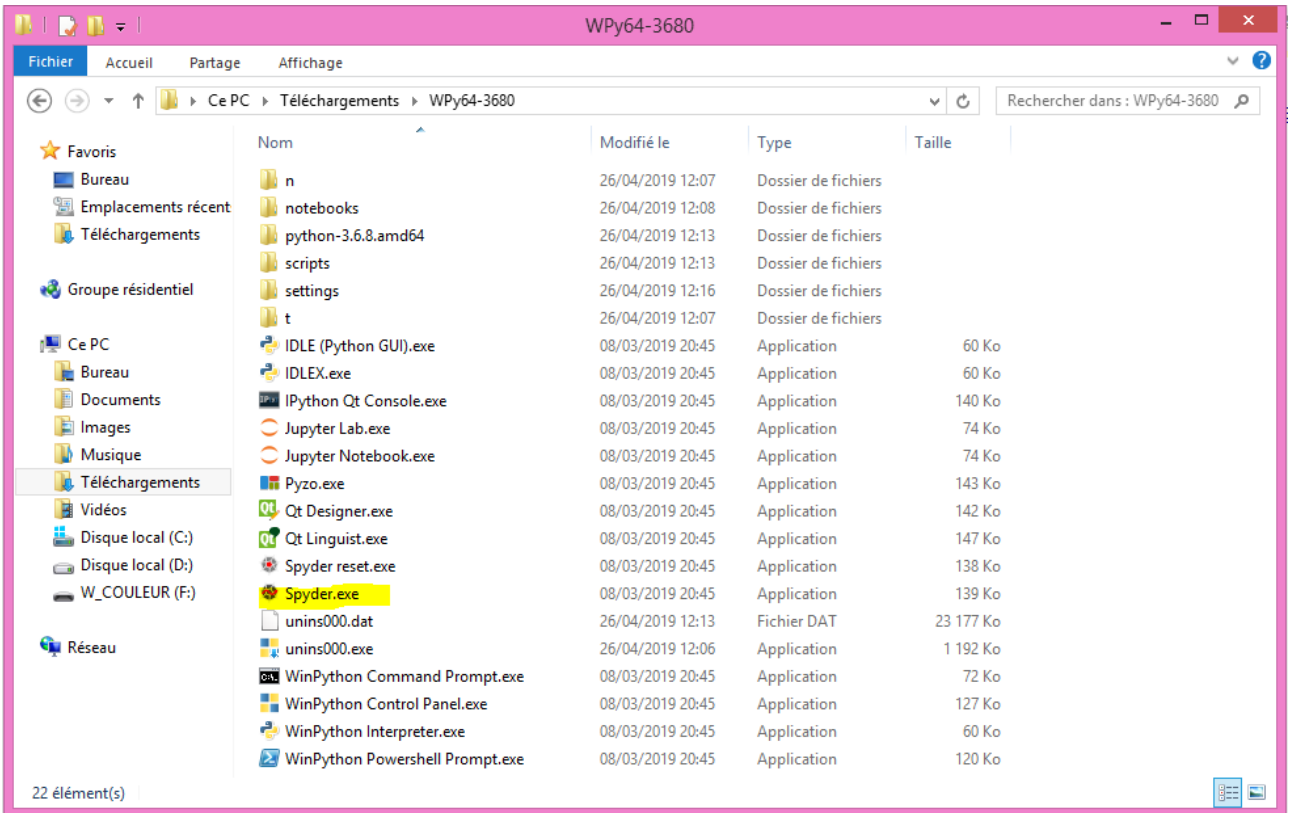


FIGURE 2. – Exécution de l'IDE

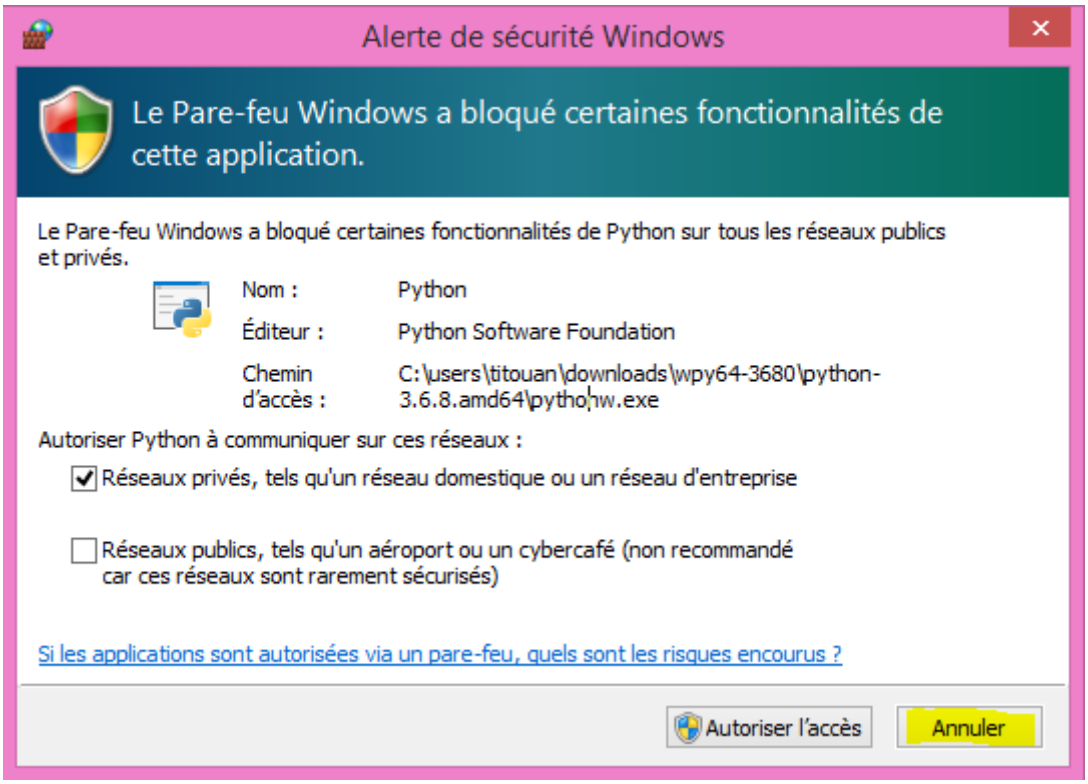


FIGURE 2. – Annulation de l'accès réseau

Une fois l'éditeur ouvert, rendez-vous dans la console en bas à droite et tapez la commande suivante, qui installera les outils dont nous aurons besoin par la suite :

```
1 pip install --user pygame dotmap
```

Une fois la commande lancée, redémarrez le logiciel.

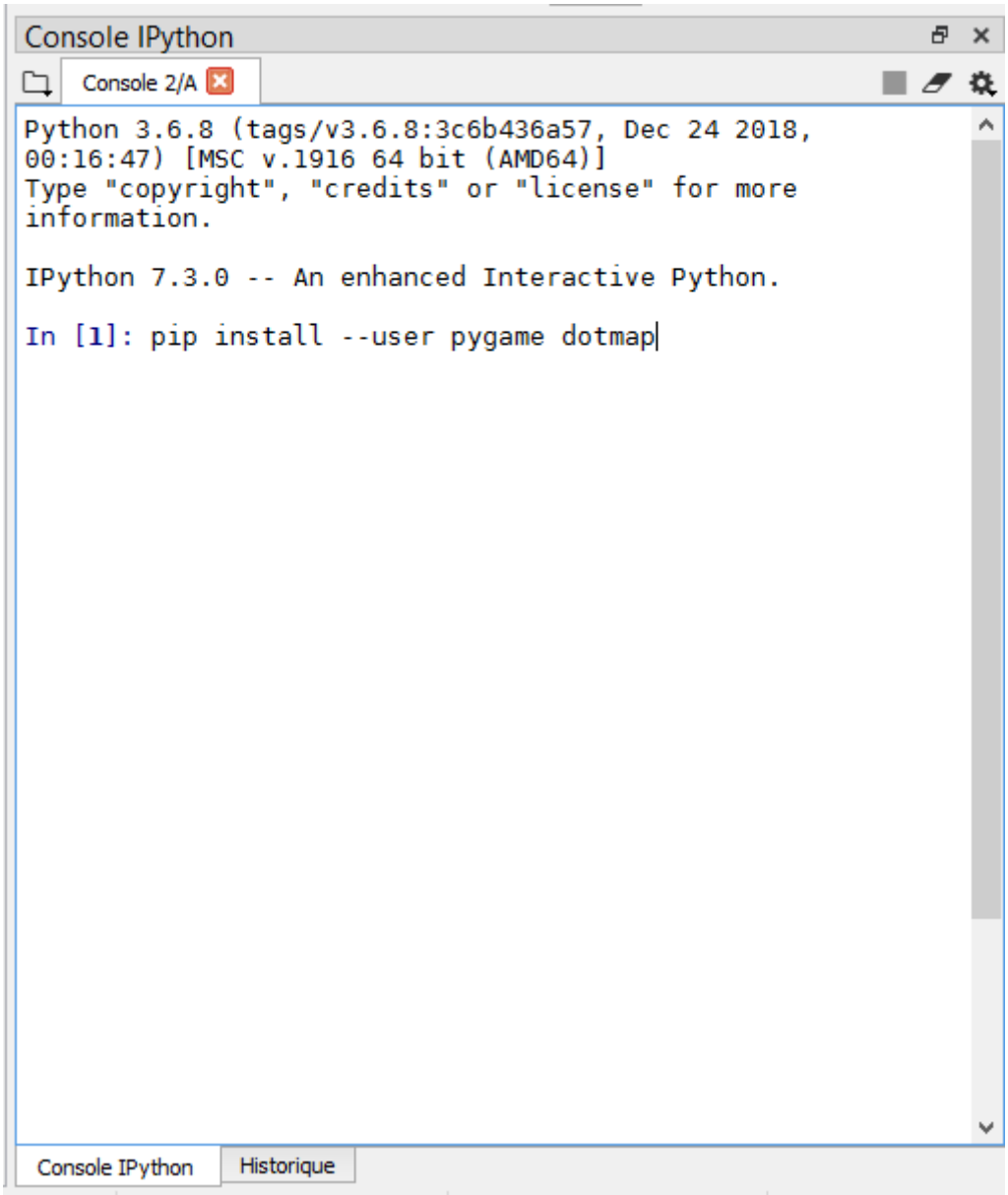


FIGURE 2. – Installation des outils requis

Le logiciel relancé, vous pouvez maintenant ouvrir votre fichier de l'atelier en utilisant l'icône , puis l'exécuter en cliquant sur dans la barre du haut. Si avant d'éxécuter le programme, une fenêtre comme celle ci-dessous s'ouvre, choisissez les paramètres indiqués :

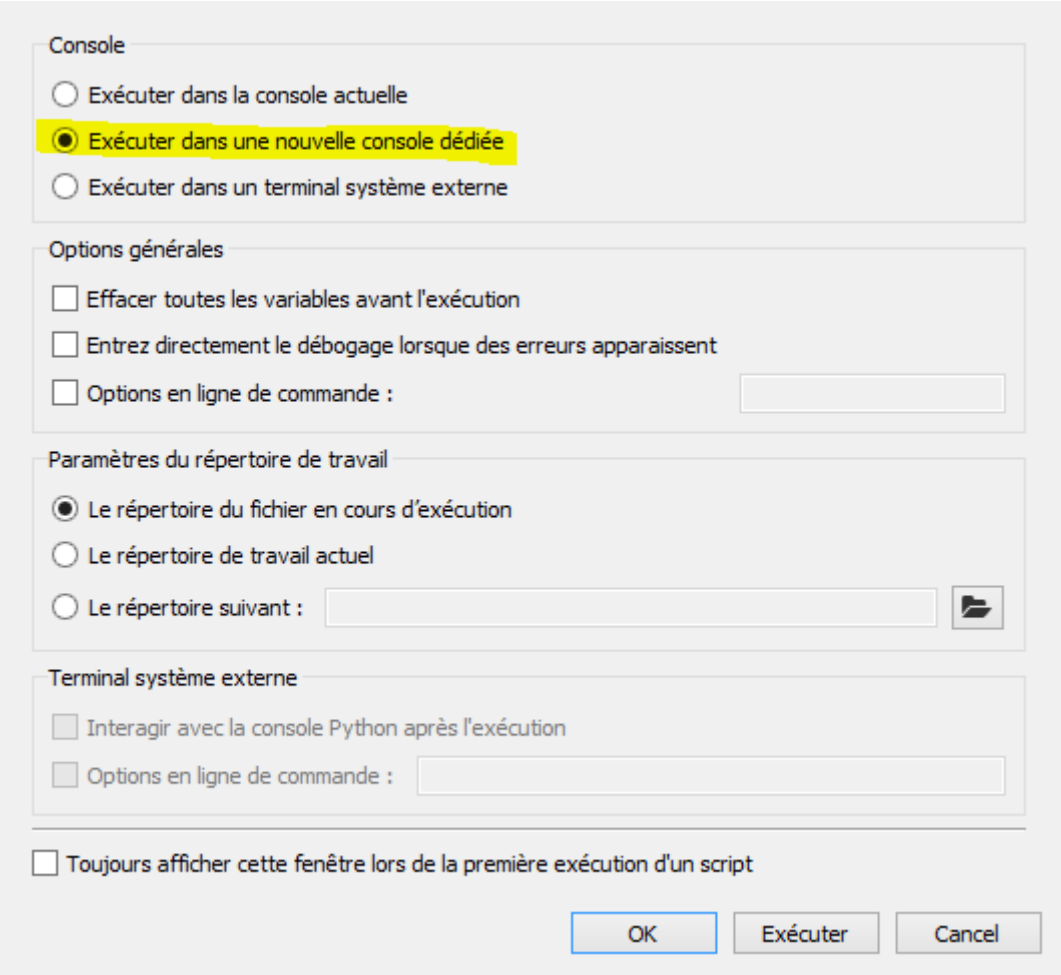


FIGURE 2. – Configuration du logiciel

Votre code est maintenant ouvert et prêt à être modifié ; amusez-vous !

3. Rappel des fonctions élémentaires de Python

En guise d'introduction à la programmation, nous vous avons proposé d'étudier les bases de Python par un court diaporama. Cette partie revient sur les notions étudiées dans ce diaporama, sans plus rentrer dans les détails ; un lien sera proposé à la fin de la partie si certain · e · s d'entre vous souhaitent aller plus loin avec Python.

i L'entièreté du code de cette partie doit être tapée dans la *console* Python à droite de Spyder. Nous n'utiliserons la fenêtre de texte à gauche que par la suite.

3.1. Découverte de l'interpréteur

3.1.1. Calculs simples

Un ordinateur n'est fondamentalement qu'une machine à calculer, et est bien plus rapide que l'être humain à ce niveau, c'est pourquoi nous en avons tant besoin pour effectuer des tâches complexes. Avant d'effectuer toute autre opération, il est alors bon de voir comment utiliser Python comme simple calculatrice. Pour rappel, le langage Python interprétera et exécutera tout calcul présent dans son code. Quelques exemples sont proposés ci-dessous :

```
>>> 1 + 1
2
>>> 6 * 7
42
>>> 4 - 2
2
>>> (6 - 4) * 2
4
>>> 3.6 / 2
1.8
```

Comme vous pouvez le voir, les opérations sont effectuées en les tapant directement dans la console Python. Notez toutefois que la virgule `,` est remplacée par un point `.` et que le signe de multiplication usuel est représenté par une étoile `*`. Pour s'exercer, il est par exemple possible de calculer le triple de 5,2.

3.1.2. Notion de variable

Nos opérations sont bien pratiques, mais concrètement, hormis pour faire une calculatrice moche, il sera nécessaire de stocker de l'information afin de créer des programmes en Python. Heureusement pour nous, votre ordinateur dispose de mémoire, et peut se souvenir pour nous de *valeurs*. On appelle ces valeurs stockées des **variables**, qui permettent simplement de représenter des informations (texte, chiffres, listes, etc.) par un *nom*.

i **Une variable**

C'est une sorte de **boîte** dans laquelle on peut mettre une valeur.
Ces valeurs sont typiquement :

- des nombres ;
- des textes ;
- des listes.

```
>>> x = 3
>>> x
```

```
3
>>> x + 1
4
>>> x
3
```

Ici, on associe au nom `x` (qu'on appellera «variable `x`») la valeur 3 ; cette valeur ne changera pas dans la suite du programme, puisqu'on n'associe qu'une seule fois une valeur à x.

On peut également, comme vous l'avez lu, mettre un texte dans une variable en l'entourant de guillemets (pour que Python comprenne bien que c'est un texte). On parle parfois aussi de *chaîne de caractères* pour parler de textes en informatique, donc ne vous étonnez pas si vous croisez ce terme.

```
>>> nom = "Zeste de Code"
>>> nom
'Zeste de Code'
```

Les trois possibilités pour une variable sont :

— la **déclaration** : `x = 3`, on associe une valeur à x ;
— l'**utilisation** : `x` permet par exemple en console de montrer la valeur de x ;
— la **modification** : `x = x + 1` donnerait par exemple à x la valeur 4.

En guise d'exemple :

— taper `y = 4` puis taper `y = y + 2` ;
— la valeur de `y` a-t-elle changé ?

3.1.3. Appeler une fonction

Parfois, on ne sait pas comment réaliser une certaine action, ou l'on a pas envie de refaire quelque chose qui aurait déjà été développé par quelqu'un d'autre ; dans ce cas, on peut faire appel aux fonctions.

Les fonctions peuvent avoir besoin d'informations pour travailler (par exemple, une fonction qui affiche un texte sur l'écran va avoir besoin du texte à afficher), et si ça fait sens, peuvent *retourner* ce sur quoi elles ont travaillé (par exemple, une fonction qui génère un nombre aléatoire va retourner un tel nombre pour qu'on puisse l'utiliser).

Une fonction

C'est un morceau de code que quelqu'un d'autre a déjà écrit pour vous, que l'on appelle avec son nom pour ne pas le ré-écrire.
Elle peut prendre des informations en *paramètre* pour travailler dessus et *retourner* son travail si nécessaire.

```
>>> print("Bonjour !")
Bonjour !
>>> x = 3
>>> print(x)
3
```

Ici, la fonction `print` nous sert à afficher du texte, mais nous n'avons aucune idée de comment elle fonctionne en interne (c'est « magique »).

On notera ici que x est passée directement à la fonction, on dit ainsi que la variable x est donnée en argument. Pour le cas du texte `"Bonjour !"`, il est passé entre guillemets, ce qui indique que la chaîne de caractères `"Bonjour !"` est donnée en argument.

3.1.3.1. Des fonctions qui travaillent Parfois, les fonctions retournent leur travail ; dans ce cas, on peut stocker ce travail dans une variable de la même façon qu'on y enregistrait un nombre ou un texte plus haut (et c'est là que ça commence à être particulièrement utile !).

Par exemple, la fonction `len` (de *length*, taille en anglais) retourne la taille de ce qu'on lui donne en paramètre (typiquement un texte ou une liste). On peut l'utiliser comme suit :

```
>>> taille_du_prenom = len("Alban")
>>> taille_du_prenom
5
```

3.1.3.2. Des fonctions cachées Parfois, les fonctions sont cachées, et il nous faut dire à Python où elles se trouvent avant d'être en mesure de les utiliser, par exemple :

```
>>> import math
>>> math.ceil(2.2)
3
```

La fonction `math.ceil` arrondit le nombre passé à la fonction à l'entier supérieur. Ces fonctions commençant par `math.` doivent être *importées* avant d'être utilisées, on importe pour cela la *bibliothèque* math.

3.1.3.3. Des fonctions qui utilisent plein de choses Parfois, les fonctions ont besoin de plusieurs informations pour travailler. Par exemple, la fonction `min` retourne le plus petit des nombres donnés, elle a dont besoin de *deux* nombres (ou plus). Dans ce cas, on sépare les paramètres par une virgule :

```
>>> min(21, 84)
21
```

3.1.3.4. Des fonctions qui n'utilisent rien Certaines fonctions n'ont simplement pas besoin d'infos pour travailler. Par exemple, il existe une fonction qui génère un nombre aléatoire entre 0 et 1 : c'est aléatoire, donc elle ne se base sur rien. Dans un tel cas, **il faut tout de même mettre une paire de parenthèses vide après**, pour que Python comprenne bien qu'on veut utiliser la fonction.

Si on l'oublie, Python ne plante pas, mais fait des choses bizarres et certainement pas ce qu'on veut. Voyez plutôt:

```
## La méthode pour générer un nombre aléatoire entre 0 et 1 est
    dans
# la bibliothèque « random » (cf. deux paragraphes plus haut).
>>> import random

## Kwa ?!
>>> random.random
<built-in method random of Random object at 0x55bc7fe33b28>

## Ah bah oui c'est mieux !
>>> random.random()
0.38599908375461944
```

Ce sera le cas de certaines fonctions du Snake, par exemple celles retournant la liste des cases de la grille (la grille est connue, pas besoin de lui rappeler : la fonction ne prend donc aucun paramètre).

i Au passage, les lignes commençant par `#` sont des commentaires que Python va ignorer complètement. C'est pratique pour expliquer le code ou laisser des notes pour se souvenir de la façon dont il fonctionne. D'ailleurs, n'hésitez pas à en utiliser dans vos codes si vous voulez écrire en français à côté d'un bout de code ce qu'il fait : c'est quelque chose de courant et c'est très pratique pour mieux comprendre ce que l'on écrit — autant en tant que débutant que de confirmé d'ailleurs !

3.2. Les conditions

3.2.1. Les conditions `if`

En plus de faire des calculs, un ordinateur est capable de comparer des valeurs ; on appelle ces opérations des *conditions*.

Une condition

C'est une expression qui permet de n'exécuter un morceau de code seulement si elle est vraie.

Exprimées par les opérateurs anglais `if` (si) et `else` (sinon), elle permettent de réaliser des choses comme suit:

```
1 age = 23
2 if age > 17:
3     print("Tu es majeur")
```

Ici, **si** l'âge entré **est supérieur** à 18 ans, on affiche que la personne est majeure.

En ce qui concerne les opérateurs utilisables, nous n'en verrons que cinq :

— `>` : supérieur à ;
— `<` : inférieur à ;
— `==` : égal à ;
— `!=` : différent de ;
— `in` : présent dans la liste donnée après *in*.

Notons bien la différence entre le `=`, qui donne une valeur à une variable, et le `==` qui en teste la valeur.

3.2.2. Les conditions `if` et `else`

En plus de l'opérateur `if` permettant d'effectuer une comparaison, nous vous parlions tout à l'heure du `else`, qui s'exécute si la condition du `if` le précédant n'est pas validée :

```
1 age = 23
2 if age > 17:
3     print("Tu es majeur")
4 else:
5     print("Tu es mineur")
```

Si l'âge entré n'est pas supérieur à 17 (c'est-à-dire si il est inférieur ou égal à 17), alors nous afficherons « Tu es mineur ».

3.2.3. Plusieurs `if` avec `elif`

Dans certains cas, il est nécessaire de ne tester une condition `if` que lorsque la précédente ne s'est pas exécutée ; on pourrait ainsi être tentés de faire quelque chose comme suit :

```
1 age = 15
2
3 if age >= 21:
4     print("Majorite internationale")
5 else:
6     if age >= 18:
7         print("Tu es majeur")
8     else:
9         print("Tu es mineur")
```

La syntaxe est très lourde, et le langage Python dispose donc d'une syntaxe courte, combinant le `else` et le `if`, elle est donc nommée `elif`, et donne un résultat comme suit :

```
1 age = 15
2
3 if age >= 21:
4     print("Majorite internationale")
5 elif age >= 18:
6     print("Tu es majeur")
7 else:
8     print("Tu es mineur")
```

3.3. Les boucles `for`

Aucune des instructions vues jusqu'à présent ne permet de faire vraiment plus qu'un être humain ne le pourrait ; l'ordinateur devient véritablement intéressant lorsqu'il s'agit d'effectuer des actions répétitives de façon très rapide. Pour cela, nous pouvons avoir recours aux boucles, qui ne sont qu'un moyen de répéter une action en changeant à chaque fois une variable.

Une boucle

C'est un morceau de code répété un certain nombre de fois selon certains critères.

Il existe plusieurs types de boucles en Python, mais nous allons ici n'en utiliser qu'une : la boucle appelée `for`, permettant de lister le "contenu" d'une liste (contenue dans une variable ou retournée par une fonction).

Les boucles `for` ressemblent à ceci :

```python
for age in range(5, 26):
    print(age)
    if age > 17:
        print(" = majeur \n")
    else:
        print(" = mineur \n")
```

Ici, la boucle utilise la fonction range, qui donne simplement une liste contenant dans l'ordre les nombres de 5 à 26 (dans cet exemple). Pour chaque âge, la condition le compare à 18 et affiche si l'âge en question correspond à une personne majeure ou mineure.

Pour donner un exemple concret, le jeu du Snake propose une fonction qui retourne la liste des cases de la grille à l'écran, et une boucle permet de les parcourir et de les remplir les unes après les autres.

3.4. Certains variables sont plus grosses que d'autres

Nous n'entreront pas dans le détail car c'est un peu complexe, mais il vous faut savoir que certaines variables peuvent également être de «très grosses boîtes» contenant en leur sein d'autres variables et des fonctions.

Il ne vous est pas utile de comprendre précisément comment ça marche, mais sachez que les "sous-variables" et les "sous-fonctions" d'une telle grosse variable s'utilisent de la même façon que présenté plus haut, mais en les préfixant de la variable mère et d'un point, comme suit :

```
## Une sous-variable (on parle aussi de "propriété")
>>> jeu.taille
3

## Une sous-fonction (on parle aussi de "méthode")
>>> jeu.effacer_ecran()

## On peut aussi imbriquer plusieurs couches !
# Par exemple, une sous-fonction dans une sous-variable
>>> jeu.serpent.grandir()
```

Dans le jargon, si vous voulez creuser par la suite et aller plus loin, de telles variables sont appelées "objets".

Ce domaine des *objets* (la « programmation orientée objets ») est très puissant mais aussi bien plus complexe que cette initiation. Vous pouvez tout à fait vous y intéresser, mais ne brûlez pas les étapes de peur de vous perdre .

Pour ceux souhaitant étudier ces fonctions élémentaires plus en détails, notre site propose un tutoriel Python pour débutants , qui pourra vous guider de façon plus posée et plus poussée que l'introduction rapide de 2 heures.

4. Aide-mémoire des fonctions de la bibliothèque du goûter

Ce document est une documentation complète des fonctions de la bibliothèque de l'atelier Zeste de Code ; n'hésitez pas à vous référer à lui afin de *rechercher une fonctionnalité*.

4.1. Fonctions du jeu

4.1.1. Présentation générale

4.1.1.1. Déclaration de l'objet Jeu Après avoir importé la bibliothèque Zeste de Code, vous aurez accès aux fonctions principales du jeu ; pour rappel, un objet Jeu est créé comme suit :

```
1 mon_jeu = Jeu(initialisation, boucle)
```

Où `initialisation` et `boucle` sont deux fonctions qui doivent avoir été définies précédemment (voir la partie d'initiation au code) ; ces fonctions prendront en paramètre l'instance du jeu afin de pouvoir la modifier.

4.1.1.2. Gestion des événements

Variables passées à la boucle Lorsque le jeu est récupéré à l'intérieur de la boucle, il dispose de variables accessibles de façon directe (hors fonctions / méthodes) :

— `largeur`, qui contient la largeur de la fenêtre ;
— `hauteur`, qui contient la hauteur de la fenêtre ;
— `evenements`, qui contient la liste des événements déclenchés depuis le dernier tour de boucle.

La variable événements nous intéressera particulièrement, et nous pourrons vérifier si un certain événement est déclenché en utilisant une condition :

```
1 def boucle(jeu):
2     # [...]
3
```

```
    if Evenements.QUITTER in jeu.evenements:
        print("Je veux quitter")
```

Variables de la bibliothèque Dans la suite de ce guide, nous vous présenterons les fonctions ayant trait à la zone de jeu et au serpent. Toutefois, certains objets transcendent cette classification et ne peuvent aller dans aucune des deux catégories ; ces objets sont importés par défaut et ne requièrent donc pas l'appel par une méthode.

Le premier objet concerne la gestion des événements, et est naturellement nommé `Evenements` ; nous l'avons vu ci-dessus sans bien comprendre à quoi il correspondait ; cet objet contient deux variantes qui nous intéressent :

```
## Déclenché lorsque le joueur demande à fermer le jeu
Evenements.QUITTER
## Déclenché lorsqu'une touche est appuyée
Evenements.TOUCHE_APPUYEE
```

Récupération de la touche appuyée Une fois l'événement de `TOUCHE_APPUYEE` intercepté, il serait bon de savoir sur quelle touche le joueur a appuyé afin de déplacer le serpent dans la bonne direction ; pour cela, il suffit de tester `mon_jeu.evenements[Evenement.TOUCHE_APPUYEE]`, magique non ? La valeur peut alors être égale à (notons la seconde variable interne de la bibliothèque : `Touches`) :

```
Touches.FLECHE_DROITE
Touches.FLECHE_GAUCHE
Touches.FLECHE_HAUT
Touches.FLECHE_BAS
Touches.ESPACE
```

Pas plus de détails ici afin de vous faire chercher par vous-même ; n'oubliez pas qu'en informatique, il faut tester, vous ne pouvez rien casser.

4.1.2. Ajout d'images et de texte

4.1.2.1. Déclaration des variables internes Une fois l'objet de jeu créé, et récupéré dans une des fonctions `initialisation` ou `boucle`, il est possible de déclarer à tout moment une image par la méthode `ajouter_image`, appelée avec le nom à donner à l'image ainsi que le chemin relatif de l'image.

Le chemin relatif d'un fichier est l'endroit où il se trouve par rapport au fichier actuel. Par exemple, l'image `image.png` située dans le dossier `images` du dossier du fichier actuel est de chemin relatif `images/image.png`.

Puisqu'un exemple vaut mieux qu'un long discours :

```
def initialisation(jeu):
    # [...]

    # Déclaration de l'image du cactus
    jeu.ajouter_image("Cactus", "images/cactus.png")
```

La méthode permettant d'ajouter du texte fonctionne de la même façon, mais prend comme second paramètre le texte à afficher ; elle est nommée `ajouter_texte`.

4.1.2.2. Récupération et dessin

Afin de dessiner une image ou un texte précédemment déclaré, il faut faire appel à la fonction `dessin` ; cette fonction prend en paramètre :

1. le nom déclaré de l'image ou du texte ;
2. un objet de paramètres, pouvant prendre deux clefs : `position` et `rotation`, **toute valeur différente sera ignorée**.

Par exemple, pour dessiner un cactus (déclaré précédemment), en haut à gauche de l'écran, il suffit d'appeler la fonction :

```
def boucle(jeu):
    # [...]

    jeu.dessiner("Cactus", { "position": (0, 0) })
```

Une autre fonction de dessin utile est la fonction `effacer_ecran`, qui permet d'effacer entièrement le contenu de la zone de jeu ; elle ne prend aucun paramètre en entrée :

```
def boucle():
    # [...]

    jeu.effacer_ecran()
```

Nous n'avons pas vraiment parlé de la syntaxe avec des accolades utilisée ci-dessus dans la fonction `dessiner`, afin de ne pas trop alourdir l'introduction. Il s'agit de *dictionnaires*

en Python, et vous pouvez vous renseigner sur la documentation officielle en français afin de mieux comprendre ce qui se cache derrière cette syntaxe nouvelle .

4.1.3. Gestion des positions et collisions

4.1.3.1. Itérateur principal Dès le second objectif, vous aurez besoin de réaliser une boucle sur l'ensemble de la grille du jeu ; pour cela, un une fonction `grille` existe, et donne la position (x, y) d'un morceau de grille à chaque tour de boucle ; pour bien comprendre, voici un petit exemple d'utilisation :

```
1 def boucle(jeu):
2     # [...]
3
4     for carreau in jeu.grille():
5         print(carreau)
```

Ce code affichera la liste de tous les carreaux possibles dans la console.

4.1.3.2. Fonctions utiles Afin de vérifier une collision entre deux objets, il existe une fonction `collision` qui renvoie `True` (vrai) lorsqu'il y a collision et `False` (faux) sinon. Cette fonction prend deux paramètres, la position du premier objet, et la position du second ; cette fonction est principalement destinée à être utilisée dans des conditions; par exemple, ce code vérifiera s'il y a collision entre la `pomme` et la variable nommée `morceau.position`.

```
1 if jeu.collision(pomme, morceau.position):
2     print("Collision Serpent-Pomme")
```

Afin d'afficher les cactus sur un bord, vous pourriez avoir besoin de savoir si une position correspond ou non à un bord, pour cela, une fonction `jeu.est_un_bord()` existe, et prend comme unique paramètre la position à tester ; elle retourne un booléen (vrai ou faux, comme `jeu.collision()` au dessus). Pas d'exemple ici, on vous laisse essayer d'utiliser cette fonction par vous-même afin de bien comprendre son principe.

4.1.3.3. Gestion de la pomme Enfin, pour les plus avancés, vous pouvez essayer d'afficher une pomme à l'écran et de faire en sorte que le serpent puisse la manger. Pour générer la position de la pomme, deux solutions sont possibles :

— la solution simple, utiliser `jeu.position_aleatoire_pomme()`, fonction intégrée retournant simplement une position (x, y) aléatoire bien choisie ;
— la solution plus complexe, qui consiste à créer cette fonction par vous-même ; nous donnerons évidemment des indices aux participants arrivant jusqu'ici.

L'idée est de commencer par utiliser la fonction intégrée puis éventuellement de la (re)coder soi-même par la suite.

4.1.4. Fonctions diverses et bonus

Cette partie regroupe les fonctions qui ne trouvent leur place nulle part ailleurs ; l'une d'entre elle est très utile, les autres sont des bonus destinés aux participants les plus avancés.

4.1.4.1. Quitter le jeu La fonction très utile mentionnée ci-dessus est la fonction `quitter`, qui fermera simplement le jeu ; il sera nécessaire de l'utiliser lors de l'appui sur la croix de fermeture par le joueur, ou à la mort du serpent... éventuellement.

4.1.4.2. Fonctions bonus Quelques détails du jeu vous ont été cachés dans les parties précédentes, mais les objectifs bonus nous obligent à révéler certains de nos secrets les mieux gardés.

Tout d'abord, la fonction d'initialisation prend en réalité non pas deux mais bien quatre paramètres ; il est en effet possible de changer la taille de la fenêtre de jeu en passant en paramètres la largeur puis la hauteur, par exemple :

```
1 mon_jeu = Jeu(initialisation, boucle, 1024, 576)
```

créera une fenêtre plus grande que celle par défaut.

Il faudra veiller à ce que les deux paramètres de largeur et de hauteur soient des multiples de 32, sinon... à vous de tester.

La seconde fonction utile pour les bonus permet d'afficher du texte en plus grand que celui par défaut ; il s'agit de la fonction `init_text`. Cette méthode est, pour être exact, déjà appelée lors de l'initialisation du jeu (mais masquée, bien entendu). Elle permet de régler la police et la taille du texte ; un exemple pour comprendre :

```
1 def initialisation(jeu):
2     jeu.init_text("sans-serif", 32)
```

Cet appel rendra le texte en police sans-serif, et en taille 32, ce sont d'ailleurs les paramètres par défaut.

4.2. Fonctions du serpent

4.2.1. Présentation générale

4.2.1.1. Déclaration de l'objet Serpent Après avoir importé la bibliothèque Zeste de Code, vous aurez accès aux fonctions concernant le serpent. Une nouvelle instance de serpent devra ensuite être crée, préférentiellement à l'intérieur du jeu :

```
def initialisation(jeu):
    jeu.serpent = Serpent()
```

4.2.1.2. Constantes du serpent Dans l'objet Serpent précédemment créé, en plus des méthodes mentionnées ci-dessous, trois constantes (variables qui ne changent pas) sont très importantes.

Gestion de la direction La première, `jeu.serpent.DIRECTIONS` permet de donner au serpent une direction ; elle est particulièrement utilisée avec la fonction `deplacer`, qui la prend en argument. Cet objet se décline en cinq clefs :

```
## Le serpent va vers la droite
jeu.serpent.DIRECTIONS.DROITE

## Le serpent se déplace à gauche
jeu.serpent.DIRECTIONS.GAUCHE

## Le serpent monte
jeu.serpent.DIRECTIONS.HAUT

## Le serpent part en bas
jeu.serpent.DIRECTIONS.BAS

## Le serpent s'arrête (pas tout à fait une direction)
jeu.serpent.DIRECTIONS.STOP
```

Gestion des rotations Pour les plus avancés d'entre vous, il sera nécessaire de détecter quand le serpent tourne et dans quel sens. C'est à cet effet qu'a été créée la constante `ROTATIONS` :

```
## Le serpent tourne dans le sens des aiguilles d'une montre
jeu.serpent.ROTATIONS.HORAIRE

## Le serpent tourne dans le sens inverse
jeu.serpent.ROTATIONS.ANTI_HORAIRE
```

Parties du serpent Le serpent se décompose automatiquement en diverses parties :

— une tête, pour indiquer l'avant ;
— une queue, pour indiquer l'arrière ;
— le reste est composé de morceaux de corps.

Afin de détecter ces différentes parties, une constante `jeu.serpent.PARTIES` existe, et peut prendre les valeurs suivantes :

```
## Renseigne la tête du serpent (donc l'avant)
jeu.serpent.PARTIES.TETE

## Renseigne une partie de corps du serpent
jeu.serpent.PARTIES.CORPS

## Renseigne la queue du serpent (donc l'arrière)
jeu.serpent.PARTIES.QUEUE
```

Notons que lors de l'itération, les parties sont ordonnées dans le sens inverse.

Lorsque le serpent grandit, la bibliothèque ajoute automatiquement un morceau de corps juste après la tête, ce qui a pour effet de l'allonger.

4.2.2. Déplacement et taille

Dans cette partie, sont détaillées les fonctions concernant la taille du serpent ainsi que la gestion de ses déplacements.

4.2.2.1. Taille du serpent et agrandissement En premier lieu, voyons une variable très utile, qui contient la taille du serpent : `serpent.taille`. Par exemple, au début du jeu, cette variable vaudra 3.

Pour faire grandir le serpent, nous en avons parlé plus haut, il est possible d'appeler la fonction `serpent.grandir()`. Elle ne prend aucun paramètre et ne retourne rien, mais modifie en interne la taille du serpent en ajoutant en morceau juste après la tête.

4.2.2.2. Fonction de déplacement Afin de déplacer le serpent, il faut appeler la méthode `serpent.deplacer(direction)` avec direction, une des constantes de direction vues ci-dessus. Par exemple, pour déplacer le serpent d'une case vers le haut :

```
serpent.deplacer(jeu.serpent.DIRECTIONS.HAUT)
```

4.2.3. Position du serpent

Un itérateur existe et permet d'obtenir tous les morceaux de serpent dans l'ordre allant de la queue à la tête, il s'agit de `serpent.morceaux(taille)` ; le seul argument est la taille du serpent, ou plus simplement le nombre de morceaux à prendre. Pour afficher les positions successives du serpent :

```
def boucle(jeu):
    # [...]

    for morceau in jeu.serpent.morceaux(taille):
        print(morceau.position)
```

Comme nous pouvons le voir, morceau contient une information `position`, mais aussi une information `direction_rotation`, qui contient une des constantes de rotation vues ci-dessus, et enfin `type`, qui contient une constante de partie du serpent (tête, queue ou corps).

Pour obtenir la position de la tête du serpent, une variable peut vous simplifier la vie, c'est la fonction `serpent.position_tete`, qui contient simplement la position (x, y) de la tête du serpent.

5. Correction complète des exercices

Dans cette section, vous trouverez les énoncés des problèmes du Snake ainsi que leur correction complète. Nous vous conseillons de ne regarder la solution que par parties, afin d'essayer de faire l'exercice le plus possible par vous-même. Si quelqu'un peut vous aider, comme le jour du goûter, c'est encore mieux .

5.1. Premier objectif : Remplissage de la fenêtre

Le premier objectif consiste en la découverte du fonctionnement de la bibliothèque ; cet objectif reste très guidé par rapport aux suivants : il se veut comme un intermédiaire entre le cours et la partie en autonomie. Le jour de l'événement, cet objectif a été intégralement réalisé avec vous. Il est important pour bien réussir de comprendre ce que sont les fonctions `initialisation` et `boucle` ainsi que leur fonctionnement au sein du jeu.

L'ordre recommandé pour cet objectif est le suivant :

1. création des variables internes et affichage de la taille du serpent (noté SO1) ;
2. création de la fenêtre de jeu et gestion de la fermeture d'icelle (noté SO2).

Cet objectif est très varié, et peut donc vous prendre beaucoup de temps, car il nécessite la prise en main de la bibliothèque ; ne regardez la solution que si vous êtes réellement perdus, et si possible par petits morceaux, en essayant toujours de les comprendre.

Correction du premier objectif

5.2. Second objectif : affichage des éléments graphiques

Le second objectif vise à afficher les éléments graphiques principaux que sont les cactus et le serpent. Il présente le fonctionnement des boucles, et initie à la gestion de tableaux ; il est très peu guidé, et doit donc être réalisé en utilisant correctement la documentation. La réalisation des trois sous-objectifs peut être longue et demander beaucoup de patience, mais elle est nécessaire afin de poser les bases. Les rudiments de la programmation vus précédemment (structures conditionnelles et variables) seront aussi réutilisés afin de les consolider.

L'objectif principal se décline en trois sous-objectifs, dont l'un a été réalisé lors de la présentation orale :

— affichage des cactus (noté SO1) ;
— affichage du serpent comme un ensemble de corps (noté SO2) ;
— affichage du serpent avec une tête et une queue (noté SO3).

La correction complète peut être trouvée ci-dessous :

Correction du second objectif

5.3. Troisième objectif : animation du serpent

Le troisième objectif, ayant trait à la gestion des animations du serpent par les interactions du joueur, complexifie beaucoup le code contrairement aux précédents. Il ajoute en effet une grosse partie de gestion des événements afin de vérifier l'appui des différentes touches de mouvement (les flèches directionnelles) ; la recherche dans la documentation sera ainsi très importante, mais aucune notion nouvelle n'est abordée à partir d'ici.

Les deux sous-objectifs visés ici sont :

— animation du serpent vers la droite (noté SO1) ;
— animation dans les autres directions (noté SO2).

Le corrigé complet est disponible ci-dessous :

Correction du troisième objectif

5.4. Dernier objectif : gestion des collisions

Cet objectif est très complet et revient sur tout ce qui a été vu avant, et vous permettra d'acquérir une vue globale du code écrit ci-avant. En plus de cela, l'objectif finalise en quelque sorte le jeu en y ajoutant toutes les collisions nécessaires, ainsi que la pomme (enfin !).

Quatre sous-objectifs sont visés, et peuvent nécessiter du temps :

— ajout d'une pomme sur la grille, à une position aléatoire (noté SO1) ;
— gestion des collisions entre le serpent et la pomme, et changement de position d'icelle (noté SO2) ;
— gestion des collisions entre le serpent et le bord (noté SO3) ;
— gestion des collisions entre le serpent et lui-même (difficile, noté SO4).


Correction du quatrième objectif

5.5. Quelques bonus... sans corrigé !

Pour les plus intéressés, il reste encore quelques améliorations à apporter à notre jeu, nous en proposons ici quatre, mais il est possible de rajouter encore de nombreuses choses !

Le premier bonus, plutôt simple, résout un problème présent dans les objectifs précédents : le serpent peut se mordre en allant en arrière. Or, dans la plupart des jeux Snake, il n'est pas possible d'aller vers l'arrière ; vous êtes donc invités à résoudre ce problème.

Le second objectif bonus ajoute un compteur de points (qui est simplement la taille du serpent moins la taille initiale). L'idée est ici de se familiariser avec les fonctions de texte, particulièrement la fonction `ajouter_texte`, qui fonctionne de la même façon que `ajouter_image`.

Le troisième bonus, un peu plus complexe, vise à utiliser ces fonctions de texte afin d'afficher au joueur son score en fin de partie et lui permettre de rejouer.

Enfin, les participants les plus doués pourront proposer une solution pour gérer les rotations du serpent proprement en utilisant l'image `coin.png` disponible dans le dossier `images`. Cet objectif est complexe car il demande une bonne compréhension du fonctionnement des objets, ainsi qu'une idée (au moins vague), de la façon dont est géré le dessin du serpent en interne (vous êtes invités à aller regarder le code de la bibliothèque, entièrement commenté en français !)


Et maintenant

Au delà de ces quelques propositions, c'est votre jeu, donc **vous pouvez faire absolument ce que vous voulez de plus ou moins complexe !** N'hésitez pas à aller voir le code de la bibliothèque si vous voulez comprendre le fonctionnement interne que l'on vous a caché au début, ou à expérimenter vos propres idées aussi farfelues que les limites de votre imagination !
Aussi, nous vous présentions un Snake, mais **vous pouvez aussi tenter de faire vous même de zéro d'autres jeux 2D**, existants ou sortant tout droit de votre imagination. Pour vous proposer le Snake, nous avons utilisé un outil que l'on vous a masqué derrière

notre propre bibliothèque : PyGame, qui est justement conçu pour la réalisation de jeux en deux dimensions. Zeste de Savoir propose un début de cours sur l'utilisation de PyGame , et vous pouvez aller également consulter des exemples et la documentation sur le site officiel .
Et bien sûr, il est possible d'aller beaucoup plus loin que ça : sites web, logiciels de bureau, jeux 3D (beaucoup plus complexes par contre), jeux en réseau (là aussi c'est plus complexe !), et autres concepts encore plus avancés (IA, calculs scientifiques, traitement d'images (pensez aux filtres Instagram), par exemple — mais cette liste est loin d'être exhaustive)… Si vous vous y intéressez, c'est pas les possibilités qui manquent !

6. Conclusion

Les organisateurs remercient les participant · e · s de ce second Zeste de Code et s'excusent de n'avoir pas pu terminer le Snake le jour de l'événement, même si celui-ci était très bien avancé pour certain · e · s. Nous remercions @Situphen et @Ekron, qui malgré leur absence le jour de l'évènement, ont contribué malgré tout à la réalisation de cet atelier.

Pour toute question, n'hésitez pas à poster sur le forum de Zeste de Savoir – catégorie Programmation – ou à contacter les organisateurs par MP ; leurs pseudos figurent en haut de ce billet : @**Etoile Filante**, @TAlone, @Bibou, @Amaury.

Contenu masqué

Contenu masqué n°1 : Correction du premier objectif

```
1  ########################################
2  ###         Premier objectif         ###
3  ###      (c) Zeste de Savoir (c)     ###
4  ###           Licence GPL            ###
5  ###         Auteur : TAlone          ###
6  ########################################
7
8  # Le premier objectif consiste en la découverte du fonctionnement
       de la bibliothèque :
9  # (sous-objectif 1) - création des variables internes et affichage
       de la taille du serpent ;
10 # (sous-objectif 2) - création de la fenêtre de jeu et gestion de
       la fermeture d'icelle.
11
12 # Import de la bibliothèque de Zeste de Code (fonctions
       d'abstraction)
```

Contenu masqué

```
13 from bibliotheque import *
14
15 # (OP) Fonction principale d'initialisation
16 def initialisation(jeu):
17         # (SO1) Création d'un serpent à l'écran
18         jeu.serpent = Serpent()
19
20         # (SO2) Ajout de l'image spéciale qui sera automatiquement
               mise en fond
21         jeu.ajouter_image("fond", "images/fond.png")
22
23 # (OP) Fonction executée regulièrement
24 def boucle(jeu):
25         # (SO2) Fermeture du jeu lors de l'appui sur la croix de la
               fenêtre
26         if Evenements.QUITTER in jeu.evenements:
27                 jeu.quitter()
28
29         # (SO2) Effacement de l'écran, et remplissage avec les
               tiles de fond
30         jeu.effacer_ecran()
31
32         # (SO1) Déclaration d'une variable contenant la taille du
               serpent
33         taille = jeu.serpent.taille
34
35         # (SO1) Affichage de la variable de taille
36         print(taille)
37
38 # (OP) Lancement du jeu à partir des fonctions d'abstraction
39 Jeu(initialisation, boucle)
```

Retourner au texte.

Contenu masqué n°2 : Correction du second objectif

```
1 ########################################
2 ###          Second objectif         ###
3 ###      (c) Zeste de Savoir (c)     ###
4 ###           Licence GPL            ###
5 ###         Auteur : TAlone          ###
6 ########################################
7
8 # Ce second objectif à pour objet d'afficher les bordures et le
      serpent :
```

Contenu masqué

```
9  # (sous-objectif 1) - affichage des cactus ;
10 # (sous-objectif 2) - affichage du serpent comme un ensemble de
       corps ;
11 # (sous-objectif 3) - affichage du serpent avec une tête et une
       queue.
12
13 # Import de la bibliothèque de Zeste de Code (fonctions
       d'abstraction)
14 from bibliotheque import *
15
16 # Fonction principale d'initialisation
17 def initialisation(jeu):
18         # Création d'un serpent à l'écran
19         jeu.serpent = Serpent()
20
21         # (SO1) Déclaration du cactus
22         jeu.ajouter_image("Cactus", "images/cactus.png")
23
24         # (SO2) Ajout du corps
25         jeu.ajouter_image("Corps", "images/corps.png")
26         # (SO3) Ajout de la queue
27         jeu.ajouter_image("Queue", "images/queue.png")
28         # (SO3) Ajout de la tête
29         jeu.ajouter_image("Tête", "images/tete.png")
30
31         # Ajout de l'image spéciale qui sera automatiquement mise
               en fond
32         jeu.ajouter_image("fond", "images/fond.png")
33
34 # Fonction exécutée régulièrement
35 def boucle(jeu):
36         # Fermeture du jeu lors de l'appui de la croix
37         if Evenements.QUITTER in jeu.evenements:
38                 jeu.quitter()
39
40         # Effacement de l'écran, et remplissage avec les images de
               fond
41         jeu.effacer_ecran()
42
43         # (SO1) Itération sur tous les morceaux de grille
44         for carreau in jeu.grille():
45                 # (SO1) Si l'on est sur un côté...
46                 if jeu.est_un_bord(carreau):
47                         # (SO1) ...dessine un cactus
48                         jeu.dessiner("Cactus", { "position":
                               carreau })
49
50         # Déclaration d'une variable contenant la taille du serpent
51         taille = jeu.serpent.taille
52
```

```
53          # (SO2) Dessin d'un certain nombre de morceaux à l'écran
54          for morceau in jeu.serpent.morceaux(taille):
55                  # (SO3) Choix de l'image en fonction de la partie à
                        dessiner et dessin à l'écran
56                  if morceau.type == jeu.serpent.PARTIES.TETE:
57                          jeu.dessiner("Tête", morceau)
58                  elif morceau.type == jeu.serpent.PARTIES.QUEUE:
59                          jeu.dessiner("Queue", morceau)
60                  else:
61                          # (SO2) Dessin du corps
62                          jeu.dessiner("Corps", morceau)
63
64  # Lancement du jeu à partir des fonctions d'abstraction
65  Jeu(initialisation, boucle)
```

Retourner au texte.

Contenu masqué n°3 : Correction du troisième objectif

```
1   ########################################
2   ###         Troisième objectif       ###
3   ###      (c) Zeste de Savoir (c)     ###
4   ###           Licence GPL            ###
5   ###         Auteur : TAlone          ###
6   ########################################
7
8   # Le troisième objectif vise à animer le serpent
9   # (sous-objectif 1) - en ligne droite ;
10  # (sous-objectif 2) - dans les autres directions.
11
12  # Import de la bibliothèque de Zeste de Code (fonctions
        d'abstraction)
13  from bibliotheque import *
14
15  # Fonction principale d'initialisation
16  def initialisation(jeu):
17          # Création d'un serpent à l'écran
18          jeu.serpent = Serpent()
19
20          # (SO1) Donne une direction par défaut au serpent
21          jeu.direction_serpent = jeu.serpent.DIRECTIONS.STOP
22
23          # Déclaration du cactus
24          jeu.ajouter_image("Cactus", "images/cactus.png")
25
```

```
26          # Ajout du corps
27          jeu.ajouter_image("Corps", "images/corps.png")
28          # Ajout de la queue
29          jeu.ajouter_image("Queue", "images/queue.png")
30          # Ajout de la tête
31          jeu.ajouter_image("Tête", "images/tete.png")
32
33          # Ajout de l'image spéciale qui sera automatiquement mise
                en fond
34          jeu.ajouter_image("fond", "images/fond.png")
35
36  # Fonction exécutée régulièrement
37  def boucle(jeu):
38          # Fermeture du jeu lors de l'appui de la croix
39          if Evenements.QUITTER in jeu.evenements:
40                  jeu.quitter()
41
42          # (SO1) Lorsqu'une touche est appuyée
43          if Evenements.TOUCHE_APPUYEE in jeu.evenements:
44                  # (SO1) On stocke la touche appuyée
45                  touche = jeu.evenements[Evenements.TOUCHE_APPUYEE]
46
47                  # (SO1) ...on vérifie la direction droite...
48                  if touche == Touches.FLECHE_DROITE:
49                          # (SO1) ...et on note que si elle est
                                enfoncée, le serpent doit désormais
                                aller à droite.
50                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.DROITE
51                  # (SO2) Ensuite on fait de même avec les autres
                        directions.
52                  elif touche == Touches.FLECHE_HAUT:
53                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.HAUT
54                  elif touche == Touches.FLECHE_BAS:
55                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.BAS
56                  elif touche == Touches.FLECHE_GAUCHE:
57                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.GAUCHE
58
59          # Effacement de l'écran, et remplissage avec les images de
                fond
60          jeu.effacer_ecran()
61
62          # Itération sur tous les morceaux de grille
63          for carreau in jeu.grille():
64                  # (SO1) Si l'on est sur un côté...
65                  if jeu.est_un_bord(carreau):
66                          # (SO1) ...dessine un cactus
```

Contenu masqué

```
67                          jeu.dessiner("Cactus", { "position":
                                carreau })
68
69          # Déclaration d'une variable contenant la taille du serpent
70          taille = jeu.serpent.taille
71
72          # Dessin d'un certain nombre de morceaux à l'écran
73          for morceau in jeu.serpent.morceaux(taille):
74                  # Choix de l'image en fonction de la partie à
                        dessiner et dessin à l'écran
75                  if morceau.type == jeu.serpent.PARTIES.TETE:
76                          jeu.dessiner("Tête", morceau)
77                  elif morceau.type == jeu.serpent.PARTIES.QUEUE:
78                          jeu.dessiner("Queue", morceau)
79                  else:
80                          # Dessin du corps
81                          jeu.dessiner("Corps", morceau)
82
83          # (SO1) Déplacement du serpent
84          jeu.serpent.deplacer(jeu.direction_serpent)
85
86  # Lancement du jeu à partir des fonctions d'abstraction
87  Jeu(initialisation, boucle)
```

Retourner au texte.

Contenu masqué n°4 : Correction du quatrième objectif

```
 1  ########################################
 2  ###         Quatrième objectif       ###
 3  ###      (c) Zeste de Savoir (c)     ###
 4  ###           Licence GPL            ###
 5  ###         Auteur : TAlone          ###
 6  ########################################
 7
 8  # Il est dans ce quatrième objectif question des collisions :
 9  # (sous-objectif 1) - objectif intermédiaire visant à ajouter la
        pomme ;
10  # (sous-objectif 2) - entre le serpent et la pomme ;
11  # (sous-objectif 3) - entre le serpent et le bord ;
12  # (sous-objectif 4) - entre le serpent et lui-même.
13
14  # Import de la bibliothèque de Zeste  de Code (fonctions
        d'abstraction)
15  from bibliotheque import *
```

Contenu masqué

```
16
17 # Fonction principale d'initialisation
18 def initialisation(jeu):
19         # Création d'un serpent à l'écran
20         jeu.serpent = Serpent()
21
22         # (SO1) Création d'une pomme
23         jeu.pomme = jeu.position_aleatoire_pomme()
24
25         # Donne une direction par défaut au serpent
26         jeu.direction_serpent = jeu.serpent.DIRECTIONS.STOP
27
28         # Déclaration du cactus
29         jeu.ajouter_image("Cactus", "images/cactus.png")
30         # (SO1) Déclaration de la pomme
31         jeu.ajouter_image("Pomme", "images/pomme.png")
32
33         # Ajout du corps
34         jeu.ajouter_image("Corps", "images/corps.png")
35         # Ajout de la queue
36         jeu.ajouter_image("Queue", "images/queue.png")
37         # Ajout de la tête
38         jeu.ajouter_image("Tête", "images/tete.png")
39
40         # Ajout de l'image spéciale qui sera automatiquement mise
               en fond
41         jeu.ajouter_image("fond", "images/fond.png")
42
43 # Fonction exécutée régulièrement
44 def boucle(jeu):
45         # Fermeture du jeu lors de l'appui de la croix
46         if Evenements.QUITTER in jeu.evenements:
47                 jeu.quitter()
48
49         # Lorsqu'une touche est appuyée
50         if Evenements.TOUCHE_APPUYEE in jeu.evenements:
51                 # On stocke la touche appuyée
52                 touche = jeu.evenements[Evenements.TOUCHE_APPUYEE]
53
54                 # ...on vérifie la direction droite...
55                 if touche == Touches.FLECHE_DROITE:
56                         # ...et on note que si elle est enfoncée,
                               le serpent doit désormais aller à
                               droite.
57                         jeu.direction_serpent =
                               jeu.serpent.DIRECTIONS.DROITE
58                 # On fait de même avec les autres directions.
59                 elif touche == Touches.FLECHE_HAUT:
60                         jeu.direction_serpent =
                               jeu.serpent.DIRECTIONS.HAUT
```

Contenu masqué

```
61                  elif touche == Touches.FLECHE_BAS:
62                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.BAS
63                  elif touche == Touches.FLECHE_GAUCHE:
64                          jeu.direction_serpent =
                                jeu.serpent.DIRECTIONS.GAUCHE
65
66          # (SO2) Si il y a collision entre la pomme et la tête du
                serpent
67          if jeu.collision(jeu.serpent.position_tete,
                jeu.pomme.position):
68                  # (SO2) Le serpent grandit
69                  jeu.serpent.grandir()
70                  # (SO2) La pomme change de position
71                  jeu.pomme = jeu.position_aleatoire_pomme()
72
73          # Effacement de l'écran, et remplissage avec les images de
                fond
74          jeu.effacer_ecran()
75
76          # Itération sur tous les morceaux de grille
77          for carreau in jeu.grille():
78                  # Si l'on est sur un côté...
79                  if jeu.est_un_bord(carreau):
80                          # ...dessine un cactus
81                          jeu.dessiner("Cactus", { "position":
                                carreau })
82
83                          # (SO3) Si le coin est en contact avec la
                                tête du serpent
84                          if jeu.collision(carreau,
                                jeu.serpent.position_tete):
85                                  # (SO3) Ferme le jeu si il y a
                                        contact entre la tête et un
                                        bord
86                                  jeu.quitter()
87
88          # Déclaration d'une variable contenant la taille du serpent
89          taille = jeu.serpent.taille
90
91          # Dessin d'un certain nombre de morceaux à l'écran
92          for morceau in jeu.serpent.morceaux(taille):
93                  # Choix du sprite en fonction de la partie à
                        dessiner et dessin à l'écran
94                  if morceau.type == jeu.serpent.PARTIES.TETE:
95                          jeu.dessiner("Tête", morceau)
96                  elif morceau.type == jeu.serpent.PARTIES.QUEUE:
97                          jeu.dessiner("Queue", morceau)
98                  else:
99                          # Dessin du corps
```

Contenu masqué

```
100                         jeu.dessiner("Corps", morceau)
101
102                 # (SO4) Vérification que le serpent ne se mord pas
103                 # Attention : la tête est incluse dans la liste des
                        morceaux, il faut donc la retirer
104                 if jeu.collision(jeu.serpent.position_tete,
                        morceau.position) and morceau.type !=
                        jeu.serpent.PARTIES.TETE:
105                         # (SO4) Ferme le jeu si le serpent se
                                rentre dedans
106                         jeu.quitter()
107
108         # (SO1) Dessin de la pomme
109         jeu.dessiner("Pomme", jeu.pomme)
110
111         # Déplacement du serpent
112         jeu.serpent.deplacer(jeu.direction_serpent)
113
114 # Lancement du jeu à partir des fonctions d'abstraction
115 Jeu(initialisation, boucle)
```

Retourner au texte.