

Python 3 : objectif jeux



Didier Müller, juin 2020

www.apprendre-en-ligne.net



Introduction

0.1. Buts et moyens

Comme son nom l'indique, ce livre a pour but d'apprendre le langage Python en programmant principalement des jeux, qui serviront de prétexte pour découvrir les différentes facettes du langage Python. La difficulté sera progressive : on commencera par des jeux très simples et sans fioritures, pour aller vers des jeux de plus en plus « jolis » et sophistiqués. Chaque jeu permettra d'introduire de nouvelles notions, tout en répétant celles déjà vues auparavant.

0.2. Programmer ?

La **programmation** consiste à « expliquer » en détails à un ordinateur ce qu'il doit faire, sachant qu'il ne comprend évidemment pas une langue humaine, mais seulement ce qu'on appelle un **langage de programmation** (par exemple C, Python, Java, etc.). En d'autres termes, il faudra traduire une idée simple (par exemple « trier des nombres dans l'ordre croissant »), en un vrai raisonnement parfaitement structuré et détaillé, que l'on appelle un **algorithme**.

Un langage de programmation a de nombreux points communs avec un langage humain : il a une syntaxe (l'orthographe des mots) et une grammaire (la façon d'agencer les mots). La différence la plus importante est qu'un langage informatique ne tolère **aucune** erreur, alors que l'on peut mal parler une langue tout en se faisant comprendre quand même.



0.2.1. Types d'erreurs

La programmation est une tâche complexe, et on y commet de nombreuses erreurs.

Erreurs de syntaxe

Un programme ne peut être exécuté que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête (on parle communément de « plantage ») et vous obtenez un message d'erreur. Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme. **Tous** les détails ont de l'importance : le respect des majuscules et des minuscules, l'orthographe, la ponctuation, ...

Erreurs sémantiques

Le second type d'erreur est l'erreur sémantique ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, il n'y aura aucun message d'erreur, mais le résultat ne sera pas celui que vous attendiez.

Erreurs à l'exécution

Le troisième type d'erreur est l'erreur en cours d'exécution (*Run-time error*), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se

Pour des raisons anecdotiques, les erreurs de programmation s'appellent des « bugs ». *bug* est à l'origine un terme anglais servant à désigner de petits insectes gênants, tels les punaises. Il est arrivé à plusieurs reprises que des cadavres de ces insectes provoquent des court-circuits et donc des pannes incompréhensibles.

présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus).



Recherche des erreurs et expérimentation

Débuguer efficacement un programme demande beaucoup de perspicacité et ce travail ressemble à une enquête policière. Vous examinez les résultats, et vous devez émettre des hypothèses pour reconstituer les processus et les événements qui ont logiquement entraîné ces résultats.

0.2.2. Les 4 règles d'or pour bien programmer

Programmer ne suffit pas. Il faut **bien** programmer, de sorte que quelqu'un qui voudra réutiliser votre programme puisse le faire facilement. En un mot, le programme doit être **compréhensible**. La première chose consistera à **décomposer un problème compliqué en plusieurs sous-problèmes simples**, donc à découper le programme en plusieurs sous-programmes.

Voici les règles d'or à respecter :



Règle numéro 1.

Ne pas écrire de longs sous-programmes (pas plus de 10-15 lignes de code).



Règle numéro 2.

Chaque sous-programme doit avoir un objectif clair.



Règle numéro 3.

Écrire des commentaires pour expliquer les parties les plus subtiles du programme.



Règle numéro 4 (jamais respectée).

Éviter le copier/coller, source d'innombrables erreurs.

Vous trouverez sur le site web compagnon (voir § 0.4) un extrait des PEP8 PEP257 (Python Extension Proposal) originaux qui reprend les points importants pour l'apprentissage des bonnes méthodes de programmation en Python (voir § 0.3). Les auteurs originaux sont Guido **van Rossum** et Barry **Warsaw**.

0.2.3. Citations de sagesse

« L'enseignement de l'informatique ne peut faire de personne un programmeur expert plus que l'étude des pinceaux et du pigment peut faire de quelqu'un un peintre expert. » - Eric S. Raymond

« Programmer, c'est comme se donner des coups de pied dans le visage, tôt ou tard, votre nez va saigner. » - Kyle Woodbury

« Je ne suis pas un excellent programmeur. Je suis juste un bon programmeur avec d'excellentes habitudes. » - Kent Beck

« N'importe quel idiot peut écrire du code qu'un ordinateur peut comprendre. Les bons programmeurs écrivent du code que les humains peuvent comprendre. » - Martin Fowler

« La vérité ne peut être trouvée qu'à un endroit : le code. » - Robert C. Martin



Guido von Rossum

0.3. Python

Python est un langage portable, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par Guido **van Rossum** et de nombreux contributeurs bénévoles.

Il est apprécié par les pédagogues qui y trouvent un langage où la syntaxe permet une initiation aisée aux concepts de base de la programmation. Enfin, le langage Python gagne en popularité car il est le langage favori des « data scientists », notamment. Il permet également de faire du Web aisément avec *Django*.

0.3.1. Principales caractéristiques du langage

Détaillons quelques caractéristiques de Python :

- Python est **portable** : il fonctionne non seulement sur Linux, mais aussi sur MacOS, et les différentes variantes de Windows.
- Python est **gratuit**, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.
- La **syntaxe** de Python est très simple et, combinée à des **types de données évolués** (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. À fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python est **orienté objet**. Il supporte l'**héritage multiple** et la **surcharge des opérateurs**.
- Python est un langage qui **continue à évoluer**, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Nous utiliserons dans ce cours la **version 3**. Il est à noter qu'un programme écrit en Python 2 n'est pas compatible avec la version 3, et nécessitera quelques modifications.

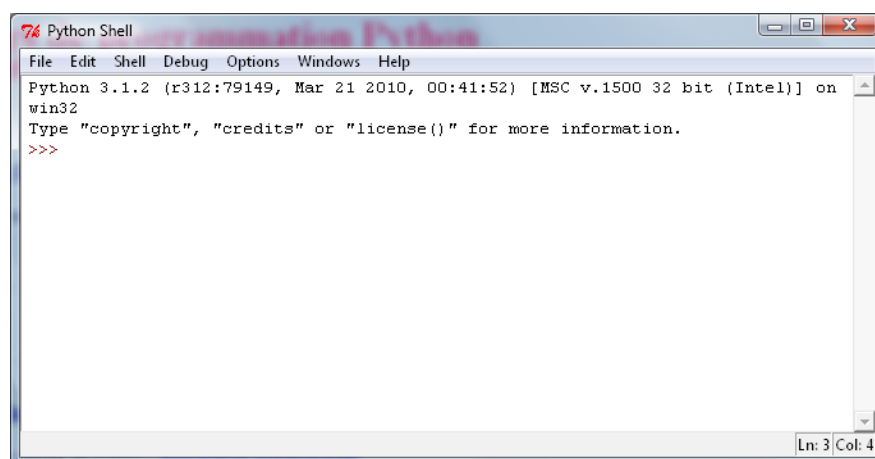
0.3.2. Installation de Python 3

Un guide d'installation est disponible sur le site compagnon (voir § 0.4).

Allez sur le site officiel de Python : www.python.org/download/

1. Choisissez la dernière version (non bêta) adaptée à votre système d'exploitation.
2. Dans le programme d'installation, le plus simple est de cliquer chaque fois sur *Next* jusqu'au bouton *Finish*.
3. Python est maintenant installé. Vous le trouverez dans votre liste de programmes. Pour le démarrer, choisissez dans le répertoire Python le programme **IDLE (Python GUI)**. La fenêtre suivante devrait alors apparaître :

La version (ici 3.1.2) peut varier.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.2 (r312:79149, Mar 21 2010, 00:41:52) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

0.4. Site web compagnon

Vous trouverez sur le site associé à ce cours les programmes des jeux, afin d'éviter de perdre du temps à les recopier et afin de pouvoir les tester facilement. Vous trouverez aussi les corrigés des exercices et les ressources à utiliser.

L'adresse est www.apprendre-en-ligne.net/pj/

0.5. Contenu du cours

Dans ce cours, les jeux et autres amusements auront une place prépondérante, même si, de temps à autre, surtout dans les premiers chapitres, on trouvera des exercices concernant d'autres sujets.

Première partie : bases de Python

Au chapitre 1, nous aborderons un élément essentiel des jeux : le **hasard**. Nous aborderons du même coup en douceur quelques concepts importants de la programmation : l'affichage, les opérateurs, les boucles et les conditions.

Au chapitre 2, le jeu **Devine mon nombre** permettra de voir deux concepts fondamentaux de la programmation : les conditions et les boucles. Il sera aussi question de variables. On calculera plus loin les probabilités de perdre des armées lors de combats au **Risk**.

Au chapitre 3, on jouera à **Pierre, papier, ciseaux** contre l'ordinateur, ce qui nous permettra d'introduire les procédures et les fonctions.

On retrouvera ce jeu au chapitre 4, mais cette fois dans une version graphique. On calculera aussi des probabilités avec **Let's make a deal !**

Au chapitre 5, nous découvrirons le **jeu de Juniper Green**, où nous devrons utiliser des listes. Nous analyserons aussi le célèbre jeu pour enfants **Le verger**.

Au chapitre 6, nous aborderons un des piliers de la théorie des jeux : le **dilemme du prisonnier**, qui nous permettra de découvrir les dictionnaires et les fonctions lambda.

Au chapitre 7, nous utiliserons plusieurs jeux de lettres : le **pendu**, le **mot le plus long**, le **Scrabble** et **Motus**, ce qui nous amènera tout naturellement à étudier les fichiers et les chaînes de caractères.

Deuxième partie : graphisme

Nous programmerons une version graphique du pendu au chapitre 8.

Le chapitre 9 est un peu spécial, puisqu'il faudra écrire un programme qui nous permettra de jouer au **Memory** contre l'ordinateur et qu'il pourra déboucher sur un travail de groupe.

Dans le chapitre 10, nous utiliserons le **Blackjack** pour introduire l'importante notion de programmation orientée objet.

Le chapitre 11, exceptionnellement, ne concernera pas directement les jeux, mais sera quand même amusant : il sera en effet question de dessins et nous verrons comment utiliser des visages stylisés pour représenter des données statistiques.

Qui dit jeux dit souvent damiers. Au chapitre 12, nous approfondirons le chapitre précédent en dessinant différents damiers (qui nous permettront de programmer le jeu **Le loup et les moutons** et **la course de la dame**) et nous nous prendrons un moment pour le peintre Vasarely.

Le chapitre 13 traitera d'un classique de la programmation pour débutants : les automates cellulaires, et en particulier le célèbre **jeu de la vie** de **Conway**. Nous programmerons aussi le casse-tête **Gasp**.

Au chapitre 14, nous aborderons la récursivité avec le célèbre jeu du **démineur**. Nous verrons aussi comment **sortir d'un labyrinthe**.

Le chapitre 15 sera la cerise sur le gâteau. En effet, nous allons programmer un **jeu d'échecs**. En fait, pas tout à fait. Nous allons faire ce qui se fait très souvent en informatique : partir d'un programme (relativement simple) disponible sur le web, puis l'améliorer. Cela demandera évidemment de le comprendre avant de le retoucher... Nous créerons dans un premier temps une interface graphique, puis, dans un deuxième temps, nous essaierons de le rendre plus fort.

Enfin, au chapitre 16, nous verrons comment bouger des images lors d'une **course d'escargots**, et d'un **Space Invaders** simplifié. Nous retrouverons aussi avec nostalgie le premier jeu vidéo à avoir connu un succès populaire : **Pong**. Finalement, nous verrons comment simuler une **planche de Galton**.

0.6. Remarques sur les exercices

Ce livre est destiné à l'enseignement dans des classes. L'une des difficultés de l'enseignement de l'informatique est que chaque élève a son propre rythme, et que les niveaux des élèves lors des premières leçons peuvent être très différents. Cela signifie qu'après quelques leçons déjà, certains élèves peuvent avoir plusieurs chapitres d'avance sur les plus lents. Pour y remédier, il faut avoir quelques exercices supplémentaires pour occuper les plus rapides.

En face de chaque exercice, vous trouverez une de ces trois icônes :



indique un exercice relativement facile que tous les élèves devront faire.



indique un exercice que tous les élèves devront faire, mais qui est un peu plus difficile. S'ils y passent trop de temps, ils auront avantage à essayer de regarder et comprendre le corrigé.



indique un exercice difficile que les élèves le plus rapides pourront faire s'ils le souhaitent, en attendant les élèves les plus lents.

De plus, 256 exercices plus courts sont disponibles dans le **Défi Turing**, qui propose des problèmes de maths où l'informatique sera souvent nécessaire.

L'adresse est www.apprendre-en-ligne.net/turing/

Presque toutes les bases théoriques pour faire ces exercices seront acquises à la fin du chapitre 7.





0.7. Ce que vous avez appris dans l'introduction

- Ce qu'est la programmation et quels sont les types d'erreurs que vous rencontrerez.
- Les 4 règles d'or pour bien programmer.
- Les principales caractéristiques de Python 3 et comment l'installer.



Chapitre 1

Le hasard

1.1. Thèmes abordés dans ce chapitre

- les nombres pseudo-aléatoires
- le module `random`
- `print(...)`
- la boucle `for ... in range(...)`
- la condition `if ... elif ... else`
- opérateurs de comparaisons

1.2. Le hasard dans les jeux

Dans beaucoup de jeux, le hasard a une place plus ou moins importante. Un **jeu de hasard** est un jeu dont le déroulement est partiellement ou totalement soumis à la chance. Celle-ci peut provenir d'un tirage ou d'une distribution de cartes, d'un jet de dé, etc. Lorsque le jeu est totalement soumis au hasard, on parle de jeu de hasard pur. Lorsque le joueur doit déterminer son action en fonction d'événements aléatoires passés ou futurs et de probabilités, on parle plus volontiers de **jeu de hasard raisonné**.

1.2.1. Jeux de hasard pur

- La **bataille** se joue avec un jeu de 32 cartes ou de 52 cartes. Chaque joueur joue la première carte de son paquet mélangé aléatoirement et il existe une règle pour chaque combinaison de carte. Les joueurs n'ont alors rien à décider.
- Le **jeu de l'oie** est un jeu de plateau contenant un chemin de cases et se jouant avec deux dés. Les règles pour chaque case sont fixées, le joueur ne décide donc rien.
- Le **jeu de la Loterie** (ou Loto) se joue en pariant sur plusieurs nombres (choisis ou aléatoires). Chaque nombre est choisi de manière aléatoire et de manière équiprobable.

1.2.2. Jeux de hasard raisonné

- Le jeu de **poker** se joue avec un jeu de 52 cartes. La distribution des cartes est le seul élément aléatoire du jeu. La manière de miser, parier, bluffer, etc. est au choix du joueur.
- Dans la plupart des jeux de cartes, comme le **bridge** ou le **jass**, la distribution des cartes est l'élément aléatoire.
- Les cartes du jeu **Magic : l'assemblée** sont des cartes spéciales pour ce jeu, avec des actions spécifiques pour chacune d'entre elles. Le hasard provient du mélange de ces cartes, le paquet de cartes étant construit par les joueurs eux-mêmes. Pour certaines cartes, on utilise également un dé ou un jeu de pile ou face.

- Le **Backgammon** se joue sur un tablier (plateau) avec quinze pions et deux dés. L'avancement des pions sur le tablier s'effectue en fonction des valeurs des dés, mais le joueur choisit quels pions il avance.
- Dans le jeu de **Monopoly**, l'avancement des pions se fait de manière aléatoire avec un dé, des cartes actions sont tirées de manière aléatoire, cependant les stratégies d'achat de terrains et de maisons se font par les joueurs.
- Le **Yam's** (ou **Yahtzee**) se joue avec cinq dés. Le but est de réaliser des figures en trois lancers de dés. Le joueur choisit quels dés il relance.
- Dans certains jeux comme le **blackjack**, le joueur joue contre la banque qui a un jeu totalement aléatoire.

1.2.3. Autres

La stratégie optimale du jeu de pierre-papier-ciseaux (lorsque son adversaire joue de manière aléatoire équiprobable), est également de jouer de manière aléatoire équiprobable. Le jeu est alors un jeu de hasard pur. Cependant il est difficile de générer une suite de valeurs aléatoires par soi-même. Ainsi, si votre adversaire joue avec une certaine stratégie, vous pouvez chercher à obtenir une stratégie optimale. On peut parler de jeu de hasard raisonné. Ce jeu échappe donc au classement jeu de hasard pur/jeu de hasard raisonné.

Il existe évidemment aussi des jeux où le hasard n'a pas sa place : les échecs, les dames, le morpion, etc. Quoique... il faut bien déterminer d'une manière ou d'une autre qui commence.

1.3. Comment générer du hasard sur ordinateur?

En informatique, le « vrai » hasard n'existe pas ! On ne fait qu'imiter le hasard. C'est moins facile que l'on pourrait le croire, car un ordinateur n'improvise pas : il ne sait que suivre son programme. Il lui est donc difficile de produire, à partir d'une procédure purement mathématique, des chiffres réellement aléatoires de façon totalement imprévisible. Si on connaît la procédure mathématique, on peut obtenir la même suite de nombres. Ce n'est donc pas du hasard. Il faut se contenter de produire des séquences de nombres qui ont toutes les apparences du hasard. Dans la pratique, on se contente de nombres « pseudo-aléatoires » générés à partir d'une variable difficile à reproduire.

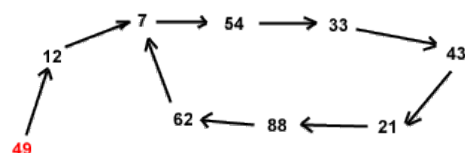
Pour obtenir une réelle dose d'imprévu, il faut faire intervenir dans le calcul une variable extérieure dont la valeur est imprévisible. L'une des méthodes les plus sûres consiste à relier un ordinateur à une source radioactive, puisqu'il est impossible de prédire le moment où un atome émettra un rayonnement radioactif.

1.3.1. Nombres pseudo-aléatoires

Une **séquence pseudo-aléatoire** (*Pseudo Random Sequence* en anglais) est une suite de nombres entiers x_0, x_1, x_2, \dots prenant ses valeurs dans l'ensemble $M = \{0, 1, 2, \dots, m-1\}$. Le terme x_n ($n > 0$) est le résultat d'un calcul (à définir) sur le ou les termes précédents. Le premier terme x_0 est appelé le **germe** (*seed* en anglais).

Algorithme général de récurrence

1. Choisir x_0 dans M
2. Poser $x_{n+1} = f(x_n)$, pour $n > 0$
où f est une application adéquate de M dans M .



Une suite construite comme indiqué dans l'algorithme ci-dessus **contient toujours un cycle** de nombres se répétant à partir d'un certain endroit. Pour éviter de devoir employer plusieurs fois le même nombre au cours d'une simulation, on cherchera à rendre la longueur de ce cycle, appelée **période**, aussi grande que possible.

Définir la fonction f est tout un art. Le lecteur intéressé pourra en savoir plus en allant sur la page : www.apprendre-en-ligne.net/random

1.3.2. Le hasard dans Python

Le langage Python dispose d'un module `random`, qui contient plusieurs fonctions qui nous seront utiles pour générer du hasard, par exemple simuler le jet d'un dé, mélanger des cartes, tirer une carte au sort, etc.

1.4. Le module `random`

Il est temps d'écrire notre premier programme en Python. Traditionnellement, il est d'usage d'écrire « Hello world ! » à l'écran.

```
print("Hello world !")
```

On peut aussi utiliser des apostrophes au lieu des guillemets :

```
print('Hello world !')
```

Nous allons maintenant utiliser le module `random`. Pour cela, il faut d'abord l'importer :

```
from random import *
```

L'étoile indique que l'on importe toutes les fonctions que contient le module. On peut aussi énumérer les fonctions qu'il nous faut, par exemple s'il ne nous en faut qu'une :

```
from random import randint
```

La documentation complète de ce module est disponible dans la documentation officielle Python, à l'adresse : <http://docs.python.org/3.3/library/random.html>. Vous y trouverez la description des fonctions qui ne figurent pas ci-dessous. En effet, nous n'utiliserons que quelques-unes d'entre elles.

1.4.1. `random()`

Parmi les fonctions du module `random`, la plus simple s'appelle... `random()`. Remarquez les parenthèses ! Cette fonction retourne un nombre réel entre 0 (compris) et 1 (non compris).

L'exécution de ce programme produira un résultat différent à chaque fois :

```
from random import random
print(random())
```

a écrit : 0.23474765730874114

mais, si vous essayez, vous obtiendrez très certainement autre chose...

Il est parfois utile, pour déboguer un programme par exemple, ou pour faire plusieurs simulations identiques, de choisir le germe de la séquence pseudo-aléatoire :

```
from random import random, seed
seed(666, 3)
print(random())
```

écrira chaque fois : 0.45611964897696833

Le paramètre `666` est le germe, `3` est la version de Python utilisée.

1.4.2. `randint()`

La fonction `randint(a,b)` retourne un nombre entier entre `a` et `b` (bornes comprises). On peut ainsi facilement simuler le jet d'un dé :

```
from random import randint
print(randint(1,6))
```

Il est fastidieux de toujours relancer le programme pour faire un lancer de dé. On peut faire 100 lancer avec une **boucle** :

```
from random import randint
for x in range(100):
    print(randint(1,6), end=" ")
```

a produit les 100 résultats suivants :

```
3 5 6 1 6 6 4 1 3 1 6 4 6 1 4 4 2 6 3 1 6 3 6 3 4 3 1 4 4 3 6 4 4 6 3 6 1 1
6 5 6 2 2 6 2 3 1 6 1 6 2 6 6 5 2 5 1 6 4 4 1 6 4 6 6 1 6 4 3 5 4 2 2 5 3 1
3 1 5 2 4 4 3 5 2 6 1 1 6 1 6 2 3 1 2 6 5 5 5
```



La **variable** `x` sert juste à compter de 0 à 99.

Remarquez bien le **décalage** vers la droite de la ligne `print(...)`. En Python, contrairement aux autres langages de programmation les plus courants, cette **indentation** n'est pas seulement esthétique, cela fait partie de la syntaxe ! Si ce *bloc* n'est pas indenté, une fenêtre avec le message « **expected an indented block** » apparaîtra.

L'instruction `end=" "` permet de séparer les valeurs par un espace plutôt que par un saut de ligne. On aurait pu séparer les valeurs par autre chose, une virgule par exemple : `end=","`. Essayez !



Exercice 1.1

1. Écrivez un programme qui simule le jet de trois dés à six faces (on additionne le résultat des trois dés).
2. Écrivez un programme qui simule 50 jets de trois dés à six faces.
3. Écrivez un programme qui simule 200 jets de deux dés à huit faces.

1.4.3. choice()

En Python, on peut créer et utiliser des **listes** (de lettres, de mots, de chiffres, ...). Il suffit de mettre les éléments entre crochets :

```
liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

La fonction `choice(liste)` permet de tirer au sort un de ces éléments :

```
from random import choice
liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print(choice(liste))
```

a produit comme résultat : b

Exercice 1.2

Trouvez un moyen de simuler un dé pipé à six faces :

- 1 apparaît dans 10 % des cas,
- 2, 3, 4 et 5 dans 15 % des cas
- 6 dans 30 % des cas.

Écrivez un programme qui simule 100 jets de ce dé.

Dans une liste, les caractères doivent être mis en guillemets ou entre apostrophes. Ce n'est pas nécessaires avec les nombres.





Exercice 1.3

Écrivez un programme que génère un mot de passe de 8 caractères aléatoires, choisis parmi les 26 lettres minuscules et les 10 chiffres.

1.4.4. shuffle()

La fonction `shuffle(liste)` mélange les éléments d'une liste. Idéal pour mélanger un paquet de cartes !

```
from random import shuffle

liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
shuffle(liste)
print(liste)
```

a produit comme résultat : ['g', 'c', 'f', 'd', 'a', 'e', 'h', 'b']



Exercice 1.4

La *Royal Rumble* est une bataille royale exposant 30 catcheurs durant un combat. La particularité du match est que les catcheurs ne sont que deux au départ et qu'un nouvel entrant arrive au bout d'un temps prédéfini, et ceci de façon régulière jusqu'à ce que tous les participants aient fait leur entrée.

Écrivez un programme qui donnera un ordre d'entrée aléatoire des catcheurs, dont voici la liste (2013) :

Wade Barrett, Daniel Bryan, Sin Cara, John Cena, Antonio Cesaro, Brodus Clay, Bo Dallas, The Godfather, Goldust, Kane, The Great Khali, Chris Jericho, Kofi Kingston, Jinder Mahal, Santino Marella, Drew McIntyre, The Miz, Rey Mysterio, Titus O'Neil, Randy Orton, David Otunga, Cody Rhodes, Ryback, Zack Ryder, Damien Sandow, Heath Slater, Sheamus, Tensai, Darren Young, Dolph Ziggler.

1.4.5. sample()

La fonction `sample(liste, k)` tire au sort k éléments d'une liste.

Contrairement à `shuffle()`, `sample()` ne modifie pas `liste`.

```
from random import sample

liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print(sample(liste, 3))
```

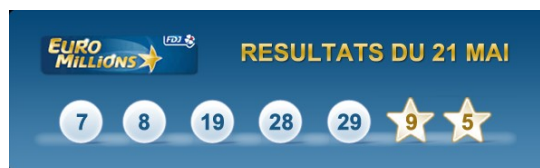
a produit comme résultat la sous-liste : ['b', 'h', 'g']



Exercice 1.5

Écrivez un programme qui simule un tirage *Euro Millions* : cinq numéros tirés au sort entre 1 et 50, suivi de deux étoiles numérotées de 1 à 12.

Par exemple :



Aide : L'instruction `numéros = list(range(a,b))` crée la liste des entiers entre a (compris) et b (**non** compris). Cela vous évitera d'écrire une liste de 50 nombres.



Exercice 1.6

Le **bingo** est un jeu de société où les nombres tirés au sort sont annoncés les uns à la suite des autres, dans un ordre aléatoire. Supposons que les boules soient numérotées de 1 à 90.

Simulez une partie de bingo. Autrement dit, il faut mélanger les boules.

1.5. Les conditions

Imaginons que l'on veuille simuler un dé pipé à six faces, comme dans l'exercice 1.2, mais avec les probabilités suivantes :

- 1 apparaît dans 12.52 % des cas
- 2 apparaît dans 13.09 % des cas
- 3 apparaît dans 21.57 % des cas
- 4 apparaît dans 19.87 % des cas
- 5 apparaît dans 11.23 % des cas
- 6 apparaît dans 21.72 % des cas

Comme la somme des probabilités doit faire 100 %, c'est-à-dire 1, on va pouvoir utiliser `random()` avec une série de conditions `si... sinon si... sinon si... sinon...` (en Python `if... elif... elif... else...`).

L'idée est de découper l'intervalle $[0, 1[$ en 6 sous-intervalles de différentes largeurs. Si `random()` est plus petit que 0.1252 (12.52 %), alors on écrira 1. Si `random()` est compris entre 0.1252 et 0.2561 (0.1252+0.1309), alors on écrira 2. Et ainsi de suite.

```
from random import random

rnd = random()
if rnd < 0.1252:
    print(1)
elif rnd < 0.2561:
    print(2)
elif rnd < 0.4718:
    print(3)
elif rnd < 0.6705:
    print(4)
elif rnd < 0.7828:
    print(5)
else:
    print(6)
```

Remarquez bien le décalage du texte.

On a utilisé une **variable** `rnd` dans laquelle on a stocké le résultat de la fonction `random()`. On n'avait pas le choix, puisque chaque fois que l'on appelle `random()`, on obtient un résultat différent.

Pour comparer des valeurs, on dispose des opérateurs suivants :

Symbole	Nom	Exemple	Résultat
<	Plus petit que	$0 < 6$	True
>	Plus grand que	$0 > 6$	False
<=	Plus petit ou égal à	$5 <= 5$	True
>=	Plus grand ou égal à	$5 >= 6$	False
==	Égal à	$10 == 10$	True
!=	Différent de	$10 != 11$	True

Tableau 1.1: opérateurs de comparaison

En Python, il est aussi possible de vérifier si un nombre est encadré par deux autres :



```
if 0.3 < rnd < 0.4:
```

Notez bien la différence entre `a=1`, qui est une **affectation** (on **donne la valeur** 1 à la variable `a`), et `a==1` qui est une **comparaison** (on **teste** si la valeur de `a` est égale à 1).

Il n'y a que deux valeurs possibles pour le résultat d'une comparaison : `True` (vrai) ou `False` (faux) : ce sont des valeurs *booléennes*. Attention aux majuscules !



Exercice 1.7

Simulez 100 jets d'une pièce truquée qui, en moyenne, montre face dans 57.83 % des cas.



Exercice 1.8

Dans le jeu de cartes *Hearthstone*, il y a quatre types de rareté des cartes : il y a environ 1 % de cartes **légendaires**, 4 % d'**épiques**, 23 % de **rares** et 72 % de **communes**.

Sachant que chaque paquet doit contenir au moins une carte rare ou supérieure, générez un paquet de 5 cartes tirées au sort en respectant ces proportions.






Un paquet sera simplement écrit sous la forme : L C R E C

Si un paquet ne contient que des communes, on indiquera que ce paquet n'est pas valide.



Exercice 1.9

Les cinq solides platoniciens peuvent être utilisés comme dés réguliers, car toutes leurs faces sont identiques.

Les cinq polyèdres réguliers convexes (solides de Platon)				
Tétraèdre	Hexaèdre ou Cube	Octaèdre	Dodécaèdre	Icosaèdre
				

On veut générer un nombre aléatoire de la manière suivante :

1. on lance le tétraèdre pour choisir lequel des quatre autres dés on va utiliser ;
2. on lance ce dé et on écrit le résultat.

Générez 100 nombres aléatoires en suivant ce protocole.

1.6. Ce que vous avez appris dans ce chapitre



- Le « vrai » hasard n'existe pas en informatique. On génère une suite de nombres qui a *l'apparence* du hasard (§ 1.3)
- C'est dans le module externe `random` que se trouvent les fonctions pseudo-aléatoires (§ 1.4)
- Vous savez écrire des choses à l'écran avec la fonction `print()`.
- Vous avez découvert trois concepts fondamentaux des langages de programmation, dont nous reparlerons plus en détails dans le chapitre 2 :
 - les boucles (§ 1.4.2),
 - les variables,
 - la condition `si... sinon` (§ 1.5).
- Vous savez comment définir une liste (§ 1.4.3). Le chapitre 5 sera essentiellement consacré à cette structure de données.
- Vous connaissez tous les opérateurs de comparaison (tableau 1.1). Attention, en particulier, à ne pas confondre `=` et `==`. C'est une erreur fréquente.



Chapitre 2

Devine mon nombre !

2.1. Thèmes abordés dans ce chapitre

- commentaires
- modules externes, import
- variables
- boucle `while`
- condition : `if... elif... else`
- la fonction de conversion `int`
- `input()`
- exceptions

2.2. Règles du jeu

Ce jeu est très simple. L'ordinateur tire un nombre au hasard entre 1 et 30 et vous avez cinq essais pour le trouver. Après chaque tentative, l'ordinateur vous dira si le nombre que vous avez proposé est trop grand, trop petit, ou si vous avez trouvé le bon nombre.

Exemple de partie

```
J'ai choisi un nombre entre 1 et 30
A vous de le deviner en 5 tentatives au maximum !
Essai no 1
Votre proposition : 15
Trop petit
Essai no 2
Votre proposition : 22
Trop grand
Essai no 3
Votre proposition : 17
Trop grand
Essai no 4
Votre proposition : 16
Bravo ! Vous avez trouvé 16 en 4 essais
```

Remarque : les nombres en gras ont été entrés au clavier par le joueur.

2.3. Code du programme



devine.py

```
# Devine mon nombre
from random import randint

nbr_essais_max = 5
nbr_essais = 1
borne_sup = 30
mon_nombre = randint(1, borne_sup) # nombre choisi par l'ordinateur
ton_nombre = 0 # nombre proposé par le joueur

print("J'ai choisi un nombre entre 1 et", borne_sup)
print("A vous de le deviner en", nbr_essais_max, "tentatives au maximum !")

while ton_nombre != mon_nombre and nbr_essais <= nbr_essais_max:
    print("Essai no ", nbr_essais)
    ton_nombre = int(input("Votre proposition : "))
    if ton_nombre < mon_nombre:
        print("Trop petit")
    elif ton_nombre > mon_nombre:
        print("Trop grand")
    else:
        print("Bravo ! Vous avez trouvé", mon_nombre, "en", nbr_essais, "essai(s)")
        nbr_essais += 1

if nbr_essais > nbr_essais_max and ton_nombre != mon_nombre:
    print("Désolé, vous avez utilisé vos", nbr_essais_max, "essais en vain.")
    print("J'avais choisi le nombre", mon_nombre, ".")
```

2.4. Analyse du programme

Reprenons ce programme ligne par ligne pour l'expliquer en détails.

2.4.1. Commentaires

```
# Devine mon nombre
```

Ceci est un commentaire. Les commentaires n'ont pas d'influence sur le programme lui-même ; ils sont là pour aider à la lecture et à la compréhension du code.



Règle 1

Le commentaire ne doit pas être redondant avec le code. Inutile de commenter des choses évidentes ! D'une manière générale, mieux le code est écrit, moins il y aura besoin de commentaires.



Règle 2

Pour déterminer ce qu'il faut indiquer dans le commentaire, se poser la question « pourquoi ? » et non pas « comment ? ». En effet, on arrivera souvent à comprendre ce que fait une fonction sans commentaires, mais on ne verra pas toujours son utilité.

2.4.2. Variables

```
nbr_essais_max = 5
nbr_essais = 1
borne_sup = 30
mon_nombre = randint(1, borne_sup) # nombre choisi par l'ordinateur
ton_nombre = 0 # nombre proposé par le joueur
```

Nous avons ici cinq variables qu'il **faut** initialiser. Cela signifie qu'il faut leur donner une valeur de départ. Si on ne le fait pas, l'interpréteur Python va envoyer le message d'erreur du genre :

```
NameError: name 'nbr_essais_max' is not defined
```

C'est au moment où l'on initialise une variable que l'interpréteur Python la crée. On peut voir une variable comme une boîte qui va contenir une valeur : ce peut être un nombre, une chaîne de caractères, une liste, etc. Écrire `nbr_essais = 1` a pour effet de déposer dans cette boîte la valeur entière 1. On ne pourra pas mettre autre chose que des nombres entiers dans cette variable par la suite.



Dans la variable `mon_nombre` va être stockée une valeur aléatoire entière, qui changera à chaque exécution du programme. Il est à noter que si l'on avait omis la ligne

```
from random import randint
```

l'interpréteur Python aurait écrit le message d'erreur : `NameError: name 'randint' is not defined`

```
print("J'ai choisi un nombre entre 1 et", borne_sup)
print("A vous de le deviner en", nbr_essais_max, "tentatives au maximum !")
```

Ces deux lignes écrivent à l'écran le texte entre guillemets, ainsi que les valeurs contenues dans les variables `borne_sup` et `nbr_essais_max`. En l'occurrence, on verra s'écrire sur l'écran :

```
J'ai choisi un nombre entre 1 et 30
A vous de le deviner en 5 tentatives au maximum !
```

Règles pour les noms des variables

Le nom d'une variable est composé des lettres de a à z, de A à Z, et des chiffres 0 à 9, mais il ne doit pas commencer par un chiffre.

Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc. sont interdits, à l'exception du caractère `_` (souligné). Le tiret (-) est bien sûr interdit puisqu'il correspond aussi à la soustraction.

La casse est significative : `spam` et `Spam` sont des variables différentes !

Python compte 33 **mots réservés** qui ne peuvent pas non plus être utilisés comme noms de variable (ils sont utilisés par le langage lui-même) :

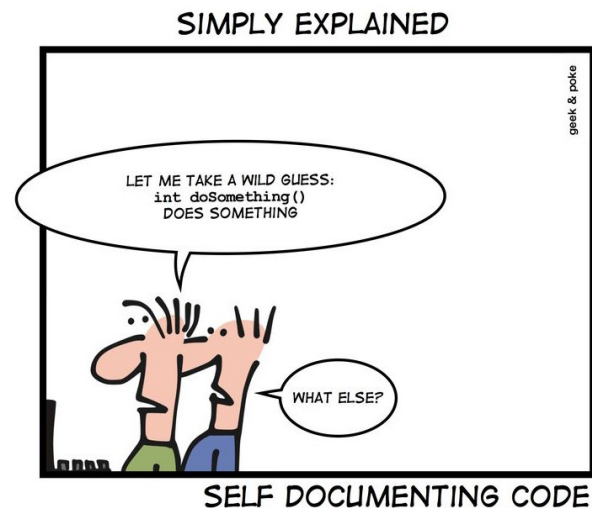


<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>	<code>continue</code>	<code>def</code>
<code>del</code>	<code>elif</code>	<code>else</code>	<code>except</code>	<code>False</code>	<code>finally</code>	<code>for</code>
<code>from</code>	<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>	<code>lambda</code>
<code>None</code>	<code>nonlocal</code>	<code>not</code>	<code>or</code>	<code>pass</code>	<code>raise</code>	<code>return</code>
<code>True</code>	<code>try</code>	<code>while</code>	<code>with</code>	<code>yield</code>		

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une convention largement respectée. N'utilisez les majuscules ou les soulignés qu'à l'intérieur du nom, pour en augmenter la lisibilité. Par exemple : `finDeMot` ou `fin_de_mot`.



Utilisez des noms de variable qui ont un sens afin d'augmenter la compréhension du programme. Cela vous évitera d'ajouter des commentaires pour expliquer l'utilité de ces variables.



Affectations

En Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
a = b = 3
```

On peut aussi effectuer des affectations *parallèles* à l'aide d'un seul opérateur :

```
a, b = 3, 2.54
```

Dans cet exemple, les variables `a` et `b` prennent **simultanément** les nouvelles valeurs 3 et 2.54. **Cela est particulièrement utile quand on veut échanger les valeurs de deux variables.** Il suffit d'écrire :

```
a, b = b, a
```

Comme les affectations sont simultanées, les nouvelles valeurs de `a` et `b` seront respectivement 2.54 et 3.

Notons enfin au passage qu'une instruction du type :

```
a + 1 = 3
```

est tout à fait illégale !



Opérations sur les variables entières

Dans notre programme, toutes les variables sont du type entier. Les opérations que l'on peut faire avec les entiers sont les suivantes :

Les priorités sont les mêmes que sur une calculatrice standard. On peut utiliser des parenthèses pour changer les priorités.

Symbole	Nom	Exemple	Résultat
+	Addition	3+4	7
-	Soustraction	8-3	5
*	Multiplication	5*2	10
//	Division entière	14//3	4
%	Reste de la division entière	14%3	2
/	Division	14/3	4.666...
**	Élévation à la puissance	3**4	81

Tableau 2.1: opérateurs sur les nombres entiers

2.4.3. Boucle while (tant que)

```
while ton_nombre != mon_nombre and nbr_essais <= nbr_essais_max:
    print("Essai no ", nbr_essais)
    ton_nombre = int(input("Votre proposition : "))
    if ton_nombre < mon_nombre:
        print("Trop petit")
    elif ton_nombre > mon_nombre:
        print("Trop grand")
    else:
        print("Bravo ! Vous avez trouvé", mon_nombre, "en", nbr_essais, "essai(s)")
    nbr_essais += 1
```

En informatique, on dit *indenté* plutôt que *décalé à droite*.

Voici une *boucle Tant que*. Tant que la valeur stockée dans `mon_nombre` sera différente de la valeur stockée dans `ton_nombre` et que le nombre d'essais effectués sera inférieur ou égal au nombre d'essais maximum, alors toute la partie du code qui est indentée vers la droite sera exécutée en boucle.

Symbole	Nom	Exemple	Résultat
==	égal	3 == 3	True
!=	différent	3 != 4	True
>	supérieur	5 > 5	False
>=	supérieur ou égal	5 >= 5	True
<	inférieur	6 < 2	False
<=	inférieur ou égal	0 <= 1	True

Tableau 2.2: opérateurs de comparaison

2.4.4. Incrémentation

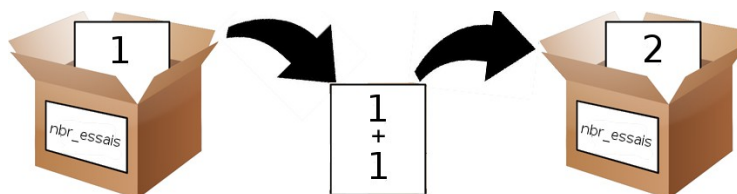
L'incrémentation est une des instructions les plus déroutantes pour un débutant :

```
nbr_essais += 1
```

On peut aussi écrire cette instruction ainsi, sans que cela soit beaucoup plus clair :

```
nbr_essais = nbr_essais + 1
```

Voici ce qui se passe. On prend la valeur de la variable `nbr_essais`, on y ajoute 1, puis on remet le résultat dans la variable `nbr_essais`. Donc, si `nbr_essais` avait la valeur 1, après l'instruction `nbr_essais += 1` il aura la valeur 2.



Beaucoup de boucles se terminent par une incrémentation, comme dans notre programme. Faute de quoi, la condition d'arrêt (ici `nbr_essais <= nbr_essais_max`) ne sera jamais satisfaite, ce qui provoquera une **boucle infinie**. C'est une des erreurs les plus courantes en programmation.



Exercice 2.1

Écrivez un programme qui demande à l'utilisateur d'écrire des mots.

Tant que l'utilisateur n'aura pas pressé sur la touche 'Enter' toute seule, l'ordinateur demandera un nouveau mot. Le programme écrira ce mot, précédé de numéro de rang.

Poliment, l'ordinateur écrira « Au revoir » avant que le programme se termine.

2.4.5. Boucles imbriquées

Il est tout à fait possible, et parfois nécessaire, **d'imbriquer** une boucle dans une autre boucle.

Par exemple, imaginons que l'on veuille écrire toutes les heures et minutes d'une journée. Cela commencera par 0 heure 0 minute, 0 heure 1 minute, 0 heure 2 minutes, ..., 0 heure 59 minutes, 1 heure 0 minute, 1 heure 1 minute, ...

On voit qu'une première boucle parcourra les heures, tandis que la deuxième parcourra les minutes, et que l'on incrémentera l'heure seulement quand 60 minutes seront passées.

Devine mon nombre !

```
h=0
while h<24:
    m=0
    while m<60:
        print(h, "heure(s) ", m, "minute(s)")
        m+=1
    h+=1
```



Exercice 2.2

Ajoutez une boucle au programme ci-dessus pour écrire les secondes.



Exercice 2.3

Partie 1

Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 53. Le résultat commencera comme ceci :

```
1 x 53 = 53
2 x 53 = 106
3 x 53 = 159
```

Aide : Il existe une fonction de formatage qui permet d'aligner joliment les nombres en colonne. Par exemple,

```
print('{:4d}'.format(a))
```

écrira l'entier a sur 4 espaces et le calera sur la droite.

La page de référence pour le formatage est :

<http://docs.python.org/py3k/library/string.html#string-formatting>

Partie 2

Modifiez la partie 1 pour obtenir un programme qui affiche toutes les tables de multiplication de 2 à 12. Chaque table comprendra 12 lignes. Alignez joliment les nombres dans les tables.

2.4.6. Conversion de types

```
ton_nombre = int(input("Votre proposition : "))
```

La fonction `input` permet à l'utilisateur d'entrer une chaîne de caractères (*string* en anglais) au clavier. En écrivant `int(input())`, on transforme le type `string` en un type entier. La variable `ton_nombre` contiendra donc un nombre entier, qui pourra être utilisé comme tel.



Exercice 2.4 : livret

Écrivez un programme Python pour tester votre livret. Le programme demandera d'abord quel est le plus grand nombre (`Nmax`) qu'il pourra utiliser. Puis il proposera 10 questions de livret avec deux nombres tirés au sort dans l'intervalle `[2, Nmax]`.

En cas de réponse erronée, le programme reposera la même question, tant que la bonne réponse n'aura pas été donnée.

Exemple d'une partie

```

nombre maximum ? 15

6 x 12 = 72
14 x 7 = 98
8 x 9 = 72
5 x 12 = 60
13 x 14 = 183
Faux ! Réessayez !
13 x 14 = 192
Faux ! Réessayez !
13 x 14 = 182
4 x 5 = 20
9 x 7 = 63
8 x 8 = 64
12 x 14 = 168
10 x 12 = 120

```

2.4.7. Conditions

```

if ton_nombre < mon_nombre:
    print("Trop petit")
elif ton_nombre > mon_nombre:
    print("Trop grand")
else:
    print("Bravo ! Vous avez trouvé",mon_nombre,"en",nbr_essais,"essais")

```

Quand le joueur propose un nombre, il y a trois possibilités : soit son nombre est trop petit, soit il est trop grand, soit c'est le bon nombre. Ces trois possibilités correspondront à trois réponses différentes de l'ordinateur.

Cela se traduira en Python par l'utilisation des instructions `if... elif... else...`. On aurait pu écrire plusieurs instructions au lieu d'une. Il aurait suffi de garder le même décalage. Par exemple :

```

if ton_nombre < mon_nombre:
    print("Trop petit")
    print("Dommage!")

```



Exercice 2.5 : livret sous pression du temps

Améliorez le programme de l'exercice 2.4 : à la fin de la partie, affichez le nombre d'erreurs (avec la bonne orthographe) et le temps utilisé pour répondre aux dix questions.

Aide : dans le module `time` se trouve la fonction `time()`, qui donne le nombre de secondes écoulées depuis le 1^{er} janvier 1970 à 00:00:00.



Exercice 2.6 : calcul mental

Améliorez le programme de l'exercice 2.5. Cette fois-ci, on ne veut pas se contenter d'exercer les multiplications, mais aussi l'addition, la soustraction et la division entière. L'opération à tester sera tirée au sort pour chaque question.



Exercice 2.7

Écrivez un programme qui simule 1000 lancers d'une pièce de monnaie. Vous afficherez seulement le nombre de piles et le nombre de faces obtenus.

Écrivez une version avec une boucle `for` (voir chapitre 1) et une autre avec une boucle `while`.



Exercice 2.8 : les paquets de cartes Hearthstone

Améliorez le programme de l'exercice 1.8.

Générez n paquets de cartes **Hearthstone** valides.

Comptez le nombre total de cartes de chaque rareté afin de vérifier que les pourcentages obtenus sont « proches » des pourcentages théoriques (plus vous générerez de paquets, plus les pourcentages seront proches de la théorie).

Exemple de sortie avec 10 paquets (50 cartes) :

```
R C R C C
C R C C C
C R C C L !
C R R C C
C C C C C non valide
C C C R C
C C R R C
R C C L C !
C C C C C non valide
C C C R C
R C C C C
E C E C C
```

```
2 légendaire(s) : 4.00 %
2 épique(s) : 4.00 %
12 rare(s) : 24.00 %
34 communes : 68.00 %
```

Vous ajouterez un point d'exclamation après les paquets contenant au moins une légendaire.



Exercice 2.9

Modifiez le code du § 2.3 :

1. Avant de commencer le jeu, le programme demandera le prénom du joueur. Quand René trouvera le bon nombre, le programme le félicitera par la phrase :
Bravo, René ! Vous avez trouvé 16 en 4 essai(s).
2. À la fin de la partie, le programme proposera une nouvelle partie au joueur, qui répondra par oui ou non. Quand le joueur arrêtera de jouer, le programme indiquera le pourcentage de réussite et le nombre moyen de tentatives pour trouver le nombre (on ne comptabilisera le nombre de coups qu'en cas de réussite).

Exemple de partie

```
Quel est votre prénom ? René
J'ai choisi un nombre entre 1 et 30
A vous de le deviner en 5 tentatives au maximum !
Essai no 1
Votre proposition : 15
Bravo, René ! Vous avez trouvé 15 en 1 essai(s)
Voulez-vous rejouer (o/n) ? o
J'ai choisi un nombre entre 1 et 30
A vous de le deviner en 5 tentatives au maximum !
Essai no 1
Votre proposition : 1
Trop petit
Essai no 2
Votre proposition : 9
Trop petit
Essai no 3
Votre proposition : 14
Trop petit
Essai no 4
Votre proposition : 19
```

```
Trop petit
Essai no 5
Votre proposition : 21
Trop petit
Désolé, vous avez utilisé vos 5 essais en vain.
J'avais choisi le nombre 25 .
Voulez-vous rejouer (o/n) ? n
Pourcentage de réussite: 50.0 %
Nombre moyen de tentatives: 1.0
```

2.5. Exceptions



devine-try.py

Si l'utilisateur entre autre chose qu'un nombre entier, le programme va générer une erreur et s'arrêter. Pour éviter cela, il faut remplacer la ligne :

```
ton_nombre = int(input("Votre proposition : "))
```

par

```
while True:
    try:
        ton_nombre = int(input("Votre proposition : "))
        break
    except ValueError:
        print("Réponse non valide. Réessayez !")
```

C'est une boucle infinie (`while True`) dont le seul moyen de sortir (`break`) est d'entrer un nombre entier. Tant que l'exception `ValueError` est levée, un message d'erreur créé par l'utilisateur est affiché à l'écran et le programme repose la question.



Exercice 2.10

Invertissons les rôles ! C'est l'ordinateur qui essaiera de deviner le nombre que vous avez en tête et vous lui indiquerez si le nombre qu'il proposera est trop petit ou trop grand.

Convention : entrez 1 si le nombre de l'ordinateur est trop grand, 2 s'il est trop petit et 0 si l'ordinateur a trouvé le bon nombre.

Prévoyez les cas où l'utilisateur écrit autre chose qu'une des trois réponses attendues.



Exercice 2.11 : Risk

Au jeu « Risk », les combats se déroulent ainsi : l'assaillant désigne par leurs noms le territoire visé et le territoire attaquant. L'assaillant prend trois dés de même couleur (rose). Il lance autant de dés qu'il engage d'armées (3 au maximum).

Exemple : Un assaillant a 4 armées sur l'Ontario. Il attaque le Québec avec maximum 3 armées à chaque coup de dés mais il peut attaquer, s'il le désire, avec seulement 1 ou 2 armées en jetant 1 ou 2 dés. Le défenseur, lui, ne peut lancer que 2 dés au maximum, même s'il a 3 armées ou plus sur son territoire.

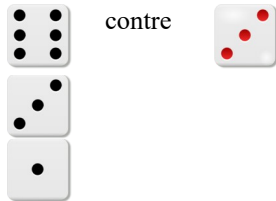


On compare séparément chacun de ses dés avec ceux de l'adversaire en commençant par le plus fort de chaque côté. Les dés les plus forts gagnent. Mais, en cas d'égalité de points, le défenseur l'emporte, même s'il lance moins de dés.

Voici plusieurs configurations de dés. L'assaillant a les dés noirs et le défenseur les dés rouges. L'assaillant peut attaquer avec 1, 2 ou 3 dés; le défenseur peut utiliser 1 ou 2 dés.

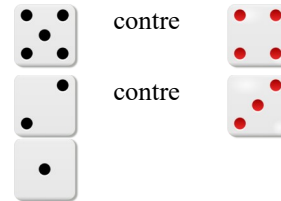
Devine mon nombre !

Exemple 1



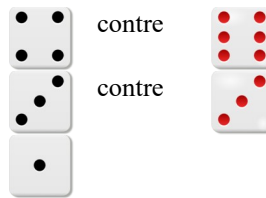
Le défenseur perd 1 armée.

Exemple 2



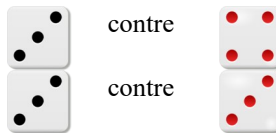
Chaque adversaire perd 1 armée.

Exemple 3



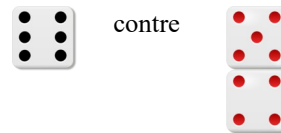
À égalité des points, le défenseur l'emporte. L'assaillant perd 2 armées.

Exemple 4



L'assaillant perd 2 armées.

Exemple 5



Le défenseur perd 1 armée.

Calculez à l'aide d'un programme et notez dans le tableau ci-après la probabilité de chacun des événements y figurant, en fonction du nombre de dés que chacun des deux joueurs choisit de lancer.

		Nombre de dés lancés par le défenseur	
		2	1
Nombre de dés lancés par l'assaillant	3	L'assaillant perd 2 armées : %	L'assaillant perd 1 armée : %
		Le défenseur perd 2 armées : %	Le défenseur perd 1 armée : %
		Chaque joueur perd 1 armée : %	
	2	L'assaillant perd 2 armées : %	L'assaillant perd 1 armée : %
		Le défenseur perd 2 armées : %	Le défenseur perd 1 armée : %
		Chaque joueur perd 1 armée : %	
1	L'assaillant perd 1 armée : %	L'assaillant perd 1 armée : %	
	Le défenseur perd 1 armée : %	Le défenseur perd 1 armée : %	



2.6. Ce que vous avez appris dans ce chapitre

- On peut écrire dans le code d'un programme des commentaires qui aideront à sa lisibilité. Il faut savoir les utiliser avec intelligence (§ 2.4.1). Un bon programmeur écrit des programmes lisibles !
- Vous savez tout sur les variables (§ 2.4.2) et comment s'en servir : noms autorisés, initialisation, affectation, incrémentation (§ 2.4.4).
- Il y a plusieurs genres d'affectation en Python : simples, multiples, simultanées (§ 2.4.2).
- En plus de la boucle `for` vue rapidement au chapitre 1, il existe une boucle `while` (§ 2.4.3). On utilise plutôt la boucle `for` quand on sait d'avance combien d'itérations on devra effectuer (par exemple, lancer 100 fois un dé), mais une boucle `while` fait aussi l'affaire. La boucle `while` s'utilise quand on ne sait pas au départ le nombre d'itérations.
- On peut imbriquer des boucles.
- Il y a différents types de données : les chaînes de caractères, les entiers, les réels, les booléens.
- La fonction `input()` retourne toujours une chaîne de caractères (§ 2.4.6). On peut convertir cette chaîne de caractères en un entier (avec `int`) ou en un réel (avec `float`).
- Le tableau 2.1 résume toutes les opérations possibles avec les entiers et le tableau 2.2 les opérateurs de comparaison.
- Les exceptions sont utiles pour éviter des erreurs qui pourraient provoquer l'arrêt du programme (§ 2.5).
- Il existe une fonction de formatage qui permet d'aligner joliment les nombres en colonne (exercice 2.3).



Chapitre 3

Pierre, papier, ciseaux

3.1. Nouveaux thèmes abordés dans ce chapitre

- procédures, fonctions
- paramètres
- variables globales et locales
- effets de bord
- types mutables

3.2. Règles du jeu

Tout le monde connaît le jeu pierre-papier-ciseaux, aussi connu sous le nom de « chifoumi ». Deux joueurs se montrent simultanément leur main qui symbolisera une pierre (poing fermé), un papier (main tendue) ou des ciseaux (l'index et le majeur forment un V).



La pierre bat les ciseaux, les ciseaux battent le papier et le papier bat la pierre. Si les deux joueurs jouent le même symbole, il y a égalité.

Nous allons dans un ce chapitre écrire une version textuelle du jeu. Nous verrons au chapitre 4 comment utiliser les images ci-dessus.

Exemple de partie

```
Pierre-papier-ciseaux. Le premier à 5 a gagné !
1 : pierre, 2 : papier, 3 : ciseaux ? 3
Vous montrez ciseaux - Je montre pierre
vous 0    moi 1
1 : pierre, 2 : papier, 3 : ciseaux ? 2
Vous montrez papier - Je montre papier
vous 0    moi 1
1 : pierre, 2 : papier, 3 : ciseaux ? 2
```


Devine mon nombre !

```
Vous montrez papier - Je montre papier
vous 0    moi 1
1 : pierre, 2 : papier, 3 : ciseaux ? 2
Vous montrez papier - Je montre ciseaux
vous 0    moi 2
1 : pierre, 2 : papier, 3 : ciseaux ? 1
Vous montrez pierre - Je montre ciseaux
vous 1    moi 2
1 : pierre, 2 : papier, 3 : ciseaux ? 1
Vous montrez pierre - Je montre ciseaux
vous 2    moi 2
1 : pierre, 2 : papier, 3 : ciseaux ? 1
Vous montrez pierre - Je montre papier
vous 2    moi 3
1 : pierre, 2 : papier, 3 : ciseaux ? 1
Vous montrez pierre - Je montre papier
vous 2    moi 4
1 : pierre, 2 : papier, 3 : ciseaux ? 3
Vous montrez ciseaux - Je montre papier
vous 3    moi 4
1 : pierre, 2 : papier, 3 : ciseaux ? 2
Vous montrez papier - Je montre ciseaux
vous 3    moi 5
```

Remarque : les nombres en gras ont été entrés au clavier par le joueur.

3.3. Code du programme



ppc.py

```
# jeu pierre, papier, ciseaux
# l'ordinateur joue au hasard

from random import randint

def ecrire(nombre):
    if nombre == 1:
        print("pierre",end=" ")
    elif nombre == 2:
        print("papier",end=" ")
    else:
        print("ciseaux",end=" ")

def augmenter_scores(mon_coup,ton_coup):
    global mon_score, ton_score
    if mon_coup == 1 and ton_coup == 2:
        ton_score += 1
    elif mon_coup == 2 and ton_coup == 1:
        mon_score += 1
    elif mon_coup == 1 and ton_coup == 3:
        mon_score += 1
    elif mon_coup == 3 and ton_coup == 1:
        ton_score += 1
    elif mon_coup == 3 and ton_coup == 2:
        mon_score += 1
    elif mon_coup == 2 and ton_coup == 3:
        ton_score += 1

ton_score = 0
mon_score = 0
fin = 5
print("Pierre-papier-ciseaux. Le premier à",fin,"a gagné !")
no_manche = 0
while mon_score < fin and ton_score < fin:
    ton_coup = int(input("1 : pierre, 2 : papier, 3 : ciseaux ? "))
    while ton_coup < 1 or ton_coup > 3:
        ton_coup =int(input("1 : pierre, 2 : papier, 3 : ciseaux ? "))
    print("Vous montrez",end=" ")
    ecrire(ton_coup)
    mon_coup = randint(1,3)
```

```
print("- Je montre",end=" ")
ecrire(mon_coup)
print() # aller à la ligne
augmenter_scores(mon_coup,ton_coup)
print("vous",ton_score," moi",mon_score)
```

3.4. Analyse du programme

Reprenons ce programme ligne par ligne pour l'expliquer en détails.

```
# jeu pierre, papier, ciseaux
# l'ordinateur joue au hasard
```

Il est toujours utile de décrire ce que fait le programme au début du code.

```
from random import randint
```

Revoici le module `random` d'où nous importons la fonction `randint(a,b)`, qui renvoie un nombre entier compris entre les bornes `a` et `b`, les bornes étant comprises.

3.4.1. Procédures

Quand les programmes deviennent plus longs, ils deviennent évidemment plus difficiles à comprendre. Aussi est-il fortement conseillé de « découper » le programme en sous-programmes, aussi appelés **procédures** ou **fonctions** (nous verrons la différence entre les deux un peu plus loin). Idéalement, chaque sous-programme devrait être visible entièrement sur l'écran pour faciliter sa lecture, et donc sa compréhension.

Les procédures sont souvent appelées plusieurs fois dans un programme, et elles ont la plupart du temps des **paramètres**. Le résultat de la procédure dépendra du ou des paramètres.

Voici une procédure permettant d'écrire à l'écran soit « pierre », soit « papier », soit « ciseaux ». Cela dépendra du paramètre `nombre`. Ainsi `ecrire(2)` affichera « papier ».

```
def ecrire(nombre):
    if nombre == 1:
        print("pierre",end=" ")
    elif nombre == 2:
        print("papier",end=" ")
    else:
        print("ciseaux",end=" ")
```

On remarquera un deuxième paramètre dans les procédures `print : end=" "`. Cela indique à Python qu'il faut continuer d'écrire sur la même ligne, après avoir inséré un espace.

La deuxième procédure du programme est chargée de mettre à jour les scores des joueurs en fonctions de leur coup.

```
def augmenter_scores(mon_coup,ton_coup):
    global mon_score, ton_score
    if mon_coup == 1 and ton_coup == 2:
        ton_score += 1
    elif mon_coup == 2 and ton_coup == 1:
        mon_score += 1
    elif mon_coup == 1 and ton_coup == 3:
        mon_score += 1
    elif mon_coup == 3 and ton_coup == 1:
        ton_score += 1
    elif mon_coup == 3 and ton_coup == 2:
        mon_score += 1
    elif mon_coup == 2 and ton_coup == 3:
        ton_score += 1
```

Les variables `mon_coup` et `ton_coup` sont **globales**. Cela signifie qu'elles ont été déclarées dans le programme principal, donc hors de la procédure. Dans un tel cas, la ligne

```
global mon_score, ton_score
```

est indispensable, sous peine du message d'erreur `UnboundLocalError: local variable 'mon_score' referenced before assignment`.

Les variables `mon_coup` et `ton_coup` ont été passées en arguments. Il n'y a donc pas lieu de les déclarer comme globales.

3.4.2. Effets de bord

En informatique, une procédure est dite à **effet de bord** si elle modifie autre chose que sa valeur de retour. Par exemple, une procédure peut modifier une variable globale, ou modifier un ou plusieurs de ses arguments. Les effets de bord rendent souvent le comportement des programmes difficile à comprendre et sont à **éviter** dans la mesure du possible.

La procédure `augmenter_scores(mon_coup, ton_coup)` est à effet de bord puisqu'elle modifie les variables globales `mon_score` et `ton_score`.

Comment éviter les effets de bord ?

On pourrait, plutôt que de les déclarer comme variables globales, les passer comme arguments de la fonction et les modifier. Mais, en Python, les arguments ne peuvent être modifiés que s'ils sont d'un type **mutable**.



Pour le moment, nous n'avons vu que les deux premiers types du tableau 3.1.

Types	Mutables
Chaîne de caractères	non
Valeur numérique	non
Liste	oui
Tuple	non
Dictionnaire	oui

Tableau 3.1 : Types mutables et non mutables

Comme `mon_score` et `ton_score` sont des entiers, ils ne sont pas mutables et ne pourront pas être modifiés. Cela ne produira pas de message d'erreur, mais les deux scores resteront à 0.

Le paragraphe suivant montre une façon d'éliminer les effets de bord en utilisant des variables locales.

3.4.3. Fonction

La différence entre une fonction et une procédure, c'est qu'une fonction renvoie une (parfois plusieurs) valeurs. La valeur retournée est toujours placée après l'instruction `return`. Une fois cette instruction exécutée, le programme quitte immédiatement la fonction.

Transformons la procédure `augmenter_scores` en une fonction :



ppc2.py

```
def augmenter_scores(mon_coup, ton_coup, mon_score, ton_score):
    mon_nouveau_score = mon_score
    ton_nouveau_score = ton_score
    if mon_coup == 1 and ton_coup == 2:
        ton_nouveau_score += 1
    elif mon_coup == 2 and ton_coup == 1:
        mon_nouveau_score += 1
    elif mon_coup == 1 and ton_coup == 3:
        mon_nouveau_score += 1
    elif mon_coup == 3 and ton_coup == 1:
        ton_nouveau_score += 1
    elif mon_coup == 3 and ton_coup == 2:
        mon_nouveau_score += 1
    elif mon_coup == 2 and ton_coup == 3:
        ton_nouveau_score += 1
    return mon_nouveau_score, ton_nouveau_score
```

Cette fonction va renvoyer deux valeurs (`mon_nouveau_score` et `ton_nouveau_score`) qu'il faudra stocker dans des variables (dans notre exemple `mon_score` et `ton_score`). On appellera donc cette fonction par cette ligne :

```
mon_score, ton_score = augmenter_scores(mon_coup, ton_coup, mon_score, ton_score)
```



Comme les variables `mon_nouveau_score` et `ton_nouveau_score` sont déclarées dans la fonction `augmenter_scores`, elles seront **locales** à cette fonction et ne pourront pas être appelées depuis l'extérieur, contrairement aux variables **globales**. Si d'aventure il y a dans le code des variables de même nom, que ce soit dans le programme principal ou dans d'autres fonctions, il s'agira de variables **différentes** ! Pour reprendre l'image du chapitre 2, il y aura deux boîtes de même nom mais différentes, ayant probablement des contenus différents.

La durée de vie des variables locales est limitée. Elles n'existent que pendant l'exécution du sous-programme dans lequel elles se trouvent. Leur emplacement en mémoire n'est pas fixé à l'avance comme pour les variables globales mais un nouvel espace mémoire leur est alloué à chaque exécution du sous-programme. Cela a pour conséquence qu'elles ne conservent pas leur contenu d'une exécution à l'autre du sous-programme.

Exercice 3.1



Voici un exercice assez difficile, mais qui résume toutes les subtilités concernant les variables locales, les variables globales, les paramètres et les effets de bord.

Avertissement ! Les programmes ci-dessous sont mal écrits et ne sont surtout pas à prendre en exemple...

Vous remarquerez qu'ils se ressemblent beaucoup. Faites bien attention à tous les détails !

Que vont écrire les programmes suivants ?

Programme 1

```
def truc():
    n=10
    p=p+1

n, p = 5, 20 # on peut initialiser plusieurs variables sur une seule ligne
truc()
print(n,p)
```

Programme 2

```
def truc():
    global p
    n=10
    p=p+1

n, p = 5, 20
truc()
print(n,p)
```

Programme 3

```
def truc():
    global n, p
    n=10
    p=p+1

n, p = 5, 20
truc()
print(n,p)
```

Programme 4

```
def truc(p) :  
    n=10  
    p=p+1  
  
n, p = 5, 20  
truc(p)  
print(n,p)
```

Programme 5

```
def truc(p) :  
    global p  
    n=10  
    p=p+1  
  
n, p = 5, 20  
truc(p)  
print(n,p)
```

Programme 6

```
def truc(p) :  
    global n  
    n=10  
    p=p+1  
  
n, p = 5, 20  
truc(p)  
print(n,p)
```

Programme 7

```
def truc(n) :  
    global p  
    n=10  
    p=p+1  
  
n, p = 5, 20  
truc(p)  
print(n,p)
```



Exercice 3.2

Réécrivez le programme de l'exercice 2.3 (partie 2), où il était question d'écrire les tables de multiplication de 2 à 12, mais en utilisant une procédure `table(n)` qui écrira la table de multiplication de `n`.



Exercice 3.3

Écrivez une fonction `maximum(a,b,c)` qui renvoie le plus grand des trois nombres `a`, `b`, `c` fournis en arguments.



Exercice 3.4

Écrivez un programme qui convertit des degrés Celsius en degrés Fahrenheit, et vice-versa. Définissez deux fonctions qui effectuent les conversions.

Formules : si F est la température en °F et C la température en °C, alors

$$F = \frac{9}{5}C + 32 \quad \text{et} \quad C = \frac{5}{9}(F - 32)$$



Exercice 3.5

D'après un sondage effectué auprès de 2000 personnes, un humain joue *pierre* dans 41% des cas, *papier* dans 30 % des cas, et *ciseaux* dans 29 % des cas. Sachant cela, on va tenter de rendre l'ordinateur plus « malin ».

Modifiez le code du § 3.3. Il faudra remplacer la ligne

```
mon_coup = randint(1,3)
```

par

```
mon_coup = coup_ordi()
```

et programmer la fonction `coup_ordi()` qui retournera *papier* dans 41 % des cas, *ciseaux* dans 30 % des cas et *pierre* dans 29 % des cas.



Exercice 3.6

Écrivez une fonction $f(x)$ qui retourne la valeur de $f(x) = 2x^3 - 3x - 1$.

Écrivez une procédure `tabuler` avec quatre paramètres : `f`, `borneInf`, `borneSup` et `pas`. Cette procédure affichera les valeurs de la fonction `f`, pour `x` compris entre `borneInf` et `borneSup`, avec un incrément de `pas`.

Exemple : `borneInf = -2.0`, `borneSup = 2.0` et `pas = 0.5`.

-2.0	-11.0
-1.5	-3.25
-1.0	0.0
-0.5	0.25
0.0	-1.0
0.5	-2.25
1.0	-2.0
1.5	1.25
2.0	9.0



3.5. Ce que vous avez appris dans ce chapitre

- Les procédures et les fonctions permettent de découper le programme en sous-programmes. Cela permet d'augmenter la lisibilité et d'éviter d'écrire plusieurs fois la même chose.
- La différence entre une fonction et une procédure, c'est qu'une fonction renvoie une ou plusieurs valeurs (§ 3.4.3).
- Il existe des variables globales et des variables locales (§ 3.4.3).
- Les effets de bord sont à éviter, si possible (§ 3.4.2).
- Il existe des types mutables et d'autres non mutables (tableau 3.1). On ne peut pas modifier les valeurs de type non mutable.



Chapitre 4

Pierre, papier, ciseaux

(version graphique)

4.1. Thèmes abordés dans ce chapitre

- Le module tkinter : Label, Button
- Fenêtre
- Événements
- Réceptionnaire d'événements

4.2. Règles du jeu

Voir § 3.2.

Exemple de partie

Nous allons dans ce chapitre programmer une version graphique du jeu. Cela ressemblera à ceci :



4.3. Programmes pilotés par des événements

Ce paragraphe est largement inspiré du livre de Swinnen « Apprendre à programmer avec Python », chapitre 8.

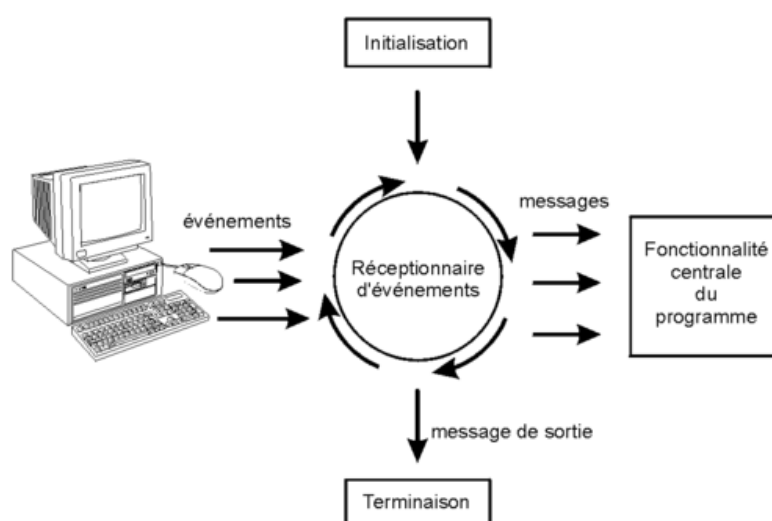
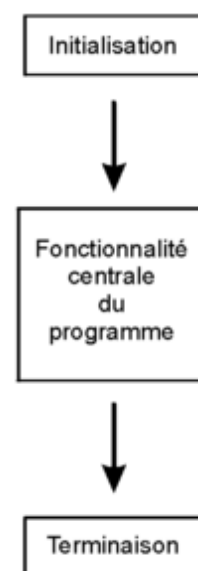
Tous les programmes d'ordinateur comportent *grosso modo* trois phases principales : une **phase d'initialisation**, laquelle contient les instructions qui préparent le travail à effectuer (appel des modules externes nécessaires, ouverture de fichiers, connexion à un serveur de bases de données ou à Internet, etc.), **une phase centrale** où l'on trouve la véritable fonctionnalité du programme (c'est-à-dire tout ce qu'il est censé faire : afficher des données à l'écran, effectuer des calculs, écrire dans un fichier, etc.), et enfin une **phase de terminaison** qui sert à stopper « proprement » les opérations (par exemple fermer les fichiers restés ouverts, fermer les fenêtres, etc.).

Dans un programme « en mode texte », ces trois phases sont simplement organisées suivant un schéma linéaire comme dans l'illustration ci-contre. En conséquence, ces programmes se caractérisent par une interactivité très limitée avec l'utilisateur. Celui-ci ne dispose pratiquement d'aucune liberté : il lui est demandé de temps à autre d'entrer des données au clavier, mais toujours dans un ordre prédéterminé correspondant à la séquence d'instructions du programme.

Dans le cas d'un programme qui utilise une interface graphique, par contre, l'organisation interne est différente. On dit que le programme est **piloté par les événements**. Après sa phase d'initialisation, un programme de ce type se met en quelque sorte « en attente », et passe la main à un autre logiciel, lequel est plus ou moins intimement intégré au système d'exploitation de l'ordinateur et « tourne » en permanence.

Ce **réceptionnaire d'événements**, comme on l'appelle, scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et réagit immédiatement lorsqu'un événement y est détecté. Lorsqu'il détecte un événement, le réceptionnaire envoie un message spécifique au programme, lequel doit être conçu pour réagir en conséquence.

La phase d'initialisation d'un programme utilisant une interface graphique comporte un ensemble d'instructions qui mettent en place les divers composants interactifs de cette interface (fenêtres, boutons, labels, etc.). D'autres instructions définissent les messages d'événements qui devront être pris en charge : on peut en effet décider que le programme ne réagira qu'à certains événements en ignorant tous les autres.



Alors que dans un programme « en mode texte », la phase centrale est constituée d'une suite d'instructions qui décrivent à l'avance l'ordre dans lequel la machine devra exécuter ses différentes tâches, on ne trouve dans la phase centrale d'un programme avec interface graphique qu'un ensemble de fonctions indépendantes. Chacune de ces fonctions est appelée spécifiquement lorsqu'un événement particulier est détecté par le système d'exploitation : elle effectue alors le travail que l'on

attend du programme en réponse à cet événement, et rien d'autre.

Il est important de bien comprendre ici que pendant tout ce temps, le réceptionnaire continue à « tourner » et à guetter l'apparition d'autres événements éventuels. S'il se produit d'autres événements, il peut donc arriver qu'une deuxième fonction (ou une troisième, une quatrième, ...) soit activée et commence à effectuer son travail « en parallèle » avec la première qui n'a pas encore terminé le sien. Les systèmes d'exploitation et les langages modernes permettent en effet ce **parallélisme** (que l'on appelle aussi **multitâche**).

4.4. Code du programme



ppc-graphique.py

```
# jeu pierre, papier, ciseaux
# l'ordinateur joue au hasard

from random import randint
from tkinter import *

def augmenter_scores(mon_coup,ton_coup):
    global mon_score, ton_score
    if mon_coup == 1 and ton_coup == 2:
        ton_score += 1
    elif mon_coup == 2 and ton_coup == 1:
        mon_score += 1
    elif mon_coup == 1 and ton_coup == 3:
        mon_score += 1
    elif mon_coup == 3 and ton_coup == 1:
        ton_score += 1
    elif mon_coup == 3 and ton_coup == 2:
        mon_score += 1
    elif mon_coup == 2 and ton_coup == 3:
        ton_score += 1

def jouer(ton_coup):
    global mon_score, ton_score, score1, score2
    mon_coup = randint(1,3)
    if mon_coup==1:
        lab3.configure(image=pierre)
    elif mon_coup==2:
        lab3.configure(image=papier)
    else:
        lab3.configure(image=ciseaux)
    augmenter_scores(mon_coup,ton_coup)
    score1.configure(text=str(ton_score))
    score2.configure(text=str(mon_score))

def jouer_pierre():
    jouer(1)
    lab1.configure(image=pierre)

def jouer_papier():
    jouer(2)
    lab1.configure(image=papier)

def jouer_ciseaux():
    jouer(3)
    lab1.configure(image=ciseaux)

def reinit():
    global mon_score, ton_score, score1, score2, lab1, lab3
    ton_score = 0
    mon_score = 0
    score1.configure(text=str(ton_score))
    score2.configure(text=str(mon_score))
    lab1.configure(image=rien)
    lab3.configure(image=rien)

# variables globales
ton_score = 0
mon_score = 0
```

```
# fenetre graphique
fenetre = Tk()
fenetre.title("Pierre, papier, ciseaux")

# images
rien = PhotoImage(file='rien.gif')
versus = PhotoImage(file='versus.gif')
pierre = PhotoImage(file='pierre.gif')
papier = PhotoImage(file='papier.gif')
ciseaux = PhotoImage(file='ciseaux.gif')

# labels
textel = Label(fenetre, text="Humain :", font=("Helvetica", 16))
textel.grid(row=0, column=0)

texte2 = Label(fenetre, text="Machine :", font=("Helvetica", 16))
texte2.grid(row=0, column=2)

texte3 = Label(fenetre, text="Pour jouer, cliquez sur une des icones ci-dessous.")
texte3.grid(row=3, columnspan=3, pady=5)

score1 = Label(fenetre, text="0", font=("Helvetica", 16))
score1.grid(row=1, column=0)

score2 = Label(fenetre, text="0", font=("Helvetica", 16))
score2.grid(row=1, column=2)

lab1 = Label(fenetre, image=rien)
lab1.grid(row=2, column=0)

lab2 = Label(fenetre, image=versus)
lab2.grid(row=2, column=1)

lab3 = Label(fenetre, image=rien)
lab3.grid(row=2, column=2)

# boutons
bouton1 = Button(fenetre,command=jouer_pierre)
bouton1.configure(image=pierre)
bouton1.grid(row=4, column=0)

bouton2 = Button(fenetre,command=jouer_papier)
bouton2.configure(image=papier)
bouton2.grid(row=4, column=1)

bouton3 = Button(fenetre,command=jouer_ciseaux)
bouton3.configure(image=ciseaux)
bouton3.grid(row=4, column=2)

bouton4 = Button(fenetre,text='Recommencer',command=reinit)
bouton4.grid(row=5, column=0, pady=10, sticky=E)

bouton5 = Button(fenetre,text='Quitter',command=fenetre.destroy)
bouton5.grid(row=5, column=2, pady=10, sticky=W)

# demarrage :
fenetre.mainloop()
```

4.5. Analyse du programme

Reprenons ce programme pour l'expliquer en détails.

```
# jeu pierre, papier, ciseaux
# l'ordinateur joue au hasard

from random import randint
from tkinter import *
```

C'est le module externe `tkinter` qui va nous permettre de faire des graphiques.

Il faudra gérer des **événements** (*events*) provenant des périphériques (le clavier ou la souris). Par

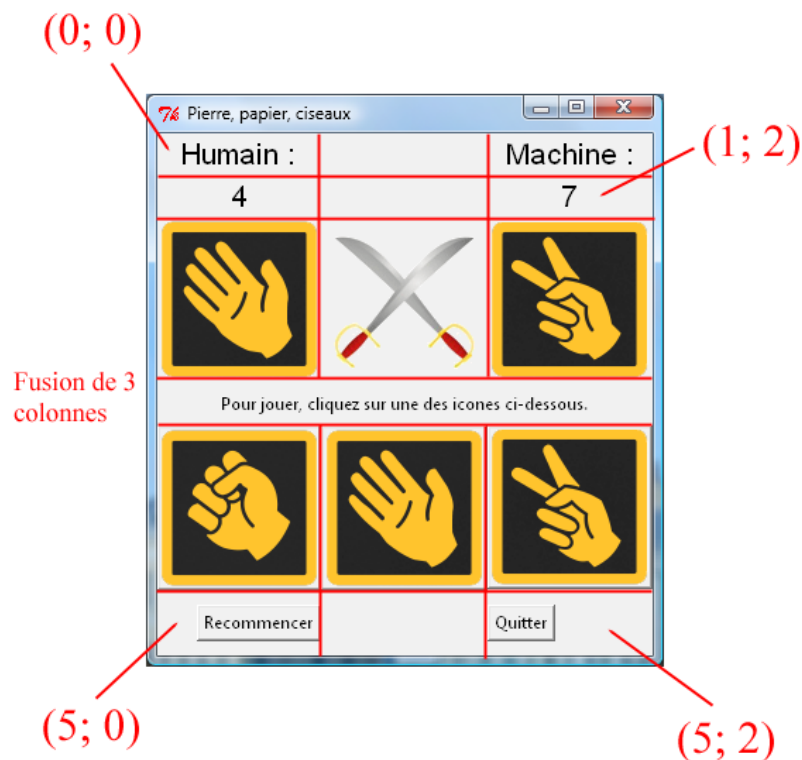
exemple, un événement pourrait être que la touche « espace » a été pressée, ou que la souris a été déplacée. Dans notre jeu, le seul événement qui nous intéressera sera le clic du bouton gauche de la souris. C'est en effet via la souris que le joueur va choisir le coup qu'il va jouer.

4.5.1. Les fenêtres

Il faudra ouvrir une fenêtre graphique dans laquelle on va placer divers éléments (des boutons, des textes, des étiquettes, etc.).

```
# fenetre graphique
fenetre = Tk()
fenetre.title("Pierre, papier, ciseaux")
```

Pour notre jeu, la fenêtre a été divisée virtuellement en plusieurs zones correspondant aux cases d'une grille. Chaque case est repérée par sa ligne et sa colonne. **Attention !** La numérotation commence à 0. Ainsi, la case en haut à gauche a pour coordonnées (0; 0). La case placée sur la deuxième ligne et la troisième colonne a pour coordonnées (1; 2).



On voit que ces zones n'ont pas la même taille. Elles s'adaptent automatiquement aux dimensions des objets qu'elles contiennent.

4.5.2. Importation des images

Nous aurons besoin de cinq images : les symboles pierre, papier, ciseaux, une case noire pour le début du jeu (« rien ») et les deux épées croisées pour faire joli. Toutes ces images, au format gif, doivent se trouver dans le même répertoire que le programme.

C'est la fonction `PhotoImage` du module `tkinter` qui importe les images.

```
# images
rien = PhotoImage(file = 'rien.gif')
versus = PhotoImage(file = 'versus.gif')
pierre = PhotoImage(file = 'pierre.gif')
papier = PhotoImage(file = 'papier.gif')
ciseaux = PhotoImage(file = 'ciseaux.gif')
```

4.5.3. Les labels

Les labels peuvent être du texte ou des images. Sur la première ligne de la grille, on a écrit les noms des joueurs (Humain et Machine). On a indiqué aussi la police utilisée (Helvetica) et la taille (16).

```
# labels
texte1 = Label(fenetre, text="Humain :", font=("Helvetica", 16))
texte1.grid(row=0, column=0)

texte2 = Label(fenetre, text="Machine :", font=("Helvetica", 16))
texte2.grid(row=0, column=2)
```

La quatrième ligne (row=3) est spéciale : elle est composée d'une seule case, qui s'étend sur la largeur de 3 colonnes (columnspan=3). Pour que le texte ne soit pas trop serré, on a mis une marge de 5 pixels en dessus et en dessous (pady=5).

```
texte3 = Label(fenetre, text="Pour jouer, cliquez sur une des icônes ci-dessous.")
texte3.grid(row=3, columnspan=3, pady=5)
```

Le contenu des labels peut varier durant l'exécution du programme. Par exemple, les scores vont évidemment changer en cours de partie. Au début, ils sont initialisés à 0 (text="0").

Les scores des deux joueurs sont placés sur la deuxième ligne (row=1) et sur la première (column=0) et troisième colonne (column=2). On a indiqué aussi la police utilisée (Helvetica) et la taille (16).

```
score1 = Label(fenetre, text="0", font=("Helvetica", 16))
score1.grid(row=1, column=0)

score2 = Label(fenetre, text="0", font=("Helvetica", 16))
score2.grid(row=1, column=2)
```

La troisième ligne (row=2) est composée de 3 labels : le coup joué par l'humain (rien au départ), les deux épées et le coup joué par la machine (rien au départ).

```
lab1 = Label(fenetre, image=rien)
lab1.grid(row=2, column=0)

lab2 = Label(fenetre, image=versus)
lab2.grid(row=2, column=1)

lab3 = Label(fenetre, image=rien)
lab3.grid(row=2, column=2)
```

4.5.4. Les boutons

Les boutons peuvent être du texte ou des images. Dans notre fenêtre, il y a 5 boutons : les symboles pierre, papier et ciseaux, ainsi que les boutons « Recommencer » et « Quitter ».

Commençons par les boutons images.

```
# boutons
bouton1 = Button(fenetre, command=jouer_pierre)
bouton1.configure(image=pierre)
bouton1.grid(row=4, column=0)

bouton2 = Button(fenetre, command=jouer_papier)
bouton2.configure(image=papier)
bouton2.grid(row=4, column=1)

bouton3 = Button(fenetre, command=jouer_ciseaux)
bouton3.configure(image=ciseaux)
bouton3.grid(row=4, column=2)
```

Pour chacun de ces boutons, la première ligne crée le bouton en indiquant dans quelle fenêtre il se trouve et la commande qui sera appelée quand on cliquera dessus. Cette commande est une procédure sans paramètres. La deuxième ligne indique quelle image sera associée au bouton. Quant à la troisième ligne, elle indique dans quelle case de la grille se trouve le bouton.

Passons maintenant aux boutons contenant du texte.

```
bouton4 = Button(fenetre, text='Recommencer', command=reinit)
bouton4.grid(row=5, column=0, pady=10, sticky=E)

bouton5 = Button(fenetre, text='Quitter', command=fenetre.destroy)
bouton5.grid(row=5, column=2, pady=10, sticky=W)
```

La création d'un bouton est presque identique à celle d'un bouton image, sauf que l'on précise le texte qui figurera dans le bouton. La deuxième ligne donne l'emplacement du bouton. On a précisé ici que le bouton « Recommencer » est calé à droite (`sticky=E`, avec E comme East) et le bouton « Quitter » à gauche (`sticky=W`, avec W comme West). Il existe aussi les lettres N pour North, donc calé en haut et S pour South, donc calé en bas.

4.5.5. Les procédures

On va retrouver des procédures écrites pour la version non graphique, parfois légèrement modifiées, ainsi que les procédures appelées par les boutons.

```
def augmenter_scores(mon_coup, ton_coup):
    global mon_score, ton_score
    if mon_coup == 1 and ton_coup == 2:
        ton_score += 1
    elif mon_coup == 2 and ton_coup == 1:
        mon_score += 1
    elif mon_coup == 1 and ton_coup == 3:
        mon_score += 1
    elif mon_coup == 3 and ton_coup == 1:
        ton_score += 1
    elif mon_coup == 3 and ton_coup == 2:
        mon_score += 1
    elif mon_coup == 2 and ton_coup == 3:
        ton_score += 1
```

Cette procédure est la même que dans la version non graphique.

```
def jouer(ton_coup):
    global mon_score, ton_score, score1, score2
    mon_coup = randint(1,3)
    if mon_coup==1:
        lab3.configure(image=pierre)
    elif mon_coup==2:
        lab3.configure(image=papier)
    else:
        lab3.configure(image=ciseaux)
    augmenter_scores(mon_coup, ton_coup)
    score1.configure(text=str(ton_score))
    score2.configure(text=str(mon_score))
```

Cette procédure `jouer` a été modifiée pour s'adapter au graphisme. Quand le joueur presse sur un des trois boutons, l'ordinateur tire au hasard un coup, puis modifie l'image du label correspondant au coup choisi (`lab3.configure(image=pierre)`). Les nouveaux scores sont ensuite calculés, puis les labels `score1` et `score2` sont mis à jour, grâce à la procédure `configure`.

```
def jouer_pierre():
    jouer(1)
    lab1.configure(image=pierre)

def jouer_papier():
    jouer(2)
    lab1.configure(image=papier)
```



```
def jouer_ciseaux():  
    jouer(3)  
    lab1.configure(image=ciseaux)
```

Ces trois procédures sont appelées par les boutons. Elles mettent à jour le label indiquant le coup choisi par le joueur.

```
def reinit():  
    global mon_score, ton_score, score1, score2, lab1, lab3  
    ton_score = 0  
    mon_score = 0  
    score1.configure(text=str(ton_score))  
    score2.configure(text=str(mon_score))  
    lab1.configure(image=rien)  
    lab3.configure(image=rien)
```

Cette dernière procédure est appelée par le bouton « Recommencer ». Elle réinitialise les variables globales, ainsi que les labels scores et les labels montrant les coups joués.

4.5.6. Gestion des événements

Le récepteur d'événements est lancé par l'instruction :

```
fenetre.mainloop()
```

Dans notre programme de jeu, la phase de terminaison consistera simplement à fermer la fenêtre. C'est le bouton « Quitter » qui appellera la fonction `destroy` :

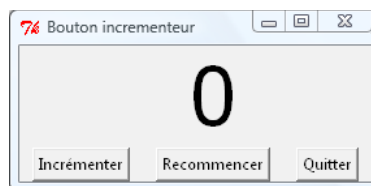
```
bouton5 = Button(fenetre, text='Quitter', command=fenetre.destroy)
```

Il est à noter que ce bouton n'était pas indispensable. En effet, cliquer le bouton rouge habituel en haut à droite de la fenêtre aurait eu le même effet.

Exercice 4.1



Reproduisez la fenêtre ci-dessous :

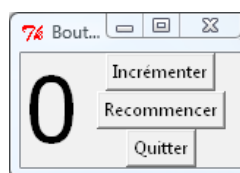


Le bouton « Incrémenter » augmentera de 1 le compteur. Le bouton « Recommencer » remettra le compteur à 0 et le bouton « Quitter » fermera la fenêtre.

Exercice 4.2



Reproduisez la fenêtre ci-dessous :



Les effets des boutons seront les mêmes que dans l'exercice 4.1.



Exercice 4.3

Reproduisez la frise ci-dessous, en utilisant une boucle (`while` ou `for`).



Exercice 4.4

Reproduisez une frise comme ci-dessous, en tirant les cartes au hasard. Chaque colonne représente une manche de pierre-papier-ciseaux. Comme pour l'exercice 4.3, utilisez une boucle.



Exercice 4.5

Modifiez le code du § 4.4.

Ajoutez une deuxième fenêtre pour que l'on puisse voir l'historique des 20 derniers coups joués dans la partie.

Attention ! Pour ouvrir une nouvelle fenêtre, il faudra utiliser l'instruction :

```
historique = Toplevel()
historique.title("Historique")
```

En effet, on ne peut pas utiliser l'instruction `Tk()` deux fois dans le même programme. Cette fenêtre sera une « fille » de la fenêtre principale, et elle disparaîtra en même temps qu'elle.

Cette deuxième fenêtre ressemblera à celle de l'exercice 4.4, mais en écrivant quelle est la ligne du joueur et celle de l'ordinateur. Quand 20 coups auront été joués, recommencez à dessiner les symboles depuis la première colonne, sans rien effacer.



Exercice 4.6

Modifiez le code du § 4.4.

Ajoutez un quatrième symbole : le puits. La pierre et les ciseaux tombent dans le puits (donc le puits gagne contre eux). Par contre, le papier recouvre le puits (le puits perd donc contre le papier).

Attention ! Les probabilités de gain changent, et la stratégie aussi !



Exercice 4.7

Let's Make A Deal ! est un show télévisé américain qui a commencé le 30 décembre 1963. À la fin du jeu, l'animateur, Monty Hall, vous offrait la possibilité de gagner ce qui se trouvait derrière une porte. Il y avait trois portes : derrière l'une d'entre elles se trouvait un prix magnifique (par exemple un voyage) et derrière les deux autres un prix moins intéressant (par exemple une chèvre ou une barre de chocolat). Vous choisissiez alors une porte.

Pour ménager le suspense, Monty Hall, avant de révéler ce qu'il y avait derrière la porte que vous aviez choisie, ouvrait une des deux autres portes (derrière laquelle ne se trouvait **jamais** le plus beau cadeau).

Il vous posait enfin la dernière question : « Conservez-vous votre premier choix, ou bien choisissez-vous l'autre porte encore fermée ? »

- Programmez ce jeu avec une interface graphique. Vous pourrez vous inspirer de cette page web : www.apprendre-en-ligne.net/random/monty/
- Quelle est la meilleure stratégie : changer de porte ou garder la porte choisie en premier ?



4.6. Ce que vous avez appris dans ce chapitre

- C'est dans le module `tkinter` que l'on trouve les procédures permettant de créer une interface graphique.
- Quand il y a une interface graphique, le programme est piloté par des événements (§ 4.3). C'est très différent des programmes « en mode texte » que l'on a vu dans les trois premiers chapitres.
- Dans le paragraphe 4.5 et les exercices, vous avez vu comment :
 - ouvrir et utiliser des fenêtres graphiques,
 - importer des images au format gif,
 - utiliser les labels,
 - définir des boutons,
 - gérer des événements.



Chapitre 5

Le jeu de Juniper Green

5.1. Nouveaux thèmes abordés dans ce chapitre

- objets
- classes
- méthodes
- listes
- opération sur les listes
- tuples

5.2. Règles du jeu

Le jeu a été créé par Richard **Porteous**, enseignant à l'école de Juniper Green, auquel il doit son nom. Il s'est réellement fait connaître grâce à Ian **Stewart**, qui en décrit les règles dans la revue *Pour la Science*, dans le numéro de juillet 1997.

Ce jeu comporte quatre règles :

1. Le joueur 1 choisit un nombre entre 1 et N_{max} .
2. À tour de rôle, chaque joueur doit choisir un nombre parmi les multiples ou les diviseurs du nombre choisi précédemment par son adversaire et inférieur à N_{max} .
3. Un nombre ne peut être joué qu'une seule fois.
4. Le premier nombre choisi doit être pair.

Le perdant est le joueur qui ne trouve plus de multiples ou de diviseurs communs au nombre choisi par l'adversaire.

Exemple de partie

```
Jouons avec des nombres entre 1 et 20.
Je choisis comme nombre de départ 2
Nombres valides: [1, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Que jouez-vous ? 4
Nombres valides: [1, 8, 12, 16, 20]
Je joue 8
Nombres valides: [1, 16]
Que jouez-vous ? 16
Nombres valides: [1]
Je joue 1
Nombres valides: [3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20]
Que jouez-vous ? 11
Bravo!
```

Remarque : les nombres en gras ont été entrés au clavier par le joueur.

5.3. Les objets en Python (premier contact)

En Python, tout est objet ! On rencontre souvent cette phrase dans les livres sur Python et cela en dit long sur l'importance de ce concept...

Qu'est-ce qu'un **objet** ? Dans un premier temps, nous allons dire que c'est une structure de données qui contient des fonctions permettant de la manipuler. On appelle ces fonctions des **méthodes**.

Comme l'idée de cet ouvrage est de partir d'un programme pour découvrir de nouveaux concepts, nous allons prendre comme premier exemple les listes.

Les listes forment une **classe** (la classe 'list'). Un objet est issu d'une classe. Tous les objets de cette classe se ressembleront, mais ils auront leur existence propre. Par exemple, toutes les listes auront le même aspect (voir § 5.4), mais elles n'auront pas le même contenu.

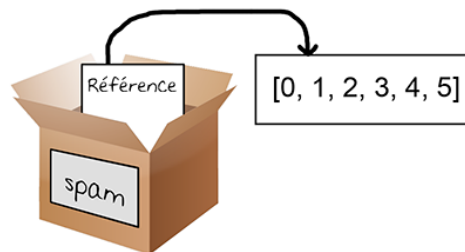
Autre exemple de classe venant de la vie courante : une Peugeot 3008 et une Ferrari Testarossa sont des objets de la classe 'voiture'.

5.4. Les listes



Comme son nom l'indique, une **liste** est une suite **ordonnée** de valeurs, qui peuvent être de types différents. Les éléments sont listés entre deux crochets et séparés par des virgules. On peut aussi voir une liste comme un tableau unidimensionnel où les éléments sont repérés par un **indice**. Ce dernier donne la position de l'élément dans la liste. Attention ! Comme dans beaucoup de langages, **l'indice du premier élément est 0** et non pas 1.

① spam = [0, 1, 2, 3, 4, 5]



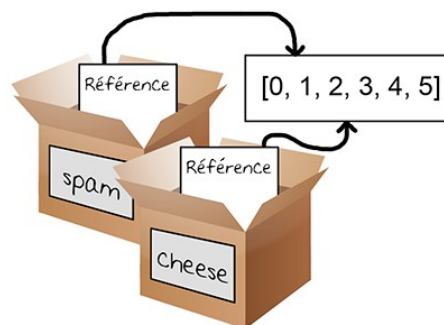
Dans le chapitre 2, nous avons représenté une variable par une boîte contenant une valeur. Dans le cas d'une liste, la boîte contient en fait une référence vers une liste, et non pas la liste elle-même. Cela a une grande importance lorsque l'on veut copier une liste. Si l'on se contente d'écrire

```
cheese = spam
```

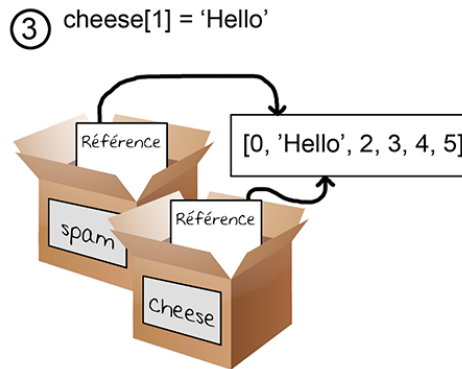


on n'aura pas copié la liste, mais la référence vers cette liste, comme indiqué sur le schéma ci-dessous :

② cheese = spam



Cela implique que si l'on modifie une des deux listes, l'autre sera modifiée de la même façon (voir ci-dessous).



5.4.1. Création des listes

Si l'on tape `type(spam)`, on obtiendra la réponse `<class 'list'>`

Pour définir une liste vide, l'instruction est logiquement :

```
spam = []
```

On peut évidemment directement créer une liste avec des éléments. Par exemple :

```
spam = [2, 3, 5, 7, 11]
```

On peut faire des listes de listes, par exemple pour représenter un tableau bidimensionnel.

Les éléments de la liste peuvent être de types différents (ici, l'ordre, un entier, une chaîne de caractères, un réel, un caractère et une liste) :

```
spam = [3, "salut", 3.1415, "x", [1, 2, 3]]
```

Si tous les éléments de la liste doivent avoir la même valeur au départ, on peut multiplier la valeur désirée par le nombre d'éléments que l'on veut. Si on veut une liste de 10 zéros, on écrira :

```
spam = [0]*10
```

ce qui donnera la liste `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`.

La fonction `range()` génère par défaut une séquence de nombres entiers de valeurs croissantes, et différant d'une unité. Si vous appelez `range()` avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, en commençant à partir de zéro (c'est-à-dire que `range(n)` génère les nombres entiers de 0 à $n-1$). Ainsi,

```
spam = list(range(10))
```

créera la liste `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.

On peut aussi utiliser `range()` avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus « sophistiquées » :

```
spam = list(range(3, 12))
```

produira la liste `[3, 4, 5, 6, 7, 8, 9, 10, 11]`. On voit que `range(a, b)` génère tous les nombres entiers de a jusqu'à $b-1$.

```
spam = list(range(3, 12, 2))
```

produira la liste `[3, 5, 7, 9, 11]`. Donc, `range(a, b, p)` génère des nombres entiers de a à $b-1$ avec un pas de p .

5.4.2. Manipulation des listes

La fonction `len` s'applique aussi aux chaînes de caractères, aux tuples, ... Ce n'est pas une méthode spécifique aux listes.

La fonction intégrée `len` permet de connaître la longueur (*length* en anglais) de la liste :

```
spam = [2,3,5,7,11]
len(spam)
```

donnera comme résultat 5. Les indices de la liste `spam` vont donc de 0 à 4.

Pour ajouter un élément en fin de liste, on a la méthode `append()` :

```
spam.append(13)
```

qui donnera la liste `[2, 3, 5, 7, 11, 13]`. Remarquez la forme caractéristique d'une méthode : `objet.méthode()`.

On peut aussi enlever un élément de la liste avec la méthode `remove()` :

```
spam.remove(5)
```

donnera comme résultat `[2, 3, 7, 11, 13]`.

Il ne faut pas confondre cette instruction avec `del(spam[5])`, qui permet d'enlever l'élément d'indice 5, donc le 6^{ème} élément de la liste. Par exemple, si `spam=[2, 3, 5, 7, 11, 13]`, alors

```
del(spam[5])
```

donnera comme résultat `[2, 3, 5, 7, 11]`.

On peut aussi facilement concaténer deux listes avec un simple `+` :

```
spam = [2,3,5,7,11]
cheese = [9,8,7,6,5]
fusion = cheese + spam
```

donnera la liste `[9, 8, 7, 6, 5, 2, 3, 5, 7, 11]`.

On peut trier la liste avec la méthode `sort()` :

```
fusion.sort()
```

donnera la liste `[2, 3, 5, 5, 6, 7, 7, 8, 9, 11]`.

Il arrive parfois que l'on doive inverser la liste :

```
fusion.reverse()
```

donnera la liste `[11, 9, 8, 7, 7, 6, 5, 5, 3, 2]`.

5.4.3. Accès aux éléments d'une liste

On peut accéder à un élément d'une liste en précisant son indice, mais on peut aussi accéder à plusieurs éléments en donnant un intervalle ; le résultat sera alors une liste. Reprenons notre liste `spam` et observons quelques exemples :

```
spam = [2,3,5,7,11,13]
print(spam[2])
```

donnera comme résultat 5. C'est un **nombre entier**.

```
print(spam[1:3])
```

donnera comme résultat `[3, 5]`. C'est une **liste** composée des éléments `spam[1]` et `spam[2]`.


```
print(spam[2:3])
```

donnera comme résultat [5]. C'est une **liste**, comme l'indiquent les crochets, contenant un seul élément.

```
print(spam[2:])
```

donnera comme résultat [5, 7, 11, 13].

```
print(spam[:2])
```

donnera comme résultat [2, 3].

```
print(spam[-2])
```

donnera comme résultat 11. C'est le deuxième élément depuis la fin de la liste.

```
print(spam[-1])
```

donnera comme résultat 13. C'est le dernier élément de la liste.

On peut savoir si un élément appartient à une liste avec l'instruction `in`.

```
5 in spam
```

donnera comme résultat `True`. On peut aussi connaître l'indice de cet élément :

```
spam.index(5)
```

donnera comme résultat 2. L'élément 5 est bien en 3^{ème} position de [2, 3, 5, 7, 11, 13].

On peut modifier un élément de la liste. Par exemple,

```
spam[0]=1
```

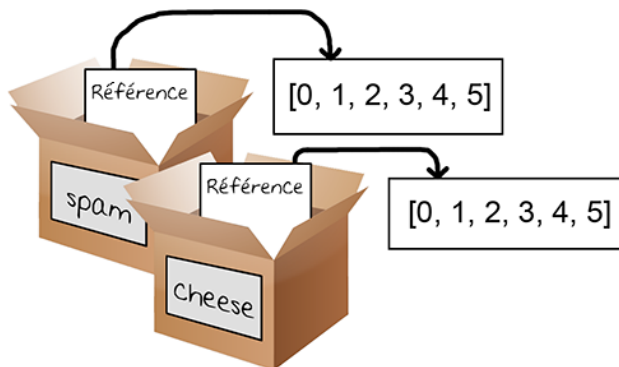
modifiera la liste `spam` ainsi :

```
[1, 3, 5, 7, 11, 13]
```

5.4.4. Copie d'une liste et insertion dans une liste

Il y a plusieurs façons de copier une liste. La plus courte est :

```
cheese = spam[:]
```



Rappelons encore une fois, car c'est important, que `cheese = spam` ne copie pas la liste, mais crée une nouvelle référence sur la liste `spam`.

Insertion d'un ou de plusieurs éléments n'importe où dans une liste

La technique du « slicing » (découpage en tranches) permet de modifier une liste à l'aide du seul opérateur []. On peut insérer un ou des éléments dans une liste. Par exemple,

```
capitales = ['Paris', 'Rome', 'Berne', 'Vienne']  
capitales[2:2] = ['Berlin']
```

donnera comme résultat :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne']
```

et

```
capitales[5:5] = ['Oslo', 'Londres']
```

donnera comme résultat :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne', 'Oslo', 'Londres']
```

Attention aux particularités suivantes :

- Si vous utilisez l'opérateur [] **à la gauche** du signe égal pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une « tranche » dans la liste cible (c'est-à-dire deux index réunis par le symbole :), et non un élément isolé dans cette liste.
- L'élément que vous fournissez **à la droite** du signe égal *doit lui-même être une liste*. Si vous n'insérez qu'un seul élément, il vous faut donc le mettre entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément `capitales[1]` n'est pas une liste (c'est la chaîne 'Rome'), alors que l'élément `capitales[1:3]` en est une.

Suppression / remplacement d'éléments

On peut aussi supprimer et/ou remplacer des éléments par « slicing ». Partons de cette liste :

```
['Paris', 'Rome', 'Berlin', 'Berne', 'Vienne', 'Oslo', 'Londres']
```

Remplaçons la tranche [2:5] par une liste vide, ce qui correspond à un effacement.

```
capitales[2:5] = []
```

produira la liste :

```
['Paris', 'Rome', 'Oslo', 'Londres']
```

Remplaçons une tranche par un seul élément. Notez encore une fois que cet élément doit lui-même être une liste.

```
capitales[1:3] = ['Madrid']
```

produira la liste :

```
['Paris', 'Madrid', 'Londres']
```

Remplaçons une tranche de deux éléments par une autre qui en compte trois.

```
capitales[1:] = ['Lisbonne', 'Prague', 'Moscou']
```

produira la liste :

```
['Paris', 'Lisbonne', 'Prague', 'Moscou']
```

5.4.5. Les boucles for

En Python, une boucle `for` parcourt simplement une liste :

```
liste = ['a','b','c']
for lettre in liste:
    print(lettre,end=' ')
```

écrira a b c.

On peut aussi faire de simples compteurs avec une boucle `for` :

```
for chiffre in range(5):
    print(chiffre,end=' ')
```

écrira 0 1 2 3 4.



Exercice 5.1

Créez une liste de 15 nombres entiers générés au hasard entre 1 et 10. Certains nombres pourront y figurer à plusieurs exemplaires. Écrivez un programme qui recopie cette liste dans une autre, en omettant les doublons. La liste finale devra être triée.



Exercice 5.2

Soient les listes suivantes :

```
liste1 = [31,28,31,30,31,30,31,31,30,31,30,31]
```

```
liste2 = ['Janvier','Février','Mars','Avril','Mai','Juin','Juillet','Août',
          'Septembre','Octobre','Novembre','Décembre']
```

Écrivez un programme qui *insère* dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant :

```
liste2 = ['Janvier',31,'Février',28,'Mars',31, ...]
```

Insérez ensuite le nombre 29 entre 28 et 'Mars'.

```
liste2 = ['Janvier',31,'Février',28,29,'Mars',31, ...]
```



Exercice 5.3

Réécrivez le code de l'exercice 2.11 (calcul des probabilités au jeu « Risk ») en utilisant des boucles `for` et des listes.

5.5. Code du programme



juniper-green.py

```
# Jeu de Juniper-Green
from random import randint

def multiples(n):
    #renvoie la liste des multiples de n <= Nmax
    mult=[]
    i=2
    while i*n <= Nmax :
        if i*n in possibles:      # on l'ajoute seulement s'il n'a pas été joué
            mult.append(i*n)
```

Le jeu de Juniper Green

```
        i += 1
        return mult

def diviseurs(n):
    #renvoie la liste des diviseurs de n
    div = []
    i=n
    while i >= 1:
        if n%i == 0 and n//i in possibles:      # on l'ajoute s'il n'a pas été joué
            div.append(n//i)
        i-=1
    return div

Nmax = 20
possibles = list(range(1,Nmax+1))      # liste des nombres pas encore utilisés
mon_nombre=2*randint(1,Nmax/2)        # l'ordinateur choisit un nombre pair
possibles.remove(mon_nombre)          # on enlève de la liste "possibles" tous les
                                      # nombres joués

# Début du jeu

print("Jouons avec des nombres entre 1 et",Nmax)
print("Je choisis comme nombre de départ",mon_nombre)
valides = diviseurs(mon_nombre) + multiples(mon_nombre)
while valides != []:
    print("Nombres valides:",valides)
    ton_nombre=int(input("Que jouez-vous ? "))
    while ton_nombre not in valides:
        ton_nombre=int(input("Incorrect. Que jouez-vous ? "))
    possibles.remove(ton_nombre)
    valides = diviseurs(ton_nombre) + multiples(ton_nombre)
    if valides == []:
        print("Bravo!")
    else:
        mon_nombre = valides[randint(0,len(valides)-1)]
        print("Nombres valides :",valides)
        print("Je joue",mon_nombre)
        possibles.remove(mon_nombre)
        valides = diviseurs(mon_nombre) + multiples(mon_nombre)
    if valides == []:
        print("Vous avez perdu!")
```

5.6. Analyse du programme

Comme les informations concernant les listes ont été données avant de montrer le code, nous ne verrons ici que le fonctionnement général du programme. Le lecteur devrait pouvoir comprendre le code assez aisément.

Une liste, nommée `possibles`, contient tous les nombres qui n'ont pas encore été joués. Au début, elle contient tous les nombres de 1 à `Nmax`. Chaque fois qu'un nombre a été joué, il est retiré de cette liste.

Le programme utilise la liste `valides` pour savoir quels nombres on peut jouer. Le jeu s'arrêtera quand cette liste sera vide. Au début du jeu, elle contient tous les diviseurs et les multiples du nombre de départ choisi par l'ordinateur (`mon_nombre`).

Ces deux listes sont des variables globales.

À tour de rôle, chaque joueur (la machine ou le joueur humain) choisit un nombre dans la liste `valides`, tant qu'elle n'est pas vide. Ce nombre est retiré de la liste `possibles`. La liste `valides` est mise à jour, et on recommence jusqu'à la victoire d'un des deux joueurs.



Exercice 5.4

Modifiez le code du § 5.5.

Avant de commencer le jeu, le programme demandera au joueur d'entrer la valeur de `Nmax`. Il demandera aussi si le joueur veut jouer en premier ; il faudra évidemment adapter le programme en conséquence.



Exercice 5.5

Modifiez le code de l'exercice 5.4.
 Dans la version de l'exercice 5.4, l'ordinateur tire simplement au hasard un nombre parmi ceux qui sont valides. Trouvez une meilleure stratégie et programmez-la !



Exercice 5.6

Une variante pour jouer seul au jeu de Juniper Green consiste à trouver la plus longue partie possible pour un N_{max} donné.

Programmez une méthode *probabiliste* : faites des milliers de parties en jouant des coups (légaux) au hasard et gardez en mémoire la partie la plus longue, que vous afficherez à la fin.

Note : la partie trouvée ne sera pas forcément la plus longue, mais plus vous ferez de parties, plus vous vous en approcherez.

Il semble que la longueur maximale soit de 41 coups.

Record actuel : 38 coups (11.12.2014, plusieurs fois égalé depuis mais jamais battu)

Je joue 5000000 parties avec des nombres entre 1 et 50
 Plus longue partie: 38 coups
 22, 1, 17, 34, 2, 26, 13, 39, 3, 33, 11, 44, 4, 24, 6, 30, 15, 45, 9, 18, 36, 12, 48, 16, 32, 8, 40, 20, 10, 50, 25, 5, 35, 7, 28, 14, 42, 21

Autres parties de 38 coups :

(5.12.2019)
 46, 23, 1, 33, 11, 22, 44, 2, 26, 13, 39, 3, 36, 12, 24, 4, 28, 14, 42, 21, 7, 35, 5, 25, 50, 10, 20, 40, 8, 32, 16, 48, 6, 30, 15, 45, 9, 27

(19.12.2019)
 34, 17, 1, 26, 13, 39, 3, 33, 11, 22, 44, 2, 40, 20, 10, 50, 25, 5, 35, 7, 21, 42, 14, 28, 4, 8, 32, 16, 48, 24, 12, 36, 18, 9, 45, 15, 30, 6



Exercice 5.7 : les vignettes Panini

Collectionner les vignettes autocollantes vendues à l'occasion de la coupe du monde de football : voici une activité (onéreuse) à laquelle s'adonnent avec joie de nombreux enfants (et leurs parents).



Imaginons que l'on commence un nouvel album. Il faudra coller 700 vignettes. Supposons que l'on achète les vignettes à l'unité. Évidemment, on ne sait pas quelle vignette on achète : elle est scellée dans une pochette. Si on ne fait pas d'échanges, combien de vignettes faudra-t-il acheter pour remplir l'album ?

Répondez à cette question à l'aide d'un programme Python.



Exercice 5.8 : Le Verger

Le **Verger** (Obstgarten) est un jeu de société coopératif créé par Anneliese Farkaschovsky. Il a été édité la première fois en 1986 par la société Haba.

Le jeu contient :

- un plan de jeu présentant 4 arbres fruitiers (prunier, pommier, poirier, cerisier) ainsi qu'un corbeau au centre.
- 40 fruits (10 pour chaque arbre du plan de jeu) en bois à disposer sur chacun des arbres.
- 4 paniers pour les joueurs (1 panier par joueur).
- un puzzle corbeau de 9 pièces que l'on reconstituera au fil du jeu.
- un gros dé.



But du jeu

Le but du jeu est de récupérer tous les fruits avant d'avoir reconstitué le puzzle corbeau.

Règles du jeu

Le plus jeune joueur commence en lançant le dé. Ce dé a quatre faces de couleur (jaune pour la poire, vert pour la pomme, bleu pour la prune et rouge pour la cerise), ainsi qu'une face « panier », et une face « corbeau ».

- S'il tombe sur une face couleur, il prend le fruit correspondant à la couleur, et le met dans son panier. S'il n'y a plus de fruit sur l'arbre, le joueur passe son tour.
- S'il tombe sur « panier », il prend deux fruits de son choix.
- S'il tombe sur « corbeau », il place une des pièces de puzzle corbeau sur le plateau de jeu.

Le jeu continue avec le joueur suivant.

Le gagnant

Les joueurs gagnent tous ensemble s'ils ont réussi à cueillir tous les fruits des arbres.

Les joueurs perdent ensemble si le puzzle corbeau de 9 pièces est reconstitué entièrement avant qu'ils n'aient pu récupérer tous les fruits.

Programme demandé

Écrivez un programme qui simulera 100'000 parties de ce jeu. Vous indiquerez comme résultat le pourcentage de victoires des joueurs, ainsi que le nombre moyen de coups dans une partie.



Exercice 5.9 : Les cartes de loto

0	1	2	3	4	5	6	7	8
2		23		43	51			81
	16	24		48		61	77	
4	19		38			64		88

Écrivez un programme produisant une carte de loto (voir l'exemple ci-dessus, avec en rouge les numéros des colonnes). Il faudra tout d'abord trouver un algorithme permettant de créer une carte en respectant ces contraintes :

- Une carte contient 9 colonnes et 3 lignes.
- Il y a sur la carte 15 numéros différents choisis parmi les nombres de 1 à 90.
- Chaque ligne contient 5 numéros (et donc 4 espaces vides).
- Il y a toujours au moins un numéro par colonne.
- Il peut y avoir 3 numéros dans une colonne, mais seulement dans la colonne 8.
- La colonne 0 contient les numéros de 1 à 9.
- La colonne 1 contient les numéros de 10 à 19.
- La colonne 2 contient les numéros de 20 à 29.
- ...
- La colonne 7 contient les numéros de 70 à 79.
- La colonne 8 contient les numéros de 80 à 90.

5.7. Tuples

Python propose un type de données appelé **tuple**, qui est une liste non modifiable.

Du point de vue syntaxique, un tuple est une collection d'éléments séparés par des virgules :

```
alpha = 'a', 'b', 'c', 'd', 'e'
print(alpha)
```

donnera comme résultat :

```
('a', 'b', 'c', 'd', 'e')
```

Bien que cela ne soit pas nécessaire, il est fortement conseillé de mettre le tuple en évidence entre parenthèses pour améliorer la lisibilité du code. Comme ceci :

```
alpha = ('a', 'b', 'c', 'd', 'e')
```

Les opérations que l'on peut effectuer sur des tuples sont les mêmes que celles que pour les listes, si ce n'est que les tuples ne sont **pas mutables**. Il est donc impossible d'ajouter ou de supprimer des éléments d'un tuple.

5.7.1. Utilité des tuples

Les tuples trouvent principalement leur utilité dans deux cas :

- ils sont utilisés quand une fonction doit renvoyer plusieurs valeurs ;
- comme ils ne sont pas mutables, ils peuvent servir de clés pour un dictionnaire (voir chapitre 6), contrairement aux listes.

Le code ci-dessous définit une fonction appelée `cercle` qui prend en paramètre le rayon `r` du

cercle et qui renvoie deux valeurs : la circonférence et l'aire du cercle. En utilisant un tuple comme type de la valeur de retour, une fonction peut retourner plusieurs valeurs.

```
import math

def cercle(r):
    #Retourne le couple (p,a) : périmètre et aire d'un cercle de rayon r
    return (math.pi*2*r,math.pi*r*r)

(p,a) = cercle(4)
print("périmètre =",p)
print("aire =",a)
```

Répetons que les parenthèses des tuples ne sont pas indispensables. Voici la version sans parenthèses :

```
import math

def cercle(r):
    #Retourne le couple (p,a) : périmètre et aire d'un cercle de rayon r
    return math.pi*2*r,math.pi*r*r

p,a = cercle(4)
print("périmètre =",p)
print("aire =",a)
```



Exercice 5.10

Réécrivez le code de l'exercice 4.4 du chapitre précédent, en utilisant un tuple pour désigner les trois coups possibles (pierre, papier ou ciseaux), à la place d'une fonction.



5.8. Ce que vous avez appris dans ce chapitre

- Vous avez pour la première fois rencontré la notion d'**objet**, qui est très importante en Python.
- Les listes sont des structures de données très utiles. Les principales instructions ont été présentées pour les créer et les manipuler.
- L'instruction `for` permet de parcourir une liste.
- Un tuple ressemble beaucoup à une liste, à part qu'il n'est pas modifiable.



Chapitre 6

Le dilemme du prisonnier

6.1. Nouveaux thèmes abordés dans ce chapitre

- dictionnaire
- fonction lambda

6.2. Les dictionnaires

Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu de disposer ces informations dans un ordre précis, ils associent chaque objet contenu à une **clé** (la plupart du temps, une chaîne de caractères).

6.2.1. Création d'un dictionnaire

Un dictionnaire est un type **modifiable**. Nous pouvons donc d'abord créer un dictionnaire vide, puis le remplir. Contrairement aux listes, il n'y a pas de fonction spéciale (telle que `append()`) à appeler pour ajouter un objet.

```
dico = {}  
dico['red'] = 'rouge'  
dico['yellow'] = 'jaune'  
dico['blue'] = 'bleu'  
dico['green'] = 'vert'  
dico['pink'] = 'rose'
```

On pourra afficher ce dictionnaire à l'écran facilement :

```
print(dico)
```

```
{'blue': 'bleu', 'pink': 'rose', 'green': 'vert', 'yellow': 'jaune', 'red': 'rouge'}
```

Le dictionnaire apparaît comme une suite d'éléments séparés par des virgules, le tout entre accolades. Chaque élément est composé d'une paire d'objets (ici deux chaînes de caractères, mais on pourrait avoir d'autres types d'objets) séparés par un double point. Le premier objet est la **clé** et le second est une **valeur**.

On remarquera que les paires n'apparaissent pas dans l'ordre dans lequel elles ont été introduites. Rappelez-vous qu'un dictionnaire n'est pas ordonné.

Le dilemme du prisonnier

Pour accéder à une valeur, il suffit de connaître sa clé :

```
print(dico['pink'])
```

affichera `rose`.

Les dictionnaires sont des objets. On peut donc leur appliquer des méthodes spécifiques. La méthode `keys()` renvoie les clés utilisées dans le dictionnaire :

```
print(dico.keys())
```

```
dict_keys(['blue', 'pink', 'green', 'yellow', 'red'])
```

Pour connaître les valeurs, la méthode à utiliser est `values()` :

```
print(dico.values())
```

```
dict_values(['bleu', 'rose', 'vert', 'jaune', 'rouge'])
```

On peut aussi extraire du dictionnaire une séquence équivalente de tuples avec la méthode `items()`. Cela nous sera utile quand nous voudrons parcourir un dictionnaire avec une boucle.

```
print(dico.items())
```

```
dict_items([('blue', 'bleu'), ('pink', 'rose'), ('green', 'vert'), ('yellow', 'jaune'), ('red', 'rouge')])
```

6.2.2. Manipuler un dictionnaire

La fonction intégrée `len()` permet de connaître le nombre d'entrées du dictionnaire :

```
len(dico)
```

donnera comme résultat `5`.

On a vu au § 6.2.1 comment ajouter des entrées. On peut aussi remplacer la valeur correspondant à une clé. Par exemple :

```
dico['blue'] = 'bleu ciel'
```

Pour résumer, si la clé existe, la valeur correspondante est remplacée ; si elle n'existe pas, une nouvelle entrée est créée.

On peut supprimer une entrée avec le mot-clé `del` :

```
del(dico['blue'])  
print(dico)
```

```
{'pink': 'rose', 'green': 'vert', 'yellow': 'jaune', 'red': 'rouge'}
```

La méthode `pop()` supprime également la clé précisée, mais elle renvoie en plus la valeur supprimée. Cela peut être utile parfois.

```
couleur = dico.pop('pink')  
print(couleur)
```

écrira `rose`.

D'une façon analogue à ce qui se passe avec les listes et les tuples, l'instruction `in` permet de savoir si un dictionnaire contient une clé déterminée :

```
'red' in dico
```

donnera comme résultat `True`.

Si l'on donne une clé qui n'est pas dans le dictionnaire, cela provoquera une erreur :

```
print(dico['cyan'])
```

provoquera l'erreur : **KeyError: 'cyan'**

Pour pallier ce problème, il existe la méthode `get()` :

```
print(dico.get('red', 'inconnu'))
```

écrira rouge, car red est dans le dictionnaire.

```
print(dico.get('cyan', 'inconnu'))
```

écrira inconnu, car cyan n'est pas dans le dictionnaire.

Le premier argument transmis à cette méthode est la clé de recherche, le second est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

6.2.3. Parcours d'un dictionnaire

Vous pouvez utiliser une boucle `for` pour traiter successivement tous les éléments contenus dans un dictionnaire, mais attention :

- au cours de l'itération, ce sont les **clés** du dictionnaire qui seront successivement affectées à la variable de travail, et non les valeurs ;
- l'ordre dans lequel les éléments seront parcourus est **imprévisible** (puisque'un dictionnaire n'est pas ordonné).

```
for i in dico:
    print(i, dico[i])
```

écrira :

```
blue bleu
pink rose
green vert
yellow jaune
red rouge
```

La manière de procéder suivante est plus élégante, pour un résultat identique :

```
for cle, valeur in dico.items():
    print(cle, valeur)
```

La méthode `items()` renvoie une suite de tuples (clé, valeur).

On peut aussi ne parcourir que les clés :

```
for cle in dico.keys():
    print(cle)
```

ou que les valeurs :

```
for valeur in dico.values():
    print(valeur)
```



Trier un dictionnaire

Il n'est pas possible de trier un dictionnaire, puisque, encore une fois, c'est une structure non ordonnée. Par contre, il est possible d'écrire un dictionnaire selon un certain ordre, mais il faut d'abord créer une liste de tuples (une liste, elle, peut être triée sans problèmes).

Voici la ligne de code pour créer une liste de tuples triée selon les clés :

Le dilemme du prisonnier

On verra au § 6.3 ce que sont ces lambda...

```
dico_trie = sorted(dico.items(), key=lambda x : x[0])
print(dico_trie)
```

Le résultat sera :

```
[('blue', 'bleu'), ('green', 'vert'), ('pink', 'rose'), ('red', 'rouge'), ('yellow', 'jaune')]
```

Et voici la ligne de code pour créer une liste de tuples triée selon les valeurs :

```
dico_trie = sorted(dico.items(), key=lambda x : x[1])
print(dico_trie)
```

Le résultat sera :

```
[('blue', 'bleu'), ('yellow', 'jaune'), ('pink', 'rose'), ('red', 'rouge'), ('green', 'vert')]
```



6.2.4. Copie d'un dictionnaire

La méthode `copy()` permet d'effectuer une vraie copie d'un dictionnaire. En effet, comme pour les listes (voir § 5.4.4), l'instruction `mon_dico = dico` **ne crée pas un nouveau dictionnaire**, mais une nouvelle **référence** vers le dictionnaire `dico`.

```
mon_dico = dico.copy()
```



Exercice 6.1

Construisez le dictionnaire des douze mois de l'année avec comme valeurs le nombre de jours respectif.

Utilisez la méthode `pprint()` du module `pprint` pour afficher ce dictionnaire à l'écran.



Exercice 6.2

Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques.



Exercice 6.3

En cryptographie, un chiffre de substitution simple consiste à remplacer chaque lettre d'un texte par une autre lettre (toujours la même). On utilise pour cela une table chiffante. Par exemple :

Clair	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Chiffré	W	B	H	A	Y	P	O	D	Q	Z	X	N	T	S	F	L	R	U	V	M	C	E	K	J	G	I

Ainsi, le mot «RENARD» sera chiffré « UYSWUA ».

1. Écrivez un programme qui permet de (dé)chiffrer un texte avec la table ci-dessus.
2. Utilisez votre programme pour déchiffrer le message suivant :

HYHQYVMCSDHQPPUYAYVCBVMQMCMQFS

P. S. On peut parcourir les lettres d'une chaîne de caractères avec l'instruction :

```
for lettre in chaine:
```



Exercice 6.4

Vous avez à votre disposition sur le site web compagnon un texte non accentué et non ponctué. Écrivez un programme qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte, et qui en fait un histogramme comme l'exemple fictif ci-dessous :

```

: |||||
a : |||||
d : ||||
e : |||||
i : |||||
l : |||||
n : |||||
o : |||||
p : ||
r : |||||
s : |||||
t : |||||
u : |||||

```

6.3. Les fonctions lambda

Python permet une syntaxe intéressante pour définir des mini-fonctions d'une ligne. Empruntées au langage Lisp, ces fonctions dites *lambda* peuvent être employées partout où une fonction est nécessaire. Attention ! Les fonctions lambda sont limitées à **une seule** instruction.

La syntaxe d'une fonction lambda est la suivante :

```
ma_fonction = lambda arg1,arg2,... :instruction
```

où `arg1,arg2,...` est la liste des arguments de `ma_fonction`. Par exemple :

```
carre = lambda x: x*x
print(carre(9))
```

donnera comme résultat 81.

```
mult = lambda x,y=1: x*y
print(mult(6))
```

donnera comme résultat 6.

```
print(mult(6,3))
```

donnera comme résultat 18.

Il est évidemment possible que l'instruction soit une fonction définie ailleurs. Par exemple :

```
def signe(x):
    if x>0:
        return 1
    elif x<0:
        return -1
    else:
        return 0







f = lambda x : 2*signe(x)
print(f(-5))
```

donnera comme résultat -2.

6.4. Le dilemme du prisonnier - règles du jeu

Dans le jeu du « dilemme du prisonnier », deux détenus sont emprisonnés dans des cellules séparées. La police fait à chacun des deux le même marché :

« Tu as le choix entre dénoncer ton complice ou non. Si tu le dénonces et qu'il te dénonce aussi, vous aurez une remise de peine d'un an tous les deux. Si tu le dénonces et que ton complice te couvre, tu auras une remise de peine de 5 ans (et tu seras libéré), mais ton complice tirera le maximum. Mais si vous vous couvrez mutuellement, vous aurez tous les deux une remise de peine de 3 ans. »

		coopère 		trahit	
coopère 	coopère	 -3 -3	 0 -5		
	trahit	 -5 0	 -1 -1		

Dans cette situation, il est clair que si les détenus s'entendent, ils s'en tireront globalement mieux que si l'un trahit l'autre. Mais l'un peut être tenté de s'en tirer encore mieux en trahissant son complice. Craignant cela, l'autre risque aussi de trahir son complice pour ne pas être le dindon de la farce. Le dilemme est donc : « faut-il coopérer avec son complice (et donc le couvrir) ou non ? »

Le dilemme du prisonnier devient plus intéressant et plus réaliste lorsque la durée de l'interaction n'est pas connue. On peut alors envisager de se souvenir du comportement d'un joueur à son égard et développer une stratégie en rapport. Par exemple, si je sais que mon adversaire ne coopère jamais, mon intérêt de ne pas coopérer non plus, sous peine d'être systématiquement grugé. Par contre, si je sais que mon adversaire coopérera toujours quoi qu'il arrive, j'aurai intérêt à être vicieux et ne jamais coopérer pour maximiser mon gain.

Exemple de partie

```
Coups et score du joueur Donnant donnant
['C', 'T', 'T', 'T', 'T', 'C', 'T', 'C', 'T', 'T']
19
```

```
Coups et score du joueur Aléatoire
['T', 'T', 'T', 'T', 'C', 'T', 'C', 'T', 'T', 'C']
19
```

6.5. Code du premier programme (les duels)

Voyons tout d'abord un premier programme où deux joueurs s'affrontent en duel.



dilemme.py

```
# dilemme du prisonnier itéré, version duel
from random import choice

choix = ['T','C'] # T : trahit, C : coopère

def gain(lui,moi):
    if lui=='C' and moi=='C':
        return 3
    elif lui=='C' and moi=='T':
        return 5
    elif lui=='T' and moi=='C':
        return 0
    elif lui=='T' and moi=='T':
        return 1
```

```

# Toujours seul
# ne coopère jamais
def toujours_seul(liste_lui,liste_moi):
    return 'T'

# Bonne poire
# coopère toujours
def bonne_poire(liste_lui,liste_moi):
    return 'C'

# Aléatoire
# joue avec une probabilité égale 'T' ou 'C'
def aleatoire(liste_lui,liste_moi):
    global choix
    return choice(choix)

# Donnant donnant
# coopère seulement si l'autre joueur a coopéré au coup précédent.
def donnant_donnant(liste_lui,liste_moi):
    if len(liste_lui)>0:
        return liste_lui[-1]
    else: # premier coup
        return 'C'

# Majorité
# coopère seulement si l'autre joueur a coopéré en majorité.
def majorite(liste_lui,liste_moi):
    if len(liste_lui)>0:
        if liste_lui.count('C') > len(liste_lui)//2:
            return 'C'
        else:
            return 'T'
    else: # premier coup
        return 'C'

# Une partie entre deux joueurs différents

liste = {}
score = {}

liste['Aléatoire'] = []
liste['Donnant donnant'] = []

for joueur in liste.keys():
    score[joueur] = 0

nb_coups = 0
nb_total_coups = 10 # à modifier

while nb_coups < nb_total_coups :
    coup_joueur1 = aleatoire(liste['Donnant donnant'],liste['Aléatoire'])
    coup_joueur2 = donnant_donnant(liste['Aléatoire'],liste['Donnant donnant'])
    liste['Aléatoire'].append(coup_joueur1)
    liste['Donnant donnant'].append(coup_joueur2)
    score['Aléatoire'] += gain(coup_joueur2,coup_joueur1)
    score['Donnant donnant'] += gain(coup_joueur1,coup_joueur2)
    nb_coups += 1

for joueur in liste.keys():
    print("Coups et score du joueur",joueur)
    print(liste[joueur])
    print(score[joueur])
    print()

```

Analyse du programme

La première partie du programme ne présente rien de nouveau : on définit les gains et les stratégies des joueurs (les noms des stratégies sont données par les noms des joueurs).

Le dilemme du prisonnier

La grande nouveauté apparaît avec les dictionnaires :

```
liste = {}
score = {}
```

On crée d'abord deux dictionnaires : `liste` donnant la liste des coups joués par les joueurs, et `score` donnant leur score respectif.

On initialise ensuite les deux listes de coups :

```
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
```

puis on met les scores à 0 :

```
for joueur in liste.keys():
    score[joueur] = 0
```

Enfin, le duel commence et durera `nb_total_coups` :

```
nb_coups = 0
nb_total_coups = 10 # à modifier
while nb_coups < nb_total_coups :
```

Chaque joueur joue selon sa stratégie en tenant compte des coups précédents (le premier paramètre est la liste des coups de l'adversaire, le second la liste des coups du joueur) :

```
coup_joueur1 = aleatoire(liste['Donnant donnant'],liste['Aléatoire'])
coup_joueur2 = donnant_donnant(liste['Aléatoire'],liste['Donnant donnant'])
```

On ajoute ce coup à la liste des coups :

```
liste['Aléatoire'].append(coup_joueur1)
liste['Donnant donnant'].append(coup_joueur2)
```

puis on met à jour les scores et on recommence pour le coup suivant :

```
score['Aléatoire'] += gain(coup_joueur2,coup_joueur1)
score['Donnant donnant'] += gain(coup_joueur1,coup_joueur2)
nb_coups += 1
```

On affiche finalement les coups joués lors du duel et les scores des deux joueurs :

```
for joueur in liste.keys():
    print("Coups et score du joueur",joueur)
    print(liste[joueur])
    print(score[joueur])
    print()
```

6.6. Code partiel du second programme (le tournoi)

Dans ce second programme, on pourra faire s'affronter plus de deux joueurs. Chaque joueur affrontera en duel chacun des autres joueur, y compris lui-même !



tournoi.py

```
# dilemme du prisonnier itéré, version tournoi
```

Le début du code est strictement identique à la version duel. Voir § 6.5.

```
# Le tournoi
liste = {}
strategie = {}
score = {}
```

```

duel = {}

# ajouter des joueurs ci-dessous, selon les modèles des joueurs existants
# commencer ici
liste['Toujours seul'] = []
liste['Bonne poire'] = []
liste['Majorité'] = []
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
# ...

strategie['Toujours seul'] = lambda lui, moi : toujours_seul(lui,moi)
strategie['Bonne poire'] = lambda lui, moi : bonne_poire(lui,moi)
strategie['Majorité'] = lambda lui, moi : majorite(lui,moi)
strategie['Aléatoire'] = lambda lui, moi : aleatoire(lui,moi)
strategie['Donnant donnant'] = lambda lui, moi : donnant_donnant(lui,moi)
# ...
# terminer là

nb_total_coups = 10 # à modifier

for joueur in liste.keys():
    score[joueur] = 0

for i in liste.keys(): # i et j sont les joueurs
    for j in liste.keys() :
        liste[i] = [] # on recommence une partie
        liste[j] = []
        if i>=j:
            nb_coups = 0
            score_joueur1 = 0
            score_joueur2 = 0
            while nb_coups < nb_total_coups :
                coup_joueur1 = strategie[i](liste[j],liste[i])
                coup_joueur2 = strategie[j](liste[i],liste[j])
                liste[i].append(coup_joueur1)
                if i!=j:
                    liste[j].append(coup_joueur2)
                score_joueur1 += gain(coup_joueur2,coup_joueur1)
                score_joueur2 += gain(coup_joueur1,coup_joueur2)
                nb_coups += 1
            duel[(i,j)] = score_joueur1
            if i!=j:
                duel[(j,i)] = score_joueur2
            score[i] += score_joueur1
            if i!=j:
                score[j] += score_joueur2

# affichage des résultats

def trie_par_valeur(d):
    #retourne une liste de tuples triée selon les valeurs
    return sorted(d.items(), key=lambda x: x[1])

def trie_par_cle(d):
    #retourne une liste de tuples triée selon les clés
    return sorted(d.items(), key=lambda x: x[0])

score_trie = trie_par_valeur(score)
score_trie.reverse()
for i in range(0,len(score_trie)):
    print(score_trie[i][0],":",score_trie[i][1])
print()
duel_trie = trie_par_cle(duel)
for i in range(0,len(duel_trie)):
    print(duel_trie[i][0][0],"contre",duel_trie[i][0][1],"gagne",duel_trie[i][1],"pts")

```

Analyse du programme

On aura besoin ici de quatre dictionnaires :

```
liste = {}
```

Le dilemme du prisonnier

```
strategie = {}
score = {}
duel = {}
```

Le dictionnaire `liste` mémorise la liste des coups durant un duel :

```
liste['Toujours seul'] = []
liste['Bonne poire'] = []
liste['Majorité'] = []
liste['Aléatoire'] = []
liste['Donnant donnant'] = []
```

Les stratégies sont « rangées » dans un dictionnaire. Remarquez l'utilisation des fonctions `lambda`. On est obligé de les utiliser parce que les fonctions-stratégies ont des paramètres.

```
strategie['Toujours seul'] = lambda lui, moi : toujours_seul(lui,moi)
strategie['Bonne poire'] = lambda lui, moi : bonne_poire(lui,moi)
strategie['Majorité'] = lambda lui, moi : majorite(lui,moi)
strategie['Aléatoire'] = lambda lui, moi : aleatoire(lui,moi)
strategie['Donnant donnant'] = lambda lui, moi : donnant_donnant(lui,moi)
```

Si la fonction n'a pas de paramètre, on peut se passer du `lambda`.

Par exemple :

```
def moi():
    print("Je m'appelle Jean")

fonct = {}
fonct['moi'] = moi()
fonct['moi']
```

écrira « Je m'appelle Jean ».

On met à zéro les scores des joueurs :

```
for joueur in liste.keys():
    score[joueur] = 0
```

Chaque joueur (*i*) affrontera une fois en duel chacun des autres joueurs (*j*), y compris lui-même. La condition `if i>=j` : est là pour n'autoriser qu'un duel et non pas deux.

Il faut aussi faire attention, quand *i* égale *j*, de ne pas compter les scores à double, de ne pas mettre tous les coups dans une seule liste et encore de mémoriser le bon score. Cela explique les trois `if i!=j` : du code ci-dessous.

```
for i in liste.keys(): # i et j sont les joueurs
    for j in liste.keys() :
        liste[i] = [] # on recommence une partie
        liste[j] = []
        if i>=j:
            nb_coups = 0
            score_joueur1 = 0
            score_joueur2 = 0
            while nb_coups < nb_total_coups :
                coup_joueur1 = strategie[i](liste[j],liste[i])
                coup_joueur2 = strategie[j](liste[i],liste[j])
                liste[i].append(coup_joueur1)
                if i!=j:
                    liste[j].append(coup_joueur2)
                score_joueur1 += gain(coup_joueur2,coup_joueur1)
                score_joueur2 += gain(coup_joueur1,coup_joueur2)
                nb_coups += 1
            duel[(i,j)] = score_joueur1
            if i!=j:
                duel[(j,i)] = score_joueur2
            score[i] += score_joueur1
            if i!=j:
```

```
score[j] += score_joueur2
```

Le dictionnaire `duel` va nous servir à détailler les scores à la fin de la simulation. On pourra ainsi mieux analyser les résultats du tournoi.

La fin du code permet d'afficher les résultats, triés par ordre décroissant pour les scores, et par ordre alphabétique pour les duels.

```
def trie_par_valeur(d):
    #retourne une liste de tuples triée selon les valeurs
    return sorted(d.items(), key=lambda x: x[1])

def trie_par_cle(d):
    #retourne une liste de tuples triée selon les clés
    return sorted(d.items(), key=lambda x: x[0])

score_trie = trie_par_valeur(score)
score_trie.reverse()
for i in range(0, len(score_trie)):
    print(score_trie[i][0], ":", score_trie[i][1])
print()
duel_trie = trie_par_cle(duel)
for i in range(0, len(duel_trie)):
    print(duel_trie[i][0][0], "contre", duel_trie[i][0][1], "gagne", duel_trie[i][1], "pts")
```



Exercice 6.5 - Tournoi

Le but de cet exercice est de définir une stratégie, c'est-à-dire de définir une règle pour savoir quand accepter et quand refuser la coopération avec un autre joueur. Les seules informations connues sont la liste des coups que l'adversaire a joués jusque-là, et la vôtre.

1. Programmez quelques stratégies dans le programme du § 6.5. N'oubliez pas le commentaire qui décrira succinctement votre stratégie.
2. Testez ces stratégies en modifiant le programme du § 6.6 et choisissez la meilleure.
3. Donnez votre stratégie préférée au professeur en vue du tournoi qui mettra en lice tous les élèves de la classe. Le nom de cette stratégie sera votre prénom.

Références

- Delahaye J.-P., *L'altruisme récompensé ?*, Pour la Science 181, novembre 1992, pp. 150-6
- Delahaye J.-P., Mathieu Ph., *L'altruisme perfectionné*, Pour la Science 187, mai 1993, pp. 102-7
- Delahaye J.-P., *Le dilemme du prisonnier et l'illusion de l'extorsion*, Pour la Science 435, janvier 2014, pp. 78-83
- Delahaye J.-P., Mathieu Ph., *Adoucir son comportement ou le durcir*, Pour la Science 462, avril 2016, pp. 80-5



Exercice 6.6 - Le dilemme du renvoi d'ascenseur

Situation ambiguë : je suis face à un inconnu au rez-de-chaussée de cette grande tour, devant ce petit ascenseur à une seule place dont la porte est ouverte. Il n'y a pas de bouton pour le rappel lorsqu'il sera monté, et je n'ai pas du tout, mais alors pas du tout, envie de gravir les huit étages à pied. L'inconnu me dit : « Laissez-moi passer en premier, car, arrivé en haut, je vous renverrai l'ascenseur. »

Dois-je accepter ? Pourquoi devrais-je le croire et accepter qu'il passe devant moi ? D'ailleurs tiendra-t-il sa promesse si je le laisse passer ? C'est le dilemme du renvoi d'ascenseur.

Lorsque nous imaginons que nous devons y jouer à de multiples reprises avec toutes sortes de gens, cette situation fait ressortir de multiples aspects inattendus. Cette variante du dilemme itéré du prisonnier (voir ex. 6.5) possède des propriétés remarquables.

Le dilemme du prisonnier

Refaites le tournoi de l'exercice 6.5 avec les gains ci-dessous :

```
def gain(lui,moi):
    if lui=='C' and moi=='C':
        return 3
    elif lui=='C' and moi=='T':
        return 8 # au lieu de 5
    elif lui=='T' and moi=='C':
        return 0
    elif lui=='T' and moi=='T':
        return 1
```

Que constatez-vous ?

Pourrez-vous trouver une meilleure stratégie pour ce dilemme ?

Référence : Delahaye J.-P., Mathieu Ph., *Le dilemme du renvoi d'ascenseur*, Pour la Science 269, mars 2000, pp. 102-6

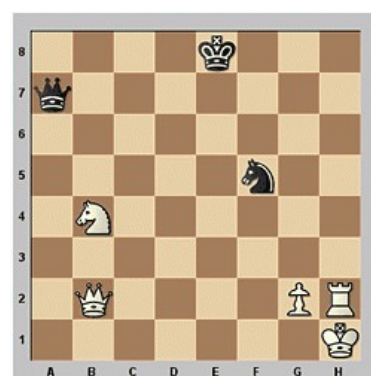


Exercice 6.7

Représentez la position du jeu d'échecs ci-contre en utilisant un dictionnaire.

La clé sera la position d'une case (un tuple, par exemple ('A', 7), la valeur sera la pièce qui s'y trouve (par exemple « dame noire »).

Ne mettez dans le dictionnaire que les cases occupées.



Exercice 6.8 - Le jeu des échelles

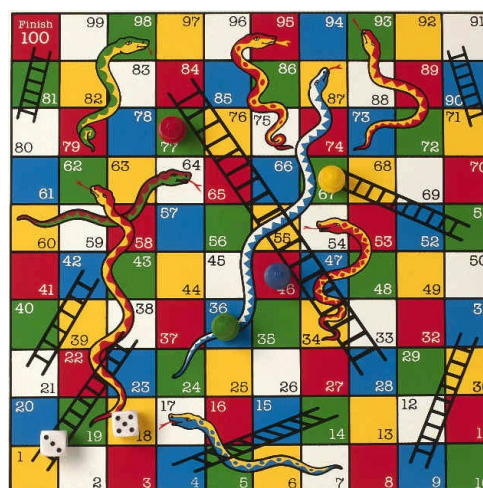
Règles du jeu

Le joueur lance un dé et avance d'autant de cases que de points sur le dé.

S'il tombe sur une case dans laquelle il y a le pied d'une échelle, il monte le long de celle-ci jusqu'en haut. S'il tombe sur une case dans laquelle il y a la tête d'un serpent, il doit redescendre jusqu'à la queue du serpent.

La partie se termine quand on arrive sur la case 100. Si le joueur tire un dé qui ne lui permet pas d'arriver exactement sur la case 100, il recule du nombre de points supplémentaires sur le dé.

Évidemment, une case ne peut être le départ ou l'arrivée de d'une seule échelle et un seul serpent.



Première question

La question qui nous intéresse est la suivante : « Comment de fois faudra-t-il (en moyenne) lancer le dé pour terminer la partie ? » Simulez 100'000 parties sur le plateau ci-dessus.

Deuxième question (plus difficile)

Déplacez (ou pas) horizontalement les serpents et les échelles du plateau ci-dessus de manière à :

- maximiser le nombre de lancers de dés
- minimiser le nombre de lancers de dés.



Exercice 6.9

Première partie

Écrivez un programme qui code un message en Morse :

entrée : code morse

donnera comme résultat :

```
['-.-.', '----', '-.-.', '.', ' ', '---', '----', '-.-.', '...', '.']
```

Vous trouverez l'alphabet Morse à l'adresse : www.apprendre-en-ligne.net/crypto/morse/



Écouteurs
obligatoires !

Seconde partie

Utilisez ensuite la méthode `Beep` du module `winsound` pour transformer les traits et les points en signaux sonores.

Voici ce que dit la documentation Python sur la méthode `Beep` :

```
winsound.Beep(frequency, duration)
```

`Beep` the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, **RuntimeError** is raised.



6.7. Ce que vous avez appris dans ce chapitre

- Vous avez vu comment créer un dictionnaire et le manipuler.
- Faites bien attention qu'un dictionnaire n'est pas ordonné. Cela peut parfois occasionner de mauvaises surprises...
- Vous avez aussi vu les fonctions `lambda`. Elles peuvent paraître superflues, mais sont utiles dans certains contextes, comme celui du jeu du dilemme du prisonnier, version Tournoi (§ 6.6). Généralement, on utilise plutôt le mot-clé `def`.
- Enfin, vous avez vu (et surtout entendu) une manière primitive de générer du son avec Python (exercice 6.9).



Chapitre 7

Jeux de lettres

7.1. Nouveaux thèmes abordés dans ce chapitre

- fichiers
- chaînes de caractères (string)
- ensembles (set)

7.2. Fichiers

On divise habituellement les fichiers en deux catégories.

Les **fichiers binaires** ne contiennent pas exclusivement du texte. Ils ne peuvent être « lus » que par des logiciels spécialisés. Une image JPEG, un fichier PDF, ou un MP3 sont des fichiers binaires.

Les **fichiers textes** sont « compréhensibles » par un humain : ils contiennent du texte (lettres, ponctuations, nombres, ...). Leur contenu est souvent organisé en lignes et on peut les lire avec des éditeurs de textes simples comme *NotePad++*, *WordPad*, etc.

Ici, nous allons exclusivement nous intéresser aux fichiers textes.

Pour jouer à des jeux de lettres, il nous faut d'abord constituer une liste de mots. On pourrait mettre tous ces mots dans une liste dans le corps du programme, mais il y a plusieurs inconvénients :

- on verra les mots en ouvrant le programme, ce qui gâchera le plaisir de les trouver ;
- si l'on a beaucoup de mots dans notre stock, cela nuira à la lecture du programme ;
- l'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

D'une manière générale, quand la quantité de données est importante, mieux vaut stocker ces dernières dans un **fichier**.

Un fichier est une **ressource**. L'accès aux ressources doit pouvoir être partagé entre les différents programmes qui peuvent en avoir besoin. Cela veut dire qu'il est nécessaire d'*acquérir* cette ressource avant de s'en servir, puis de la *libérer* après usage. Pour un fichier, cela se fait en l'ouvrant (`open`), puis en le fermant (`close`) :

```
# Ouverture d'un fichier en lecture:
fichier = open("/etc/passwd", "r")
...
# Fermeture du fichier
fichier.close()
```

Un fichier ouvert
doit toujours être
fermé après usage !

Le premier paramètre de la fonction `open` est le nom du fichier à manipuler, avec son chemin d'accès. Le second paramètre est le **mode** dans lequel on souhaite ouvrir le fichier. Le mode « r » (pour *read*) indique que nous allons ouvrir le fichier pour **lire** des données dedans. Tant qu'un fichier est ouvert, il ne peut pas être utilisé par un autre programme.

On peut aussi ouvrir un fichier pour **écrire** en utilisant le mode « w » (pour *write*) :

```
# Ouverture d'un fichier en écriture:
fichier = open("/etc/passwd", "w")
```



Mais attention ! Dans ce cas, si le fichier existe déjà, son contenu est **écrasé** ! Si vous voulez plutôt **ajouter** des données à la fin d'un fichier, il faut utiliser le mode « a » (pour *append*):

```
# Ouverture d'un fichier en ajout:
fichier = open("/etc/passwd", "a")
```

7.2.1. Lire dans un fichier

La lecture dans un fichier texte se fait de manière séquentielle, une ligne après l'autre (c'est la raison pour laquelle on parle de fichier à accès **séquentiel**). Il n'est pas possible de lire une ligne au milieu du fichier avant d'avoir lu toutes celles placées avant.

Il est donc nécessaire de connaître la structure du fichier pour pouvoir le lire correctement. Par exemple, dans les fichiers que nous allons utiliser, chaque ligne du fichier comportera un seul mot.

La méthode `readlines` lit **toutes les lignes** du fichier et les met dans une liste. Elle est utile si on souhaite lire tout le fichier d'un coup :

```
toutesleslignes = fichier.readlines()
```

La méthode `readline` (au singulier) lit **la ligne suivante** dans le fichier.

```
ligne = fichier.readline()
```

Enfin, pour traiter l'une après l'autre toutes les lignes d'un fichier, on peut utiliser une boucle `for` :

```
for ligne in fichier:
    ...
```

7.2.2. Écrire dans un fichier

Écrire dans un fichier se fait très simplement en utilisant la méthode `write()`.

```
fichier.write("ce qu'il faut écrire\n")
```

Les symboles `\n` indiquent que l'on saute à la ligne suivante.

Ces données sont enregistrées dans le fichier les unes à la suite des autres. Chaque nouvel appel de `write()` continue l'écriture à la suite de ce qui est déjà enregistré.

7.3. Chaînes de caractères



un string
↑
une string

Les chaînes de caractères forment la classe 'str' (pour *string*). Une chaîne de caractères est une suite de symboles placées entre guillemets ("salut") ou entre apostrophes ('salut'). L'intérêt d'avoir ces deux délimiteurs est de pouvoir écrire des chaînes contenant des apostrophes ("aujourd'hui") ou des guillemets ('Pressez "enter" pour continuer').

Les méthodes de la classe 'str' sont nombreuses. Elles sont détaillées dans la documentation officielle de Python à cette adresse :

<http://docs.python.org/dev/library/stdtypes.html#string-methods>

Nous ne verrons ici que les plus utiles.

Prenons comme exemple de chaîne de caractères :

```
chaîne = "ma Chaîne de Caractères"
```

7.3.1. Accès à des sous-chaînes

La fonction intégrée `len`, que l'on a déjà rencontrée, permet de connaître la longueur (*length* en anglais) d'une chaîne de caractères :



```
len(chaîne)
```

donnera comme résultat 23. Les positions de la string `chaîne` vont donc de 0 à 22.

On peut accéder à des lettres isolées de la chaîne ou à des sous-chaînes d'une manière analogue à celle utilisée avec les listes.

```
print(chaîne[0])
```

donnera le premier caractère de la chaîne ; dans notre exemple 'm'.

```
print(chaîne[3:])
```

donne la sous-chaîne commençant à la troisième position, à savoir 'Chaîne de Caractères'. (la numérotation des places commence à 0, comme pour les listes). Inversement,

```
print(chaîne[:9])
```

donne la sous-chaîne se terminant à la huitième (!) position, à savoir 'ma Chaîne'. En combinant les deux :

```
print(chaîne[3:9])
```

on obtiendra la sous-chaîne 'Chaîne'.

Il est possible de transformer une chaîne de caractères en une liste de « mots ». Il faut juste indiquer ce qui joue le rôle de séparateur. Par exemple, si le séparateur est l'espace,

```
chaîne.split(' ')
```

produira comme résultat : ['ma', 'Chaîne', 'de', 'Caractères'].

Pour avoir la liste des caractères de la chaîne, rien de plus facile :

```
list(chaîne)
```

retournera la liste ['m', 'a', ' ', 'C', 'h', 'a', 'î', 'n', 'e', ' ', ' ', 'd', 'e', ' ', ' ', 'C', 'a', 'r', 'a', 'c', 't', 'è', 'r', 'e', 's'].

On peut parcourir tous les caractères d'une chaîne avec l'instruction `for` :

```
for ch in chaîne:
    print(ch, end=" ")
```

produira comme résultat : m a C h a î n e d e C a r a c t è r e s

On peut savoir si une sous-chaîne se trouve dans une chaîne :

```
'de' in chaîne
```

donnera `True`. Il est aussi possible de connaître la position où commence une sous-chaîne :

```
print(chaîne.index('de'))
```

donnera 10, car « de » apparaît pour la première fois à la position numéro 10. Si la sous-chaîne

recherchée n'existe pas, le programme renvoie un message d'erreur :

```
print(chaine.index('u'))
```

produit **ValueError: substring not found**

7.3.2. Mises en forme

Pour tout écrire en majuscules :

```
print(chaine.upper())
```

donnera comme résultat : 'MA CHAÎNE DE CARACTÈRES'

Pour tout écrire en minuscules :

```
print(chaine.lower())
```

donnera comme résultat : 'ma chaîne de caractères'

Pour écrire la première lettre en majuscules et les autres en minuscules :

```
print(chaine.capitalize())
```

donnera comme résultat : 'Ma chaîne de caractères'

7.3.3. Manipulations des strings



Attention ! Contrairement aux listes, **les chaînes de caractères ne sont pas modifiables**. Ainsi, une affectation du type :

```
chaine[0]="T"
```

donnera le message d'erreur : **TypeError: 'str' object does not support item assignment**

On peut cependant faire des opérations sur des chaînes de caractères. On peut par exemple enlever des caractères qui se trouvent en début ou en fin de chaîne avec la méthode `strip()` :

```
print('grand-maman'.strip('-agmn'))
```

donnera comme résultat : 'rand'

Si la fonction `strip()` n'a pas d'argument, on enlèvera par défaut des espaces. Si on veut enlever des caractères en début de chaîne, il existe la méthode `lstrip()` ; si on veut enlever des caractères en fin de chaîne on utilisera la méthode `rstrip()` :

```
print('grand-maman'.lstrip('-agmn'))
```

donnera comme résultat 'rand-maman', tandis que

```
print('grand-maman'.rstrip('-agmn'))
```

donnera comme résultat 'grand'

On peut concaténer deux chaînes avec l'opérateur `+` :

```
chaine1 = "ma Chaîne"  
chaine2 = " de Caractères"  
chaine = chaine1 + chaine2  
print(chaine)
```

donnera 'ma Chaîne de Caractères'.

On peut aussi répéter une chaîne de caractères plusieurs fois :

```
chaine = "salut "
chaine = chaine*4
print(chaine)
```

donnera : 'salut salut salut salut '

7.4. Le pendu - règles du jeu

Le jeu du pendu consiste à retrouver un mot le plus vite possible (avant que le dessin du pendu soit terminé) en proposant des lettres. Si la lettre appartient au mot, elle est écrite aux bons emplacements, sinon on continue de dessiner le pendu.

Nous allons commencer par une version non graphique où il s'agira de deviner un mot avec le moins d'essais possible. Nous verrons une version graphique au chapitre 8.

Exemple de partie

```
-----
Entrez une lettre ou '?' pour abandonner : E
-----
Entrez une lettre ou '?' pour abandonner : U
U---U-
Entrez une lettre ou '?' pour abandonner : A
U-A-U-
Entrez une lettre ou '?' pour abandonner : N
U-ANU-
Entrez une lettre ou '?' pour abandonner : S
U-ANUS
Entrez une lettre ou '?' pour abandonner : R
URANUS
Bravo ! Le mot URANUS a été trouvé en 6 coups
```

Remarque : les lettres en gras ont été entrées au clavier par le joueur.

7.5. Code du programme



pendu.py

```
# Le jeu du pendu
from random import choice

fichier = open("liste_mots.txt", "r")
liste_mots = fichier.readlines() # met tous les mots du fichiers dans une liste
mot = choice(liste_mots) # prend au hasard un mot dans la liste
mot = mot.rstrip() # supprime le caractère "saut à la ligne"
fichier.close()

mot_devine = "-" * len(mot)
print(mot_devine)
nbr_essais = 0

while mot_devine != mot:
    lettre = input("Entrez une lettre ou '?' pour abandonner : ")
    lettre = lettre[0] # évite des erreurs si un mot est entré au lieu d'une lettre
    if lettre == '?':
        print('Le mot était',mot)
        break
    lettre = lettre.upper()
    for i in range(len(mot)):
        if lettre == mot[i]:
            mot_devine = mot_devine[:i] + lettre + mot_devine[i+1:]
    print(mot_devine)
    nbr_essais += 1

if mot == mot_devine:
    print('Bravo ! Le mot',mot,'a été trouvé en',nbr_essais,'coups')
```

7.6. Analyse du programme

```
from random import choice
```

La fonction `choice` permet de choisir un élément au hasard dans une liste. Ici, il s'agira de choisir un mot.

```
fichier = open("liste_mots.txt", "r")
liste_mots = fichier.readlines() # met tous les mots du fichiers dans une liste
mot = choice(liste_mots)         # prend au hasard un mot dans la liste
mot = mot.rstrip()               # supprime le caractère "saut à la ligne"
fichier.close()
```

Tous les mots se trouvent dans le fichier `liste_mots.txt`. Ce fichier doit se trouver au même niveau que le programme.

```
mot_devine = "-" * len(mot)
print(mot_devine)
nbr_essais = 0
```

Au début, on écrit une suite de tirets pour indiquer la longueur du mot. Ces tirets seront remplacés par des lettres au fur et à mesure que le joueur les découvrira.

```
while mot_devine != mot:
```

La boucle principale se répète tant que l'on n'a pas découvert le mot.

```
    lettre = input("Entrez une lettre ou '?' pour abandonner : ")
    lettre = lettre[0] # évite des erreurs si un mot est entré au lieu d'une lettre
    if lettre == '?':
        print('Le mot était', mot)
        break
```

On demande une lettre au joueur. Il peut en fait entrer n'importe quoi, mais cela ne le fera pas avancer dans sa quête... S'il entre un point d'interrogation, alors on donnera le mot à trouver et on sortira de la boucle par l'instruction `break`.

```
        lettre = lettre.upper()
```

Comme dans le fichier contenant notre stock de mots, tous les mots sont écrits en majuscules, on s'assure que la lettre entrée est bien une majuscule.

```
            for i in range(len(mot)):
                if lettre == mot[i]:
                    mot_devine = mot_devine[:i] + lettre + mot_devine[i+1:]
            print(mot_devine)
            nbr_essais += 1
```

La variable `i` parcourt toutes les positions du mot. Si la lettre proposée se trouve à la position `i`, alors on remplace le tiret par cette lettre (3^{ème} ligne du bout de code ci-dessus). On écrit ensuite le nouveau résultat intermédiaire et on incrémente le nombre d'essais.

```
if mot == mot_devine:
    print('Bravo ! Le mot', mot, 'a été trouvé en', nbr_essais, 'coups')
```

Après être sorti de la boucle, on écrit un petit message de félicitation si le bon mot a été découvert.



Exercice 7.1

Imaginons que la fonction `choice` n'existe pas. Comment choisir un mot au hasard dans la liste `liste_mots` ?



Exercice 7.2

Réutilisez le fichier `liste_mots.txt` pour un autre jeu qui consistera à retrouver un mot, l'ordinateur ayant écrit ses lettres par ordre alphabétique. Par exemple :

```
AEIIMMNPRT
Entrez l'anagramme : imprimante
Bravo !
```

7.7. Ensembles

Les ensembles en Python sont les mêmes ensembles qu'en mathématiques. On peut leur appliquer les mêmes opérations : union, intersection, différence symétrique, différence. On peut aussi tester si un ensemble est inclus dans un autre.

Voici un programme qui résume toutes ces opérations.



ensembles.py

```
ens1 = set([1, 2, 3, 4, 5, 6])
ens2 = set([5, 6, 7, 8])
ens3 = set([2, 4, 6])

# union
print(ens1 | ens2)
# intersection
print(ens1 & ens2)
# différence symétrique
print(ens1 ^ ens2)
# différence
print(ens1 - ens2)

# inclusion
print(ens2.issubset(ens1))
print(ens3.issubset(ens1))

# transformation en une liste
listel = list(ens1)
print(listel)
```

Voici le résultat de ce programme :

```
{1, 2, 3, 4, 5, 6, 7, 8}
{5, 6}
{1, 2, 3, 4, 7, 8}
{1, 2, 3, 4}
False
True
[1, 2, 3, 4, 5, 6]
```

Remarquez que dans un ensemble, chaque élément n'apparaît qu'**une seule fois**.

Notons enfin que l'on peut facilement convertir une chaîne de caractères en un ensemble :

```
set('carnaval')
```

retournera l'ensemble : {'l', 'n', 'a', 'c', 'v', 'r'}



Exercice 7.3

Écrivez un programme qui remplace les voyelles d'un texte (éventuellement accentuées) par une étoile. Par exemple « Il était une fois » deviendra « *l *t***t *n* f**s ».



Exercice 7.4

Pour toutes les questions ci-dessous, utilisez le fichier dico.txt.

- Calculez le pourcentage de mots français où la seule voyelle est le « e » (il peut y en avoir plusieurs dans le mot).
Par exemple : exemple, telle, égrener.
- Calculez le pourcentage de mots français où deux lettres identiques se suivent.
Par exemple : femmme, panne, créer.
- Affichez les mots de moins de 10 lettres ayant comme lettre centrale un « w ». Le mot doit avoir un nombre impair de lettres.
Par exemple : edelweiss, kiwis.
- Affichez la liste des mots contenant la suite de lettres « dile ».
Par exemple : crocodile, prédilection.
- Affichez les mots de moins de 5 lettres qui commencent et finissent par la même lettre.
Par exemple : aura, croc, dard.
- Affichez tous les mots palindromes.
Par exemple : serres, kayak, ressasser.
- Affichez tous les mots anacycliques : un mot lu de droite à gauche donne un autre mot.
Par exemple : les – sel, bons – snob.
- Affichez tous les mots composés de deux séquences de lettres qui se répètent.
Par exemple : papa, chercher.



dico.txt



Exercice 7.5

Pour toutes les questions ci-dessous, utilisez le fichier dico.txt.

Dans le jeu du pendu, les mots les plus difficiles à trouver sont ceux qui ont peu de voyelles et des consonnes rares. Voici les lettres de la plus fréquente à la moins fréquente, en français :

Lettres courantes : E A I S T N R U L O D M P

Lettres rares : C V Q G B F J H Z X Y K W

- Affichez la liste des mots n'ayant que des voyelles et des lettres rares.
- Affichez la liste des mots de plus de 15 lettres n'ayant que des lettres courantes.
- Affichez la liste des mots n'ayant aucune des lettres E A I S T N R U.
- Affichez la liste des mots ayant exactement deux voyelles et plus de 9 caractères (tiret compris). Par exemple : transports, check-list.
- Affichez la liste des mots commençant par au moins 4 consonnes consécutives (tiret non autorisé).



dico.txt



Exercice 7.6

L'argot *javanais*, apparu en France dans la dernière moitié du 19^{ème} siècle, consiste à intercaler dans les mots la syllabe « av » entre une consonne et une voyelle. Nous n'utiliserons ici que les règles simplifiées (*) :

- On rajoute « av » après chaque consonne (ou groupe de consonnes comme par exemple ch, cl, ph, tr,...) d'un mot.

- Si le mot commence par une voyelle, on ajoute « av » devant cette voyelle.
- On ne rajoute jamais « av » après la consonne finale d'un mot.

Écrivez un programme qui traduit un texte français en javanais. Par exemple,

Je vais acheter des allumettes au supermarché.

deviendra

Jave vavais avachavetaver daves avallavumavettaves avau savupavermavarchavé.

(*) Pour les règles complètes, voir [https://fr.wikipedia.org/wiki/Javanais_\(argot\)](https://fr.wikipedia.org/wiki/Javanais_(argot))

7.8. Le mot le plus long – règles du jeu

Les règles sont élémentaires : on entre une série de lettres et l'ordinateur essaie de trouver le plus long mot possible avec ces lettres. Il peut y avoir plusieurs fois la même lettre dans la série.

Exemple de partie

```
entrez votre tirage : urtefaaijg
10 lettres
Le script a mis 0.184 s pour trouver des mots de 9 lettres
['fatiguera', 'jaugerait']
```



Exercice 7.7

Nous allons utiliser les ensembles dans l'implémentation du jeu « Le mot le plus long ». Mais avant cela, nous aurons besoin d'un lexique ordonné, que nous allons élaborer à partir du fichier texte « dico.txt », qui contient une liste de 323'467 mots classés par ordre alphabétique.

Utilisez « dico.txt » pour faire un dictionnaire (au sens Python du terme, voir chapitre 6), où la clé sera la longueur du mot, et où la valeur sera la liste alphabétique des mots de cette longueur. Si des mots contiennent un ou des tirets, il faudra supprimer les tirets ! Ainsi, le mot « porte-avions » deviendra « porteavions ». Idem avec les apostrophes.

7.9. Code du programme



motlepluslong.py

```
# Le mot le plus long
from time import time

def dictionnaire_ordonne():
    # on lit le fichier et on range les mots alphabétiquement selon leur longueur
    fichier = open("dico.txt", "r")
    dict_ord = {}
    for longueur in range(26):
        dict_ord[longueur+1] = []
    mot = fichier.readline()
    while mot != '':
        mot = mot.strip('\n')
        if '-' in mot:
            mot = mot.replace('-', '')
        if "'" in mot:
            mot = mot.replace("'", '')
        dict_ord[len(mot)].append(mot)
        mot = fichier.readline()
    fichier.close()
    return dict_ord

def lettres_multiples_ok(mot, tirage):
    # teste si chaque lettre figure suffisamment de fois dans le tirage
    for lettre in mot:
        if lettre in tirage:
            tirage.remove(lettre)
```

```

        else:
            return False
        return True

def trouver_plus_long_mot(dico, tirage):
    longueur_mot = len(tirage)
    solution = []
    set_tirage = set(tirage)
    while longueur_mot > 0:
        for mot in dico[longueur_mot]:
            if set(mot).issubset(set_tirage):
                # les lettres du mot sont un sous-ensemble du tirage
                tirage_test = list(tirage)
                if lettres_multiples_ok(mot, tirage_test):
                    solution.append(mot)
        if solution != [] or longueur_mot == 1:
            return solution, longueur_mot
        longueur_mot -= 1

dico = dictionnaire_ordonne()
jouer = True
while jouer:
    tirage = input('Entrez votre tirage : ').lower()
    print(len(tirage), 'lettres')
    t0 = time()
    solution, longueur = trouver_plus_long_mot(dico, tirage)
    if solution == []:
        print('Pas de mot trouvé !')
    else:
        t1 = time() - t0
        print('Le script a mis', '{0:.3f}'.format(t1),
              's pour trouver des mots de', longueur, 'lettres')
        print(solution)
    rejouer = input('Rejouer ? (o/n) : ')
    jouer = (rejouer == "o")

```

7.10. Analyse du programme

```
from time import time
```

C'est dans le module `time` que se trouve la fonction... `time()`, qui renvoie un nombre réel correspondant au nombre de secondes écoulées depuis le 1^{er} janvier 1970 à 00:00:00. Nous allons utiliser cette fonction pour chronométrer la durée de la recherche du mot le plus long ; nous verrons comment un peu plus loin.

```

def dictionnaire_ordonne():
    # on lit le fichier et on range les mots alphabétiquement selon leur longueur
    fichier = open("dico.txt", "r")
    dict_ord = {}
    for longueur in range(25):
        dict_ord[longueur+1] = []

```

La fonction `dictionnaire_ordonne` était l'objet de l'exercice 7.5. Le dictionnaire est « découpé » en 25 listes, la liste k comprenant tous les mots de k lettres classés alphabétiquement, avec k allant de 1 à 25. Rappelons que le plus long mot de la langue française comporte 25 lettres : « anticonstitutionnellement ».

```

mot = fichier.readline()
while mot != '':
    mot = mot.strip('\n')
    if '-' in mot:
        mot = mot.replace('-', '')
    if '"' in mot:
        mot = mot.replace('"', '')
    dict_ord[len(mot)].append(mot)
    mot = fichier.readline()

```


On lit tous les mots du fichier, ligne par ligne (il y a un mot par ligne dans le fichier) ; on supprime le caractère saut de ligne :

```
mot = mot.strip('\n')
```

et les tirets qui se trouveraient dans le mot (idem pour les apostrophes)

```
mot = mot.replace('-', '')
```

puis on ajoute à la liste correspondante ce mot.

```
dict_ord[len(mot)].append(mot)
```

Le fichier est finalement fermé et le dictionnaire ordonné est retourné.

```
fichier.close()
return dict_ord
```

La fonction `lettres_multiples_ok()` teste si chaque lettre figure suffisamment de fois dans le tirage. L'idée est simple : les lettres du mot sont enlevées une à une de la liste tirage. Si la lettre courante du mot n'est pas dans la liste tirage, le mot ne peut pas être écrit et la fonction renvoie False.

```
def lettres_multiples_ok(mot, tirage):
    # teste si chaque lettre figure suffisamment de fois dans le tirage
    for lettre in mot:
        if lettre in tirage:
            tirage.remove(lettre)
        else:
            return False
    return True
```

La fonction `trouver_plus_long_mot()` parcourt le dictionnaire ordonné, en commençant par la liste des plus longs mots possibles.

```
def trouver_plus_long_mot(dico, tirage):
    longueur_mot = len(tirage)
    solution = []
    set_tirage = set(tirage)
    while longueur_mot > 0:
        for mot in dico[longueur_mot]:
```

La ligne qui suit fait un prétraitement : avant de voir si les lettres sont en suffisance, on regarde d'abord si toutes les lettres du mot sont dans le tirage.

```
        if set(mot).issubset(set_tirage):
            # les lettres du mot sont un sous-ensemble du tirage
            tirage_test = list(tirage)
            if lettres_multiples_ok(mot, tirage_test):
                solution.append(mot)
```

Si une solution a été trouvée, on la retourne. Sinon, on essaie avec la liste des mots ayant une lettre de moins.

```
        if solution != [] or longueur_mot==1:
            return solution, longueur_mot
        longueur_mot -= 1
```

Le programme principal ne présente pas de difficultés.

```
dico = dictionnaire_ordonne()
jouer = True
while jouer:
    tirage = input('Entrez votre tirage : ').lower()
    print(len(tirage), 'lettres')
```

Jeux de lettres

```
t0 = time()
solution, longueur = trouver_plus_long_mot(dico, tirage)
if solution == []:
    print('Pas de mot trouvé !')
else:
    t1 = time()-t0
    print('Le script a mis', '{0:.3f}'.format(t1),
          's pour trouver des mots de', longueur, 'lettres')
    print(solution)
rejouer = input('Rejouer ? (o/n) : ')
jouer = (rejouer == "o")
```

Pour chronométrer le temps de calcul, on enregistre « l'heure » dans la variable `t0` avant de lancer la recherche du mot le plus long, puis, juste après la recherche, on enregistre la différence entre « la nouvelle heure » et `t0`.

```
t1 = time()-t0
```



Exercice 7.8

Est-il possible de se passer des ensembles dans le programme du § 7.9 ? Essayez de supprimer du programme la ligne :

```
if set(mot).issubset(set_tirage)
```

ainsi que les lignes ayant un rapport avec ce prétraitement.

Votre programme est-il plus rapide ? Pour le savoir, testez-le avec la fonction `time()`.



Exercice 7.9

Modifiez le programme du § 7.9 pour qu'il écrive toutes les anagrammes des mots d'une longueur donnée par l'utilisateur.

Par exemple :

```
Longueur des mots : 5
abats : ['basat', 'batas']
abeti : ['batie', 'beait']
abime : ['amibe', 'iambe']
...
zoner : ['ornez']
```



Exercice 7.10

Au Scrabble (version française), les valeurs des lettres sont les suivantes :

- A, E, I, L, N, O, R, S, T, U 1 point
- D, G, M 2 points
- B, C, P 3 points
- F, H, V 4 points
- J, Q 8 points
- K, W, X, Y, Z 10 points



Modifiez le programme du § 7.9 pour qu'il écrive les mots qui rapporteraient le plus de points au Scrabble, en utilisant les lettres données par l'utilisateur.

Par exemple :

```
Entrez votre tirage : deefirrsv
['feviers', 'fevrier', 'fievres'] : 13 points au Scrabble
```

Exercice 7.11



La **saisie intuitive** est une technologie conçue afin de simplifier la saisie de texte sur les claviers téléphoniques.

Là où une combinaison de touches correspond, dans le système traditionnel, à un et un seul mot, le système de saisie intuitive doit faire face à un problème : la pluralité des combinaisons de lettres correspondant à une même saisie. Par exemple, le nombre 7243 correspond à $4 \times 3 \times 3 \times 3 = 108$ combinaisons de lettres possibles. Mais seules quelques-unes correspondront (peut-être) à un mot français.



Écrivez un programme permettant de trouver tous les mots correspondant à un nombre donné.

```
Entrez le code votre mot (p. ex. 665587783) : 7243
Le script a mis 0.015 s pour trouver :
['page', 'paie', 'rage', 'raid', 'raie', 'sage', 'scie']
```

Exercice 7.12



Le jeu « Motus » repose sur la recherche de mots d'un nombre fixé de lettres (entre 7 et 10). Un candidat propose un mot qui doit contenir le bon nombre de lettres et être correctement orthographié, sinon il est refusé. Le mot apparaît alors sur une grille : les lettres bien placées sont colorées en rouge, les lettres présentes mais mal placées sont en jaune.

C	A	S	T	O	R
C	I	N	E	M	A
C	Y	P	R	E	S
C	I	T	R	O	N

Une lettre ne peut être colorée au maximum que le nombre de fois que cette lettre apparaît dans le mot. Par exemple, si le mot cherché est CABLES et que l'on propose CABANE, on aura la coloration :

C	A	B	A	N	E
---	---	---	---	---	---

et non pas

C	A	B	A	N	E
---	---	---	---	---	---

car il n'y a qu'un « A » dans CABLES.

Mots non valables

- les noms propres ;
- les mots mal orthographiés ;
- les mots composés (chewing-gum, week-end, ...);
- les verbes conjugués (seuls les infinitifs et les participes passés et présents sont acceptés) ;
- les mots qui ne commencent pas par la première lettre indiquée ;
- les mots qui ne respectent pas le nombre de lettres donné.

Votre programme devra :

- prendre un mot au hasard dans le fichier « dico.txt ». Ce mot devra avoir entre 7 et 10 lettres. Contrairement aux règles originales, on permettra les formes conjuguées ;
- indiquer le nombre de lettres du mot à trouver et donner la première lettre ;
- écrire le numéro de l'essai ;
- écrire sous la proposition du joueur :
 - une lettre majuscule si elle est bien placée ;
 - une lettre minuscule si elle est mal placée ;
 - un point si la lettre n'est pas dans le mot, ou si elle a moins d'occurrences ;
 - un message d'erreur si le mot est non valide.

Exemples de partie

Vous cherchez un mot de 7 lettres. Vous avez 6 essais.

O.....
1 orageux
OrA.e..
2 opaline
O.A.I.e
3 otaries
OTARIES
Bravo !

Vous cherchez un mot de 8 lettres. Vous avez 6 essais.

F.....
1 fouinant
FOUi.a.t
2 febriles
Fe..i..s
3 fouettes
FOUet..s
4 foutasse
Pas dans le dictionnaire
5 foutaise
FOUTAISE
Bravo !



Vous cherchez un mot de 7 lettres. Vous avez 6 essais.

T.....
1 trousse
T.o.s..
2 passeur
Première lettre non valide
3 traceur
T.aC...
4 tamisees
Longueur non valide
5 tapisse
TA..s..
6 tapoter
TA.o...
Désolé, la réponse était TANCONS



Exercice 7.13

Deux joueurs vont s'affronter en lançant plusieurs fois de suite une pièce de monnaie. Avant la partie, chacun des joueurs choisit une séquence de trois résultats (chaque résultat étant Pile ou Face). Puis, on lance la pièce de monnaie autant de fois qu'il le faut (en notant à chaque fois le résultat) jusqu'à ce qu'une des deux séquences apparaisse.

Par exemple, avant de commencer à lancer la pièce, le joueur 1 choisit la séquence *PFP* (Pile-Face-Pile) et le joueur 2 choisit la séquence *FPP*. On lance ensuite la pièce plusieurs fois de suite et voici les issues obtenues successivement :

P P F F P F F F P P

Cette partie a duré 10 lancers et la première des deux séquences sortie est celle du joueur 2, qui est alors déclaré vainqueur.

A priori, les deux joueurs devraient avoir la même chance de gagner... en tout cas, c'est ce que l'intuition nous laisse penser. Ce n'est pourtant pas le cas. Si vous connaissez la séquence choisie par l'adversaire, vous pourrez définir la vôtre et gagner plus souvent que votre adversaire.

En simulant des milliers de parties, trouvez la meilleure séquence du joueur 2 à opposer à chacune des 8 séquences possibles du joueur 1.

Déterminez ensuite la stratégie gagnante.



7.11. Le défi Turing

Avec vos connaissances actuelles en Python, vous pourrez résoudre plus ou moins facilement presque tous les exercices du défi Turing : www.apprendre-en-ligne.net/turing/

Le défi Turing est une série d'énigmes mathématiques qui pourront difficilement être résolues sans un programme informatique. **Attention !** Votre programme devra trouver la réponse **en moins d'une minute !**

256 problèmes à résoudre ! Vous avez du grain à moudre...



7.12. Ce que vous avez appris dans ce chapitre

- Vous avez vu comment stocker et lire des données dans un fichier texte.
- Une chaîne de caractères (*string*) est une suite ordonnée de caractères (lettres, chiffres, symboles). Elle n'est pas modifiable, mais on peut quand même la manipuler grâce aux nombreuses méthodes de la classe 'str'.
- Vous avez vu que l'on peut manipuler des ensembles (*set*) comme en mathématiques. Les ensembles sont parfois pratiques, notamment pour éliminer des doublons.

Index alphabétique

A

`add_cascade()` 9-5
`add_command()` 9-6
affectation 1-7, 2-4
`after()` 9-3, 12-4
alpha-bêta (élagage) 15-14
anagramme 7-12
`append` 5-4
arbre de jeu 15-10
argot javanais 7-8
ASCII 8-5
attribut 10-2
automate cellulaire 12-1

B

`Beep` 6-12
bibliothèque d'ouvertures 15-20
`bind` 12-5
blackjack 10-4
booléen 1-7
boucle infinie 2-5
boucles imbriquées 2-5
bouton radio 13-11
`Button` 4-6, 8-4
`Button-1` 12-4
`break` 2-8, 7-6

C

calcul mental 2-7
canevas 8-4
`canvas()` 8-4
`capitalize()` 7-4
cavalier (parcours du) 14-10
cellulaire (automate) 12-1
Chernoff (figure de) 11-7
chiffre de César 8-6
chiffre de substitution 6-4
chiffre de Vigenère 8-6
`choice()` 1-4, 7-5
Chomp (jeu de) 14-12
`chr()` 8-5
classe 5-2, 10-2
clic droit de la souris 12-5
`close()` 7-1
code ASCII 8-5
commentaire 2-2
comparaison 2-5
condition 2-7
`configure` 4-6
constructeur 10-2
conversion de types 2-6
`coord()` 16-2
copie d'une liste 5-5
`copy()` 6-4
couleurs 11-1, 11-3
`create_arc()` 11-5
`create_image()` 11-5
`create_line()` 11-4

`create_oval()` 11-4
`create_polygon()` 11-4
`create_rectangle()` 11-4
`create_text()` 8-4
crible d'Ératosthène 5-7
curseur 14-8

D

damier 14-1
`def` 3-3
`del()` 5-4
`delete` 8-3
démineur 13-5
`destroy` 4-7, 11-3
Devine mon nombre ! 2-1
diagramme à secteurs 11-6
dictionnaire 6-1
dilemme du prisonnier 6-6
dilemme du renvoi d'ascenseur 6-11
disque 11-4
drag and drop 14-7

E

Echecs 14-9, 15-1
effet de bord 3-4
encapsulation 10-8
ensemble 7-7
Euro Millions 1-5
évaluation 15-12
événement 4-2, 16-4
`except` 2-8
exception 2-8

F

fenêtre 4-5, 8-4
FEN 14-11
fichier 7-1
figure de Chernoff 11-7
`find_closest()` 9-5
fonction 3-4
fonction lambda 6-5, 8-7
`for ... in ...` 1-4, 5-7, 6-3, 6-4
formatage 2-6, 7-11
frame 9-7
frames multiples 13-11
`from ... import ...` 1-3, 1-4

G

Gasp 12-9
`get()` 6-3
glisser-déposer 14-7
global 3-4
grid 4-6

H

héritage 10-9
héritage multiple 10-9
heuristique de Warnsdorff 14-10

hexagone 14-4
histogramme 11-6
horloge de Berlin 11-11

I

if... elif... else... 1-6, 2-7
importation des images 4-5
in 5-5, 6-2, 7-3
incrémentation 2-5
indentation 1-4
index() 5-5, 7-3
indice 5-2
initialisation 2-2
insertion dans une liste 5-5
instance 10-2
input() 2-6
int 2-6
issubset() 7-7
items() 6-2

J

jeu d'échecs 15-1
jeu de la vie 12-1
jour de la semaine 5-11
Juniper Green 5-1

K

Keith (algorithme de) 5-11
keys() 6-2

L

Label 4-6
labyrinthe 13-1
lambda (fonction) 6-5, 8-7
le loup et les moutons 14-11
le mot le plus long 7-9
Let's Make A Deal ! 4-9
len() 5-4, 7-3
licence GNU GPL 15-2
list() 5-3, 7-3
listes 1-4, 5-2
livret 2-6, 2-7
localtime() 11-12
lower() 7-4
lstrip() 7-4
Logo 11-10

M

mailbox 15-4
mainloop 4-8
marche de l'ivrogne 11-5
Memory 9-1
Menu() 9-5
menus 9-5
méthode 5-2, 10-2
minimax 15-12
modèle de Schelling 12-8
Morse (code) 6-12
mots réservés 2-3
Motus 7-13, 8-8
mutable 3-4

N

nombres premiers 5-7
nombres pseudo-aléatoires 1-2

O

objet 5-2, 10-2
open() 7-1
opérateurs de comparaison 1-6
opérateurs sur les entiers 2-4
ord() 8-5
ovale 11-4

P

pack() 8-4, 8-7
parcours du cavalier 14-10
pendu 7-5, 8-1
PhotoImage() 4-5, 8-4
Pierre, papier, ciseaux 3-1, 4-1
place() 8-4, 8-8
planche de Galton 16-3
planète Torus 12-6
polygone 11-4
polymorphisme 10-10
polymorphisme ad hoc 10-10
polymorphisme d'héritage 10-11
polymorphisme paramétrique 10-10
pong 16-5
print() 1-3
procédure 3-3
programmation orientée objet 10-1
programme piloté par des événements 4-2

R

randint 1-3
random (module) 1-3
random() (fonction) 1-4
randrange 12-4
range 5-3
readline() 7-2
readlines() 7-2
récursivité 12-4, 13-1
rectangle 11-4
remove 5-4
replicator 12-5
return 3-4
reverse() 5-4
RGB 11-3
Risk 2-9, 5-10
rstrip() 7-4
RVB 11-3

S

sample() 1-5
Schelling (modèle de) 12-9
Scrabble 7-12
secteur 11-5
segment 11-4
seed 1-4
set 7-7
shuffle() 1-5
sort() 5-4

sous-chaine 7-3

split() 7-3

strip() 7-4

T

time (module) 7-9

time() (fonction) 7-11

title 4-5, 8-4

Tk 4-5, 8-4

tkinter 4-4

Toplevel 4-9

tortue 11-10

try 2-8

tuple 5-11

types mutables et non mutables 3-4

U

update() 16-2

upper() 7-4, 8-6

V

values() 6-2

variable 2-2, 2-3

variable globale 3-5

variable locale 3-5

Vasarely 14-4

verger (le) 5-10

W

Warnsdorff 12-10

winsound 6-12

while 2-4

write() 7-2