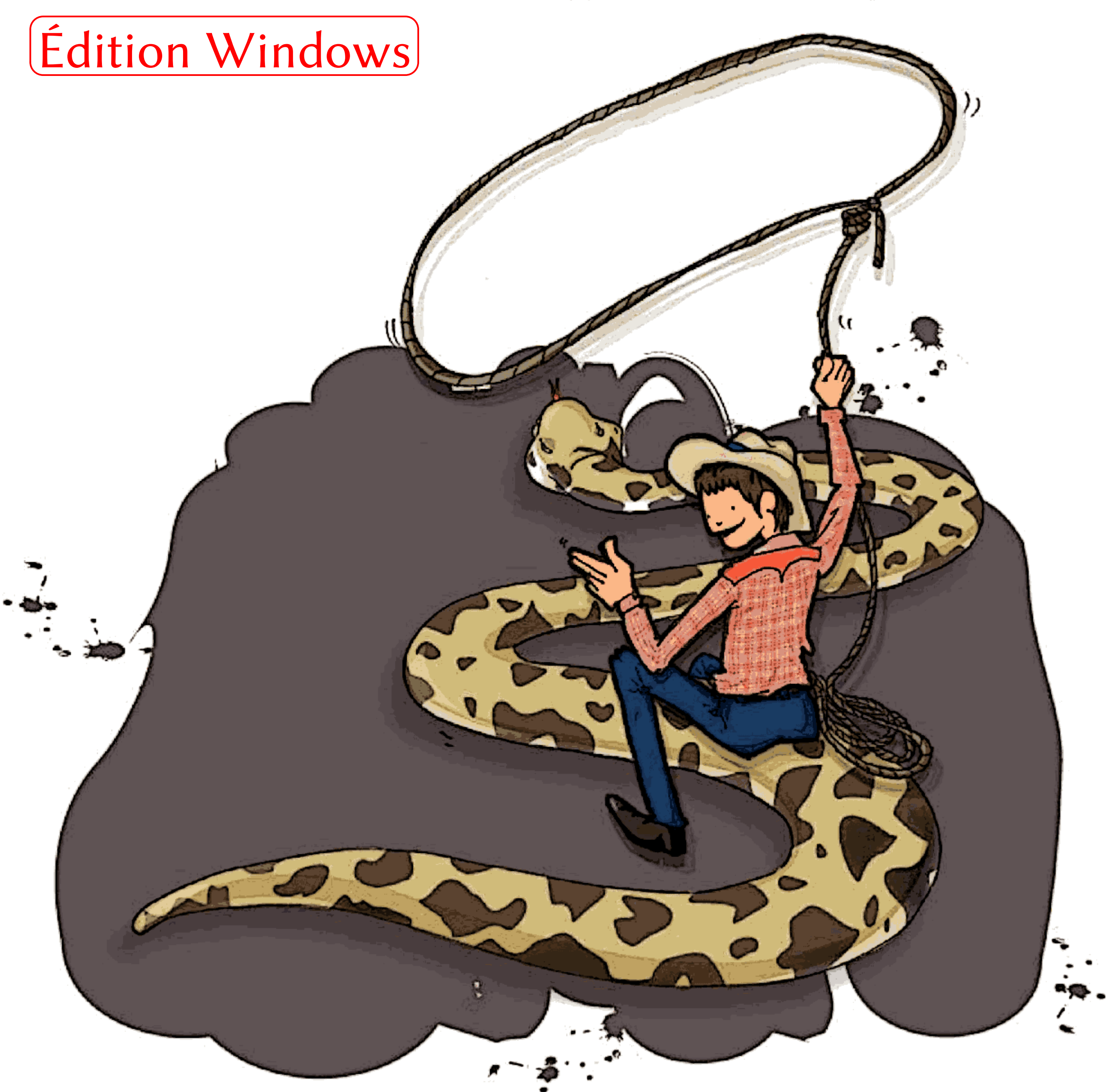
Domptage de serpent pour les enfants

Apprendre à programmer avec Python

Édition Windows



Écrit par Jason R. Briggs
Traduit et adapté par Michel Weinachter

Domptage de serpents pour les enfants, apprendre à programmer en Python
par Jason R. Briggs
traduit et adapté par Michel Weinachter

Version : 0.7.7
Version française : 0.0.9

Copyright © 2007-2009

Publié par... ah, personne en fait.

Remarques : livres@weinachter.com

L'ensemble des illustrations créées ou modifiées pour la traduction ont été faites en utilisant *the GIMP* et *Inkscape*. Illustration de couverture par Nuthapitol C., illustrations par Nuthapitol C. et Michel Weinachter, cliparts : `http://openclipart.org` et `http://commons.wikimedia.org`.

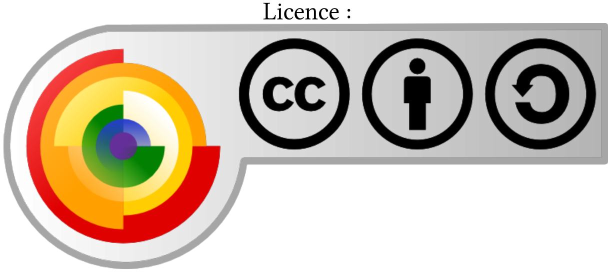
Édité avec TeXMaker majoritairement sous Gnu/Linux et quelquefois en utilisant Portable Keyboard Layout (avec une disposition fr-oss) sous Windows.
Composé avec XeTeX et XeLaTeX en utilisant les fontes *Linux Libertine*, Linux Biolinum, `Computer Modern`, DejaVu pour quelques symboles, et Firefly pour quatres caractères chinois 漢字.

Site web :
`http://www.briggs.net.nz/log/writing/snake-wrangling-for-kids`

Remerciements de l'auteur : merci à Guido van Rossum (pour la bienveillante dictature du langage Python), les membres de la liste de diffusion Edu-Sig (pour leurs avis et commentaires utiles) et à l'auteur David Brin (l'instigateur original de ce livre).

Remerciements du traducteur : merci à Jason R. Briggs, Gael Lickindorf, Anne, Christophe, Thaïs & Anne-Sophie.

Licence :



Cette Œuvre est licenciée selon les termes du Contrat Public Creative Commons : Paternité-Partage des Conditions Initiales à l'Identique 2.0 France Pour voir une copie de cette licence vous pouvez vous rendre à l'adresse suivante : `http://creativecommons.org/licenses/by-sa/2.0/fr/`.

Vous êtes libres :
- **de reproduire**, distribuer et communiquer cette création au public
- **de modifier** cette création

Selon les conditions suivantes :
- **Paternité**. Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).
- **Partage des Conditions Initiales à l'Identique**. Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.
  – À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers la page web : `http://creativecommons.org/licenses/by-sa/2.0/fr/`.
  – Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.
  – Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur : copies réservées à l'usage privé du copiste, courtes citations, parodie...).
Une version complète du Contrat est disponible sur :
`http://creativecommons.org/licenses/by-sa/2.0/fr/legalcode`.

Sommaire

Préface v

1 Tous les serpents ne vont pas vous mordre 1

2 8 multipliés par 3,57 égal… 11

3 Tortues et autres choses lentes 29

4 Comment poser une question 37

5 Encore et encore 47

6 Une sorte de recyclage 63

7 Un court chapitre à propos des fichiers 73

8 Tortues à profusion 83

9 Un peu de graphiques 103

10 Où aller à partir de ce point ? 129

A Réponses aux « À vous de jouer » 131

B Quelques fonctions intégrées 141

C Tous les mots clef de Python 3 151

D Quelques modules de Python 165

E Gâteaux 177

Index 179

Table des matières 181

Préface

Une note aux parents…

Chers parents et autres personnes attentionnées,

Vous vous demandez peut-être pourquoi apprendre à programmer ?

Apprendre à programmer permettra à votre enfant d'améliorer sa logique. Un ordinateur ne fait que ce qu'on lui a demandé. Si le programme ne fonctionne pas c'est que sa logique interne est mauvaise.

De plus, savoir comment fonctionnent les ordinateurs permettra à l'enfant de comprendre qu'ils ne fonctionnent pas grâce à de la poudre magique mais grâce à la magie du génie humain.

Vous vous demandez peut-être, pourquoi Python ?

Python est un langage simple mais pas simpliste. Les commandes Python ont des rôles indépendants : « il doit y avoir une manière évidente, de préférence une seule, de faire les choses ». Ces commandes sont donc en nombre limitées, ce qui permet de se concentrer sur la logique du programme et non pas sur les commandes à utiliser. Néanmoins Python est puissant, d'ailleurs des organismes comme l'INRIA ou la NASA utilisent Python. Il est utilisé par des gouvernements pour des infrastructures critiques. Les entreprises l'utilisent comme Google qui fournit d'ailleurs un environnement Python en ligne.

Python est un langage de haut niveau qui ne contient pas de concepts liés au matériel ou au système d'exploitation ce qui permet de réaliser des programmes simples sans se focaliser sur des éléments non directement productifs. Python est interactif, sa ligne de commande permet de réaliser des tests sans passer par des étapes complexes.

Par ailleurs, Python impose une écriture compréhensible car les différents blocs des programmes sont indiqués par les indentations du texte.

À propos de ce livre.

Ce livre existe en trois versions : Linux, Mac Os X et Windows. La version que vous avez en main est la version Windows. Si vous n'utilisez pas Windows, vous pouvez télécharger la version adaptée (dès qu'elle sera publiée) sur : `http://code.google.com/p/swfk-fr`.

Comment installer Python ?

De manière à ce que votre enfant puisse commencer à programmer, vous avez besoin d'installer Python sur votre ordinateur. Ce livre a été récemment mis à jour pour Python 3.1 ; cette version de Python est la plus récente et n'est pas compatible avec les versions antérieures. Si vous avez une version plus ancienne installée, vous devez télécharger la dernière version pour utiliser ce livre.

Installer Python est une tâche assez simple, mais il y a quelques différences selon le système d'exploitation que vous utilisez.

Si vous venez juste d'acheter un nouvel ordinateur, que vous n'avez pas idée de quoi faire avec et que les phrases précédentes vous ont remplit de frissons glacés, vous devriez sûrement trouver quelqu'un pour faire ça.

Selon votre ordinateur et la vitesse de votre connexion à Internet, cette installation devrait vous prendre entre quelques minutes et plusieurs heures.

Premièrement, allez sur `www.python.org` et téléchargez le dernier installateur pour Python 3.1. À la date de l'écriture de ce livre vous pouvez le trouver à l'adresse `http://www.python.org/ftp/python/3.1/python-3.1.msi`.

Double-cliquez sur l'icône de l'installateur de Python pour Windows (vous-vous rappelez où vous l'avez téléchargé, n'est-ce pas ?), et suivez les instructions pour l'installer à l'endroit par défaut (qui est probablement `C:\Python31` ou quelque chose de très similaire).

Après l'installation...

... Vous pourriez avoir besoin de vous asseoir à côté de votre enfant pour quelques premiers chapitres, mais heureusement après quelques exemples, il devrait chasser vos mains du clavier et faire les choses par lui-même. Il devrait, au moins, savoir comment utiliser un éditeur de texte quelconque avant de commencer (non, pas un traitement de texte comme Word, un vrai éditeur de texte à l'ancienne, sans gestion d'effets de style, comme le bloc-notes) il devrait au moins être capable d'ouvrir et de fermer des fichiers, de créer des fichiers « texte » et sauvegarder ce qu'il fait. Mis à part ça, ce livre va essayer de lui enseigner le *b+a, ba* à compter de cette page.

Merci pour votre temps, bien cordialement.

Le Livre

1

Tous les serpents ne vont pas vous mordre

1.1 Bonjour, je suis votre livre

Il y a des chances que vous [1] ayez reçu ce livre pour votre anniversaire. Tante Gertrude allait vous donner des chaussettes disparates qui auraient été deux tailles trop grandes (et que vous n'auriez pas porté plus grand de toute manière). À la place, elle a entendu quelqu'un parler de ce livre à imprimer, s'est rappelée que vous aviez un de ces ordinateurs-machin-chose, que vous aviez essayé de lui montrer comment l'utiliser au dernier Noël (vous aviez abandonné quand elle avait commencé à parler à la souris), et l'a fait imprimer.

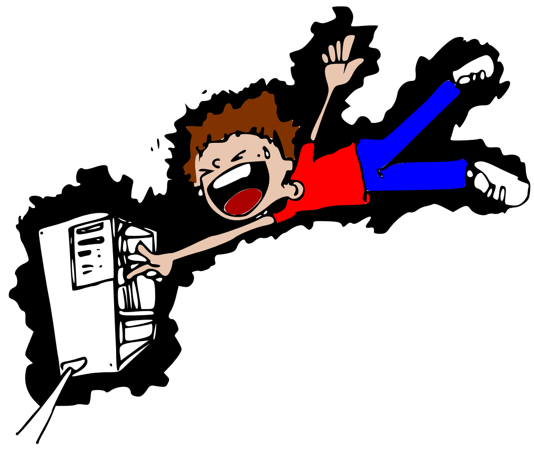
Soyez juste soulagés que vous n'ayez pas eu ces vieilles chaussettes moisies.

J'espère que vous n'avez pas été trop désappointé quand j'ai jailli à leur place du papier recyclé d'emballage. Un livre qui ne parle pas vraiment (O.K., qui ne parle pas du tout), avec un titre de mauvaise augure sur la couverture qui parle d'« apprendre… ». Mais prenez un moment pour penser à ce que je ressens. Si vous étiez un personnage d'un roman qui parle de magiciens et que j'étais dans la bibliothèque de votre chambre, j'aurais probablement des dents… ou peut-être des yeux verts. Je pourrais contenir des images animées ou être capable de faire des hurlements de fantôme quand vous ouvreriez mes pages. À la place, je suis imprimé sur des feuilles de papier A4 cornées, agrafées ensemble ou peut-être mises dans une chemise. Comme pourrais-je le savoir ? Je n'ai pas d'yeux.

Je donnerai n'importe quoi pour une belle mâchoire pleine de dents aiguisées…

Malgré tout n'est pas aussi catastrophique qu'il y parait. Même si je ne peux pas parler… Ou mordre vos doigts quand vous ne regardez pas… Je peux vous parler un petit peu de ce qui fait fonctionner les ordinateurs. Pas la partie physique, avec les fils, les puces, les câbles et les éléments qui pourraient plus que probablement vous électrocuter aussitôt que vous les toucheriez (donc ne le faites pas !) mais la partie cachée qui court à l'intérieur de ces fils, ces puces, ces câbles et ces machins qui font que les ordinateurs sont vraiment utiles.

1. Mon traducteur a fait le choix de traduire « you » par vous, le voussoiement lui semble plus adapté que le tutoiement généralisé dans certains livres pour enfants.



C'est un petit peu comme les pensées qui tournent dans votre tête. Si vous n'aviez pas de pensée, vous seriez assis sur le sol de votre chambre, regardant vaguement vers la porte et bavant sur l'avant de votre t-shirt. Sans programme, les ordinateurs seraient seulement utiles comme des cales-porte — et même comme cela ils ne seraient pas très utiles parce que vous devriez prendre garde à bien les enjamber pendant la nuit. Et il n'y a rien de pire que de se cogner un orteil dans l'obscurité.

Je suis juste un livre et même moi je sais ça.

Votre famille a peut-être une Playstation, une Xbox ou une Wii qui trônent dans le salon. Elles ne seraient que d'une faible utilité sans programme (jeu) pour les faire fonctionner. Les lecteurs de DVD, certains réfrigérateurs et même la plupart des voitures, ont tous des programmes informatiques qui les rendent plus utiles qu'ils ne pourraient l'être autrement. Votre lecteur de DVD utilise des programmes pour jouer des disques ; votre réfrigérateur peut disposer d'un programme simple pour s'assurer qu'il n'utilise pas trop d'électricité, mais continue de garder les aliments froids ; certaines voitures peuvent avoir un ordinateur avec un programme pour avertir si elles risquent d'avoir un accident.

Si vous savez comment écrire un programme informatique, vous pouvez faire toutes sortes de choses utiles, comme programmer vos propres jeux ou créer vos propres pages web qui peuvent réagir au lieu d'un bête machin coloré. Être capable de programmer pourrait peut-être même vous aider dans votre travail scolaire.

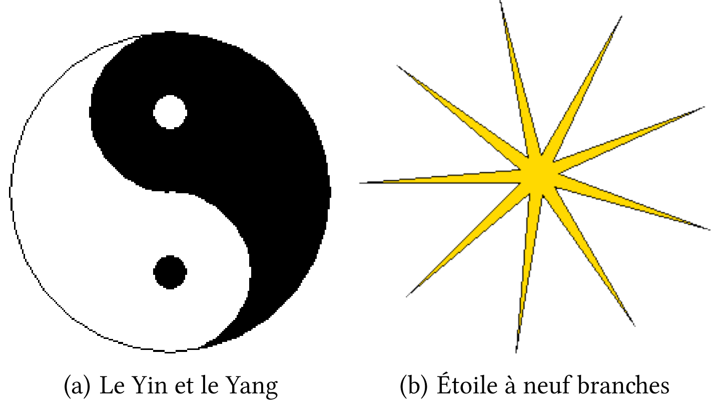
Cela dit, passons maintenant à un sujet un peu plus intéressant.

1.2 Qu'est ce que nous allons voir ?

Nous allons voir comment poser des questions et prendre en compte la réponse : par exemple demander à quelqu'un son nom et lui faire des remarques sur celui-ci.

Nous allons aussi voir comment faire faire des dessins comme sur la Figure 1.1.

Nous verrons aussi comment afficher un message de bienvenue différent tous les jours.



(a) Le Yin et le Yang (b) Étoile à neuf branches

FIGURE 1.1 – Des dessins faits avec Python

1.3 Quelques mots à propos des langages

Comme les humains, certainement les baleines, possiblement les dauphins et peut-être même les parents (même si cela fait débat pour ces derniers), les ordinateurs ont leur propre langage. En fait, aussi comme les humains, ils ont plus d'un langage. A, B, C, D et E ne sont pas juste des lettres, ce sont aussi des langages de programmation (ce qui prouve que les adultes n'ont pas d'imagination et devraient lire soit un dictionnaire soit un répertoire avant de nommer quoique ce soit). Oh et si vous ajoutez quelques plus ou dièses (+, #) après certaines de ces lettres que je viens de lister, il s'agit encore de langages de programmations qui sont presque les mêmes et n'en diffèrent que légèrement.

D'autres sont nommés d'après des personnes, en utilisant de simples acronymes (les premières lettres d'une série de mots), voir pour certains à partir d'une émission de télévision.

Qu'est-ce que je vous avez dit ? Pas d'imagination !

Par chance, de nombreux langages sont tombés en désuétude ou ont disparu complètement ; mais la liste des différentes manières de « parler » à un ordinateur reste vraiment ennuyeusement longue. Je vais seulement parler d'un de ceux-ci — sinon nous ferions aussi bien de ne pas commencer.

Il serait alors plus productif de vous asseoir dans votre chambre et de baver sur le devant de votre t-shirt.

1.4 L'ordre des serpents constricteurs non-venimeux...

... ou pythons, pour faire plus court.

Le python est un serpent mais le Python est un langage de programmation. Néanmoins, son nom ne vient pas directement du reptile sans patte ; il s'agit d'un de ces rares langages de programmations nommés d'après une émission de télévision. Le Monty Python's Flying Circus était une émission humoristique britannique populaire durant les années 1970 (et en fait, encore vraiment populaire actuellement) qui nécessite d'avoir un certain âge pour la

trouver amusante. Toute personne en dessous de… disons douze ans… pourrait se demander d'où peut provenir tout ce battage [2].

Il y a un certain nombre de choses à propos de Python (le langage de programmation, pas le serpent ou l'émission de télévision) qui le rende extrêmement pratique quand vous allez apprendre à programmer. Pour nous, en ce moment l'important est que vous puissiez commencer et faire des choses vraiment rapidement.

C'est maintenant la partie où vous espérez que Maman, Papa ou quiconque est en charge de l'ordinateur, a lu la partie au début de ce livre nommée « Une note aux parents… ».

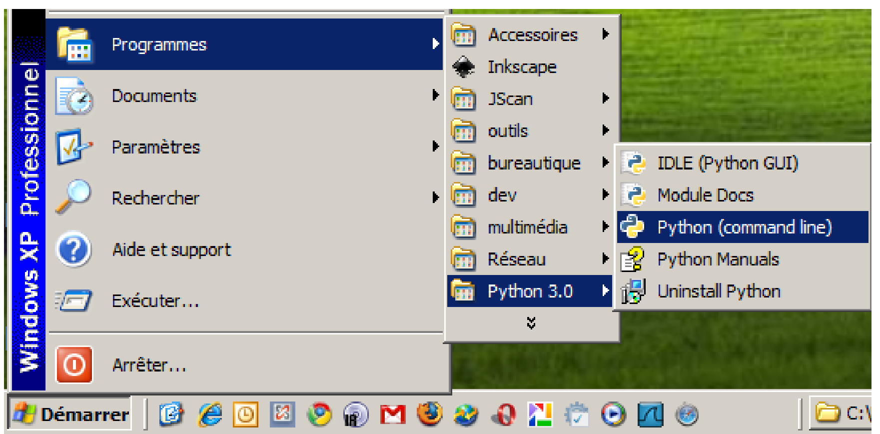


FIGURE 1.2 – Python dans le menu de Windows.

Il y a une bonne manière de vérifier s'ils l'ont réellement lu : cliquez sur le bouton « démarrer » en bas à gauche de l'écran, cliquez sur « Programmes » (qui doit avoir un petit triangle à coté), et avec optimisme dans la liste des programmes vous devriez voir « Python 3.1 » (ou quelque chose approchant).

La figure 1.2 montre ce que vous devriez voir. Cliquez sur « Python (command line) » et vous devriez voir quelque chose comme sur la figure 1.3.

```
Python (command line)
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

FIGURE 1.3 – Ligne de commande Python sous Windows.

2. Mis à part « the fish slapping dance » ou danse tape-poisson qui est drôle quelque soit votre âge.

La ligne de commande Python fonctionne sous Windows à partir de la ligne de commande Windows et hérite de certaines limitations en particulier pour l'affichage de certains caractères (typiquement les caractères français comme « œ » ou « € » qui seront remplacés par des « ? ») sans que cela ne pose de problème important.

Cliquez sur « IDLE (Python GUI) » et vous devriez voir quelque chose comme sur la figure 1.4.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
Ln: 3 Col: 4
```

FIGURE 1.4 – Shell Python sous Windows.

Le shell [3] Python est plus puissant et plus convivial : en particulier il met en valeur en les coloriant les éléments clef des commandes qui sont passées ce qui limite le risque d'erreur.

Si vous découvrez qu'ils n'ont pas lu la section du début…

… parce que s'il manque quelque chose quand vous essayez de suivre ces instructions — alors retournez au début du livre et brandissez le sous leur nez alors qu'ils essayent de lire le journal, et soyez plein d'optimisme. Si vous avez du mal à les convaincre de s'extraire du canapé, dire : « s'il-vous-plait, s'il-vous-plait, s'il-vous-plait… » encore et encore jusqu'à ce que cela devienne ennuyant devrait vraiment bien fonctionner. Bien sûr, l'autre chose que vous pouvez faire est de retourner au début du livre et suivre les instructions de la préface pour installer Python vous même.

1.5 Votre premier programme en Python

Avec de la chance, si vous avez atteint ce point, vous avez réussi à démarrer la console Python, qui est un moyen de lancer des commandes Python et des programmes. Quand vous

3. Le mot *shell* signifie en anglais coquille, c'est le nom donné à l'interface d'un système d'exploration, c'est à dire la partie que manipule l'utilisateur pour utiliser un système d'exploitation. Par extension, ce nom est donné à certaines autres interfaces.

lancez la console (ou après avoir entré une commande), ce que vous voyez en premier est appelé une « invite de commande » (invite pour faire court ou « *prompt* » en anglais). Dans la console Python, l'invite est matérialisée par trois chevrons, ou trois symboles plus-grand-que (>) pointant vers la droite :

ne pas saisir
```
>>>
```

Si vous placez suffisamment de commandes Python ensembles, vous avez un programme que vous pouvez lancer même en dehors de la console... Mais pour le moment nous allons garder les choses simples et taper nos commandes directement dans la console, à l'invite (>>>). Ainsi pourquoi ne pas commencer à taper ce qui suit [4] :

à taper
```
print("Bonjour␣le␣monde␣!")
```

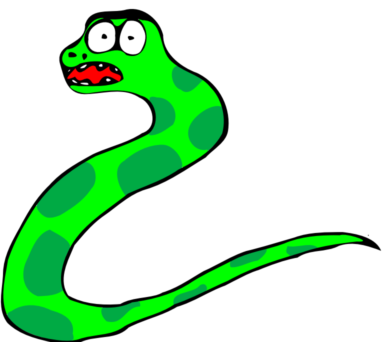
Note du traducteur : la plupart des langages de programmation utilisent des mots clefs en anglais. Python ne fait pas exception à la règle. Vous avez donc l'occasion d'apprendre quelques mots d'anglais en plus d'un langage de programmation. Le verbe « *(to) print* » signifie en anglais « imprimer » ou « afficher » du moyen anglais *prente* issu du participe passé *preint, prient* du verbe *priendre*, du latin *premere* qui signifie presser. Par la suite les explications et traductions des mots anglais seront mises en note de bas de page.

Pour information les extraits de code sont mis en valeur dans des cadres colorés. Les cadres bleus contiennent des exemples que vous pouvez taper sans difficulté particulière :

à taper
```

```

Python est résistant : les erreurs que vous pourriez obtenir sont sans gravité, vous pouvez expérimenter sans peur. Il vous sera même indiqué parfois des tests à réaliser pour le pousser à la faute. Python couinera un peu, mais ce n'est pas grave.



Les cadres rouges contiennent des erreurs, si vous les tapez vous obtiendrez le message d'erreur indiqué :

erreur
```

```

4. Le ␣ dans les codes symbolise *une* espace, c'est à dire le caractère issu de l'appui sur la barre d'espace. Quand cela est utile, ce symbole sera affiché.

Les cadres verts contiennent des exemples valides mais avec des subtilités qui viennent d'être expliquées ou qui vont l'être prochainement, si vous les tapez en oubliant des espaces (symbolisée par ␣) vous pouvez obtenir des messages d'erreur :

`à taper avec attention`

```
```

Les cadres gris contiennent des exemples qui ne sont pas à taper :

`ne pas saisir`

```
```

Assurez-vous d'inclure les guillemets (ceux qui sont comme cela `"  "`), et d'appuyer sur entrée à la fin de la ligne. Normalement vous devriez avoir quelque chose comme :

`à taper si cela n'est pas déjà fait`

```
>>>␣print("Bonjour␣le␣monde␣!")
Bonjour␣le␣monde
```

L'invite réapparaît pour vous faire savoir que la console Python est prête à accepter plus de commandes.

Félicitations ! Vous venez juste de créer votre premier programme en Python. La commande « `print` » est une fonction qui écrit tout ce qui est entre les parenthèses sur la sortie de la console, nous la verrons plus en détail ultérieurement.

1.6 Votre second programme Python… Le même à nouveau ?

Les programmes Python ne seraient pas très utiles si vous aviez à taper les commandes chaque fois que vous voulez faire quelque chose, ou si vous écriviez un programme pour quelqu'un et qu'il avait à le taper avant qu'il ne puisse l'utiliser.

Le traitement de texte, que vous pouvez utiliser pour faire des dossiers pour l'école, est probablement constitué de dix à cent millions de lignes de code. Selon le nombre de lignes imprimées sur une page et même si vous imprimez sur les deux faces, cela représente quatre cents mille pages imprimées, ou une pile de quarante mètres de haut. Imaginez-vous juste quand vous ramenez un tel logiciel à la maison depuis le magasin, il faudrait plus que quelques voyages pour ramener autant de papier. Et vous feriez mieux d'espérer qu'il n'y ait pas une rafale de vent quand vous ramenez les piles de papier.

Heureusement il y a une alternative à toute ces saisies de texte [5] ou rien ne serait jamais fini. Ouvrez le « Bloc-notes » (cliquez sur « Démarrer », « Programmes » et vous devriez le trouver dans le sous menu « Accessoires ») et tapez la commande exactement comme vous l'aviez tapée dans la console auparavant :

ne pas saisir
```
print("Bonjour le monde !")
```

Cliquez sur le menu « Fichier » (dans le Bloc-notes) puis sur « Enregistrer », il vous sera alors demandé de choisir un nom de fichier, entrez « `bonjour.py` » et sauvez le fichier sur le Bureau. Pour ce faire cliquez sur l'icône « `Bureau` », changez le type de « `Fichiers texte (*.txt) en « Tous les fichiers »` », puis changer le « `*.txt` » en « `bonjour.py` ». Double cliquez sur l'icône bonjour.py sur votre bureau (voir figure 1.5 et pendant un bref instant une fenêtre noire va apparaître.

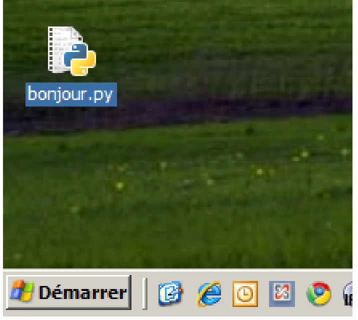


FIGURE 1.5 – Programme Python sur le bureau.

Elle va disparaître trop rapidement pour que vous puissiez la lire mais « Bonjour le monde ! » est apparu dans la fenêtre une fraction de seconde ; nous y reviendrons plus tard et nous montrerons que le texte s'y est bien imprimé.

5. On parle aussi de dactylographie, écriture avec les doigts.

Ainsi, nous venons de voir que les sympathiques personnes qui ont créé Python, nous ont gentiment prémuni d'avoir à taper la même chose encore et encore et encore et encore... Comme le faisait certains dans les années 1980. Non, je suis sérieux, certains l'ont fait. Allez demander à votre maman ou votre papa s'ils n'ont pas possédé un MO5 ou un Amstrad CPC [6] quand ils étaient plus jeunes. S'ils en ont eu un vous pouvez les montrer du doigt et rigoler.

Croyez moi sur ce point, vous ne comprendrez peut-être pas mais, eux, ils comprendront.

Soyez néanmoins prêts à courir.

Mon cousin, le livre en version anglaise, parle de ZX81 [7] un ordinateur qui a été très populaire dans les pays anglo-saxons.

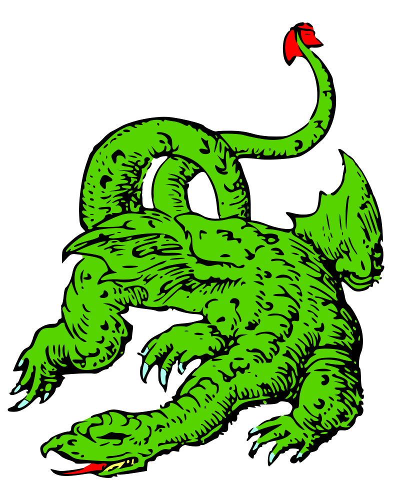
La fin du début

Bienvenue dans le monde merveilleux de la programmation. Nous avons commencé vraiment simplement avec une application « Bonjour le monde » [8]. tout le monde commence avec cela quand on apprend à programmer. Dans le prochain chapitre nous commencerons à faire des choses plus utiles avec la console Python et alors nous regarderons comment faire un programme.

6. Ces ordinateurs étaient populaires dans les années 1980 dans les pays francophones.

7. Le Sinclair ZX81, vendu dans les années 1980 a été un des premiers ordinateurs vraiment abordable. Un grand nombre de jeunes garçons et de jeunes filles ont été complètement passionnés et tapaient les codes de jeux imprimés dans des magazines populaires consacrés au ZX81 pour découvrir après des heures de dactylographie que ces saletés ne fonctionnaient jamais correctement.

8. « Hello world » en anglais



2

8 multipliés par 3,57 égal...

2.1 Les calculs en Python

Combien font 8 multipliés par 3,57 ? Vous pouvez utiliser une calculatrice, n'est-ce pas ? Ou bien vous êtes peut-être extrêmement intelligent et vous pouvez faire la multiplication de tête mais ce n'est pas l'objet. Vous pouvez aussi faire la même chose avec la console Python. Démarrez la console à nouveau (voir le chapitre 1 pour plus d'informations, si vous avez sauté le début pour quelque étrange raison) et une fois que vous voyez l'invite de commande, tapez `8*3.57` et appuyez sur entrée [1].

à taper

```
>>> 8*3.57
28.559999999999999
```

L'étoile « * », ou touche astérisque, est utilisée pour les multiplications, à la place du symbole multiplier normal « x » que vous utilisez à l'école. Utiliser la touche étoile est nécessaire, autrement comment les ordinateurs sauraient si vous voulez dire la lettre « x » ou le symbole de multiplication « x » ? Pour une équation n'est-ce pas légèrement utile ?

1. Les anglo-saxons utilisent le point « . » comme séparateur décimal contrairement aux francophones.

Python est cassé !

Si vous avez juste pris une calculatrice et entré `8x3.57` la réponse qui devrait être affichée est la suivante : `28.56`.

Pourquoi Python est différent ? Est-il cassé ?

En fait non, la raison peut-être trouvée dans la manière dont les nombres à virgule flottante (nombres rationnels) sont gérés par l'ordinateur. Il s'agit d'un problème compliqué pour les débutants [a]. Le mieux, pour le moment, est juste de vous rappeler que lorsque vous travaillez avec des fractions (ou des nombres avec une virgule) quelques fois le résultat n'est pas précisément ce que vous attendiez. Cela est vrai pour les multiplications, les divisions, les additions et les soustractions [b].
Le module [c] « `decimal` » peut être utilisé pour obtenir un comportement plus conforme à vos attentes. Par exemple, le code suivant permet obtenir le bon résultat :

```
>>>import decimal
>>> 8*decimal.Decimal('3.57')
Decimal('28.56')
```

Vous trouverez plus de renseignements, en anglais, à l'adresse suivante :
`http://docs.python.org/3.1/library/decimal.html`.

a. Le problème provient du fait que l'ordinateur ne gère pas des fractions mais justement des nombres à virgules avec une quantité finie de chiffres.
b. Il est bien sûr possible d'implémenter en Python des procédés semblables à ceux des calculatrices comme dans calculext : `http://python.jpvweb.com/mesrecettespython/calculext`.
c. La section 6.2 explique l'utilisation des modules.

Supposons que vous faites le ménage une fois par semaine, pour cela vous recevez 5€ d'argent de poche, et vous livrez les journaux cinq fois par semaine et avez ainsi 30€ de plus par semaine. Combien gagnerez-vous par an ?

Si nous l'écrivons sur le papier, nous devrons écrire quelque chose comme `(5€+30€)x52` ce qui corresponde à 5€ plus 30€ multiplié par 52 semaines d'une année. Bien sûr nous sommes intelligents et nous savons que cinq plus trente font trente cinq et que l'équation se simplifie en `35€x52` ce qui peut facilement être calculé à la main.

Mais nous pouvons faire tous ces calculs dans la console tout aussi bien :

à taper
```
>>> (5 + 30) * 52
1820
>>> 35 * 52
1820
```

Mais que ce passe-t-il si nous dépensons 10€ par semaine ? Combien nous restera-t-il à la fin de l'année ? Nous pouvons écrire l'équation à nouveau sur le papier quelques différentes manières, mais nous allons juste taper dans la console :

à taper
```
>>> (5 + 30 - 10) * 52
1300
```

Ce qui correspond à 5€ plus 30€ moins 10€ multipliés par les 52 semaines de l'année. Il vous reste ainsi 1300€ à la fin de l'année. Bon d'accord, ce que nous avons fait n'est pas si utile que cela jusqu'à maintenant. Nous pouvons faire tout cela avec une calculatrice. Mais nous y reviendrons plus tard et verrons comment rendre cela plus utile.

Vous pouvez faire des multiplications, des additions (évidemment), des soustractions et des divisions dans la console Python ; ainsi qu'un tas d'autres opérations mathématiques que nous n'allons pas décrire plus avant maintenant. Pour le moment les symboles des opérations mathématiques simples (en fait appelés opérateurs) sont visibles dans le Tableau 2.1 :

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division

Table 2.1 – Opérateurs.

La raison pour laquelle la barre oblique « / » est utilisée pour les divisions est qu'il serait assez difficile de dessiner une ligne de fraction ; en plus les concepteurs de clavier n'ont pas jugé utile de mettre une touche pour le caractère division « ÷ » comme celui que nous sommes supposés utilisés pour écrire les équations. Par exemple, si vous avez 100 œufs et vingt boites, vous pouvez avoir envie de savoir combien d'œufs iront dans chaque boite. Vous diviserez alors 100 par 20, en écrivant l'équation suivante :

$$\frac{100}{20}$$

Ou si vous savez poser une division :

$$\begin{array}{r|l} 100 & 20 \\ \cline{2-2} -100 & 5 \\ \cline{1-1} 0 & \end{array}$$

Ou encore :

$$100 \div 20$$

Néanmoins en langage Python vous pourrez juste le taper comme « `100/20` ».

Ce qui est vraiment plus simple, je pense. Mais encore, je suis un livre — qu'est-ce que j'en sais ?

2.2 L'usage des parenthèses et « l'ordre des opérations »

Nous utilisons les parenthèses dans les langages de programmation pour contrôler ce qui est appelé « l'ordre des opérations ». Une opération consiste en l'usage d'un opérateur (un de ces symboles dans le Tableau 2.1 ci-avant). Il y a plus d'opérateurs que ces symboles simples (addition, soustraction, multiplication et division). Il suffit de savoir pour le moment que les multiplications et les divisions ont toutes deux une précédence (priorité) plus élevé que l'addition et la soustraction. Ce qui signifie que quand il y a une multiplication ou une division qui font partie d'une équation vous devez les faire avant les additions et les soustractions qui en font partie.

Dans l'équation suivante, toutes les opérateurs sont des additions « + » les nombres sont ajoutés dans l'ordre :

à taper

```
>>> print(5 + 30 + 20)
55
```

Similairement, dans cette équation, il y a seulement des opérateurs d'addition et de soustraction, de nouveau Python prend en compte chaque nombre et opération dans l'ordre d'apparition [2] :

à taper

```
>>> print(5 + 30 - 20)
15
```

Mais dans l'équation suivante, il y a un opérateur de multiplication donc les nombres 30 et 20 doivent pris en compte en premier. Cette équation est une autre manière de dire : « multiplier 30 et 20 puis ajouter 5 au résultat », la multiplication est effectuée en premier car elle a une précédence plus élevée que l'addition :

à taper

```
>>> print(5+30*20)
605
```

Qu'est-ce qui arrive si nous ajoutons des parenthèses ? L'équation suivante donne le résultat :

à taper

```
>>> print((5+30)*20)
700
```

Pourquoi le résultat est-il différent ? Parce que les parenthèses contrôlent l'ordre des opérations. Avec les parenthèses, Python sait comment calculer en utilisant les opérations à l'intérieur des parenthèses en premier puis les opérations à l'extérieur. Ainsi cette équation est une autre manière de dire : « ajouter 5 et 30 puis multiplier par 20 ». L'usage des parenthèses peut devenir plus compliqué. Il peut y avoir des parenthèses à l'intérieur d'autres parenthèses :

2. Python s'écrit de la gauche vers la droite comme les écritures latines et cyrilliques

à taper
```
>>> print(((5+30)*20)/10)
70.0
```

Dans ce cas, Python évalue les parenthèses les plus à *l'intérieur* puis celles à l'extérieur et enfin les autres opérations. Ainsi cette équation est une autre manière de dire : « ajouter 5 et 30 puis multiplier le résultat par 20 finalement diviser le résultat par 10 » [3]. Le résultat sans parenthèse est à nouveau légèrement différent :

à taper
```
>>> 5+30*20/10
65.0
```

Dans ce cas 30 est multiplié par 20 en premier puis le résultat est divisé par 10 finalement 5 est ajouté au dernier résultat.

Rappelez-vous que les multiplications et les divisions sont toujours effectuées avant les additions et les soustractions sauf si des parenthèses sont utilisées pour contrôler l'ordre des opérations.

2.3 Il n'y a rien d'aussi inconstant qu'une variable

Une « variable » est un terme utilisé en programmation pour décrire un endroit où entreposer des choses. Ces « choses » peuvent être des nombres, des textes, des listes de nombres et de textes ou toutes sortes d'objets trop nombreux pour être listés ici. Pour le moment, nous allons juste nous représenter une variable comme quelque chose qui ressemble un peu à une boite aux lettres.

Vous pouvez mettre des lettres ou des colis dans une boite aux lettres, de la même manière vous pouvez mettre des objets (nombres, textes, listes, *etc.*) dans une variable. Cette

3. Il convient de noter que le résultat est `70.0` au lieu de 70 en effet Python 3 considère le résultat de toute division comme un nombre rationnel.

idée de boite aux lettres est une des nombreuses manières dont de nombreux langages de programmation fonctionnent, mais pas tous. En Python, les variables sont légèrement différentes. Plutôt que d'être une boite aux lettres avec des choses à l'intérieur, une variable est plus comme une étiquette qui est collée sur l'extérieur de la boite aux lettres. Nous pouvons mettre l'étiquette sur un objet différent ou même attacher l'étiquette (peut-être avec un bout de ficelle) à plus d'un objet mais pas en même temps. Nous créons une variable en lui donnant un nom en utilisant le signe « = » puis nous disons à Python sur quoi vous voulons faire pointer ce nom. Par exemple :

à taper
```
>>> fred = 100
```

Nous venons à l'instant de créer une variable appelée « `fred` » et dit qu'elle pointait vers un nombre valant 100. C'est un peu comme dire à Python de se souvenir de ce nombre car nous voulons nous en servir plus tard. Pour trouver vers quoi une variable pointe, nous pouvons taper `print` dans la console suivi du nom de la variable entre parenthèses puis appuyer sur la touche « Entrée ». Par exemple :

à taper
```
>>> fred = 100
>>> print(fred)
100
```

Nous pouvons maintenant dire à Python que la variable « `fred` » doit pointer vers quelque chose d'autre :

à taper
```
>>> fred = 200
>>> print(fred)
200
```

Sur la première ligne nous avons dit que nous voulions que fred pointe maintenant vers le nombre 200. Puis sur la seconde ligne, nous avons demandé sur quoi fred était en train de pointer pour prouver que fred a changé. Nous pouvons faire pointer plus d'un nom vers le même objet :

à taper
```
>>> fred = 200
>>> jean = fred
>>> print(jean)
200
```

Dans le code ci-dessus, nous disons que nous voulons que le nom (ou l'étiquette) « `jean` » de pointer sur la même chose que fred. Bien sûr « `fred` » n'est pas un nom vraiment utile pour une variable. Ce nom ne nous dit pas à quoi cette variable va être utilisée. C'est plus simple pour une boite aux lettres, vous l'utilisez pour le courrier. Mais une variable peut avoir un grand nombre d'usages différents et nous pouvons pointer vers un tas d'objets divers, ainsi nous voulons généralement quelque chose un peu plus informatif comme nom.

Supposons que vous lanciez la console Python, tapiez « `fred=200` » puis alliez passer au loin dix ans à escalader le Mont Everest, traverser le désert du Sahara, faire du saut à l'élastique depuis un pont en Nouvelle-Zélande et finalement naviguer jusqu'au fleuve Amazone. Quand vous reviendrez à votre ordinateur [4], vous rappellerez vous ce que le nombre 200 signifie (et à quoi correspondait-il) ?

Je ne pense pas y arriver.

Je viens juste de le faire à l'instant et je n'ai déjà plus la moindre idée de ce « `fred=200` » signifie (mis à part que ce nom pointe vers le nombre 200). Donc après dix ans, vous n'aurez aucune chance de vous en souvenir.

Aha ! Mais que ce passerait-il si nous avions appelé notre variable : nombre_d_étudiants ?

à taper

```
>>> nombre_d_étudiants=200
```

Nous pouvons le faire car les noms de variables peuvent être faits de lettres, de nombres et de tirets bas « _ » ; même si nous ne pouvons pas commencer avec un nombre [5]. Si vous revenez dans dix ans, « nombre_d_étudiants » continue d'avoir une signification. Vous pouvez taper :

à taper

```
>>> print(nombre_d_étudiants)
200
```

Et vous saurez immédiatement que vous parliez de 200 étudiants. Ce n'est pas toujours important d'utiliser des noms de variables signifiants. Vous pouvez utiliser n'importe quoi depuis des lettres solitaires (comme « `a` ») jusqu'à de longues phrases ; des fois, si vous faites quelque chose de rapide, un nom simple et court est tout aussi utile. Cela dépend vraiment si vous voulez être capable de retrouver le nom de la variable plus tard et savoir ce que vous avez bien pu penser au moment où vous l'avez saisi.

```
ceci_est_aussi_un_nom_de_variable_valide_mais_peut_être_pas_très_utile
```

2.4 Utilisation des variables

Maintenant que nous savons comment créer une variable, comment l'utilisons nous ? Vous rappelez-vous cette équation que nous avons vue plus tôt ? Celle à propos du travail et de combien d'argent nous avions à la fin de l'année, si vous aviez gagné 5€ par semaine à faire le ménage, 30€ par semaine à livrer les journaux et dépensé 10€ par semaine. Nous avions alors fait :

4. S'il fonctionne encore !

5. Toutes les lettres peuvent être utilisées néanmoins gardez à l'esprit que dans un projet partagé (sur Internet ou en entreprise) il est généralement conseillé d'utiliser des lettres de l'alphabet latin sans accent.

à taper
```
>>> print((5 + 30 - 10) * 52)
1300
```

Que ce passerait-il si nous transformions les trois premiers nombres en variables ? Essayez de taper ce qui suit :

à taper
```
>>> ménage=5
>>> livraison_journal=30
>>> dépenses=10
```

Nous venons juste de créer des variables nommées « ménage », « livraison_journal » et « dépenses ». Nous pouvons alors retaper l'équation pour avoir :

à taper
```
>>> print((ménage+livraison_journal-dépenses)*52)
1300
```

Ce qui nous donne exactement la même réponse. Que ce passerait-il si vous gagnez 2€ de plus par semaine en faisant plus de ménage ? Changez la valeur de la variable « ménage » pour qu'elle vaille 7 puis tapez sur la flèche du haut « ↑ » (si vous utilisez la console [6]) ou sur « `Alt-p` » (si vous utilisez le shell [7]) sur votre clavier assez de fois pour que l'équation réapparaisse, enfin appuyez sur la touche entrée :

à taper
```
>>> ménage=7
>>> print((ménage+livraison_journal-dépenses)*52)
1404
```

C'est nettement moins d'écriture pour trouver maintenant que vous finirez avec 1404€ de plus à la fin de l'année. Vous pouvez essayer de changer les autres variables, puis appuyer sur la flèche du haut pour réaliser le calcul à nouveau, et voir quels effets cela a.

Si vous dépensez deux fois plus par semaine :

à taper
```
>>> dépenses=20
>>> print((ménage+livraison_journal-dépenses)*52)
884
```

Il ne vous restera que 884€ d'économies à la fin de l'année. Cela est néanmoins marginalement utile. Nous n'avons pas encore atteint ce qui est réellement utile. Mais pour le moment il suffit de savoir que les variables sont utilisées pour stocker des choses.

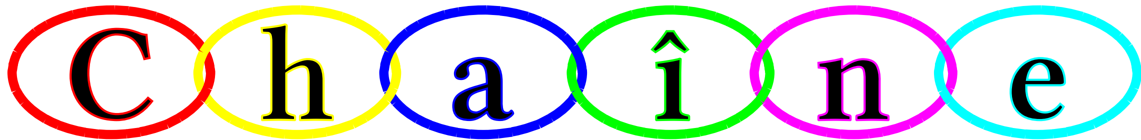
Pensez à une boite aux lettres avec une étiquette dessus !

6. Les flèches vers le haut et vers le bas peuvent être utilisées dans la console pour naviguer dans l'historique des commandes de la console.

7. Vous pouvez utiliser Alt+p (précédent) ou Alt+n (nouveau) pour naviguer dans l'historique du shell.

2.5 Un maillon de la chaîne ?

Si vous êtes attentif et que vous n'êtes pas qu'à la recherche de quelques bons mots, vous devez vous rappeler que j'ai mentionné que les variables peuvent être utilisées pour toutes sortes de choses et pas seulement des nombres. En programmation, la plupart du temps, nous appelons les textes des « chaînes ». Cela peut sembler un peu étrange mais vous pouvez réfléchir que les textes sont des tas de lettres « enchaînées » (ou jointes). Chaque caractère est un peu un maillon de la chaîne. Peut-être cela fait-il plus sens ?



Mais peut-être que non, finalement.

Dans ce cas tout ce que vous avez besoin de savoir est que les chaînes sont juste un tas de lettres, de nombres et des symboles. Votre nom peut être une chaîne. Comme votre adresse. Le premier programme Python que nous avions créé au Chapitre 1 utilisait une chaîne : « Bonjour le monde ! ».

En python nous créons une chaîne en plaçant des guillemets « " » [8] autour du texte. Ainsi nous pouvons reprendre notre variable fred inutile et la faire pointer vers une chaîne comme cela :

à taper
```
>>> fred = "Ceci est une chaîne."
```

Et nous pouvons voir vers quoi pointe la variable fred en tapant print(fred) :

à taper
```
>>> print(fred)
Ceci est une chaîne.
```

Nous pouvons aussi utiliser des apostrophes « ' » pour créer une chaîne :

à taper
```
>>> fred = 'Cela est une autre chaîne.'
>>> print(fred)
Cela est une autre chaîne.
```

Néanmoins, si vous essayez de taper plus d'une ligne de texte et que votre chaîne utilise des apostrophes « ' » ou des guillemets « " », vous aurez un message d'erreur dans la console similaire au message suivant :

erreur
```
>>> fred = "Voici deux
  File "<stdin>", line 1
    fred = "Voici deux
                     ^
SyntaxError: EOL while scanning string literal
```

8. Les guillemets utilisés en programmation sont ceux du clavier. C'est à dire ni les guillemets français «», ni anglais “”, ni allemands „“ !

Nous parlerons des erreurs plus tard mais pour le moment si vous avez plus d'une ligne de texte, vous pouvez utiliser trois apostrophes ou trois guillemets « " » :

à taper
```
>>> fred = '''Voici deux
... lignes de texte dans une seule chaîne.'''
```

Affichons le contenu pour voir si cela a bien fonctionné :

à taper
```
>>> print(fred)
Voici deux
lignes de texte dans une seule chaîne.
```

Au passage, nous pouvons voir trois points (...) après avoir tapé quelque chose qui continue sur une autre ligne (comme les chaînes multilignes). En fait, vous en verrez bien d'autres par la suite.

2.6 Tours de chaînes

Il y a une question intéressante : que vaut dix fois cinq, « 10*5 » ? La réponse est, bien sûr, cinquante.

Bon d'accord, ce n'est pas intéressant du tout.

Mais que vaut dix fois « a » (10*'a') ? Cela peut paraître une question sans queue ni tête, mais il y a une réponse dans le monde de Python :

à taper
```
>>> print(10 * "a")
aaaaaaaaaa
```

Cela fontionne aussi avec des chaînes de plus d'un caractère :

à taper
```
>>> print(10 * "abcd")
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

Une autre astuce avec une chaîne consiste à utiliser des *valeurs embarquées*. Vous pouvez faire cela avec « %s » qui est comme une marque (ou un paramètre subsituable) pour une valeur que vous voulez inclure dans une chaîne. C'est plus simple à expliquer avec un exemple :

à taper
```
>>> montexte = "J'ai %s ans."
>>> print(montexte % 12)
J'ai 12 ans.
```

Dans la première ligne, la variable « montexte » est créée pointant sur une chaîne qui contient des mots et un paramètre substituable « %s » qui est une petite balise qui dit à la console Python : « remplace moi avec quelque chose ». Ainsi au niveau de la ligne suivant, quand nous appelons « print(montexte % 12) » nous utilisons le symbole « % » pour dire à Python de remplacer la balise avec le nombre 12. Nous pouvons recycler cette chaîne et lui passer différentes valeurs :

à taper

```
>>> montexte = "Bonjour %s, comment ça va aujourd'hui ?"
>>> nom1 = "Guillaume"
>>> nom2 = "Johan"
>>> print(montexte % nom1)
Bonjour Guillaume, comment ça va aujourd'hui ?
>>> print(montexte % nom2)
Bonjour Johan, comment ça va aujourd'hui ?
```

Dans l'exemple ci dessus, trois variables (montexte, nom1 et nom2) sont créées, la première chaîne incluant une balise « %s ». Puis nous affichons la variable « montexte » et que nous utilisons l'opérateur « % » en passant en variable « nom1 » et « nom2 ». Vous pouvez utiliser plus d'un paramètre substituable :

à taper

```
>>> montexte = "Bonjour %s et %s, comment ça va ?"
>>> nom1 = "Guillaume"
>>> nom2 = "Johan"
>>> print(montexte % (nom2,nom1))
Bonjour Johan et Guillaume, comment ça va aujourd'hui ?
```

Quand vous voulez utiliser plus d'un marqueur, vous devez entourer les valeurs de remplacement par des parenthèses. Ainsi (nom2, nom1) est la manière approprié de passer deux variables. Un ensemble de valeurs entourées par des parenthèses « (» et «) » (et non pas des crochets []) est appelé un *n-uplet*[9] et ressemble un peu à une liste dont nous allons parler juste après.

2.7 Pas vraiment une liste de courses

Du lait, du fromage, du céleri, de la confiture et du sirop : ce n'est pas vraiment une liste de courses, mais c'est suffisant pour notre propos. Si vous conservez ces informations dans une variable vous pouvez créer une chaîne :

à taper

```
>>> courses = "lait, fromage, céleri, confiture, sirop"
>>> print(courses)
lait, fromage, céleri, confiture, sirop
```

Une autre manière de faire est de créer une « liste » qui est un type particulier d'objet en Python :

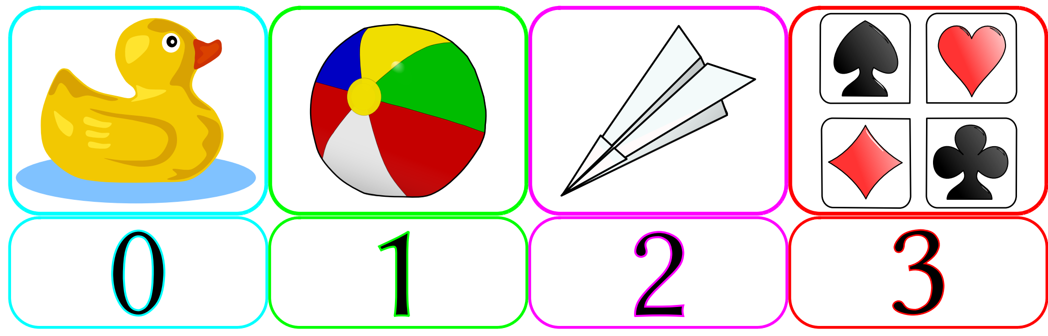
9. N-uplet provient de terminaison de ces ensembles à partir de quatre éléments : quadr*uplet*, quint*uplet*...

à taper
```
>>> liste_de_courses = ["lait", "fromage", "céleri", "confiture", "sirop"]
>>> print(liste_de_courses)
['lait', 'fromage', 'céleri', 'confiture', 'sirop']
```

C'est plus à taper, mais c'est aussi plus utile. Nous pouvons afficher le troisième élément dans la liste en utilisant sa position, nommée index, à l'intérieur de crochets « [] » :

à taper
```
>>> print(liste_de_courses[2])
céleri
```

Les listes commencent à la position 0 [10] ; ainsi le premier élément est le numéro 0, le deuxième est 1, le troisième est 2. Cela semble étrange à la plupart des gens, mais normal pour les programmeurs. Bientôt quand vous monterez les escaliers, vous commencerez à compter à partir de zéro plutôt que de un. Cela va vraiment déranger votre petit frère ou votre petite sœur.



Nous pouvons utiliser tous les objets de la liste du troisième au cinquième, en utilisant deux points « : » à l'intérieur des crochets :

à taper
```
>>> print(courses[2:5])
['céleri', 'confiture', 'sirop']
```

Utiliser « [2:5] » est une autre manière de dire que nous sommes intéressés par les objets depuis l'index 2 jusqu'à l'index 5, mais sans l'inclure. Et, bien sûr, comme nous commençons à compter à 0, le 3ᵉ dans la liste est, en fait, le numéro 2 et le 5ᵉ est, en fait, le numéro 4. Les listes peuvent être utilisées pour conserver toutes sortes d'objets. Elles peuvent stocker des nombres :

à taper
```
>>> maliste = [ 1, 2, 5, 10, 20 ]
```

Des chaînes :

à taper
```
>>> maliste = [ 'a', 'bbb', 'ccccccc', 'ddddddddd' ]
```

10. Python a été créé en 1991 après l'invention du zéro au IIIᵉ siècle av. J.-C. puis sa généralisation au Vᵉ siècle.

Des mélanges de nombres et de chaînes :

à taper
```
>>> maliste = [1, 2, 'a', 'bbb']
>>> print(maliste)
[1, 2, 'a', 'bbb']
```

Et même des listes de listes :

à taper
```
>>> liste1 = [ 'a', 'b', 'c' ]
>>> liste2 = [ 1, 2, 3 ]
>>> maliste = [ liste1, liste2 ]
>>> print(maliste)
[['a', 'b', 'c'], [1, 2, 3]]
```

Dans l'exemple ci-dessus, une variable appelée « `liste1` » est créée avec trois lettres, « `liste2` » est créée avec trois nombres et « `maliste` » est créée en utilisant liste1 et liste2. Les choses peuvent devenir rapidement compliquées si vous créez des listes de listes de listes de listes de listes de listes... Mais, par chance, il n'y a généralement pas vraiment besoin de faire des choses aussi compliquées en Python [11]. Néanmoins, il est utile de savoir que vous pouvez mettre toutes sortes d'objets dans une liste en Python.

Et pas seulement vos courses.

Remplacer des objets

Nous pouvons remplacer un objet dans une liste en fixant sa valeur de manière similaire à celle que nous utilisons pour assigner une valeur à une variable. Par exemple, nous pouvons changer le céleri en laitue en assignant une valeur en index 2 :

à taper
```
>>> liste_de_courses[2] = "laitue"
>>> print(liste_de_courses)
['lait', 'fromage', 'laitue', 'confiture', 'sirop']
```

Revenons à mon histoire d'étiquette : rappelez-vous je vous parlais d'étiquette et de variable qui pointait vers un objet. Imaginons que nous soyons plusieurs à utiliser la même liste de course. Je peux demander à mon cousin *Poche* de m'aider.

à taper
```
>>> liste_courses_poche=liste_de_courses
>>> liste_courses_poche[3]="moutarde"
>>> print(liste_de_courses)
['lait', 'fromage', 'laitue', 'moutarde', 'sirop']
```

Comme nous le voyons les modifications vues comme faites sur « `liste_courses_poche` » et de « `liste_de_courses` », ont eu lieu sur l'objet qui portait l'étiquette. Les variables pointent sur des objets mais ne sont pas les objets.

11. Python n'est pas un langage de Shadock.

Ajouter plus d'objets...

Nous pouvons ajouter des objets à une liste en utilisant une méthode appelée « `append` [12] ».

Une méthode est une action qui dit à Python ce que nous voulons qu'il fasse. Nous parlerons des méthodes plus tard, mais pour le moment, pour ajouter un objet à notre liste de courses, nous pouvons faire comme suit :

à taper
```
>>> liste_de_courses.append('chocolat')
>>> print(liste_de_courses)
['lait', 'fromage', 'laitue', 'confiture', 'sirop', 'chocolat']
```

Ce qui à défaut d'autre chose, est déjà une liste de courses améliorée.

... Et enlever des objets

Nous pouvons retirer des objets d'une liste en utilisant la commande « `del` » (abréviation de *delete* [13]).

Par exemple, pour retirer le 5e élément dans la liste (sirop) :

à taper
```
>>> del liste_de_courses[5]
>>> print(liste_de_courses)
['lait', 'fromage', 'laitue', 'confiture', 'chocolat']
```

Rappelez-vous que les positions commencent à zéro, donc « `liste_de_courses[4]` » faisait en fait référence au cinquième élément.

Deux listes valent mieux qu'une

Nous pouvons joindre des listes ensemble en les additionnant, comme si nous additionnions deux nombres :

à taper
```
>>> liste1 = [ 1, 2, 3 ]
>>> liste2 = [ 4, 5, 6 ]
>>> print(liste1 + liste2)
[1, 2, 3, 4, 5, 6]
```

Nous pouvons aussi additionner les deux listes et placer le résultat dans une troisième variable :

12. Le mot « *append* » signifie ajouter en anglais, du verbe latin *appendere* de *ad* (à la suite) et *pendere* (pendre).
13. Le mot « *delete* » signifier détruire ou effacer en anglais du participe passé *deletus* du verbe latin *delere* (détruire).

à taper
```
>>> liste1 = [ 1, 2, 3 ]
>>> liste2 = [ 4, 5, 6 ]
>>> liste3 = liste1 + liste2
>>> print(liste3)
[1, 2, 3, 4, 5, 6]
```

Et nous pouvons multiplier une liste de la même manière que nous avons multiplié une chaîne :

à taper
```
>>> liste1 = [ 1, 2 ]
>>> print(liste1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Dans l'exemple ci-dessus, multiplier liste1 par cinq est une autre manière de dire « répéter liste1 cinq fois ». Malgré tout la division « / » et la soustraction « - » ne fonctionnent pas sur les listes. Ainsi nous aurons des erreurs quand nous essayerons les exemples suivants :

erreur
```
>>> liste1 / 20
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

ou :

erreur
```
>>> liste1 - 20
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'type' and 'int'
```

Vous aurez même des messages d'erreur sacrément moches.

2.8 N-uplets et listes

Un n-uplet (mentionné plus tôt) est un petit peu comme une liste mais plutôt que d'utiliser des crochets vous utilisez des parenthèses. Vous pouvez utiliser des n-uplets de manière semblable aux listes :

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

La différence principale est que, contrairement aux listes, les n-uplets ne peuvent pas être changés une fois que vous les avez créés. Si vous essayez de remplacer une valeur comme vous l'avez fait plus tôt avec la liste, vous aurez un autre message d'erreur :

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

Cela ne signifie pas que vous ne pouvez pas changer la variable qui pointe vers le n-uplet pour la faire pointer vers quelque chose d'autre. Par exemple, ce code fonctionnera bien :

à taper
```
>>> mavar = (1, 2, 3)
>>> mavar = [ 'Une', 'liste', 'de', 'chaînes.' ]
```

D'abord nous avons créé une variable « `mavar` » qui pointait sur un n-uplet de trois nombres. Puis nous avons changé mavar pour qu'elle pointe vers une liste de chaînes. Cela peut sembler bizarre au départ.

C'est un peu comme des coffres qui seraient fermés à clef avec des verrous. Chaque coffre a un verrou (le nom de la variable) qui permet de l'ouvrir. Vous mettez quelque chose dans un coffre et vous l'identifiez avec votre cadenas. Puis finalement vous décidez d'utiliser le verrou pour un autre coffre.

Vous ne pouvez pas changer ce qu'il y a dans le coffre fermé. Mais vous pouvez toujours utiliser le verrou sur un autre coffre.

Mais si vous pouvez ouvrir et fermer le coffre et ajouter et retirer ce qui vous plait ; c'est une liste.

2.9 À vous de jouer

Dans ce chapitre nous avons vu comment calculer des équations mathématiques simples en utilisant Python de manière interactive. Nous avons vu comment les parenthèses pouvaient changer les résultats d'une équation en contrôlant l'ordre des opérations qui étaient utilisées. Nous avons trouvé comment dire à Python de se rappeler de valeurs pour un usage futur, en utilisant des variables. En plus nous avons vu que Python utilise des « chaînes » pour stocker du texte et des n-uplets et des listes pour gérer plus d'un objet.

2.9.1 Exercice 1

Faites une liste de vos jouets favoris et nommez la « jouets ». Faites une liste de vos plats préférés et nommez la « plats ». Joignez ces deux listes nommez le résultat « préférés ». Finalement affichez la variable « préférés » [14].

2.9.2 Exercice 2

Si vous avez trois boites qui contiennent vingt-cinq chocolats et dix sacs qui contiennent trente-deux bonbons, combien avez-vous de friandises au total ?

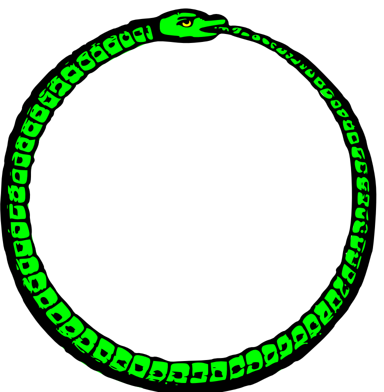
Note : il s'agit de faire le calcul dans la console ou le shell Python.

14. Vous trouverez les réponses aux section « À vous de jouer » en annexe A.

2.9.3 Exercice 3

Créez des variables pour votre prénom et votre nom. Maintenant créez une chaîne et utilisez des paramètres substituables pour y insérer votre prénom et votre nom.

Vous trouverez des pistes de réponses dans la section A.1.



3 Tortues et autres choses lentes

3.1 Un reptile plus lent que Python

Il y a certaines similarités entre les tortues du monde réel et une tortue Python. Dans le monde réel, une tortue est un reptile, parfois vert, qui se déplace très lentement et porte sa maison sur le dos. Dans le monde de Python, une tortue est une petite flèche noire qui se déplace lentement dans une fenêtre. Néanmoins il n'est nulle part fait mention d'une maison.

En fait, considérant le fait que la tortue Python laisse une trace derrière elle dans la fenêtre, cela en fait moins une tortue qu'un petit escargot ou une limace. Néanmoins, je suppose qu'un module appelé « limace » n'aurait pas été très attirant. C'est pour cela qu'il était plus sensé de garder les tortues. Il suffit d'imaginer une tortue qui transporterait quelques feutres et qui dessinerait en avançant.

Dans un passé lointain et sombre il y avait un simple langage de programmation appelé Logo. Logo était utilisé pour contrôler une tortue robot (appelé Irving). Au cours du temps la tortue a évolué depuis un robot qui pouvait se déplacer sur le sol à une petite flèche qui peut se déplacer sur l'écran.

Ce qui revient à montrer que les choses ne s'améliorent pas tout le temps avec les avancés de la technologie. Une petite tortue robot aurait été nettement plus drôle.

3.2 À la recherche de la tortue perdue

Le module « `turtle`[1] » de Python est juste un petit peu comme la tortue Logo. Nous verrons plus tard ce qu'est un module, mais pour le moment pensez juste que c'est quelque chose.

Néanmoins, Python a de nombreuses autres capacités que Logo. Le module « `turtle` » est une manière pratique d'apprendre comment les ordinateurs tracent des images sur votre

1. Le mot *turtle* signifie tortue en anglais et semble venir du français.

écran.

Bon commençons et voyons juste comment cela fonctionne. La première étape est de dire à Python que nous voulons utiliser « `turtle` » en important le module :

La gestion des fenêtres de Windows, différente de celle des autres systèmes d'exploitation, empêche de laisser le shell Python et la fenêtre « `turtle` » se recouvrir sans dysfonctionnement. En effet, si le shell Python et la fenêtre « `turtle` » se recouvrent une seule fenêtre verra son contenu mis à jour. Pour contourner ce problème, vous pouvez utiliser la ligne de commande Python ou réduire la largeur de la fenêtre du shell Python.

Vous pouvez voir sur la figure 3.1 le shell Python dont la largeur a été réduite de manière à empêcher le recouvrement.

Si le shell n'est pas assez réduit et que la fenêtre « `turtle` » et le shell sont superposées, il suffit de les déplacer ou de les redimensionner.

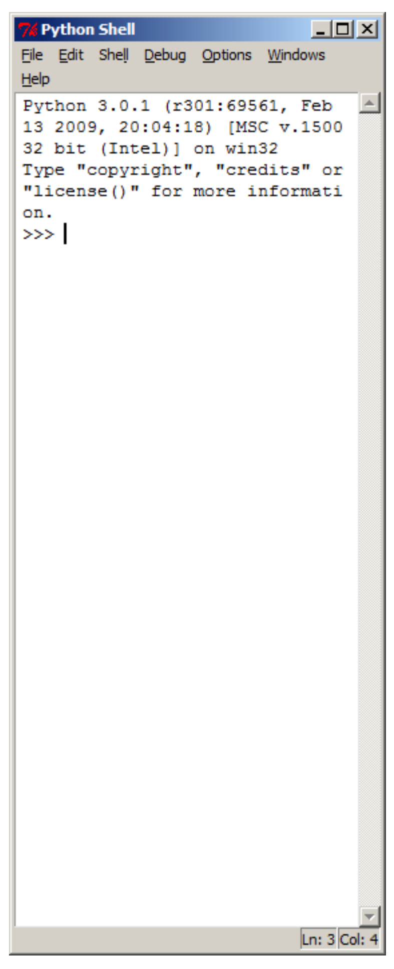


FIGURE 3.1 – Shell Python redimensionné pour utiliser turtle

Pour importer le module turtle faites :

à taper
```
>>> import turtle
```

Puis nous avons besoin d'afficher une feuille pour dessiner. Cette feuille est appelée en informatique zone de dessin. Cette zone de dessin est comme la toile qu'un artiste utilise pour peindre. Dans notre cas c'est juste un espace blanc pour dessiner :

à taper
```
>>> tortue = turtle.Pen()
```

Attention Python fait la différence entre les minuscules et les capitales [2], il convient d'écrire « `Pen` [3] » et non pas « `pen` ». Si vous vous êtes trompés retapez juste : « `tortue = turtle.Pen()` ».

2. On dit qu'il est sensible à la casse.
3. Le mot *pen* signifie stylo, du vieux français *penne* lui même du latin tardif *penna* signifiant plume.

Quand nous voulons quitter « `turtle` » il est conseillé de fermer au préalable la zone de dessin en faisant « `turtle.bye()` ».

à taper
```
>>> turtle.Pen()
```

Dans `tortue = turtle.Pen()` nous appelons une fonction particulière (Pen) du module turtle qui crée automatiquement une zone de dessin dans laquelle nous pouvons écrire et nous faisons pointer une variable vers le résultat de cette fonction de manière à pouvoir dessiner.

Une fonction est un bout de code réutilisable (à nouveau nous étudierons les fonctions plus tard) qui fait quelque chose qu'il est utile de faire plusieurs fois. Dans notre cas la fonction « `Pen` » retourne un objet qui représente notre tortue (enfin notre flèche en forme de tortue). Nous associons cet objet à la variable « `tortue` ».

Quand nous tapons le code dans la console Python, nous voyons une zone blanche apparaître (la zone de dessin dite *canvas* [4] en programmation) qui ressemble à la figure 3.2

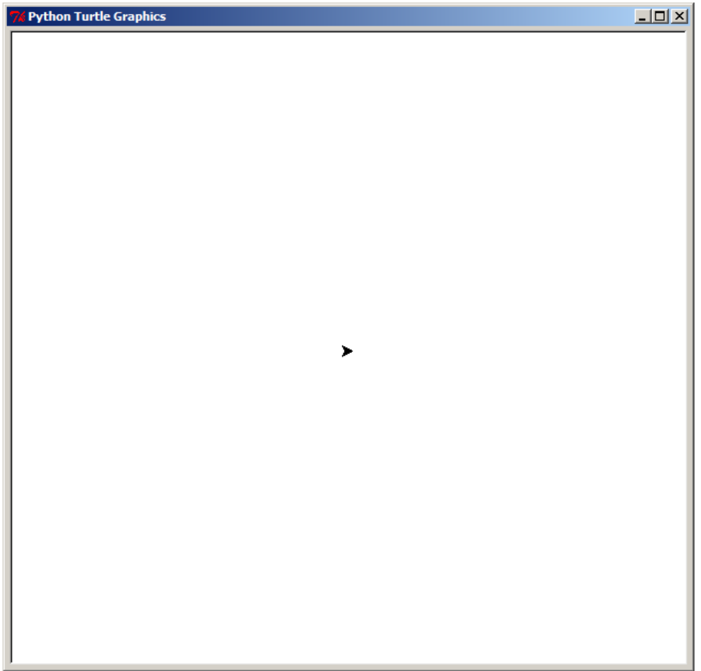


FIGURE 3.2 – Une flèche représente la tortue

Oui la petite flèche au milieu de l'écran est notre tortue. Et non, elle ne ressemble pas vraiment à une tortue.

La *Fausse Bonne Idée* serait de faire :

erreur
```
>>> turtle.Pen()
```

La zone de dessin serait créée mais sans étiquette pour identifier le stylo, nous ne pourrions pas dessiner dessus.

4. Le mot *canvas* signifie toile en anglais du français canevas.

Vous pouvez envoyer des instructions à la tortue, en utilisant des fonctions sur cet objet que nous venons de créer en appelant « `turtle.Pen` ». Comme nous avons assigné cet objet à la variable tortue, nous utilisons « `tortue` » pour envoyer des instructions. Une instruction de notre tortue est « `forward`[5] ».

L'instruction « `forward` » dit à notre tortue d'avancer vers l'avant quelque soit la direction vers laquelle elle pointe. Disons à la tortue de bouger vers l'avant de 50 *pixels* (nous allons parler des pixels dans une minute) :

à taper
```
>>> tortue.forward(50)
```

Vous devriez obtenir quelque chose qui ressemble à la figure 3.3.

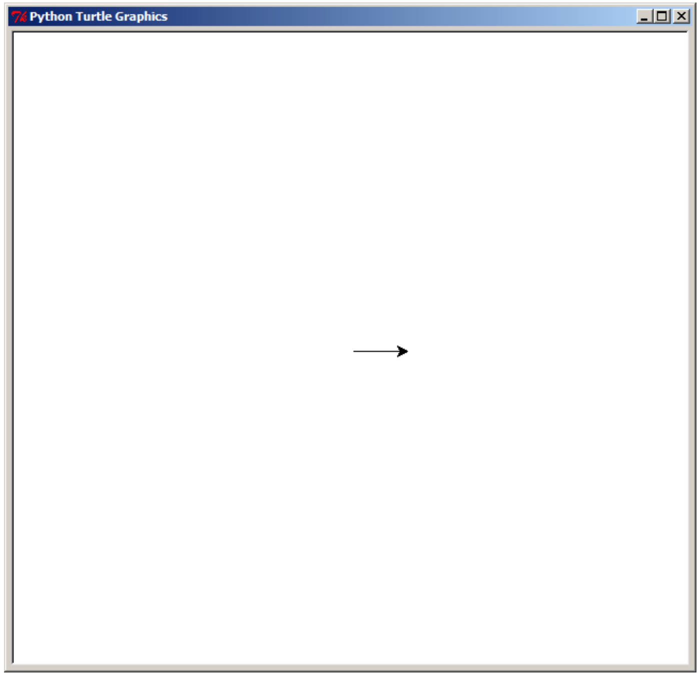


FIGURE 3.3 – La tortue dessine une ligne

De son point de vue, la tortue a avancé de cinquante pas. Du nôtre, elle s'est déplacée de 50 pixels.

Bon qu'est-ce que qu'un pixel ?

Un pixel est un point sur l'écran. Quand vous regardez l'écran de votre ordinateur tout est fait de petits points (approximativement) carrés. Les programmes que vous utilisez, et les jeux auxquels vous jouez sur l'ordinateur ou votre console, sont faits d'une multitude de différents points colorés arrangés sur l'écran. En fait si vous regardez l'écran avec une loupe vous devriez être capable de voir de quoi sont faits ces points. Ainsi si nous zoomons dans la zone de dessin au niveau de la ligne qui vient d'être tracée par la tortue, nous pouvons voir que la flèche qui représente la tortue est aussi un paquet de points carrés, comme vous pouvez le voir sur la figure 3.4.

Nous parlerons plus de ces points, ou pixels, dans un prochain chapitre. Maintenant nous allons dire à la tortue de tourner à gauche ou à droite :

5. Le mot *forward* vers l'avant du vieil anglais foreweard, *fore* (avant) + *-weard -ward* (suffixe de mouvement).

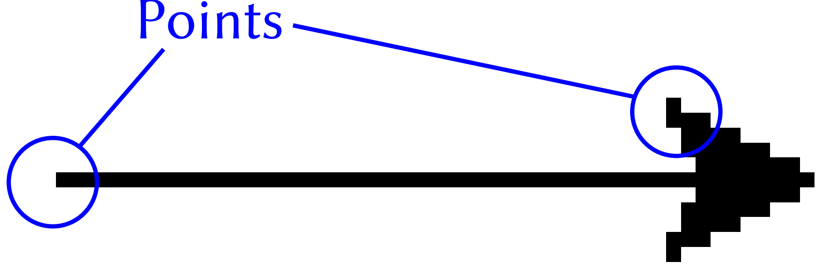


FIGURE 3.4 – Zoom sur la tortue

à taper
```
>>> tortue.left(90)
```

L'instruction « `left(90)` [6] » dit à la tortue de tourner vers la gauche de 90 degrés (souvent notés °). Vous n'avez peut-être pas encore appris ce que sont les degrés à l'école mais la manière la plus simple de se les représenter est qu'ils sont un peu comme les divisions d'une montre (voir la figure 3.5).

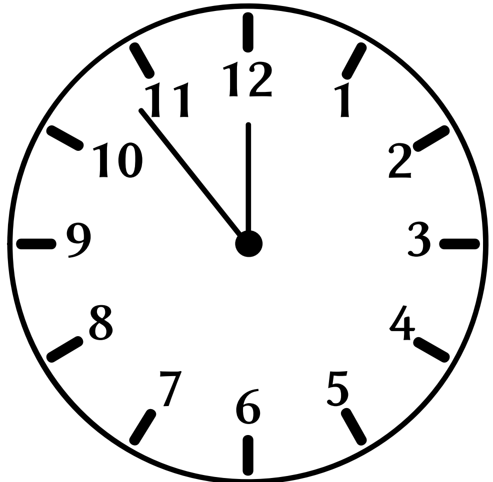


FIGURE 3.5 – Le cadran d'une montre et ses divisions

Contrairement à la montre et à ses 12 divisions (ou 60 si vous comptez les minutes plutôt que les heures), il y faut 360 degrés pour faire un tour complet. Ainsi vous comptez 360 divisions sur l'horloge, vous avez 90 où est normalement 3, 180 là où est normalement 6 et 270 où sont normalement 9. Zéro et 360 seraient en haut au départ où normalement est 12. La figure 3.6 vous montre les degrés sur une horloge.

6. Le mot *left* signifie gauche en anglais, du vieil allemand *lucht* (gauche).

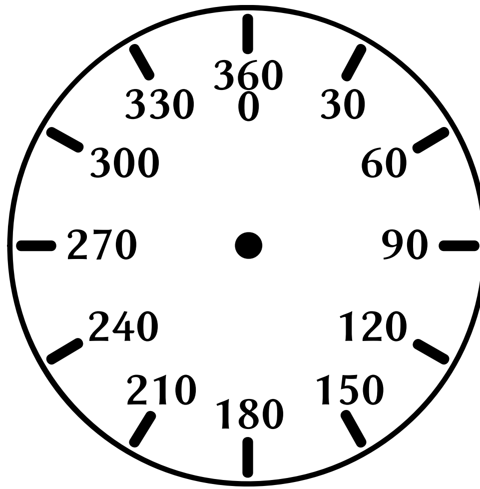


FIGURE 3.6 – Les degrés

Ainsi que veulent vraiment dire « `left(90)` » ?

Si vous êtes debout et face à une direction, pointez un bras directement dans la continuité de votre épaule, voilà c'est 90 degrés de la direction qui vous fait face (c'est un angle droit). Si vous pointez le bras gauche c'est 90 degrés vers la gauche. Si vous pointez le bras droit c'est 90 degrés vers la droite. Quand la tortue Python tourne vers la gauche, elle met son nez sur un point puis pivote son corps pour pointer vers la nouvelle direction (comme si vous aviez tourné votre corps de manière à faire face vers où le bras pointait).

Ainsi suite à « `tortue.left(90)` » la tortue-flèche pointe vers le haut, comme montré dans la figure 3.7.

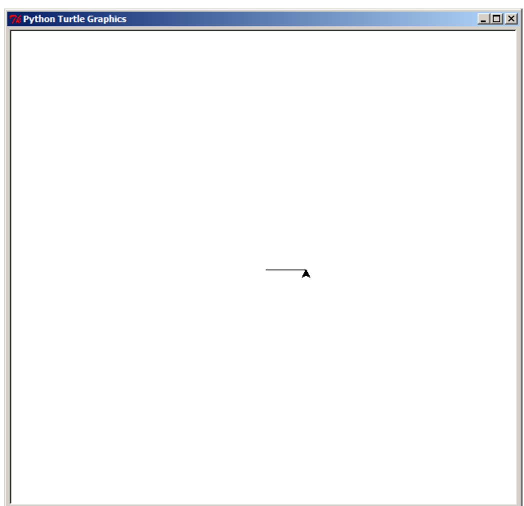


FIGURE 3.7 – Tortue après avoir tourné de 90°

Recommençons à nouveau quelques fois, en utilisant l'historique :

à taper
```
>>> tortue.forward(50)
>>> tortue.left(90)
>>> tortue.forward(50)
```

```
>>> tortue.left(90)
>>> tortue.forward(50)
>>> tortue.left(90)
```

Notre tortue a dessiné un carré et pointe maintenant dans la même direction et au même endroit que là où elle avait commencé (voir figure 3.8).

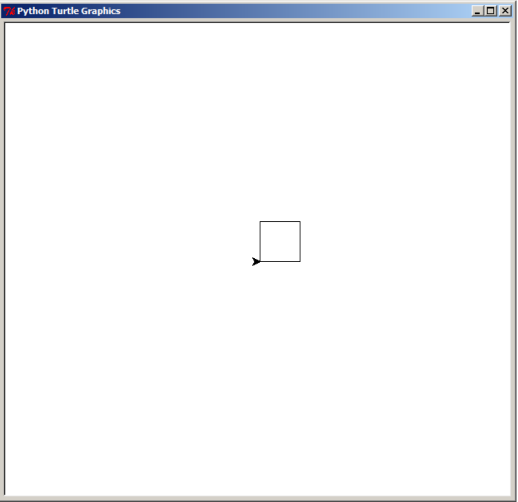


FIGURE 3.8 – Tortue après un carré

Nous pouvons effacer ce qu'il y a sur la zone de dessin en utilisant « `clear()` » (effacer en anglais).

Il existe quelques autres fonctions que vous pouvez utiliser :

- « `right()` » [7] qui tourne la tortue vers la droite ;
- « `reset()` » [8] qui elle aussi efface l'écran mais replace automatique la tortue à sa position initiale ;
- « `backward()` » [9] qui fait reculer la tortue ;
- « `up()` » [10] qui dit à la tortue de ne plus dessiner (qui soulève le stylo) ;
- « `down()` » [11] qui dit à la tortue de dessiner à nouveau.

Vous pouvez utiliser ces fonctions de la même manière que vous avez utilisé les autres :

à taper

```
>>> tortue.reset()
>>> tortue.backward(100)
>>> tortue.right(90)
>>> tortue.forward(100)
>>> tortue.up()
>>> tortue.forward(100)
>>> tortue.down()
>>> tortue.forward(100)
```

Nous reviendrons au module « `turtle` » bientôt.

7. droite en anglais
8. remise à zéro en anglais
9. vers l'arrière en anglais
10. vers le haut
11. vers le bas

3.3 À vous de jouer

Dans ce chapitre nous avons vu comment utiliser la tortue pour dessiner de simples lignes, la faire tourner à gauche et à droite. Nous avons vu que la tortue utilise des degrés pour tourner, un peu comme des minutes sur un horloge.

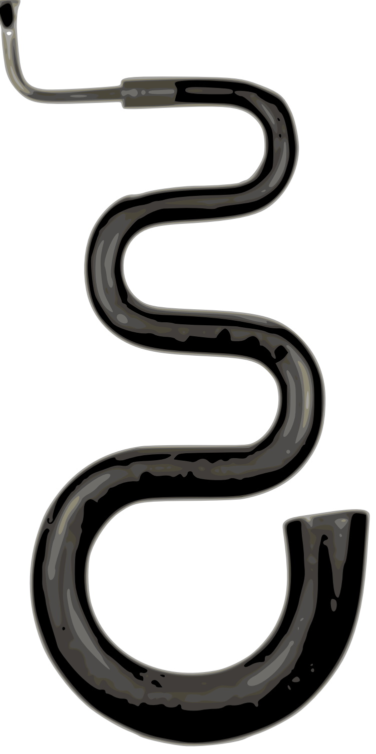
Vous trouverez des pistes de réponses dans la section A.2.

Exercice 1

Créez une zone de dessin en utilisant la fonction Pen() de « `turtle` » et dessinez un rectangle.

Exercice 2

Créez une autre zone de dessin en utilisant la fonction Pen() de « `turtle` » et dessinez un triangle.



4 Comment poser une question

Attention, l'ensemble des exemples de ce chapitre sont entourés de vert car il faut copier avec exactitude les espaces *ajoutées* dans ces codes. Toutes les indentations de ce chapitre sont composées de quatre espaces (quatre appuis sur la barre d'espace). Ces indentations définissent des blocs que nous verrons dans le chapitre suivant.

Si vous n'entrez pas les espaces correctement vous pourriez avoir une erreur du type :

erreur
```
IndentationError: expected an indented block
```

Les espaces et les des tabulations « ⇆ » sont extrêmement importantes en Python. Nous en parlerons en détail dans le chapitre suivant.

4.1 Avec des « si » on mettrait Paris en bouteille

En termes de programmation, une question signifie usuellement que nous voulons faire des choses différentes selon la réponse à la question. Cela est appelé un « test si ».

Par exemple :

Quel âge avez-vous ? Si vous avez plus que 18 ans, vous êtes majeur !

Cela peut être écrit en Python avec le « `test si` » suivant :

ne pas saisir
```
if âge > 18:
    print('Vous êtes majeur !')
```

Une représentation graphique peut être observée figure 4.1.

Un test *si* est fait d'un « `if` [1] » suivi de ce que nous appelons une « condition » (plus de détails dans une seconde) suivi par deux points « : ». Les lignes qui suivent le « `if` » doivent constituer un bloc ; si la réponse à la question est « oui » (ou « vrai » comme nous le disons en termes de programmation) alors les commandes dans le bloc seront exécutées.

1. Si l'if est un arbuste en français, il s'agit ici du mot anglais *if* signifie « si » en français.

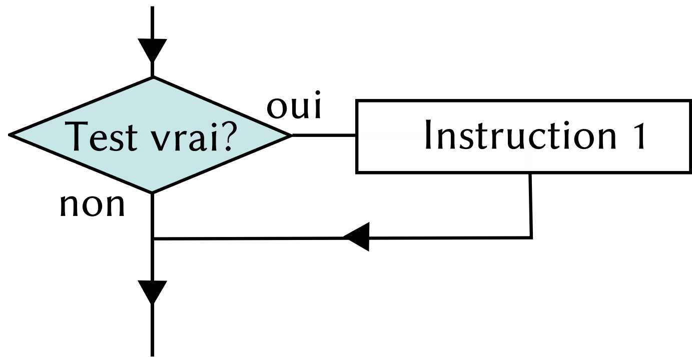


FIGURE 4.1 – Diagramme d'un test « si »

Une condition est un élément de programmation qui retourne « oui » (vrai) ou « non » (faux). Certains symboles (ou opérateurs) sont utilisés pour créer des conditions, comme dans la table 4.1 :

Opérateur	Opération
==	Égal
!=	Différent
>	Plus grand que
<	Plus petit que
>=	Plus grand ou égal à
<=	Plus petit ou égal à

TABLE 4.1 – Opérateurs de condition

Par exemple si vous avez 10 ans alors la condition « `votre_âge == 10` » retournera vrai (oui). Si vous n'avez pas dix ans elle retournera faux.

> Rappelez-vous : ne confondez pas les *deux* symboles utilisés dans une condition « == », avec le signe égal utilisé pour assigner une variable. Si vous utiliser un seul symbole « = » dans une condition, vous obtiendrez un message d'erreur.

Présumons que vous avez attribué votre âge à une variable « `âge` » et bien si vous avez douze ans la condition « `âge > 10` » sera *vraie*.

Si vous avez huit ans elle retournera *faux*. Si vous avez dix ans elle retournera *faux* car la condition est vérifiée pour *strictement* plus grand « > » que dix et non pas pour plus grand ou égal « >= » à dix.

Essayons quelques exemples :

à taper avec attention
```
>>> âge = 10
>>> if âge > 10:
...     print('Arrivé ici !')
```

Si vous avez obtenu une erreur c'est que vous avez oublié quelques espaces (que je vous montre ici en les remplaçant par des @) :

ne pas saisir
```
>>> âge = 10
>>> if âge > 10:
... @@@@print('Arrivé ici !')
```

Si vous entrez l'exemple ci dessus dans une console qu'arrivera-t-il ?

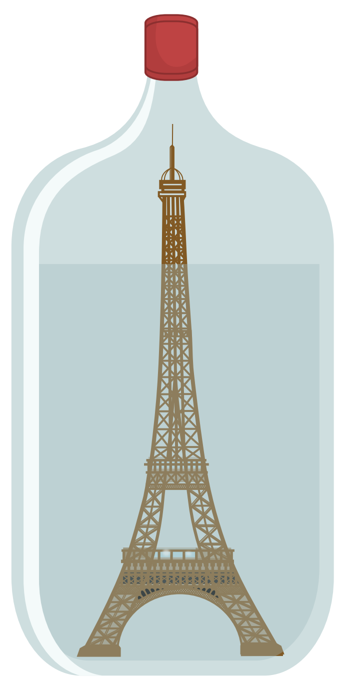
Rien !

Parce que la valeur de la variable « `âge` » n'est pas strictement plus grande que dix, la commande « `print` » dans le bloc ne sera pas lancée. Maintenant étudions :

à taper avec attention
```
>>> âge = 10
>>> if âge >= 10:
...     print('Arrivé ici !')
Arrivé ici !
```

Si nous essayons cet exemple, alors nous pouvons voir le message affiché. La même chose arrivera avec l'exemple suivant :

à taper avec attention
```
>>> âge = 10
>>> if âge == 10:
...     print('Arrivé ici !')
Arrivé ici !
```



4.2 Fait cela ! Ou sinon...

Nous pouvons aussi étendre un test « si » et faire quelque chose quand la condition n'est pas vraie. Par exemple, afficher le mot « Bonjour » sur la console si votre âge est de douze

ans mais « Au revoir » s'il est différent. Pour faire cela nous utilisons un test « si sinon », ce qui est une autre manière de dire « si quelque chose est vrai, fais ceci sinon cela » :

à taper avec attention
```
>>> âge = 12
>>> if âge == 12:
...     print('Bonjour !')
... else:
...     print('Au revoir !')
Bonjour !
```

Le test « si sinon » utilise « `if` » pour si et « `else` [2] » pour sinon.

Rentrez l'exemple ci-dessus et vous devriez voir « `Bonjour !` » affiché dans la console. Changez la valeur de la variable « `âge` » à une autre valeur et « `Au revoir !` » sera affiché :

à taper avec attention
```
>>> âge = 8
>>> if âge == 12:
...     print('Bonjour !')
... else:
...     print('Au revoir !')
Au revoir !
```

Une représentation graphique peut être observée figure 4.2.

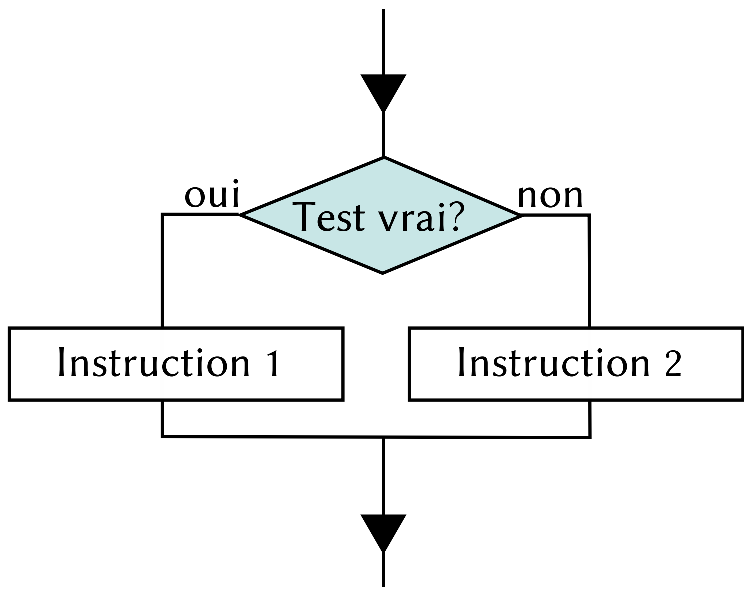


FIGURE 4.2 – Diagramme d'un test « si »

4.3 Fais cela, ou cela, ou cela ! Ou sinon...

Nous pouvons étendre le test « si » encore plus loin en utilisant « `elif` » (l'abréviation de else-if, « sinon, si »). Par exemple, nous pouvons vérifier si votre âge est égal à dix, puis à onze, puis à douze et ainsi de suite :

2. Le mot *else* signifie « sinon » et provient du latin *alius* (autre) et *alter* (l'autre de deux).

à taper avec attention

```
>>> âge = 12
>>> if âge == 10:
...     print('Vous avez 10 ans.')
... elif âge == 11:
...     print('Vous avez 11 ans.')
... elif âge == 12:
...     print('Vous avez 12 ans.')
... elif âge == 13:
...     print('Vous avez 13 ans.')
... else:
...     print('Je ne connais pas votre âge.')
Vous avez 12 ans.
```

Dans le code ci-dessus, le code vérifie à la deuxième ligne si la valeur de la variable « âge » est égale à dix. Si elle ne l'est pas alors il saute à la quatrième ligne pour vérifier si la valeur de la variable « âge » est égale à onze. À nouveau celle-ci ne l'est pas donc il saute à la sixième ligne pour vérifier si la valeur de la variable « âge » est égale à douze. Dans ce cas cela est le cas donc Python va dans le bloc à la septième ligne et exécute la commande « `print` ». Vous avez probablement remarqué qu'il a cinq groupes dans ce code aux lignes 3, 5, 7, 9 et 11.

Une représentation graphique peut être observée figure 4.3.

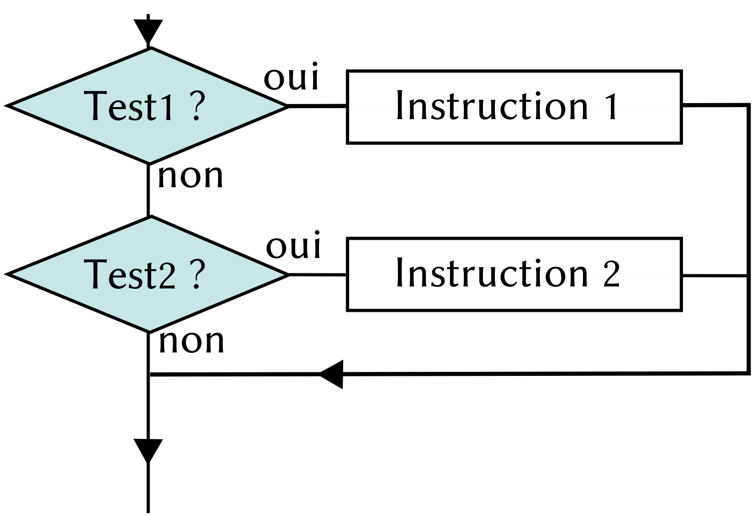


FIGURE 4.3 – Diagramme d'un test « si »

4.4 Combiner des conditions

Vous pouvez combiner des conditions ensemble en utilisant les mots clef « `and` » qui signifie « et » et « `or` » qui signifie « ou ».

Nous pouvons condenser l'exemple précédent, un petit peu, en utilisant « `or` » pour joindre les conditions ensemble :

à taper avec attention
```
>>> if âge == 10 or âge == 11 or âge == 12 or âge == 13:
...     print('Vous avez %s ans.' % âge)
... else:
...     print('Heu ?')
```

Si n'importe laquelle des conditions de la première ligne est vraie, c'est à dire si « `âge` » est égal à 10, 11, 12 ou 13, alors le bloc de code ligne 2 sera exécuté. Sinon Python ira à la quatrième ligne. Nous pouvons condenser ce code un peu plus en utilisant les opérateurs « `and` », « `>=` », « `<=` » :

à taper avec attention
```
>>> if âge >= 10 and âge <= 13:
...     print('Vous avez %s ans.' % âge)
... else:
...     print('Heu ?')
```

Vous avez probablement compris que si les deux conditions de la première ligne sont vraies alors le bloc de la deuxième ligne est exécuté (si « `âge` » est plus grand ou égal à dix et « `âge` » est plus petit ou égal à 13. Donc si la variable « `âge` » vaut douze alors « `Vous avez 12 ans.` » sera affiché sur la console parce que douze est plus grand que dix et plus petit que treize.

4.5 Rien

Il y a une autre sorte de valeur qui peut être assignée à une variable et dont nous n'avons pas encore parlé dans les chapitres précédents : **rien** !

De la même manière que les nombres, les chaînes et les listes peuvent être assignés à des variables, « rien » est aussi un type de valeur qui peut être assigné. En Python une valeur vide est désignée comme « `None` [3] ».

Une variable valant « `None` » (dans d'autres langages, « `null` » est parfois utilisé) peut être utilisée comme les autres variables :

à taper
```
>>> maval = None
>>> print(maval)
None
```

{}==∅==None

« `None` » est une manière de réinitialiser une variable comme étant non utilisée ou une manière de créer une variable sans fixer sa valeur avant de l'utiliser.

3. Le mot *none* signifie aucun, personne en anglais, contraction de *no one* pas un.

Par exemple si votre équipe de football lève des fonds pour de nouveaux uniformes, vous voulez peut-être attendre que toute l'équipe soit revenue avec l'argent récolté, avant d'additionner les montants. En termes de programmation, vous pouvez avoir une variable pour chaque joueur de l'équipe et initialiser ces variables à « None » :

à taper

```
>>> joueur1 = None
>>> joueur2 = None
>>> joueur2 = None
```

Nous pouvons utiliser un test « si » pour vérifier ces variables de manière à déterminer si tous les membres de l'équipe sont revenus avec l'argent qu'ils ont levé :

à taper avec attention

```
>>> if joueur1 is None or joueur2 is None or joueur3 is None:
...     print('''S'il-vous-plait, attendez jusqu'à ce
...que tous les joueurs soient revenus.''')
... else:
...     print('Vous avez levé %s€' % (player1 + player2 + player3))
```

Le test « si » contrôle si une des variables a une valeur nulle et affiche le premier message si cela est la cas.

Le mot *is* est la troisième personne du verbe *to be*, c'est à dire « être » en français. Le test est donc « Est ce que joueur1, joueur2 ou joueur3 sont nuls ? ».

Si chaque variable a une valeur non nulle alors le second message est affiché avec la somme totale récoltée. Si vous essayez le code avec toutes les variables qui pointent vers « None », vous aurez le premier message (n'oubliez pas de créer les variables d'abord ou vous aurez un message d'erreur) :

à taper avec attention

```
>>> if joueur1 is None or joueur2 is None or joueur3 is None:
...     print('''S'il-vous-plait, attendez jusqu'à ce
...que tous les joueurs soient revenus.''')
... else:
...     print('Vous avez levé %s€' % (player1 + player2 + player3))
S'il-vous-plait, attendez jusqu'à ce
que tous les joueurs soient revenus.
```

Même si vous avez modifié une ou deux variables, vous continuerez d'avoir le même message :

à taper avec attention

```
>>> joueur1=100
>>> joueur2=300
>>> if joueur1 is None or joueur2 is None or joueur3 is None:
...     print('''S'il-vous-plait, attendez jusqu'à ce
...que tous les joueurs soient revenus.''')
... else:
...     print('Vous avez levé %s€' % (player1 + player2 + player3))
S'il-vous-plait, attendez jusqu'à ce
...que tous les joueurs soient revenus.
```

Finalement une fois toutes les variables modifiées vous verrez le message du second bloc :

à taper avec attention
```
>>> joueur1=100
>>> joueur2=300
>>> joueur3=400
>>> if joueur1 is None or joueur2 is None or joueur3 is None:
...     print('''S'il-vous-plait, attendez jusqu'à ce
...que tous les joueurs soient revenus.''')
... else:
...     print('Vous avez levé %s€' % (player1 + player2 + player3))
Vous avez levé 800€
```

4.6 Quelle est la différence ?

Quelle est la différence entre « « 10 » » et « '10' » ? Pas grande mis à part les apostrophes, pouvez-vous penser. Et bien, si vous avez bien lu les chapitres précédents vous savez que le premier est un nombre et que le second est une chaîne. Cela les rend plus différents que ce à quoi vous pourriez vous attendre. Auparavant nous avions comparé la valeur de la variable « âge » à un nombre dans un test « si » :

à taper avec attention
```
>>> if âge == 10:
...     print('Vous avez 10 ans.')
```

Si vous aviez attribué dix à la variable « âge » la fonction « print » sera appelée :

à taper avec attention
```
>>> âge = 10
>>> if âge == 10:
...     print('Vous avez 10 ans.')
...
Vous avez 10 ans.
```

Mais si « âge » a pour valeur « "10" » alors la fonction « print » ne sera pas appelée :

à taper avec attention
```
>>> âge = '10'
>>> if âge == 10:
...     print('Vous avez 10 ans.')
...
```

Pourquoi le code dans le bloc n'a pas été exécuté ? Parce qu'une chaîne est différente d'un nombre, même s'ils se ressemblent :

à taper
```
>>> âge1 = 10
>>> âge2 = '10'
>>> print(âge1)
```

```
10
>>> print(âge2)
10
```

Regardez ! Ils semblent exactement identiques. Pourtant, parce que l'un est une chaîne et l'autre est un nombre, ils ont des valeurs différentes. Néanmoins après « `âge="10"` », « `âge == 10` » sera toujours faux tant qu'« `âge` » sera une chaîne. Nous sommes trompés par Python dont la fonction « `print` » est capable de convertir un nombre en chaîne pour l'afficher.

Il y a probablement une meilleure manière de penser les choses : c'est de considérer dix livres et dix briques. Le nombre d'éléments peut être le même mais vous ne diriez pas que dix livres sont exactement la même chose que dix briques, n'est-ce-pas ? Par chance, en Python nous avons des fonctions magiques qui peuvent transformer des chaînes en nombres et des nombres en chaînes même si elles ne peuvent pas vraiment transformer les briques en livres. Par exemple, pour convertir la chaine « `'10'` » en un nombre vous pouvez utiliser la fonction « `int` » :

à taper

```
>>> âge_en_chaîne = '10'
>>> âge_en_nombre = int(âge)
>>> print(âge_en_chaîne*2)
1010
>>> print(âge_en_nombre*2)
20
```

L'abbreviation « `int` » est utilisé pour *integer* qui signifie entier en anglais. C'est-à-dire un nombre entier (sans virgule).

La variable « `âge_en_nombre` » contient le nombre dix et pas une chaîne. Pour convertir un nombre en une chaîne, vous pouvez utiliser la fonction « `str` » :

à taper

```
>>> âge_en_nombre = 10
>>> âge_en_chaîne = str(âge_en_nombre)
```

L'abréviation « `str` » est utilisée au lieu de *string* qui signifie ficelle en anglais mais plus particulièrement chaîne de caractères en programmation [4].

« `âge_en_chaîne` » contient la chaîne « `'10'` » et non pas un nombre. Revenons à notre test « si » qui n'imprimait rien :

à taper avec attention

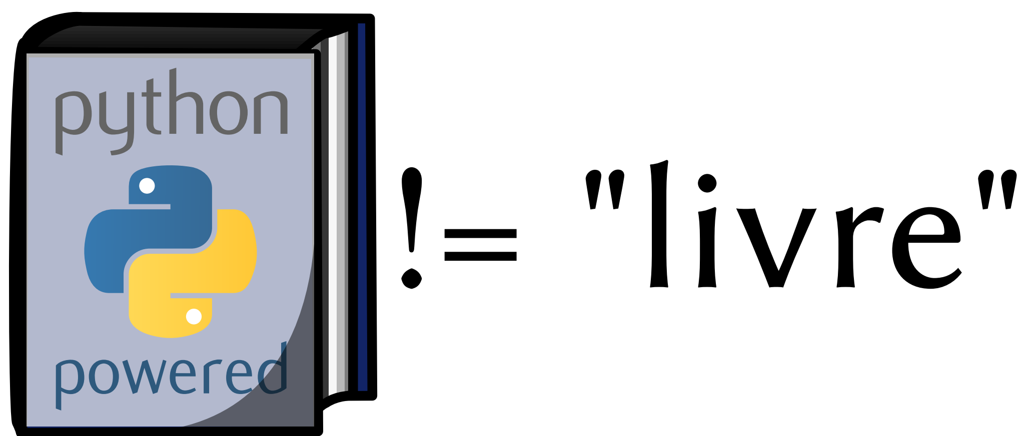
```
>>> âge = '10'
>>> if âge == 10:
...     print('Vous avez dix ans.')
...
```

Si nous convertissons la variable avant le test alors nous aurons un résultat différent :

4. L'appellation française est pour une fois nettement plus compréhensible que la forme originale en anglais.

à taper avec attention

```
>>> âge = '10'
>>> âge_converti=int(âge)
>>> if âge_converti == 10:
...     print('Vous avez dix ans.')
...
Vous avez dix ans.
```



5 Encore et encore

5.1 Rabâchage

Il n'y pas grand chose de pire que de faire la même chose encore et encore [1]. C'est la raison pour laquelle vos parents vous disent de compter les moutons pour essayer de dormir et cela n'a rien à voir avec les incroyables pouvoirs somnifères des mammifères laineux. Cela a tout à voir avec le fait que répéter quelque chose sans fin est ennuyeux, et que votre esprit devrait tomber dans le sommeil plus facilement s'il n'est pas en train de se concentrer sur quelque chose d'intéressant.

Les programmeurs, eux aussi, n'aiment pas particulièrement répéter les choses. Cela les endort aussi. C'est la vraie raison pour laquelle tous les langages de programmation de haut niveau [2] ont un concept appelé une boucle. Par exemple, pour afficher « hello » cinq fois en Python vous pouvez faire comme suit :

ne pas saisir

```
>>> print("hello")
hello
>>> print("hello")
hello
>>> print("hello")
hello
>>> print("hello")
hello
>>> print("hello")
hello
```

Ce qui est... Plutôt ennuyant, pour être poli.

Ou nous pouvons utiliser une boucle :

à taper avec attention

```
>>> for x in range(0, 5):
...     print('hello')
```

1. http://fr.wikipedia.org/wiki/Sisyphe
2. Les boucles ont été introduites avec le langage Fortran en 1958.

```
...
hello
hello
hello
hello
hello
```

Comme au chapitre précédent les espaces sont importants, je vous les montre de manière plus explicite avec des arobases :

ne pas saisir
```
>>> for x in range(0, 5):
... @@@@print('hello')
...
```

Le mot *for* signifie « pour » en anglais, *in* signifie « dans » et *range* signifie gamme (de produit), éventail (de valeurs), série (de valeurs).

La fonction « `range` » permet de créer facilement et rapidement une liste (une série) de nombre entre un début et une fin, par exemple :

à taper
```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Si nous ne fournissons qu'un nombre à « `range` » la liste part de zéro (« `range(n) == range(0,n)` » :

à taper
```
>>> print(list(range(5)))
[0, 1, 2, 3, 4]
```

Le mot *list* signifie liste (oui je sais vous aviez deviné).

Dans le cas de Python la boucle utilisée s'appelle un itérateur. Le code « `for x in range(0,5)` » dit en fait à Python de créer une liste de nombre (0, 1, 2, 3, 4) puis pour chacun de ces nombres de stocker cette valeur dans la variable « `x` » et enfin d'exécuter la commande dans le bloc après les deux points pour chaque valeur de « `x` ». Nous pouvons utiliser ce « `x` » dans notre déclaration qui contient « `print` » si nous le voulons :

à taper avec attention
```
>>> for x in range(0, 5):
...     print('hello %s' % x)
hello 0
hello 1
hello 2
hello 3
hello 4
```

Une représentation graphique peut être observée figure 5.1.

Si nous déroulions la boucle que nous venons d'éxécuter, cela pourrait ressembler à quelque chose comme cela :

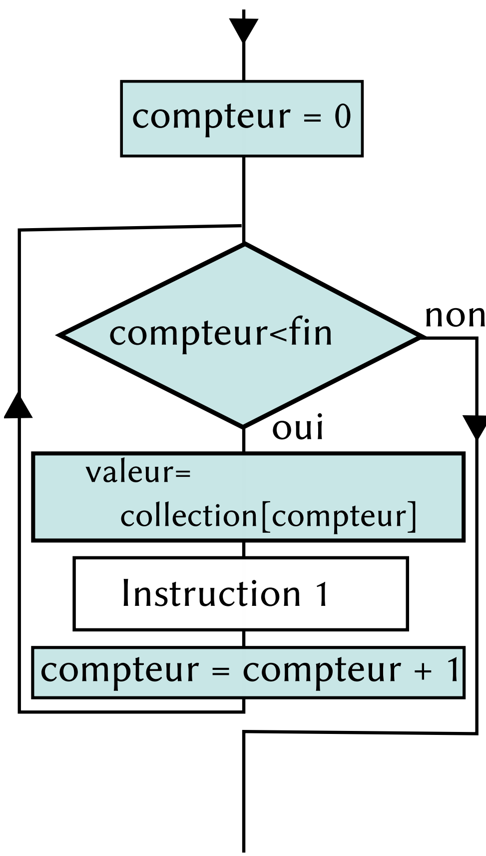


FIGURE 5.1 – Diagramme d’un itérateur

ne pas saisir

```
x = 0
print('hello %s' % x)
x = 1
print('hello %s' % x)
x = 2
print('hello %s' % x)
x = 3
print('hello %s' % x)
x = 4
print('hello %s' % x)
```

Ainsi cette boucle (cet itérateur) nous a épargné d’écrire huit lignes de code supplémentaires. C’est vraiment utile, d’autant plus que le programmeur moyen est plus encore paresseux qu’un hippopotame (un jour chaud) quand il s’agit de taper quelque chose. Les bons programmeurs détestent faire les choses plus d’une fois, donc les boucles sont une des structures les plus utiles dans un langage de programmation.

Nous n’avons pas obligation d’utiliser « `range` », nous pouvons utiliser les listes que nous avons déjà utilisées :

à taper avec attention

```
>>> courses=['lait', 'fromage', 'laitue', 'confiture', 'sirop', 'chocolat']
>>> for i in courses:
```

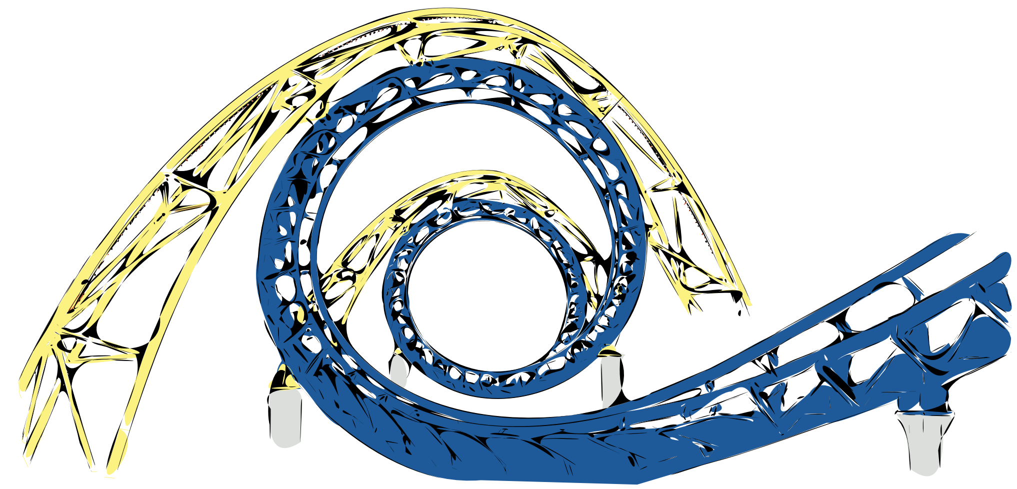
```
...     print(i)
lait
fromage
laitue
confiture
sirop
chocolat
```

Le code si dessus est une manière de dire : « pour chaque élément de la liste, stocke la valeur dans la variable `i` et affiche le contenu de cette variable »[3]. À nouveau si nous déroulons la boucle nous aurons quelque chose comme :

ne pas saisir

```
>>> courses=['lait', 'fromage', 'laitue', 'confiture', 'sirop', 'chocolat']
>>> print(courses[0])
lait
>>> print(courses[1])
fromage
>>> print(courses[2])
laitue
>>> print(courses[3])
confiture
>>> print(courses[4])
sirop
>>> print(courses[5])
chocolat
```

De nouveau la boucle nous a épargné beaucoup d'écriture.



5.2 Quand est-ce qu'un bloc n'est pas compact ?

Quand c'est un bloc de code.

3. Les variables utilisées dans les boucles simples ont rarement des noms explicites. Dans une boucle plus complexe « `produit_à_acheter` » pourrait être plus opportun.

Qu'est-ce qu'un « bloc de code », alors ?

Un bloc de code est une série d'instructions que vous voulez grouper ensemble. Par exemple, dans la boucle ci-dessus vous pourriez avoir envie de faire plus qu'afficher des éléments. Peut-être que vous voudriez acheter chaque objet et afficher qu'il l'a bien été. Supposons que nous avons une fonction appelée « `acheter` » vous pourriez écrire quelque chose comme cela :

ne pas saisir

```
>>> for i in courses :
...         acheter(i)
...         print(i)
```

Ne vous cassez pas la tête à taper cet exemple dans la console Python car nous n'avons pas de fonction « `acheter` » et que vous auriez un message d'erreur si vous tentiez de la lancer. Néanmoins cet exemple permet de montrer un bloc fait de deux commandes :

ne pas saisir

```
acheter(i)
print(i)
```

De manière plus utile nous pouvons aussi tracer un cercle avec la tortue sans fonction cercle :

à taper avec attention

```
>>> import turtle
>>> tortue=turtle.Pen()
>>> for i in range(360):
...     tortue.forward(1)
...     tortue.left(1)
...
```

Vous pouvez observer le résultat sur la figure 5.2.

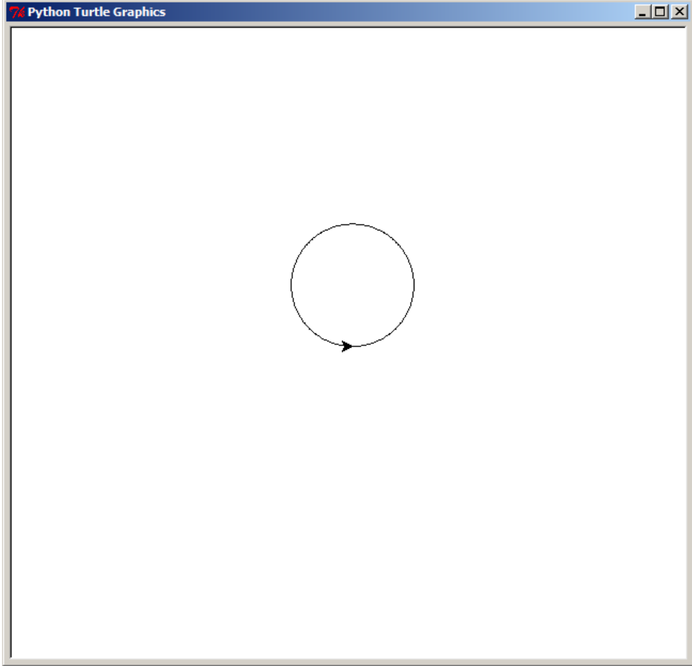


FIGURE 5.2 – Un cercle fait en 360 étapes

En Python, les espaces et les tabulations « ⇆ » sont extrêmement importantes (bis). Le code qui est positionné à la même position horizontalement est groupé ensemble en blocs.

La figure 5.3 montre le découpage en bloc selon la position.

```
Ceci est le bloc 1
Ceci est le bloc 1
Ceci est le bloc 1
    Ceci est le bloc 2
    Ceci est le bloc 2
    Ceci est le bloc 2
Ceci est encore le bloc 1
Ceci est encore le bloc 1
Ceci est encore le bloc 1
      Ceci est le bloc 3
      Ceci est le bloc 3
        Ceci est le bloc 4
        Ceci est le bloc 4
```

FIGURE 5.3 – Les blocs en Python

On commence un nouveau bloc après une ligne finissant par « : ». Le nouveau bloc est défini par rapport à son indentation (la distance avec le bord gauche). Le bloc initié par « : » doit avoir une indentation plus grande (être plus à droite) que le bloc précédent.

Attention vous devez être cohérents avec vos espacements. Par exemple, nous aurons une erreur avec :

erreur
```
>>> for i in courses:
...     acheter(i)
...       print(i)
```

La seconde ligne (avec la fonction « `acheter(i)` ») commence avec **quatre** espaces. La troisième ligne commence avec **six** espaces. Regardons attentivement le code en mettant en valeur les espaces avec des arobases :

ne pas saisir

```
>>> for i in courses:
... @@@@acheter(i)
... @@@@@@print(i)
```

Cela causera une erreur. Quand nous commençons un bloc avec quatre espaces vous devez continuer quatre espaces. Et cela tant que vous ne créez pas un nouveau bloc avec « : » ou que vous ne finissiez pas le bloc en revenant à un niveau d'indentation précédent.

Il est d'usage d'indenter les blocs avec des espaces, ou pour le moins, de ne pas mélanger les espaces et les tabulations. De plus on utilise généralement des pas d'indentation constant de quatre espaces. Si vous commencez un bloc dans un bloc indenté de quatre espaces, il convient d'indenter le nouveau de huit (2x4). Par exemple, on tapera :

ne pas saisir

```
@@@@un premier bloc
@@@@un premier bloc
@@@@un premier bloc
@@@@@@@@un deuxième bloc
@@@@@@@@un deuxième bloc
@@@@@@@@un deuxième bloc
```

Pourquoi voudrions-nous mettre un bloc « dans » un autre ? Généralement nous faisons cela quand le deuxième bloc repose sur le premier d'une certaine façon, comme dans une boucle.

Si nous commençons une boucle dans le premier bloc alors les instructions que nous voulons exécuter encore et encore sont dans le second, le second bloc repose sur le premier pour fonctionner correctement.

Dans une console Python, une fois que vous commencez à taper du code dans un bloc, Python continue ce bloc jusqu'à ce que vous pressiez « entrée » sur une ligne vide (ou que vous changez de bloc). Vous pouvez observer trois points au début de chaque ligne qui indiquent que vous continuez à être dans un bloc.

Essayons maintenant quelques *vrais* exemples. Ouvrez la console et tapez ce qui suit en vous rappelant de presser la barre d'espace quatre fois au début des lignes qui commencent par « `print` » :

à saisir avec attention

```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
...     print(i)
...     print(i)
...
```

```
a
a
b
b
c
c
```

Après le second « `print` » appuyez sur la touche entrée sur la ligne blanche pour dire à la console que vous souhaitez finir le bloc. Cela affichera chaque élément deux fois.

L'exemple suivant va créer un message d'erreur :

erreur

```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
...     print(i)
...       print(i)
...
File <stdin>, line 3
print(i)
^
IndentationError: unexpected indent
```

La seconde ligne avec « `print` » a six espaces et non quatre sans avoir été introduite par deux points. Python n'aime pas cela car l'indentation doit rester la même au sein d'un même bloc.

Pense-bête

Si vous commencez un bloc de code avec quatre espaces vous devez continuer avec quatre espaces. De plus il est conseillé de toujours utiliser des indentations multiples de la première indentation choisie comme 4, 8, 12.
La plupart des programmeurs Python utilisent des indentations de quatre espaces. Pour faciliter l'éventuelle fusion de code, il est donc conseillé d'utiliser, vous aussi des indentations de quatre espaces.
De plus si vous utilisez IDLE comme éditeur (pas dans le Shell) l'appui sur la touche tabulation « ⇆ » insérera quatre espaces. Le shell IDLE ajoute automatiquement une indentation réalisée avec une tabulation. Le passage d'un bloc à un autre sur une ligne peut être fait en utilisant les flèches horizontales.

Vois un exemple qui met en œuvre deux blocs de code :

ne pas saisir

```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
```

```
...     print(i)
...     for j in maliste:
...         print(j)
...
```

Où sont les blocs dans ce code que va-t-il faire ? Il a deux blocs, le premier fait parti de la première boucle :

ne pas saisir
```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
...     print(i)                # Ces deux lignes sont
...     for j in maliste:       # le premier bloc
...         print(j)
...
```

Le deuxième bloc est la ligne « `print` » solitaire dans la seconde boucle :

ne pas saisir
```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
...     print(i)
...     for j in maliste:
...         print(j)            # Second bloc
...
```

Au passage vous pouvez observer des dièses « # » qui sont utilisés pour marquer des commentaires. Les commentaires sont du texte qui n'est pas pris en compte pour l'éxécution mais qui apporte des informations sur le code et permettent de le documenter (pour un usage ultérieur par exemple).

Pouvez-vous déduire ce que ce petit bout de code va faire ? Il va afficher les trois lettres de « `maliste` », mais combien de fois ? Si nous étudions chaque ligne nous pouvons probablement trouvez ce nombre de fois. Nous savons que la première boucle va parcourir tous les objets de la liste et exécuter les commandes du premier bloc. Donc elle va afficher une lettre puis lancer la seconde boucle. Cette boucle va elle aussi parcourir les éléments de la liste et exécuter les commandes du deuxième bloc. Ainsi nous pouvons comprendre ce qui est affiché quand le code est exécuté, ce sera un « `a` » suivi de « `a` », « `b` », « `c` » puis un « `b` » suivi de « `a` », « `b` », « `c` » et ainsi de suite.

Entre le code dans console et voyez par vous-même :

à saisir avec attention
```
>>> maliste = [ 'a', 'b', 'c' ]
>>> for i in maliste:
...     print(i)
...     for j in maliste:
...         print(j)
```

```
...
a
a
b
c
b
a
b
c
c
a
b
c
```

Et si nous faisions maintenant quelque chose de plus utile que juste afficher des lettres ? Rappelez-vous les calculs que nous avions faits au début de ce livre sur les économies que vous pourriez réaliser. Il s'agissait de calculer les économies si vous gagniez 10€ par semaine en faisant le ménage, 30€ en distribuant le journal et que vous dépensiez 10€.

Cela ressemblait à cela :

ne pas saisir

```
>>> (5 + 30 - 10) * 52
```

C'est à dire 5€+30€-10€ multiplié par 52 semaines dans l'année.

Il pourrait être utile de voir de combien vos économies augmentent durant l'année plutôt que de savoir ce qu'elles seront en toute fin d'année. Nous pouvons calculer cela avec un itérateur. Mais premièrement nous devons mettre ces nombres dans des variables :

à taper

```
>>> ménage = 5
>>> journaux = 30
>>> dépenses = 10
```

Nous pouvons faire le calcul original en utilisant ces variables :

à taper

```
>>> (ménage + journaux - dépenses) * 52
1300
```

Ou nous pouvons voir nos économies augmenter au cours de l'année en créant une autre variable appelée « `économies` » et l'utiliser dans une boucle :

à taper

```
>>> économies=0
>>> for semaine in range(1, 53):
...     économies = économies + ménage + journaux - dépenses
...     print('Semaine %s = %s€' % (semaine,économies))
...
```

À la première ligne la variable « économies » est créée avec pour valeur zéro car nous n'avons encore rien économisé. Si nous ne créons pas cette variable nous ne pourons pas dire à la première étape que les économie de la première semaine sont égales à celles d'avant qu'on débute. On pourrait écrire :

moche
```
>>> économies=économies + ménage + journaux - dépenses
>>> for semaine in range(2, 53):
...     économies = économies + ménage + journaux - dépenses
...     print('Semaine %s = %s€' % (semaine,économies))
...
```

Mais cela ferait du code à taper en plus, pour rien.

Les bons programmeurs ne tapent pas du code pour rien.

La deuxième ligne commence un itérateur (une boucle) qui va exécuter les commandes dans le bloc constitué de la troisième et de la quatrième ligne. À chaque itération (à chaque tour de la boucle) la variable « semaine » est chargée avec le nombre suivant de la série 1 à 52.

La troisième ligne est un peu plus compliquée. Simplement, pour chaque semaine nous voulons ajouter ce que nous avons économisé à nos « économies » totales. Pensez à la variable « économie » comme à une tirelire. En langage Python la troisième ligne veut dire : remplace le contenu de la variable « économies » par mes économies actuelles plus ce que j'ai gagné cette semaine.

Le symbole « = » est un bout de code astucieux pour dire : calcule ce qu'il y a à droite en premier puis garde le pour plus tard en utilisant le nom qui est à gauche.

Si la ligne contenant « économie=économie+... » vous semble trop difficile à comprendre, sachez que l'on peut aussi écrire :

à taper
```
>>> économies=0
>>> for semaine in range(1, 53):
...     économies += ménage + journaux - dépenses
...     print('Semaine %s = %s€' % (semaine,économies))
...
```

C'est à dire ajouter à « économie » le résultat de « ménage + journaux - dépenses ». La quatrième ligne est une instruction « print » légèrement compliquée. Elle affiche le numéro de la semaine et le montant total des économies réalisé pour cette semaine. Si cette ligne n'a pas beaucoup de sens pour vous, rafraichissez vos connaissances sur les chaînes en consultant la section 2.6 « Tours de chaînes » page 20.

Si vous exécutez le programme vous aurez comme résultat :

ne pas saisir
```
Semaine 1 = 25€
Semaine 2 = 50€
```

```
Semaine 3 = 75€
Semaine 4 = 100€
Semaine 5 = 125€
Semaine 6 = 150€
[...]
Semaine 49 = 1225€
Semaine 50 = 1250€
Semaine 51 = 1275€
Semaine 52 = 1300€
```

5.3 Tant que nous en sommes à parler de boucles…

L'itérateur n'est pas la seule sorte de boucle que nous pouvons faire en Python. Il y a aussi une boucle « tant que ». Alors que dans le cas d'un itérateur nous savons exactement quand nous allons sortir de la boucle, dans une boucle « tant que » nous ne savons pas forcément quand nous allons sortir de la boucle. Imaginez un escalier avec vingt marches. Vous savez que vous pouvez facilement monter vingt marches, vous pouvez employer un itérateur.

ne pas saisir

```
>>> for marche in range(0,20):
...     print(marche)
```

Maintenant imaginons que l'escalier permet de monter en haut d'une montagne. Vous pouvez être épuisé avant d'atteindre le sommet. Ou les conditions météorologiques peuvent devenir mauvaises vous obligeant à vous arrêter. C'est une boucle « tant que ».

ne pas saisir

```
>>> marche = 0
>>> continuer=True
>>> while continuer :
...     print(marche)
...     if marche==10000 :
...         continuer=False
...     elif fatigué():
...         continuer=False
...     elif mauvais_temps():
...         continuer=False
...     else:
...         marche += 1
```

Ne vous ennuyez pas à taper l'exemple ci-dessous car nous ne nous sommes pas ennuyés à créer les fonctions « `fatigué()` » et « `mauvais_temps()` »

Le mot *while* signifie « tant que » en anglais, *True* signifie « vrai » et *False* signifie « faux ».

Cet exemple montre les bases d'une boucle tant que. Tant que « `continuer` » est vrai le bloc de code sera exécuté. Dans le bloc nous affichons la valeur de « `marche` » puis nous

vérifions si nous sommes arrivés « `marche==10000` », si nous sommes fatigués ou si le temps est mauvais.

Les boucles tant que peuvent être représentées comme sur la figure 5.4.

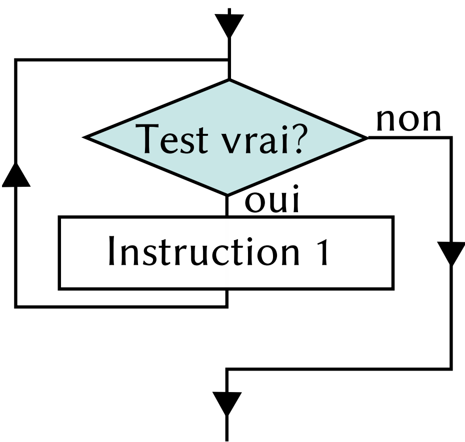


FIGURE 5.4 – Diagramme d'une boucle tant que

Cet algorithme [4] peut aussi s'écrire [5] :

ne pas saisir

```
>>> marche = 0
>>> while marche < 10001:
...     print(step)
...     if fatigué():
...         break
...     elif mauvais_temps():
...         break
...     else:
...         marche = marche + 1
```

Tant que « `marche` » vaut moins de 10001 (tant que nous ne sommes pas arrivés) le bloc de code sera exécuté. Dans le bloc nous affichons la valeur de « `marche` » puis nous vérifions si nous sommes fatigués ou si le temps est mauvais. Si c'est le cas l'instruction « `break` » nous fait sortir de la boucle (nous sautons hors de la boucle à la ligne qui suit le bloc). Sinon nous ajoutons un à marche puis la condition de la boucle est contrôlée à nouveau.

Le mot *break* signifie casser en anglais, *break out*, s'évader.

Les étapes d'une boucle « tant que » sont simplement :
– vérifier la condition ;
– exécuter le code dans le bloc ;
– répéter.

Le plus souvent, une boucle tant que prendra en compte plusieurs conditions à la fois :

ne pas saisir

```
>>> marche = 0
>>> while marche < 10001 and en_forme() and beau_temps():
```

4. Un algorithme est un ensemble d'opérations ordonné qui permet de réaliser une opération plus complexe.
5. Ces exemples se veulent didactiques, nous verrons plus tard comment ces écritures peuvent être condensées.

```
...     print(step)
...     marche = marche + 1
```

Ou moins imagé mais que vous pouvez saisir tout de suite :

à taper

```
>>> x = 45
>>> y = 80
>>> while x < 50 and y < 100:
...     x = x + 1
...     y = y + 1
...     print(x, y)
```

Avant cette boucle nous créons une variable « x » initialisé à 45 et une variable « y » initialisé à 80. Deux conditions doivent être vraies pour que le bloc de la boucle soit exécuté : x vaut moins de 50 et y moins de 100. Tant que les deux conditions sont vraies le bloc de code est exécuté. Ce bloc ajoute un aux deux variables et les affiche. Le résultat est juste :

ne pas saisir

```
46 81
47 82
48 83
49 84
50 85
```

Peut-être vous demandez-vous pourquoi ces nombres et seulement ces nombres sont imprimés ?

Nous commençons à compter à partir de 45 pour « x » et de 80 pour « y ». Puis nous incrémentons (ajoutons un) à chaque variable à chaque « tour » de boucle. Le contrôle des conditions vérifie que x fait moins de 50 et y fait moins de 100. Après avoir parcouru la boucle cinq fois (en ajoutant un à chaque variable) la valeur de x atteint cinquante. Maintenant la première condition « x < 50 » est fausse donc Python arrête de boucler.

Un autre usage courant des boucle tant que est de créer des boucles semi-éternelles. Il s'agit de boucles qui s'exécutent pour toujours ou du moins jusqu'à ce quelque chose arrive dans le code qui va l'arrêter. Par exemple :

ne pas saisir

```
>>> while True:
...     plein de code ici
...     plein de code ici
...     plein de code ici
...     if une_condition == True:
...         break
```

La condition pour cette boucle tant que est juste « True » (vrai). Ainsi le code du bloc va toujours s'exécuter (c'est pourquoi la boucle est dite éternelle ou infinie). Néanmoins,

si la variable « `une_condition` » devient vraie alors le code sortira de la boucle grâce à l'instruction « `break` ». Vous trouverez un meilleur exemple de ce type de boucle infinie dans l'annexe C (dans la section à propos du module « `random` ») mais vous devriez attendre d'avoir lu le prochain chapitre avant d'y jeter un coup d'œil.

5.4 À vous de jouer

Dans ce chapitre nous avons vu comment utiliser des boucles pour réaliser des actions répétitives. Nous avons utilisé des blocs de code à l'intérieur de boucle pour les taches à répéter.

Vous trouverez des pistes de réponses dans la section A.3.

5.4.1 Exercice 1

Que pensez vous qu'il va arriver avec le code suivant ?

à taper

```
>>> for x in range(0, 20):
...         print('x vaut %s' % x)
...         if x < 9:
...                 break
```

5.4.2 Exercice 2

Un lac est envahi par des nénuphars issus d'un jardin. Le premier jour un seul nénuphar est présent dans le lac. Tous les jours le nombre de nénuphars double (est multiplié par deux). Sachant que le lac sera complètement envahi quand il y aura plus de mille nénuphars, combien de jours faut-il pour arriver à ce stade ?

6 Une sorte de recyclage

6.1 De l'importance du recyclage

Pensez à tous les déchets que vous créez chaque jour. Les bouteilles d'eau et d'autres boissons, les paquets de chips, les emballages, les sacs en plastique, les sacs de course, les journaux, les magazines et ainsi de suite.

Maintenant pensez à ce qui arriverait si tous ces déchets étaient mis en pile sur votre pas de porte.

Bien sûr vous recyclez probablement autant que faire ce peut, ce qui est heureux, car personne n'aime devoir escalader un tas d'ordure pour aller à l'école. Ainsi les bouteilles en verres mises dans la poubelle pour le recyclage sont fondues et transformées en nouvelles bouteilles et saladiers. Le papier est transformé en papier recyclé. Le plastique est transformé en plastique dur et en polaires. C'est pourquoi votre pas de porte ne disparaît pas sous des tonnes de détritus.

Nous tendons à réutiliser des objets que nous créons plutôt que de creuser un gouffre sur la Terre pour fabriquer les mêmes choses encore et encore.

Recycler ou réutiliser dans le monde la programmation est aussi important. Votre programme ne va pas disparaître sous une pile d'immondices, mais si vous ne réutilisez pas une partie de ce que vous faites, vous ne pourrez plus utiliser vos doigts qui seront trop douloureux à cause de toute cette dactylographie.

Il y a de nombreuses manières différentes de réutiliser du code en Python (et dans les langages de programmation en général). Nous en avons déjà aperçu certaines avec le module « turtle » et avec les fonctions « print » et « range ».

Les fonctions permettent de réutiliser du code. Ainsi si vous pouvez écrire du code une fois puis l'utiliser dans vos programmes encore et encore. Commençons par essayer un exemple simple de fonction :

à taper
```
>>> def mafonction(un_nom):
...     print('Bonjour %s.' % un_nom)
...
```

La fonction ci-dessus a pour nom « mafonction » et pour paramètre « un_nom ». Un paramètre est une variable qui est seulement disponible à l'intérieur du « corps » de la fonction. C'est à dire dans le bloc qui est directement après la ligne qui commence par « def ». Au cas où vous vous poseriez la question « def » est utilisé pour le verbe *to define* c'est à dire définir. Vous pouvez utiliser une fonction en l'appelant par son nom (défini juste après le « def ») accolé à des parenthèses qui entourent la valeur passée en paramètre :

à taper
```
>>> mafonction('Thaïs')
Bonjour Thaïs.
>>> mafonction('Fripon')
Bonjour Fripon.
```

Nous pouvons définir une fonction pour qu'elle prenne plusieurs paramètres, par exemple deux :

à taper
```
>>> def mafonction(prénom, nom):
...     print('Bonjour %s %s.' % (prénom, nom))
...
```

Puis, nous pouvons l'utiliser d'une manière similaire :

à taper
```
>>> mafonction('Élodie','Dupont')
Bonjour Élodie Dupont.
```

Nous pouvons aussi créer des variables et appeler la fonction avec des variables utilisées comme paramètres :

à taper
```
>>> prénom_copain = 'Caroline'
>>> nom_copain = 'Robertson'
>>> mafonction(prénom_copain, nom_copain)
Bonjour Caroline Robertson.
```

Nous pouvons, de plus, utiliser la déclaration « `return` » (retourner) pour retourner des valeurs :

à taper
```
>>> def économies(ménage, journaux, dépenses):
...     return ménage + journaux - dépenses
...
>>> print(économies(10, 10, 5))
15
```

Cette fonction prend trois paramètres puis additionne les deux premiers avant de soustraire le dernier (dépenses). Le résultat est alors retourné et peut utilisé comme paramètre d'une autre fonction (ici « `print` »).

Ce résultat peut aussi être utilisé pour initialiser une variable de la même manière que n'importe quelle valeur.

à taper
```
>>> mes_économies = économies(20, 10, 5)
>>> print(mes_économies)
25
```

Néanmoins, une variable que nous utilisons à l'intérieur du corps d'une fonction n'est pas accessible (utilisable) une fois que l'exécution est finie :

erreur
```
>>> def test_variable():
...     a = 10
...     b = 20
...     return a * b
...
>>> print(test_variable())
200
>>> print(a)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Dans l'exemple ci-dessus nous créons une fonction « `test_variable` » qui multiplie deux variables « `a` » et « `b` » puis retourne le résultat. Si nous appelons cette fonction comme paramètre de « `print` » nous avons pour réponse 200. Mais si nous appelons « `print` » pour afficher le contenu de « `a` » nous aurons le message d'erreur « `'a' is not defined` » ('a'

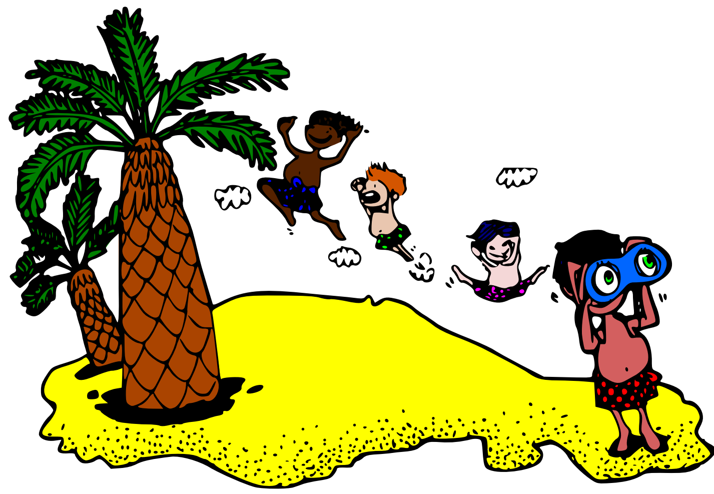
n'est pas défini). En programmation, l'endroit où est utilisable une variable est appelé la « portée » de cette variable.

Pensez à une petite île au milieu de l'océan qui serait trop éloignée de tout pour qu'il soit possible de quitter l'île à la nage. Occasionnellement un avion vole au dessus et largue des feuilles de papier sur l'île (les paramètres de la fonction) que les habitants collent ensembles pour constituer un message. Ils le mettent, ensuite, dans une bouteille et jettent la bouteille à la mer (la valeur retournée). Ce que font les insulaires (ou les îliens en Bretagne) sur l'île et combien sont-ils pour faire le message, ne fait aucune différence pour la personne qui va ramasser la bouteille et lire le message qui était dedans. C'est sûrement la manière la plus simple de se représenter ce qu'est la portée. Mais il y a un petit problème avec cette représentation. Un des insulaires possède des jumelles très puissantes (et l'île est proche de la côte) il peut ainsi voir le continent. Il peut même voir ce que font les continentaux et cela peut changer le message que les insulaires écrivent.

à taper

```
>>> x = 100
>>> def test2_variable():
...     a = 10
...     b = 20
...     return a * b * x
...
>>> print(test2_variable())
20000
```

Ainsi, même si les variables a et b ne peuvent pas être utilisée en dehors de la fonction, la variable x qui a été créée hors de la fonction est utilisable à l'intérieur. Essayez de penser à l'insulaire avec des jumelles et il est possible que cette image vous aide, un peu, à bien comprendre les portées.



L'itérateur que nous avions créé plus tôt pour afficher les économies durant une année peut facilement être intégré dans une fonction.

à taper

```
>>> def économies_annuelles(ménage, journaux, dépenses):
...     économies = 0
```

```
...     for semaine in range(1, 53):
...         économies += ménage + journaux - dépenses
...         print('Semaine %s = %s' % (semaine, économies))
...
```

Nous pouvons alors utiliser la fonction « `économies_annuelles()` » :

à taper
```
>>> économies_annuelles(10, 10, 5)
Semaine 1 = 15
Semaine 2 = 30
Semaine 3 = 45
Semaine 4 = 60
Semaine 5 = 75
Semaine 6 = 90
Semaine 7 = 105
Semaine 8 = 120
Semaine 9 = 135
Semaine 10 = 150
[...]
```

Puis, nous pouvons l'utiliser encore :

à taper
```
>>> économies_annuelles(25, 15, 10)
Semaine 1 = 30
Semaine 2 = 60
Semaine 3 = 90
Semaine 4 = 120
Semaine 5 = 150
[...]
```

Ce qui est plus pratique que de retaper l'itérateur à chaque fois que vous voulez essayer des valeurs différentes. Les fonctions peuvent, par ailleurs, être groupées ensemble dans des « modules » qui sont *très* utiles en Python.

Plus à propos des modules, bientôt...

6.2 Modules

Nous avons déjà vu quelques différentes manières de réutiliser du code. L'une d'elles est la fonction. Nous pouvons créer nous-mêmes des fonctions au sein de nos programmes ou utiliser les fonctions de base de Python comme « `print` », « `range` » « `int` » ou « `str` ».

Une autre méthode est d'utiliser des fonctions associées à des objets que nous appelons en utilisant le point « . » [1].

1. Nous verrons ultérieurement les méthodes

Il est aussi possible d'utiliser des modules qui sont une manière de grouper des fonctions et des objets ensembles de manière utile.

Un exemple de module est le module « `time` », c'est à dire « temps » en anglais :

à taper
```
>>> import time
```

La commande « `import` » est utilisée pour dire à Python que nous voulons accéder à un module. Dans l'exemple ci-dessus, nous disons que nous voulons utiliser le module « `time` ». Nous pouvons alors appeler les fonctions et les objets qui sont disponibles dans ce module en utilisant le symbole du point « `.` », comme dans l'exemple ci-après :

à taper
```
>>> print(time.localtime())
time.struct_time(tm_year=2009, tm_mon=3, tm_mday=19, tm_hour=23,
tm_min=22, tm_sec=4, tm_wday=3, tm_yday=78, tm_isdst=0)
```

La fonction « `localtime` » est une fonction interne au module « `time` » qui retourne l'heure et la date actuelle, découpées en année (*year*), mois (*month*), jour dans le mois « `month day` », heure (*hour*), minutes (*minutes*), secondes (*seconds*), jour dans la semaine à compté de `0` pour lundi(*week day*), jour dans l'année (*year day*), et une indication d'un éventuel ajustement de l'heure).

Ces éléments sont stockés dans un n-uplet (voir la section 2.8 page 25). Nous pouvons utiliser une autre fonction du module pour avoir quelque chose de plus compréhensible :

à taper
```
>>> t = time.localtime()
>>> print(time.asctime(t))
Sat Nov 18 22:37:15 2006
```

Nous pouvons aussi utiliser les deux fonctions chaînées sur la même ligne :

à taper
```
>>> print(time.asctime(time.localtime()))
Sat Nov 18 22:37:15 2006
```

Tout cela est bien beau mais la date est en anglais ! Cela est du au fait que la commande « `time.asctime()` » fournit la date en caractères ASCII (American Standard Code for Information Interchange). Comme vous pouvez le voir Tableau 6.1 de nombreux caractères usuels en français et dans de nombreuses autres langues ne sont pas présents dans cette table. Néanmoins l'usage de l'ASCII perdure car il garantit la bonne compréhension par tous les types d'ordinateurs ou assimilés.

Heureusement, Python sait aussi parler d'autres langues dont le français. Le module « `locale` » permet de localiser un certain nombre de fonctions (c'est à dire d'utiliser la langue choisie par l'utilisateur) :

à taper
```
>>> import locale
>>> locale.setlocale(locale.LC_ALL,'')
```

<table>
<tr><td>␣</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td></tr>
<tr><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td></tr>
<tr><td>></td><td>?</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr>
<tr><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td><td>P</td><td>Q</td></tr>
<tr><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td></tr>
<tr><td>\</td><td>]</td><td>^</td><td>_</td><td></td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td></tr>
<tr><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td></tr>
<tr><td>o</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td></tr>
<tr><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>~</td><td></td><td></td><td></td><td></td></tr>
</table>

TABLE 6.1 – caractères ASCII imprimables

Nous indiquons ici à Python d'employer la langue utilisée par votre système d'exploitation (Windows dans votre cas) qui est probablement le français.

Puis, nous pouvons utiliser la fonction « `time.strftime()` » qui permet un grand nombre de mises en forme [2] :

à taper

```
>>> temps=time.strftime('Nous somme le %A %d %B %Y et il est %H h %M min et %S s')
>>> print(temps)
Nous somme le jeudi 19 mars 2009 et il est 23 h 28 min et 24 s
```

Supposons que vous voulez demander à quelqu'un d'entrer une valeur. Vous pouvez faire cela en utilisant la fonction « `print` » et le module « `sys` » importé de la même manière que nous avions importé le module « `time` » :

à taper

```
>>> import sys
```

À l'intérieur du module « `sys` » un objet appelé « `stdin` » pour *standard input*.

Le mot *standard input* est le terme anglais pour « entrée standard ».

L'objet « `stdin` » a une méthode (ou fonction) vraiment utile appelé « `readline` » qui est utiliser pour lire (*read*) une ligne (*line*) de texte que quelqu'un a tapé sur le clavier (après qu'il ait tapé sur la touche « `Entrée` ». Vous pouvez tester « `readline` » en entrant la commande suivante dans la console Python :

à taper

```
>>> print(sys.stdin.readline())
```

Si vous tapez des mots et que vous pressez la touche entrée ce que vous venez de taper sera réaffiché. Repensons au code que nous avions écrit plus tôt en utilisant le test si :

ne pas saisir

```
>>> if âge >= 10 and âge <= 13:
...     print('Vous avez %s ans.' % âge)
... else:
...     print('Heu ?')
```

2. La fonction « `time.strftime()` » est décrite de manière plus détaillée en annexe.

Plutôt que de créer une variable âge et de l'initialiser avec une valeur avant de l'utiliser nous pouvons entrer une valeur au moment de la création de celle-ci au lancement d'une fonction :

à taper
```
>>> def votre_âge(âge):
...     if âge >= 6 and âge <= 13:
...         print('Vous avez %s ans.' % âge)
...     else:
...         print('Heu ?')
...
```

Notre fonction « `votre_âge()` » peut être appelée en passant un nombre comme valeur en paramètre. Nous allons tester que cela fonctionne correctement :

à taper
```
>>> votre_âge(9)
Heuh ?
>>> votre_âge(10)
Vous avez 10 ans.
```

Maintenant que nous savons qu'il n'y a pas de problème avec notre fonction, nous pouvons, avec « `readline` », demander à quelqu'un d'entrer son âge.

à taper
```
>>> def votre_âge():
...     print('S'il vous plait entrez votre âge :')
...     âge = int(sys.stdin.readline())
...     if age >= 6 and age <= 13:
...         print('Vous avez %s ans.' % âge)
...     else:
...         print('Heu ?')
...
```

Comme « `readline` » retourne ce que la personne a entré comme texte, c'est à dire une chaîne, nous avons besoin de convertir cette chaîne en nombre avec la fonction « `int` ». Une fois converti en nombre nous pouvons utiliser la valeur stockée en utilisant « `âge` » dans un test si (vous pourrez trouver des informations à ce sujet section 4.6, page 44).

Maintenant essayons notre fonction, appelez « `votre_âge()` » sans paramètre et le texte que vous aviez indiqué va apparaître :

à taper
```
>>> votre_âge()
S'il vous plait entrez votre âge :
6
Vous avez 6 ans.
>>> votre_âge()
S'il vous plait entrez votre âge :
15
Heu ?
```

Deux choses sont importantes à noter :
- « `readline` » renvoie toujours une chaîne de caractères ;
- notre fonction ne fonctionne que si vous entrez des entiers (nous verrons plus tard comme régler ce problème).

Les modules « `sys` » et « `time` » sont juste l'un des nombreux modules inclus dans Python. Vous trouverez plus d'information sur certains modules Python (mais pas tous) en Annexe D.

6.3 À vous de jouer

Dans ce chapitre nous avons vu comment recycler des choses en Python en utilisant des fonctions et des modules. Nous verrons dans le chapitre suivant comment sauver vos programmes pour pouvoir les réutiliser ou rendre utilisables par d'autres personnes.

Nous avons évoqué la « portée » d'une variable, c'est à dire que les variables extérieures aux fonctions peuvent être utilisées à l'intérieur de celles-ci, alors que les variables définies à l'intérieur des fonctions ne pouvaient pas être utilisées à l'extérieur de celles-ci.

Nous avons aussi appris à créer des fonctions avec « `def` ».

Vous trouverez des pistes de réponses dans la section A.4.

6.3.1 Exercice 1

Dans le deuxième exercice de la section 5.4, nous avions créé un itérateur pour calculer la prolifération des nénuphars jusqu'à ce qu'il y ait mille nénuphars dans l'étang. Essayez de créer une fonction qui prend comme paramètres :
- le nombre initial de nénuphar ;
- le coefficient de reproduction quotidien (par combien le nombre de nénuphars est multiplié tout les jours).

Vous pouvez appeler cette fonction de la manière suivante « `calcul_nénuphars(2, 1.5)` ».

6.3.2 Exercice 2

Prenez la fonction que vous venez de créer et modifiez la de manière à pouvoir calculer le nombre de nénuphars pour différents nombres de jours pour lesquels nous voulons un résultat. Vous pouvez appeler cette fonction de la manière suivante « `calcul_nénuphars(2, 1.5, 5)` »

6.3.3 Exercice 3

Plutôt qu'une simple fonction dans laquelle nous passons les valeurs en paramètres, nous pouvons faire un mini-programme qui demande les valeurs en utilisant la fonction « `sys.stdin.readline()` ».

Ces valeurs seront alors passées à la fonction « `calcul_nénuphars` » pour réaliser les calculs.

De manière à créer cette fonction et si nous voulons pouvoir entrer des nombres à virgule, il convient de remplacer la fonction « `int` » par la fonction « `float` » dont nous n'avons pas encore parlé. Cette fonction convertit une chaîne en nombre dit à virgule flottante (nombre

rationnel dont nous avons parlé brièvement au chapitre 2 page 11). Les nombres à virgule sont représentés en Python en utilisant un point « `.` » comme « `20.3` » pour 20,3 ou « `2541.933` » pour 2541,933. Vous pouvez voir à nouveau l'exemple proposé dans la section précédente.

7 Un court chapitre à propos des fichiers

7.1 Sauvegarde des programmes

Vous savez déjà probablement ce qu'est un fichier (*file* en anglais pour l'informatique). Dans votre classe vous avez probablement des fiches — par exemple de lecture, de botanique— qui sont réunies ensembles au sein de fichiers. Ces fichiers sont réunis dans des boites pour faciliter leur utilisation.

C'est assez similaires en informatique, nous mettons les informations dans des fichiers que nous rangeons dans des répertoires (appelés aussi dossiers), la Figure 7.1 montre cette organisation.

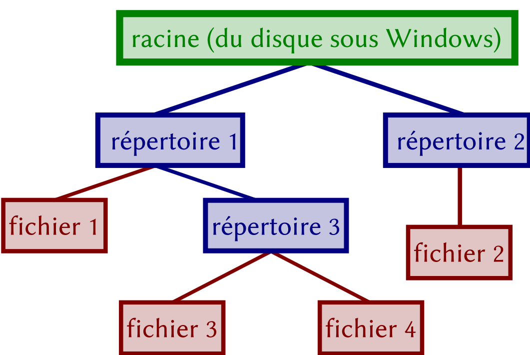


FIGURE 7.1 – Organisation des fichiers et des répertoires

Rappelez-vous au premier chapitre nous avions créé un fichier « `bonjour.py` » et nous ne pouvions pas voir le résultat de l'exécution de programme.

Nous allons maintenant voir comment utiliser ce type de fichier. Premièrement cliquez avec le bouton droit de la souris sur le fichier bonjour.py et choisissez « `éditer avec IDLE` » comme sur la Figure 7.2.

Deux fenêtres s'ouvrent alors : l'éditeur et le shell comme montrés sur la Figure 7.3.

Selon vos préférences vous pouvez afficher l'une ou l'autre des fenêtres .

En tant que livre je préfère une présentation de type une page à gauche et une page à droite. Pour ce faire je clique droit sur la barre des tâches et je choisis mosaïque verticale.

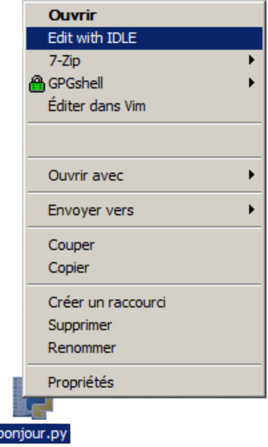


FIGURE 7.2 – Édition avec IDLE

FIGURE 7.3 – Fenêtres d'IDLE

Comme vous le voyez, le code que vous aviez entré est affiché dans la fenêtre « Bonjour.py ». Sélectionnez cette fenêtre puis appuyez sur la touche « `F5` » ou cliquez sur « `Run` » (courir en anglais normal, exécuter en informatique) puis « `run module` ». Le résultat s'affiche dans le shell.

Comme vous le voyez vous pouvez éditer, sauvegarder et réutiliser le code que vous réalisez.

Modifier le code comme suit :

à taper

```
import sys

print("Bonjour le monde")
sys.stdin.readline()
```

Puis lancez le avec la touche « `F5` » l'éditeur va vous demander de sauvegarder le fichier pour pouvoir l'exécuter, répondez oui car c'est ce que nous voulons. Le programme s'exécute et attend que vous appuyez sur entrée. Plus intéressant vous pouvez double-cliquer sur le fichier « `bonjour.py` ». Le message s'affiche et reste affiché.

Vous pouvez, si vous le souhaitez, compléter ce programme et le montrer à votre entourage. Par exemple, vous pouvez poser une question et composer une réponse à partir de celle-ci.

à taper

```
import sys

print("Bonjour, quel est votre nom ?")
nom=sys.stdin.readline()[:-1]
print("Bonjour, %s. Comment allez vous ?" % nom)
état=sys.stdin.readline()[:-1]
print("Moi aussi je vais %s." % état)
print("Appuyez sur entrée pour quitter.")
sys.stdin.readline()
```

Pour information, le « `[:-1]` » indique à Python de prendre tous les caractère rentrés moins un. Le dernier caractère correspond au retour à la ligne créé par la touche entrée.

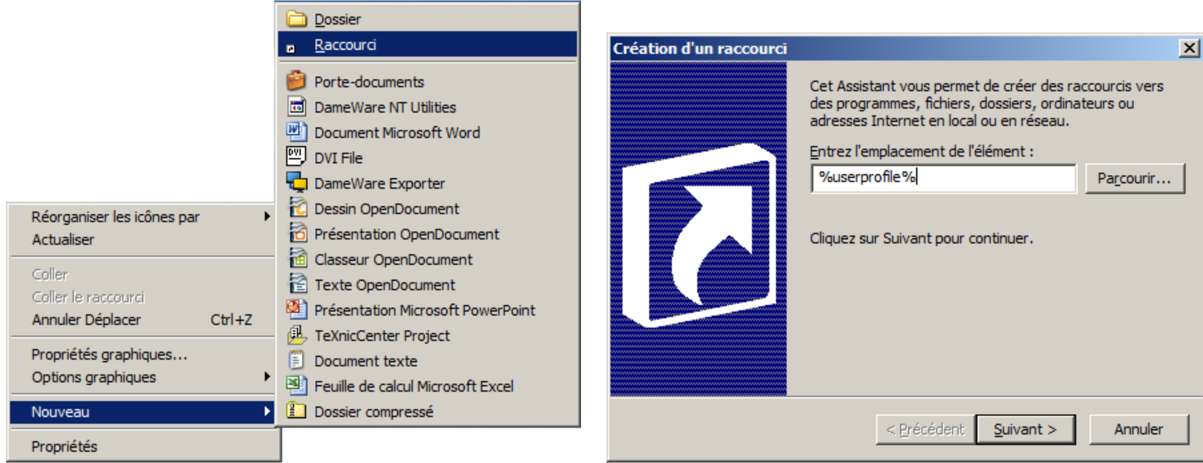
7.2 Si ça ne passe pas la fenêtre, passons par le soupirail

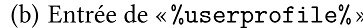
Microsoft a décidé au cours des évolutions de son système d'exploitation de changer les endroits (répertoires) utilisés pour le bureau (ce qui affiché une fois l'ordinateur démarré) et les documents. Ces répertoires dont le nom était traduit, ont maintenant toujours des noms anglais.

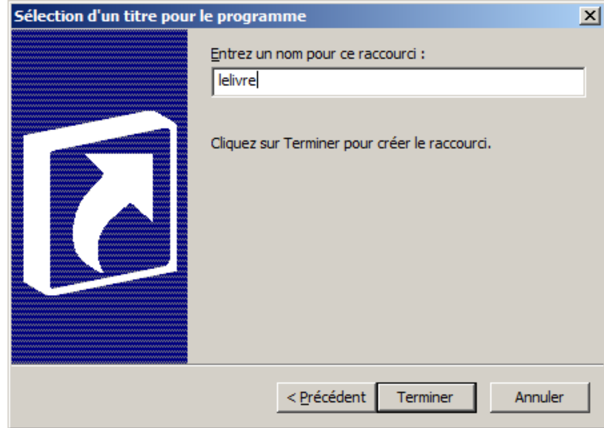
De plus certains chemins simples d'accès comme « `C:\` » ne sont plus toujours accessibles pour des raisons de sécurité.

Il existe des modules qui permettent à Python de connaitre le chemin vers votre bureau ou votre répertoire personnel sans programmation mais ils nécessiteraient une installation supplémentaire. Or nous ne voulons pas déranger vos parents (ou assimilés) à nouveau... Ils sont tellement bien tranquilles sur le canapé.

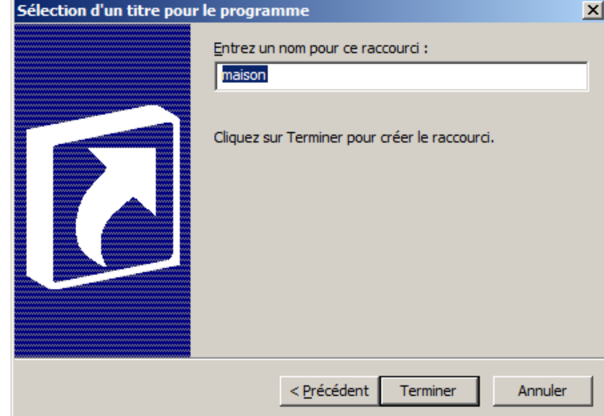
Nous pourrions, sans programmation, créer un raccourci sur le bureau pour que vous puissiez accéder facilement à votre répertoire personnel qui contient le « `bureau` » et « `mes documents` ». Cette solution est présentée pour information Figure 7.4 page 76. Mais cette solution est assez longue et finalement assez éloignée de mon sujet.



(a) Clic droit sur le bureau (b) Entrée de « %userprofile% »



(c) Nom du raccourci par défaut



(d) Entrée de « maison »

(e) Le raccourci sur le bureau

FIGURE 7.4 – Étapes de la création d'un raccourci vers le répertoire personnel

C'est pourquoi je vous propose de retrouver le chemin de votre bureau avec Python. Dans le shell ou dans l'éditeur, faites « Ctrl+N » ou « file » puis « New Window », une nouvelle fenêtre de l'éditeur s'ouvre alors.

Vous pouvez identifier la variante du système d'exploitation que vous utilisez :

à taper

```
import os,platform

print(platform.release())
print(os.environ['USERPROFILE'])
```

Après avoir tapé F5 et sauvegardé le fichier (avec l'extension « .py ») vous devriez avoir quelque chose comme :

résultat

```
XP
C:\Documents and Settings\lelivre
```

Nous venons d'obtenir la version de Windows et le chemin vers votre répertoire personnel. Utilisons alors ces données pour avoir le chemin vers votre bureau :

à taper

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")
```

Exécutons ce module : il ne se passe rien ! C'est parce que le fichier que nous venons de créer est un module qui introduit une fonction « `chemin_bureau()` » mais ne la lance pas. Nous devons modifier légèrement le fichier pour lancer la fonction et, par exemple, afficher le résultat :

à taper

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

print(chemin_bureau())
```

Le résultat que vous devez obtenir doit ressembler à ce qui suit :

résultat

```
C:\Documents and Settings\lelivre\Bureau
```

Notez que mon nom d'utilisateur est « `lelivre` », le votre est sûrement votre prénom.

Examinons ce que Python vient de faire. En premier Python importe deux modules « `os` » et « `platform` ». Le module « `platform` » apporte des informations sur la plateforme qui sert à lancer les programmes (version du système d'exploitation par exemple). Le module « `os` » apporte une interface pour communiquer avec le système d'exploitation, *operating system* en anglais.

Puis une fonction « `chemin_bureau` » est créée qui ne prend aucun paramètre en entrée. Python regarde alors le type de Windows utilisé puis selon le résultat retourne le chemin vers le bureau au format approprié en joignant le chemin vers votre répertoire personnel et la chaîne utilisé par votre version de Windows pour identifier le bureau.

Enfin nous avons indiqué à Python que nous voulions qu'il affiche avec « `print` » le résultat de la commande « `chemin_bureau` ».

Dans un programme plus important on crée généralement une fonction « `main` » qui sera exécutée en premier. Cette convention facilite la lecture d'un programme par d'autres personnes.

7.3 Un peu de lecture, maintenant ?

Quand Python est installé sur votre ordinateur, un tas de fonctions et de modules sont aussi installés. Certaines fonctions sont disponibles par défaut. La fonction « `range` » que nous avons déjà vue est de ce type. nous allons maintenant regarder la fonction « `file` » que nous n'avons pas encore utilisée.

Pour voir comment les fichiers sont utilisés ouvrez le bloc-notes, tapez quelques lignes et sauvez le fichier sur votre bureau :

1. cliquez sur le menu fichier puis enregistrer (ou simplement faites « `Ctrl+S` ») ;
2. cliquez sur Bureau ;
3. dans le champ « `nom du fichier` » où il y a écrit « `*.txt` », tapez « `test.txt` ».

Ouvrez le fichier qui contient la fonction « `chemin_bureau` » (avec un clic droit « `edit with IDLE` » et essayez ce qui suit à la fin du fichier :

à taper

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'test.txt'))
print(fichier.read())
```

Lancez le programme (avec « `F5` ») et le contenu du fichier que vous venez de créer devrait s'afficher dans le shell.

Que peut bien faire ce code ? La première ligne importe les modules « `os` » et « `platform` ». Les lignes 3 à 7 définissent la fonction pour le chemin vers le bureau. La ligne 9 ouvre le fichier que vous venez tout juste de créer en utilisant le chemin vers le bureau et le nom du fichier.

La fonction « `open` » crée un type de valeur particulier (appelé génériquement objet) qui représente le fichier. Dans le cas présent l'objet créé est de type « `file` » (c'est à dire fichier). La variable « `fichier` » pointe vers l'objet de type « `file` » que nous venons de créer.

La variable « `fichier` » n'est pas le fichier en lui-même mais une sorte de gros doigt qui pointe vers le fichier.

La dernière ligne appelle la fonction « `read` » propre aux objets de type « `file` ». La fontion « `read` » lit le contenu du fichier et « `print` » l'affiche dans le shell. Parce que la variable « `fichier` » pointe vers un objet qui contient des informations et des fonctions (appelées méthodes pour des objets), nous devons utiliser le point « `.` » pour utiliser ces méthodes disponibles dans l'objet.

> L'Annexe B (vers la fin de ce livre) contient plus d'information sur les commandes intégrées à Python.

7.4 Écrivons En Python ! (rime faible)

Nous avons déjà créé un objet « `file` » dans la section précédente, en utilisant Python :

ne pas saisir

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'test.txt'))
print(fichier.read())
```

Un objet « `file` » n'a pas seulement une fonction « `read` ». Après tout, un fichier ne serait pas très utile si on pouvait seulement prendre des fiches mais ne jamais en rajouter. Nous pouvons créer un nouveau fichier vide en passant un autre paramètre à la fonction « `open` ».

Contrôlez d'abord qu'il n'existe pas de fichier « `nouveau.txt` » sur votre bureau. Si cela est le cas vous pouvez soit le renommer (par exemple en « `ancien.txt` » , soit changer « `nouveau.txt` » en ce que vous voulez dans les exemples qui suivent. Enfin, une fois ce contrôle effectué, vous pouvez essayer le code suivant :

à taper par exemple en reprenant l'existant

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'),'w')
```

Exécutez ce programmes puis regardez sur votre bureau, un fichier « `nouveau.txt` » vient d'être créé. Le paramètre « `w` » (pour *write*, écrire en anglais) est utilisé pour indiquer à Python que nous voulons écrire dans un (nouveau) fichier. Si le fichier n'existe pas il est créé, s'il existe son contenu est détruit.

Nous pouvons maintenant utiliser la fonction « `write` ». Une fois que nous aurons écrit il convient d'indiquer à Python que nous ne désirons plus écrire dans le fichier en utilisant la fonction « `close` » (*close* signifie fermer en anglais). Notez au passage qu'il est de bon ton d'utiliser aussi la fonction « `close` » quand on finit d'utiliser un fichier.

à taper par exemple en reprenant l'existant

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'),'w')
fichier.write('''Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !''')
fichier.close()
```

Si vous ouvrez le fichier avec votre éditeur de texte favori (le bloc note à priori) vous verrez qu'il contient le texte :

contenu de nouveau.txt

```
Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !
```

Ou mieux encore nous pouvons utiliser Python pour lire ce texte :

à taper par exemple en reprenant l'existant

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'),'w')
fichier.write('''Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !''')
fichier.close()

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'))
print(fichier.read())
fichier.close
```

À ce point quelque chose pourrait vous surprendre : le résultat de la commande ci-dessus reste nos trois lignes. Or nous avons écrit *deux* fois dans le fichier. Que s'est-il passé ?

L'ouverture en mode « `w` » indique que nous souhaitons écrire en écriture à partir de zéro. Le contenu initial du fichier a été détruit. Nous pouvons modifier le « `w` » en « `a` » pour *append* qui signifie ajouter en anglais.

à taper par exemple en reprenant l'existant

```
import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['USERPROFILE'],"Bureau")
    else :
        return os.path.join(os.environ['USERPROFILE'],"Desktop")

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'),'a')
fichier.write('''Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !''')
fichier.close()

fichier = open(os.path.join(chemin_bureau(),'nouveau.txt'))
print(fichier.read())
fichier.close
```

Si nous exécutons ce programme le résultat devrait être :

résultat en mode ajout

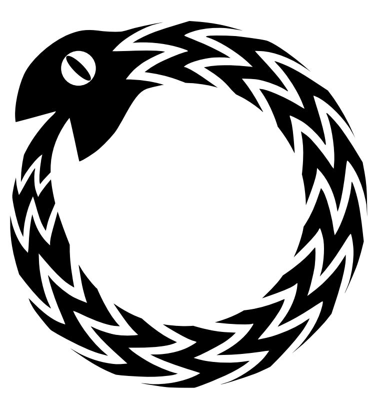
```
Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !
```

Bizarrement le résultat est sur cinq lignes et non pas six (deux fois trois). En fait quand nous saisissons le texte sur plusieurs lignes le caractère entrée que nous ne voyons pas est pris en compte. La dernière ligne ne finissait pas le caractère entrée et la première ne commençait pas par celui-ci. Nous pouvons modifier la chaîne pour que les nouvelles lignes commencent par un retour à la ligne :

à taper

```
fichier.write('''
Ceci est un fichier de test.
Ce fichier s'étend sur plusieurs lignes.
Trois en fait !''')
```

Notez bien que pour alléger la notation l'ensemble du code de votre fichier n'a pas été repris. Par ailleurs et pour information, le retour à la ligne peut aussi être saisi « `\n` ». Cette notation est simple à retenir : « `\n` » comme **n**ouvelle ligne.



8 Tortues à profusion

8.1 Le retour de la tortue

Revenons au module tortue que nous avions commencé à examiner au chapitre 3 page 29. Nous allons examiner tout un ensemble de fonctions du module turtle.

Vous vous demandez peut-être comment avoir des informations sur toutes ces commandes, en fait Python inclut une aide interne (sans parler de l'aide disponible dans le menu démarrer). Vous pouvez faire « `dir(turtle)` » pour avoir la liste de toutes les commandes incluses dans turtle. La commande « `dir` » est disponible pour tous les modules. De plus les modules bien développés incluent une aide en ligne par exemple « `help(turtle.Pen())` ». Cette aide est en anglais, malheureusement pour les francophones. Néanmoins, cela pourrait être pire dans une langue dont vous ne comprendriez même pas les caractères. Vous pouvez essayer ces commandes dans la console ou le shell :

à taper

```
>>> dir(str)
>>> help(str.isalpha)
>>> import turtle
>>> dir(turtle)
>>> help(turtle.Pen)
```

Pour information, la méthode « `isalpha` » teste une chaîne pour savoir si elle est composée de lettres uniquement et renvoie vrai (`True`) ou faux (`False`).

Revenons, à nouveau, sur le module « `turtle` ». Rappelez-vous, pour afficher une zone de dessin et dessiner dessus, nous avions besoin d'importer le module « `turtle` » et de créer un objet « `Pen` » :

ne pas saisir

```
>>> import turtle
>>> tortue = turtle.Pen()
```

À l'époque nous fermions la fenêtre en lançant « `turtle.bye()` ».

Maintenant que nous savons utiliser IDLE pour créer, exécuter et sauvegarder des programmes nous ne souhaitons pas ouvrir une fenêtre de dessins a chaque exécution, la commande « `turtle.bye()` » permettrait de fermer la fenêtre dessin mais nous ne verrions

plus le résultat de nos programmes c'est pourquoi je vous propose d'utiliser la méthode « `turtle.exitonclick` » (*exit* signifie sortie, *on* signifie sur, *click* signifie clic).

N'oubliez pas de créer une nouvelle fenêtre (*new windows* dans le menu « `File` » d'IDLE ou de rouvrir un programme existant pour taper votre code.

à taper
```
import turtle
tortue = turtle.Pen()
tortue.forward(100)
turtle.exitonclick
```

Lancez le programme (sauvez le sous le nom de votre choix finissant par « `.py` » si nécessaire) puis cliquez sur la fenêtre de dessin pour quitter le programme.

Nous pouvons maintenant utiliser des fonctions simples pour déplacer notre tortue sur le zone de dessin et dessiner des formes simples. Mais encore plus intéressant nous pouvons utiliser ce que nous avons appris dans les chapitres précédents. Par exemple, le code que nous avions utilisé pour créer un carré était :

ne pas saisir
```
>>> tortue.forward(50)
>>> tortue.left(90)
>>> tortue.forward(50)
>>> tortue.left(90)
>>> tortue.forward(50)
>>> tortue.left(90)
```

Nous pouvons maintenant le ressaisir en utilisant un itérateur :

à taper
```
import turtle
tortue = turtle.Pen()
for x in range(4):
    tortue.forward(50)
    tortue.left(90)

turtle.exitonclick
```

Ce qui fait nettement moins à saisir, mais nous pouvons maintenant faire des choses un peu plus intéressantes. Essayez donc ce qui suit :

à taper
```
import turtle
tortue = turtle.Pen()
for x in range(8):
    tortue.forward(100)
    tortue.left(225)

turtle.exitonclick()
```

Ce code produit une étoile à huit branches comme montré sur la Figure 8.1 (la tortue tourne de 225 degrés puis avance de 100 pixels, huit fois).

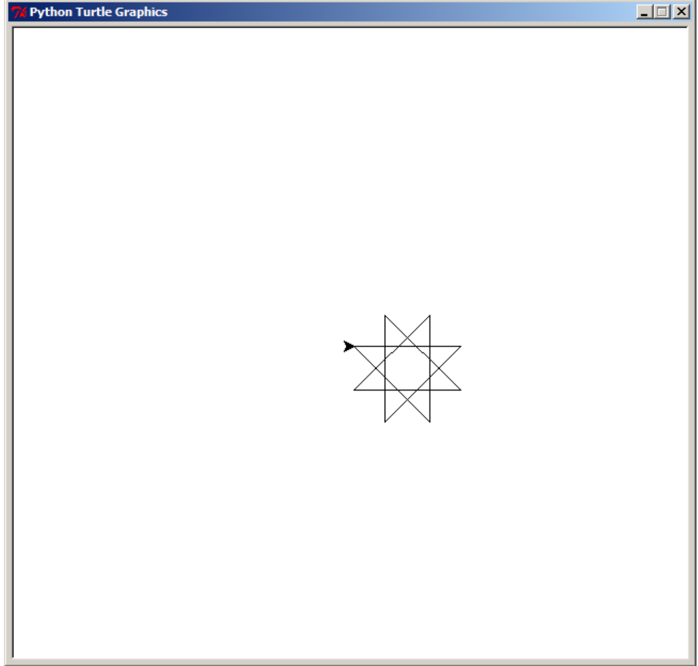


FIGURE 8.1 – Étoile à huit branches

8.2 Je jure […] de dire *toute* la vérité, rien que la vérité

Comme nous avons pu le remarquer la tortue est lente, très lente… En fait on peut choisir la vitesse de la « tortue » avec la fonction « `turtle.speed` » (*speed* signifie vitesse en anglais). La fonction « `turtle.speed` » prend en paramètre un entier entre zéro et dix. D'après le manuel, la vitesse de dessin va de très lent pour un, à très rapide pour dix. La vitesse normale (par défaut) est six. Pour une valeur de zéro aucune animation n'a lieu, c'est à dire que l'affichage du résultat est presque instantané :

à taper

```
import turtle
turtle.speed(0)
tortue = turtle.Pen()

for x in range(36):
    tortue.forward(200)
    tortue.left(170)

turtle.exitonclick()
```

Comme vous pouver le voir sur la Figure 8.2 on obtient une étoile avec plein de branches.

À ce point, deux choses ont pu vous surprendre :
- la vitesse la plus rapide est assez lente ;
- une flèche est apparue au début du dessin lors du lancement de l'instruction « `turtle.speed` ».

En fait, il y a deux expplications à ces constatations. Le système d'affichage interne de « `turtle` » est lent, le terme instantané est donc très faux. Deuxièmement, l'instruction

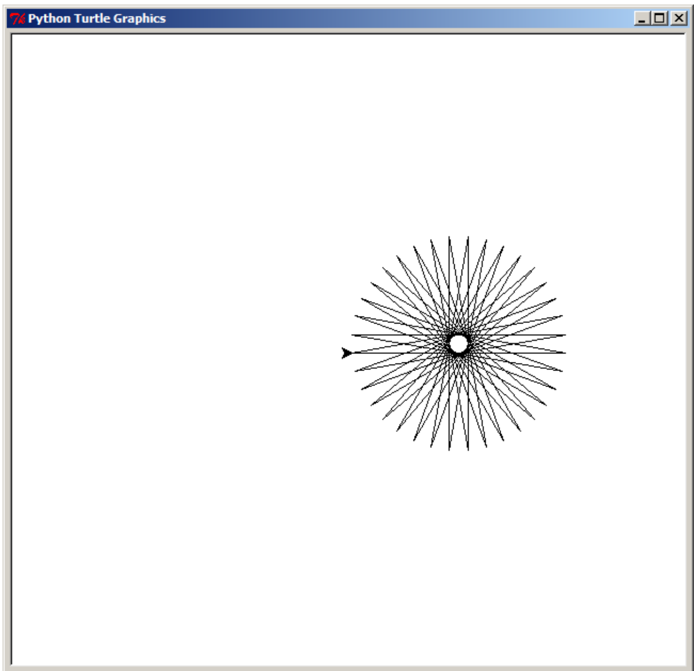


FIGURE 8.2 – Étoile à trente six branches

« `turtle.Pen()` » ne crée que par effet de bord une zone d'affichage. Cette instruction crée un stylo qui va écrire sur la zone de dessin. Si celle-ci n'existe pas, cette zone est alors créée au passage. Inversement le module « `turtle` » inclut un stylo par défaut et « `turtle.forward` » créera un stylo pour réaliser une écriture.

Malheureusement l'instruction « `turtle` » crée aussi un stylo... Heureusement il existe des instructions pour cacher ou faire apparaître les stylos.

à taper

```
import turtle
turtle.hideturtle()
turtle.speed(0)

tortue = turtle.Pen()
tortue.hideturtle()


for x in range(36):
    tortue.forward(200)
    turtle.speed(0)
    tortue.left(170)

tortue.showturtle()
turtle.exitonclick()
```

L'instruction « `turtle.hideturtle()` » cache le stylo par défaut (*hide* cacher en anglais) plus généralement si une « `variable` » est de type « `Pen` » on peut cacher ce stylo avec « `variable.hideturtle()` ». Il est possible de cacher le stylo par défaut. S'il n'existe pas encore, il est créé caché et ne sera jamais visible sauf si nous décidons de l'afficher avec

« showturle ». L'instruction « showturtle » est généralement utilisée avant de dessiner et l'instruction « hideturtle » après avoir dessiné.

Par exemple on peut faire :

à taper

```
import turtle
turtle.hideturtle()
turtle.speed(0)

listestylos=[]

for x in range(36):
    listestylos.append(turtle.Pen())

for stylo in enumerate(listestylos):
    stylo[1].left(10*stylo[0])
    stylo[1].forward(10*(stylo[0]+1))
    stylo[1].hideturtle()

turtle.exitonclick()
```

Dans cet exemple nous créons une liste de trente six stylos puis nous les utilisons et enfin nous les cachons. Plusieurs remarques concernant ce code peuvent être faites :

- nous ajoutons des éléments sur une liste vide ;
- nous utilisons la fonction « enumerate ».

Pour pouvoir ajouter avec « append » des éléments à une liste, il est nécessaire que la liste existe au préalable, c'est pourquoi nous l'initialisons avec une liste vide « [] ».

La fonction « enumerate » (énumérer en anglais) est utilisée pour générer des couples index et valeur (n-uplet de deux éléments) pour l'itérateur. Cette construction permet de faire un code lisible, au lieu de créer une variable artificielle et inutile « x » utilisée dans un code moche :

moche

```
for x in range(len(listestylos)):
    listestylos[x].left(10*x)
    listestylos[x].forward(10*(x+1))
    listestylos[x].hideturtle()
```

La fonction « enumerate » fournit à chaque pas un couple « index, valeur ». Dans notre cas l'index dans « stylo[0] » varie entre zéro et trente-cinq, la valeur dans « stylo[1] » est de type « Pen » et correspond aux trente six stylos que nous avions créé dans le premier itérateur.

L'abréviation « len » est utilisée à la place de *length* qui signifie longueur (de la liste dans notre cas).

Pour la dernière fois nous indiquons ci-dessous la totalité du code du programme : Par la suite nous indiquerons uniquement le code qui nécessite votre attention :

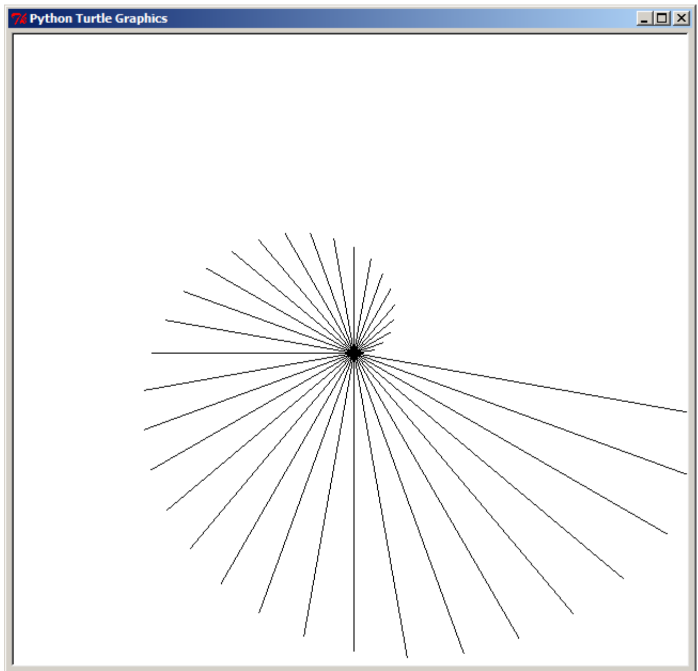


FIGURE 8.3 – Escargot réalisé avec 36 stylos

ne sera plus rappelé

```
import turtle
turtle.hideturtle()
turtle.speed(0)
tortue = turtle.Pen()
```

à taper

```
for x in range(1,24):
        tortue.forward(200)
        tortue.left(94)
```

ne sera plus rappelé

```
tortue.hideturtle()
turtle.exitonclick()
```

Ce qui donne une jolie étoile présentée Figure 8.4.

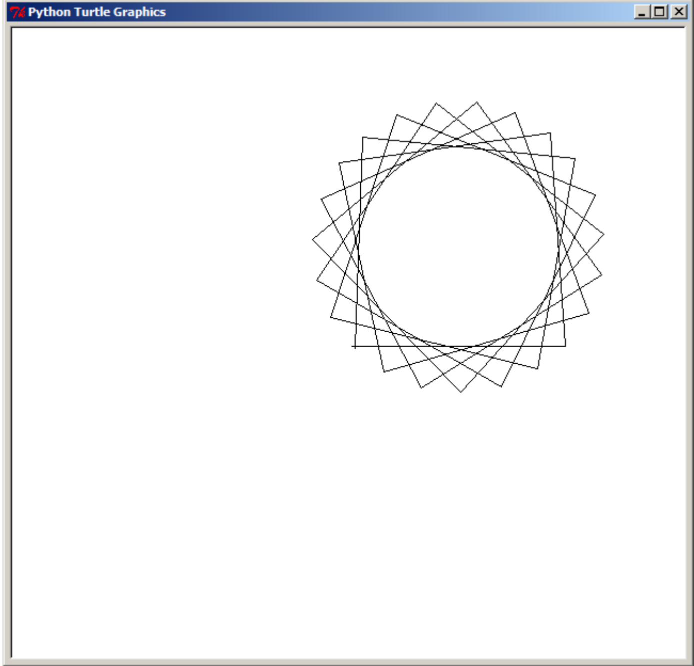


FIGURE 8.4 – Étoile à 24 branches

Maintenant quelque chose d'un peu plus compliqué... Nous n'avons pas décrit tous les opérateurs. Un opérateur fort utile est « % » qui est le reste d'une division entière. C'est à dire ce qui reste dans une division, avant de s'embêter avec les virgules. Cet opérateur est appelé *modulo*. Si vous n'avez pas encore vu la division en classe retenez juste que si le résultat de cette opération entre deux nombres est nul, c'est que le premier nombre est divisible par le premier.

Dans notre cas nous voulons savoir si « x » est pair (divisible par deux) pour une liste d'entiers successifs entre un et dix neuf :

à taper

```
for x in range(1,19):
    tortue.forward(100)
    if x % 2 == 0:
        tortue.left(175)
    else:
        tortue.left(225)
```

Le résultat de ce code est une étoile à neuf branches que vous pouvez observer sur la Figure 8.5.

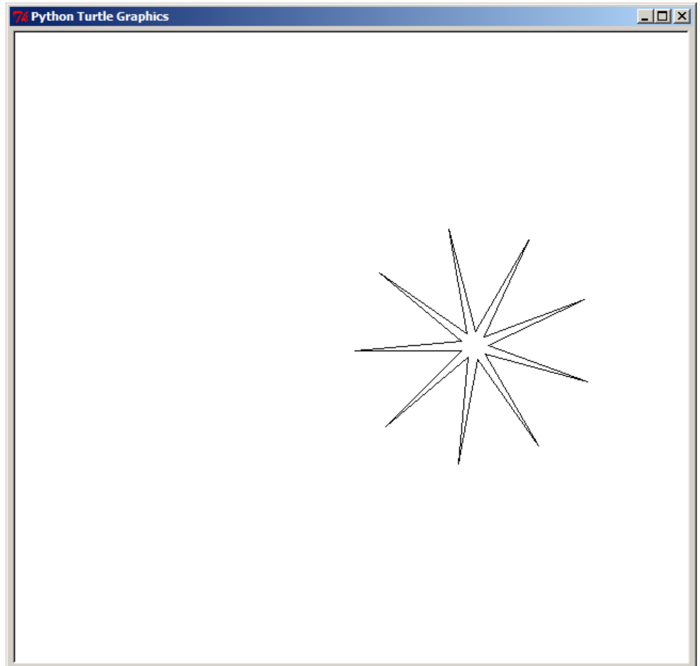


FIGURE 8.5 – Étoile à neuf branches

Nous ne sommes pas obligé de nous limiter à tracer des étoiles et des formes simples. Si nous utilisons une combinaison d'appels à des fonctions, notre tortue peut dessiner différentes choses. Nous pouvons par exemple dessiner une voiture :

à copier-coller depuis le fichier du livre

```
tortue.color(1,0,0)    #color (couleur) en (rouge,vert,bleu)
tortue.begin_fill()    #begin (commencer) fill (remplir)
tortue.forward(100)
tortue.left(90)
tortue.forward(20)
tortue.left(90)
tortue.forward(20)
tortue.right(90)
tortue.forward(20)
tortue.left(90)
tortue.forward(60)
tortue.left(90)
tortue.forward(20)
tortue.right(90)
tortue.forward(20)
tortue.left(90)
tortue.forward(20)
tortue.end_fill()      #end (finir) fill (remplir)
tortue.color(0,0,0)    #(rouge,vert,bleu) entre 0 (foncé) et 1 (clair)
tortue.up()
tortue.forward(10)
tortue.down()
tortue.begin_fill()
```

```
tortue.circle(10)      #cercle de diamètre 10 pixels
tortue.end_fill()
tortue.setheading(0)   #set (fixer) heading (orientation)
tortue.up()
tortue.forward(90)
tortue.right(90)
tortue.forward(10)
tortue.setheading(0)   #orientation=0 direction initiale
tortue.begin_fill()
tortue.down()
tortue.circle(10)
tortue.end_fill()
```

Ce qui est une longue, très longue, méthode pour tracer une voiture vraiment moche et simpliste que vous pouvez voir sur la Figure 8.6.

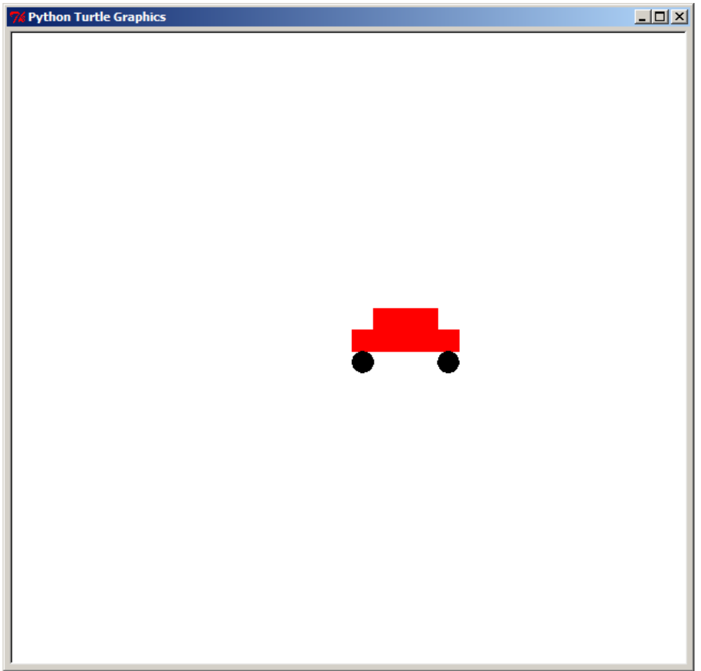


FIGURE 8.6 – Voiture primitive

Néanmoins cet exemple montre quelques autres fonctions de « `turtle` » :

- « `color` » qui change la couleur du stylo ;
- « `begin_fill` » et « `end_fill` » pour remplir des zones ;
- et « `circle` » pour dessiner des cercles d'une taille donnée.

8.3 Connaissez vous les couleurs ?

La fonction « `color` » prend trois paramètres. Le premier pour la quantité de rouge, le seconde pour la quantité de vert et le dernier pour la quantité de bleu.

Pourquoi rouge, vert et bleu ?

Si vous avez déjà joué avec différentes couleurs de peinture, vous connaissez déjà une partie de la réponse à cette question. Quand vous mélangez deux couleurs de peinture différentes vous obtenez une troisième couleur. En informatique les trois couleurs primaires sont rouge, vert et bleu. C'est à dire qu'à partir de ces trois couleurs on obtient toutes les couleurs qui peuvent être affichées sur l'écran [1]. La Figure 8.7 vous montre le principe de l'addition des couleurs.

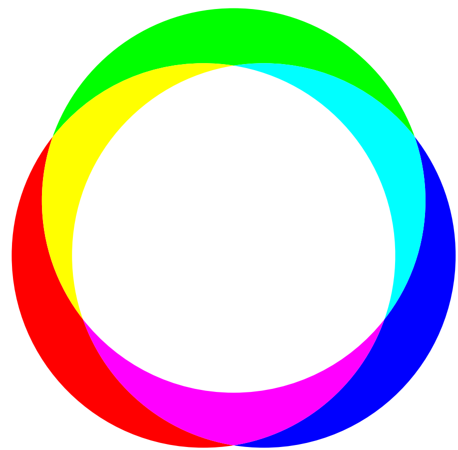


FIGURE 8.7 – Addition des couleurs primaires

La lumière blanche est composée d'un mélange des différentes couleurs. Chaque peinture crée de la couleur car elle absorbe certaines couleurs de la lumière blanche et ne laisse qu'une partie repartir. Quand vous mélangez des peintures, les différentes couleurs absorbées s'ajoutent. C'est pourquoi quand vous mélanger une peinture rouge et une peinture bleu vous avez du violet, plus foncé. De même si vous mélangez de nombreuses couleurs vous obtenez, habituellement, du brun boueux.

Sur un ordinateur, c'est un peu pareil mise à part que nous combinons des émissions de lumière. Si nous combinons du rouge et du bleu nous obtenons du magenta plus clair. Nous avons trois projecteurs que nous pouvons régler de zéro à cent pourcent (en fait de 0 à 1).

La fonction « `color` » comme la plupart des fonctions manipulant les couleurs en informatique prend les couleurs dans l'ordre rouge, vert et bleu. Traçons maintenant un disque jaune :

à taper
```
tortue.color(1,1,0)
tortue.begin_fill()
tortue.circle(100)
tortue.end_fill()
```

Dans l'exemple ci-dessus, nous appelons la fonction « `color` » avec 100% de rouge, 100% de vert et 0% de bleu (en d'autres termes 1, 1 et 0). Pour faire des expériences plus facilement avec différentes couleurs, transformons ce code en fonction :

1. Pour la peinture, les trois couleurs primaires sont jaune, magenta, cyan.

à taper
```
def mondisque(rouge, vert, bleu):
    tortue.color(rouge, vert, bleu)
    tortue.begin_fill()
    tortue.circle(100)
    tortue.end_fill()

mondisque(0, 1, 0)
mondisque(0, 0.5, 0)
```

Nous pouvons tracer un disque vert vif avec « `mondisque(0, 1, 0)` » en utilisant le projecteur vert réglé au maximum. Nous pouvons tracer un disque vert foncé avec « `mondisque(0, 0.5, 0)` » en utilisant le projecteur réglé à la moitié. Comme nous utilisons de la lumière, moins de couleur signifie généralement un résultat plus sombre. C'est comme avec une lampe torche : si la batterie est chargée vous avez une lumière claire qui devient de plus en plus sombre avec la décharge de la batterie.

De manière à le voir par vous même, essayez de tracer un cercle avec le rouge à fond puis à moitié (1 et 0,5), puis avec le bleu à fond puis à moitié.

à taper
```
mondisque(1, 0, 0)
mondisque(0.5, 0, 0)
mondisque(0, 0, 1)
mondisque(0, 0, 0.5)
```

Différentes combinaisons de rouge, de vert et de bleu produirons une grande variété de couleurs. Vous pouvez avec une couleur orangée en utilisant 100% de rouge, 85% de vert et pas de bleu : « `mondisque(1, 0.85, 0)` ».

Une couleur rose bonbon peut être réalisée en combinant 100% de rouge, 70% de vert et 75% de bleu : « `mondisque(1, 0.7, 0.75)` ».

Vous aurez du orange en combinant 100% de rouge et 65% de vert. Le brun peut être obtenu en combinant 100% de rouge, 30% de vert et 15% de bleu :

à taper
```
mondisque(1, 0.65, 0)
mondisque(0.6, 0.3, 0.15)
```

N'hésitez pas à expérimenter par vous même.

Vous pouvez d'ailleurs utiliser « `tortue.clear()` » pour effacer ce qui a été tracé.

8.4 Remplissage

Vous avez probablement déjà compris que les fonctions « `begin_fill` » et « `end_fill` » signalaient le début et la fin du zone à remplir. Le remplissage ne s'effectue réellement qu'une fois que la commande « `end_fill` » est invoquée et que Python peut déterminer avec exactitude la zone à remplir.

Premièrement, créons une fonction « `moncarré` » :

à taper
```
def moncarré(taille):
    for i in range(4):
        tortue.forward(taille)
        tortue.left(90)
```

Cette fonction dessine des carrés de taille variable selon votre choix. Vous pouvez l'essayer avec plusieurs tailles :

à taper
```
moncarré(25)
moncarré(50)
moncarré(75)
moncarré(100)
```

Notre tortue devrait dessiner semblable à ce qui est présenté sur la Figure 8.8.

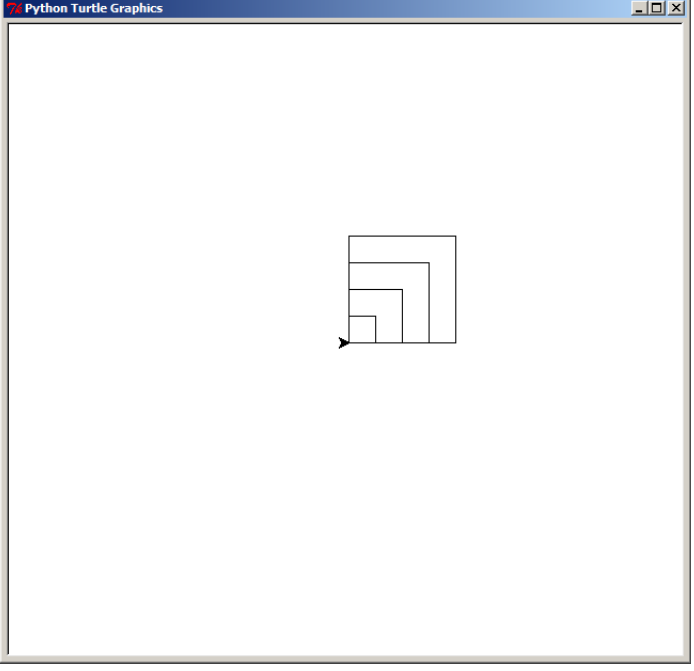


FIGURE 8.8 – 4 carrés

Maintenant nous pouvons essayer un carré rempli. Premièrement nous allons commencer le remplissage avec « `begin_fill` ». Puis nous l'arrêterons avec « `end_fill` ».

à taper
```
tortue.begin_fill()
moncarré(100)
tortue.end_fill()
```

Ce qui va produire un dessin comme le carré de la Figure 8.9.

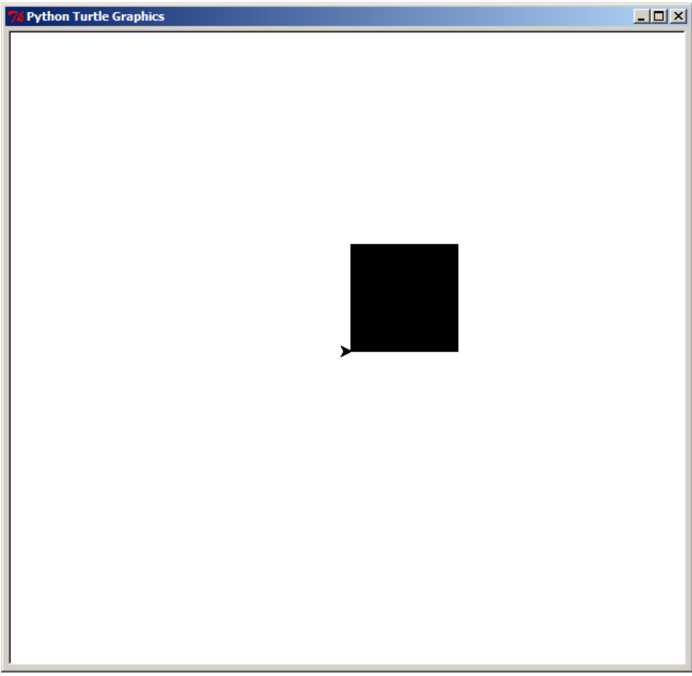


FIGURE 8.9 – Un carré rempli

Comment changer la fonction de manière à ce que nous puisions dessiner un carré, rempli ou non ? Nous avons besoin d'un autre paramètre, ce qui est légèrement plus compliqué de la manière suivante :

à taper

```
def moncarré(taille, rempli):
    if rempli == True:
        tortue.begin_fill()
    for i in range(4):
        tortue.forward(taille)
        tortue.left(90)
    if rempli == True:
        tortue.end_fill()

moncarré(50, True)
moncarré(150, False)
```

Dans les deux premières lignes de la fonction nous vérifions si le paramètre « `rempli` » vaut « `True` ». Si c'est le cas alors la fonction « `begin_fill` » est lancée. Nous bouclons alors quatre fois pour dessiner un carré avant de vérifier une seconde fois si le paramètre « `rempli` » vaut vrai. Si cela est le cas nous lançons alors la fonction « `end_fill` ».

Puis nous traçons un carré plein et un carré vide pour tracer l'image de la Figure 8.10. Maintenant que j'y pense, elle ressemble à un drôle d'œil carré.

Vous pouvez dessiner toute sorte de formes et les remplir avec des couleurs. Revenons à une des étoiles que nous avions tracée plus tôt. Le code original ressemblait à cela :

ne pas saisir

```
for x in range(1,19):
    tortue.forward(100)
```

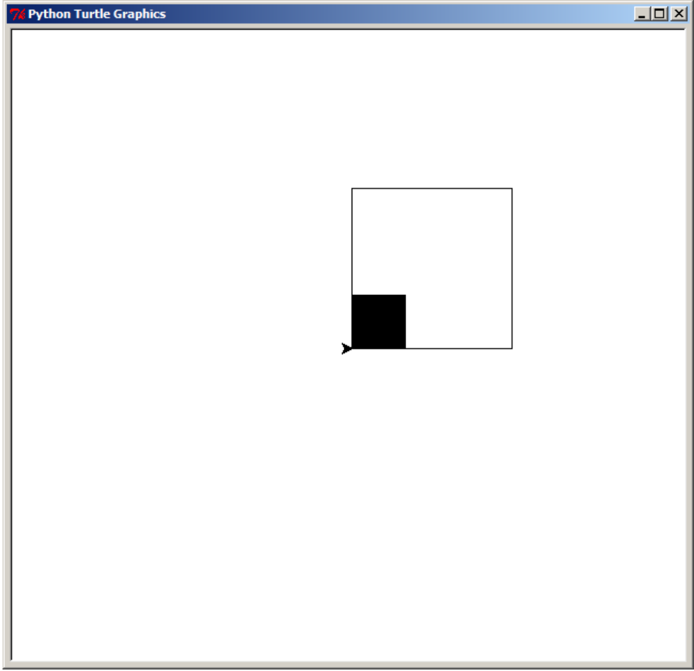


FIGURE 8.10 – Œil fait avec des carrés

```
    if x % 2 == 0:
        tortue.left(175)
    else:
        tortue.left(225)
```

Nous pouvons utiliser les mêmes « tests si » de la fonction « `moncarré` » et utiliser le paramètre de taille comme paramètre dans la fonction « `forward` ».

à taper

```
def monétoile(taille, rempli) :
    if rempli == True:
        tortue.begin_fill()
    for x in range(1,19):
        tortue.forward(taille)
        if x % 2 == 0:
            tortue.left(175)
        else:
            tortue.left(225)
    if rempli == True:
        tortue.end_fill()
```

À la deuxième et à la troisième ligne, nous contrôlons si le paramètre « `rempli` » vaut vrai, si c'est le cas nous activons le remplissage. Nous faisons similairement, à la dixième et onzième ligne, mais nous arrêtons la zone de remplissage, c'est à ce moment que le remplissage se fait.

L'autre différence à propos de cette fonction est que nous passons la taille de l'étoile (en fait d'une branche de l'étoile) comme paramètre et que nous utilisons cette valeur à la cinquième ligne.

Maintenant choisissons la couleur de l'étoile, par exemple orangé (vous vous rappelez peut-être qu'orangé est fait en mélangent 100% de rouge, 85% de vert et pas de bleu), puis appelons la fonction.

à taper
```
tortue.color(1, 0.85, 0)
monétoile(120, True)
```

La tortue devrait tracer l'étoile de la Figure 8.11.

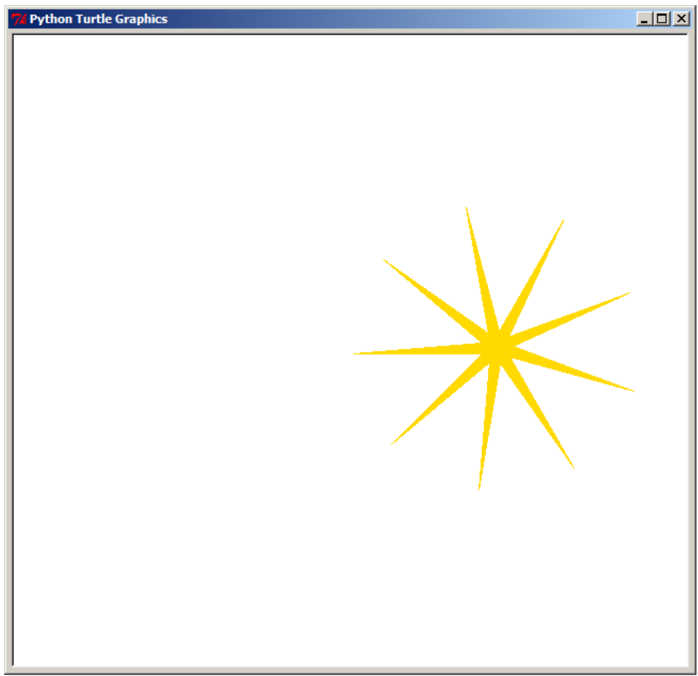


FIGURE 8.11 – Étoile orangée simple

Nous pouvons ajouter un contour à l'étoile en changeant la couleur à nouveau (cette fois par exemple en noir) et redessiner en plus une étoile similaire mais sans remplissage.

à taper
```
tortue.color(0, 0, 0)
monétoile(120, False)
```

Cette fois-ci l'étoile ressemble à la Figure 8.12

8.5 Obscurité

J'ai une question pour vous : qu'arrive-t'il quand vous éteignez toutes les lumières pendant la nuit ?

Tout devient noir. La même chose arrive avec les couleurs sur un ordinateur. Absence de couleur signifie absence de lumière. Ainsi la fonction « `mondisque` » avec les trois couleurs à zéro produit un disque noir. L'inverse est vrai aussi, si les trois couleurs sont à un, la couleur résultante est le blanc.

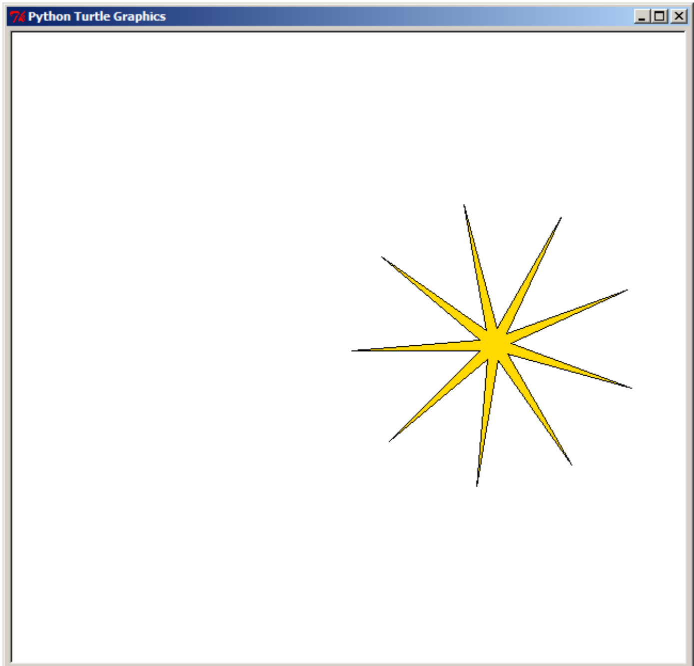


FIGURE 8.12 – Étoile orangée finalisée

Pour faire un joli dessin, je me dois de vous informer que les fonctions peuvent prendre des paramètres optionnels. Des paramètres facultatifs qui, s'ils sont utilisés, modifient le résultat d'une fonction.

Par exemple la méthode « `circle` », en plus du rayon, peut prendre deux paramètres optionnels : l'angle « `extent` » (*extent* signifie étendue en anglais) et le nombre d'étapes pour réaliser le tracé « `steps` » (*step* signifie étape en anglais).

La méthode circle s'utilise « `circle(rayon,angle,étapes)` », si aucun angle n'est fourni alors l'angle vaut 360 degrés (le cercle est complet). Si l'on choisi des angles plus petits des arcs de cercle seront tracés. Par exemple, 90 degrés permettront de faire de quarts de cercle et 180 degrés permettront de faire des demi-cercles.

Pour être exact *on nous cache tout, on nous dit rien* [2], le rayon peut être choisi positif ou *négatif*. Si le rayon est positif la tortue avance en tournant vers la gauche. Si le rayon est négatif la tortue avance en tournant vers la droite [3].

L'angle « `extent` » peut lui aussi être choisi négatif ou positif. Si l'angle est négatif la tortue va reculer au lieu d'avancer. Les paramètres optionnels peuvent facilement être utilisés dans l'ordre (c'est à dire les premiers mais pas les derniers). Faisons quelques tests :

à taper

```
tortue.circle(-100)
tortue.circle(100)
tortue.left(90)
tortue.circle(100, 90)
tortue.circle(100, -90)
tortue.right(90)
tortue.circle(100, -180)
```

2. Chanson de Jacques Dutronc

3. Si vous n'avez pas encore vu les nombres négatifs ailleurs que sur un thermomètre, ce n'est pas grave retenez juste le changement de sens.

```
tortue.right(90)
tortue.circle(-100, -90)
```

Vous devriez avoir un résultat similaire à celui de la Figure 8.13.

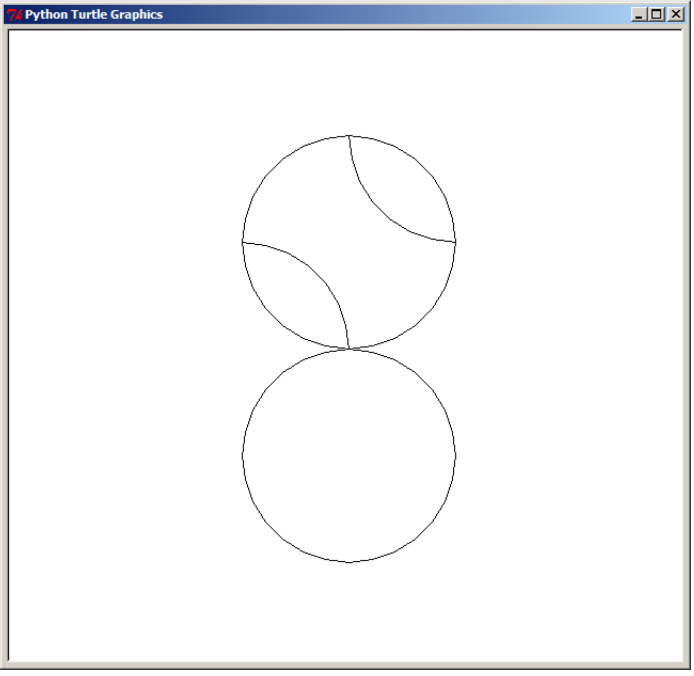


FIGURE 8.13 – Tests avec les rayons et les arcs de cercle

Le nombre d'étapes est automatiquement déterminé sauf si on l'impose. Néanmoins il peut être utile de choisir des valeurs pour tracer des polygones réguliers. Il est possible de modifier un paramètre optionnel, sans modifier les autres, en le nommant. Attention le nombre d'étapes « `steps` » doit être strictement supérieur à zéro. La fonction « `circle` » réalise une division du cercle théorique par le nombre d'étapes pour trouver les sommets du polygone correspondant aux différentes étapes. Il se trouve que la division par zéro n'est pas une très bonne idée et n'a pas un résultat déterminé ce qui ne plaît pas trop aux ordinateurs en général et à Python en particulier.

à taper

```
for i in range(1,6):
        tortue.circle(100, steps=i)
```

La Figure 8.14 montre le résultat de ces instructions. Comme vous pouvez le voir, le « cercle » en deux étapes est un segment qui est dessiné par un aller-retour avec un retour à la position de départ. Le « cercle » en trois étapes est un triangle. Le « cercle » en quatre étapes est un carré et ainsi de suite. Rappelez-vous notre cercle fait avec un itérateur quand nous ne connaissions pas la fonction « `circle` ». En fait, il s'agissait d'un cercle en 360 étapes. La fonction « `circle` » calcule automatiquement le nombre adéquat d'étapes pour que le cercle affiché ressemble vraiment à un cercle.

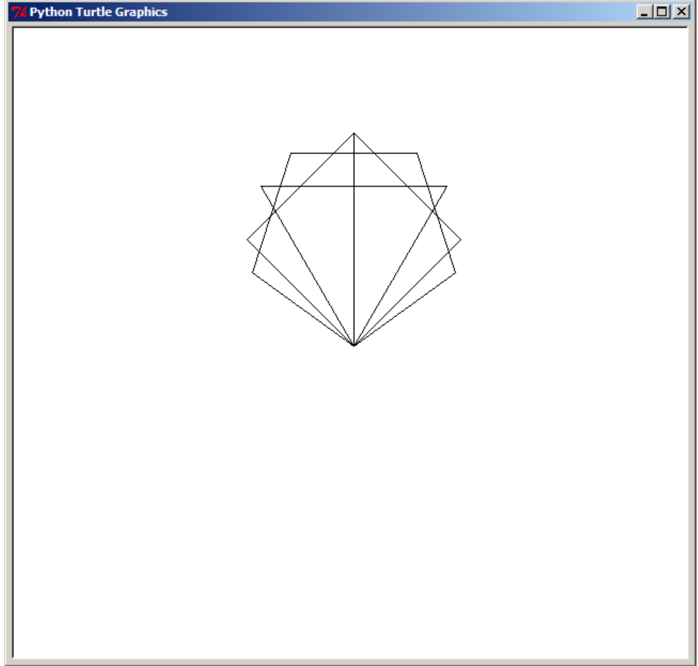


FIGURE 8.14 – Polygones obtenus grâce à « `circle` »

Nous pouvons aussi définir notre propre fonction avec des paramètres optionnels. En ajoutant, à coté du nom du paramètre, la valeur par défaut qui sera utilisée si aucune valeur n'est fournie. Par exemple, « `def moncercle(rayon, angle=360, plein=False) :` » défini une fonction « `moncercle` » dont l'angle est par défaut 360 degré comme sur la fonction « `circle` » mais qui peut être plein (par défaut) ou pas.

Nous allons maintenant tracer le symbole du Yin « 陰 » et du Yang « 陽 ».

à taper

```
def moncercle(rayon, angle=360, plein=False) :
    if plein:
        tortue.begin_fill()
        tortue.circle(rayon, angle)
        tortue.end_fill()
    else :
        tortue.circle(rayon, angle)

tortue.color(0, 0, 0)
moncercle(100, 180, True)
moncercle(50, plein=True)
tortue.left(90)
tortue.forward(60)
tortue.left(90)
tortue.color(1, 1, 1)
moncercle(10, plein=True)
tortue.up()
tortue.right(90)
tortue.forward(40)
tortue.left(90)
```

```
tortue.down()
moncercle(-50, 180, True)
tortue.left(180)
tortue.color(0, 0, 0)
moncercle(100)
tortue.left(90)
tortue.up()
tortue.forward(60)
tortue.down()
tortue.left(90)
moncercle(10, plein=True)
```

Avec un peu de chance vous devriez obtenir le résultat de la Figure 8.15.

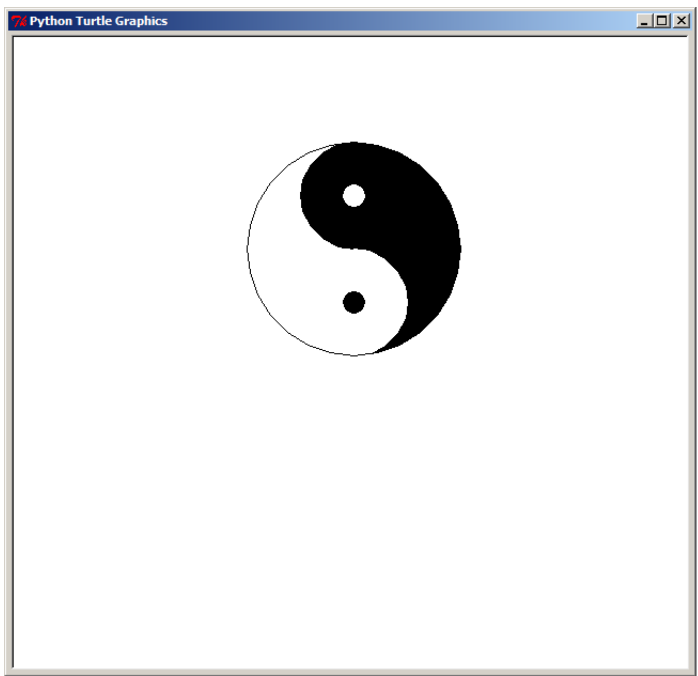


FIGURE 8.15 – Symbole du Yin et du Yang

Vous pouvez sauvegarder le programme en utilisant l'extension de fichier « .pyw » (py comme PYthon et w comme Windows) sur le bureau typiquement. Si on exécute le programme en double cliquant dessus seul l'affichage de dessin se fera sans fenêtre texte derrière contrairement à ce qui ce passe avec l'extension « .py ». Par exemple j'ai appelé mon programme « yinyang.pyw ». Ce programme peut être exécuté sur tout ordinateur où Python est installé. Il est possible de convertir les programmes Python en exécutable avec « py2exe » mais cela n'est pas expliqué dans ce livre car cela nécessite l'installation d'un logiciel supplémentaire [4].

4. Py2exe est un logiciel libre téléchargeable sur le site http://www.py2exe.org/.

8.6 À vous de jouer

Dans ce chapitre nous avons complété nos connaissances du module « `turtle` » et nous l'avons utilisé pour tracer des figures plus complexes. Nous avons utilisé des fonctions de manière à réutiliser notre code et en particulier pour tracer des dessins de couleurs différentes.

Vous trouverez des pistes de réponses dans la section A.5.

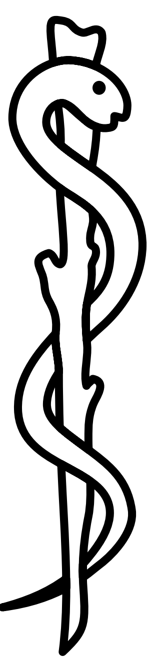
8.6.1 Exercice 1

Nous avons tracé des étoiles, des carrés et des cercles. Pouvez-vous tracer un octogone régulier ? Un octogone est une forme (un polygone) à huit cotés. Un octogone régulier a tous ses cotés égaux ainsi que tous ses angles aux sommets.

Indice : 360 divisés par 8 font 45, un octogone est plus proche d'un cercle qu'un carré.

8.6.2 Exercice 2

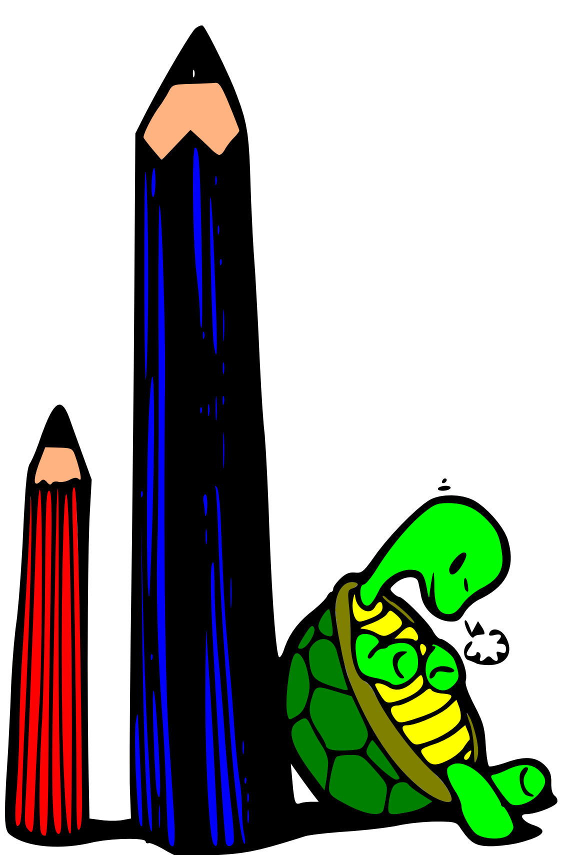
Maintenant convertissez le code pour dessiner un octogone en une fonction qui permet de créer un octogone rempli avec une couleur.



9 Un peu de graphismes

9.1 Qui va lentement, va surement.

Le problème avec l'utilisation d'une tortue pour dessiner est... que... les tortues... sont... vraiment... lentes.



Même quand une tortue va à la vitesse maximale, elle continue de ne pas aller très vite. Pour les tortues ce n'est pas vraiment un problème — elles ont du temps à perdre — mais quand on parle de graphismes sur un ordinateur, c'est une toute autre affaire. Si vous avez une Nintendo DS, une PSP ou si vous jouez à des jeux sur votre ordinateur, pensez un moment aux graphismes que vous voyez affichés sur l'écran. Il y a différentes manières de représenter des graphismes dans un jeu. Il y a les jeux en deux dimensions (2D) où les images sont « plates » et dans lesquels les personnages se déplacent verticalement et horizontalement

mais pas en profondeur. Ces jeux sont courants sur les consoles mobiles et les téléphones mobiles. Et il y a des jeux en trois dimensions (3D) dans lesquelles les images ressemblent à la réalité et qui ajoutent de la « profondeur ».

Tous ces types d'affichage graphiques ont une chose en commun — le besoin de dessiner sur l'écran vraiment rapidement. Avez-vous déjà essayé de faire vos propre animations ? Celles où vous prenez un bloc de papier vierge et où dans le coin de la première page vous dessinez quelque chose (peut-être un bonhomme fil de fer) puis dans le coin de de la page suivante vous dessinez le même bonhomme avec une jambe qui a légèrement bougé. Puis sur la page d'après vous dessinez le même bonhomme avec la jambe un peu plus bougée encore. Et cela recommence, ainsi de suite, sur toutes les pages du bloc. Quand vous avez fini, si vous faites défiler les pages assez vite, il semblera que le bonhomme bouge. Ce sont les bases de la méthode utilisée pour toutes les animations, que vous regardiez un dessin animé ou un jeu vidéo sur une console ou un ordinateur. Vous dessinez quelque chose puis vous le dessinez à nouveau légèrement modifié pour donner l'illusion du mouvement. C'est pourquoi une tortue n'est pas douée pour faire la plupart des graphismes. De manière à faire apparaître des animations qui sont des images qui bougent, vous avez à dessiner chaque image de l'animation très vite.

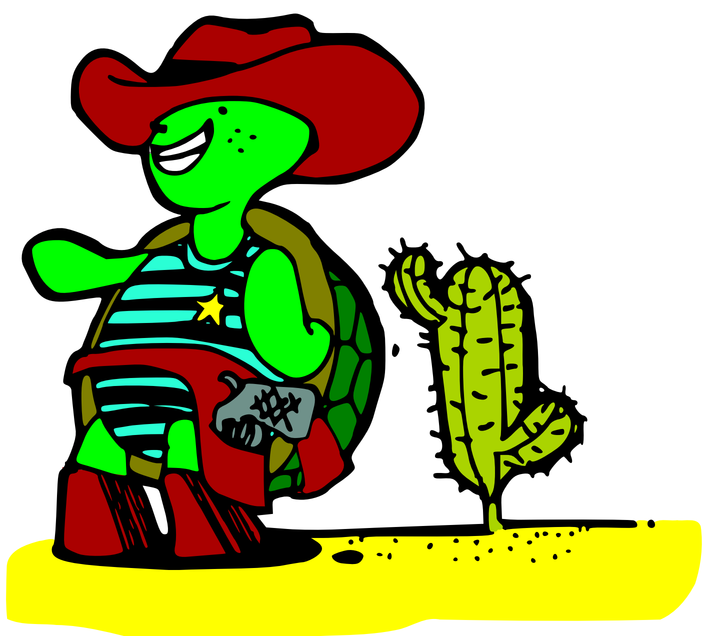
Les graphismes en trois dimensions sont faits très différemment des graphismes en deux dimensions, mais là aussi, l'idée-force est la même. Le temps que votre tortue ait fini de tracer une petite partie de l'image, il serait temps de tourner la page est de commencer à dessiner la suivante...

9.2 Tortue de course

Chaque langage de programmation a des méthodes différentes pour dessiner sur un écran. Certaines méthodes sont rapides et d'autres sont lentes, ce qui veut dire que les programmeurs qui développent des vrais jeux doivent être vraiment attentif au langage qu'ils choisissent pour travailler. Néanmoins, la plupart des graphismes de jeux sont fait avec des bibliothèques (des sortes de super modules) qui sont utilisables de manière similaire avec plusieurs langages de programmation.

Python a aussi différentes manière de faire des graphismes (incluant « `turtle` » que nous avons déjà utilisé) mais les meilleurs méthodes (d'un point de vue graphique) sont dans des modules et des bibliothèques qui ne sont pas inclues dans Python lui même (ni dans aucun autre langage). Vous passerez probablement quelques années (ou mois pour les plus motivés) à programmer avant de comprendre comment installer et utiliser une de ces bibliothèques complexes.

Heureusement, un module plus adapté est fourni avec Python et nous pouvons utiliser pour faire des graphismes simples, plus rapidement qu'avec une tortue). Peut-être assez rapide pour être appelé la « tortue de course ».



Ce module est appelé « `tkinter` » pour *tool kit interface*, c'est à dire boîte à outils d'interface [1]. Si vous avez regardé attentivement l'icône en haut à gauche des fenêtre d'IDLE et de « `turtle` » vous avez pu voir pour *tool kit*. Tkinter peut être utilisé pour faire de vraies applications comme IDLE (vous pourriez même créer un traitement de texte si vous le vouliez). Il peut aussi être utilisé pour faire de simples dessin. Nous pouvons créer une simple application avec un bouton avec le code suivant :

à taper et sauver dans un nouveau fichier .py

```
import tkinter
racine = tkinter.Tk()
racine.mainloop()
```

À la première ligne nous importons le contenu du module « `tkinter` », ainsi nous pouvons utiliser les fonctions de celui-ci. La plus utile « `Tk()` » est utilisée à la deuxième ligne pour créer une fenêtre vide dans laquelle nous pourrons ajouter des choses plus tard. Finalement, à la troisième ligne, nous utilisons la méthode « `mainloop` » (boucle principale) de notre fenêtre vide « `racine` ». Cette boucle principale est une boucle infinie que nous n'avons pas a gérer directement qui parcourt la fenêtre et ces éléments pour voir si quelque chose à changé. Pour le moment notre application est vide, vous pouvez la fermer en utilisant le bouton de fermeture de fenêtre de votre système d'exploitation qui devrait ressembler à .

Nous allons maintenant ajouter un élément à notre application, Pour commencer, je vous propose un bouton :

à taper et sauver dans un nouveau fichier .py

```
import tkinter
racine = tkinter.Tk()
bouton = tkinter.Button(racine, text="Cliquez moi !")
bouton.pack()
racine.mainloop()
```

1. Tkinter n'est pas le module le plus rapide, mais il est fournit nativement avec Python. Des modules plus rapides existent comme pyopengl (`http://pyopengl.sourceforge.net/`) ou pygame (`http://www.pygame.org/`).

À la troisième ligne, nous créons un nouveau bouton avec la commande « `Button` » de « `tkinter` » et nous lui attribuons le nom « `bouton` ». Le bouton est créé en passant comme paramètre l'objet « `racine` » que nous avions préalablement créé. Tous les petits « machins [2] » mis dans les fenêtres doivent avoir un parent. Ce parent est l'objet qui contient les différents contrôles. On doit passer en paramètre le parent qui est généralement une fenêtre. Dans notre cas il s'agit de « `racine` ». Le deuxième paramètre est un paramètre optionnel nommé qui indique quel texte mettre dans le bouton.

La quatrième ligne indique au bouton de se dessiner lui-même. La commande « `pack` » indique au bouton comment il doit être « tassé » (ajouté, empilé) dans la fenêtre parente (ici « `racine` »). Sans paramètre la méthode utilisée va « tasser » les composants vers le haut (*top*) par le bas. À ce point, la fenêtre « `racine` » que nous avions créé se rétrécie elle-même à la taille de notre bouton qui contient les mots : « Cliquez moi ! ». Ce bouton ne fait pas grand chose (rien en fait) mais vous pouvez au moins cliquer dessus.

Paramètres nommés

Ce n'est pas la première fois que nous utilisons des « paramètres nommés ». Ceux-ci fonctionnent comme les paramètres normaux mis à part qu'ils peuvent apparaître dans n'importe quel ordre, c'est pourquoi il faut fournir leur nom.
Par exemple, supposons que nous avons une fonction rectangle qui prend deux paramètres largeur et hauteur. Normalement nous devons appeler cette fonction en utilisant quelque chose comme « `rectangle(200,100)` » ce qui signifie que nous voulons dessiner un rectangle de 200 pixels de large par 100 pixels de haut. Mais que se passerait-il si les paramètres pouvaient apparaitre dans n'importe quel ordre ? Comment savoir ce qui est la largeur et ce qui est la hauteur ? Dans ce cas il vaut mieux dire qui est qui, par exemple « `rectangle(hauteur=100, largeur=200)` ». En réalité, l'idée générale derrière les paramètres nommés est un peu plus compliquée que cela. Ils peuvent être utilisés diversement pour faire des fonctions plus flexibles — mais c'est un sujet pour un livre plus avancé que cette introduction à la programmation.

Nous pouvons légèrement changer notre exemple précédent (n'oubliez de fermer la fenêtre que nous venions de créer). Premièrement, nous allons créer une fonction pour afficher du texte, puis nous allons l'appeler en cliquant sur le bouton.

à taper

```
import tkinter

def coucou() :
    print("coucou")

```

2. Il s'agit du terme québécois, le terme français est « `composants d'interface graphique` », généralement appelé composant ou « `widget` » de *window gadget* (gadget de fenêtre).

```
racine = tkinter.Tk()
bouton = tkinter.Button(racine, text="Cliquez moi !", command=coucou)
bouton.pack()

racine.mainloop()
```

Le paramètre « command » indique que nous voulons lancer une fonction à chaque fois que nous cliquons sur le bouton. Dans notre cas le paramètre « command » vaut « coucou », c'est à dire que la fonction « coucou » va être lancée à chaque clic. Au passage, nous remarquons que Python considère tout élément (ici une fonction) comme un objet qui peut être manipulé. Si vous cliquez sur le bouton vous allez voir « coucou » écrit sur la console à chaque fois que le bouton est cliqué.

La fenêtre « racine » a une méthode « destroy » qui permet de détruire cette fenêtre : il s'agit du *destructeur* de l'objet). Elle peut être appelée de la même manière.

à taper

```
import tkinter

def coucou() :
    print("coucou")

racine = tkinter.Tk()
bouton = tkinter.Button(racine, text="Cliquez moi !", command=coucou)
bouton.pack()

bouton_sortir = tkinter.Button(racine, text="Sortir", command=racine.destroy)
bouton_sortir.pack()

racine.mainloop()
```

Si vous cliquez sur le bouton « Sortir » l'application s'arrête.

Comme vous le voyez ci-dessus, j'ai été obligé de diminuer la taille de mes caractères pour faire tenir la ligne de création du bouton. À ce propos Python ignore les indentations à l'intérieur des parenthèses, cela est utile, même quand on n'est pas un livre.

ne pas saisir

```
import tkinter

def coucou() :
    print("coucou")

racine = tkinter.Tk()
bouton = tkinter.Button(racine, text="Cliquez moi !", command=coucou)
bouton.pack()

bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()

racine.mainloop()
```

9.3 Dessins simples

Les boutons ne sont pas vraiment utiles si vous voulez dessiner des choses sur l'écran. Ainsi nous avons besoins de créer et ajouter une sorte différente de composant : une zone de dessin appelé « `Canvas` » . Quand nous créons une zone de dessin, nous avons besoin de passer la hauteur et la largeur en paramètre, contrairement au bouton qui prenait en paramètre un texte et une commande. Mis à part la hauteur (*height*) et la largeur (*width*) le code est similaire à celui utilisé pour le bouton.

à taper

```
import tkinter
racine = tkinter.Tk()

canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()

bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()
racine.mainloop()
```

Comme dans l'exemple du bouton, une fenêtre racine est créée à la deuxième ligne. Quand vous « tassez » [3] la zone de dessin « `canvas` » ligne cinq, la fenêtre racine augmente de taille pour l'accueillir. Nous pouvons maintenant dessiner une ligne dans la zone de dessin en utilisant des coordonnées en pixels. Les coordonnées sont les positions des pixels dans la zone de dessin (verticalement et horizontalement). Dans une zone de dessin « `tk` » les coordonnées décrivent à quelle distance nous sommes du coin en haut à gauche. Ces coordonnées sont données d'abord horizontalement de gauche vers la droite puis verticalement de haut en bas. L'ensemble des valeurs horizontales (de gauche vers la droite) est traditionnellement appelé axe des « x ». L'ensemble des valeurs verticales (de haut en bas) est traditionnellement appelé axe des « y ». Le coin en haut à gauche, appelé l'origine, a pour coordonnées (0,0). Comme la zone de dessin fait 500 pixels de large et 500 pixel de haut le coin en bas à droite est situé aux coordonnées (499,499) [4]. Ainsi la ligne de la Figure 9.1 peut être tracée en utilisant (0,0) pour coordonnées de départ et (499,499) pour coordonnées d'arrivée. Pour ce faire nous utilisons la méthode « `create_line` » [5] du « `Canvas` » qui prend pour paramètre les coordonnées de points et les relie par des lignes.

à taper

```
import tkinter
racine = tkinter.Tk()

canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()

```

3. Le verbe *to pack* signifie emballer ou tasser
4. Ces valeurs sont théoriques, en pratique, en raison d'approximations faites par « `tk` », les valeurs effectives peuvent être légèrement différentes. Sur l'ordinateur du traducteur l'origine est située à (2,2) et le coin en bas à droite est à (501,501).
5. Le verbe *to create* signifie créer en anglais et *line* signifie ligne.

```
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()
canvas.create_line(0, 0, 499, 499)

racine.mainloop()
```

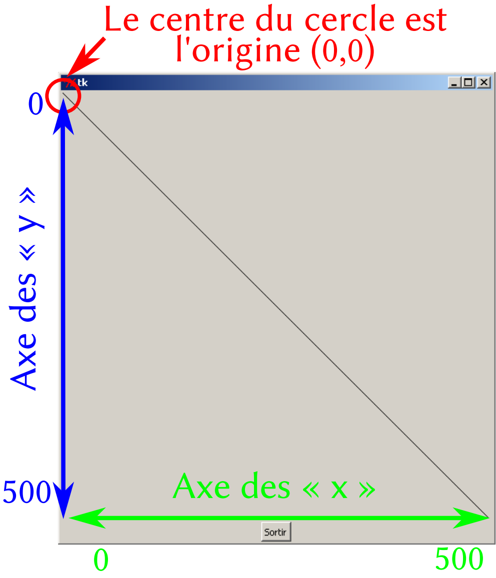


FIGURE 9.1 – Canvas (zone de dessin) avec les axes x et y

Si vous avez du mal à lire les coordonnées, il convient de les séparer avec :
- des espace (comme ici) ;
- ou un retour à la ligne.

exemple
```
canvas.create_line(  0,  0,
                   499,499)
```

L'équivalent de :

ne pas taper
```
racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
canvas.create_line(0,0,499,499)
```

en « `turtle` » est :

ne pas taper
```
turtle.setup(width=500, height=500)
tortue = turtle.Pen()
tortue.up()
tortue.goto(-250,250)
tortue.down()
tortue.goto(500,-500)
```

Pour faire la même chose (beaucoup plus lentement) le code est nettement plus long. Et encore nous avons utilisé la fonction « `goto` »[6] dont je ne vous avais pas encore parlé[7].

Avec « `tkinter` » le code est plus simple à comprendre et plus rapide. Il y a un grand nombre de méthodes disponibles avec l'objet « `Canvas` » dont certaines qui ne nous seront pas très utiles, mais jetons un coup d'œil aux fonctions intéressantes.

9.4 Dessiner des boîtes

Dans « `turtle` », nous dessinons une boîte carrée en faisant : avancer, tourner, avancer, tourner, tourner, avancer, tourner, tourner, avancer et tourner. Éventuellement vous pouvez tracer un rectangle à la place d'un carré en changeant de combien vous avancez. Avec « `tkinter` » dessiner un carré ou un rectangle est considérablement plus simple — vous avez juste besoin de connaître les coordonnées des coins. Dans un souci de simplification et en raison de mon embonpoint déjà trop important, je rappelle ici pour une dernière fois les éléments qui seront à répéter tout le long de ce chapitre.

à taper (ne sera plus indiqué par la suite)

```
import tkinter
racine = tkinter.Tk()

canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()

bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()
```

à taper

```
résultat=canvas.create_rectangle(50, 50,
                                 10, 10)
print(résultat)
```

à taper (ne sera plus indiqué par la suite)

```
racine.mainloop()
```

La fonction « `create_rectangle` » est utilisée pour créer des rectangles à partir des coordonnées de deux points. Les deux points sont les extrémités d'une diagonale. Le rectangle en question a des côtés horizontaux et verticaux.

Dans l'exemple ci-dessus nous dessinons un carré dont le premier coin (en bas à droite) est situé aux coordonnées (50,50) et dont le second coin (en haut à gauche) est situé aux coordonnées (10,10). Vous vous demandez ce qu'est ce nombre que nous avons mis dans « `résultat` » puis que nous affiché dans le shell ? C'est un numéro d'identification de la forme que nous venons de tracer (que cela soit une ligne, un rectangle ou un cercle). Nous reviendrons à ce numéro plus tard.

6. Le verbe *to go* signifie aller et *to* signifie « à ».

7. La méthode « `goto` » de « `turtle` » dit à la tortue de se déplacer par rapport à sa position actuelle d'un certain nombre de points horizontalement et verticalement.

Les paramètres que nous avons passé pour créer le rectangle sont donc : la coordonnée horizontale x du premier point, la coordonnée verticale y du premier point, la coordonnée horizontale x du second point, la coordonnée verticale y du second point. On les appelle généralement x_1, y_1, x_2 et y_2. Nous pouvons tracer un rectangle différent en choisissant x_2 plus grand :

à taper
```
résultat=canvas.create_rectangle(50, 50,
                                 100, 10)
print(résultat)
```

De même, nous pouvons changer ce rectangle en choisissant y_2 encore plus grand :

à taper
```
résultat=canvas.create_rectangle(50, 50,
                                 10, 200)
print(résultat)
```

Le dernier rectangle créé, est créé en disant simplement : place le premier coin 50 pixels sur la droite et 50 pixels vers le bas à partir de l'origine (le coin en haut à gauche) ; puis place un second coin 10 pixels vers la droite et 200 pixels vers le bas à partir de l'origine. À ce moment vous devriez avoir un résultat qui ressemble à la Figure 9.2.

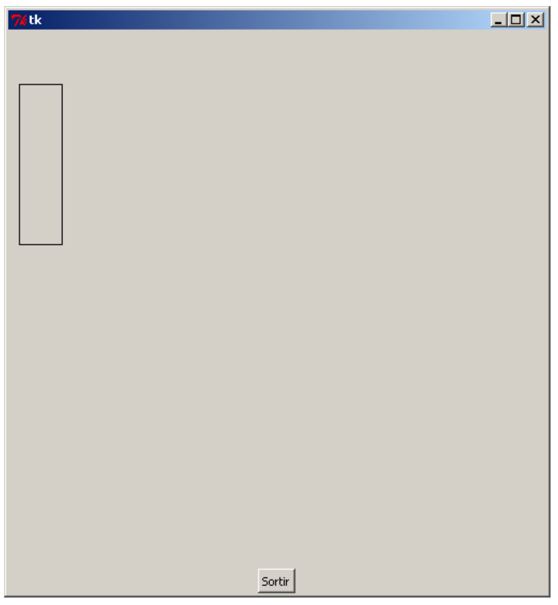


FIGURE 9.2 – Un rectangle réalisé avec « `tk` »

Essayons de remplir le « `Canvas` » avec différentes tailles de rectangles. Nous pouvons pour cela utiliser un module appelé « `random` » [8]. Premièrement nous allons importer le module en lançant : « `import random` ». Pour la lisibilité du code il est conseillé de changer la première ligne de votre programme et d'y placer : « `import tkinter, random` ».

Puis nous pouvons créer une fonction en utilisant des nombres aléatoires pour les deux coins de chaque rectangle. La fonction à utiliser est appelé « `randrange` ». La fonction « `randrange` »

8. Le mot *random* signifie élément aléatoire, c'est à dire dont on ne peut pas prédire la valeur. *Random* vient du vieux français *randir* qui signifie courir.

peut prendre pour paramètre un entier seulement (d'autres paramètres optionnels existent) et fournit alors un résultat entier aléatoire supérieur ou égal à zéro et strictement inférieur au nombre rentré.

Nous pouvons une fonction pour créer des rectangles aléatoires :

à taper
```
def rectangle_aléatoire(largeur, hauteur):
    x1 = random.randrange(largeur)
    y1 = random.randrange(hauteur)
    x2 = random.randrange(largeur)
    y2 = random.randrange(hauteur)
    return canvas.create_rectangle(x1, y1, x2, y2)
```

Dans les deux premières lignes du corps de la fonction nous créons des variables pour le premier coin du rectangle en passant la largeur et la hauteur de la zone où nous allons dessiner nos rectangles. La fonction « `randrange` » avec un seul nombre en argument (pour plus de détail voir TOFIX) retourne un nombre aléatoire inférieur au nombre entré. Par exemple « `randrange(10)` » vous donnera un nombre entre 0 et 9 et « `randrange(100)` » vous donnera un nombre entre 0 et 99 et ainsi de suite. Les deux lignes suivantes créent les coordonnées pour le deuxième point. Puis nous créons le rectangle aléatoire avec ses variables. Ce rectangle aléatoire peut être un carré si $x_1+y_2=y_1+x_2$, c'est à dire si les quatre côtés sont égaux.

Nous pouvons utiliser notre fonction avec :

à taper
```
rectangle_aléatoire(500,500)
```

Nous pouvons aussi remplir le « `Canvas` » avec une centaine de rectangles aléatoires :

à taper
```
for x in range(0, 100):
    rectangle_aléatoire(500,500)
```

Au cas où vous vous seriez perdus votre programme complet doit donner :

à taper
```
import tkinter, random

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()

def rectangle_aléatoire(largeur, hauteur):
    x1 = random.randrange(largeur)
    y1 = random.randrange(hauteur)
    x2 = random.randrange(largeur)
    y2 = random.randrange(hauteur)
```

```
    return canvas.create_rectangle(x1, y1, x2, y2)

for x in range(0, 100):
    rectangle_aléatoire(500,500)

racine.mainloop()
```

Et devrait produire quelque chose qui ressemble à la Figure 9.3.

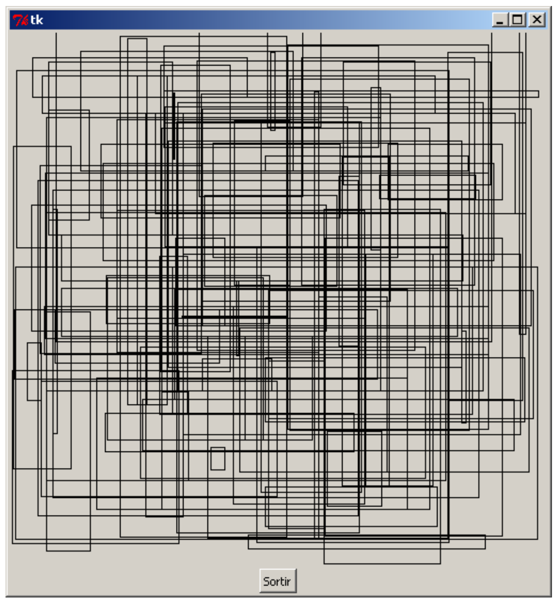


FIGURE 9.3 – Plein de rectangles

C'est un peu le bazar mais c'est intéressant, néanmoins. Notez au passage que chaque exécution du programme donne, très probablement, un résultat différent. Par conséquent votre résultat est sûrement différent du mien. Notez aussi comme l'affichage est rapide comparé à celui de « `turtle` ».

Vous rappelez-vous que dans le chapitre précédent, nous avions modifié la couleur de la tortue avec des pourcentages des trois couleurs primaires : rouge, vert et bleu ? Avec « `tkinter` » vous pouvez utiliser les couleur d'un manière similaire où d'une manière conçue initialement comme plus simple.

Commençons par la méthode faussement simple.

La méthode « `create_rectangle` » peut prendre un argument optionnel nommé « `fill` » [9]. cet argument est une chaîne de caractères. Le nom des couleurs en anglais peut être en utilisé par exemple (*red* signifie rouge) :

exemple

```
canvas.create_rectangle(0, 0, 500, 500, fill='red')
```

Nous pouvons définir notre fonction « `rectangle_aléatoire` » de manière à disposer d'un paramètre optionnel pour le remplissage.

9. Le nom *fill* signifie en anglais remplissage, du verbe « `to fill` » remplir.

à taper

```
def rectangle_aléatoire(largeur, hauteur, remplissage=None):
    x1 = random.randrange(largeur)
    y1 = random.randrange(hauteur)
    x2 = random.randrange(largeur)
    y2 = random.randrange(hauteur)
    if remplissage==None:
        canvas.create_rectangle(x1, y1, x2, y2)
    else :
        canvas.create_rectangle(x1, y1, x2, y2, fill=remplissage)
```

Nous pouvons alors créer une liste des couleurs en anglais [10].

à taper

```
liste_couleur=['black', 'white', 'green',
               'red', 'blue', 'orange',
               'yellow', 'pink', 'purple',
               'violet', 'magenta', 'cyan']

for couleur in liste_couleur :
        rectangle_aléatoire(500,500,couleur)
```

Cette méthode a quatre inconvénients :

– il faut connaître les noms des couleurs en anglais ;
– les noms des couleurs anglais ne correspondent pas forcément exactement aux même couleurs en français ;
– les couleurs nommées ne permettent pas de faire toutes les couleurs possibles ;
– les différents système d'exploitation (Linux, Mac Os X, Windows) n'ont pas tous les mêmes couleurs nommées (couleur différente voir absente ce qui crée une erreur).

Et maintenant la méthode faussement compliquée.

Avec le module « `turtle` » nous avions fait de l'orangé avec 100% de rouge, 85% de vert et pas de bleu. Dans « `tkinter` » nous pouvons créer un rectangle orangé en utilisant

à taper

```
rectangle_aléatoire(500,500,'#ffd800')
```

La chaîne « `#ffd800` » est en réalité une autre manière d'écrire des nombres. Le format que nous utilisons couramment est le format décimal : c'est à dire que nous utilisons dix chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Quand nous comptons nous commençons à 0 (en informatique en tout cas) et nous quand nous dépassons 9 nous utilisons deux chiffres 10 (1 puis 0).

Les ordinateurs sont beaucoup plus bêtes que nous et compte en réalité en binaire : ils utilisent deux chiffres : 0 et 1. Quand ils dépassent 1 (si si !) ils utilisent deux chiffres 10 (1 puis

10. Les couleurs utilisées sont : noir, blanc, vert, rouge, bleu, orange, jaune, rose, une sorte de violet, une autre sorte de violet, magenta, cyan. Remarquez que *purple* souvent traduit à tord pourpre et une sorte de violet en anglais.

0). La suite des nombre en binaire donne : 0, 1, 10, 11, 100, 101, 111, 1000, 1001... Par exemple deux mille neuf s'écrit en binaire 11111011001 ce qui est un peu long. Cette utilisation du binaire provient du fait que les ordinateurs ont des mémoires qui sont soient soit remplies soit vide. Cet élément d'information s'appelle un « bit » (prononcé bite ☺).

Il existe d'ailleurs une blague : « Il existe 10 *types de personnes dans le monde, ceux qui comprennent le binaire et les autres »* [11].

Nos grand anciens ont donc décidé d'utiliser une notation plus pratique pour les humains mais simple à utiliser l'hexadécimal : c'est à dire que nous utiliserons seize chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e et f. Quand nous comptons en hexadécimal nous commençons à zéro et quand nous dépassons f (quinze) nous utilisons deux chiffres 1 puis 0, c'est à dire 10. Généralement en informatique nous utilisons deux chiffres qui permettent de compter de zéro à deux cent cinquante cinq « `ff` » . Cet élément d'information s'appelle un « octet » (prononcé octé). L'octet correspond à huit bits. C'est cette unité de mesure qui est utilisée sous la forme de kilooctet [12], de mégaoctet [13], de gigaoctet [14] voir de teraoctet [15].

Revenons à notre couleur « `#ffd800` » comment lit on cette valeur ? En fait le # est là pour indiquer qu'il s'agit d'un nombre hexagonal puis viennent trois octets par couple de deux caractère : les deux premiers « `ff` » pour rouge, les deux suivant « `d8` » pour vert et les deux derniers « `00` » pour bleu. La valeur décimale se lit : seize fois le premier chiffre plus le second. Le nombre « `ff` » vaut seize fois quinze (f) plus quinze (deux cent cinquante cinq). Le nombre « `d8` » vaut seize fois treize « `d` » plus huit (deux cent seize). Inversement nous pouvons définir une fonction pour traduire les pourcentages de couleurs dans les valeurs utilisables par « `create_rectangle` ».

à taper
```
def couleurshex(rouge, vert, bleu):
    rouge = 255*(rouge/100.0)
    vert = 255*(vert/100.0)
    bleu = 255*(bleu/100.0)
    return '#%02x%02x%02x' % (rouge, vert, bleu)
```

Si nous appelons la fonction « `couleurshex` » avec 100 pour le rouge, 85 pour le vert et 0 pour le bleu nous donne dans le shell « `#ffd800` ».

à taper
```
print(couleurshex(100,85,0))
```

Nous pourrions écrire le code suivant pour créer un rectangle orangé.

exemple
```
rectangle_aléatoire(500,500,couleurshex(100,85,0))
```

11. Le 10 étant à comprendre en binaire c'est à dire deux.
12. Pour information il s'agit généralement de kibioctet, 2^{10} octets c'est à dire 1024 octets.
13. Souvent, il s'agit d'un mébioctet qui vaut 2^{20} (1024x1024) octets.
14. Souvent, il s'agit d'un gibioctet qui vaut 2^{30} (1024x1024x1024) octets.
15. Souvent, il s'agit d'un tébioctet qui vaut 2^{40} (1024x1024x1024x1024) octets.

Nous pouvons créer une couleur violet vif en utilisant 98% de rouge, 1% de vert et 77% de bleu :

exemple
```
rectangle_aléatoire(500,500,couleurshex(98,1,77))
```

Nous pouvons aussi créer une fonction « couleur_surprise ».

à taper
```
def couleur_surprise():
    return '#%02x%02x%02x' %(random.randrange(256),
                             random.randrange(256),
                             random.randrange(256))
```

Puis nous pouvons nous en servir avec « rectangle_aléatoire ». Pour information voila ce à quoi votre code doit ressembler.

à taper
```
import tkinter, random

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()

def rectangle_aléatoire(largeur, hauteur, remplissage=None):
    x1 = random.randrange(largeur)
    y1 = random.randrange(hauteur)
    x2 = random.randrange(largeur)
    y2 = random.randrange(hauteur)
    if remplissage==None:
        canvas.create_rectangle(x1, y1, x2, y2)
    else :
        canvas.create_rectangle(x1, y1, x2, y2, fill=remplissage)

def couleur_surprise():
    return couleurshex(randrange(256),randrange(256),randrange(256))


for x in range(0, 100):
    rectangle_aléatoire(500,500, couleur_surprise())

racine.mainloop()
```

Vous obtenez alors un résultat similaire à celui de la Figure 9.4. Ce programme vaut peut-être la peine d'être mis sur le bureau avec l'extension « .pyw » pour le monter à d'autres personnes ?

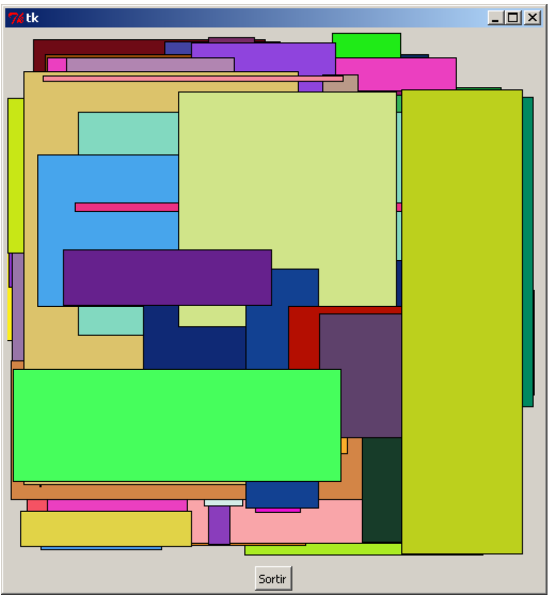


FIGURE 9.4 – Rectangles colorés aléatoires

9.5 Dessiner des ovales

Comme nous avons dessiné des rectangles, nous pouvons dessiner des ovales comme vous pouvez le voir sur la Figure 9.5.

à taper
```
canvas.create_oval(5, 5, 300, 200)
```

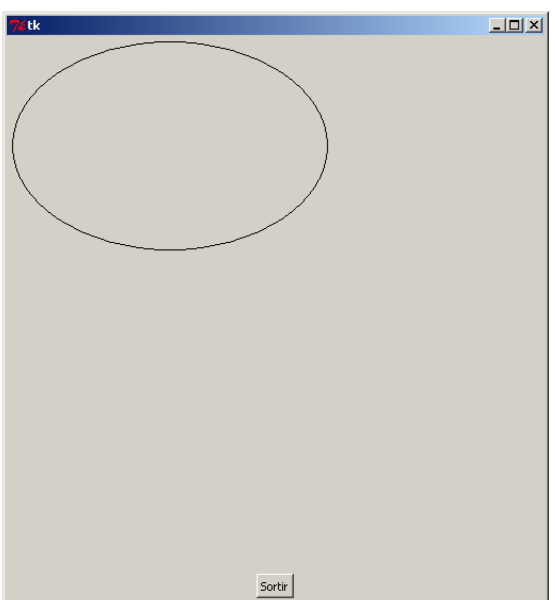


FIGURE 9.5 – Un ovale

L'ovale est créé avec la fonction « `create_oval` » inscrit dans un rectangle imaginaire décrit par les points (5,5) et (300,200). Dessinons un rectangle pour mieux comprendre cette explication.

à taper
```
canvas.create_rectangle(5, 5, 300, 200, outline="#ff0000")
canvas.create_oval(5, 5, 300, 200)
```

Le paramètre « `outline` » (bordure) permet d'indiquer la couleur du bord d'un rectangle ou d'un ovale. Ici « `#ff0000` » permet de dessiner un rectangle rouge. Vous pouvez le voir sur la Figure 9.6.

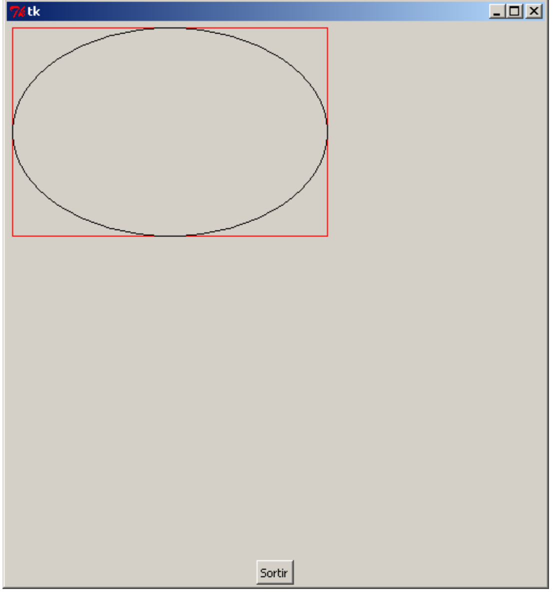


FIGURE 9.6 – Un ovale inscrit dans un rectangle

Cet exemple dessine d'abord un rectangle entre (5,5) et (300,200) puis un ovale dans qui est inscrit dans un rectangle défini par les coins (5,5) et (300,200). La fonction « `create_oval` » donnera exactement le même résultat sans avoir dessiner le rectangle auparavant.

En langage mathématique les ovales que nous dessinons s'appellent des ellipses. S'il est possible avec la commande « `create_oval` » des ellipses, il existe un type d'ellipse particulier que vous connaissez bien : le cercle ! Si le rectangle imaginaire dans lequel nous dessinons l'ellipse est un carré, le résultat est un cercle comme vous pouvez le voir sur la Figure 9.7.

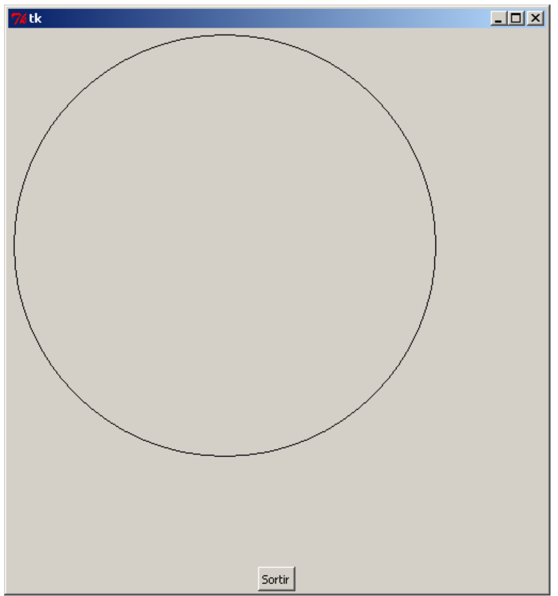


FIGURE 9.7 – Un cercle est une ellipse !

à taper
```
canvas.create_oval(5, 5, 300, 300)
```

9.6 Dessiner des arcs

Un arc de cercle est un bout de cercle. Dans « tkinter » nous pouvons dessiner des arcs de cercle ou d'ellipse. Pour rappel, une ellipse est une sorte de cercle aplati. Les arcs sont définis comme des portions d'ovale eux même décrits à partir de deux points comme à la section précédente.

à taper
```
canvas.create_arc(10, 10, 200, 100, extent=180, style='arc')
```

Cette commande crée un rectangle virtuel (qui n'existe pas) trace virtuellement une ellipse puis prend un bout de celle-ci. De la même manière que pour les cercles de « turtle » le paramètre « extent » indique l'extension de l'arc. Le style « 'arc' » indique quels sont les bords à tracer. Deux autres styles sont disponibles le mode par défaut « 'pieslice' » [16]qui trace les rayons depuis le centre du cercle ou de l'ellipse, et le style « 'chord' » [17] trace l'arc et la ligne entre les deux extrémités de l'arc.

Nous pouvons matérialiser cette construction avec le code suivant.

à taper
```
canvas.create_rectangle(10, 10, 400, 300, outline="#ff0000")
canvas.create_arc(10, 10, 400, 300, extent=180, style='arc')
```

Vous pouvez voir le résultat sur la Figure 9.8.

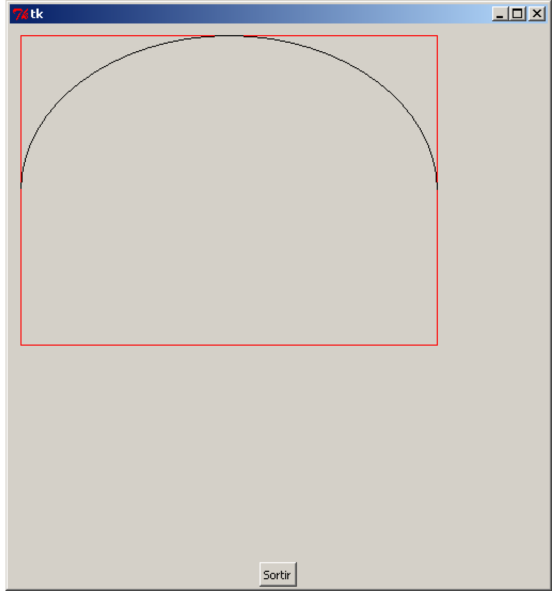


FIGURE 9.8 – Arc inscrit dans un rectangle

Si vous ne savez pas encore ce que sont les degrés rappelez vous le cercle des degrés de la Figure 3.6 page Figure 3.6. Si cela n'est pas clair rappelez vous que

– un arc de 90° est un arc d'un quart de tour ;
– un arc de 180° est un arc d'un demi tour ;
– un arc de 359° est un arc de presqu'un tour complet.

16. Le mot *pie slice* signifie part (pour *slice*) de tarte (pour *pie*).
17. Le nom *chord* signifie corde en anglais

Pour note : 360° sont en fait égaux à 0°, malheureusement quelques fois pour nous « `tkinker` » sait cela et ne dessinera rien si vous rentrez 360° ou 0°. Voici quelques lignes de code pour montrer quelques arcs descendant le long de la zone de dessin. Ainsi vous pouvez voir les différences entre différents angles d'arcs sur la Figure 9.9

à taper
```
canvas.create_arc(10, 10, 200, 100, extent=45, style='arc')
canvas.create_arc(10, 80, 200, 160, extent=90, style='arc')
canvas.create_arc(10, 160, 200, 240, extent=135, style='arc')
canvas.create_arc(10, 240, 200, 320, extent=180, style='arc')
canvas.create_arc(10, 320, 200, 400, extent=359, style='arc')
```

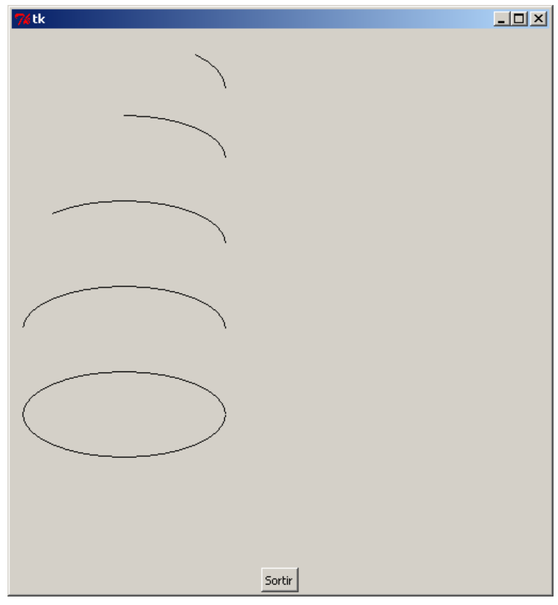


FIGURE 9.9 – Différents arcs

9.7 Dessiner des polygones

Un polygone est une figure avec trois côtés ou plus. Les triangles, les carrés, les rectangles, les pentagones, les hexagones et de nombreux autres sont tous des exemples de polygones. Les polygones peuvent être réguliers, avec des angles égaux de des cotés égaux, ou irréguliers.

La fonction « `create_polygon` » permet de créer des polygones à partir des coordonnées de sommets. Par exemple, pour tracer un triangle, vous avez besoin de fournir trois couples de coordonnées (un couple pour chaque commet). Voici un petit exemple que vous pouvez voir sur la Figure 9.10.

à taper ne sera pas répété
```
import tkinter, random

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
```

```
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()
```

à taper

```
canvas.create_polygon(10, 10, 100, 10, 100, 50)
```

à taper ne sera pas répété

```
racine.mainloop()
```

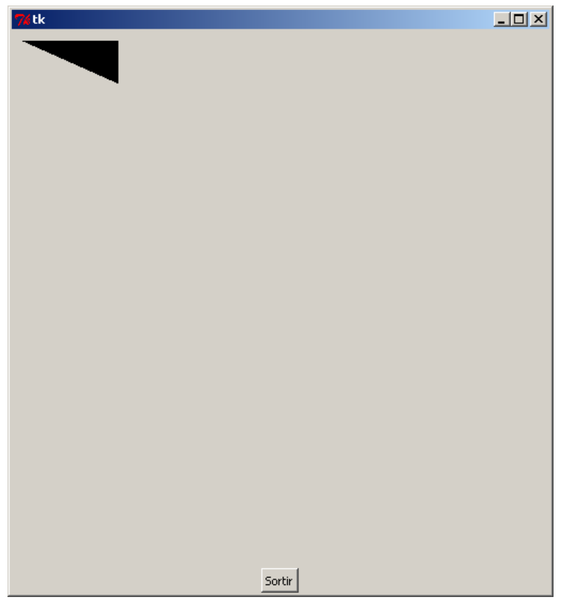


FIGURE 9.10 – Un triangle plein

Comme on le voit les polygones sont pas défaut remplis de noir. Le paramètre nommé « `fill` » permet de choisir le remplissage. Si l'on fixe ce paramètre à une chaîne vide « `""` », il n'y aura pas de remplissage.

à taper

```
canvas.create_polygon(10, 10, 100, 10, 100, 50, fill="")
```

Cette fonction ne donne aucun résultat. En effet les polygones sont remplis et sans bordure par défaut. Il convient donc de fixer une couleur de la bordure avec « `outline` ».

à taper

```
canvas.create_polygon(10, 10, 100, 10, 100, 50,
                      fill="", outline="#000000")
```

Nous pouvons aussi dessiner un autre polygone irrégulier et un polygone régulier.

à taper

```
canvas.create_polygon(10, 10, 100, 10, 100, 50,
                      fill="", outline="#000000")
canvas.create_polygon(200, 10, 240, 30,
                      120, 100, 140, 120,
                      fill="", outline="black")
canvas.create_polygon(260, 10, 500, 10,
                      500, 260, 260, 260,
                      fill="#00ff00", outline="black")
```

Vous pouvez voir le résultat sur la figure Figure 9.11.

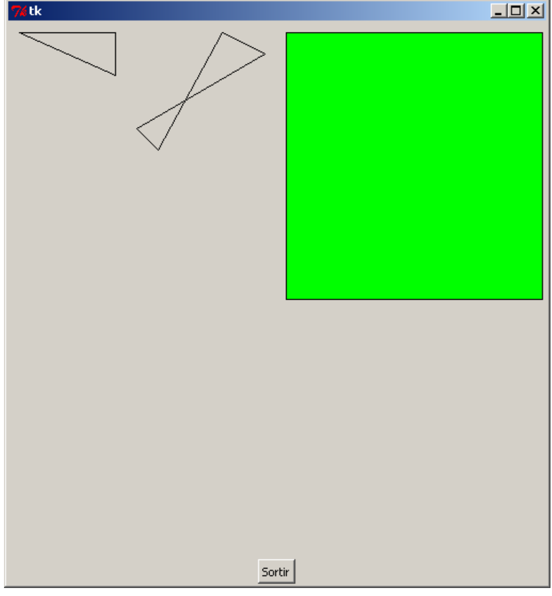


FIGURE 9.11 – Divers polygones

9.8 Dessiner des images

Nous avons besoin en premier lieu d'une image préexistante, Puis nous utiliserons la méthode « `image` » de l'objet « `canvas` ». Cela semble un peu illogique, mais cela fonctionne comme suit : Pour pouvoir manipuler une image, vous devez savoir où elle est pour pouvoir y accéder avec Python. Je vous propose de créer une image « `test.gif` » sur le bureau. En effet, Python (pour être exact tkinter) peut par défaut utiliser des images aux formats « `gif` », « `pgm` » et « `ppm` » [18].

à taper

```
import tkinter

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()


import os,platform

def chemin_bureau() :
    if platform.release()=='XP' :
        return os.path.join(os.environ['HOME'],"Bureau")
```

18. La bibliothèque « PIL », c'est à dire *Python imaging library*, peut être utilisée pour manipuler d'autres formats d'image.

```
    else :
        return os.path.join(os.environ['HOME'],"Desktop")

cheminfichier = os.path.join(chemin_bureau(),'test.gif')

monimage = tkinter.PhotoImage(file=cheminfichier)
canvas.create_image(0, 0, image=monimage, anchor=tkinter.NW)

racine.mainloop()
```

Dans les premières lignes nous inialisons une fenêtre et une zone de dessin comme précédemment. Puis l'image « `monimage` » est chargée à partir du fichier « `test.gif` » en utilisant la fonction « `tkinter.Photoimage` ».

Nous utilisons ensuite la méthode « `canvas.create_image` » de notre zone de dessin pour afficher notre image. Vous devriez voir apparaître quelque chose qui ressemble à la Figure 9.12.

FIGURE 9.12 – Image gif affichée avec Tk

Comme indiqué précédement PhotoImage peut traiter des fichiers d'images avec aux formats « `gif` », « `pgm` » et « `ppm` ». Il y a de nombreux types d'images différents qui peuvent être créés comme par exemple les images « `jpeg` » qui sont stockées par les appareils photos avec l'extension « `.jpg` ». La bibliothèque de traitement d'image de Python (*Python imaging library* ajoute la possibilité de traiter de charger la plupart des types d'images ainsi qui certaines méthodes de traitement comme agrandir ou réduire, changer les couleurs ou tourner les images. Néanmoins, installer et utiliser la bibliothèque de traitement d'image de Python et légèrement en dehors du sujet de ce livre [19].

19. La bibliothèque de traitement d'image de Python peut être trouvée à l'adresse `http://www.pythonware.com/products/pil/index.htm`. Malheureusement à la date d'écriture de ce livre (23 juillet 2009) le portage vers Python 3 n'a pas encore été achevé.

Notez que vous pouvez utiliser la fonction « `os.getcwd()` » pour connaître le répertoire courant utilisé par Python. Dans une ligne de commande ou un *shell* Python vous pouvez taper les commandes suivantes.

à taper
```
>>> import os
>>> print(os.getcwd())
C:\Python30
```

Il en résultera probablement quelque chose comme le chemin indiqué à la dernière ligne. Vous pouvez copier des ressources (comme les images) auxquelles vos programmes accèdent très souvent dans ce répertoire. Vous pourriez par exemple y copier l'image « `test.gif` » et la charger avec la commande « `monimage = tkinter.PhotoImage(file=testk.gif)` ».

9.9 Aminimations simples

Jusqu'à maintenant nous avons dessiné des images qui ne bougeaient pas. Quand est-il des animations ? Les animations ne sont pas forcément le point fort de Tk, mais je peux vous y initier.

Les animations ne sont d'ailleurs pas non plus le point fort des livres !

Par exemple nous pouvons créer un triangle puis le faire bouger à travers l'écran avec le code suivant :

à taper
```
import tkinter

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()


import time

numéro_forme=canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 100):
    canvas.move(numéro_forme, 5, 0)
    racine.update()
    time.sleep(0.05)

racine.mainloop()
```

Juste après avoir lancé le programme un triangle va commencer à se déplacer (vous pouvez le voir à mi-chemin sur la Figure 9.10. Comment cela fonctionne-t-il ?

Après une initialisation nous dessinons un triangle avec « `canvas.create_polygon` ». Nous stockons le numéro de la forme retourné par « `canvas.create_polygon` » dans la variable « `numéro_forme` », pour information il s'agit de la forme numéro un. Puis nous initialison un itérateur qui va prendre les valeurs de un à quatre-vingt dix-neuf. Le bloc utilisé juste après est le code pour bouger le triangle lentement. Tout d'abord la méthode « `canvas.move` »[20] prend trois argument le numéro d'un élément à bouger, le nombre de points horizontaux vers la droite pour le déplacement (un nombre négatif crée un déplacement vers la gauche) et le nombre de points verticaux vers le bas (un nombre négatif crée un déplacement vers le haut). Dans notre exemple le triangle (objet un) est déplacé à chaque pas de cinq points vers la droite à chaque pas de l'itérateur. Puis nous utilisons la méthode « `racine.update` » pour obliger l'objet racine à se mettre à jour (*update*). Si nous n'indiquons pas cela tkinter attendrait jusqu'à la fin de la boucle avant de bouger le triangle à sa position finale. Enfin, nous disons à Python d'attendre cinq centièmes de seconde avec la méthode « `time.sleep` » du module « `time` »[21].

Nous pouvons si nous le souhaitons ajouter une deuxième boucle en utilisant « canvas.move(numéro_forme, -5, 0) » pour ramener le triangle à sa position initiale.

Nous pouvons aussi déplacer le triangle en diagonale par exemple.

à taper

```
import tkinter

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()


import time

numéro_forme=canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 100):
    canvas.move(numéro_forme, 5, 5)
    racine.update()
    time.sleep(0.05)
for x in range(0, 100):
    canvas.move(numéro_forme, -5, -5)
    racine.update()
    time.sleep(0.05)


racine.mainloop()
```

20. Le mot «*move*» signifie bouger en anglais de mouvoir.
21. Le mot *sleep* signifie dormir et le mot *time* signifie temps.

9.10 Réagir aux événements

Nous pouvons aussi faire réagir le triangle quand quelqu'un presse une touche en liant une action à un événement. Les événements sont des choses qui arrivent pendant que le programme fonctionne et qu'il peut traiter comme le fait que quelqu'un presse une touche, actionne un bouton ou même ferme une fenêtre. Vous pouvez configurer Tk pour surveiller ces événements et faire quelque chose en réaction. Pour commencer à gérer les événements nous avons besoin de commencer par créer une fonction. Supposons que nous voulons que le triangle bouge vers le bas quand la touche entrée est pressée. Nous pouvons définir une fonction pour déplacer le triangle.

ne pas taper

```
def déplacertriangle(event):
    canvas.move(1, 5, 0)
```

Figure 9.14 : The triangle moving down the screen.

La fonction « `déplacerobjet` » doit avoir un seul paramètre « `event` » (événement en anglais) pour réagir à un événement. Ainsi nous disons à Python que cette fonction doit être utilisé pour être lié à un événement.

à taper

```
import tkinter

racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=500, height=500)
canvas.pack()
bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()


import time

numéro_forme=canvas.create_polygon(10, 10, 10, 60, 50, 35)
numéro_figure=canvas.create_polygon(110, 110, 110, 160, 150, 135)

def déplaceobjet(event):
    canvas.move(numéro_figure, 0, 3)

canvas.bind_all('<KeyPress-Return>', déplaceobjet)

racine.mainloop()
```

Le premier paramètre de la méthode « `canvas.bind_all` » (*bind :* lier, *all :* tout) quel est l'événement que nous voulons voir surveiller par Tk. Dans ce cas précis, il s'agit de l'événement « `KeyPress-Return` » qui à lieu lorsqu'on presse la touche (*key*) entrée (*return*). Le deuxième argument à Tk quelle fonction est associée à cet événement ; ici nous

utilisons la fonction « déplaceobjet ». Nous disons à Tk quelle fonction lancer quand la touche entrée est pressée. Lancez ce code, contrôlez que la fenêtre crée est sélectionnée et appuyez sur entrée.

Attelons nous maintenant à changer la direction de déplacement du triangle. Tout d'abord changeons la fonction « déplacerobjet » de manière à ce quelle réagisse différemment selon la touche appuyée.

à taper
```
def déplaceobjet(event):
    if event.keysym == 'Up':
        canvas.move(numéro_figure, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(numéro_figure, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(numéro_figure, -3, 0)
    else:
        canvas.move(numéro_figure, 3, 0)
```

L'objet « event » qui est passé à « déplaceobjet » contient de nombreuses attributs [22]. Un de ces attribut est « keysym » qui est une chaîne qui contient le nom de la touche appuyée en anglais. Si « keysym » contient la chaîne « Up », nous appelons « canvas.move » avec les paramètres « (numéro_figure, 0, -3) » de manière à déplacer la forme identifiée par son numéro de zéro pixels vers la droite et de moins trois pixels vers le bas c'est à dire de trois pixels vers le haut. Si la chaîne est « Down » nous déplaçons l'objet vers le bas, si c'est « Left » vers la gauche et sinon vers la droite. Notez que nous aurions pu choisir de déplacer l'objet vers la droite seulement quand la touche droite « Right » est pressée.

à taper
```
import tkinter
racine = tkinter.Tk()
canvas = tkinter.Canvas(racine, width=800, height=500)
canvas.pack()


numéro_figure=canvas.create_polygon(110, 110, 110, 160, 150, 135)


def déplaceobjet(event):
    if event.keysym == 'Up':
        canvas.move(numéro_figure, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(numéro_figure, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(numéro_figure, -3, 0)
    else:
```

22. Les attributs sont des valeurs nommées qui décrivent quelque chose de l'objet. Par exemple, un attribut du ciel est sa couleur qui est bleu quelque fois, un attribut d'une voiture est sont nombre de places. En programmation, un attribut a un nom et une valeur.

```
        canvas.move(numéro_figure, 3, 0)

canvas.bind_all('<KeyPress>', déplaceobjet)

racine.mainloop()
```

Avec cet exemple le triangle se déplace maintenant dans la direction de la flèche que vous pressez ou vers la droite pour toutes les touches.

10 Où aller à partir de ce point ?

Congratulations ! Vous avez été jusqu'au bout. Ce que vous avez probablement appris de ce livre sont les concepts fondamentaux qui rendront l'apprentissage d'autres langages de programmation beaucoup plus simple. Même si Python est un langage de programmation brillant, un langage particulier n'est pas toujours le meilleur outil pour toutes les tâches. Donc n'ayez pas peur de regarder pour d'autres manières de programmer votre ordinateur si cela vous intéresse.

Par exemple, si vous êtes intéressés dans la programmation de jeux, vous pouvez peut-être regarder quelque chose comme BlitzBasic (www.blitzbasic.com) qui utilise le langage de programmation Basic. Ou encore, Flash qui est utilisé dans de nombreux sites web pour des animations et des jeux ; par exemple le site de Nickelodeon (www.nick.com) utilise beaucoup de Flash. Si vous êtes intéressez par la programmation en Flash un bon départ sera sûrement « Flash CS4 pour les nuls » un livre qui existe en version poche ; vous pouvez aussi lire « ActionScript 3 - Développez des jeux en Flash ». Recherchez « jeu Flash » sur http://www.eyrolles.com ou sur www.amazon.fr. Certains autres livres comme « Hands on Darkbasic Pro : A Self-study Guide to Games Programming : v. 2 » ou « Game Programming for Teens » n'existent qu'en anglais. Soyez conscients que les outils de développement BlitzBasic, DarkBasic et Flash coûtent de l'argent et que les livres en parlant sont généralement payants à l'inverse de Python. L'implication de vos parents doit donc être acquise avant même de pouvoir commencer.

NDT : Les langages les plus utilisés pour programmer sont le C et le C++ pour les jeux sur ordinateurs et consoles et Java (J2ME) pour les téléphones cellulaires. Ces langages ont l'avantage d'être gratuits et de disposer de nombreuses documentations. Par contre ils sont moins simples d'abord que le Basic. Pour les sites web le JavaScript est maintenant utilisé pour des applications complexes.

Si vous voulez continuer avec Python pour faire des jeux vous devriez aller sur www.pygame.org où vous trouverez la bibliothèque la plus utilisée pour faire des jeux. Deux livres en anglais pourront alors vous être utiles : « Beginning Game Development with Python and Pygame : From Novice to Professional » et « Game Programming With Python ».

Si vous n'êtes pas spécialement intéressés dans la programmation de jeux mais que vous voulez apprendre plus à propos de Python vous pouvez commencer par « Apprendre à programmer avec Python » http://www.framasoft.net/IMG/pdf/python_notes-2.

pdf. Vous pouvez aussi jeter un coup d'œil à « Dive into Python » de Mark Pilgrim www.diveintopython.org. Il y a aussi un tutoriel libre disponible sur http://docs.python.org/tut/tut.html.

Il y a tout un ensemble de sujets que nous n'avons pas couvert dans cette simple introduction donc, tout du moins pour Python, il y a encore beaucoup à apprendre et avec quoi s'amuser. Bonne chance et amusez vous à programmer.

Réponses aux « À vous de jouer »

Vous trouverez ici les réponses aux questions posées dans la plupart des chapitres dans la section « À vous de jouer ».

A.1 Réponses aux cas pratiques de la section 2.9

1. La réponse à l'**exercice 1** devrait ressembler à ce qui suit :

à taper
```
>>> jouets= [ 'Lego', 'Playmo', 'mandala', 'vélo' ]
>>> plats = [ 'crêpes', 'gaufres', 'betterave' ]
>>> préférés = jouets + plats
>>> print(préférés)
['Lego', 'Playmo', 'mandala', 'vélo', 'crêpes', 'gaufres', 'betterave']
```

2. La réponse à l'**exercice 2** est simplement d'ajouter le résultat de « 3*25 » et le résultat de « 10*32 ». L'équation suivant montre le résultat de cette équation :

à taper
```
>>> print(3 * 25 + 10 * 32)
395
```

Ce qui fait beaucoup de friandises !

Néanmoins, si vous avez suivi la partie sur l'usage de parenthèses dans le chapitre 2, vous pouvez avoir décidé de mettre des parenthèses dans cette équation. Vous devriez avoir fait quelque chose comme :

à taper
```
>>> print((3 * 25) + (10 * 32))
395
```

La réponse est la même, car la multiplication est faite avant l'addition. Dans les deux équations les deux multiplications sont faites en premier et les résultats sont additionnés. Malgré tout la seconde équation est peut-être légèrement meilleure que la

première. En effet, l'ordre des opérations est immédiatement évident pour le lecteur. Un programmeur moins compétent (qui ne connaîtrait pas aussi bien l'ordre des opérations) pourrait penser que dans la première équation vous multipliez 3 par 25, puis additionnez 10 et multipliez le résultat par 32 (la réponse complément fausse est 2720). Avec les parenthèses, il est un peu plus simple de comprendre ce qui est calculé en premier.

3. La réponse à l'**exercice 2** devrait ressembler à :

à taper
```
>>> prénom = 'Morgane'
>>> nom = 'Paul'
>>> print('Mon nom est %s %s.' % (prénom, nom))
Mon nom est Morgane Paul.
```

A.2 Réponses aux cas pratiques de la section 3.3

1. La réponse à l'**exercice 1** devrait ressembler à ce qui suit : un rectangle est semblable à un carré mis à part que deux côtés peuvent être plus longs que les deux autres. En disant à la tortue de faire les actions suivantes, vous pouvez aisément dessiner un rectangle :
   – avancer d'un certain nombre de points ;
   – tourner à angle droit ;
   – avancer d'un certain nombre de points ;
   – tourner à angle droit dans le même sens que la première rotation ;
   – avancer du même nombre de points que pour le premier déplacement ;
   – tourner à angle droit dans le même sens que la première rotation ;
   – avancer du même nombre de points que pour le deuxième déplacement.

   Par exemple, le code suivant tracera un rectangle que vous pouvez voir sur la Figure A.1.

à taper
```
import turtle

tortue = turtle.Pen()
tortue.forward(150)
tortue.left(90)
tortue.forward(50)
tortue.left(90)
tortue.forward(150)
tortue.left(90)
tortue.forward(50)
```

2. La réponse à l'**exercice 2** devrait ressembler à ce qui suit :

   Un triangle est un peu plus compliqué à dessiner car vous avez besoin de connaître plus d'informations sur les angles et les longueurs des côtés. Si vous n'avez pas étudié les angles à l'école cela pourrait être un peu plus difficile que vous ne vous y attendiez. Vous pouvez dessiner un triangle simple (voir la Figure A.2 avec le code suivant :

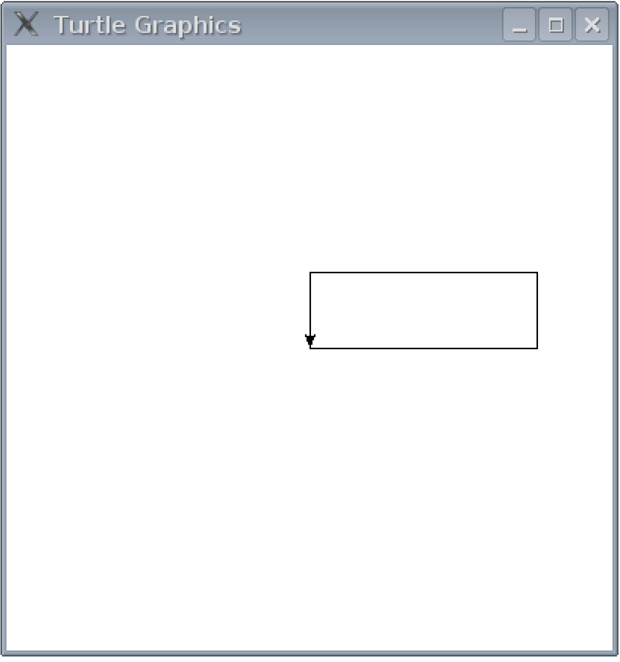


Figure A.1 – Tortue dessinant un rectangle.

à taper
```
import turtle

tortue = turtle.Pen()
tortue.forward(100)
tortue.left(135)
tortue.forward(70)
tortue.left(90)
tortue.forward(70)
```

A.3 Réponses aux cas pratiques de la section 5.4

1. La réponse à l'**exercice 1** est que seule la chaîne `x vaut 0` sera affichée. Regardons le code en détail pour comprendre pourquoi.

   ne pas taper
   ```
   >>> for x in range(0, 20):
   ...         print('x vaut %s' % x)
   ...         if x < 9:
   ...                 break
   x vaut 0
   ```

   La raison en est que durant la première itération la valeur de variable `x` est zéro. Comme zéro vaut moins que neuf, la commande `break` est lancée et fait sortir de la boucle.

2. Pour répondre à l'**exercice 2** je vous propose d'écrire un petit programme qui va calculer le nombre de nénuphars chaque jour et l'afficher. Quand le nombre de nénuphars dépassera mille il suffit d'arrêter la boucle.

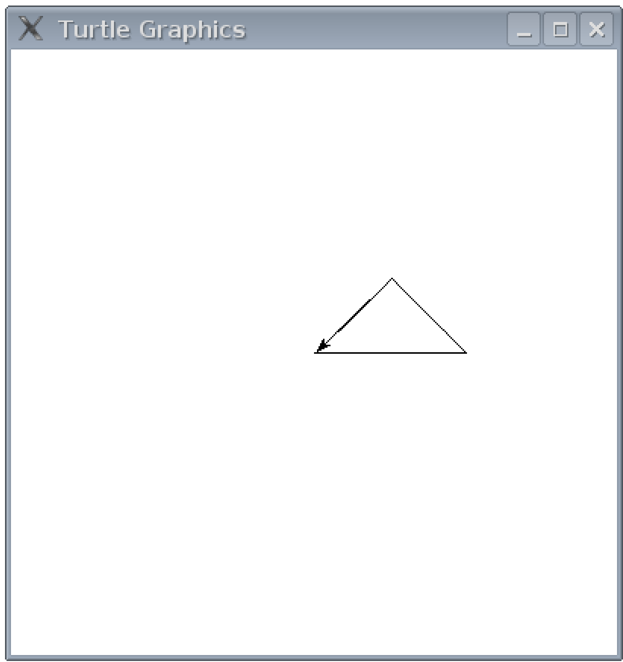


FIGURE A.2 – La tortue dessine un triangle

Pour information voila à quoi peut ressembler un tel programme.

ne pas taper

```
>>> jours = 0
>>> nénuphars = 1
>>> while nénuphars < 1000:
...     jours+=1
...     nénuphars*=2
...     print('Au bout de %s jours, il y a %s nénuphars.'
...            % (jours, nénuphars))
...
Au bout de 1 jours, il y a 2 nénuphars.
Au bout de 2 jours, il y a 4 nénuphars.
Au bout de 3 jours, il y a 8 nénuphars.
Au bout de 4 jours, il y a 16 nénuphars.
Au bout de 5 jours, il y a 32 nénuphars.
Au bout de 6 jours, il y a 64 nénuphars.
Au bout de 7 jours, il y a 128 nénuphars.
Au bout de 8 jours, il y a 256 nénuphars.
Au bout de 9 jours, il y a 512 nénuphars.
Au bout de 10 jours, il y a 1024 nénuphars.
>>> print("Il faut seulement %s jours pour
...que le lac soit envahi." % jours)
Il faut seulement 10 jours pour que le lac soit envahi.
```

Aux deux premières lignes nous initialisons les variables « `nénuphars` » à un et « `jours` » à zéro. À la troisième ligne nous créons une boucle « tant que » en utilisant comme condition d'arrêt que le nombre de nénuphars devient supérieur ou égal à mille. Puis,

dans le bloc nous augmentons le nombre de jours de un jour à chaque pas et multiplions le nombre de nénuphars par deux à chaque pas. Enfin, nous affichons le résultat du calcul de chaque pas. Une fois sortis de la boucle nous synthétisons le résultat.

A.4 Réponses aux cas pratiques de la section 6.3

1. La réponse à l'**exercice 1** devrait ressembler à ce qui suit :

   Transformer la boucle tant que en fonction est vraiment très simple. Votre fonction devrait ressembler à ça.

   à taper
   ```
   >>> def calcul_nénuphars(nénuphars,coefficient):
   ...     jours=0
   ...     while  nénuphars < 1000:
   ...         jours+=1
   ...         nénuphars*=coefficient
   ...         print('Au bout de %s jours, il y a %s nénuphars.' %
   ...(jours, nénuphars))
   ...     print("Avec un coefficient de %s, Il faut %s
   ...jours pour que le lac soit envahi" % (coefficient,jours))
   ```

   Si vous comparez la fonction avec le code initial, vous devriez remarque que, sauf pour la première ligne, le code n'a presque pas changé. Seul le « `*=2` » a été changé en « `*=coefficient` ». La dernière dernière ligne a, elle aussi, était modifiée mais uniquement pour changer l'information affichée. Comme « `nénuphars` » était déjà une variable, il n'y a pas de changement à faire quand nous l'utilisons comme un paramètre. Vous pouvez contrôler que le résultat est le même quand vous utilisez le code dans une fonction ou en dehors [1].

   à taper
   ```
   >>> calcul_nénuphars(1, 2)
   Au bout de 1 jours, il y a 2 nénuphars.
   Au bout de 2 jours, il y a 4 nénuphars.
   Au bout de 3 jours, il y a 8 nénuphars.
   Au bout de 4 jours, il y a 16 nénuphars.
   Au bout de 5 jours, il y a 32 nénuphars.
   Au bout de 6 jours, il y a 64 nénuphars.
   Au bout de 7 jours, il y a 128 nénuphars.
   Au bout de 8 jours, il y a 256 nénuphars.
   Au bout de 9 jours, il y a 512 nénuphars.
   Au bout de 10 jours, il y a 1024 nénuphars.
   Avec un coefficient de 2, Il faut 10 jours pour que le lac soit
   envahi.
   ```

2. La réponse à l'**exercice 2** devrait ressembler à ce qui suit :

1. Une bonne habitude de de créer le maximum de fonction, ce n'est pas compliqué à faire et on ne sait jamais : cela pourra toujours être utilisé ailleurs.

à taper
```
>>> def calcul_nénuphars(nénuphars,coefficient,jours_max):
...     jours=0
...     while  jours < jours_max:
...         jours+=1
...         nénuphars*=coefficient
...     print('''Au bout de %s jours avec un coefficient de %s,
...il y a %s nénuphars.''' % ...(jours, coefficient, nénuphars))
```

Changer la fonction pour passer le nombre de jours comme un paramètre n'a demandé qu'un faible nombre de changements.

Nous pouvons maintenant facilement changer le nombre de nénuphars, le coefficient de croissance et le nombre de jours :

à taper
```
>>> calcul_nénuphars(2, 1.5, 20)
Au bout de 20 jours avec un coefficient de 1.5,
il y a 6650.51346016 nénuphars.
```

Nous remarquons que notre programme permet d'avoir des bouts de nénuphars, en effet le nombre de nénuphars est un nombre à virgule. Nous imaginerons que les bouts de nénuphars sont des bébés ou des petits nénuphars. Python pourrait gérer ce genre de problème mais ce n'est pas l'objet de ce livre.

3. La réponse à l'**exercice 3** devrait ressembler à ce qui suit : Si vous avez gardé la session python ouverte vous n'avez pas à redéfinir la fonction « `calcul_nénuphars` ». De plus les questions affichées sont aussi optionnelles, vous apprendrez dans le chapitre suivant à sauvegarder vos programmes et l'affichage des questions deviendra alors importante car elle seront posées à des tiers (d'autres personnes).

à taper
```
>>> def calcul_nénuphars(nénuphars,coefficient,jours_max):
...     jours=0
...     while  jours < jours_max:
...         jours+=1
...         nénuphars*=coefficient
...     print('Au bout de %s jours avec un coefficient de %s,
...il y a %s nénuphars.' % ...(jours, coefficient, nénuphars))
>>> import sys
>>> def demande() :
>>>     print('Entrez le nombre initial de nénuphars :')
>>>     rnénuphars = float(sys.stdin.readline())
>>>     print('Entrez le nombre de coefficient :')
>>>     rcoefficient = float(sys.stdin.readline())
>>>     print('Entrez le nombre de jours :')
>>>     rjours = float(sys.stdin.readline())
>>>     calcul_nénuphars(rnénuphars, rcoefficient, rjours)
>>> demande()
```

```
Entrez le nombre initial de nénuphars :
2
Entrez le nombre de coefficient :
1.2
Entrez le nombre de jours :
3
```

A.5 Réponses aux cas pratiques de la section 8.6

1. Pour répondre à la première question il y a la manière facile et la manière difficile. La manière difficile n'est pas difficile car elle est compliquée. Elle est difficile car elle demande de taper beaucoup de choses :

   na pas saisir

   ```
   import turtle
   tortue = turtle.Pen()
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   tortue.right(45)
   tortue.forward(50)
   ```

   Vous pouvez voir dans ce code que nous disons à la tortue d'avancer de cinquante pixels puis de tourner à droite de quarante-cinq degrés. Nous le faisons huit fois ! Ce qui fait sept fois de trop. La manière la plus simple est présentée ci-dessous, elle produit aussi l'octogone que vous pouvez observer sur la Figure A.3.

   à taper

   ```
   import turtle
   tortue = turtle.Pen()
   for x in range(0,8):
       tortue.forward(50)
       tortue.right(45)
   ```

2. Si vous regardez à nouveau les fonctions étudiées dans chapitre 8, vous verrez comme créer une forme remplie. Nous pouvons convertir notre code de dessin d'un octogone

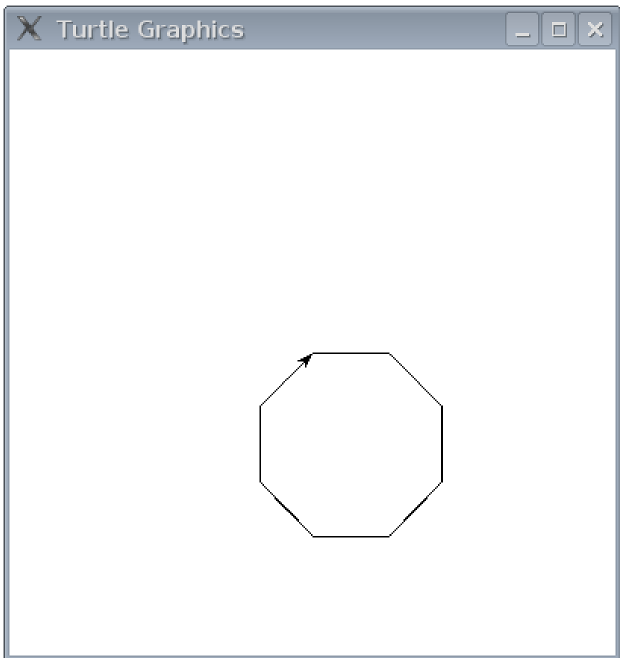


FIGURE A.3 – La tortue dessine un octogone.

en une fonction qui utilise une couleur et nous voulons aussi pouvoir réutiliser cette fonction par la suite.

à taper

```
import turtle
tortue = turtle.Pen()
def octogone(red, green, blue):
    tortue.color(red, green, blue)
    tortue.begin_fill()
    for x in range(0,8):
        tortue.forward(50)
        tortue.right(45)
    tortue.end_fill()
octogone(0, 0, 1)
```

Nous fixons une couleur puis démarrons le remplissage. Puis nous lançons la boule pour dessiner un octogone, puis nous arrêtons le remplissage pour que Python remplisse la forme que nous venons de dessiner.

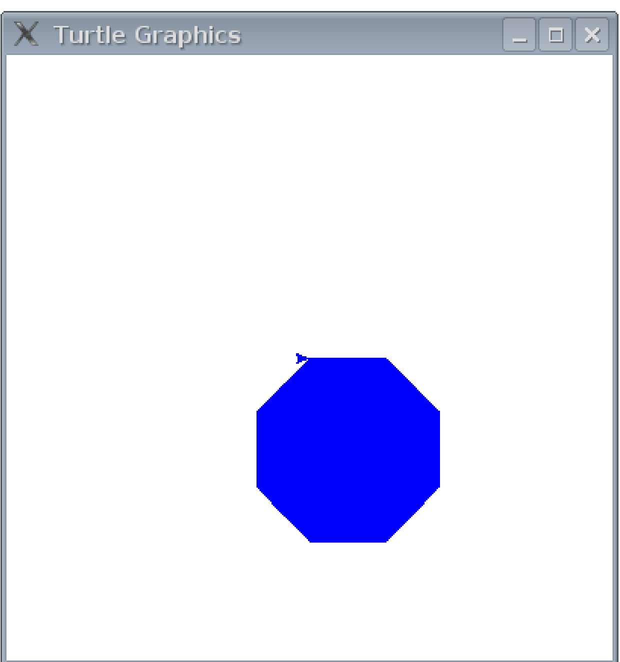


FIGURE A.4 – La tortue dessine un octogone bleu.

B

Quelques fonctions intégrées

Python a des fonctions intégrées — fonctions qui peuvent être utilisées sans avoir besoin de les importer avec « `import` » avant de pouvoir les utiliser. Certaines de ces fonctions couramment utilisées sont listées si dessous.

abs(x)

La fonction « `abs` » retourne la valeur absolue d'un nombre. La valeur absolue est un nombre qui est toujours positif qui correspond à la distance du nombre par rapport à zéro. Retourne la valeur absolue d'un nombre. L'argument doit être un entier simple ou long, ou un nombre à virgule flottante réel ou complexe. Si l'argument est un nombre complexe, sa magnitude est retournée (racine de la somme des carrés des parties réelle et imaginaire).

La valeur absolue de 10 est 10 et la valeur absolue de -20,5 est 20,5.

```
>>> print(abs(10))
10
>>> print(abs(-20.5))
20.5
```

bool(x)

La fonction « `bool` » retourne soit vrai ou faux selon la valeur passée en paramètre. Cette fonction retourne généralement vrai sauf si :

- la variable vaut « `None` », zéro ou est un conteneur vide comme « `''` », « `()` », « `[]` » ou « `{}` » ;
- l'objet possède une méthode « `__iszero__` » qui retourne vrai ou une méthode « `__len__` » qui retourne zéro.

Un objet utilisé dans un test sera traité comme si la commande « `bool` » était utilisée au préalable.

```
>>> print(bool(0))
False
>>> print(bool(1))
```

```
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

Pour d'autres objets « `bool` » retourne vrai pour toute valeur non nulle.

```
>>> print(bool(None))
False
>>> print(bool(''))
False
>>> print(bool('a'))
True
```

cmp

La fonction « `cmp` » **comp**are deux valeurs et retourne un nombre négatif si la première valeur est inférieure à la seconde ; retourne zéro si les deux valeurs sont identiques et retourne un nombre positif si la première valeur est plus grande que la seconde. Par exemple un vaut moins que deux :

```
>>> print(cmp(1,2))
-1
```

Deux est égal à deux :

```
>>> print(cmp(2,2))
0
```

Par contre deux est plus grand que un.

```
>>> print(cmp(2,1))
1
```

On peut comparer autre chose que des nombres. Vous pouvez utiliser d'autre valeurs comme des chaînes.

```
>>> print(cmp('a','b'))
-1
>>> print(cmp('a','a'))
0
>>> print(cmp('b','a'))
1
```

Mais faites attention avec les chaînes ; la valeur retournée pourrait ne pas être exactement ce à quoi vous vous attendiez...

```
>>> print(cmp('a','A'))
1
>>> print(cmp('A','a'))
-1
```

Un « a » minuscule est en fait plus grand « A » majuscule, bien sûr…

```
>>> print(cmp('aaa','aaaa'))
-1
>>> print(cmp('aaaa','aaa'))
1
```

Mais une chaîne de trois a, « aaa », est plus petite qu'une chaîne de quatre a, « a ».

dir

La fonction « dir » retourne une liste d'information à propos d'une valeur. Vous pouvez l'utiliser sur des chaînes, des nombres, des fonctions, des modules, des objets, des classes et presque n'importe quoi. Sur certaines choses, l'information ne pas être très utile, voir ne pas avoir beaucoup de sens. Par exemple, lancer « dir » sur le nombre un résulte…

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__',
'__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__getattribute__',
'__getnewargs__', '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__',
'__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__', '__nonzero__',
'__oct__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__str__', '__sub__', '__truediv__',
'__xor__']
```

Ce qui fait un grand nombre de fonctions spéciales. Ou alors appeler la fonction « dir » sur la chaîne « 'a' » donne :

```
>>> dir('a')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__',
'__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Ce résultat montre qu'il y a des méthodes utiles comme « capitalize » qui change la première lettre d'une chaîne en capitale et qu'on utilisera de préférence pour les majuscules.

```
>>> print('jean'.capitalize())
Jean
```

Il y a aussi les méthodes :

- « isalnum » qui retourne vrai si une chaîne est alphanumérique, c'est dire contient des lettres et des chiffres ;
- « isalpha » qui retourne vrai si une chaîne contient seulement des lettres ;
- et ainsi de suite.

La fonction « dir » peut être utile quand vous avez une variable et que vous voulez rapidement savoir ce que vous voulez faire avec.

enumerate

La fonction « enumerate » est utilisée à chaque fois que l'on veut parcourir une liste ou tout conteneur itérable en disposant de l'index et de la valeur de chaque élément. Le résultat de la fonction enumerate peut être énuméré comme une liste qui contient les doublets « (index,élément) ».

```
>>> for saison in ['printemps', 'été', 'automne', 'hiver']:
...     print(saison)
printemps
été
automne
hiver
>>> for i, saison in enumerate(['printemps', 'été', 'automne', 'hiver']):
...     print(i, saison)
0 printemps
1 été
2 automne
3 hiver
```

eval

La fonction « eval » prend une chaîne comme paramètre et l'exécute comme une expression Python. Elle est similaire à l'instruction « exec » [1] bien que légèrement différente. Avec « exec » vous pouvez créer des mini-programmes en Python à partir de votre chaîne, avec des boucles et des fonctions. La fonction « eval » permet seulement d'exécuter des expressions simples comme par exemple :

```
>>> x=10
>>> y=5
>>> à_évaluer="%s*%s" % (x,y)
>>> print(à_évaluer)
10*5
>>> eval(à_évaluer)
50
```

1. La fonction « exec » n'est pas présentée dans ce livre.

exec

La fonction « `exec` » est une fonction qui est utilisée pour convertir des chaînes en code python. Par exemple, vous pouvez créer une variable avec une chaîne comme suit :

```
>>> myvar = 'hello there'
```

Puis imprimer le contenu de cette variable

```
>>> print(myvar)
hello there
```

Mais vous pouvez aussi y mettre à la place du code Python.

```
>>> myvar = 'print("hello there")'
```

Et alors, vous pouvez utiliser « `exec` » pour transformer cette chaîne en un mini programme Python et l'exécuter.

```
>>> exec(myvar)
hello there
```

C'est une idée étrange et elle pourrait ne pas avoir de sens jusqu'à ce que vous en aillez besoin. Comme « `assert` », il s'agit d'une de ces fonctionnalités avancées qui sont utilisées dans des certains programmes sophistiqués.

file

La fonction « `file` » est utilisée pour ouvrir un fichier et retourner un objet de type « `file` » qui dispose de méthodes qui permettent d'accéder aux informations contenues dans le fichier. Ces méthodes permettent par exemple d'accéder au contenu d'un fichier ou à sa taille. Vous pouvez trouver plus d'informations à propos de la fonction « `file` » et des objets « `file` » dans le chapitre 7.

float

La fonction « `float` » convertit une chaîne ou un nombre en nombre à virgule flottante. Un nombre à virgule flottante est un nombre à virgule contenant un nombre limité de chiffres (il s'agit d'un type particulier de nombre rationnel). En Python le séparateur décimal n'est pas la virgule contrairement à l'usage français mais le point conformément à l'usage anglo-saxon. Par exemple, le nombre « `10` » est un entier (*integer*) mais les les nombres « `10.0` », « `10.1` » ou « `10.253` » sont des nombres à virgule flottante (*float*).

Vous pouvez convertir une chaîne en nombre à virgule flottante en passant la chaîne en paramètre à la fonction « `float` » :

```
>>> float('12')
12.0
```

Vous pouvez bien sûr utiliser une chaîne contenant un point (l'équivalent de la virgule) :

```
>>> float('123.456789')
123.456789
```

Un nombre entier peut être converti en un nombre à virgule flottante en utilisant « `float` » :

```
>>> float(200)
200.0
```

Bien sûr, convertir un nombre à virgule flottante retourne juste un nombre à virgule flottante identique.

```
>>> float(100.123)
100.123
```

Appeler la fonction « `float` » sans argument retourne « `0.0` ».

int

La fonction « `int` » convertit une chaîne ou un nombre en un nombre entier. Par exemple :

```
>>> int(123.456)
123
>>> int('123')
123
```

Cette fonction ne réagit pas exactement comme la fonction « `float` » en ce qui concerne les chaînes. En effet si vous essayer de convertir une chaîne qui représente un nombre à virgule flottante avec la fonction « `int` » vous aurez un message d'erreur :

```
>>> int('123.456')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.456'
```

Par contre, si vous appelez la fonction « `int` » sans argument alors « `0` » est retourné.

len

La fonction « `len` » retourne la longueur d'un objet. Dans le cas d'une chaîne, elle retourne le nombre de caractères de la chaîne :

```
>>> len('Ceci est une chaîne.')
20
```

Pour une liste ou un n-uplet, elle retourne le nombre d'éléments :

```
>>> maliste = [ 'a', 'b', 'c', 'd' ]
>>> print(len(maliste))
4
>>> monnuplet = (1,2,3,4,5,6)
>>> print(len(monnuplet))
6
```

Pour un dictionnaire, elle retourne le nombre d'entrées dans celui-ci :

```
>>> mondictionnaire = { 'a' : 100, 'b' : 200, 'c' : 300 }
>>> print(len(mondictionnaire))
3
```

Vous pourriez trouver la fonction « `len` » utile dans des boucles. si vous voulez parcourir une liste et connaître la position de chaque élément. Par exemple, on peut parcourir une liste avec le code suivant :

```
>>> maliste = [ 'a', 'b', 'c', 'd' ]
>>> for élément in maliste:
...     print(élément)
```

Cela affichera tous les éléments de la liste (a, b, c, d)— mais que faire si vous voulez afficher l'index de chaque élément de la liste ? Dans ce cas vous pourriez avoir l'idée saugrenue et stupide de vouloir d'abord trouver la valeur de la liste, puis d'incrémenter un compteur pour afficher tous les éléments [2]. Utiliser des variables auxiliaires n'est pas très « pythonesque » et carrément pas élégant.

Très moche !

```
>>> maliste = [ 'a', 'b', 'c', 'd' ]
>>> longueur = len(maliste)
>>> for x in range(0, longueur):
...     print("L'élément à l'index %s est %s." % (x, maliste[x]))
...
L'élément à l'index 0 est a.
L'élément à l'index 1 est b.
L'élément à l'index 2 est c.
L'élément à l'index 3 est d.
```

On stocke la longueur de la liste dans une variable « `longueur` » puis nous l'utilisons avec la fonction « `range` » pour créer notre liste.

```
>>> maliste = [ 'a', 'b', 'c', 'd' ]
>>> for (x,élément) in enumerate(maliste):
...     print("L'élément à l'index %s est %s." % (x, élément))
```

2. Ceci n'est pas forcément stupide dans tous les langages de programmation...

```
...
L'élément à l'index 0 est a.
L'élément à l'index 1 est b.
L'élément à l'index 2 est c.
L'élément à l'index 3 est d.
```

Cette formulation est plus simple et plus compréhensible pour la plupart des êtres humains.

La fonction « `len` » peut être utilisée pour des usages plus utiles comme mesurer la longueur d'un mot de passe ou compter le nombre de jouets qu'a un enfant.

```
>>> motdepasse="simple"
>>> if len(motdepasse) < 8 :
...     print("Le mot de passe est trop court.")
...
Le mot de passe est trop court.
>>> listejouets=['voiture', 'poupée', 'puzzle']
>>> print("Il y a %s jouets." % len(listejouets))
Il y a 3 jouets.
```

max

La fonction « `max` » retourne le plus grand objet d'une liste, d'un n-uplet ou même d'une chaîne. Par exemple :

```
>>> maliste = [ 5, 4, 10, 30, 22 ]
>>> print(max(maliste))
30
```

La fonction « `max` » fonctionne sur une chaîne constituée d'éléments séparés par des virgules ou des espaces :

```
>>> s = 'a,b,d,h,g'
>>> print(max(s))
h
```

De plus, vous n'avez pas obligation d'utiliser des listes, des n-uplets ou des chaînes. Vous pouvez aussi utiliser la fonction « `max` » directement avec un nombre quelconque d'arguments :

```
>>> print(max(10, 300, 450, 50, 90))
450
```

min

La fonction « `mini` » fonctionne de la même manière que la fonction « `max` » mis à part qu'elle retourne le plus petit élémenet de la liste, du n-uplet ou de la chaîne :

```
>>> mylist = [ 5, 4, 10, 30, 22 ]
>>> print(min(mylist))
4
```

print

La fonction « `print` » affiche des textes composés à partir de chaînes, de nombres ou d'objets, dans la console ou dans le shell.

```
print('Bonjour !')
print(10)
print(x)
```

range

La fonction « `range` » est principalement utilisée dans des boucles quand vous voulez répéter les mêmes actions un certain nombre de fois. Nous avons d'abord vu la fonction « `range` » dans le chapitre 5. Ainsi nous avons déjà vu comment l'utiliser avec un ou deux arguments. En fait nous pouvons aussi l'utiliser avec trois arguments.

```
>>> for x in range(5):
...     print(x)
...
0
1
2
3
4
```

Ce que vous n'avez pas réalisé, c'est que la fonction « `range` » retourne en fait un objet spécial (appelé un itérateur) qui est utilisé par la boucle. Vous pouvez convertir l'itérateur en une liste (bizarrement en utilisant la fonction « `list` »), ainsi si vous affichez la liste retournée vous verrez les nombres qu'elle contient.

```
>>> print(list(range(5)))
[0, 1, 2, 3, 4]
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

Vous obtenez une liste de nombre que vous pouvez utiliser ailleurs dans votre programme grâce à une variable.

```
>>> ma_liste_de_nombres = list(range(0, 30))
>>> print(ma_liste_de_nombres)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

La fonction « `range` » peut aussi prendre un troisième argument, appelé « pas » ; les deux premiers arguments étant le début et la fin. Si le pas n'est pas fourni à la fonction, c'est à dire si vous ne fournissez que le début et la fin, il est considéré comme valant un. Mais qu'arrive-t-il si nous utilisons deux comme pas ? Vous pouvez voir le résultat dans l'exemple qui suit :

```
>>> ma_liste_de_nombres = list(range(0, 30, 2))
>>> print(ma_liste_de_nombres)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

La différence (l'écart) entre deux nombres successifs de la liste vaut deux. Le pas détermine la valeur de cet écart. Nous pouvons utiliser un pas plus grand :

```
>>> mylist = list(range(0, 500, 50))
>>> print(mylist)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
```

Ce qui crée une liste de zéro à cinq cent (sans inclure cinq cent, bien sûr) en incrémentant de cinquante en cinquante.

sum

La fonction « `sum` » ajoute les éléments d'une liste et retourne le résultat.

```
>>> maliste = list(range(0, 500, 50))
>>> print(maliste)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]

>>> print(sum(mylist))
2250
```

Tous les mots clef de Python 3

Les mots clef de Python (en fait, pour la plupart des langages des programmation) sont des mots importants qui sont utilisés par le langage lui-même. Si vous essayez d'utiliser ces mots spéciaux comme des variables ou de les utiliser incorrectement, vous aurez des messages d'erreur bizarres (des fois drôles, des fois incompréhensibles) de la part de Python.

Vous trouverez dans la Tableau C.1 l'ensemble des mots clef de Python 3.

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

Table C.1 – Les mots clef de Python 3

Voici une description de ces mots clef classés dans l'ordre alphabétique de Python, c'est à dire d'abord les majuscules dans l'ordre alphabétique puis les minuscules dans l'ordre alphabétique [1].

False

Le mot clef « `False` » [2] est utilisé pour représenter ce qui est faux. Il peut s'agir du résultat d'un test ou d'une assignation directe.

Une variable qui contient « `False` » (faux) ou « `True` » est appelée un booléen [3].

Le code suivant affichera « `Les deux booléens sont faux.` » car il est faux de dire que un est inférieur à zéro donc « `booléen1` » contient « `False` » et « `booléen2` » contient « `False` » car nous lui attribuons la valeur « `False` ».

1. Ce classement est issu de la table ASCII (voir le Tableau 6.1) ; c'est le même classement qui est utilisé pour la fonction « `cmp` ».
2. Attention « `False` » s'écrit avec une majuscule.
3. Booléen se prononce « bou-lé-un ».

```
booléen1=1<0
booléen2=False
if booléen1 or booléen2 :
    print("Un des deux booléens est vrai.")
else :
        print("Les deux booléens sont faux.")
```

None

Le mot clef « None » [4] (aucun) correspond au contenu d'une variable qui ne pointe pas encore vers quelque chose. On peut aussi faire pointer une variable vers « None » pour libérer de l'espace mémoire en particulier si l'objet n'est plus utilisé dans la suite du programme. Attention, l'utilisation d'un test sur une variable inconnu ne créé pas la variable mais crée une erreur, on utilise alors « None » pour ne pas avoir d'erreur.

```
x=None
for i in range(10):
    if x==None :
        x=0
    else :
        x=3*x+1
```

True

Le mot clef « True » [5] est utilisé pour représenter ce qui est vrai. Il peut s'agir du résultat d'un test ou d'une assignation directe.

Une variable qui contient « True » (vrai) ou « False » est appelée un booléen [6].

Le code suivant affichera « Un des deux booléens est vrai. » car il est vrai de dire que un est supérieur à zéro donc « booléen1 » contient « True » et « booléen2 » contient « False » car nous lui attribuons la valeur « False ».

```
booléen1=1>0
booléen2=False
if booléen1 or booléen2 :
    print("Un des deux booléens est vrai.")
else :
        print("Les deux booléens sont faux.")
```

4. Attention « None » s'écrit avec une majuscule.
5. Attention « True » s'écrit avec une majuscule.
6. Booléen se prononce « bou-lé-un ».

and

Le mot clef « `and` » est utiliser pour lier deux expressions entre elles pour un test — par exemple un « test si » ou une boucle « tant que » — pour dire que les deux expressions doivent être vraies pour que le résultat soit vrai [7].

```
if âge > 10 and âge < 20:
    print("Vous êtes un adolescent.")
```

La phrase ne sera affichée que si « `âge` » est strictement plus grand que dix *et* strictement plus petit de vingt.

as

Le mot clef « `as` » est utilisé pour donner un autre nom à un module importé. Par exemple si vous avez un module dont le nom ressemble à « `jesuisunmodulepythonpastrèsutile` », il sera particulièrement ennuyeux d'avoir à taper le nom de ce module à chaque fois que vous voudrez l'utiliser :

```
>>> import jesuisunmodulepythonpastrèsutile
>>>
>>> jesuisunmodulepythonpastrèsutile.action()
J'ai fait quelque chose.
>>>  jesuisunmodulepythonpastrèsutile.autre_action()
J'ai fait quelque chose d'autre !
```

À la place vous pouvez donner un nouveau nom quand vous l'importer puis utiliser ce nouveau nom (une sorte de surnom) :

```
>>> import jesuisunmodulepythonpastrèsutile as pasutile
>>>
>>> pasutile.action()
J'ai fait quelque chose.
>>>  pasutile.autre_action()
J'ai fait quelque chose d'autre !
```

Vous n'utiliserez néanmoins pas souvent le mot clef « `as` ».

assert

Le mot clef « `assert` » est un mot clef avancé qui sert à contrôler le comportement d'un programmes lors de tests. Si le code n'est pas conforme à ce qui était attendu le programme s'arrêtera en indiquant précisément où était l'erreur. Ce mot clef « » est généralement utilisé pour des programmes de taille importante.

7. Les valeurs non nulles sont considérées comme vraies, une série d'expression « `and` » retourne la dernière valeur vraie de gauche vers la droite ou la première valeur fausse de gauche vers la droite.

break

Le mot clef « break » est utilisé pour arrêter une boucle et sortir de celle-ci. Vous pouvez utiliser « break » comme suit :

```
>>> âge = 10
>>> for x in range(1, 100):
...     print('décompte %s' % x)
...     if x == age:
...         print('fin du décompte')
...         break
...
décompte 1
décompte 2
décompte 3
décompte 4
décompte 5
décompte 6
décompte 7
décompte 8
décompte 9
décompte 10
fin du décompte
```

Jetez un coup d'œil au chapitre 5 pour avoir plus d'information à propos des boucles.

class

Le mot clef « class » est utilisé pour définir un type d'objet. Un objet contient des attributs (des sortes de sous-variables) et des méthodes (des sortes de sous fonction). Vous avez utilisé tout au long de ce livre des méthodes et des attributs.

L'existence d'objet est une fonctionnalité fournie par de nombreux langages de programmation qui est vraiment très utile pour des programmes compliqués. Néanmoins la définition de nouveaux objets et leur utilisation est trop longue à expliquer dans le cadre de ce livre [8].

def

Le mot clef « def » est utilisé pour définir des fonctions comme vous avez pu le voir dans le chapitre 6.

del

Le mot clef « del » est utilisé pour éliminer quelque chose. Par exemple si vous avez une liste de choses à faire dans votre carnet de textes mais que vous avez fini cette tâche ; vous

8. Ce genre de d'idée est expliqué dans des livres écrits plus petit pour qu'on puisse mettre plus d'idées dedans.

pourriez barrer cette tâche de la liste et en ajouter une nouvelle.

Donner à manger au chat
Changer la litière
Prendre une douche
~~Ranger ma chambre~~
Faire une ballade à vélo

Si nous avons la même liste en Python cela donne :

```
>>> àfaire=['Donner à manger au chat'
... 'Changer la litière'
... 'Prendre une douche'
... 'Ranger ma chambre'
```

... 'Faire une ballade à vélo'

Nous pouvons enlever « `'Ranger ma chambre` » (après l'avoir fait ☺) et ajouter un nouvel élément en utilisant la méthode « append ».

```
>>> del àfaire[3]
>>> àfaire.append('Faire une ballade à vélo')
```

Nous pouvons alors afficher la nouvelle liste :

```
>>> for i in àfaire :
...     print(i)
...
Donner à manger au chat
Changer la litière
Prendre une douche
Faire une ballade à vélo
```

Vous pouvez voir le chapitre 2 pour plus d'information à propos des listes.

elif

Le mot clef « `elif` » est utilisé avec le mot clef « `if` » pour les tests si (voir ci-dessous).

else

Le mot clef « `else` » est aussi utilisé avec « `if` » pour les tests si (voir ci-dessous)...

except

Le mot clef « `expect` » est utilisé pour géré des problèmes dans le code. C'est une expression complexe utilisée dans des programmes complexe et qui ne peut pas être expliqué suffisamment simplement dans ce livre.

finally

Le mot clef « `finally` », lui aussi utilisé quand une erreur arrive, sert à exécuter des actions pour finir d'arranger les choses pour que le reste du code puisse s'exécuter normalement.

for

Le mot clef « `for` » est utilisé pour créer une boucle. Par exemple :

```
for x in range(0,5):
    print('x vaut %s' % x)
```

La boucle ci-dessus exécute le bloc de code (la ligne avec « `print` ») cinq fois ce qui crée la sortie :

```
x vaut 0
x vaut 1
x vaut 2
x vaut 3
x vaut 4
```

from

Quand vous importez un module, vous pouvez importer tout le module ou seulement la partie dont vous avez besoin en utilisant le mot clef « `from` ». Par exemple, le module « `turtle` » z une fonction « `Pen` » qui est utilisé pour créer un objet « `Pen` » pour dessiner. Vous pouvez importer le module « `turtle` » en entier puis utiliser la fonction « `Pen` » :

```
>>> import turtle
>>> t = turtle.Pen()
```

Ou, vous pouvez juste importer la fonction « `Pen` » et l'utiliser directement sans avoir besoin de faire référence au module « `turtle` ».

```
>>> from turtle import Pen
>>> t = Pen()
```

Bien sûr, cela veut dire que vous ne pouvez pas utiliser les parties non importées d'un module. Par exemple, le module « time » a des fonctions appelées « localtime » et « gmtime ». Si vous importez uniquement « localtime » puis essayer d'utiliser « gmtime », vous obtiendrez une erreur.

Le code suivant fonctionnera bien :

```
>>> from time import localtime
>>> print(localtime())
(2007, 1, 30, 20, 53, 42, 1, 30, 0)
```

Mais l'utilisation de « gmtime » ne sera pas possible.

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

Quand Python vous dit « 'gmtime' is not defined », Python vous dit qu'il ne connait pas encore la fonction « gmtime » ? Si il y a un grand nombre de fonctions d'un module que vous voulez utiliser et que vous ne voulez pas les utiliser avec le nom du module (c'est à dire « time.localtime » ou « module.fonction ») vous pouvez tout importer d'un module en utilisant le caractère étoile « * ».

```
>>> from time import *
>>> print(localtime())
(2007, 1, 30, 20, 57, 7, 1, 30, 0)
>>> print(gmtime())
(2007, 1, 30, 13, 57, 9, 1, 30, 0)
```

Dans ce cas nous avons importé toutes les fonctions du module « time » et nous pouvons faire référence aux fonctions du module en utilisant leur nom.

global

Dans le chapitre 6, nous avions parlé de la portée d'une variable. La portée d'une variable est « l'espace » où est utilisable la variable. Si une variable est définie en dehors d'une fonction, elle est habituellement utilisable à l'intérieur de celle-ci. Si elle est définie à l'intérieur d'une fonction, elle n'est habituellement pas utilisable en dehors de cette dernière.

L'utilisation du mot clef « global » est exception qui confirme la règle. La définition d'une variable en utilisant « global » permet d'utiliser cette variable dans l'ensemble du programme.

```
>>> def test():
...     global a
...     a = 1
...     b = 2
```

Que pensez vous qu'il arrivera si vous appelez « print(a) » puis « print(b) » après avoir exécuter la fonction « test » ? Le premier va fonctionner, le deuxième créera un message d'erreur qui sera affiché.

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

La variable « a » est maintenant globale (utilisable dans tout le programme) mais la variable « b » est toujours visible uniquement à l'intérieur de la fonction. Notez que vous devez utiliser « global » avant de stocker quoi que ce soit dans votre variable globale.

if

Le mot clef « if » est utilisé pour prendre des décisions ; il est parfois utilisé avec des mots clef comme « else » et « elif » (else if). Un test si est une manière de dire : « si quelque chose est vrai, alors réalise cette action ». Par exemple :

```
if prix_du_jouet > 1000:
    print('Ce jouet est hors de prix.')
elif prix_du_jouet > 100:
    print('Ce jouet est cher.')
else:
    print('Je voudrais ce jouet.')
```

Ce code signifie que si le prix du jouet est supérieur à 1000€, il est hors de prix, sinon que si son prix est supérieur à 100€, il est cher et que sinon on peut dire à ses parents que l'on voudrait ce jouet [9]. Le chapitre 4 contient plus d'information à propos des tests si.

import

Le mot clef « import » est utilisé pour dire à Python de charger un module pour qu'il puisse être utilisé, comme dans l'exemple ci-dessous.

```
>>> import sys
```

Ce code dit à Python de rendre utilisable le module « sys ».

9. Malheureusement pour vous il est probable que vos parents ont une échelle de prix plus basse...

in

Le mot clef « in » est utilisé dans des expressions pour trouver un objet dans une collection d'objets. Par exemple, trouver si un nombre peut être trouvé dans une liste (une collection) de nombre.

```
>>> if 1 in [1,2,3,4]:
...     print('Le nombre un est dans la liste.')
Le nombre un est dans la liste.
```

Ou voir si on peut trouver une laitue dans la liste de course.

```
>>> courses = [ 'œufs', 'lait', 'fromage' ]
>>> if 'laitue' in courses:
...     print('Il y a de la laitue dans la liste de courses.')
... else:
...     print('Il n'y a pas de la laitue dans la liste de courses.')
...
Il n'y a pas de la laitue dans la liste de courses.
```

is

Le mot clef **is** est utiliser pour contrôler si deux variables pointent vers deux objets strictement identiques. Par exemple « 2 » (entier) et « 2.0 » (flottant) seront vu comme différents pour « is » alors que « == » les considéra comme égaux (mais différents). Pour résumer quand vous comparez deux choses avec « == » le résultat peut être vrai alors qu'il pourra être faux avec « is ».

Il s'agit là d'un mécanisme complexe qui tend à rendre confus les humaines. Donc pour le moment continuez d'utiliser « == ».

lambda

Le mot clef « lambda » est utilisé pour créer de petites fonctions temporaires. Néanmoins l'usage qui en est s'il permet de faire de beaux programmes est si compliqué que le simple fait d'écrire une explication ferait s'enflammer ce livre.

Donc le mieux est de ne pas en parler.

not

Si quelque chose est vrai utilisé « not » le rendra faux et inversement. Par exemple si nous créons une variable « x » à vrai et que nous utilisons « not » est résultat est faux.

```
>>> x = True
\end{verbatim}
```

```
>>> print(not x)
False
```

Cela peut ne pas sembler vraiment utile jusqu'à ce que vous commencerez à utiliser « not » dans des tests si. Par exemple si vous avez dix ans est que vous considérez que le meilleur âge est dix ans, vous pouvez vouloir dire :

> Le meilleur nombre d'année n'est pas 1. Le meilleur nombre d'année n'est pas 2. Le meilleur nombre d'année n'est pas 3. Le meilleur nombre d'année n'est pas 4. Le meilleur nombre d'année n'est pas 5. … Le meilleur nombre d'année n'est pas 9. Le meilleur nombre d'année est 10. Le meilleur nombre d'année n'est pas 11. … Le meilleur nombre d'année n'est pas 19.

```
if age < 10 or age > 10:
    print("Le meilleur nombre d'année n'est pas %s."  % age)
else :
    print("Le meilleur nombre d'année est %s."  % age)
```

Mais on peut aussi écrire :

```
if not age == 10:
    print("Le meilleur nombre d'année n'est pas %s."  % age)
else :
    print("Le meilleur nombre d'année est %s."  % age)
```

L'expression « if not age == 10 : » veut dire « si " age " ne vaut pas dix alors ».

or

Le mot clef « or » est utilisé pour joindre deux expression dans un test pour dire qu'une des deux doit être vraie [10].

```
>>> if "langue" in liste or "bananes" in liste :
...     print('Miam')
... elif "tofu" in liste or "mélasse" in liste :
...     print('Beurk')
```

Dans ce cas, nous regardons si des plats que nous aimons sont dans la liste de plats proposés : si il y a de la langue ou des bananes, on affiche « Miam » sinon si il y a tofu ou de la mélasse ou affiche « Beurk ».

10. Les valeurs non nulles sont considérées comme vraies, une série d'expression « and » retourne la première valeur vraie de gauche vers la droite ou la dernière valeur fausse de gauche vers la droite.

pass

Le mot clef « pass » indique à Python de rien faire ! Cela peut sembler étrange mais cela peut être très utile.

Quelques fois vous écrivez un programme et vous voulez voir comment il fonctionne avant même d'avoir fini de l'écrire complètement. Le problème c'est que vous ne pouvez pas avoir de « : », pour un test si ou une boucle par exemple, sans bloc de code à exécuter.

Par exemple, vous pouvez écrire :

```
>>> if age > 10:
...     print('plus de dix ans')
...
```

Par contre, si vous écrivez :

```
>>> if age > 10:
...
```

Vous allez avoir un message d'erreur qui devrait ressembler à quelque chose comme cela : You'll get an error message in the console that looks something like this :

```
  File "<stdin>", line 2
    ^
IndentationError: expected an indented block
```

Il s'agit d'un message d'erreur qui indique que Python attendait l'indentation qui correspond au bloc de code.

Le mot clef « pass » peut être utilisé dans ce cas, ainsi vous pouvez écrire le bloc exigé par Python sans que celui-ci ne fasse rien. Par exemple, vous pouvez écrire une boucle avec un test si à l'intérieur. Peut-être n'avez vous pas décidé ce que vous voulez mettre si la condition est vraie. Peut-être vous mettrez un « print » , peut-être vous mettrez un « break ». Dans ce cas, vous pouvez utiliser « pass » et votre code fonctionnera (même s'il ne fera pas ce que vous ne savez pas encore vouloir pour le moment).

Le code suivant

```
>>> for x in range(1,7):
...     print('x vaut %s' % x)
...     if x == 5:
...         pass
```

affichera ce qui suit :

```
x vaut 1
x vaut 2
x vaut 3
x vaut 4
x vaut 5
x vaut 6
```

Plus tard vous remplacerez dans le code l'instruction « pass » pas quelque chose qui fera quelque chose.

raise

Encore un mot clef pour programmeur avancé, « raise » est utilisé pour créer un erreur—ce qui peut sembler une étrange chose à faire, mais cela est, dans certains programmes, vraiment très utile.

return

Le mot clef « return » est utilisé pour indiqué la valeur renvoyée par une fonction et faire sortir immédiatement de la fonction. Par exemple, vous pourriez avoir envie de créer une fonction qui retourne la quantité d'argent que vous avez économisé.

```
>>> def monargent():
...     return somme_d_argent
```

Quand vous appelez cette fonction, la valeur retournée peut-être assignée à une autre variable :

```
>>> money = mymoney()
```

ou affichée

```
>>> print(mymoney())
```

try

Le mot clef « try » est utilisé au début d'un bloc de code qui finit par le mot clef « except » ou [11] « finally ». Tous ensemble ces mots clefs sont utilisés pour que le programme gère bien les erreurs—par exemple pour être sûr que le programme affiche un message utile plutôt qu'un horrible message d'erreur avant de s'arrêter.

while

Le mot clef « while » est utilisé pour faire des boucles, un peu comme « for ». La boucle tant que crée par « while » continue de s'exécuter tant qu'une expression est vraie. Vous devez être vraiment attentifs quand vous utilisez une boucle tant que car si l'expression est toujours vrai, la boucle continuera sans fin (on parle de boucle infinie).

LE code suivant est incorrect.

11. Comme dans la plupart de langues le « ou » français est inclusif, c'est à dire qu'il faut comprendre : ou « except » ou « finally » ou encore « except » et en plus « finally ».

"ne pas taper"

```
>>> x = 1
>>> while x == 1:
...     print('Bonjour !')
```

Si vous exécutiez le code ci-dessus, il s'exécuterait sans fin. À moins, bien sût, que vous fermiez la console Python ou que vous l'arrêtiez avec « `Ctrl+C` » (en appuyant en même temps sur la touche « `Ctrl` » et la touche « `C` »).

Par contre le code suivant :

```
>>> x = 1
>>> while x < 10:
...     print('Bonjour !')
...     x = x + 1
```

affichera « `Bonjour !` » neuf fois en ajoutant un à la variable « `x` » tant que ce dernier vaut moins de dix. C'est évidemment un peu comme une boucle avec un itérateur mais cela peut avoir un intérêt dans certaines situations.

with

Le mot clef « `with` » est un mot clef très avancé.

yield

Le mot clef « `yield` » est un autre mot clef très avancé.

Quelques modules de Python

Python a un grand nombre de modules disponibles pour faire toutes sortes de choses. Vous pouvez vous renseigner sur ces derniers en lisant la documentation de Python à l'adresse suivante `http://docs.python.org/modindex.html`. Quelqu'un des modules les plus utiles sont décrits ici. Un avertissement si vous décidez de regarder la documentation de Python : la liste des modules est vraiment longue et certains de ces modules sont très compliqués.

D.1 Le module « random »

D.1.1 Introduction à random

Si vous avez déjà joué à faire deviner un chiffre entre un et cent ou si vous avez déjà lancé un dé, vous savez déjà ce que fait le module « `random` ». Le module « `random` » contient un certain nombre de fonctions qui sont utiles pour générer des données aléatoires, c'est à dire tirées au hasard.

De toutes les fonctions de « `random` », les plus utiles sont sans contestation possible les fonctions `randint`, `choice` et `shuffle`.

D.1.2 randint

La première fonction, `randint`, prend un nombre entier au hasard entre le premier argument et le deuxième argument, par exemple entre un et six ou un et cent. Par exemple :

```
>>> import random
>>> print(random.randint(1, 6))
5
>>> print(random.randint(1, 100))
861
>>> print(random.randint(1000, 5000))
3795
```

Nous pouvons utiliser cette fonction pour créer un jeu de devinette, simple (et ennuyant) en utilisant une boucle tant que :

```
import random
import sys
nombre = random.randint(1, 100)
print('''Le but est de deviner un nombre 1 and 100.
Vous pouvez quitter à tout moment avec "q" puis entrée.''')
while True:
    print('Proposez un nombre 1 and 100 :')
    prop = sys.stdin.readline()
    if prop=="q" :
        print('Vous avez perdu.')
        break
    i = int(prop)
    if i == nombre:
        print('Vous avez deviné juste!')
        break
    elif i < nombre:
        print('Essayez plus haut.')
    elif i > nombre:
        print('Essayer plus bas.')
```

D.1.3 choice

Utilisez « `choice` », si vous avez une liste et que vous voulez tirer un élément au hasard. Par exemple :

```
>>> import random
>>> liste1 = [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]
>>> print(random.choice(liste1))
c
>>> liste2 = [ 'glace', 'pancakes', 'pavlova' ]
>>> print(random.choice(liste2))
pancakes
```

Et finalement, utilisez « `shuffle` » si vous voulez mélanger aléatoirement une liste (comme pour mélanger des cartes) :

```
>>> import random
>>> liste1 = [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]
>>> liste2 = [ 'glace', 'pancakes', 'pavlova' ]
>>> random.shuffle(liste1)
>>> print(liste1)
['h', 'e', 'a', 'b', 'c', 'g', 'f', 'd']
```

```
>>> random.shuffle(liste2)
>>> print(list2)
>>> ['pancakes', 'glace', 'pavlova']
```

D.2 Le module « sys »

D.2.1 Introduction à sys

Le module « sys » contient des fonctions « système » utiles. Le terme système est juste un mot étrange pour désigner tout ce qui fait la liaison entre le matériel (l'ordinateur que l'on peut toucher) et les logiciels (dont le votre en particulier). Certaines des fonctions les plus utiles en Python sont contenues dans le module « sys ». Entre autres les foncions `exit`, `stdin`, `stdout` et `version`.

D.2.2 exit

La fonction `exit` est une manière d'arrêter Python ainsi que la console le cas échéant. Par exemple si vous tapez :

```
>>> import sys
>>> sys.exit()
```

La console Python va s'arrêter. Selon que vous utilisez Windows, Mac ou Linux, un certain nombre de choses peuvent arriver — mais, à la fin, le résultat sera le même : La console Python s'arrêtera.

D.2.3 stdin

La fonction « `stdin` » a été utilisée ailleurs dans ce livre (voir chapitre 6) pour demander à quelqu'un utilisant un programme d'entrée des informations. Par exemple :

```
>>> import sys
>>> mavaleur = sys.stdin.readline()
Ceci est ma valeur.
>>> print(mavaleur)
Ceci est ma valeur.
```

D.2.4 stdout

La fonction « `stdout` » est l'opposé de « `stdin` » — elle est utilisée pour écrire des messages. D'une certaine manière elle ressemble à « `print` » mais elle fonctionne plus comme « `file` ». Quelques fois il est plus pratique d'utiliser « `stdout` » plutôt que « `print` ».

```
>>> import sys
>>> l = sys.stdout.write('Ceci est un test.')
Ceci est un test.>>>
```

Remarquez-vous ou l'invite « >>> » réapparait ? Ce n'est pas une erreur, elle réapparait à la fin du message. C'est parce que, contrairement à « print », « stdout.write » ne rajoute pas automatiquement un retour à la ligne. Pour faire la même chose avec « write » nous pouvons faire comme cela :

```
>>> import sys
>>> l = sys.stdout.write('Ceci est un test.\n')
Ceci est un test.
>>>
```

ou encore :

```
>>> import sys
>>> l = sys.stdout.write('''Ceci est un test.
''')
Ceci est un test.
>>>
```

La fonction « stdout.write » retourne le nombre de caractères écrits —essayez de faire « print(l) » ou « print(sys.stdout.write('Ceci est un test.\n')) », pour voir le résultat.

Le « \n » est le caractère *d'échappement* pour une nouvelle ligne, c'est à dire ce que vous avez quand vous appuyez sur la touche entrée. Un caractère d'échappement est une suite de quelques caractères que l'on utilise dans les chaîne quand on ne peut pas taper directement de caractère [1]. Par exemple si vous voulez créer une chaîne avec un saut de ligne au milieu, sans utiliser de triple apostrophe, vous aurez une erreur :

ne fonctionne pas

```
>>> s = 'test test
  File "<stdin>", line 1
    s = 'test test
                  ^
SyntaxError: EOL while scanning single-quoted string
```

À la place vous pouvez utiliser le caractère d'échappement de du retour à la ligne :

```
>>> s = 'test test\ntest'
```

Pour finir avec « sys », l'attribut « version » est utilisée pour afficher la version de Python que vous utilisez :

1. L'usage des triples apostrophes en Python réduit notablement l'usage des caractères d'échappement par rapport à d'autres langages.

```
>>> import sys
>>> print(sys.version)
3.0.1+ (r301:69556, Apr 15 2009, 15:59:22)
[GCC 4.3.3]
```

D.3 Le module « locale »

Le module « locale » est utilisé pour « localiser », c'est à dire rendre utilisable localement par un utilisateur, des fonctions de Python.

Il est généralement utilisé sous la forme :

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL,'')
```

Cet appel permet à Python d'utiliser la langue locale du système. Il est à noter que toutes les langues du monde ne sont pas prévues et qu'il peut être nécessaire de réaliser des traductions.

D.4 Le module « time »

D.4.1 Non je ne radote pas : time.time

Le module « time » contient des fonctions pour gérer le temps et afficher des résultats compréhensibles par l'utilisateur. Le mot « *time* » signifie temps en anglais. Néanmoins, si vous essayer la fonction qui semble la plus simple « time) », le résultat pourrait bien être différent de ce que vous imaginiez :

```
>>> import time
>>> print(time.time())
1250020582.13
```

Le nombre retourné par la fonction « time » est le nombre de secondes écoulées depuis le 1er janvier 1970 à 00 :00 :00 (zéro heure) en temps universel coordonnée (c'est à dire une heure de moins qu'en France métropolitaine en hiver et deux heures en été). Vous pouvez penser que cela n'est pas très utile, néanmoins cela l'est, des fois. Par exemple, si vous créez un programme et que vous voulez savoir à quelle vitesse s'exécute-t'il, vous pouvez enregistrer l'heure au début puis faire la différence avec l'heure à la fin pour connaitre le temps passé.

Par exemple, combien cela prendra-t'il de temps pour afficher tous les nombre de 1 à 100 000 ? Nous pouvons facilement créer une fonction pour les afficher :

```
>>> def compter(max):
...     for x in range(1, max+1):
...         print(x)
```

Puis, nous pouvons appeler la fonction :

```
>>> compter(100000)
```

Mais si nous voulons savoir combien de temps cela prend, nous pouvons modifier la fonction et utiliser le module « `time` » :

```
>>> import time
>>> def compter(max):
...     t1 = time.time()
...     for x in range(0, max):
...         print(x)
...     t2 = time.time()
...     print("Compter jusqu'à %s a pris  %s secondes" % (max,t2-t1))
```

Si nous utilisons à nouveau notre fonction, nous avons alors :

```
>>> compter(100000)
1
2
3
.
.
.
99997
99998
99999
100000
Compter jusqu'à 100000 a pris  18.7396810055 secondes
```

Comment cela fonctionne-t'il ? La première fois, nous appelons la fonction « `time` », puis nous assignons le résultat à la variable « `t1` ». Nous créons alors un itérateur pour afficher tous les nombres voulus. Puis nous appelons à nouveau la fonction « `time` » et nous mettons le résultat dans la variable « `t2` ». Comme le parcours de la boucle a pris du temps la valeur de « `t2` » est plus grande (plus tard) que « `t1` », le nombre de seconde depuis le 1er janvier 1970 a augmenté). Donc si nous soustrayons « `t1` » à « `t2` » nous avons le nombre de secondes qu'il a fallut pour afficher les nombres.

D'autres fonctions sont disponibles dans le module « `time` » dont : `asctime`, `ctime`, `localtime`, `sleep`, `strftime` et `strptime`.

D.4.2 asctime

La fonction « `asctime` » prend une date sous la forme d'un n-uplet (rappelez-vous un n-uplet est une liste de valeurs qui ne peut pas être changé et la convertit en une forme lisible (pour les anglophones). Vous pouvez aussi l'appeler sans argument, elle affichera alors la date et l'heure actuelle :

```
>>> import time
>>> print(time.asctime())
Sun May 27 20:11:12 2007
```

Pour l'appeler avec un argument nous devons d'abord créer un n-uplet avec les valeurs correctes associées à une date (c'est à dire qui correspondent à quelque chose). Commençons par assigner un n-uplet à la variable « `t` » :

```
>>> t = (2007, 5, 27, 10, 30, 48, 6, 0, 0)
```

Les valeurs dans le n-uplet sont dans l'ordre : l'année, le mois, le jour, les heures, les minutes, les secondes, le jour (0 pour lundi, 1 pour mardi et ainsi de suite jusqu'à 6 pour dimanche), le jour de l'année (obligatoire mais non traité par « `asctime` ») et enfin l'existence d'un éventuel changement d'heure pour l'économie d'énergie (1 si économie d'énergie, 0 sinon).

```
>>> import time
>>> t = (2007, 5, 27, 10, 30, 48, 6, 0, 0)
>>> print(time.asctime(t))
Sun May 27 10:30:48 2007
```

Mais faites attention à mettre des valeurs valides dans le n-uplet. Vous pouvez obtenir un résultat qui n'est pas possible si vous rentrez n'importe quoi :

```
>>> import time
>>> t = (2007, 5, 27, 10, 30, 48, 0, 0, 0)
>>> print(time.asctime(t))
Mon May 27 10:30:48 2007
```

Ici, nous avons fixé la valeur du jour de la semaine à zéro (pour lundi, *monday*) alors que le 27 mai 2007 était un dimanche (6).

D.4.3 ctime

La fonction « `ctime` » est utilisée pour convertir un nombre de secondes en un texte lisible. Par exemple nous pouvons l'utiliser avec la fonction « `time` » vue au début de cette section.

```
>>> import time
>>> t = time.time()
>>> print(t)
1180264952.57
>>> print(time.ctime(t))
Sun May 27 23:22:32 2007
```

D.4.4 localtime

La fonction « `localtime` » retourne la date et l'heure actuelles comme un n-uplet dont la séquence est la même que celle que nous venons d'utiliser.

```
>>> import time
>>> print(time.localtime())
(2007, 5, 27, 23, 25, 47, 6, 147, 0)
```

Nous pouvons aussi fournir cette valeur à « `asctime` » :

```
>>> import time
>>> t = time.localtime()
>>> print(time.asctime(t))
Sun May 27 23:27:22 2007
```

D.4.5 sleep

La fonction « `sleep` » est vraiment utilise quand vous devez retarder un programme pour un certain temps. Le mot *sleep* signifie dormir en anglais. Par exemple, si vous voulez afficher un nombre chaque seconde la boucle suivante ne sera pas adaptée :

```
>>> for x in range(61):
...     print(x)
...
1
2
3
4
```

Elle imprimera immédiatement tous les nombre de un à soixante. Néanmoins si vous dites à Python de dormir entre chaque affichage, vous aurez un résultat beaucoup plus proche de ce que vous voulez :

```
>>> for x in range(61):
...     print(x)
...     time.sleep(1)
...
```

Il y aura ainsi entre chaque affichage un délais. Pour être exact le temps d'affichage est non nul donc l'affichage des soixante nombres prendra dans ce cas un peu plus d'une minute (soixante temps d'affichage plus soixante secondes d'attente). Idéalement il faudrait donc ajouter un peu moins qu'une seconde à chaque pas, néanmoins l'auto-adaptation de ce délais sort du cadre de ce livre.

Dire à l'ordinateur de dormir peut ne pas sembler utile alors quand fait c'est souvent important. Pensez à votre réveil, quand vous appuyez sur le bouton, il arrête de bipper pour quelques minutes (au moins jusqu'à ce que quelqu'un vous appelle pour le petit déjeuner). La fonction « `sleep` » pourrait être utilisée dans ce genre de situation.

D.4.6 strftime

La fonction « `strftime` » est utilisée pour réaliser l'affichage de la date et de l'heure à votre convenance. Attention au « `f` » de « `strftime` », il existe une autre fonction « `strptime` » avec un « `p` » qui converti une chaîne en n-uplet. Commençons d'abord par examiner « `strftime` ». Juste avant nous avons vu comment changer un n-uplet en chaîne en utilisant « `asctime` ».

```
>>> t = (2007, 5, 27, 10, 30, 48, 6, 0, 0)
>>> print(time.asctime(t))
Sun May 27 10:30:48 2007
>>> t=time.localtime()
>>> print(time.asctime(t))
Thu Aug 13 08:56:15 2009
```

La fonction « `localtime` » fournit un n-uplet déjà renseigné avec toutes des informations correspondant à l'heure locale (non universelle, qui est affichée sur votre ordinateur) à l'instant où la commande est lancée si aucun argument n'est donné. La commande « `localtime` » peut aussi prendre en argument un nombre de secondes depuis le premier janvier 1970.

Cela fonctionne bien pour les anglophones, mais si vous n'aimez pas la manière dont la chaîne est présenté — comment faire si vous voulez afficher la date et pas l'heure, si vous voulez avoir un résultat en français plutôt qu'en anglais ?

code

```
import time,locale

t=time.localtime()
print(time.strftime('%d %b %Y', t))

locale.setlocale(locale.LC_ALL,'')
print(time.strftime('%d %b %Y', t))
```

résultat

```
13 Aug 2009
13 août 2009
```

Comme vous pouvez voir « `strftime` » prend deux arguments : le premier est le format souhaité de la date (décrivant comment la date ou l'heure vont être affichés) et le second est un n-uplet qui contient une date. Ici nous avons généré le n-uplet avec la commande « `localtime` ». Le format « `%d %b %Y` » est une autre manière de dire : affiche le jour, le mois puis l'année. Remarquez que l'utilisation de « `setlocale` » permet d'afficher ce résultat dans la langue choisi par l'utilisateur pour son système d'exploitation.

Nous pouvons aussi afficher le mois comme un nombre :

```
>>> print(time.strftime('%d/%m/%Y', t))
13/08/2009
```

Ce format est une manière de dire : affiche le jour, puis une barre oblique, puis le mois sous la forme d'un nombre, puis une barre oblique et finalement l'année. Il y a différentes possibilités que vous pouvez utiliser pour formater la date :

%a	le nom raccourci du jour de la semaine (lun., mar., mer., jeu., ven., sam. ou dim.)
%A	le nom jour de la semaine
%b	le nom raccourci du mois
%B	le nom du mois
%c	la date et l'heure complète dans un format similaire à « `asctime` » mais dans la langue locale
%d	le numéro du jour dans le mois
%H	le nombre d'heures sur une base de vingt quatre heures (14 pour deux heures de l'après -midi)
%I	le nombre d'heures sur une base de douze heures (2 pour quatorze heures)
%j	le numéro du jour dans l'année
%m	le numéro du mois
%M	le nombre de minutes
%p	non disponible en français, en anglais le matin sous la forme AM ou l'après-midi sous la forme PM
%S	le nombre de secondes
%U	le numéro de la semaine dans l'année
%w	le numéro du jour dans la semaine (dimanche vaut 0, lundi 1...)
%x	la date sous une forme numérique simple : jour/mois/année (03/25/07) si la langue est européenne
%X	l'heure sous une forme numérique simple : heure :minutes :secondes (10 :30 :53)
%y	l'année sur deux chiffres (09 pour 2009)
%Y	l'année sur quatre chiffres (2009)

D.4.7 strptime

La fonction « `strptime` » est presque l'inverse de la fonction « `strftime` » — elle prend une chaîne comme argument et la convertie en un n-uplet qui contient la date et l'heure. Elle prend comme format une chaîne sous la même forme (%d, %H...) :

```
>>> import time,locale
>>> locale.setlocale(locale.LC_ALL,'')
>>> t = time.strptime('13 août 2009', '%d %b %Y')
>>> print(t)
time.struct_time(tm_year=2009, tm_mon=8, tm_mday=13, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=225, tm_isdst=-1)
```

Si la date avait été sous la forme jour/mois/année (par exemple 01/02/2007), nous pourrions utiliser « `t = time.strptime('01/02/2007', '%d/%m/%Y')` » ou encore « `t =`

`time.strptime('01/02/2007', '%x')` ».

Nous pouvons combiner « `strptime` » et « `strftime` » pour convertir une chaîne d'un format vers un autre.

```
>>> import time
>>> def convert_date(datestring, format1, format2):
...     t = time.strptime(datestring, format1)
...     return time.strftime(format2, t)
...
```

Nous pouvons utiliser cette fonction en passant une date sous la forme d'une chaîne, le format de cette dernière et le format souhaité.

```
>>> print(convert_date('03/05/2007', '%m/%d/%Y', '%d %B %Y'))
05 mars 2007
```

Gâteaux

Vous trouverez ici quelques exemples de code qui ne proviennent pas du livre anglais.

E.1 Les textes en Tk

à taper

```
import tkinter

def coucou() :
    text.insert("%d.%d" % (0, 0),"coucou ")

racine = tkinter.Tk()

text= tkinter.Text(racine, height=20, width=50)
text.pack()

bouton_coucou = tkinter.Button(racine, text="Cliquez moi !",
                               command=coucou)
bouton_coucou.pack()

bouton_sortir = tkinter.Button(racine, text="Sortir",
                               command=racine.destroy)
bouton_sortir.pack()

racine.mainloop()
racine.destroy()
```

Index

%, 20

append, 24

caractère d'échappement, 168
chaîne, 19
console, 4

date/time formats, 174
del, 24

fonctions intégrées
- abs, 141
- bool, 141
- cmp, 142
- dir, 143
- enumerate, 144
- eval, 144
- exec, 145
- file, 145
- float, 145
- int, 146
- len, 146
- max, 148
- min, 149
- print, 149
- range, 149
- sum, 150

installation, vi
int, 70

liste, 21

modules, 165
- locale, 169
- random, 165
  - choice, 166
  - randint, 165
- shuffle, 166
- sys, 167
  - exit, 167
  - stdin, 167
  - stdout, 167
  - version, 168
- time, 169
  - asctimes, 170
  - ctime, 171
  - localtime, 172
  - sleep, 172
  - strftime, 173
  - strptime, 174
  - time (fonction), 169

mots clef
- and, 153
- as, 153
- assert, 153
- break, 154
- class, 154
- def, 154
- del, 154
- elif, 155
- else, 155
- except, 156
- False, 151
- finally, 156
- for, 156

from, 156
global, 157
if, 158
import, 158
in, 159
is, 159
lambda, 159
None, 152
not, 159
or, 160
pass, 161
raise, 162
return, 162
True, 152
try, 162
while, 162
with, 163
yield, 163

n-uplet, 21

opérateurs, 13

paramètre substituable, 21
print, 7

variable, 15

Table des matières

Préface v

1 Tous les serpents ne vont pas vous mordre 1
- 1.1 Bonjour, je suis votre livre 1
- 1.2 Qu'est ce que nous allons voir ? 2
- 1.3 Quelques mots à propos des langages 3
- 1.4 L'ordre des serpents constricteurs non-venimeux... 3
- 1.5 Votre premier programme en Python 5
- 1.6 Votre second programme Python... Le même à nouveau ? 7

2 8 multipliés par 3,57 égal... 11
- 2.1 Les calculs en Python 11
- 2.2 L'usage des parenthèses et « l'ordre des opérations » 14
- 2.3 Il n'y a rien d'aussi inconstant qu'une variable 15
- 2.4 Utilisation des variables 17
- 2.5 Un maillon de la chaîne ? 19
- 2.6 Tours de chaînes 20
- 2.7 Pas vraiment une liste de courses 21
- 2.8 N-uplets et listes 25
- 2.9 À vous de jouer 26
  - 2.9.1 Exercice 1 26
  - 2.9.2 Exercice 2 26
  - 2.9.3 Exercice 3 27

3 Tortues et autres choses lentes 29
- 3.1 Un reptile plus lent que Python 29
- 3.2 À la recherche de la tortue perdue 29
- 3.3 À vous de jouer 36

4 Comment poser une question 37
- 4.1 Avec des « si » on mettrait Paris en bouteille 37

4.2 Fait cela ! Ou sinon... 39
4.3 Fais cela, ou cela, ou cela ! Ou sinon... 40
4.4 Combiner des conditions 41
4.5 Rien 42
4.6 Quelle est la différence ? 44

5 Encore et encore **47**
5.1 Rabâchage 47
5.2 Quand est-ce qu'un bloc n'est pas compact ? 50
5.3 Tant que nous en sommes à parler de boucles... 58
5.4 À vous de jouer 61
5.4.1 Exercice 1 61
5.4.2 Exercice 2 61

6 Une sorte de recyclage **63**
6.1 De l'importance du recyclage 63
6.2 Modules 67
6.3 À vous de jouer 71
6.3.1 Exercice 1 71
6.3.2 Exercice 2 71
6.3.3 Exercice 3 71

7 Un court chapitre à propos des fichiers **73**
7.1 Sauvegarde des programmes 73
7.2 Si ça ne passe pas la fenêtre, passons par le soupirail 75
7.3 Un peu de lecture, maintenant ? 78
7.4 Écrivons
En Python ! (rime faible) 79

8 Tortues à profusion **83**
8.1 Le retour de la tortue 83
8.2 Je jure [...] de dire *toute* la vérité, rien que la vérité 85
8.3 Connaissez vous les couleurs ? 91
8.4 Remplissage 93
8.5 Obscurité 97
8.6 À vous de jouer 102
8.6.1 Exercice 1 102
8.6.2 Exercice 2 102

9 Un peu de graphiques **103**
9.1 Qui va lentement, va surement. 103
9.2 Tortue de course 104
9.3 Dessins simples 108
9.4 Dessiner des boîtes 110
9.5 Dessiner des ovales 117
9.6 Dessiner des arcs 119

9.7 Dessiner des polygones 120
9.8 Dessiner des images 122
9.9 Aminimations simples 124
9.10 Réagir aux événements 126

10 Où aller à partir de ce point ? 129

A Réponses aux « À vous de jouer » 131
A.1 Réponses aux cas pratiques de la section 2.9 131
A.2 Réponses aux cas pratiques de la section 3.3 132
A.3 Réponses aux cas pratiques de la section 5.4 133
A.4 Réponses aux cas pratiques de la section 6.3 135
A.5 Réponses aux cas pratiques de la section 8.6 137

B Quelques fonctions intégrées 141

C Tous les mots clef de Python 3 151

D Quelques modules de Python 165
D.1 Le module « `random` » 165
D.1.1 Introduction à random 165
D.1.2 randint 165
D.1.3 choice 166
D.2 Le module « `sys` » 167
D.2.1 Introduction à sys 167
D.2.2 exit 167
D.2.3 stdin 167
D.2.4 stdout 167
D.3 Le module « `locale` » 169
D.4 Le module « `time` » 169
D.4.1 Non je ne radote pas : time.time 169
D.4.2 asctime 170
D.4.3 ctime 171
D.4.4 localtime 172
D.4.5 sleep 172
D.4.6 strftime 173
D.4.7 strptime 174

E Gâteaux 177
E.1 Les textes en Tk 177

Index 179

Table des matières 181