Chapitre 1

Arithmétique, variables, instructions

Notions introduites

- les modes interactif et programme de Python
- expressions arithmétiques
- messages d'erreur
- manipuler des variables
- séquence d'instructions
- lire et écrire des chaînes de caractères

Le langage de programmation Python permet d'interagir avec la machine à l'aide d'un programme appelé *interprète Python*[1]. On peut l'utiliser de deux façons différentes. La première méthode consiste en un dialogue avec l'interprète. C'est le *mode interactif*. La seconde consiste à écrire un programme dans un fichier source puis à le faire exécuter par l'interprète. C'est le *mode programme*.

1.1 Mode interactif

En première approximation, le mode interactif de l'interprète Python se présente comme une calculatrice[2]. Les trois chevrons `>>>` constituent l'invite de commandes de Python, qui indique qu'il attend des instructions. Si par

1. Le site `https://www.nsi-premiere.fr` qui accompagne ce livre présente plusieurs environnements pour travailler avec Python.
2. On suppose ici que l'interprète Python vient d'être lancé, quelle que soit la solution retenue.

exemple on tape `1+2` puis la touche Entrée, l'interprète Python calcule et affiche le résultat.

```
>>> 1+2
3
>>>
```

Comme on le voit, les chevrons apparaissent de nouveau. L'interprète est prêt à recevoir de nouvelles instructions.

Arithmétique

En Python, on peut saisir des combinaisons arbitraires d'opérations arithmétiques, en utilisant notamment les quatre opérations les plus communes.

```
>>> 2 + 5 * (10 - 1 / 2)
49.5
>>>
```

L'addition est notée avec le symbole `+`, la soustraction avec le symbole `-`, la multiplication avec le symbole `*` et la division avec le symbole `/`. La priorité des opérations est usuelle et on peut utiliser des parenthèses. Dans l'exemple ci-dessus, on a utilisé des espaces pour améliorer la lisibilité. Ces espaces sont purement décoratifs et ne modifient pas la façon dont l'expression est calculée. En particulier, elles n'agissent pas sur la priorité des opérations.

```
>>> 1+2 * 3
7
```

Erreurs. L'interprète n'accepte que des expressions arithmétiques complètes et bien formées. Sinon, l'interprète indique la présence d'une erreur.

```
>>> 1 + * 2
  File "<stdin>", line 1
    1 + * 2
        ^
SyntaxError: invalid syntax
```

Ici, le message `SyntaxError: invalid syntax` indique une erreur de syntaxe, c'est-à-dire une instruction mal formée. Nous verrons plus loin d'autres catégories d'erreur. On peut ignorer pour l'instant la ligne `File "<stdin>", line 1`. Les deux lignes suivantes montrent à l'utilisateur l'endroit exact de l'erreur de syntaxe avec le symbole `^`, ici le caractère `*`.

Erreurs. Un autre type d'erreur se manifeste lorsque l'on donne à l'interprète une expression qui est correcte du point de vue de l'écriture mais dont le résultat n'a pas de sens. Ainsi toute tentative de division par zéro produit un message spécifique.

```
>>> 2 / (3 - 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Attention, les nombres « à virgule » s'écrivent à l'anglo-saxonne avec un point, et non avec une virgule qui a un autre sens en Python. En essayant d'appliquer des opérations arithmétiques à des nombres écrits avec des virgules on obtient toute une panoplie de comportements inattendus.

```
>>> 1,2 + 3,4
1, 5, 4
```

```
>>> 1,2 * 3
1, 6
```

En Python, la virgule sert à séparer deux valeurs. On en verra différentes utilisations aux chapitres 4, 5 et 14.

Les nombres de Python. Les nombres de Python sont soit des entiers relatifs, simplement appelés entiers, soit des nombres décimaux, appelés flottants.

Les entiers peuvent être de taille arbitraire (ce qui n'est pas le cas dans de nombreux langages de programmation). Ces entiers ne sont limités que par la mémoire disponible pour les stocker. L'exercice 6 page 22 propose une expérience pour observer que les entiers de Python peuvent être très grands. Le chapitre 19 reviendra sur la représentation des entiers dans un ordinateur.

Les nombres flottants, en revanche, ont une capacité limitée et ne peuvent représenter qu'une partie des nombres décimaux. Ainsi, des nombres décimaux trop grands ou trop petits ne sont pas représentables. Si on saisit un nombre décimal en tapant `1` suivi de cinq cents `0` et d'un point, on obtient la valeur `inf`, qui représente une valeur trop grande pour être représentée de cette manière. Des nombres comme π, $\sqrt{2}$, qui ne sont pas des nombres décimaux, ne peuvent être représentés que de manière approximative en Python. Ces approximations ne seront cependant pas problématiques dans les situations que nous rencontrerons. Le chapitre 20 reviendra sur la représentation des nombres réels dans un ordinateur.

À propos de la division. Si on effectue la division de deux entiers avec l'opération `/`, on obtient un nombre décimal. Ainsi, `7 / 2` donne le nombre `3.5`. (Le chapitre 20 expliquera comment de tels nombres décimaux sont représentés dans la machine.) Si on veut effectuer en revanche une division entière, alors il faut utiliser l'opération `//`. Ainsi, `7 // 2` donne le nombre `3`, qui est cette fois un entier, à savoir le quotient de 7 par 2 dans la division euclidienne. On peut également obtenir le reste d'une telle division euclidienne avec l'opération `%`. Ainsi, `7 % 2` donne l'entier `1`. (La division euclidienne est celle que l'on a apprise à l'école primaire.)
Attention : les opérations `//` et `%` de Python ne coïncident avec la division euclidienne que lorsque le diviseur est positif. Lorsqu'il est négatif, le reste est alors également négatif. Voici une illustration des quatre cas de figure :

	$a = 7$ $b = 3$	$a = -7$ $b = 3$	$a = 7$ $b = -3$	$a = -7$ $b = -3$
a `//` b	2	-3	-3	2
a `%` b	1	2	-2	-1

Dans tous les cas, on a l'égalité $a = (a \texttt{ // } b) \times b + a \texttt{ \% } b$ et l'inégalité $|a \texttt{ \% } b| < |b|$. Une autre façon de le voir, sans doute plus simple, consiste à définir $a \texttt{ // } b$ comme la partie entière (par défaut) du nombre réel a/b, c'est-à-dire le plus grand entier inférieur ou égal à a/b. Cela étant, il est rare que l'on divise par un nombre négatif. On peut même le décourager purement et simplement pour éviter toute différence accidentelle avec la division euclidienne.

Variables

Les résultats calculés peuvent être mémorisés par l'interprète, afin d'être réutilisés plus tard dans d'autres calculs.

```
>>> a = 1 + 1
>>>
```

La notation `a =` permet de donner un nom à la valeur à mémoriser. L'interprète calcule le résultat de `1+1` et le mémorise dans la *variable* `a`. Aucun résultat n'est affiché. On accède à la valeur mémorisée en utilisant le nom `a` [3].

```
>>> a
2
```

Plus généralement, la variable peut être réutilisée dans la suite des calculs.

3. On suppose que toutes les commandes tapées dans cette section sont tapées à la suite sans relancer à chaque fois l'interprète Python.

```
>>> a * (1 + a)
6
```

Le symbole = utilisé pour introduire la variable a désigne une opération d'*affectation*. Il attend à sa gauche un nom de variable et à sa droite une expression. On peut donner une nouvelle valeur à la variable a avec une nouvelle affectation. Cette valeur remplace la précédente.

```
>>> a = 3
>>> a * (1 + a)
12
```

Le calcul de la nouvelle valeur de a peut utiliser la valeur courante de a.

```
>>> a = a + 1
>>> a
4
```

Un nom de variable peut être formé de plusieurs caractères (lettres, chiffres et souligné). Il est recommandé de ne pas utiliser de caractères accentués et l'usage veut que l'on se limite aux caractères minuscules.

```
>>> cube = a * a * a
>>> ma_variable = 42
>>> ma_variable2 = 2019
```

Erreurs. Un nom de variable ne doit pas commencer par un chiffre et certains noms sont interdits (car ils sont des mots réservés du langage).

```
>>> 4x = 2
  File "<stdin>", line 1
    4x = 2
     ^
SyntaxError: invalid syntax
>>> def = 3
  File "<stdin>", line 1
    def = 3
        ^
SyntaxError: invalid syntax
```

Il n'est pas possible d'utiliser une variable qui n'a pas encore été définie.

```
>>> b + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Erreurs. Seul un nom de variable peut se trouver à la gauche d'une affectation.

```
>>> 1 = 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Une variable peut être imaginée comme une petite boîte portant un nom (ou une étiquette) et contenant une valeur. Ainsi, on peut se représenter une variable déclarée avec `x = 1` par une boîte appelée `x` contenant la valeur 1.

Lorsque l'on modifie la valeur de la variable `x`, par exemple avec l'affectation `x = x + 1`, la valeur 1 est remplacée par la valeur 2.

État

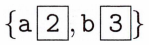
À chaque étape de l'interaction, chacune des variables introduites contient une valeur. L'ensemble des associations entre des noms de variables et des valeurs constitue l'*état* de l'interprète Python. Cet état évolue en fonction des instructions exécutées, et notamment en raison des affectations de nouvelles valeurs. On dit de ces instructions qui modifient l'état qu'elles ont un *effet de bord*.

On peut représenter l'état de l'interprète par un ensemble d'associations entre des noms de variables et des valeurs. Par exemple, l'ensemble {a 1, b 2, c 3} représente l'état dans lequel les variables `a`, `b` et `c` valent respectivement `1`, `2` et `3` et où aucune autre variable n'est définie.

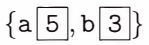
Pour simuler à la main une étape d'interaction, on indiquera l'état avant et après l'exécution de l'instruction. Considérons par exemple l'instruction suivante.

```
>>> a = a + b
```

Cette instruction modifie la valeur de `a` en fonction des valeurs de `a` et `b`. Par exemple, en partant de l'état



on obtient après exécution de l'instruction un état où l'association a 5 a remplacé a 2.



Différences entre mathématiques et informatique. En informatique, le terme de *variable* est utilisé pour indiquer le fait que la valeur associée peut *varier* au fur et à mesure de l'exécution du programme. Ce n'est donc pas la même notion que celle de variable en mathématiques, désignant elle une unique valeur (qui est, ne serait-ce que provisoirement, indéterminée).
Par ailleurs, on a parfois des manières différentes d'écrire les choses dans ces deux domaines, et le même symbole peut ne pas avoir la même signification en informatique et en mathématiques. C'est le cas en particulier du symbole d'égalité, dont l'utilisation en informatique peut paraître étrange si on la confond avec sa signification mathématique. La suite de symboles `a = a + 1` par exemple n'a pas de sens si on la voit comme une égalité. Il faut la voir comme une instruction, qui donne comme nouvelle valeur à la variable `a` le résultat de l'expression `a + 1` calculé avec la valeur courante de `a`. En l'occurrence, on vient donc d'augmenter la valeur de `a` d'une unité.
En conséquence de cette nature de l'instruction d'affectation, même la désignation de la variable par `a` peut avoir deux sens différents en informatique. La plupart du temps `a` désigne la valeur de la variable, c'est-à-dire le contenu de la boîte associée, mais à gauche du symbole d'affectation `a` est une *référence* à la boîte elle-même, dont on va modifier le contenu.
Comme il est courant de modifier la valeur d'une variable en lui ajoutant une certaine quantité, il existe en outre une instruction spécifique pour cela, notée `+=`. Ainsi, on peut, en informatique, écrire `a += 1` au lieu de `a = a + 1`.

Si d'autres variables sont définies dans l'état de départ et ne sont pas affectées par l'instruction, leur valeur reste inchangée. Ainsi, partant de

$$\{\texttt{a}\boxed{0}, \texttt{b}\boxed{1}, \texttt{d}\boxed{-12}, \texttt{x}\boxed{3}\}$$

on obtient l'état suivant.

$$\{\texttt{a}\boxed{1}, \texttt{b}\boxed{1}, \texttt{d}\boxed{-12}, \texttt{x}\boxed{3}\}$$

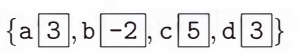
Lorsqu'une variable est affectée pour la première fois, l'association correspondante apparaît dans l'état. Ainsi, pour une instruction `a = b + c` et en partant de l'état

$$\{\texttt{b}\boxed{-2}, \texttt{c}\boxed{5}\}$$

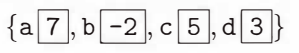
on obtient l'état suivant.

$$\{\texttt{a}\boxed{3}, \texttt{b}\boxed{-2}, \texttt{c}\boxed{5}\}$$

Dans le cas particulier où on affecte une nouvelle variable avec la valeur d'une variable qui existe déjà, par exemple avec `d = a`, il est important de comprendre que `d` et `a` ne sont pas deux noms pour la même boîte, mais deux boîtes différentes, la boîte `d` recevant la valeur contenue dans la boîte `a`.



On peut observer que les deux variables sont effectivement indépendantes : modifier la variable `a`, par exemple avec `a = 7`, n'a pas d'effet sur la variable `d`.



1.2 Mode programme

Le mode programme de Python consiste à écrire une suite d'instructions dans un fichier et à les faire exécuter par l'interprète Python. Cette suite d'instructions s'appelle un *programme*, ou encore un *code source*. On évite ainsi de ressaisir à chaque fois les instructions dans le mode interactif. Par ailleurs, cela permet de distinguer le rôle de programmeur du rôle d'utilisateur d'un programme.

Affichage

En mode programme, les résultats des expressions calculées ne sont plus affichés à l'écran. Il faut utiliser pour ceci une instruction explicite d'affichage. En Python, elle s'appelle `print`. Par exemple, dans un fichier portant le nom `test.py`, on peut écrire l'instruction suivante.

```
print(3)
```

On peut alors faire exécuter ce programme par l'interprète Python[4], ce qui affiche `3` à l'écran. L'instruction `print` accepte une expression arbitraire. Elle commence par calculer le résultat de cette expression puis l'affiche à l'écran. Ainsi, l'instruction

```
print(1+3)
```

calcule la valeur de l'expression `1+3` puis affiche `4` à l'écran.

Affichage de textes

L'instruction `print` n'est pas limitée à l'affichage de nombres. On peut lui donner un message à afficher, écrit entre guillemets. Par exemple, l'instruction

4. Voir le site `https://www.nsi-premiere.fr` pour les différentes manières d'appeler l'interprète Python sur un fichier.

```
print("Salut tout le monde !")
```

affiche le message `Salut tout le monde !` à l'écran. Le texte écrit entre guillemets est appelé une *chaîne de caractères*. Les caractères accentués sont autorisés, tout comme les caractères provenant d'autres langues, les smileys et plus généralement tous les caractères Unicode. On note que les guillemets englobants ne sont pas affichés.

Il est important de comprendre que le contenu d'une chaîne est arbitraire et n'est pas interprété par Python. Pour s'en convaincre, on peut observer la différence entre l'expression arithmétique `1+3` et la chaîne de caractères `"1+3"` en exécutant l'instruction suivante

```
print("1+3")
```

qui affiche simplement `1+3` à l'écran.

Erreurs. Que la valeur à afficher soit donnée directement ou soit le résultat d'un calcul, les parenthèses font partie de la syntaxe de l'instruction `print`. Si on omet les parenthèses, on obtient une erreur.

```
File "test.py", line 1
  print 3
        ^

SyntaxError: Missing parentheses in call to 'print'
```

La première ligne du message d'erreur indique le nom du fichier et la ligne où se situe l'erreur.

Erreurs. Des guillemets ouverts doivent impérativement être fermés explicitement. On obtiendra sinon une erreur.

```
File "test.py", line 1
  print("1)
          ^

SyntaxError: EOL while scanning string literal
```

On peut aussi noter que les chaînes de caractères et les nombres sont des entités de natures différentes. Leur combinaison par une opération telle que l'addition n'a pas de sens, et provoque une erreur.

```
>>> 1 + "2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
  'int' and 'str'
```

Séquence d'instructions

Un programme est généralement constitué de plusieurs instructions. Chaque instruction est écrite sur une ligne. L'interprète Python les exécute l'une après l'autre, dans l'ordre du fichier. Ainsi, l'exécution du programme

```
a = 34
b = 21 + a
print(a)
print(b)
```

affiche deux entiers à l'écran, sur deux lignes successives.

```
34
55
```

Si on souhaite afficher les deux entiers sur une même ligne, on peut remplacer les deux instructions `print` par une seule, en séparant les deux éléments à afficher par une virgule.

```
print(a, b)
```

Les deux éléments sont affichés sur la même ligne et séparés par un caractère espace.

```
34 55
```

Plus généralement, on peut utiliser `print` avec un nombre arbitraire d'éléments, qui peuvent être des nombres ou des chaînes de caractères.

```
a = 34
b = 55
print("la somme de", a, "et de", b, "vaut", a+b)
```

L'exécution de ce programme affiche le message suivant.

```
la somme de 34 et de 55 vaut 89
```

Interagir avec l'utilisateur

Pour permettre l'interaction du programme avec l'utilisateur, par exemple la saisie de la valeur d'une variable, il faut procéder en deux temps : d'abord utiliser l'instruction `input` pour récupérer des caractères tapés au clavier par l'utilisateur, puis utiliser l'instruction `int` pour convertir cette chaîne de caractères en un nombre entier.

```
s = input()
a = int(s)
print("le nombre suivant est ", a + 1)
```

Instruction print et retour à la ligne. Par défaut, l'instruction **print** provoque un retour à la ligne après l'affichage. Ainsi deux instructions **print** successives affichent deux lignes. On peut changer ce comportement en fournissant une autre chaîne de caractères à accoler à l'affichage principal, par exemple une espace ou même rien. Ainsi le programme

```
print("abc", end=" ")
print("def", end="")
print("gh")
```

affiche la ligne suivante

```
abc defgh
```

puis revient à la ligne avec la troisième instruction **print** .

L'instruction `input` interrompt l'exécution du programme et attend que l'utilisateur saisisse des caractères au clavier. La saisie se termine lorsqu'il appuie sur la touche Entrée. Ici, la suite des caractères saisis par l'utilisateur est stockée dans une variable `s`. Dans un second temps, la variable `a` reçoit le nombre représenté par cette suite de caractères. Ainsi, si on exécute ce programme et qu'on saisit successivement les caractères `2`, `7` et la touche Entrée, le programme affiche la ligne suivante.

```
le nombre suivant est 28
```

Erreurs. Si la suite des caractères saisis par l'utilisateur ne représente pas un nombre, on obtiendra une erreur. Par exemple, si on saisit par accident le caractère `è` au lieu de `7`, on obtient le message suivant.

```
File "test.py", line 2
ValueError: invalid literal for int() with base 10: '2è'
```

En mode programme, lorsqu'une instruction provoque une erreur, l'interprète s'arrête, affiche le message d'erreur et n'exécute pas le reste du programme.

Un programme complet

Nous avons maintenant tous les ingrédients pour écrire un programme complet (voir encadré Programme 1). Il s'agit d'un programme qui demande son année de naissance à l'utilisateur, puis calcule et affiche son âge en 2048. La première ligne est un *commentaire*. C'est une séquence de caractères qui est ignorée par l'interprète et dont le but est uniquement de documenter le

Programme 1 — calcul de l'âge

```
1 # calcul de l'âge
2 saisie = input("Entrez votre année de naissance : ")
3 annee = int(saisie)
4 age = 2048 - annee   # on calcule l'âge par soustraction
5 print("Vous aurez", age, "ans en 2048.")
```

programme. En Python, un commentaire commence par un symbole `#` et se poursuit jusqu'à la fin de la ligne. Il peut contenir n'importe quelle suite de caractères. La deuxième ligne utilise l'instruction `input` pour demander son année de naissance à l'utilisateur. Comme on le voit, on peut ajouter à l'instruction `input` une chaîne de caractères, qui sera affichée juste avant la saisie. Le résultat de cette saisie est une chaîne de caractères stockée dans la variable `saisie`. La troisième ligne convertit la saisie en entier avec l'instruction `int`, et l'entier représentant l'année de naissance est stocké dans la variable `annee`. Le programme calcule ensuite l'âge de l'utilisateur en 2048 par une soustraction et le stocke dans la variable `age`. On note ici la présence d'un second commentaire, écrit sur la même ligne qu'une instruction. Là encore, le commentaire débute avec le symbole `#` et se poursuit jusqu'à la fin de la ligne. Enfin, la dernière ligne du programme affiche l'âge de l'utilisateur avec l'instruction `print`.

Représentation de l'exécution

On peut représenter l'exécution d'un programme complet par la séquence des états correspondant à chaque instruction exécutée. On présente une telle exécution dans un tableau où chaque ligne contient un numéro désignant l'instruction exécutée, l'état de l'interprète après l'exécution de l'instruction, et les éventuelles interactions avec l'utilisateur au cours de l'exécution.

Ligne	État	Interactions
2	saisie `"1985"`	affichage : `Entrez votre année de ...` saisie : 1985
3	saisie `"1985"` annee `1985`	
4	saisie `"1985"` annee `1985` age `63`	
5	saisie `"1985"` annee `1985` age `63`	affichage : `Vous aurez 63 ans en 2048.`

À propos de l'opération `int`. L'opération `int` convertit une chaîne de caractères en un entier. La chaîne doit être uniquement composée de chiffres et éventuellement du symbole `-` en tête, sans quoi une erreur est levée. Par défaut, l'entier est lu en base 10 mais une autre base peut être spécifiée, avec la syntaxe `int(`*chaîne*`,` *base*`)`. Ainsi, `int("101010", 2)` interprète l'entier `101010` comme étant écrit en base 2, c'est-à-dire $2^5 + 2^3 + 2^1 = 42$. La base doit être comprise entre 2 et 36 et les lettres de `A` à `Z` sont utilisées pour représenter respectivement les chiffres de 10 à 35. Ainsi, `int("-2A", 16)` interprète l'entier `-2A` comme étant écrit en base 16, c'est-à-dire $-(2 \times 16 + 10) = -42$.
Si on souhaite interpréter la chaîne de caractères comme un nombre décimal plutôt que comme un nombre entier, il faut utiliser l'opération `float` à la place de l'opération `int`. Ainsi, `float("3.5")` est accepté là où `int("3.5")` provoque une erreur.

Composition. Deux instructions

```
s = input("Entrer un nombre :")
a = int(s)
```

peuvent être *composées* en une seule de la manière suivante.

```
a = int(input("Entrer un nombre :"))
```

L'effet obtenu est quasiment identique : cette version composée récupère de même une saisie de l'utilisateur sous la forme d'une chaîne de caractères et la convertit en un nombre entier, le nombre obtenu étant stocké dans la variable `a`. Dans cette dernière version cependant la chaîne de caractères intermédiaire n'est pas stockée dans une variable et n'est donc plus accessible par ailleurs.

Mode interactif et mode programme. Même si le mode programme est le mode d'utilisation standard, le mode interactif reste utile pour mettre au point un programme ou encore tester un élément du langage Python sur lequel on a un doute. Si on utilise l'environnement de développement Idle (voir `https://www.nsi-premiere.fr`), on dispose à la fois d'une fenêtre dans laquelle on écrit le texte du programme et d'une seconde fenêtre correspondant au mode interactif. Les deux modes peuvent donc être utilisés simultanément.

1.3 Bibliothèque Turtle

Python propose un certain nombres d'instructions primitives pour les besoins les plus courants, comme **print** et **input**. Le langage contient également de très nombreuses *bibliothèques*, qui apportent des collections d'instructions plus spécialisées permettant d'effectuer de nouvelles tâches.

Par exemple, il existe une bibliothèque `random` donnant accès à différentes instructions produisant des nombres aléatoires. Elle offre en particulier une instruction `randint` pouvant être utilisée sous la forme `n = random.randint(1, 6)` pour tirer au hasard un nombre entier entre `1` et `6` inclus, et stocker ce nombre dans la variable `n`, ou encore une instruction `random` pouvant être utilisée sous la forme `x = random.random()` pour tirer au hasard un nombre décimal entre `0` inclus et `1` exclu, et stocker ce nombre dans la variable `x`. L'utilisation de telles instructions demande une déclaration préalable, faite avec la ligne suivante.

```
import random
```

Cette ligne n'a besoin d'apparaître qu'une seule fois, en général au début du programme, pour chaque bibliothèque dont on voudra utiliser des instructions. Elle permet d'accéder à toutes les instructions de la bibliothèque.

Nous découvrirons différentes bibliothèques de Python à mesure des besoins de ce cours. Commençons ici par la bibliothèque `turtle`, qui permet de reproduire les fonctionnalités de base du langage de programmation éducatif Logo. Les instructions de ce langage font se déplacer une tortue munie d'un crayon à la surface d'une feuille virtuelle. On peut alors observer l'exécution du programme à travers les mouvements de la tortue et le tracé qu'elle laisse derrière elle.

Les instructions de `turtle` comprennent en premier lieu des moyens d'orienter et déplacer la tortue dans le plan cartésien à deux dimensions (en utilisant le repère standard des mathématiques, avec abscisses sur un axe horizontal croissant vers la droite et ordonnées avec axe vertical croissant vers le haut).

instruction	description
`goto(`x`, `y`)`	aller au point de coordonnées (x, y)
`forward(`d`)`	avancer de la distance d
`backward(`d`)`	reculer de la distance d
`left(`a`)`	pivoter à gauche de l'angle a
`right(`a`)`	pivoter à droite de l'angle a
`circle(`r`, `a`)`	tracer un arc de cercle d'angle a et de rayon r
`dot(`r`)`	tracer un point de rayon r

La tortue commence au point de coordonnées $(0, 0)$, situé au centre de l'écran, et est orientée par l'axe des abscisses. L'axe des ordonnées est orienté vers le haut. Les coordonnées et distances sont mesurées en pixels et les

Variantes sur l'utilisation des bibliothèques. Il existe plusieurs moyen d'éviter d'écrire entièrement le nom de la bibliothèque dans chaque instruction, comme le `random.` de `random.randint` ou `random.random`.

- On peut associer un nom plus court à la bibliothèque au moment de la charger avec le mot clé `as`.

```
>>> import random as r
>>> r.randint(1, 6)
4
```

- On peut charger explicitement certaines des instructions avec la combinaison **from / import**. Par exemple ici, les instructions `sqrt` et `cos` de la bibliothèque `math`, qui calculent une racine carrée et un cosinus.

```
>>> from math import sqrt, cos
>>> sqrt(2)
1.4142135623730951
```

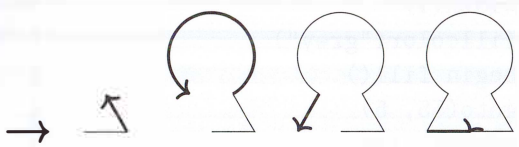
- On peut charger l'intégralité des instructions d'une bibliothèque en une seule fois avec une étoile `*`..

```
>>> from turtle import *
```

La dernière solution peut paraître pratique mais est plus délicate qu'il n'y paraît : deux bibliothèques peuvent contenir des instructions avec le même nom qui entreraient alors en conflit. Aussi on n'utilisera jamais cette possibilité sur plus d'une bibliothèque à la fois.

angles en degrés. Les arcs de cercles sont parcourus dans le sens trigonométrique si le rayon est positif, et sens horaire si le rayon est négatif. Le programme ci-dessous à gauche trace donc un dessin en suivant les étapes détaillées à droite.

```
from turtle import *
forward(60)
left(120)
forward(60)
right(90)
circle(60, 300)
right(90)
forward(60)
goto(0, 0)
```

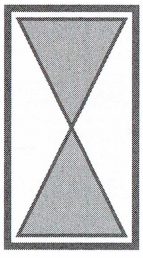


S'ajoutent à cette base une série d'instructions permettant de modifier les dessins produits par chacun des déplacements.

instruction	description
`up()`	relever le crayon (et interrompre le dessin)
`down()`	redescendre le crayon (et reprendre le dessin)
`width(`e`)`	fixer à e l'épaisseur du trait
`color(`c`)`	sélectionner la couleur c pour les traits
`begin_fill()`	activer le mode remplissage
`end_fill()`	désactiver le mode remplissage
`fillcolor(`c`)`	sélectionner la couleur c pour le remplissage

Par défaut les tracés sont faits en noir avec une épaisseur d'un pixel. Les couleurs peuvent être désignées par leur nom, sous la forme de chaînes de caractères, avec notamment : `"black"`, `"white"`, `"grey"`, `"pink"`, `"purple"`, `"blue"`, `"green"`, `"yellow"`, `"orange"`, `"red"`, `"brown"`. Pour plus de souplesse on peut également fournir un triplet de nombres compris entre 0 et 1 indiquant les niveaux respectifs de rouge, vert et bleu composant la couleur : on obtient du rouge avec `color(1, 0, 0)`, un gris foncé avec `color(0.3, 0.3, 0.3)`, une certaine teinte de violet avec `color(0.5, 0, 0.6)`, etc. Toute l'aire contenue à l'intérieur de la trajectoire de la tortue pendant une période de temps où le mode remplissage est activé prend la couleur choisie pour le remplissage (noir par défaut).

```
from turtle import *
# Rectangle épais
width(6)
color(0.2, 0.2, 0.2)
goto(60, 0)
goto(60, 110)
goto(0, 110)
goto(0, 0)
# Déplacement
up()
goto(5, 5)
down()
# Sablier gris clair
width(1)
fillcolor("grey")
begin_fill()
goto(55, 5)
goto(5, 105)
goto(55, 105)
goto(5, 5)
end_fill()
```



Enfin, quelques instructions permettent de configurer l'action de la bibliothèque Turtle en général.

instruction	description
`reset()`	tout effacer et recommencer à zéro
`speed(`s`)`	définir la vitesse de déplacement de la tortue
`title(`t`)`	donner le titre t à la fenêtre de dessin
`ht()`	ne montre plus la tortue (seulement le dessin)

Les vitesses possibles sont à choisir parmi les chaînes de caractères `"slowest"`, `"slow"`, `"normal"`, `"fast"`, `"fastest"`, ou parmi les nombres entiers entre 0 et 10 : 1 est le déplacement le plus lent, 10 le déplacement le plus rapide, et 0 le déplacement instantané.

Erreurs. Utiliser des instructions Turtle avec des valeurs invalides peut produire des effets variés, souhaitables ou non.

- Utiliser une mauvaise chaîne de caractères ou une mauvaise valeur avec `color` ou `speed` peut déclencher une erreur immédiate. Le message d'erreur est assez massif et la majeure partie, éludée ici, fait référence aux mécanismes internes de Turtle qu'on ne détaillera pas. La dernière ligne contient en revanche un bilan du problème.

```
>>> color("rouge")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 8, in color
  ...
turtle.TurtleGraphicsError: bad color string: rouge
```

- Plus délicat, Python ne relève pas d'erreur lors de la définition d'une valeur négative pour l'épaisseur du trait.

```
width(-5)
```

L'erreur survient au moment de tracer un trait avec une telle épaisseur invalide. À nouveau le message est imposant, mais sa dernière ligne contient l'information essentielle pour nous.

```
>>> forward(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
_tkinter.TclError: bad screen distance "-5"
```

À retenir. Un **langage de programmation** permet d'écrire des programmes en vue de leur exécution par un ordinateur. En **mode interactif**, l'utilisateur fournit des instructions une à une et peut **observer** immédiatement les résultats, ce qui est intéressant notamment pour expérimenter. En **mode programme**, le programmeur rassemble toutes les instructions dans un **fichier source**. La séquence d'instructions est ensuite exécutée d'une seule traite, des opérations de **lecture** permettant de demander des informations à l'utilisateur et des opérations d'**affichage** permettant de faire un compte-rendu.
Une **variable** est un nom associé à une valeur. La valeur d'une variable peut être consultée, mais aussi mise à jour grâce à l'instruction d'**affectation**.

Activité : erreurs en situation contrôlée. Un programmeur débutant peut passer beaucoup de temps à traquer les erreurs dans ses programmes. Une bonne manière de se préparer à cela consiste à s'entraîner dans des situations contrôlées. Voici trois exercices à réaliser, et à reprendre régulièrement à toutes les étapes de l'apprentissage de la programmation.

1. Prendre un programme correct, c'est-à-dire un programme qui s'exécute sans erreur et produit le bon résultat, qui peut être issu de la leçon comme d'un exercice. Le modifier. Exécuter la version modifiée et observer quelles erreurs apparaissent ou en quoi le résultat change. Voici par exemple une version du programme 1 avec plusieurs modifications (à expérimenter une par une).

```
saisie = in("Entrez votre année de naissance: ")
annee = int(sasie)
age += 2048 - annee
print("Vous aurez", age, " en 2048)
```

2. Procéder comme au point précédent, mais essayer de prédire ce qui va se passer avant d'exécuter le programme modifié.
3. À deux personnes. La première prend le rôle de programmeur. Elle choisit un programme correct, le modifie pour introduire une erreur (ou plusieurs pour complexifier l'exercice), et donne le code modifié à la deuxième personne en lui précisant le résultat qui était attendu. La deuxième personne, dans le rôle de correcteur, doit trouver un moyen de corriger le programme pour obtenir le résultat voulu.

Activité : suivi de l'exécution. L'apprentissage de la programmation nécessite d'assimiler l'articulation entre le programme qui n'est lui-même qu'un texte écrit inerte, statique, et son exécution qui est un processus actif, dynamique. L'une des difficultés est que le processus d'exécution contient généralement un grand nombre d'étapes, mais que nous n'en observons souvent que le résultat final avec par exemple l'affichage d'un résultat. Il est utile de savoir reproduire mentalement le cheminement intégral d'un programme, et les exercices suivants, à réaliser régulièrement, peuvent y aider.

1. Prendre un programme correct issu de la leçon ou d'un exercice, et tracer le tableau représentant chacune des étapes de son exécution. Commencer par exemple avec le programme 1 en choisissant pour la saisie son année de naissance au lieu de `1985`.
2. Procéder de même avec un programme incorrect, c'est-à-dire un programme produisant une erreur ou le mauvais résultat, et observer l'état au moment où le problème se manifeste et la suite des évènements ayant mené à cet état.
3. Prendre un programme, correct ou non, et y insérer des instructions **print** pour observer les valeurs de certaines variables à différentes étapes de son exécution.

Exercices

Exercice 1 Observer les résultats suivants donnés par l'interprète Python.

```
>>> 5 - 3 - 2
0
>>> 1 / 2 / 2
0.25
```

Qu'en déduire sur la manière dont sont interprétées les soustractions et les divisions enchaînées ? Écrire des expressions permettant d'observer la manière dont Python interprète les enchaînements mélangeant additions et soustractions, ou multiplications et divisions. Solution page 417 □

Exercice 2 Réécrire les expressions suivantes en explicitant toutes les parenthèses :

1. `1 + 2 * 3 - 4`
2. `1+2 / 4*3`
3. `1-a+a*a/2-a*a*a/6+a*a*a*a/24`

Solution page 417 □

Exercice 3 Réécrire les expressions suivantes en utilisant aussi peu de parenthèses que possible sans changer le résultat.

1. `1+(2*(3-4))`
2. `(1+2)+((5*3)+4)`
3. `(1-((2-3)+4))+(((5-6)+((7-8)/2)))`

Solution page 417 □

Exercice 4 Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
>>> a = 3
>>> a = 4
>>> a = a+2
>>> a
```

Solution page 417 □

Exercice 5 Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
>>> a = 2
>>> b = a*a
>>> b = a*b
>>> b = b*b
>>> b
```

Solution page 417 □

Exercice 6 Dans le mode interactif, initialiser une variable `a` avec la valeur 2, puis répéter dix fois l'instruction `a = a * a`. Observer le résultat. Quelle puissance de 2 a-t-on ainsi calculée ? Solution page 417 □

Exercice 7 Qu'affichent les instructions suivantes ?

1. `print ("1+")`
2. `print (1+)`

Solution page 417 □

Exercice 8 Que se passe-t-il quand on exécute le programme suivant ?

```
a = input("saisir un nombre : ")
print("le nombre suivant est ", a+1)
```

Le rectifier si nécessaire. Solution page 418 □

Exercice 9 Que fait la séquence d'instructions suivante ? On supposera qu'à l'origine les variables `a` et `b` contiennent chacune un nombre entier.

```
tmp = a
a = b
b = tmp
```

Solution page 418 □

Exercice 10 On met deux entiers dans deux boîtes a et b, par exemple 55 et 89. On remplace le contenu de a par la somme de celui de a et de b. Puis on remplace le contenu de b par le contenu de a moins le contenu de b. Enfin, on remplace le contenu de a par son contenu moins celui de b. Que contiennent a et b à la fin de ces opérations ? Programme cet algorithme en Python.
Solution page 418 □

Exercice 11 Écrire un programme qui demande à l'utilisateur les longueurs (entières) des deux côtés d'un rectangle et affiche son aire.
Solution page 418 □

Exercice 12 Écrire un programme qui demande à l'utilisateur d'entrer une base (entre 2 et 36) et un nombre dans cette base et qui affiche ce nombre en base 10. La notation `int(`*chaîne*`,` *base*`)` permet de convertir une chaîne représentant un entier dans une base donnée en un entier Python.
Solution page 418 □

Exercice 13 Écrire un programme qui demande à l'utilisateur d'entrer un nombre de secondes et qui l'affiche sous la forme d'heures/minutes/secondes.
Solution page 418 □

Exercice 14 On souhaite écrire un programme qui demande à l'utilisateur un nombre d'œufs et affiche le nombre de boîtes de 6 œufs nécessaires à leur transport. On considère ce programme, qui utilise la division euclidienne.

```
n = int(input("combien d'oeufs : "))
print(n // 6)
```

Tester ce programme sur différentes entrées.

1. Sur quelles valeurs de `n` ce programme est-il correct ?
2. Pourquoi n'est-il pas correct de remplacer `n // 6` par `n // 6 + 1` ?
3. Proposer une solution correcte.

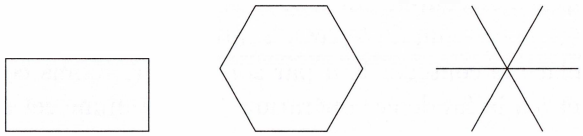
Solution page 419 □

Exercice 15 On souhaite calculer les coordonnées du point d'intersection de deux droites données sous la forme

$$\begin{cases} y = ax + b \\ y = cx + d \end{cases}$$

On suppose ici que a, b, c et d sont des entiers. Écrire un programme qui demande ces quatre valeurs à l'utilisateur puis affiche les coordonnées du point d'intersection. Que se passe-t-il si les droites sont parallèles ?
Solution page 419 □

Exercice 16 Reproduire avec Turtle les dessins suivants.



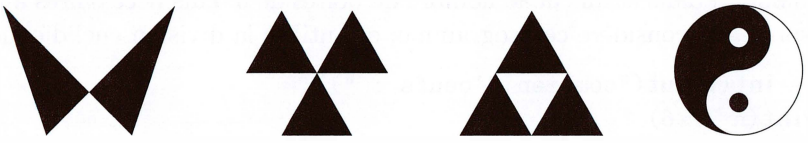
Solution page 419 □

Exercice 17 Que trace le programme Turtle suivant ?

```
goto(20, 0)
goto(0, 20)
goto(20, 20)
goto(0, 0)
goto(0, 20)
goto(10, 30)
goto(20, 20)
goto(20, 0)
```

Solution page 420 □

Exercice 18 Reproduire les dessins suivants avec Turtle, et en particulier les instructions `begin_fill()` et `end_fill()`.



Solution page 420 □

Exercice 19 Reproduire les dessins suivants avec Turtle, et en particulier l'instruction `width`.

Solution page 421 □

Exercice 20 Utiliser Turtle, et notamment les instructions `fill` et `fillcolor`, pour dessiner un drapeau français. Faire de même avec d'autres drapeaux au choix. Solution page 422 □

Exercice 21 Utiliser Turtle, et notamment les instructions `circle`, `color` et `width` pour dessiner un arc en ciel. Solution page 422 □