Objectifs du cours

Le présent document va aborder les concepts courants de la programmation, que nous allons appliquer à l'aide du langage de programmation Python dans sa version 3. Cet ouvrage a été conçu afin d'être utilisé en autonomie par un étudiant désireux d'apprendre à programmer mais ne disposant pas d'antécédents dans ce domaine. Il est composé d'un cours complet pouvant être approfondi par des recherches personnelles, de travaux dirigés (TD) et pratiques (TP).

Installation de Python 3

Pour réussir ce cours, nous allons installer les outils nécessaires sur Debian, Ubuntu et Microsoft Windows. La suite de cours sera conçu pour les systèmes Debian et Ubuntu.

Debian et Ubuntu

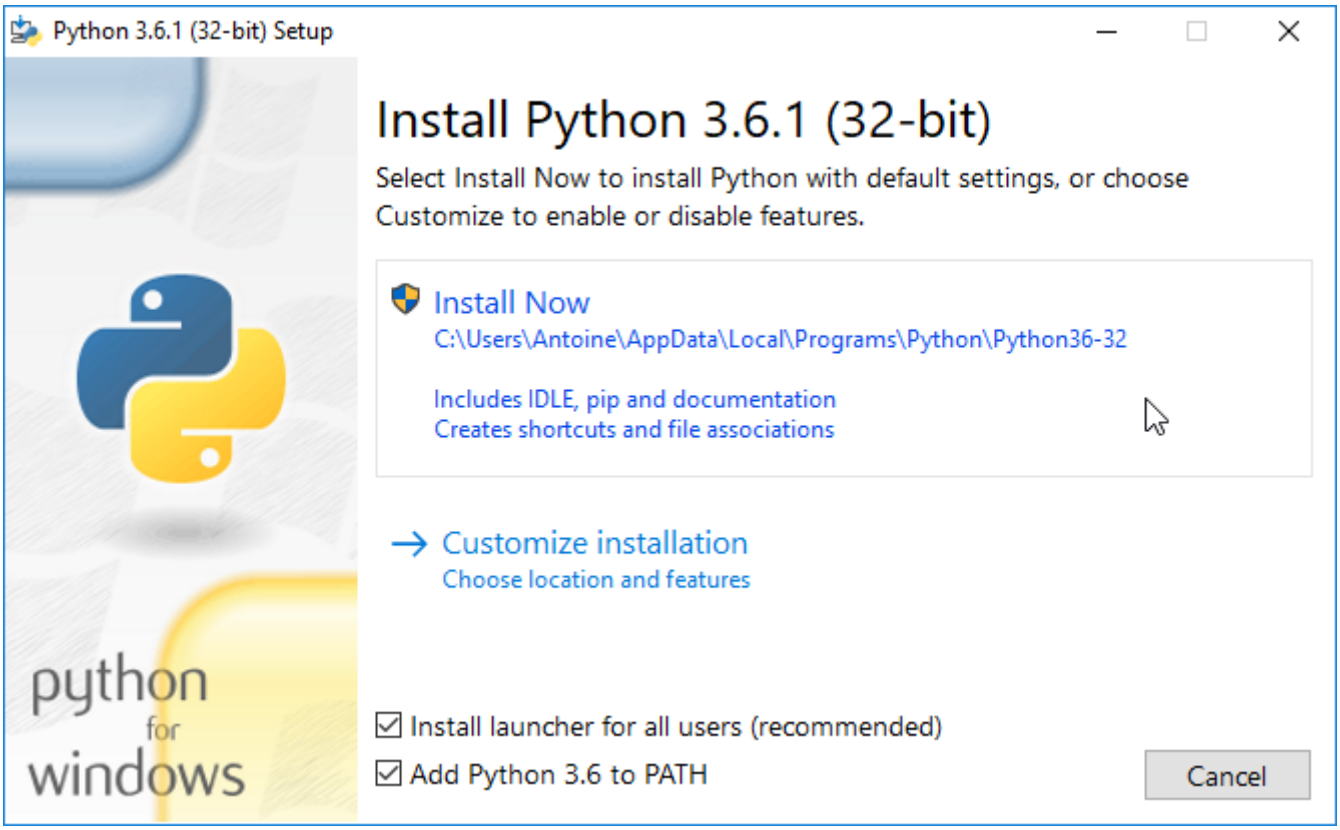
Pour ces systèmes, il est nécessaire d'installer Python 3 ainsi que la bibliothèque PySide2 :

```
apt install python3 python3-pyside2.qtwidgets python3-pyside2.qtcore
python3-pyside2.qtgui
```

Il vous sera également nécessaire d'utiliser un éditeur de texte. Libre à vous d'utiliser l'éditeur de votre choix (Atom, Geany, Vim, Emacs, nano …).

Microsoft Windows

Nous allons télécharger et installer Python 3. Pour cela, rendez-vous sur https://www.python.org/. Cliquez sur **Download** puis choisissez **Download Python 3.X.X**. Exécutez l'installeur. Cochez **Add Python 3.X to PATH** et cliquez sur **Install Now**.



Nous allons ensuite installer la bibliothèque PySide2, utilisée lors de ce cours. Pour cela, ouvrez un invité de commande et saisissez :

```
pip install -U PySide2
```

Il vous sera également nécessaire d'utiliser un éditeur de texte. Libre à vous d'utiliser l'éditeur de votre choix (Atom, Geany, Notepad++ …).

Chap 1 : Premiers pas avec Python 3

Nous allons débuter ce cours en effectuant des opérations à l'aide de l'interpréteur Python. En effet, il est possible d'utiliser Python via l'interpréteur ou en interprétant un fichier source. L'utilisation de l'interpréteur est recommandée pour expérimenter une fonctionnalité. Il nous servira de "cahier de brouillon" que vous pourrez utiliser tout au long de ce cours. Les

chapitres ultérieurs se concentreront sur l'écriture de scripts destinés à être sauvegardés et interprétés.

Au lancement de l'interpréteur Python, vous obtenez ceci :

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Ces quelques lignes nous indiquent la version de Python (ici 3.5.2) et que l'interpréteur est prêt à exécuter des commandes avec les caractères >>>. Dans les exemples de ce cours, ces mêmes caractères permettent de signaler qu'on utilise l'interpréteur.

Opérations arithmétiques

Pour vous familiariser avec le fonctionnement d'un invité de commandes, nous allons effectuer quelques opérations mathématiques simples :

- 10 + 9
- 5 × 11
- 4 + 7 × 3
- (4 + 7) × 3
- 40 ÷ 6 (division exacte)
- 40 ÷ 6 (division euclidienne soit la partie entière du résultat)
- 40 modulo 6 (reste de la division ci-dessus)
- 27,6 + 4,2

Pour cela, nous allons saisir les commandes suivantes :

```
>>>> 10+9
19
>>> 5*11
55
>>> 4 + 7*3     # Les espaces sont ignorés
25
>>> (4+7) * 3
33
>>> 40/6
6.666666666666667
>>> 40//6
6
>>> 40%6
4
>>> 27,6+4,2    # Ne fonctionne pas avec la virgule, doit être un point
>>> 27.6+4.2
31.8
```

Voici donc un tableau récapitulatif des différentes opérations arithmétiques de bases abordées :

Opération	Syntaxe
Addition	`a+b`

Soustraction	`a-b`
Multiplication	`a*b`
Division exacte	`a/b`
Division entière	`a//b`
Modulo	`a%b`
Puissance (a^b)	`a**b`
Arrondi de a avec b décimales	`round(a, b)`

Il faut ajouter à cela les opérateurs logiques que nous aborderons plus tard. **N'oubliez pas que Python respecte la priorité des opérations mathématiques.**

Les variables

À l'instar des mathématiques, nous allons utiliser des variables afin de stocker des données et travailler avec. Ces variables contiennent des données numériques (au format binaire) pouvant représenter :

- Des nombres entiers (dits entiers) et des nombres réels (la virgule est symbolisée par un **point**)
- Un texte (ou chaîne de caractères). Syntaxe : `"Bonjour à tous"`
- Des listes. Syntaxe : `[17, 12.3, "Poire"]`
- Des listes associatives. Syntaxe : `{"produit":"Pomme","prix":14.2}`
- Une fonction
- Un booléen (`True` = vrai et `False` = faux)
- …

On parle alors de **types** de variables. Nous détaillerons le fonctionnement et les opérations pouvant être effectuées sur ces différents types de variables plus tard.

Chaque variable est identifiée à partir d'un nom que vous donnez. Ce nom est à choisir afin qu'il respecte les consignes suivantes :

- Le nom de la variable doit être court mais indiquer son contenu pour faciliter la lecture du code.
- Ne doit pas être un **nom réservé** :

```
and      as       assert   break    class    continue  def     del   elif
else     except   false    finally  for      from      global  if    import
in       is       lambda   none     nonlocal not       or      pass  raise
return   true     try      while    with     yield
```

- Ne doit comporter que des lettres (a → z et A → Z), des chiffres (0 → 9) et le caractère _ (*underscore*). Les **autres symboles sont interdits**.
- Python est **sensible à la casse**, c'est-à-dire que les majuscules et minuscules sont distinguées (exemple : `livre`, `Livre` et `LIVRE` sont des variables différentes).
- Il est vivement recommandé de nommer les variables en minuscule, y compris la première lettre et de commencer les mots suivants avec une majuscule pour plus de

lisibilité (exemple : `listeDeProduits`). **Dans ce cours, cette convention de nommage sera à appliquer.**

Affectation de variables

Nous allons aborder maintenant comment affecter une valeur à une variable :

```
nombre=25
texteBienvenue="Bonjour à tous !"
pi=3.14159
```

Avec l'exemple ci-dessus, nous avons créé trois variables :

- La variable nommée `nombre` contenant l'entier 25.
- La variable nommée `texteBienvenue` contenant le texte "Bonjour à tous !".
- La variable nommée `pi` contenant la valeur approchée de π à savoir le nombre réel 3,14159.

Chacune de ces lignes effectue les opérations suivantes :

1. Allouer un **espace mémoire** pour cette variable.
2. Affecter un **nom de variable** à cet espace via un **pointeur**.
3. Définir le type de cette variable (entier, texte …).
4. Mémoriser la valeur dans l'espace affecté.

Après l'exécution des lignes présentées, voici un extrait de la mémoire vive occupée par notre programme où on y retrouve deux espaces : l'**espace de noms** et l'**espace de valeurs**, liés entre eux par un **pointeur**.

`nombre`	→ 25
`texteBienvenue`	→ Bonjour à tous !
`pi`	→ 3.14159

Il est cependant possible d'effectuer des **affectations multiples**. Un premier exemple consiste à affecter une même valeur pour plusieurs variables :

```
temperatureDijon = temperatureRouen = 15.3
```

On peut également regrouper l'affectation de valeurs à des variables avec les **affectations parallèles** :

```
temperatureTroyes, temperatureAuxerre = 17, 14.2
```

Dans l'exemple ci-dessus, les variables `temperatureTroyes` et `temperatureAuxerre` prendront respectivement les valeurs 17 et 14,2.

Il est enfin à noter qu'une variable peut contenir le **résultat d'une opération** :

```
longueurRectangle = 25
largeurRectangle = 12
```

```
perimetreRectangle = (longueurRectangle + largeurRectangle) * 2
```

La partie gauche d'un signe égal doit toujours être un nom de variable et non une expression. Par exemple, l'instruction `ageJulie + 4 = ageCharles` est **invalide**.

Il est cependant très répandu d'avoir ce type d'expression tout à fait impossible en mathématiques : `ageMartin = ageMartin + 1`. Cela permet d'ajouter 1 à la valeur de la variable `ageMartin`. On dit alors qu'on **incrémente** `ageMartin`. Vous pouvez enfin **réaffecter** une valeur à une variable à savoir écraser le contenu d'une variable par une autre valeur.

Afficher la valeur d'une variable

Pour afficher le contenu d'une variable, nous allons utiliser la fonction `print()` :

```
>>> print(nombre)
25
>>> print(texteBienvenue)
Bonjour à tous !
```

La composition d'instructions

Après avoir vu quelques opérations de base, nous pouvons d'ores et déjà les combiner ensemble pour former des instructions plus complexes. En voici un exemple :

```
masseTomatesEnGrammes, masseCurryEnKg = 3600, 0.001
print("Vous avez acheté",masseTomatesEnGrammes/1000,"kg de tomates et
",masseCurryEnKg*1000,"grammes de curry. ")
```

Aide

Pour obtenir de l'aide sur l'utilisation d'une fonction ou d'un module, tapez `help(fonction)` en remplaçant `fonction` par la fonction ou le module recherché.

```
>>> help(print)
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Exercices

1. Cochez les instructions valides :

   - ☐ `tailleMarie = tailleLea = 1.78`

- ☐ `tailleMarie + 0.7 = tailleLucie`
- ☐ `tailleThomas = tailleLucie - 0.7`
- ☐ `tailleMartin = 1,52`
- ☐ `taillePaul = 1.41, 1.63`
- ☐ `tailleAlain = tailleAlain + 0.8`

- • Écrivez les instructions nécessaires pour affecter l'entier 17 à la variable `quantiteCrayons`, le nombre réel 14,2 à la variable `volumeEau` et le texte "Temps nuageux" à la variable `meteoParis` avec et sans les affectations multiples.

- Écrivez les instructions permettant de calculer l'aire d'un disque. La formule est $A = \pi \times R^2$ avec A l'aire et R le rayon du disque valant ici 5 cm. Le résultat doit être affiché sous la forme "L'aire du disque est de XXX cm²".

On prendra ici $\pi = 3{,}14159$ et on retournera une valeur non arrondie.

Décrivez précisément ce que fait le programme suivant :

```
longueurPiece = 17                    # En mètres
largeurPiece = 9                             # En mètres
longueurCarrelage = 0.3               # En mètres
airePiece = longueurPiece*largeurPiece
aireCarreauCarrelage = longueurCarrelage**2
nombreCarreauxCarrelage = airePiece/aireCarreauCarrelage
print("Il vous faudra",nombreCarreauxCarrelage,"carreaux pour recouvrir une
pièce de",airePiece,"m2." )
```

Chap 2 : Le flux d'instructions

Un programme consiste en une suite d'instructions structurées et exécutées par un ordinateur. Cette définition est le cœur de ce chapitre. Nous allons ici écrire nos premiers scripts Python en utilisant les bases que nous avons vues dans le chapitre précédant mais en modifiant le déroulement des opérations en introduisant l'**exécution conditionnelle**.

Les scripts

Nous ne travaillerons plus à partir de l'interpréteur, mais nous éditerons des scripts que nous pourrons éditer et sauvegarder. Un script Python consiste en un fichier texte ayant pour extension **.py** et contenant du code Python. En voici un exemple :

```
#!/usr/bin/env python3

longueurPiece = 17             # En mètres
largeurPiece = 9               # En mètres
longueurCarrelage = 0.3                # En mètres
airePiece = longueurPiece*largeurPiece
aireCarreauCarrelage = longueurCarrelage**2
nombreCarreauxCarrelage = airePiece/aireCarreauCarrelage
print("Il vous faudra",nombreCarreauxCarrelage,"carreaux pour recouvrir une
pièce de",airePiece,"m2." )
```

Vous aurez remarqué qu'il s'agit des mêmes instructions que nous avons déjà rencontrées. Notez par ailleurs l'ajout de la ligne `#!/usr/bin/env python3` : cette ligne indique aux systèmes d'exploitation de type Unix qu'il s'agit d'un script Python et non d'un autre type de script (tels que Ruby, bash …) et lui fournit l'adresse de l'interpréteur avec lequel lire le script (ici `/usr/bin/env python3`). Il s'agit du **shebang**. Dans ce cours, nous **imposons** la présence de ce shebang.

Pour créer un nouveau script, utilisez un éditeur de texte tel que Notepad++, Atom ou plus simplement bloc-notes. Pour taper votre code source et enregistrez-le dans un fichier ayant pour extension **.py**.

Pour exécuter un script Python, entrez la commande suivante dans votre shell Linux :

```
python3 monScript.py
```
Pour rendre votre code exécutable directement, effectuez ceci :
```
chmod +x monScript.py
./monScript.py
```

Il est possible d'interrompre manuellement l'exécution d'un programme avec la combinaison des touches `CTRL` et `C`.

Les types d'erreurs

Malgré tout le soin que vous porterez à programmer, des erreurs se glisseront dans vos programmes. Il en existe trois types principaux :

Erreur de syntaxe

Dans un programme, la syntaxe doit être parfaitement correcte : la moindre erreur dans le nom des fonctions ou d'indentation provoquera un arrêt de fonctionnement ou une exécution erratique.

Erreur sémantique
: Le programme fonctionne, cependant, vous n'obtenez pas le résultat souhaité. Dans ce cas, c'est que le séquencement des instructions de votre programme n'est pas correct.

Erreur d'exécution
: Un élément extérieur vient perturber le fonctionnement normal de votre programme. Ces erreurs, également appelées **exceptions**, sont dues à des circonstances particulières comme par exemple votre programme doit lire un fichier mais il est absent ou l'utilisateur n'a pas entré la valeur attendue.

Les commentaires

Lors de l'écriture d'un script, il est **vivement recommandé** de commenter ce que fait le programme. Pour ajouter une ligne de commentaires, celle-ci doit commencer par un #. Cela indique à l'interpréteur d'ignorer la ligne à partir de ce caractère jusqu'à la fin de la ligne.

```
a=4                     # Voici un commentaire.
# Un commentaire sur toute la ligne.
# Exemple :
tailleTourEiffel = 324 # Exprimé en mètres jusqu'à l'antenne.
```

Dans ce cours, il sera **exigé** de commenter les opérations complexes pour en définir simplement leur fonctionnement.

Il est tout aussi **recommandé** de renseigner une **documentation** sur le script ou la fonction que vous développez pour renseigner comment elles fonctionnent et comment s'interfacer avec. La documentation doit être saisie **au début**, **précédée** et **suivie** par ceci : `"""` (trois fois les double-guillemets). En voici un exemple :

```
#!/usr/bin/env python3
"""Calculatrice pour la pose de carrelage. """

longueurPiece = 17             # En mètres
largeurPiece = 9               # En mètres
longueurCarrelage = 0.3                # En mètres
airePiece = longueurPiece*largeurPiece
aireCarreauCarrelage = longueurCarrelage**2
nombreCarreauxCarrelage = airePiece/aireCarreauCarrelage
print("Il vous faudra",nombreCarreauxCarrelage,"carreaux pour recouvrir une
pièce de",airePiece,"m2." )
```

Cette documentation sera accessible en tapant `help(fonction)`.

Séquences d'instructions

Sauf cas spéciaux, les instructions sont exécutées les unes après les autres. Cette caractéristique peut parfois jouer des tours si l'ordre des instructions n'est pas correct. Voici un exemple dans lequel l'ordre des instructions est crucial. Dans ce cas, intervertir les lignes 2 et 3 donne un résultat différent :

```
a, b = 2, 8
a=b
b=a
print(a,b)
```

Interaction utilisateur

Nous allons rendre nos programmes plus interactifs. Pour cela, nous allons demander à l'utilisateur final de saisir des données que nous utiliserons dans nos programmes. Nous allons donc utiliser la fonction `input()`. Si vous saisissez une chaîne de caractères entre les parenthèses (on parle de "**passer en argument**" la chaîne de caractères), celle-ci précédera la saisie utilisateur mais ceci est facultatif. La version 3 de Python détecte automatiquement le typage de la saisie utilisateur. Voici un exemple d'utilisation de cette fonction :

```
>>> nom = input("Entrez votre nom : ")
Entrez votre nom : Michel
>>> print("Bonjour",nom)
Bonjour Michel
```

Le type de variable retourné est une **chaîne de caractères**. Pour la convertir en réel (dit flottant), utilisez `float(variable)`. Pour la convertir en entier, utilisez `int(variable)`.

Les conditions

Nous allons introduire un des cas particuliers à l'ordre d'exécution des instructions : **les conditions**. Cela permet d'exécuter une portion de code si une certaine condition est remplie, et une autre portion si cette même condition n'est pas remplie. Voici un exemple permettant de clarifier cela :

```
nombre = int(input("Entrez un nombre : "))
inferieurA5 = False
if nombre < 5:
        print("Le nombre est inférieur à 5. ")
        inferieurA5 = True
else:
        print("Le nombre est supérieur ou égal à 5. ")
```

Dans le cas précédent, les lignes 4 et 5 seront exécutées si l'utilisateur saisit un nombre inférieur à 5. La ligne 7 est exécutée si le nombre ne remplit pas cette condition. De manière plus générale, voici la syntaxe d'une condition :

```
if condition 1:
        début bloc code si la condition 1 est vraie
        ...
        fin bloc code si la condition 1 est vraie
elif condition 2:
        début bloc code si la condition 1 est fausse et la condition 2 est
vraie
        ...
        fin bloc code si la condition 1 est fausse et la condition 2 est
vraie
else:
        début bloc code si les conditions 1 et 2 sont fausses
```

```
...
fin bloc code si les conditions 1 et 2 sont fausses
```

Pour cela, on utilise les instructions `if` (**si** en anglais) auxquelles on juxtapose la **condition**, et terminées par "**:**". Cette condition est à remplir pour exécuter le bloc de code délimité par une **tabulation** en début de chaque ligne.

On peut, ceci est facultatif, mettre une instruction `elif` (**sinon si** en anglais) permettant de vérifier une seconde condition si la première est fausse. On peut chaîner autant d'instructions `elif` que nécessaire.

Enfin, l'instruction `else` (**sinon** en anglais), elle aussi facultative, exécute un bloc de code si les conditions énumérées avec `if` et `elif` n'ont pas été remplies.

Les opérateurs de comparaison

Plusieurs opérateurs de comparaison sont disponibles.

Comparateur	Syntaxe	Types de variables
a égal à b	`a == b`	Tous
a différent de b	`a != b`	Tous
a supérieur à b	`a > b`	Numériques
a supérieur ou égal à b	`a >= b`	Numériques
a inférieur à b	`a < b`	Numériques
a inférieur ou égal à b	`a <= b`	Numériques
a [pas] dans b	`a [not] in b`	a : Tous, b : Liste ou texte

Les blocs d'instructions

Comme abordés dans la partie précédente, nous parlerons des blocs d'instructions. Un bloc d'instructions est un ensemble d'instructions (pléonasme !) associé à une **ligne d'en-tête** (nous avons déjà vu `if`, `elif` ou `else` mais on retrouve également `while`, `for`, `def` …). Ces lignes d'en-tête se terminent par un double point (:).

Un bloc de code est **délimité par son indentation**. Les lignes d'instructions d'un même bloc de code sont indentées de manière identique (possèdent le même nombre de tabulations au début de celles-ci). Les blocs peuvent être **imbriqués** comme l'illustre le schéma suivant :

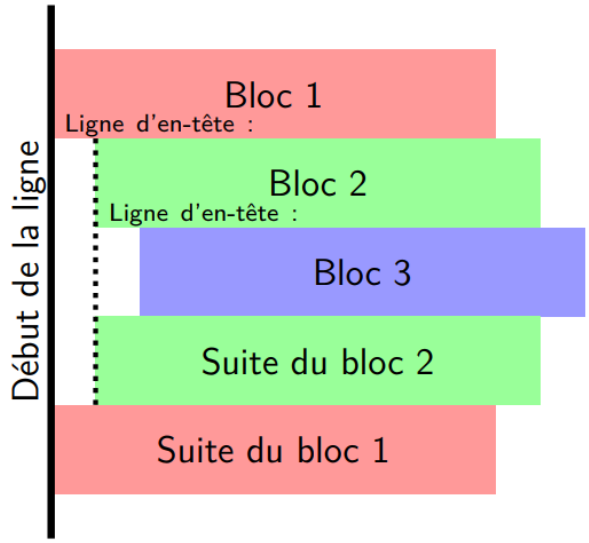


FIGURE 2.1 – Imbrication des blocs

Début de la ligne
Bloc 1 Ligne d'en-tête: Bloc 2 Ligne d'en-tête: Bloc 3 Suite du bloc 2 Suite du bloc 1

L'exemple suivant met en application l'imbrication de blocs d'instructions :

```
nombre = 17
if nombre > 10:
        print("Le nombre est supérieur à 10")
        if nombre < 20:
                print("Le nombre est inférieur à 20")
        else:
                print("Le nombre est supérieur à 20")
else:
        print("Le nombre est inférieur à 10")
```

Exercices

1. Écrivez un programme demandant l'âge de l'utilisateur et affichant si celui-ci est majeur ou mineur.
2. Écrivez un programme permettant, à partir d'un montant hors taxes saisi par l'utilisateur,
   de calculer le montant de la TVA (20% du montant hors taxes) et du montant TTC (montant hors taxes auquel on ajoute le montant de la TVA) et donnez le détail du calcul.
3. Écrivez un programme vérifiant si dans un texte saisi par l'utilisateur, celui-ci contient le mot "fraise" ou le mot "pêche".

Exemple :

```
Entrez un texte : pêche
```

```
Le texte contient le mot pêche.
Entrez un texte : fraise
Le texte contient le mot fraise.
Entrez un texte : prune
Le texte ne contient ni le mot pêche ou fraise.
```

Chap 3 : Factoriser le code

Plus vos programmes se complexifieront, plus le code source écrit sera volumineux. Pour y remédier, nous allons introduire deux nouveaux concepts : les **boucles** et les **fonctions**. Nous aborderons deux formes de boucles qui répondent à deux modes de fonctionnement différents. Nous poursuivrons ensuite sur les fonctions fournies dans Python 3 et enfin sur comment écrire ces fonctions soi-même. À partir de ce chapitre, la présence du shebang, de la documentation dans le code, les noms de variables explicites et de commentaires sont **obligatoires**.

Boucles "Tant que"

Les boucles ont pour but de répéter un ensemble d'instructions tant qu'une certaine condition est remplie. Dès que cette condition n'est plus remplie, l'interpréteur rompt la boucle et continue l'exécution du script. Il existe deux types de boucles en Python : la boucle **"tant que"** et la boucle **"pour"** que nous aborderons plus tard.

La boucle **"tant que"** (**"while"** en anglais) permet d'exécuter une portion de code tant que la condition fournie est vraie. On utilise le mot **while**. Voici un exemple trivial permettant de mettre en œuvre cette nouvelle fonction :

```
compteur = 1
while compteur <= 10:
        print(compteur)
        compteur = compteur + 1
```

L'exemple ci-dessus affichera les nombres de 1 à 10 inclus à raison de un par ligne.

La boucle n'est pas exécutée si au début la condition n'est pas remplie. Les variables constituant la condition doivent exister avant l'exécution de la boucle.

Faites attention à toujours avoir un facteur modifiant la condition de la boucle. En effet, si la condition est toujours remplie, la boucle s'exécutera pour toujours. On parle alors de **boucle infinie**. Voici un exemple de boucle infinie :

```
compteur = 1
while compteur <= 10:
        print(compteur)
```

L'instruction `break`

Il peut être nécessaire, lorsque l'on ne connaît pas à l'avance sous quelle condition on va quitter une boucle. C'est notamment le cas des communications réseau ou des lectures de fichiers. Pour arrêter immédiatement l'exécution d'une boucle à tout moment de son itération, on utilise l'instruction `break`.

Les fonctions

Une fonction est une séquence d'instructions permettant d'effectuer une tâche précise. Nous allons ici réduire le volume de notre code en regroupant les instructions en tâches et créer des fonctions associées. Nous allons voir également que Python fournit beaucoup de fonctions dont nous avons déjà vu quelques unes.

Les fonctions fournies par Python

Nous avons déjà vu dans les chapitres précédents les fonctions **print()**, **help()** et **input()**. Nous allons voir certaines fonctions disponibles utiles.

Trans-typage

Lors de la saisie de données par l'utilisateur, la valeur retournée est une chaîne de caractères. Pour effectuer des opérations mathématiques, il est nécessaire de la convertir en valeur entière ou décimale. Pour ce faire, nous utiliserons les fonctions **int()** pour convertir en entier et **float()** pour convertir en décimal. On peux également tronquer un nombre décimal avec la fonction **int()** :

```
saisieUtilisateur = "75.9"
valeurFlottante = float(saisieUtilisateur)
valeurEntiere = int(valeurFlottante)
```

À l'inverse, il est possible de convertir une donnée en chaîne de caractères avec la fonction **str()**.

Comparer des valeurs

Il est possible de récupérer la valeur minimale et maximale d'une série de valeurs séparées par des virgules avec respectivement les fonctions **min()** et **max()**.

```
valeurMaximale = max(4, 17.8, 12)
valeurMiminale = min(4, 17.8, 12)
```

Ces fonctions sont notamment utiles lorsque l'on manipule des listes, que nous verrons plus tard.

Importer des modules

Python 3 est fourni avec un ensemble de fonctions disponibles nativement. Cependant, il peut être nécessaire d'importer un module afin d'utiliser une fonction précise. Pour illustrer comment importer un module Python et utiliser les fonctions qui y sont référencées, nous allons utiliser la fonction `randint` du module `random` permettant de tirer un nombre entier au hasard.

Il est d'usage et fortement recommandé d'importer les modules en début de programme, sous le shebang. Nous utiliserons cette convention tout au long de ce cours.

Importer des fonctions d'un module

Si vous utilisez un faible nombre de fonctions d'un même module, il est recommandé d'importer que ces dernières et ainsi faire l'économie des fonctions inutiles du module :

```
from random import randint, shuffle   # Importe les fonctions randint et
shuffle (inutile dans cet exemple) du module random
aleatoire = randint(1,10)             # Nombre aléatoire entre 1 et 10
inclus
```

Importer un module complet

Si vous utilisez une grande partie des fonctions d'un module, il est très fastidieux d'importer les fonctions une à une. Dans ce cas, on importe le module complet :

```
import random                         # Importe le module random
aleatoire = random.randint(1,10)      # Nombre aléatoire entre 1 et 10
inclus (fonction du module random)
```

Importer un autre fichier Python

Vous pouvez également importer un autre script Python comme module. Pour cela, il faut que votre module soit écrit dans le même dossier que votre script. Pour l'importer, utilisez l'une des deux méthodes ci-dessus en renseignant le nom du fichier module sans son extension.

ATTENTION : Lorsque vous importez un module dans vos programmes, le code non présent dans des fonctions ou classes s'exécute. Par exemple :

Fichier monmodule.py
```
def maFonction():
        print("Ceci est ma fonction")
print("Bonjour le monde !")
```
Fichier monscript.py
```
import monmodule
print("Le ciel est bleu")
```

Dans cet exemple, la ligne 3 du fichier `monmodule.py` sera exécutée lors de l'import.

Créer des fonctions

Nous allons aborder maintenant comment créer nous-mêmes des fonctions. Pour cela, nous utiliserons l'instruction **def**. Une fonction peut prendre des **arguments** en entrée et **retourner** une valeur en fin avec l'instruction **return**, tout cela est facultatif. Voici un exemple que nous détaillerons ensuite :

```
def addition(a, b):
        resultat = a+b
        return(resultat)
resultatAddition = addition(4, 7)
print(resultatAddition)
```

La première ligne introduit notre nouvelle fonction nommée addition et nécessitant deux arguments, les variables a et b. **Ces variables, ainsi que celles déclarées au sein de cette fonction, ne sont valables que dans cette fonction.** Le corps de la fonction, composé des lignes 2 et 3, indique d'effectuer une opération sur les variables et de retourner la valeur de la variable `resultat`. **L'exécution de la fonction s'arrête lors du `return`, même si d'autres instructions la suivent.**

Une fois la fonction définie, nous l'utilisons à la ligne 4 en lui fournissant comme arguments les valeurs a=4 et b=7. On récupère le résultat de cette fonction dans la variable `resultatAddition`. On affiche le contenu de la variable `resultatAddition` à la ligne 5. **Si le résultat d'une fonction n'est pas récupéré, il est perdu.**

Il est possible d'utiliser plusieurs fois une même fonction :

```
def addition(a, b):
        return(a+b)
resultatAddition = addition(4, 7)
print(resultatAddition)
resultatAddition = addition(8, 4.12)
print(resultatAddition)
```

Lorsqu'une fonction est créée, elle prévaut sur celle fournie par défaut par Python. Si vous appelez une de vos fonctions `print`, celle-ci sera appelée en lieu et place de celle fournie de base.

Lors de l'utilisation d'une fonction, tous les arguments sont obligatoires. Vous pouvez rendre un argument facultatif en lui fournissant une **valeur par défaut** :

```
def tableAddition(valeur, fin=10):
        compteur = 1
        while compteur <= fin: # Par défaut, la valeur de fin est à 10.
                print(compteur,"+",valeur,"=",compteur+valeur)
                compteur += 1  # Équivalent à compteur = compteur + 1
tableAddition(4)        # La variable fin aura pour valeur 10
tableAddition(6, 15)    # La variable fin aura pour valeur 15
```

Vous pouvez fournir les arguments dans le désordre lors de l'appel d'une fonction en nommant vos variables :

```
def addition(a, b):
```

```
        return(a+b)
resultatAddition = addition(b=2, a=9)
print(resultatAddition)
```

Ici, la fonction aura pour arguments a=9 et b=2. Ce type d'appel peut être utile pour des fonctions ayant un grand nombre d'arguments pour plus de clarté.

Enfin, il est également possible de retourner plusieurs valeurs à la fin d'une fonction :

```
def valeursExtremes(valeurs):
        return(min(valeurs),max(valeurs))
valeurMinimale, valeurMaximale = valeursExtremes([14,96,57,10,0.7])
```

Exercices

1. Écrivez un programme permettant d'effectuer les opérations de base (+, +, ×, ÷) en mathématiques en créant des fonctions.

   Exemple :

   ```
   Entrez le premier nombre : 17
   Entrez le second nombre : 14
   Entrez l'opération (1=+, 2=-, 3=X, 4=:) : 2
   Le résultat est 3
   ```

- Créez un programme tirant un nombre au hasard entre 1 et 20 et demandant à l'utilisateur de deviner ce nombre en lui indiquant si sa proposition est supérieure ou inférieure au nombre tiré.

Exemple :

```
Devinez le nombre que j'ai en tête !
Entrez un nombre : 7
C'est plus
Entrez un nombre : 13
C'est moins
Entrez un nombre : 10
Gagné !
```

Écrivez un programme permettant de demander à l'utilisateur la réponse à la multiplication de deux nombres tirés aléatoirement. Votre programme lui posera 15 questions et calculera un score noté sur 20.

Exemple :

```
Combien font 1X1 :
2
Dommage, la réponse était 1
Combien font 4X3 :
74
Dommage, la réponse était 12
Combien font 10X10 :
100
Bien joué !
```

```
…
C'est terminé, votre note est de 15/20.
```

Chap 4 : Les séquences

Les séquences en Python sont des structures de données composées d'entités plus petites. Dans cette catégorie, on retrouve les **chaînes de caractères**, les **listes** et les **dictionnaires**. Ces types de données ont en commun des fonctions permettant d'avoir des informations sur elles ou de les modifier, ainsi qu'une boucle parcourant un à un les éléments de celles-ci. Nous allons étudier en détail ces structures, les fonctions, ainsi que la boucle "Pour" dans ce chapitre.

Les chaînes de caractères

Nous avons déjà abordé précédemment les chaînes de caractères mais sans entrer dans le détail. Il est tout d'abord primordial d'avoir à l'esprit que les chaînes de caractères sont composées de caractères accessibles par un **indice**. Le schéma suivant permet d'illustrer les caractères associés à leurs indices.

```
chaineDeCaracteres= M o n t a g n e
                    ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
                    0 1 2 3 4 5 6 7
```

On peut donc accéder aux caractères un à un à partir de leurs indices et n'utiliser qu'une partie de la chaîne en demandant un fragment de celle-ci :

```
>>> chaineDeCaracteres = "Montagne"
>>> print(chaineDeCaracteres[0])
M              # Premier caractere
>>> print(chaineDeCaracteres[3])
t              # Quatrieme caractere
```

```
>>> print(chaineDeCaracteres[-1])
e               # Dernier caractere
>>> print(chaineDeCaracteres[-2])
n               # Avant-dernier caractere
>>> print(chaineDeCaracteres[-3])
g               # Avant-avant-dernier caractère
>>> print(chaineDeCaracteres[2:5])
nta             # Tranche du troisième au cinquième caractère
>>> print(chaineDeCaracteres[:4])
Mont            # Tranche du premier au quatrième caractère
>>> print(chaineDeCaracteres[6:])
ne              # Tranche du septième au dernier caractère
>>> print(chaineDeCaracteres[7:2:-1])
engat           # Tranche du huitième au troisième caractère dans le sens
inverse
>>> print(chaineDeCaracteres[::-1])
engatnoM        # La chaîne complète dans le sens inverse
>>> print(chaineDeCaracteres[::2])
Mnan            # Les lettres d'indice pair
>>> print(chaineDeCaracteres[1::2])
otge            # Les lettres d'indice impair
```

Comme nous l'avons vu dans l'exemple précédent, on peut accéder aux caractères d'une chaîne en entrant le nom de la variable suivi de l'indice entre crochets. Un indice négatif permet d'accéder aux caractères à partir de la fin.

La syntaxe générale est : `variable[indiceDebut` (par défaut 0)`:indiceFin` (par défaut la fin de la chaîne)`:pas` (par défaut 1)`]` avec `indiceDebut` inclus, `indiceFin` exclu et `pas` le pas.

Si une valeur n'est pas renseignée, sa valeur par défaut est appliquée. Si une seule valeur est entrée entre crochets, seulement le caractère à l'indice du début est retourné (exemple lignes 2 à 10).

Enfin, il est possible de **concaténer** des chaînes de caractères **uniquement** à l'aide du symbole + :

```
>>> prenom = "Arnaud"
>>> texte = "Bonjour " + prenom + ", comment vas-tu ?"
>>> print(texte)
Bonjour Arnaud, comment vas-tu ?
```

Modifier la casse d'une chaîne de caractères

La casse désigne le fait de distinguer les lettres majuscules des lettres minuscules. Python possède les méthodes `lower()`, `upper()`, `title()`, `capitalize()` et `swapcase()` permettant de modifier la casse d'une chaîne de caractères :

```
>>> texte = "Écrit par Antoine de Saint-Exupéry"
>>> print(texte.lower())              # Tout en minuscule
écrit par antoine de saint-exupéry
>>> print(texte.upper())              # Tout en majuscule
ÉCRIT PAR ANTOINE DE SAINT-EXUPÉRY
>>> print(texte.title())              # Majuscule à chaque mot
Écrit Par Antoine De Saint-Exupéry
>>> print(texte.capitalize())         # Majuscule en début de phrase
```

```
Écrit par antoine de saint-exupéry
>>> print(texte.swapcase())           # Inverse la casse
éCRIT PAR aNTOINE DE sAINT-eXUPÉRY
```

Compter les occurrences dans une chaîne ou une liste

La méthode `count(sousChaine)` permet de compter le nombre d'occurrences de la sous-chaîne dans la chaîne ou liste :

```
>>> texte = "Écrit par Antoine de Saint-Exupéry"
>>> texte.count("a")   # Sensible à la casse
2
>>> texte.lower().count("a")
3
```

Rechercher et remplacer les éléments d'une chaîne

La méthode `relace(ancien, nouveau)` permet de remplacer l'élément `ancien` par `nouveau`.

```
>>> texte = "Écrit par Antoine de Saint-Exupéry"
>>> print(texte.replace("Écrit", "Rédigé"))
Rédigé par Antoine de Saint-Exupéry
```

Les listes et les tuples

Une liste et un tuple Python sont un ensemble **ordonné** d'éléments de tous types. Elles peuvent contenir en leur sein des chaînes de caractères, des nombres, des autres listes, des objets … Elles peuvent contenir des éléments de plusieurs types à la fois.

La différence entre une liste et un tuple est qu'une liste est modifiable et un tuple, non. Chaque élément est séparé par une virgule. Voici ci-dessous la syntaxe pour déclarer une liste et un tuple :

```
exempleListe = [27, 24.8, "Bonjour"]
exempleTuple = (27, 24.8, "Bonjour")
listeDansUneListe = [[1,2,3], [4,5,6], [7,8,9]]
```

À l'instar des chaînes de caractères, on peut accéder aux éléments d'une liste ou d'un tuple par son indice entre crochets :

```
>>> exempleListe = [27, 24.8, "Bonjour"]
>>> listeDansUneListe = [[1,2,3], [4,5,6], [7,8,9]]
>>> exempleListe[0]
27
>>> exempleListe[1:]
[24.8, 'Bonjour']
>>> listeDansUneListe[0]
[1, 2, 3]
>>> listeDansUneListe[0][1]
2
```

Il est possible de modifier, d'ajouter ou de supprimer une valeur d'une liste **et non d'un tuple**. Nous allons aborder toutes ces opérations.

Ajouter un élément en fin de liste

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

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fournitures.append("ciseaux")
>>> print(fournitures)
['cahier', 'crayon', 'stylo', 'trousse', 'gomme', 'ciseaux']
```

Modifier un élément

La modification d'un élément se fait en réaffectant la nouvelle valeur à la place de l'ancienne :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fournitures[1] = "équerre"
>>> print(fournitures)
['cahier', 'équerre', 'stylo', 'trousse', 'gomme']
>>> fournitures[3] = ["trombones","calque"]
>>> print(fournitures)
['cahier', 'équerre', 'stylo', ['trombones', 'calque'], 'gomme']
```

Ajouter un élément au cœur de la liste

Ajouter un élément se fait en modifiant une tranche de la liste dont le début et la fin de la tranche sont identiques :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fournitures[2:2] = ["cartable"]   # Doit être dans une liste ou un
tuple.
>>> print(fournitures)
['cahier', 'crayon', 'cartable', 'stylo', 'trousse', 'gomme']
>>> fournitures[4:4] = ["règle","feuilles"]
>>> print(fournitures)
['cahier', 'crayon', 'cartable', 'stylo', 'règle', 'feuilles', 'trousse',
'gomme']
```

Supprimer un élément

Il existe deux méthodes pour supprimer un élément d'une liste : `remove(element)` et `pop(indice)`.

La méthode `remove`

Cette méthode permet de supprimer la première occurrence de l'élément passé en argument :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme",
"stylo"]
>>> fournitures.remove("stylo")
>>> print(fournitures)
['cahier', 'crayon', 'trousse', 'gomme', 'stylo']
```

La méthode `pop`

Cette méthode permet de supprimer un élément par son indice et retourne l'élément supprimé :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> element = fournitures.pop(3)
>>> print(fournitures)
['cahier', 'crayon', 'stylo', 'gomme']
>>> print(element)
trousse
```

Calculer la taille d'une séquence

La fonction `len(sequence)` permet de retourner le nombre d'éléments contenus dans une séquence :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> print(len(fournitures))
5
```

Diviser une chaîne en liste

On peut séparer une chaîne de caractères en liste en utilisant la méthode `split(separateur)`. Cette méthode utilise en argument une chaîne délimitant chaque élément :

```
>>> listeFournitures = "cahier;crayon;stylo;trousse;gomme"
>>> fournitures = listeFournitures.split(";")
>>> print(fournitures)
['cahier', 'crayon', 'stylo', 'trousse', 'gomme']
```

Assembler une liste en chaîne

La méthode `separateur.join(liste)` permet de concaténer chaque élément de la liste séparé par un séparateur :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> listeFournitures = ";".join(fournitures)
>>> print(listeFournitures)
cahier;crayon;stylo;trousse;gomme
```

Trier une liste

La méthode `sort()` permet de trier dans l'ordre croissant une liste selon ses valeurs :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fournitures.sort()
>>> print(fournitures)
['cahier', 'crayon', 'gomme', 'stylo', 'trousse']
```

Inverser l'ordre d'une liste

La méthode `reverse()` permet d'inverser l'ordre des valeurs d'une liste :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fournitures.reverse()
>>> print(fournitures)
['gomme', 'trousse', 'stylo', 'crayon', 'cahier']
```

Mélanger une liste

La méthode `shuffle()` du module `random` permet de mélanger aléatoirement les valeurs d'une liste :

```
>>> from random import shuffle
>>> nombres = [1,2,3,4,5,6,7,8,9]
>>> shuffle(nombres)
>>> print(nombres)
[7, 9, 3, 1, 8, 4, 6, 5, 2]
>>> shuffle(nombres)
>>> print(nombres)
[2, 1, 9, 4, 6, 8, 5, 3, 7]
```

Trouver l'index d'un élément

La méthode `index(element)` retourne l'indice du premier élément passé en argument :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme",
"stylo"]
>>> fournitures.index("stylo")
2
```

Copie de liste

Nous allons étudier la copie de liste. On peut instinctivement tenter cette opération avec la commande suivante mais sans succès :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> copieFournitures = fournitures
>>> fournitures.remove("crayon")
>>> print(fournitures)
['cahier', 'stylo', 'trousse', 'gomme']
>>> print(copieFournitures)
['cahier', 'stylo', 'trousse', 'gomme']
```

Lors de la ligne 2, Python crée un alias à la nouvelle liste et n'effectue pas de copie. Chaque opération apportée sur chaque variable affectera la liste qui est accessible par ses alias. Ainsi, pour effectuer une véritable copie d'une liste, il est nécessaire d'utiliser la fonction `deepcopy` du module `copy` :

```
>>> from copy import deepcopy
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> copieFournitures = deepcopy(fournitures)
>>> fournitures.remove("crayon")
>>> print(fournitures)
['cahier', 'stylo', 'trousse', 'gomme']
>>> print(copieFournitures)
['cahier', 'crayon', 'stylo', 'trousse', 'gomme']
```

Création rapide d'une suite de nombre

Il peut être utile de générer une suite de nombres. Pour cela, il est recommandé d'utiliser la fonction `range(debut` (inclus, par défaut 0)`,fin` (exclu)`,pas` (par défaut 1)`)` .

```
>>> liste = [i for i in range(0,20,2)]
>>> print(liste)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Cette syntaxe peut être utilisée pour initialiser une liste avec des valeurs identiques :

```
>>> liste = [None for i in range(10)]
>>> print(liste)
[None, None, None, None, None, None, None, None, None, None]
```

Boucles "Pour"

Une boucle "**Pour**" ("**For**" en anglais) permet d'exécuter une portion de code pour chaque élément d'une liste en les affectant à une variable. Une fois que la liste est terminée, l'exécution normale du programme se poursuit.

Voici un exemple de l'utilisation de cette structure :

```
fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
for element in fournitures:
        print(element)
```

L'exemple ci-dessus affichera les éléments de la liste à raison de un par ligne. Il est possible d'utiliser la fonction `range` directement :

```
for compteur in range(15):
        print(compteur)
```

L'exemple ci-dessus affichera les nombres de 0 à 14 inclus à raison de un par ligne.

Récupérer l'élément et son indice

La fonction `enumerate(liste)` retourne l'indice et l'élément un à un :

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> for index,element in enumerate(fournitures):
...     print("Indice :  " + str(index) + " => " + element)
...
Indice :  0 => cahier
Indice :  1 => crayon
Indice :  2 => stylo
Indice :  3 => trousse
Indice :  4 => gomme
```

Parcourir deux listes en même temps

La fonction `zip(listes)` permet de parcourir plusieurs listes en même temps :

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = [7,8,9]
>>> for i in zip(a,b,c):
...     print(i)
...
```

```
(1, 4, 7)
(2, 5, 8)
(3, 6, 9)
```

Les dictionnaires

Un dictionnaire est un ensemble **désordonné** d'éléments de tous types. Chaque élément est identifié à l'aide d'une **clé**. Voici la syntaxe pour déclarer un dictionnaire. :

```
exempleDictionnaire = {"livre":74, 85:"tulipe", 74.1:"rose",
"coquelicot":False, "agrumes":["citron","orange","pamplemousse"]}
```

Le dictionnaire étant un ensemble désordonné, l'ordre de déclaration des éléments qui le composent n'a pas d'importance. Chaque élément est accessible par sa clé :

```
>>> exempleDictionnaire = {"livre":74, 85:"tulipe", 74.1:"rose",
"coquelicot":False, "agrumes":["citron","orange","pamplemousse"]}
>>> exempleDictionnaire[74.1]
'rose'
>>> exempleDictionnaire['coquelicot']
False
>>> exempleDictionnaire['coquelicot'] = 'Rouge'
>>> exempleDictionnaire['coquelicot']
'Rouge'
>>> exempleDictionnaire["agrumes"]
['citron', 'orange', 'pamplemousse']
>>> exempleDictionnaire["agrumes"][1]
'orange'
>>> exempleDictionnaire['livre'] += 1
>>> print(exempleDictionnaire['livre'])
75
```

On peut ajouter un élément dans un dictionnaire comme suit :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> quantiteFournitures["agrafes"] = 49
>>> quantiteFournitures["stylos"]["noir"] = 16
>>> print(quantiteFournitures)
{'stylos': {'bleu': 74, 'noir': 16, 'rouge': 41}, 'cahiers': 134, 'gommes':
85, 'agrafes': 49}
```

Supprimer un élément

À l'instar des listes, on utilise la méthode `pop(clé)` pour supprimer un élément par sa clé et retourner l'élément supprimé :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> element = quantiteFournitures.pop("gommes")
>>> print(quantiteFournitures)
{'stylos': {'bleu': 74, 'rouge': 41}, 'cahiers': 134}
>>> print(element)
85
```

Lister les clés d'un dictionnaire

La méthode `keys()` retourne la liste des clés du dictionnaire :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> cles = quantiteFournitures.keys()
>>> for fourniture in cles:
...     print(fourniture,"Quantité :",quantiteFournitures[fourniture])
...
stylos Quantité : {'bleu': 74, 'rouge': 41}
cahiers Quantité : 134
gommes Quantité : 85
>>> print(list(cles))
['stylos', 'cahiers', 'gommes']
```

Lister les valeurs d'un dictionnaire

La méthode `values()` retourne la liste des valeurs du dictionnaire :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> valeurs = quantiteFournitures.values()
>>> print(values)
dict_values([{'bleu': 74, 'rouge': 41}, 134, 85])
```

Copier un dictionnaire

La méthode `copy` permet de créer une copie indépendante d'un dictionnaire :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> inventaire = quantiteFournitures.copy()
>>> quantiteFournitures.pop("cahiers")
134
>>> print(quantiteFournitures)
{'stylos': {'bleu': 74, 'rouge': 41}, 'gommes': 85}
>>> print(inventaire)
{'stylos': {'bleu': 74, 'rouge': 41}, 'gommes': 85, 'cahiers': 134}
```

Parcourir un dictionnaire

On peut parcourir un dictionnaire par ses clés ou ses clés et ses valeurs avec la méthode `items()` :

```
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
>>> for cle in quantiteFournitures:
...     print(cle)
...
stylos
cahiers
gommes
>>> for cle,valeurs in quantiteFournitures.items():
...     print(cle,valeurs)
...
stylos {'bleu': 74, 'rouge': 41}
cahiers 134
```

Exercices

1. Écrivez un programme simulant le fonctionnement d'une banque en stockant le solde des comptes dans un dictionnaire. Il devra permettre le dépôt et le retrait de sommes d'argent.

   Voici les clients de la banque :

Client	Solde
Paul	154,74
Marie	418,45
Jean	96,20
Pauline	914,21

   Exemple :

```
  Banque

Client  Solde
---------------
Marie   418.45
Pauline 914.21
Jean    96.2
Paul    154.74

Choisissez une opération (D : Dépot - R : Retrait - Q : Quitter) : d
Entrez le nom du client sur lequel effectuer le dépot : Jean
Quel montant voulez-vous déposez ? 50
Opération effectuée.
  Banque

Client  Solde
---------------
Marie   418.45
Pauline 914.21
Jean    146.2
Paul    154.74

Choisissez une opération (D : Dépot - R : Retrait - Q : Quitter) : r
Entrez le nom du client sur lequel effectuer le retrait : Paul
Quel montant voulez-vous retirer ? 9000
Opération impossible : votre solde est insuffisant.
```

- Écrivez un programme de loterie en demandant à l'utilisateur de choisir 6 numéros entre 1 et 50 inclus et d'effectuer un tirage aléatoire sur les mêmes critères. Enfin, il devra vérifier le nombre de numéros gagnants.

Exemple :

```
Entrez le 1e numéro entre 1 et 50 inclus : 4
Entrez le 2e numéro entre 1 et 50 inclus : 17
Entrez le 3e numéro entre 1 et 50 inclus : 25
```

```
Entrez le 4e numéro entre 1 et 50 inclus : 55
Entrez le 4e numéro entre 1 et 50 inclus : 8
Entrez le 5e numéro entre 1 et 50 inclus : 22
Entrez le 6e numéro entre 1 et 50 inclus : 33
Votre choix : 4 - 17 - 25 - 8 - 22 - 33
Le tirage : 9 - 32 - 34 - 22 - 11 - 2
Vous avez 1 nombres gagnants.
```

- Écrivez un programme permettant de vérifier si un mot ou une phrase saisis par l'utilisateur est un palindrome, à savoir un mot lisible à la fois à l'endroit ou à l'envers tel que Serres, radar, rotor ou "Ésope reste ici et se repose".

Exemple :

```
Entrez un mot ou une phrase : laval
La phrase est un palindrome.
Entrez un mot ou une phrase : fenetre
La phrase n'est pas un palindrome.
```

- Écrivez un programme permettant de générer le calendrier d'une année non-bissextile dont le premier janvier tombe un samedi (telle que 2011). Les jours et les mois seront stockés dans des tuples. L'affichage final sera du type :

```
Samedi 1 janvier
Dimanche 2 janvier
…
Samedi 31 décembre
```

- Écrivez un programme permettant de calculer la quantité d'ingrédients de la recette ci-dessous en fonction du nombre de biscuits fourni par l'utilisateur. La liste d'ingrédients doit être stockée dans un dictionnaire.

Biscuits écossais (20 biscuits)

300g de farine • 75g de beurre • 75g de sucre roux • 1 œuf • 50 ml de lait • ½ sachet de levure chimique • 1 sachet de sucre vanillé.

Mélanger farine et levure, ajouter le beurre, le sucre et pétrir avec les doigts. Ajouter l'œuf battu et le lait. Bien mélanger. Fariner la boule de pâte, étaler la pâte sur ½ cm et découper des formes avec un emporte-pièce. Cuire 12 à 15 minutes à 190°C.

Exemple :

```
Entrez le nombre de biscuits à produire (par multiple de 20) : 80
  - Levure chimique : 2.0 sachet
  - Lait : 200.0ml
  - Oeuf : 4.0
  - Sucre vanillé : 4.0 sachet
  - Beurre : 300.0g
  - Farine : 1200g
  - Sucre roux : 300.0g
Entrez le nombre de biscuits à produire (par multiple de 20) : 17
  - Levure chimique : 0.42500000000000004 sachet
  - Farine : 255g
  - Sucre roux : 63.75g
```

```
- Lait : 42.5ml
- Beurre : 63.75g
- Oeuf : 0.8500000000000001
- Sucre vanillé : 0.8500000000000001 sachet
```

- Écrivez un programme affichant le mot le plus long d'une phrase entrée par l'utilisateur.

Exemple :

```
Entrez une phrase : Les sanglots longs des violons de l'automne blessent
mon coeur d'une langueur monotone
Le mot le plus long est L'AUTOMNE avec 9 lettres.
```

- Écrivez un programme permettant de trier une liste de nombres sans utiliser la méthode `sort()`. Réécrivez une fonction de tri de liste avec l'algorithme de tri à bulles qui consiste à comparer deux valeurs consécutives d'une liste et de les permuter quand elles sont mal triées et de répéter cela jusqu'à ce que la liste soit triée. Vous utiliserez les nombres tirés aléatoirement.

Exemple :

```
La liste avant le tri : 29 - 50 - 43 - 24 - 20 - 4 - 7 - 27 - 26 - 17 - 22
- 32 - 18 - 35 - 1
La liste après le tri : 1 - 4 - 7 - 17 - 18 - 20 - 22 - 24 - 26 - 27 - 29 -
32 - 35 - 43 - 50
```

Chap 5 : Manipuler les fichiers

Il peut être nécessaire de lire ou d'écrire des fichiers stockés sur l'ordinateur exécutant vos scripts. Consigner des données dans des fichiers permet de simplifier un programme en externalisant les données et peut être un moyen de s'interfacer avec d'autres programmes et systèmes ainsi qu'avec les utilisateurs. Nous utiliserons la fonction fournie par défaut `open()`. Avant tout, il est nécessaire de voir comment naviguer dans l'arborescence.

Navigation dans l'arborescence

En fonction du répertoire dans lequel est exécuté votre script, il peut être nécessaire de changer de répertoire de travail du script. Pour se faire, nous utiliserons la fonction `chdir(repertoire)` dans le module `os` pour changer de répertoire de travail. Nous utiliserons également la fonction `getcwd()` du même module. Il est possible de créer un dossier avec `mkdir(chemin)` :

```
>>> from os import getcwd, chdir, mkdir
>>> print(getcwd())
```

```
/home/antoine
>>> chdir('essais')
>>> print(getcwd())
/home/antoine/essais
>>> mkdir('test')
```

Ouvrir un fichier

Pour lire un fichier, il faut tout d'abord ouvrir un flux de lecture ou d'écriture de fichier avec la fonction `open(fichier,mode` (par défaut : 'rt')) avec `fichier` l'adresse du fichier à ouvrir et `mode`, le type de flux à ouvrir. Le mode est composé de deux lettres, les droits d'accès (`rwxa`) et l'encodage (`bf`). Voici le tableau détaillant les modes de flux de fichier.

Caractère	Action
`'r'`	Ouvrir en lecture seule (défaut)
`'w'`	Ouvrir en écriture. Écrase le fichier existant.
`'x'`	Ouvrir en écriture si et seulement si le fichier n'existe pas déjà.
`'a'`	Ouvrir en écriture. Ajoute au fichier existant.
`'b'`	Mode binaire.
`'t'`	Mode texte (défaut).

Pour fermer le flux de fichier avec la méthode `close()` sur la variable représentant le flux.

Il est important de fermer le flux une fois les opérations sur le fichier terminé.

Lire un fichier

Une fois le flux en lecture ouvert, on peut utiliser les méthodes `read()` qui retournent une chaîne de caractères contenant l'intégralité du fichier ou `readlines()` retournant une liste où chaque élément est une ligne du fichier.

```
>>> fichier = open("texte.txt",'rt')
>>> texte = fichier.read()
>>> print(texte)
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Pellentesque gravida erat ut lectus convallis auctor.
Fusce mollis sem id tellus auctor hendrerit.
>>> lignes = fichier.readlines()
>>> print(lignes)
['Lorem ipsum dolor sit amet, consectetur adipiscing elit.\n',
'Pellentesque gravida erat ut lectus convallis auctor.\n', 'Fusce mollis
sem id tellus auctor hendrerit.\n']
>>> fichier.close()
```

Chaque ligne est terminée par "`\n`" qui représente un retour à la ligne. Il s'agit d'un caractère spécial. Si vous écrivez ce caractère dans une chaîne de caractères, Python produira un retour à la ligne :

```
>>> texte = "Première ligne\nDeuxième ligne"
>>> print(texte)
Première ligne
Deuxième ligne
```

Écrire un fichier

On peut écrire un fichier si le flux est ouvert en écriture. Les trois flux possibles sont "`w`", "`x`" et "`a`". À l'instar de `read()` et `readlines()`, on utilisera `write(chaine)` pour écrire une chaîne de caractères et `writelines(lignes)` avec `lignes` une liste ou un tuple dont chaque élément est une ligne à écrire. **N'oubliez pas le caractère `\n` en fin de ligne pour revenir à la ligne.**

```
>>> fichier = open("texte.txt",'wt')
>>> fichier.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit.
\nPellentesque gravida erat ut lectus convallis auctor. \nFusce mollis sem
id tellus auctor hendrerit.")
>>> fichier.close()
>>> fichier = open("texte.txt",'wt')
>>> fichier.writelines(["Lorem ipsum dolor sit amet, consectetur adipiscing
elit. \n", "Pellentesque gravida erat ut lectus convallis auctor. \n",
"Fusce mollis sem id tellus auctor hendrerit."])
>>> fichier.close()
```

Formats de fichiers

Il existe différents formats standards de stockage de données. Il est recommandé de favoriser ces formats car il existe déjà des modules Python permettant de simplifier leur utilisation. De plus, ces formats sont adoptés par d'autres programmes avec lesquels vous serez peut-être amené à travailler.

Le format CSV

Le fichier *Comma-separated values* (CSV) est un format permettant de stocker des tableaux dans un fichier texte. Chaque ligne est représentée par une ligne de texte et chaque colonne est séparée par un **séparateur** (virgule, point-virgule …).

Les champs texte peuvent également être délimités par des guillemets. Lorsqu'un champ contient lui-même des guillemets, ils sont doublés afin de ne pas être considérés comme début ou fin du champ. Si un champ contient un signe pouvant être utilisé comme séparateur de colonne (virgule, point-virgule …) ou comme séparateur de ligne, les guillemets sont donc obligatoires afin que ce signe ne soit pas confondu avec un séparateur.

Voici des données présentées sous la forme d'un tableau et d'un fichier CSV :

```
Nom;Prénom;Age
"Dubois";"Marie";29
"Duval";"Julien ""Paul""";47
Jacquet;Bernard;51
Martin;"Lucie;Clara";14
```

Données sous la forme d'un fichier CSV

Nom	Prénom	Age
Dubois	Marie	29
Duval	Julien "Paul"	47
Jacquet	Bernard	51

Martin Lucie;Clara 14

Données sous la forme d'un tableau

Le module `csv` de Python permet de simplifier l'utilisation des fichiers CSV.

Lire un fichier CSV

Pour lire un fichier CSV, vous devez ouvrir un flux de lecture de fichier et ouvrir à partir de ce flux un lecteur CSV. Pour ignorer la ligne d'en-tête, utilisez `next(lecteurCSV)` :

```
import csv
fichier = open("noms.csv", "rt")
lecteurCSV = csv.reader(fichier,delimiter=";")        # Ouverture du lecteur
CSV en lui fournissant le caractère séparateur (ici ";")
for ligne in lecteurCSV:
        print(ligne)    # Exemple avec la 1e ligne du fichier d'exemple :
['Nom', 'Prénom', 'Age']
fichier.close()
```

Vous obtiendrez en résultat une liste contenant chaque colonne de la ligne en cours. Vous pouvez modifier le délimiteur de champs texte en définissant la variable `quotechar` :

```
import csv
fichier = open("noms.csv", "rt")
lecteurCSV = csv.reader(fichier,delimiter=";",quotechar="'")
# Définit l'apostrophe comme délimiteur de champs texte
for ligne in lecteurCSV:
        print(ligne)
fichier.close()
```

Vous pouvez également lire les données et obtenir un dictionnaire par ligne contenant les données en utilisant `DictReader` au lieu de `reader` :

```
import csv
fichier = open("noms.csv", "rt")
lecteurCSV = csv.DictReader(fichier,delimiter=";")
for ligne in lecteurCSV:
        print(ligne)    # Résultat obtenu : {'Age': '29', 'Nom': 'Dubois',
'Prénom': 'Marie'}
fichier.close()
```

Écrire un fichier CSV

À l'instar de la lecture, on ouvre un flux d'écriture et on ouvre un écrivain CSV à partir de ce flux :

```
import csv
fichier = open("annuaire.csv", "wt")
ecrivainCSV = csv.writer(fichier,delimiter=";")
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"])    # On écrit la ligne
d'en-tête avec le titre des colonnes
ecrivainCSV.writerow(["Dubois","Marie","0198546372"])
ecrivainCSV.writerow(["Duval","Julien \"Paul\"","0399741052"])
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"])
ecrivainCSV.writerow(["Martin","Julie;Clara","0399731590"])
```

```
fichier.close()
```

Nous obtenons le fichier suivant :

```
Nom;Prénom;Téléphone
Dubois;Marie;0198546372
Duval;"Julien ""Paul""";0399741052
Jacquet;Bernard;0200749685
Martin;"Julie;Clara";0399731590
```

Il est également possible d'écrire le fichier en fournissant un dictionnaire par ligne à condition que chaque dictionnaire possède les mêmes clés. On doit également fournir la liste des clés des dictionnaires avec l'argument `fieldnames` :

```
import csv
bonCommande = [
        {"produit":"cahier", "reference":"F452CP", "quantite":41,
"prixUnitaire":1.6},
        {"produit":"stylo bleu", "reference":"D857BL", "quantite":18,
"prixUnitaire":0.95},
        {"produit":"stylo noir", "reference":"D857NO", "quantite":18,
"prixUnitaire":0.95},
        {"produit":"équerre", "reference":"GF955K", "quantite":4,
"prixUnitaire":5.10},
        {"produit":"compas", "reference":"RT42AX", "quantite":13,
"prixUnitaire":5.25}
]
fichier = open("bon-commande.csv", "wt")
ecrivainCSV =
csv.DictWriter(fichier,delimiter=";",fieldnames=bonCommande[0].keys())
ecrivainCSV.writeheader()      # On écrit la ligne d'en-tête avec le titre
des colonnes
for ligne in bonCommande:
        ecrivainCSV.writerow(ligne)
fichier.close()
```

Nous obtenons le fichier suivant :

```
reference;quantite;produit;prixUnitaire
F452CP;41;cahier;1.6
D857BL;18;stylo bleu;0.95
D857NO;18;stylo noir;0.95
GF955K;4;équerre;5.1
RT42AX;13;compas;5.25
```

Par défaut, Python placera les guillemets autour des chaînes contenant des guillemets, une virgule ou un point virgule afin que ceux-ci ne soient pas confondus avec un délimiteur de champs ou le séparateur. Afin que tous les champs soient encadrés par les guillemets, nous allons modifier l'argument `quoting` pour `writer` ou `DictWriter` :

```
import csv
fichier = open("annuaire.csv", "wt")
ecrivainCSV = csv.writer(fichier,delimiter=";",quotechar="'",
quoting=csv.QUOTE_ALL) # quotechar modifie le caractère délimitant un champ
(par défaut : ")
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"])    # On écrit la ligne
d'en-tête avec le titre des colonnes
```

```
ecrivainCSV.writerow(["Dubois","Marie","0198546372"])
ecrivainCSV.writerow(["Duval","Julien \"Paul\"","0399741052"])
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"])
ecrivainCSV.writerow(["Martin","Julie;Clara","0399731590"])
fichier.close()
```

Nous obtenons le fichier suivant :

```
'Nom';'Prénom';'Téléphone'
'Dubois';'Marie';'0198546372'
'Duval';'Julien "Paul"';'0399741052'
'Jacquet';'Bernard';'0200749685'
'Martin';'Julie;Clara';'0399731590'
```

Le paramètre `quoting` peut prendre les valeurs suivantes.

Valeur	Action
`csv.QUOTE_ALL`	Met tous les champs entre guillemets.
`csv.QUOTE_MINIMAL`	Met les guillemets autour des chaînes contenant des guillemets et le séparateur de champs (par défaut).
`csv.QUOTE_NONNUMERIC`	Met les guillemets autour des valeurs non-numériques et indique au lecteur de convertir les valeurs non contenues entre les guillemets en nombres réels.
`csv.QUOTE_NONE`	Ne met aucun guillemet.

Le format JSON

Le format *JavaScript Object Notation* (JSON) est issu de la notation des objets dans le langage JavaScript. Il s'agit aujourd'hui d'un format de données très répandu permettant de stocker des données sous une forme structurée.

Il ne comporte que des associations clés → valeurs (à l'instar des dictionnaires), ainsi que des listes ordonnées de valeurs (comme les listes en Python). Une valeur peut être une autre association clés → valeurs, une liste de valeurs, un entier, un nombre réel, une chaîne de caractères, un booléen ou une valeur nulle. **Sa syntaxe est similaire à celle des dictionnaires Python.**

Voici un exemple de fichier JSON :

```
{
    "Dijon":{
        "nomDepartement": "Côte d'Or",
        "codePostal": 21000,
        "population": {
                "2006": 151504,
                "2011": 151672,
                "2014": 153668
        }
    },
    "Troyes":{
        "nomDepartement": "Aube",
        "codePostal": 10000,
        "population": {
```

```
                "2006": 61344,
                "2011": 60013,
                "2014": 60750
        }
    }
}
```

Il est également possible de compacter un fichier JSON en supprimant les tabulations et les retours à la ligne. On obtient ainsi :

```
{"Dijon":{"nomDepartement":"Côte
d'Or","codePostal":21000,"population":{"2006":151504,"2011":151672,"2014":1
53668}},"Troyes":{"nomDepartement":"Aube","codePostal":10000,"population":{
"2006":61344,"2011":60013,"2014":60750}}}
```

Pour lire et écrire des fichiers JSON, nous utiliserons le module `json` fourni nativement avec Python.

Lire un fichier JSON

La fonction `loads(texteJSON)` permet de décoder le texte JSON passé en argument et de le transformer en dictionnaire ou une liste.

```
>>> import json
>>> fichier = open("villes.json","rt")
>>> villes = json.loads(fichier.read())
>>> print(villes)
{'Troyes': {'population': {'2006': 61344, '2011': 60013, '2014': 60750},
'codePostal': 10000, 'nomDepartement': 'Aube'}, 'Dijon': {'population':
{'2006': 151504, '2011': 151672, '2014': 153668}, 'codePostal': 21000,
'nomDepartement': "Côte d'Or"}}
>>> fichier.close()
```

Écrire un fichier JSON

On utilise la fonction `dumps(variable, sort_keys=False)` pour transformer un dictionnaire ou une liste en texte JSON en fournissant en argument la variable à transformer. La variable `sort_keys` permet de trier les clés dans l'ordre alphabétique.

```
import json
quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
fichier = open("quantiteFournitures.json","wt")
fichier.write(json.dumps(quantiteFournitures))
fichier.close()
```

Gestion des erreurs

Lors de l'écriture de vos premiers scripts, vous avez peut-être rencontré ce type de message d'erreurs :

```
>>> resultat = 42/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

Ce message signale l'erreur ainsi que la ligne à laquelle a été faite cette erreur. Or il est courant que les erreurs ne soient pas des erreurs lors de la programmation mais une mauvaise manipulation de la part de l'utilisateur. Par exemple, vous demandez à l'utilisateur de fournir un nombre et celui-ci vous fournit un texte ou que le fichier, que vous cherchez à lire, n'existe pas.

Il faut alors gérer ce type d'erreurs afin d'éviter une interruption involontaire de notre application. Pour cela, nous utiliserons les structures `try`.

Voici un exemple sur lequel nous allons ajouter un mécanisme de gestion d'erreurs :

```
age = input("Quel est votre age : ")
age = int(age)
```

Voici l'erreur obtenue si la chaîne de caractères n'est pas un nombre :

```
>>> age = input("Quel est votre age : ")
Quel est votre age : Alain
>>> age = int(age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Alain'
```

La structure `try` se présente ainsi :

```
try:
        # La portion de code à tester
except:
        # Que faire en cas d'erreur
```

Nous obtenons donc pour notre exemple :

```
>>> age = input("Quel est votre age : ")
Quel est votre age : Alain
>>> try:
...     age = int(age)
... except:
...     print("Erreur lors de la saisie. ")
...
Erreur lors de la saisie.
```

Il est possible d'identifier l'erreur et d'effectuer une action en conséquence. Nous pouvons donc chaîner les instructions `except` **en fournissant le type d'erreur**. Le bloc `else` (optionnel) est exécuté s'il n'y a eu aucune erreur.

```
try:
        quantiteBoites = quantitePieces / nombresPiecesParBoites
except TypeError:                               # Type incompatible avec
l'opération
        print("Au moins une des variables n'est pas un nombre. ")
except NameError:                               # Variable non définie
        print("Au moins une des variables n'est pas définie. ")
except ZeroDivisionError:     # Division par 0
```

```
        print("Le nombres de pièces par boites est égal à 0. ")
else:
        print("Il faut commander " + str(quantiteBoites) + " boîtes. ")
```

Le bloc `finally` (optionnel) est exécuté dans tous les cas (s'il y a eu des erreurs ou non).

Enfin, l'instruction `pass` ne fait rien : elle permet de laisser un bloc vide ce qui est utile pour les exceptions.

Gérer les fichiers

Python fournit deux modules permettant de gérer les fichiers et les répertoires. Le module `os.path` permet de lister des fichiers et des répertoires, d'effectuer des opérations sur les URI. Le module `shutil` permet de copier, déplacer, supprimer des éléments sur les systèmes de fichiers.

Dans ce chapitre, nous travaillerons sur des systèmes de fichiers Unix (GNU/Linux, MacOS …).

Les chemins de fichiers

Nous allons étudier les fonctions permettant de manipuler les chemins de fichiers ou de répertoires du module `os.path`. La fonction `basename(URI)` renvoie le nom du fichier de l'adresse fourni en argument. À l'inverse, la fonction `dirname(URI)` renvoie le chemin jusqu'au fichier, sans le nom du fichier. **Ces fonctions fonctionnent même si le fichier n'existe pas.**

```
>>> import os.path
>>> chemin = "/tmp/dir/dir2/monFichier.txt"
>>> print(os.path.basename(chemin))
monFichier.txt
>>> print(os.path.dirname(chemin))
/tmp/dir/dir2
```

Différentier les fichiers et les répertoires

La fonction `exists(URI)` renvoie True si le fichier ou le répertoire fournis en argument existent. Les fonctions `isfile(URI)` et `isdir(URI)` permettent respectivement de vérifier si le chemin mène à un fichier et un répertoire. Ces fonctions renvoient True si c'est le cas.

```
>>> import os.path
>>> chemin = "/tmp/dir/dir2/monFichier.txt"
>>> print(os.path.exists(chemin))
True
>>> print(os.path.isfile(chemin))
True
>>> print(os.path.isdir(chemin))
False
>>> print(os.path.isdir(os.path.dirname(chemin)))
True
```

Lister le contenu d'un répertoire

La fonction `listdir(repertoire)` du module `os.path` retourne le contenu du répertoire passé en argument sans distinction entre les fichiers et les répertoires.

```
>>> import os.path
>>> print(os.listdir("/tmp/dir"))
['villes.json', 'quantiteFournitures.json', 'dir2']
```

Copier un fichier ou un répertoire

Il existe deux méthodes dans le module `shutil` permettant d'effectuer une copie. La fonction `copy(source, destination)` permet de copier un fichier, alors que `copytree` en fait de même avec les répertoires.

```
import shutil
shutil.copytree("/tmp/dir/dir2","/tmp/dir/dir3")
shutil.copy("/tmp/dir/dir2/monFichier.txt","/tmp/dir/exemple.txt")
```

Déplacer un fichier ou un répertoire

La fonction `move(source, destination)` du module `shutil` permet de déplacer un fichier ou un répertoire. Cela peut également servir à renommer le fichier ou le répertoire.

```
import shutil
shutil.move("/tmp/dir/dir3","/tmp/dir/perso")
```

Supprimer un fichier ou un répertoire

La méthode `remove(fichier)` du module `os` et la fonction `rmtree(dossier)` du module `shutil` permettent respectivement de supprimer un fichier et un répertoire.

```
import os,shutil
os.remove("/tmp/dir/exemple.txt")
shutil.rmtree("/tmp/dir/perso")
```

Sauvegarder des variables

Le module `pickle` permet de sérialiser des variables quelles qu'elles soient vers un fichier ouvert en flux binaire et de les restaurer. Cela équivaut à sauvegarder et restaurer l'état des variables.

La fonction `dump(variable, fichier)` permet d'exporter une variable vers un fichier et la fonction `load(fichier)` retourne la variable lue depuis le fichier.

L'ordre de sauvegarde et de restauration des variables doit être identique.

```
>>> import pickle
>>> texte = "Écrit par Antoine de Saint-Exupéry"
>>> quantiteFournitures = {"cahiers":134, "stylos":{"rouge":41,"bleu":74},
"gommes": 85}
```

```
>>> fournitures = ["cahier", "crayon", "stylo", "trousse", "gomme"]
>>> fichierSauvegarde = open("donnees","wb")
>>> pickle.dump(texte, fichierSauvegarde)
>>> pickle.dump(quantiteFournitures, fichierSauvegarde)
>>> pickle.dump(fournitures, fichierSauvegarde)
>>> fichierSauvegarde.close()
>>> import pickle
>>> fichierSauvegarde = open("donnees","rb")
>>> texte = pickle.load(fichierSauvegarde)
>>> quantiteFournitures = pickle.load(fichierSauvegarde)
>>> fournitures = pickle.load(fichierSauvegarde)
>>> fichierSauvegarde.close()
>>> print(texte)
Écrit par Antoine de Saint-Exupéry
>>> print(quantiteFournitures)
{'stylos': {'bleu': 74, 'rouge': 41}, 'gommes': 85, 'cahiers': 134}
>>> print(fournitures)
['cahier', 'crayon', 'stylo', 'trousse', 'gomme']
```

Exercices

1. Écrivez un programme permettant à un utilisateur de deviner un mot choisi au hasard dans un fichier nommé `mots.txt` dans lequel chaque ligne comporte un mot en capitale. L'utilisateur a 7 chances pour découvrir le mot en proposant une lettre à chaque fois. Si la lettre proposée n'est pas dans le mot, une chance lui est retirée.

   Exemple :

   ```
   - - - - - - - - - (7 chances)
   Entrez une lettre : S
   - - - - - - - - - (6 chances)
   Entrez une lettre : O
   - O - - - - - - - (6 chances)
   …
   Bravo ! Le mot était JOURNAUX.
   ou
   Perdu ! Le mot était JOURNAUX.
   ```

   Vous pouvez télécharger un fichier de mots ici :

• Écrivez un programme permettant de chiffrer un fichier texte nommé `texte.txt` à l'aide du chiffrement par décalage dans lequel chaque lettre est remplacée par une autre à distance fixe choisie par l'utilisateur. Par exemple, si la distance choisie est de 4, un A est remplacé par un D, un B par un E, un C par un F … Le résultat sera écrit dans un fichier texte nommé `chiffre.txt`.

Texte en clair	`ESOPE RESTE ICI ET SE REPOSE`
Texte chiffré (distance 7)	`LZVWL YLZAL PJP LA ZL YLWVZL`

Écrivez un programme permettant à partir d'un fichier texte en français d'en déduire la fréquence d'apparition des lettres qu'il contient. Le résultat de cette analyse sera consigné dans un fichier JSON. Pour des statistiques fiables, prenez un texte assez long. Vous pouvez utiliser une copie de Zadig, écrit par Voltaire, disponible ici :

- Le message suivant a été chiffré à l'aide de la technique de chiffrement par décalage et stocké dans le fichier `chiffre.txt` :

```
HFCMSG GS GWHIS ROBG ZS UFOBR SGH RS ZO TFOBQS OI
QSBHFS RI RSDOFHSASBH RS Z OIPS RCBH SZZS SGH ZS
QVST ZWSI SH O Z CISGH RS ZO FSUWCB UFOBR SGH ZO
QCAAIBS G SHSBR ROBG ZO DZOWBS RS QVOADOUBS
QFOMSIGS O DFCLWAWHS RI DOMG R CHVS SH RI DOMG R
OFAOBQS QSHHS JWZZS RS DZOWBS OZZIJWOZS G SHOPZWH
ROBG ZO JOZZSS RS ZO GSWBS
```

À l'aide des statistiques d'apparition des lettres issues de l'exercice précédent stockées dans le fichier `statistiques.json`, déduisez le message en clair du texte ci-dessus et stockez le dans le fichier `clair.txt`.

Écrivez un programme permettant de calculer la moyenne d'un étudiant dont les notes sont consignées dans un fichier CSV nommé `notes.csv`. Chaque colonne correspond à une matière. Vous devrez écrire un fichier JSON nommé `moyennes.json` consignant ces moyennes ainsi que la moyenne générale. Toutes les notes et les matières sont au cœfficient 1. Un exemple de fichier CSV est disponible ici :

Exemple :

```
{
        "Mathématiques": 8.411111111111111,
        "Sciences naturelles": 17.412,
        …
        "Moyenne générale": 14.965248762650994
}
```

Chap 6 : Interagir avec les bases de données

Ce chapitre aborde les bases de données SQL. Si toutefois ce type de base de données vous est inconnu, un chapitre permettant de vous y introduire est disponible en annexe.

Peu à peu, nos programmes manipuleront un très grand nombre de données et nécessiteront un système plus performant pour stocker et lire ces données. Pour cela, nous ferons appel à des **bases de données**. Les bases de données permettent de stocker de grands volumes de données sous une forme normalisée et qui peut être utilisée par plusieurs programmes différents. Nous utiliserons dans ce cours deux SGBD gratuits :

MariaDB
: Copie de MySQL, disponible gratuitement et sous licence libre (GPL). Le couple MariaDB et MySQL fait partie des SGBD les plus utilisés et les plus massivement déployés.

SQLite3
: Il s'agit d'une bibliothèque écrite en C dans le domaine public permettant d'interagir avec des bases de données stockées dans des fichiers. À l'inverse de MariaDB, il ne repose pas sur une architecture client/serveur. Ce système est particulièrement adapté

pour les petites bases de données stockées localement, comme alternative aux fichiers texte.

Utiliser une base de données SQLite3

Créer la base et insérer des données

Nous allons tout d'abord importer le module `sqlite3`. Dans l'exemple qui suit, nous allons créer une base de données avec une table contenant un répertoire téléphonique contenant le nom, le prénom, l'adresse et le numéro de téléphone fixe des contacts. Nous verrons comment insérer quelques enregistrements.

```
import sqlite3
baseDeDonnees = sqlite3.connect('contacts.db')
curseur = baseDeDonnees.cursor()
curseur.execute("CREATE TABLE Contacts (id INTEGER PRIMARY KEY
AUTOINCREMENT, nom TEXT NOT NULL, prenom TEXT NOT NULL, adresse TEXT NOT
NULL, telephoneFixe TEXT)") # Création de la base de données
baseDeDonnees.commit() # On envoie la requête SQL
curseur.execute("INSERT INTO Contacts (nom, prenom, adresse, telephoneFixe)
VALUES (?, ?, ?, ?)", ("Dupont", "Paul", "15 rue Louis Pasteur 10000
Troyes", "0325997452")) # On ajoute un enregistrement
baseDeDonnees.commit()
baseDeDonnees.close()
```

Il est possible d'ajouter un enregistrement depuis un dictionnaire. Dans l'exemple, on ajoute plusieurs enregistrements avec une boucle :

```
import sqlite3
baseDeDonnees = sqlite3.connect('contacts.db')
curseur = baseDeDonnees.cursor()
personnes = [
        {"nom":"Chabot", "prenom":"Martin", "adresse":"18 rue Général
Leclerc 13600 La Ciotat", "telephoneFixe":"0499506373"},
        {"nom":"Delbois", "prenom":"Julie", "adresse":"35 rue du Château
77176 Savigny le Temple", "telephoneFixe":"0199836074"},
        {"nom":"Rivard", "prenom":"Christelle", "adresse":"83 rue de Québec
83400 Hyères", "telephoneFixe":"0499687013"}
]
for contact in personnes:
        curseur.execute("INSERT INTO Contacts (nom, prenom, adresse,
telephoneFixe) VALUES (:nom, :prenom, :adresse, :telephoneFixe)", contact)
# On ajoute un enregistrement depuis un dictionnaire
baseDeDonnees.commit()
idDernierEnregistrement = curseur.lastrowid # Récupère l'ID de la dernière
ligne insérée.
baseDeDonnees.close()
```

L'exemple suivant illustre comment modifier des données :

```
import sqlite3
baseDeDonnees = sqlite3.connect('contacts.db')
curseur = baseDeDonnees.cursor()
curseur.execute("UPDATE Contacts SET telephoneFixe = ? WHERE id = ?",
("0598635076", 2))
baseDeDonnees.commit()
```

```
baseDeDonnees.close()
```

Récupérer des données

Pour récupérer les données, il est possible de récupérer le premier résultat avec `fetchone` ou de retourner tous les résultats avec `fetchall`. Voici un premier exemple utilisant `fetchone` :

```
>>> import sqlite3
>>> baseDeDonnees = sqlite3.connect('contacts.db')
>>> curseur = baseDeDonnees.cursor()
>>> curseur.execute("SELECT nom, prenom, telephoneFixe FROM Contacts WHERE
id = ?", ("2",))
>>> contact = curseur.fetchone()
>>> print(contact)
('Chabot', 'Martin', '0598635076')
>>> baseDeDonnees.close()
```

Dans l'exemple ci-dessus, la variable `contact` contient un tuple avec les valeurs du premier enregistrement retourné par la requête.

Voyons à présent comment récupérer plusieurs enregistrements avec la commande `fetchall` :

```
>>> import sqlite3
>>> baseDeDonnees = sqlite3.connect('contacts.db')
>>> curseur = baseDeDonnees.cursor()
>>> curseur.execute("SELECT nom, prenom, telephoneFixe FROM Contacts")
>>> for contact in curseur.fetchall():
...     print(contact)
...
('Dupont', 'Paul', '0325997452')
('Chabot', 'Martin', '0598635076')
('Delbois', 'Julie', '0199836074')
('Rivard', 'Christelle', '0499687013')
>>> baseDeDonnees.close()
```

Utiliser une base de données MariaDB/MySQL

Pour cette partie, vous devez avoir installé le paquet `python3-mysql.connector`.

Créer la base et insérer des données

Nous allons utiliser le module `mysql.connector` pour nous connecter au serveur MariaDB ou MySQL et créer notre base de données permettant de stocker un catalogue de produits.

```
import mysql.connector
baseDeDonnees =
mysql.connector.connect(host="localhost",user="catalogue",password="JieTh8T
h", database="Catalogue")
cursor = baseDeDonnees.cursor()
```

```
cursor.execute("CREATE TABLE Produits (reference CHAR(5) NOT NULL PRIMARY
KEY, nom TINYTEXT NOT NULL, prix FLOAT NOT NULL)ENGINE=InnoDB DEFAULT
CHARSET=utf8;")
baseDeDonnees.close()
```

Nous allons à présent insérer des données dans cette table.

```
import mysql.connector
baseDeDonnees =
mysql.connector.connect(host="localhost",user="catalogue",password="JieTh8T
h", database="Catalogue")
curseur = baseDeDonnees.cursor()
curseur.execute("INSERT INTO Produits (reference, nom, prix) VALUES (%s,
%s, %s)", ("ARB42", "Canapé deux places noir", 199.99))
baseDeDonnees.commit()
baseDeDonnees.close()
```

Il est également possible d'insérer des données depuis un dictionnaire :

```
import mysql.connector
baseDeDonnees =
mysql.connector.connect(host="localhost",user="catalogue",password="JieTh8T
h", database="Catalogue")
curseur = baseDeDonnees.cursor()
produits = [
        {"reference":"EIS3P", "nom":"Chaise de salle à manger", "prix":25},
        {"reference":"BA9KI", "nom":"Commode blanche", "prix":139.90},
        {"reference":"OI4HE", "nom":"Table basse", "prix":24.95},
        {"reference":"IOM9X", "nom":"Lit double", "prix":699.99}
]
for fiche in produits:
        curseur.execute("INSERT INTO Produits (reference, nom, prix) VALUES
(%(reference)s, %(nom)s, %(prix)s)", fiche)
baseDeDonnees.commit()
baseDeDonnees.close()
```

Récupérer des données

À l'instar de SQLite, on peut utiliser `fetchone` pour récupérer le premier résultat ou retourner tous les résultats avec `fetchall`. Voici comment récupérer le premier résultat d'une requête `SELECT`.

```
>>> import mysql.connector
>>> baseDeDonnees =
mysql.connector.connect(host="localhost",user="catalogue",password="JieTh8T
h", database="Catalogue")
>>> curseur = baseDeDonnees.cursor()
>>> curseur.execute("SELECT reference, nom, prix FROM Produits")
>>> print(curseur.fetchone())
('ARB42', 'Canapé deux places noir', 199.99)
>>> baseDeDonnees.close()
```

On peut retourner tous les résultats avec `fetchall` :

```
>>> import mysql.connector
```

```
>>> baseDeDonnees =
mysql.connector.connect(host="localhost",user="catalogue",password="JieTh8T
h", database="Catalogue")
>>> curseur = baseDeDonnees.cursor()
>>> curseur.execute("SELECT reference, nom, prix FROM Produits")
>>> for ligne in curseur.fetchall():
...     print(ligne)
...
('ARB42', 'Canapé deux places noir', 199.99)
('BA9KI', 'Commode blanche', 139.9)
('EIS3P', 'Chaise de salle à manger', 25.0)
('IOM9X', 'Lit double', 699.99)
('OI4HE', 'Table basse', 24.95)
>>> baseDeDonnees.close()
```

Exercices

Vous êtes nouvellement embauché dans le secrétariat de scolarité d'une université. Votre travail est d'optimiser la gestion des étudiants, des enseignants, des matières enseignées, des inscriptions des étudiants à ces dernières et des résultats obtenus. À votre arrivée, une collègue vous fournit les fichiers tableurs permettant d'accomplir ces tâches (au format CSV) :

Liste des étudiants `etudiants.csv`
Liste des enseignants `enseignants.csv`
Liste des matières `matieres.csv`
Liste des inscriptions `inscriptions.csv`
Liste des résultats `resultats.csv`

Les exercices suivant permettront d'effectuer cela en scindant le travail en différentes sous-tâches. Chaque sous-tâche fera l'objet d'un nouveau programme. Tous ces programmes utiliseront la même base de données SQLite3. Toutes les moyennes auront deux décimales.

Voici le schéma UML de la base de données de l'université :

1. Écrivez un programme permettant de créer une base de données SQLite3 nommée `universite.db` et de créer la structure de table adaptée au stockage des données. Importez le contenu des fichiers CSV dans cette base.

   Chaque table aura le même nom que le fichier CSV source.

• Écrivez un programme permettant de générer des statistiques pour l'université au format JSON dans un fichier nommé `statistiques.json` dont nous stockerons les moyennes par matière `moyenneMatiere`, la moyenne maximale `moyenneMax` et minimale `moyenneMin` par matière, le nombre d'étudiants inscrits par matière `nbEtudiants`, la moyenne de toutes les matières `moyenneTotale` et le nombre d'étudiants par département `nbEtudiantsParDepartement` (les deux premiers nombres du code postal).

Exemple :

```
{
   "nbEtudiantsParDepartement":{
      "56":8,
      "74":3,
      …
   },
   "moyenneMax":{
      "CF19":14.96,
      "VS02":12.14,
      …
   },
   "moyenneMatiere":{
      "CF19":13.37,
      "VS02":14.4,
      …
   },
   "moyenneMin":{
      "CF19":3.99,
      "VS02":4.21,
      …
   },
   "nbEtudiants":{
      "CF19":8,
      "VS02":17,
      …
   },
   "moyenneTotale":12.41
}
```


- Écrivez un programme permettant de générer un bulletin de notes par étudiant sous la forme d'un courrier stocké dans un fichier texte individuel. Chaque fichier aura pour nom le nom et le prénom de l'étudiant, séparés par un trait d'union (-) et pour extension `.txt` et sera stocké dans un dossier nommé `courriersEtudiants` créé pour cela. Chaque courrier adoptera ce modèle :

```
Université Claude Chappe
15 avenue de Moulincourbe
28094 Clairecombe
                                                          Lionel Paulin
                                                          48 Ruelle de
Locvaux
                                                          74019 Mivran

Madame, Monsieur,
Veuillez trouver dans le récapitulatif ci-dessous les résultats de vos
examens.
Matière                       Moyenne
AI90                          16.93
PQ84                          12.7
UE21                          12.0
VO38                          12.49
XO83                          13.05
ZI51                          16.33
Moyenne générale       13.92

Ce document constitue les résultats officiels. Pour toute contestation,
contactez le service scolarité.
```

- Écrivez un programme permettant l'inscription d'un nouveau étudiant à l'université et son inscription aux matières.

Exemple :

```
Entrez le nom du nouvel étudiant : Durand
Entrez le prénom du nouvel étudiant : Michel
Entrez l'adresse du nouvel étudiant : 15 rue Jean Moulin
Entrez le code postal du nouvel étudiant : 18543
Entrez la ville du nouvel étudiant : Moulincourbe
Entrez le téléphone fixe du nouvel étudiant : 0574960180
Entrez le téléphone portable du nouvel étudiant : 0641238074
Entrez le code de matière dans laquelle inscrire l'étudiant (laissez vide
pour arrêter, tapez '?' pour afficher la liste) : ?
AD23
AF04
…
Entrez le code de matière dans laquelle inscrire l'étudiant (laissez vide
pour arrêter, tapez '?' pour afficher la liste) : LK27
Entrez le code de matière dans laquelle inscrire l'étudiant (laissez vide
pour arrêter, tapez '?' pour afficher la liste) : AD24
La matière n'existe pas
Entrez le code de matière dans laquelle inscrire l'étudiant (laissez vide
pour arrêter, tapez '?' pour afficher la liste) : ZB17
Entrez le code de matière dans laquelle inscrire l'étudiant (laissez vide
pour arrêter, tapez '?' pour afficher la liste) :
Voulez-vous saisir un nouvel étudiant (O/N) ? n
```

Écrivez un programme permettant la saisie des notes obtenues aux examens.

Exemple :

```
Entrez le code de matière de l'examen (laissez vide pour arrêter, tapez '?'
pour afficher la liste) : AF04
Entrez la note obtenue par Maxime Bey : 17
…
Entrez la note obtenue par Margot Ferrero : 14
Voulez-vous saisir un nouveau résultat (O/N) ? n
```

Chap 7 : La programmation réseau

Nos programmes peuvent à présent effectuer des tâches complexes et peuvent s'interfacer entre eux par le biais de fichiers ou de bases de données. Voyons à présent comment faire communiquer plusieurs programmes fonctionnant sur des ordinateurs différents *via* le réseau informatique. L'objectif de ce chapitre est d'aborder la communication entre plusieurs ordinateurs avec le mécanisme de sockets.

Un *socket*, que nous pouvons traduire par connecteur réseau, est une interface aux services réseaux offerte par le système d'exploitation permettant d'exploiter facilement le réseau. Cela permet d'initier une session TCP, d'envoyer et de recevoir des données par cette session. Nous utiliserons pour ce chapitre le module `socket`.

Nous travaillerons avec deux scripts, le serveur permettant d'écouter les demandes des clients et d'y répondre. Le client se connectera sur le serveur pour demander le service. Il est possible d'exécuter à la fois le client et le serveur sur un même ordinateur. Pour cela, il vous suffit de renseigner `127.0.0.1` comme adresse IP pour la partie client.

Créer un serveur socket

Nous allons commencer par construire une application serveur très simple qui reçoit les connexions clients sur le port désigné, envoie un texte lors de la connexion, affiche ce que le client lui envoie et ferme la connexion.

```
#!/usr/bin/env python3
import socket
serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind(('', 50000))      # Écoute sur le port 50000
serveur.listen(5)
while True:
        client, infosClient = serveur.accept()
        print("Client connecté. Adresse " + infosClient[0])
        requete = client.recv(255)     # Reçoit 255 octets. Vous pouvez
changer pour recevoir plus de données
        print(requete.decode("utf-8"))
        reponse = "Bonjour, je suis le serveur"
        client.send(reponse.encode("utf-8"))
        print("Connexion fermée")
        client.close()
serveur.close()
```

Vous remarquerez la présence de l'instruction `encode("utf-8")`. Cette indication demande à Python de convertir la chaîne de caractères en flux binaire UTF-8 pour permettre son émission sur le réseau. L'instruction `decode("utf-8")` permet d'effectuer l'opération inverse. Nous ne pouvons pas tester le programme tant que nous n'avons pas écrit la partie cliente.

Créer un client socket

L'application cliente présentée ici permet de se connecter à un serveur spécifié sur un port désigné. Une fois la connexion établie, il enverra un message au serveur et affichera le message que le serveur lui enverra en retour.

```
#!/usr/bin/env python3
import socket
adresseIP = "127.0.0.1"        # Ici, le poste local
port = 50000   # Se connecter sur le port 50000
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((adresseIP, port))
print("Connecté au serveur")
client.send("Bonjour, je suis le client".encode("utf-8"))
reponse = client.recv(255)
print(reponse.decode("utf-8"))
print("Connexion fermée")
client.close()
```

Cela fonctionne mais le serveur présente un défaut, il ne peux pas gérer plusieurs clients à la fois. Nous allons y remédier dans la suite de ce chapitre.

L'exécution de fonctions en parallèle : le multithread

Le multithread permet l'exécution de plusieurs opérations simultanément sur les mêmes ressources matérielles (ici, l'ordinateur). Les différents threads sont traités à tour de rôle par l'ordinateur pendant un temps très court ce qui donne cette impression d'exécution parallèle.

Nous abordons cette technique pour pouvoir élaborer un serveur pouvant gérer plusieurs connexions client en même temps. Tout d'abord, nous allons nous familiariser avec le module `threading` qui met en œuvre le multithread pour les fonctions et les objets Python. Voici un exemple simple montrant le fonctionnement de cette technique :

```
#!/usr/bin/env python3
import threading
def compteur(nomThread):
        for i in range(3):
                print(nomThread + " : " + str(i))
threadA = threading.Thread(None, compteur, None, ("Thread A",), {})
threadB = threading.Thread(None, compteur, None, ("Thread B",), {})
threadA.start()
threadB.start()
```

Voici un des résultats possibles lors de l'exécution du script ci-dessus. On observe que l'affichage du thread A et B sont confondus :

```
Thread A : 0
Thread A : 1
Thread B : 0
Thread B : 1
Thread A : 2
Thread B : 2
```

Nous allons détailler la fonction créant le thread : `threading.Thread(groupe, cible, nom, arguments, dictionnaireArguments)` dont voici les arguments :

groupe
: Doit être à `None`, réservé à un usage futur.

cible
: Le nom de la fonction à exécuter.

nom
: Le nom du thread (facultatif, peut être défini à `None`).

arguments
: Un tuple donnant les arguments de la fonction cible.

dictionnaireArguments
: Un dictionnaire donnant les arguments de la fonction cible. Avec l'exemple ci-dessus, on utiliserait `{nomThread="Thread A"}`.

Créer un serveur socket acceptant plusieurs clients

Voici donc la combinaison du serveur de socket vu précédemment et la technique du multithreading pour obtenir un serveur plus complexe créant un nouveau thread à chaque client connecté. Nous avons apporté une petite modification au client car désormais, le client

demande à l'utilisateur de saisir un message qui sera affiché sur le serveur. Ce dernier répondra à tous les messages du client jusqu'à ce que l'utilisateur saisisse le mot `FIN` sur le client ce qui termine la connexion. Voici le script serveur :

```
#!/usr/bin/env python3
import socket
import threading
threadsClients = []
def instanceServeur (client, infosClient):
        adresseIP = infosClient[0]
        port = str(infosClient[1])
        print("Instance de serveur prêt pour " + adresseIP + ":" + port)
        message = ""
        while message.upper() != "FIN":
                message = client.recv(255).decode("utf-8")
                print("Message reçu du client " + adresseIP + ":" + port + "
: " + message)
                client.send("Message reçu".encode("utf-8"))
        print("Connexion fermée avec " + adresseIP + ":" + port)
        client.close()
serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind(('', 50000))      # Écoute sur le port 50000
serveur.listen(5)
while True:
        client, infosClient = serveur.accept()
        threadsClients.append(threading.Thread(None, instanceServeur, None,
(client, infosClient), {}))
        threadsClients[-1].start()
serveur.close()
```

Et voici le nouveau script client :

```
#!/usr/bin/env python3
import socket
adresseIP = "127.0.0.1"        # Ici, le poste local
port = 50000    # Se connecter sur le port 50000
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((adresseIP, port))
print("Connecté au serveur")
print("Tapez FIN pour terminer la conversation. ")
message = ""
while message.upper() != "FIN":
        message = input("> ")
        client.send(message.encode("utf-8"))
        reponse = client.recv(255)
        print(reponse.decode("utf-8"))
print("Connexion fermée")
client.close()
```

À chaque nouvelle connexion d'un client, le serveur crée un thread dédié à ce client, ce qui permet au programme principal d'attendre la connexion d'un nouveau client. Chaque client peut alors avoir une conversation avec le serveur sans bloquer les autres conversations.

Voici un exemple de l'affichage sur le serveur :

```
Instance de serveur prêt pour 127.0.0.1:57282
Message reçu du client 127.0.0.1:57282 : Je suis client 1
Instance de serveur prêt pour 127.0.0.1:57365
```

```
Message reçu du client 127.0.0.1:57365 : Je suis client 2
Message reçu du client 127.0.0.1:57282 : On
Message reçu du client 127.0.0.1:57365 : peux
Message reçu du client 127.0.0.1:57282 : envoyer
Message reçu du client 127.0.0.1:57365 : des
Message reçu du client 127.0.0.1:57282 : messages
Message reçu du client 127.0.0.1:57365 : ensemble
Message reçu du client 127.0.0.1:57282 : FIN
Connexion fermée avec 127.0.0.1:57282
Message reçu du client 127.0.0.1:57365 : FIN
Connexion fermée avec 127.0.0.1:57365
```

Créer un serveur Web

Nous allons ici utiliser la bibliothèque `http.server` pour créer rapidement un serveur Web capable de servir des fichiers à un navigateur Web. Le script ci-dessous montre comment créer cela en spécifiant le numéro de port sur lequel notre serveur Web va écouter et quel dossier il va servir.

```
import http.server
portEcoute = 80         # Port Web par défaut
adresseServeur = ("", portEcoute)
serveur = http.server.HTTPServer
handler = http.server.CGIHTTPRequestHandler
handler.cgi_directories = ["/tmp"]    # On sert le dossier /tmp
print("Serveur actif sur le port ", portEcoute)
httpd = serveur(adresseServeur, handler)
httpd.serve_forever()
```

Ce serveur retournera par défaut le fichier `index.html` du dossier servi. Si ce fichier n'existe pas, le serveur retournera la liste des fichiers du dossier.

Utiliser des services Web

De plus en plus de programmes utilisent et proposent aujourd'hui des interfaces de programmation nommées API. Cela permet de standardiser l'interaction entre les différentes applications et de découper les différentes fonctionnalités d'une application en divers modules qui communiquent ensemble avec ces interfaces.

Nous allons voir comment utiliser des services Web proposés pour enrichir nos applications Python en utilisant le module `urllib.request`. Notre premier exemple sera d'afficher la position de la Station Spatiale Internationale (ISS) qui nous est fournie par l'API http://api.open-notify.org/iss-now.json qui nous renvoie un texte au format JSON sous cette forme :

```
{
    "iss_position": {
        "longitude": "-100.8325",
        "latitude": "-12.0631"
    },
    "timestamp": 1493971107,
    "message": "success"
}
```

Voici le code source permettant de récupérer la donnée et en récupérer les parties utiles :

```
import urllib.request
import json
requete = urllib.request.Request('http://api.open-notify.org/iss-now.json')
        # La requête de l'API
reponse = urllib.request.urlopen(requete)     # Récupérer le fichier JSON
donneesBrut = reponse.read().decode("utf-8")  # Décoder le texte reçu
donneesJSON = json.loads(donneesBrut) # Décoder le fichier JSON
position = donneesJSON["iss_position"]
print("La station spatiale internationale est située à une longitude " +
position["longitude"] + " et à une latitude " + position["latitude"] + ".
")
```

Nous allons utiliser une seconde API permettant de trouver les communes associées à un code postal. L'API http://api.zippopotam.us/FR/XXXXX où `XXXXX` est le code postal recherché. Voici un exemple de données retournées :

```
{
        "post code": "21000",
        "country": "France",
        "country abbreviation": "FR",
        "places": [
                {
                        "place name": "Dijon",
                        "longitude": "5.0167",
                        "state": "Bourgogne",
                        "state abbreviation": "A1",
                        "latitude": "47.3167"
                }
        ]
}
```

Voici le programme permettant d'afficher les communes associées à un code postal :

```
import urllib.request
import json
codePostal = input("Entrez le code postal : ")
requete = urllib.request.Request('http://api.zippopotam.us/FR/' +
codePostal)
reponse = urllib.request.urlopen(requete)
donneesBrut = reponse.read().decode("utf-8")
donneesJSON = json.loads(donneesBrut)
listeCommunes = donneesJSON["places"]
print("Voici les communes ayant pour code postal " + codePostal + " : ")
for commune in listeCommunes:
        print(" - " + commune["place name"])
```

Exercices

Vous êtes nouvellement embauché dans une banque pour mettre au point le système de communication entre les distributeurs automatiques et le système central de gestion des comptes. Votre travail est de développer les applications sur ces deux systèmes pour permettre aux distributeurs de communiquer avec le système central pour effectuer les transactions.

On souhaite créer une base de données SQLite3 stockant les soldes des comptes, les informations les concernant, ainsi que les transactions effectuées. On utilisera un serveur de socket pour effectuer les communications entre les deux systèmes. Voici les différents messages pris en charge avec (C→S) un message du client vers le serveur et (S→C) un message du serveur vers le client :

- `TESTPIN numeroCompte codePIN` (C→S) : Vérifier si le code PIN saisi est correct.
- `TESTPIN OK` (S→C) : La vérification du code PIN est validée.
- `TESTPIN NOK` (S→C) : La vérification du code PIN n'est pas valide.
- `RETRAIT numeroCompte montant` (C→S) : Demande un retrait du montant défini.
- `RETRAIT OK` (S→C) : Le retrait est validé.
- `RETRAIT NOK` (S→C) : Le retrait est refusé.
- `DEPOT numeroCompte montant` (C→S) : Demande un dépôt du montant défini.
- `DEPOT OK` (S→C) : Le dépôt est validé.
- `TRANSFERT numeroCompteSource numeroCompteDestination montant` (C→S) : Demande un transfert du montant défini entre deux comptes.
- `TRANSFERT OK` (S→C) : Le transfert est validé.
- `TRANSFERT NOK` (S→C) : Le transfert est refusé.
- `SOLDE numeroCompte` (C→S) : Demande le solde du compte
- `SOLDE solde` (S→C) : Renvoie le solde du compte demandé
- `HISTORIQUE numeroCompte` (C→S) : Demande les 10 dernières opérations du compte
- `HISTORIQUE operationsEnCSV` (S→C) : Renvoie les 10 dernières opérations du compte au format CSV (date, libellé, montant déposé ou retiré).
- `ERROPERATION` (S→C) : Signale que l'opération demandée n'est pas valide.

Voici le schéma UML de la base de données de la banque :

La direction de la banque vous fournit les fichiers CSV contenant :

Liste des clients `clients.csv`
Liste des comptes `comptes.csv`
Liste des opérations `operations.csv`

Toute opération doit être précédée d'une vérification du code PIN. Les exercices suivants permettront d'effectuer cela en scindant le travail en différentes sous-tâches. Chaque tâche fera l'objet d'un nouveau programme.

1. Écrivez un programme permettant de créer sur le serveur une base de données SQLite3 nommée `banque.db` et de créer la structure de table adaptée au stockage des données. Importez le contenu des fichiers CSV dans cette base.

Écrivez le programme du serveur central de la banque écoutant sur le port `50000`.

- Écrivez le programme des distributeurs automatiques de la banque.

Chap 8 : Modélisation pour la programmation orientée objet

À présent, nous sommes capables de concevoir des programmes très complexes pouvant effectuer plusieurs opérations différentes. Nos programmes comportent un très grand nombre d'instructions. Il est temps à présent d'introduire un nouveau paradigme de programmation permettant de structurer nos programmes : la **programmation orientée objet**. Un objet est une structure de données comportant ses propres variables et ses propres fonctions.

Tout au long de ce cours, nous avons utilisé différents objets Python. On peut citer par exemple les listes, contenant elles-mêmes différentes variables (les éléments qu'elles contiennent) et différentes fonctions propres (`len`, `sort`, …) qui agissent sur les listes en question.

Dans ce chapitre et le suivant, nous allons modéliser nos propres objets et les mettre en œuvre en Python. Pour la modélisation, nous allons utiliser la représentation UML, largement utilisée en génie logiciel.

Présentation d'un objet

Une classe est une description des caractéristiques d'un objet. Elle décrit les variables que comportent tous les objets de la même classe. On parle d'une **instance** de la classe en question pour tout objet créé à partir de cette classe. La classe est donc une sorte de "moule" pour les instances d'objets. Une classe peut également comporter des fonctions agissant sur les instances de celles-ci appelées **méthodes**. On représente le type de données retournées par une méthode en terminant le nom de la méthode avec deux points suivis du type de données. Si la méthode ne retourne rien, on inscrit `void`.

La représentation UML représente une classe par un rectangle divisé horizontalement en trois parties dont le premier tiers renseigne le nom de la classe, le second décrit les variables de celle-ci et le troisième tiers liste les méthodes de la classe.

Pour illustrer le concept d'objet, nous allons prendre pour exemple la modélisation d'une personne. Cette personne a un nom, un prénom, une date de naissance et un lieu de résidence. La classe comporte également la méthode `demenager` qui permet de changer de lieu de résidence, ainsi que la méthode {calculerAge} retournant l'âge sous forme d'un entier. Nous allons modéliser cela sous la forme d'une **classe** :

Il existe en programmation orientée objet une méthode que l'on retrouve dans tous les objets, à savoir le **constructeur**. Cette méthode spéciale exécutée lors de l'instanciation de la classe permet de créer l'objet en créant les variables que contient l'objet, en définissant leur valeur initiale ou en effectuant certaines opérations. De manière analogue, il existe le **destructeur** appelé lors de la destruction de l'objet.

L'héritage

Une classe, que nous nommerons ici **classe fille** peut récupérer les variables et les méthodes d'une **classe mère**. On parle ici d'héritage. L'intérêt de cette opération est de partager des variables et des méthodes communes à différentes classes. Une classe fille peut alors ajouter des éléments pour compléter l'héritage de la classe mère ou peut écraser certaines méthodes ou variables avec son contenu propre.

On parle d'**héritage simple** dans le cas où une classe fille a recours à une seule classe mère. Par extension, on parle d'**héritage multiple** si une classe fille hérite des attributs de plusieurs classes mères.

L'héritage a plusieurs propriétés :

Transitivité
: Si une classe B hérite de la classe A, et que C hérite de B, alors C hérite de A.

Non réflexif
: Une classe ne peut hériter d'elle-même.

Non symétrique
: La classe fille hérite de la classe mère mais la classe mère n'hérite pas de la classe fille.

Sans cycle
: Il n'est pas possible que la classe B hérite de la classe A, que la classe C hérite de la classe B et que la classe A hérite de la classe C.

Nous allons illustrer ce mécanisme avec l'exemple de la classe `Chien` et la classe `Poule` qui héritent tous deux de la classe `Animaux`. La flèche terminée par un triangle indique l'héritage.

Pour illustrer l'héritage multiple, nous allons prendre l'exemple d'une imprimante multifonction permettant à la fois de scanner et d'imprimer (nous considérons que la photocopie est une combinaison des deux opérations précédentes) .

L'encapsulation

Le principe d'encapsulation est un des concepts clés de la programmation objet. Cette technique permet de regrouper des données au sein d'une classe et de pouvoir les lire et les manipuler par le biais de méthodes qui forment une interface pour les éléments environnant l'objet. De ce fait, les variables et les méthodes contenues au sein d'une classe peuvent être placées dans trois niveaux de visibilité en fonction de la portée que vous souhaitez :

Public
: La variable ou fonction est accessible à tous.

Protégé
: Les attributs sont accessibles uniquement à la classe et les classes filles.

Privé
: Les attributs ne sont visibles que par la classe elle-même.

Pour respecter le principe fondamental de l'encapsulation, toutes les variables doivent être privées et ne sont accessibles qu'à travers des méthodes conçues pour les modifier qu'après les avoir vérifiées ou les retourner formatées. Cela se représente dans un diagramme UML par un caractère précédant le nom de l'attribut ou de la méthode tel que décrit dans le tableau suivant.

Caractère	Rôle
+	Accès public
#	Accès protégé
-	Accès privé

Nous allons illustrer cela avec notre exemple précédent de classe `Personnes`.

L'association

Enfin, il est naturellement possible d'utiliser certains objets dans les variables d'autres objets. On modélise l'association entre les classes par un trait et on fait figurer le type de l'objet en vis-à-vis du nom de la variable, séparés par deux points. On peut faire figurer une association monodirectionnelle avec une flèche.

L'agrégation et la composition

L'agrégation et la composition désignent le fait qu'un objet soit composé d'autres objets. La composition entraîne l'appartenance du sous-objet à l'objet alors que l'agrégation non. Nous allons illustrer par exemple le cas d'une université composée de départements qui agrègent des professeurs. Lors de la destruction de l'université, les départements sont détruits, mais pas les professeurs. L'agrégation est représentée par un lien terminé par un losange vide (◊) et la composition est un lien terminé par un losange plein (♦). Nous allons représenter notre université par un diagramme UML.

Exercices

1. Modélisez en UML, un programme permettant de représenter les atomes de la table périodique des éléments. On y stockera son symbole, son nom complet et son numéro atomique.

- Modélisez en UML, un programme permettant de représenter un cercle, un cylindre et une sphère. Le cercle aura une méthode retournant son périmètre et son aire. Le cylindre et la sphère hériteront du cercle et ajouteront la méthode volume.

- Modélisez en UML, un programme permettant de représenter des banques contenant plusieurs comptes bancaires. Chaque compte permet le dépôt, le retrait, le transfert vers un autre compte et peut retourner l'historique des opérations et le solde.

Chap 9 : La programmation orientée objet

Nous avons vu, dans le chapitre précédent, le concept et la modélisation des objets dans la programmation. Il est temps à présent d'implémenter les objets sous la forme de code. Python est un langage orienté objet. En effet, l'intégralité des types de données que nous avons manipulés jusqu'alors sont des objets. Il est temps à présent de mettre en œuvre nos connaissances en matière de programmation orientée objet pour simplifier la structure de nos programmes devenus très complexes. Ce nouveau paradigme nous permettra dans le chapitre suivant l'utilisation des interfaces graphiques.

Implémenter une classe

Nous allons regrouper toutes les méthodes, y compris les méthodes spéciales, avec le mot-clef `class` suivi du nom de la classe et de la classe mère qu'hérite la classe. Si la classe n'hérite d'aucune classe mère, inscrivez `object`. Voici l'implémentation de la classe `Personne` que nous avons abordée dans le chapitre précédent.

Voici le code :

```
class Personne(object):
        def __init__(self):
                self.nom = "Dubois"
                self.prenom = "Paul"
                self.lieuDeResidence = "Le Havre"
        def demenager(self, nouveauLieuDeResidence):
                self.lieuDeResidence = nouveauLieuDeResidence
```

Détaillons le code ci-dessus :

- Notre classe comporte deux fonctions, la fonction `__init__` et la fonction `demenager`. La fonction `__init__` est le **constructeur**, qui permet de définir les variables dans leur état initial.

Chaque classe est identifiée à partir d'un nom que vous donnez. Ce nom est à choisir afin qu'il respecte les consignes suivantes :

- Le nom de la classe doit être court, mais indiquer son contenu pour faciliter la lecture du code.
- Ne doit pas être un **nom réservé** (Voir ici).
- Ne doit comporter que des lettres (a → z et A → Z), des chiffres (0 → 9) et le caractère _ (*underscore*). Les **autres symboles sont interdits**.
- Python est **sensible à la casse**, c'est-à-dire que les majuscules et minuscules sont distinguées (exemple : `livre`, `Livre` et `LIVRE` sont des variables différentes).
- Il est vivement recommandé de nommer les variables en minuscule, commencer les mots avec une majuscule pour plus de lisibilité (exemple : `CarteAJouer`). **Dans ce cours, cette convention de nommage sera à appliquer.**

On observe l'apparition du mot `self` au début des variables qui indique que l'on travaille sur l'instance de la classe. Voici un petit tableau présentant la syntaxe des noms de variables dans les classes.

Syntaxe	Visibilité	Portée
`variable`	Privée	La fonction actuelle seulement
`self.variable`	Publique	Toute l'instance de l'objet
`self.__variable`	Privée	Toute l'instance de l'objet

Il est également possible de protéger une méthode en la faisant précéder du double symbole souligné (__). Cette méthode ne sera accessible uniquement à l'intérieur de la classe.

Une variable privée peut cependant être lue et modifiée *via* des méthodes spécifiques nommées **accesseurs** (lecture) et **mutateurs** (modification). Cela permet d'effectuer des modifications ou des contrôles sur les données avant qu'elles soient retournées ou modifiées. Il est **vivement recommandé** de procéder ainsi plutôt que d'offrir les variables publiquement. L'exemple suivant permet de mettre en œuvre cette protection des attributs :

```
class Personne(object):
        def __init__(self):
                self.__nom = "Dubois"
                self.__prenom = "Paul"
                self.__lieuDeResidence = "Le Havre"
        def __demenager(self, nouveauLieuDeResidence):
                self.__lieuDeResidence = nouveauLieuDeResidence
        def demanderNom(self):
                return("Je suis " + self.__prenom + " " + self.__nom)
```

Dans l'exemple ci-dessus, la fonction `__demenager` est accessible uniquement depuis l'objet, alors que la méthode `demanderNom` est accessible depuis tout le programme.

Utiliser un objet

Nous avons déjà utilisé des objets. Pour créer une instance d'une classe, nous allons saisir `instanceClasse = Classe()`. Il est possible de définir les valeurs par défaut des variables de l'instance en les communiquant en argument à la méthode constructeur. Voici un exemple mettant en œuvre l'instanciation d'une classe et le passage d'arguments pour le constructeur :

```
class Personne(object):
        def __init__(self, nom, prenom, lieuDeResidence):
                self.__nom = nom
                self.__prenom = prenom
                self.__lieuDeResidence = lieuDeResidence
        def __demenager(self, nouveauLieuDeResidence):
                self.__lieuDeResidence = nouveauLieuDeResidence
        def demanderNom(self):
                return("Je suis " + self.__prenom + " " + self.__nom)
personne1 = Personne("Dupont", "Clara", "Lille")
personne2 = Personne("Martin", "Julie", "Béziers")
print(personne1.demanderNom()) # Affiche "Je suis Clara Dupont"
print(personne2.demanderNom()) # Affiche "Je suis Julie Martin"
```

Les méthodes spéciales

Python permet de définir des méthodes spéciales qui permettent de faciliter l'utilisation des objets. Nous allons présenter une méthode déclenchée quand on cherche à convertir notre objet en chaîne de caractères (`str()` ou `print()`). Cette méthode doit se nommer `__str__`. Il est également possible de le faire pour récupérer un entier ou un réel avec respectivement les méthodes `__int__` et `__float__`. En voici un exemple :

```
class Personne(object):
        def __init__(self, nom, prenom, lieuDeResidence):
                self.__nom = nom
                self.__prenom = prenom
                self.__lieuDeResidence = lieuDeResidence
```

```
        def __demenager(self, nouveauLieuDeResidence):
                self.__lieuDeResidence = nouveauLieuDeResidence
        def __str__(self):
                return("Je suis " + self.__prenom + " " + self.__nom + " et
j'habite à " + self.__lieuDeResidence)
personne1 = Personne("Dupont", "Clara", "Lille")
personne2 = Personne("Martin", "Julie", "Béziers")
print(personne1)       # "Je suis Clara Dupont et j'habite à Lille"
print(personne2)       # "Je suis Julie Martin et j'habite à Béziers"
```

Il est également possible de définir des méthodes spéciales permettant de comparer les objets avec les opérateurs de comparaison que nous avons vus précédemment (voir ici). On appelle cela la **comparaison riche**. Voici la liste des méthodes spéciales associées aux opérateurs de comparaison.

Comparateur	Syntaxe	Méthode associée
a égal à b	`a == b`	`__eq__`
a différent de b	`a != b`	`__ne__`
a supérieur à b	`a > b`	`__gt__`
a supérieur ou égal à b	`a >= b`	`__ge__`
a inférieur à b	`a < b`	`__lt__`
a inférieur ou égal à b	`a <= b`	`__le__`

Ces méthodes doivent avoir comme argument l'objet avec lequel comparer l'objet actuel. Voici une implémentation de ces comparateurs riches :

```
class Personne(object):
        def __init__(self, nom, prenom, age):
                self.__nom = nom
                self.__prenom = prenom
                self.__age = age
        def getPrenom(self):
                return(self.__prenom)
        def getAge(self):
                return(self.__age)
        def __lt__(self, autrePersonne):
                return(self.__age < autrePersonne.getAge())
personne1 = Personne("Dupont", "Clara", 24)
personne2 = Personne("Martin", "Julie", 27)
if personne1 < personne2:      # Utilise personne1.__lt__(personne2)
        print(personne1.getPrenom() + " est la plus jeune. ")
else:
        print(personne2.getPrenom() + " est la plus jeune. ")
```

L'héritage

L'héritage se matérialise par la modification de l'argument lors de la définition de la classe. Nous allons reprendre le modèle UML des animaux vus précédemment.

Nous allons implémenter ces classes en utilisant les mécanismes d'héritage :

```
class Animaux(object):
        def __init__(self, nom):
                self.__nom = nom
                self.__nombrePattes = 0
                self.__position = 0
        def avancer(self, nombrePas):
                self.__position += nombrePas
                return("Je suis à la position " + str(self.__position))
class Chien(Animaux):
        def __init__(self, nom):
                Animaux.__init__(self, nom)
                self.__nombrePattes = 4
                self.aboyer()
        def aboyer(self):
                print("Wouf !")
class Poule(Animaux):
        def __init__(self, nom):
                Animaux.__init__(self, nom)
                self.__nombrePattes = 2
                self.caqueter()
        def caqueter(self):
                print("Cot !")
```

Les lignes 11 et 17 permettent de déclencher le constructeur de la classe mère en lui communiquant les arguments nécessaires.

Exercices

Nous allons écrire un programme permettant à un utilisateur de jouer au jeu du Yahtzee contre des joueurs pilotés par l'ordinateur.

Le Yahtzee est un jeu de société se jouant à l'aide de cinq dés à six faces et où le but est d'obtenir un maximum de points.

Lorsque c'est au tour d'un joueur de jouer, celui-ci doit essayer de réaliser des combinaisons détaillées ci-dessous à l'aide des cinq dés qu'il peut jeter trois fois par tour. Le joueur est libre quant au nombre de dés à jeter sauf pour le premier jet où il doit jeter tous les dés.

À la fin du tour, le joueur doit inscrire le score obtenu dans la grille des scores, même si le joueur doit inscrire un score nul. Chaque ligne de la grille est utilisable qu'une seule fois, il n'est pas possible de remplacer le score inscrit dans une ligne.

Une partie se compose de 13 tours afin de remplir les 13 lignes de la grille des scores. La grille des scores se divise en deux parties, la grille supérieure et la grille inférieure.

Nom	Combinaison	Points obtenus	
Partie supérieure			
As	Peu importe	1 × le nombre de	obtenus
Deux	Peu importe	2 × le nombre de	obtenus
Trois	Peu importe	3 × le nombre de	obtenus

Quatre	Peu importe	4 × le nombre de	obtenus
Cinq	Peu importe	5 × le nombre de	obtenus
Six	Peu importe	6 × le nombre de	obtenus
Prime (si la somme des lignes ci-dessus est ≥ à 63 points)		35 points	
Partie inférieure			
Brelan	Trois dés identiques	Somme de tous les dés	
Petite suite	Quatre dés consécutifs	30 points	
Grande suite	Cinq dés consécutifs	40 points	
Full	Trois dés identiques + deux dés identiques	25 points	
Carré	Quatre dés identiques	Somme de tous les dés	
Yahtzee	Cinq dés identiques	50 points	
Chance	Peu importe	Somme de tous les dés	

Nous modéliserons la partie, les joueurs, le plateau de jeu et les dés sous forme d'objets. On fera la distinction entre les joueurs humains et ordinateurs par deux classes partageant la même classe mère.

Chap 10 : Les interfaces graphiques

Nous avons vu jusqu'à présent comment concevoir des applications en mode console, à savoir, n'utilisant que le mode texte comme interface avec l'utilisateur. Or, la plupart des applications utilisées par le grand public offrent une interface graphique : une fenêtre comportant des boutons, des zones de texte, des cases à cocher, … Il est temps pour nous d'aborder comment parer nos applications d'une interface graphique et ainsi rendre leur utilisation beaucoup plus aisée.

Il existe plusieurs bibliothèques graphiques en Python telles que Tkinter qui offre un choix limité d'éléments graphiques et son aspect est assez austère. Nous allons utiliser la librairie PySide2 depuis le module éponyme qui offre la plupart des composants courants et qui est assez simple d'utilisation. De plus, elle s'adapte au thème configuré sur le système d'exploitation.

Application : un générateur de mot de passe

Nous allons travailler depuis un script permettant de générer des mots de passe aléatoirement pouvant comporter des minuscules, des majuscules, des chiffres et des symboles. La longueur du mot de passe est variable.

```
from random import choice
def genererMotDePasse(tailleMotDePasse=8, minuscules=True, majuscules=True,
chiffres=True, symboles=True):
        caracteres = ""
        if minuscules:
                caracteres += "abcdefghijklmnopqrstuvwxyz"
        if majuscules:
                caracteres += "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        if chiffres:
                caracteres += "0123456789"
        if symboles:
                caracteres += "&~#{([-|_\^@)=+$]}*%!/:.;?,"
        motDePasse = ""
        for i in range(tailleMotDePasse):
                motDePasse += choice(caracteres)
        return(motDePasse)
```

Nous allons réaliser l'interface suivante.

Notre interface permet de choisir quels jeux de caractères utiliser pour notre mot de passe à l'aide des cases à cocher. La glissière permet de faire varier la taille du mot de passe. Enfin, après avoir cliqué sur le bouton *Générer*, le mot de passe apparaît dans la zone de texte. Le bouton *Vers le presse-papier* copie le mot de passe généré dans le presse-papier et le bouton *Quitter* ferme l'application.

Les composants graphiques utilisés

Nous allons utiliser divers composants graphiques aussi nommés **widgets** (pour *Window Gadgets*). Nous utiliserons donc les classes suivantes :

La boîte de dialogue
`QDialog`
Les cases à cocher
`QCheckBox`
L'étiquette "Taille du mot de passe"
`QLabel`
Le champ de texte
`QLineEdit`
La glissière
`QSlider`
Les boutons
`QPushButton`

Nous allons donc écrire une classe correspondant à notre fenêtre en héritant la classe `QDialog` et y décrire l'ensemble des widgets comme des attributs de la classe dans le constructeur :

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # Les cases à cocher
                self.__caseMinuscules = QtWidgets.QCheckBox("Minuscules")
                self.__caseMajuscules = QtWidgets.QCheckBox("Majuscules")
                self.__caseChiffres = QtWidgets.QCheckBox("Chiffres")
                self.__caseSymboles = QtWidgets.QCheckBox("Symboles")
                # Les boutons
                self.__boutonQuitter = QtWidgets.QPushButton("Quitter")
                self.__boutonCopier = QtWidgets.QPushButton("Vers le presse-
papier")
                self.__boutonGenerer = QtWidgets.QPushButton("Générer")
                # Le champ de texte
                self.__champTexte = QtWidgets.QLineEdit("")
                # La glissière
                self.__glissiereTaille =
QtWidgets.QSlider(QtCore.Qt.Horizontal)
                # Le label
                self.__labelTaille = QtWidgets.QLabel("Taille du mot de
passe : " + str(self.__glissiereTaille.value()))
```

Nous allons à présent aborder le placement des widgets dans la boîte de dialogue. PySide nous propose plusieurs méthodes pour placer les widgets. La solution la plus simple est le placement sur une grille à l'aide de la classe `QGridLayout` : chaque widget occupe une case dans une grille. Il est cependant possible de faire en sorte qu'un widget occupe plusieurs lignes ou colonnes.

Voici notre maquette d'interface dont les widgets ont été répartis sur une grille.

Pour implémenter cela, nous allons créer un objet de la classe `QGridLayout`, puis ajouter les widgets créés précédemment avec la méthode `addWidget(widget, ligne, colonne)` avec `ligne` et `colonne`, le numéro de la ligne et de la colonne souhaitées. Enfin, nous définirons le layout comme étant l'élément central de la fenêtre avec `self.setLayout(layout)`.

Nous ajoutons donc à notre constructeur la portion de code suivante :

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # ...
                layout = QtWidgets.QGridLayout()
                layout.addWidget(self.__caseMajuscules, 0, 0)
                layout.addWidget(self.__labelTaille, 0, 1)
                layout.addWidget(self.__caseMinuscules, 0, 2)
                layout.addWidget(self.__caseChiffres, 1, 0)
```

```
                layout.addWidget(self.__glissiereTaille, 1, 1)
                layout.addWidget(self.__caseSymboles, 1, 2)
                layout.addWidget(self.__champTexte, 2, 1)
                layout.addWidget(self.__boutonQuitter, 3, 0)
                layout.addWidget(self.__boutonCopier, 3, 1)
                layout.addWidget(self.__boutonGenerer, 3, 2)
                self.setLayout(layout)
```

Nous allons terminer la préparation de notre fenêtre en modifiant le titre de la boîte de dialogue avec la méthode `self.setWindowTitle(titre)`. Nous allons également définir le minimum et le maximum de la glissière avec respectivement les méthodes `setMinimum` et `setMaximum`. Nous allons cocher par défaut la case minuscules et chiffres avec la méthode `setChecked`. Nous ajoutons les lignes suivantes à notre constructeur :

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # ...
                self.setWindowTitle("Générateur de mot de passe")
                self.__caseMinuscules.setChecked(True)
                self.__caseChiffres.setChecked(True)
                self.__glissiereTaille.setMinimum(8)
                self.__glissiereTaille.setMaximum(30)
```

Nous allons enfin ajouter une icône à notre application pour que celle-ci soit reconnaissable dans la barre des tâches. Nous ajoutons les trois lignes suivantes au constructeur de notre boîte de dialogue :

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                # ...
                icone = QtGui.QIcon()
                icone.addPixmap(QtGui.QPixmap("cadenas.svg"))
                self.setWindowIcon(icone)
```

Pour exécuter notre fenêtre, on écrit les lignes suivantes dans le programme principal qui permettent de créer une application Qt en fournissant les arguments de la ligne de commande (ligne 1), instancie notre fenêtre (ligne 2) et l'affiche (ligne 3) :

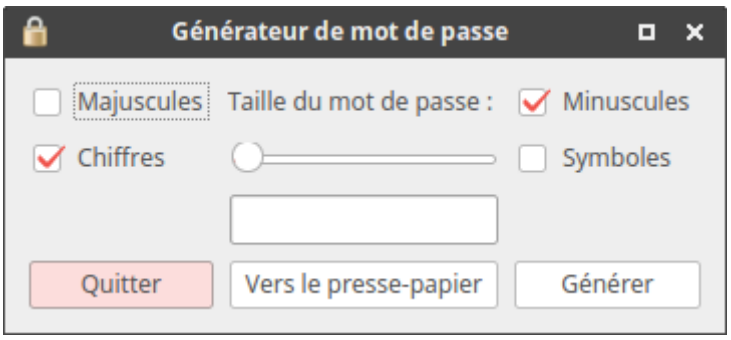
```
app = QtWidgets.QApplication(sys.argv)
dialog = MaFenetre()
dialog.exec_()
```

Voici le code complet :

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # Les cases à cocher
                self.__caseMinuscules = QtWidgets.QCheckBox("Minuscules")
                self.__caseMajuscules = QtWidgets.QCheckBox("Majuscules")
                self.__caseChiffres = QtWidgets.QCheckBox("Chiffres")
                self.__caseSymboles = QtWidgets.QCheckBox("Symboles")
                # Les boutons
                self.__boutonQuitter = QtWidgets.QPushButton("Quitter")
```

```
                self.__boutonCopier = QtWidgets.QPushButton("Vers le presse-
papier")
                self.__boutonGenerer = QtWidgets.QPushButton("Générer")
                # Le champ de texte
                self.__champTexte = QtWidgets.QLineEdit("")
                # La glissière
                self.__glissiereTaille =
QtWidgets.QSlider(QtCore.Qt.Horizontal)
                self.__glissiereTaille.setMinimum(8)
                self.__glissiereTaille.setMaximum(30)
                # Le label
                self.__labelTaille = QtWidgets.QLabel("Taille du mot de
passe : " + str(self.__glissiereTaille.value()))
                layout = QtWidgets.QGridLayout()
                layout.addWidget(self.__caseMajuscules, 0, 0)
                layout.addWidget(self.__labelTaille, 0, 1)
                layout.addWidget(self.__caseMinuscules, 0, 2)
                layout.addWidget(self.__caseChiffres, 1, 0)
                layout.addWidget(self.__glissiereTaille, 1, 1)
                layout.addWidget(self.__caseSymboles, 1, 2)
                layout.addWidget(self.__champTexte, 2, 1)
                layout.addWidget(self.__boutonQuitter, 3, 0)
                layout.addWidget(self.__boutonCopier, 3, 1)
                layout.addWidget(self.__boutonGenerer, 3, 2)
                self.setLayout(layout)
                self.setWindowTitle("Générateur de mot de passe")
                icone = QtGui.QIcon()
                icone.addPixmap(QtGui.QPixmap("cadenas.svg"))
                self.setWindowIcon(icone)
                self.__caseMinuscules.setChecked(True)
                self.__caseChiffres.setChecked(True)
app = QtWidgets.QApplication(sys.argv)
dialog = MaFenetre()
dialog.exec_()
```

Voici le résultat obtenu.



Les signaux

Nous avons obtenu une interface graphique mais celle-ci ne fonctionne pas : il est temps de relier les composants graphiques au code que nous avons écrit en début de chapitre.

Chaque widget de la fenêtre produit des signaux lorsqu'on l'utilise. Chaque signal peut être relié à un *slot* avec la méthode `connect` en fournissant en argument quelle fonction appeler lors de la réception du signal.

Pour illustrer cela, nous allons créer une méthode à notre objet `MaFenetre`, nommée `quitter`, et contenant la ligne `self.accept()` qui ferme la boîte de dialogue. Nous allons connecter le signal `clicked` (généré quand le bouton est cliqué) émis par le bouton *Quitter* à cette fonction :

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                # ...
                self.__boutonQuitter.clicked.connect(self.quitter)
        def quitter(self):
                self.accept()
```

Nous allons faire de même pour les boutons *Vers le presse-papier* et *Générer*. La copie vers le presse-papier nécessite la création d'un objet `QtGui.QApplication.clipboard()` et on y affecte, avec la méthode `setText(texte)` le contenu du champ de texte, lu avec la méthode `text()`.

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                # ...
                self.__boutonCopier.clicked.connect(self.copier)
        def copier(self):
                pressePapier = QtGui.QApplication.clipboard()
                pressePapier.setText(self.__champTexte.text())
```

De manière analogue, nous allons créer une fonction `generer` permettant d'appeler notre fonction `genererMotDePasse` en lui fournissant comme arguments la taille souhaitée, lue depuis la glissière avec la méthode `value()` et en choisissant les caractères à partir des cases à cocher avec la méthode `isChecked()` qui retourne `True` lorsqu'elles sont cochées. Enfin, la valeur retournée par la fonction sera définie comme texte du champ de texte avec la méthode `setText(texte)`.

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                # ...
                self.__boutonGenerer.clicked.connect(self.generer)
        def generer(self):
                tailleMotDePasse = self.__glissiereTaille.value()
                minuscules = self.__caseMinuscules.isChecked()
                majuscules = self.__caseMajuscules.isChecked()
                chiffres = self.__caseChiffres.isChecked()
                symboles = self.__caseSymboles.isChecked()

        self.__champTexte.setText(genererMotDePasse(tailleMotDePasse,
minuscules, majuscules, chiffres, symboles))
```

Nous allons améliorer notre programme en modifiant la valeur du label *Taille du mot de passe :* en ajoutant la valeur actuelle de la glissière. Pour cela, nous modifions la ligne créant notre label et ajouter une fonction déclenchée par la modification de la valeur de la glissière, à savoir le signal `valueChanged()`.

```
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                # ...
```

```
                self.__labelTaille = QtWidgets.QLabel("Taille du mot de
passe : " + str(self.__glissiereTaille.value()))
                # ...

        self.__glissiereTaille.valueChanged.connect(self.changerTailleMotDe
Passe)
        def changerTailleMotDePasse(self):
                self.__labelTaille.setText("Taille du mot de passe : " +
str(self.__glissiereTaille.value()))
```

Voici le code source complet de notre application :

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
from random import choice
def genererMotDePasse(tailleMotDePasse=8, minuscules=True, majuscules=True,
chiffres=True, symboles=True):
        caracteres = ""
        if minuscules:
                caracteres += "abcdefghijklmnopqrstuvwxyz"
        if majuscules:
                caracteres += "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        if chiffres:
                caracteres += "0123456789"
        if symboles:
                caracteres += "&~#{([-|_\^@)=+$]}*%!/:.;?,"
        motDePasse = ""
        for i in range(tailleMotDePasse):
                motDePasse += choice(caracteres)
        return(motDePasse)
class MaFenetre(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # Les cases à cocher
                self.__caseMinuscules = QtWidgets.QCheckBox("Minuscules")
                self.__caseMajuscules = QtWidgets.QCheckBox("Majuscules")
                self.__caseChiffres = QtWidgets.QCheckBox("Chiffres")
                self.__caseSymboles = QtWidgets.QCheckBox("Symboles")
                # Les boutons
                self.__boutonQuitter = QtWidgets.QPushButton("Quitter")
                self.__boutonCopier = QtWidgets.QPushButton("Vers le presse-
papier")
                self.__boutonGenerer = QtWidgets.QPushButton("Générer")
                # Le champ de texte
                self.__champTexte = QtWidgets.QLineEdit("")
                # La glissière
                self.__glissiereTaille =
QtWidgets.QSlider(QtCore.Qt.Horizontal)
                self.__glissiereTaille.setMinimum(8)
                self.__glissiereTaille.setMaximum(30)
                # Le label
                self.__labelTaille = QtWidgets.QLabel("Taille du mot de
passe : " + str(self.__glissiereTaille.value()))
                layout = QtWidgets.QGridLayout()
                layout.addWidget(self.__caseMajuscules, 0, 0)
                layout.addWidget(self.__labelTaille, 0, 1)
                layout.addWidget(self.__caseMinuscules, 0, 2)
                layout.addWidget(self.__caseChiffres, 1, 0)
                layout.addWidget(self.__glissiereTaille, 1, 1)
                layout.addWidget(self.__caseSymboles, 1, 2)
                layout.addWidget(self.__champTexte, 2, 1)
```

```
                layout.addWidget(self.__boutonQuitter, 3, 0)
                layout.addWidget(self.__boutonCopier, 3, 1)
                layout.addWidget(self.__boutonGenerer, 3, 2)
                self.setLayout(layout)
                self.setWindowTitle("Générateur de mot de passe")
                icone = QtGui.QIcon()
                icone.addPixmap(QtGui.QPixmap("cadenas.svg"))
                self.setWindowIcon(icone)
                self.__caseMinuscules.setChecked(True)
                self.__caseChiffres.setChecked(True)
                self.__boutonQuitter.clicked.connect(self.quitter)
                self.__boutonCopier.clicked.connect(self.copier)
                self.__boutonGenerer.clicked.connect(self.generer)

        self.__glissiereTaille.valueChanged.connect(self.changerTailleMotDe
Passe)
        def quitter(self):
                self.accept()
        def copier(self):
                pressePapier = QtWidgets.QApplication.clipboard()
                pressePapier.setText(self.__champTexte.text())
        def generer(self):
                tailleMotDePasse = self.__glissiereTaille.value()
                minuscules = self.__caseMinuscules.isChecked()
                majuscules = self.__caseMajuscules.isChecked()
                chiffres = self.__caseChiffres.isChecked()
                symboles = self.__caseSymboles.isChecked()

        self.__champTexte.setText(genererMotDePasse(tailleMotDePasse,
minuscules, majuscules, chiffres, symboles))
        def changerTailleMotDePasse(self):
                self.__labelTaille.setText("Taille du mot de passe : " +
str(self.__glissiereTaille.value()))
app = QtWidgets.QApplication(sys.argv)
dialog = MaFenetre()
dialog.exec_()
```

Les widgets courants PySide

Présentation

L'ensemble des widgets présentés ici héritent de la classe `QWidget` qui offre une méthode `setEnabled`, qui permet d'activer ou de désactiver le widget. Nous allons découvrir dans cette section plusieurs widgets courants proposés par PySide.

Le champ de texte `QLineEdit`

Le champ de texte `QLineEdit` permet à l'utilisateur de lire et de modifier une chaîne de caractères. En voici les méthodes courantes :

Méthode	Description
`text()`	Retourne le texte contenu dans le champ de texte.
`setText(texte)`	Modifie le contenu du champ de texte par le texte fourni en argument.
`clear()`	Efface le contenu du champ de texte.

`setMaxLength(taille)`	Définit la taille maximale du champ de texte.
`maxLength(taille)`	Retourne la taille maximale du champ de texte.
`copy()`	Copie le contenu du champ de texte dans le presse-papier.
`paste()`	Colle le contenu du presse-papier dans le champ de texte.
`setEchoMode(mode)`	Modifie l'affichage du contenu du champ de texte sans modifier son contenu : • `QLineEdit.Normal` : Affiche les caractères normalement (par défaut). • `QLineEdit.NoEcho` : Rien n'est affiché. • `QLineEdit.Password` : Affiche des étoiles à la place des caractères (pour les mots de passe) • `QLineEdit.PasswordEchoOnEdit` : Affiche les caractères normalement tant que le champ est sélectionné. Sinon, affiche des étoiles.
`setCompleter(completer)`	Permet de définir une instance de QCompleter pour fournir de l'auto-complétion.
`setInputMask(masque)`	Permet de configurer le format de données attendu avec une chaîne de caractères. Par exemple : • `000.000.000.000` : Adresse IPv4 • `HH:HH:HH:HH:HH:HH` : Adresse MAC • `00-00-0000` : Date au format JJ-MM-AAAA. Il est possible de rendre le masque visible en lui ajoutant "`;`" suivi d'un caractère de remplacement. Par exemple, "`00-00-0000;_`" affichera "__-__-____" tant qu'il ne sera pas rempli.

Voici quelques signaux proposés par la classe `QLineEdit` :

Signal	Déclencheur
`textChanged`	Lorsque le texte change (par l'utilisateur ou par le programme).
`textEdited`	Lorsque le texte est changé par l'utilisateur.
`cursorPositionChanged`	Lorsque le curseur est déplacé.
`returnPressed`	Lorsque la touche `Entrée` est pressée.
`editingFinished`	Lorsque le champ perd le focus ou la touche `Entrée` est pressée.

La classe `QAbstractButton`

La classe `QAbstractButton` est une classe regroupant différents boutons (comme `QPushButton`, `QCheckBox` …). Il est cependant impossible de créer un objet de cette classe.

Méthode	Description
`setIcon(icone)`	Permet d'ajouter une icône avec une instance de la classe `QtGui.QIcon` au bouton.
`setShortcut(raccourci)`	Associe au bouton un raccourci clavier sous la forme d'une chaîne de caractères (exemple : `CTRL` + `C`).
`setCheckable()`	Permet de rendre le bouton bistable (il maintient l'état après le clic). Les `QRadioButton` et `QCheckBox` sont bistables par défaut.
`setChecked()`	Permet de valider un bouton.
`isChecked()`	Retourne l'état du bouton.

Voici quelques signaux proposés par la classe `QAbstractButton` :

Signal	Déclencheur
`clicked`	Lorsque le bouton est cliqué.
`pressed`	Lorsque le bouton est appuyé.
`released`	Lorsque le bouton est relâché.
`toogled`	Lorsque le bouton change d'état comme par exemple les boutons de barre d'outils.

La case à cocher `QCheckBox`

La case à cocher permet de sélectionner entre 0 et plusieurs choix parmi l'ensemble de cases disponibles. On y retrouve les méthodes et signaux héritées de `QAbstractButton`. Par défaut les cases à cocher sont bistables.

Le bouton radio `QRadioButton`

Les boutons radios permettent un choix exclusif parmi plusieurs options présentes dans le même conteneur (layout ou widget). Un seul peut être coché à la fois. Ils héritent également des méthodes et signaux de la classe `QAbstractButton`. Le signal `toggled` permet de vérifier qu'un bouton radio change d'état.

Le bouton poussoir `QPushButton`

Le bouton poussoir `QPushButton` permet de déclencher une action lors de son clic. Ce widget hérite également de la classe `QAbstractButton`. Il est possible d'associer un menu (instance de la classe `QMenu`) avec la méthode `setMenu(menu)`.

La boîte de sélection `QComboBox`

La boîte de sélection `QComboBox` permet de choisir un élément parmi une liste. Voici les méthodes principales de cette classe :

Méthode	Description

`addItem(chaine)` `addItem(icone, chaine)`	Permet d'ajouter une chaîne de caractères (avec ou sans icône) à la liste de choix.
`addItems(liste)`	Permet d'ajouter une liste de chaîne de caractères à la liste de choix.
`currentIndex()`	Retourne l'index de l'élément actuellement sélectionné.
`currentText()`	Retourne la chaîne de caractères actuellement sélectionnée.
`setEditable()`	Permet de modifier ou non les éléments de la boîte de sélection. Par défaut, cela n'est pas possible.
`insertItem(l, chaine)` `insertItems(l, liste)`	Permet d'ajouter un ou plusieurs éléments pendant l'exécution du programme à l'index `l`.
`insertSeparator()`	Permet de grouper les éléments en ajoutant un séparateur entre ceux-ci.
`clear()`	Efface les éléments contenus.

Voici la liste des signaux proposés par `QComboBox` :

Signal	Déclencheur
`activated`	Lorsque l'utilisateur interagit avec.
`currentIndexChanged`	Lorsque l'élément sélectionné change (par le programme ou l'utilisateur). Retourne l'index de l'élément sélectionné.
`highlighted`	Retourne l'index de l'élément surligné.
`editTextChanged`	Lorsque l'utilisateur modifie le contenu de la boîte de dialogue.

Les champs numériques `QSpinBox` et `QDoubleSpinBox`

Les champs numériques `QSpinBox` et `QDoubleSpinBox` forcent l'utilisateur à saisir des données numériques. Le champ `QSpinBox` n'accepte que les valeurs entières, et le champ `QDoubleSpinBox` les valeurs décimales. Voici les méthodes de ces champs :

Méthode	Description

`setMinimum(minimum)`
`setMaximum(maximum)`
`setRange(min, max)`
	Définissent le minimum et le maximum autorisés par le champ.

`setSingleStep`	Fixe le pas d'incrémentation.
`setSuffix(chaine)` `setPrefix(chaine)`	Ajoutent un suffixe ou un préfixe au champ pour plus de lisibilité (exemple : €, £, litres, km, …).
`setValue(valeur)`	Définit la valeur du champ.
`value()`	Retourne la valeur du champ.
`setDecimals(nbDecimales)`	*Pour `QDoubleSpinBox`*. Permet de définir le nombre de décimales à l'affichage.

Ces deux champs numériques offrent un seul signal : `valueChanged` qui est émit quand l'utilisateur change la valeur contenue. La valeur retournée est la valeur du champ. Pour `QDoubleSpinBox`, la chaîne de caractères est codée en fonction de la langue (en France, on utilise une virgule (,) comme séparateur de décimale).

Les champs horodateurs `QDateEdit`, `QTimeEdit` et `QDateTimeEdit`

Ces champs sont conçus pour saisir des données temporelles. Le nom des méthodes et signaux dépendent du type de données (`QDateEdit` pour la date, `QTimeEdit` pour l'heure et `QDateTimeEdit` pour la date et l'heure). Voici les méthodes de ces classes :

Méthode	Description

`date()`
`time()`
`dateTime()`
	Retournent la valeur du champ sous la forme d'un objet `QDate`, `QTime` ou `QDateTime`.

`setDate()`
`setTime()`
`setDateTime()`
	On remplit le champ avec les objets `QDate`, `QTime` ou `QDateTime`.

`setMinimumDate()`
`setMinimumTime()`
`setMinimumDateTime()`
	Définissent le minimum du champ avec les objets `QDate`, `QTime` ou `QDateTime`.

`setMaximumDate()`
`setMaximumTime()`
`setMaximumDateTime()`
	Définissent le maximum du champ avec les objets `QDate`, `QTime` ou `QDateTime`.

`setCalendarPopup()`	Permet d'afficher un calendrier pour les objets manipulant des dates.

Les signaux émis dépendent des valeurs qui ont changés :

Signal	Déclencheur

`dateChanged`
`timeChanged`
`dateTimeChanged`
	Déclenchés lors de la modification de la valeur.

La zone de texte `QTextEdit`

À l'instar de la classe `QLineEdit`, ce widget permet d'éditer du texte, mais offre une zone d'édition plus grande et permet la mise en forme du contenu au format HTML. Voici les méthodes usuelles :

Méthode	Description
`toPlainText()`	Retourne le texte contenu dans le champ de texte.
`setText(texte)`	Modifie le texte du champ par celui en argument.
`toHtml()`	Retourne le code HTML contenu dans le champ de texte.

`setHtml(texte)`	Modifie le contenu du champ de texte par le code HTML fourni en argument.
`clear()`	Efface le contenu du champ de texte.
`copy()`	Copie le contenu du champ de texte dans le presse-papier.
`paste()`	Colle le contenu du presse-papier dans le champ de texte.
`undo()`	Annule la dernière opération.
`redo()`	Refait la dernière opération annulée.

Voici quelques signaux proposés par la classe `QLineEdit` :

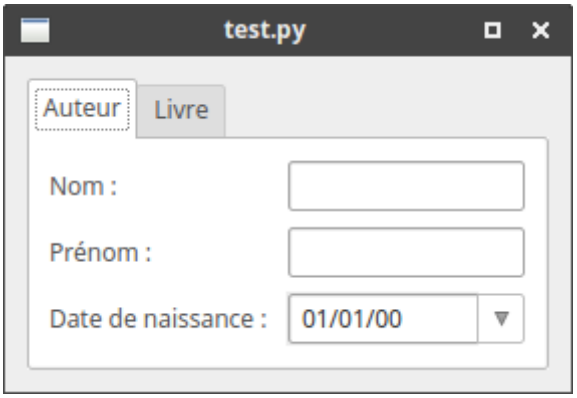
Signal	Déclencheur
`textChanged`	Lorsque le texte change (par l'utilisateur ou par le programme).
`cursorPositionChanged`	Lorsque le curseur s'est déplacé.

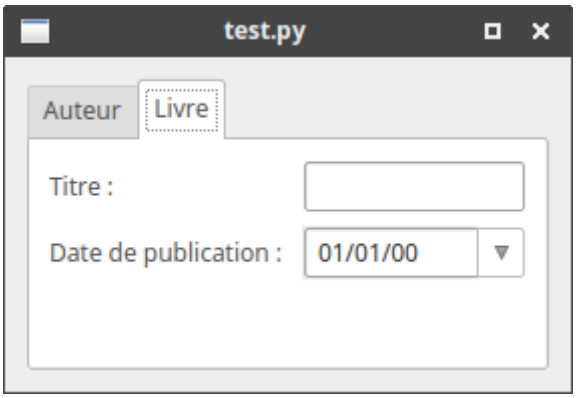
La boîte à onglets `QTabWidget`

La classe `QTabWidget` permet de regrouper des widgets dans différents onglets. Voici les différentes méthodes offertes par cette classe :

Méthode	Description
`addTab(widget, nom)`	Permet d'ajouter un onglet contenant un widget et dont le nom et le widget sont passés en argument.
`insertTab(widget, nom)`	Permet d'ajouter un onglet contenant un widget et dont le nom et le widget sont passés en argument pendant l'exécution du programme.
`tabPosition()`	Retourne l'indice de l'onglet actuellement sélectionné.

Le signal `currentChanged` est émis lorsque l'utilisateur change d'onglet. Voici un exemple de mise en œuvre de la boîte à onglets.





Voici le code source de l'exemple ci-dessous :

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class Dialog(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                # Les champs
                self.__champTexteNomAuteur = QtWidgets.QLineEdit("")
                self.__champTextePrenomAuteur = QtWidgets.QLineEdit("")
                self.__champDateNaissanceAuteur = QtWidgets.QDateEdit()
                self.__champDateNaissanceAuteur.setCalendarPopup(True)
                self.__champTexteTitreLivre = QtWidgets.QLineEdit("")
                self.__champDatePublication = QtWidgets.QDateEdit()
                self.__champDatePublication.setCalendarPopup(True)
                # Les widgets
                self.__widgetAuteur = QtWidgets.QWidget()
                self.__widgetLivre = QtWidgets.QWidget()
                # Les layouts des onglets
                self.__layoutAuteur = QtWidgets.QFormLayout()
                self.__layoutAuteur.addRow("Nom : ",
self.__champTexteNomAuteur)
                self.__layoutAuteur.addRow("Prénom : ",
self.__champTextePrenomAuteur)
                self.__layoutAuteur.addRow("Date de naissance : ",
self.__champDateNaissanceAuteur)
                self.__widgetAuteur.setLayout(self.__layoutAuteur)
                self.__layoutLivre = QtWidgets.QFormLayout()
                self.__layoutLivre.addRow("Titre : ",
self.__champTexteTitreLivre)
                self.__layoutLivre.addRow("Date de publication : ",
self.__champDatePublication)
                self.__widgetLivre.setLayout(self.__layoutLivre)
                # La boîte à onglets
                self.__tabWidget = QtWidgets.QTabWidget()
                self.__tabWidget.addTab(self.__widgetAuteur, "Auteur")
                self.__tabWidget.addTab(self.__widgetLivre, "Livre")
                # Le layout final
                self.__mainLayout = QtWidgets.QVBoxLayout()
                self.__mainLayout.addWidget(self.__tabWidget)
                self.setLayout(self.__mainLayout)
app = QtWidgets.QApplication(sys.argv)
dialog = Dialog()
dialog.exec_()
```

La boîte à regroupement `QGroupBox`

Cette classe permet de regrouper des widgets dans une boîte avec un titre. Elles sont souvent utilisées pour organiser les choix proposés. Voici les méthodes offertes par cette classe :

Méthode	Description
`setLayout(layout)`	Définit le layout passé en argument comme le layout utilisé pour cette instance.
`setChecked(bool)`	Permet de créer une boîte de regroupement optionnelle.
`isChecked()`	Pour les boîtes de regroupement optionnelles, retourne si le groupe a été coché.

La zone de défilement `QScrollArea`

Les zones de défilement sont utilisées pour l'affichage de widgets de grande taille tels que des images, des tableaux ou des zones de texte. Elles font apparaître des ascenseurs pour pouvoir faire défiler les zones non visibles. Cette classe ne contient qu'un seul widget placé avec la méthode `setWidget(widget)`.

Le panneau séparé `QSplitter`

Le panneau séparé permet de placer plusieurs widgets côte à côte séparés par un séparateur pouvant être déplacé par l'utilisateur. La géométrie des widgets dépend donc de la position de ce séparateur. Il est possible d'ajouter plusieurs composants. Voici les méthodes de cette classe :

Méthode	Description
`addWidget(widget)`	Ajoute un widget aux panneaux.
`setStretchFactor(index, entier)`	Permet de définir un cœfficient de la taille occupée par chaque widget.
`setOrientation(arg)`	Permet de modifier l'orientation du panneau séparé. Voici les arguments possibles : • `Qt.Vertical` : Empilement vertical (par défaut) • `Qt.Horizontal` : Empilement horizontal.

L'affichage en liste `QListWidget`

Cette classe permet l'affichage d'éléments sous forme d'une liste. Elle permet la vue en liste (par défaut) ou par icônes. Voici les méthodes offertes par la classe `QListWidget` :

Méthode	Description
`addItem(chaine)`	Ajoute un élément à la liste (texte seul).
`addItem(item)`	Ajoute un objet `QListWidgetItem` à la liste (texte et icône).
`insertItem(l, chaine)`	

`insertItems(l, liste)`	Permet d'ajouter un ou plusieurs éléments à la position pendant l'exécution du programme.
`setViewMode(arg)`	Permet de modifier le mode d'affichage de la liste : • `QtListView.ListMode` : Vue en liste (par défaut) • `QtListView.IconMode` : Vue en icônes
`currentRow()`	Retourne l'index de la ligne sélectionnée.
`currentItem()`	Retourne l'objet `QListWidgetItem` correspondant à la ligne sélectionnée.
`clear()`	Efface les entrées présentes.

Cette classe génère de nombreux signaux dont en voici un extrait :

Signal	Déclencheur
`currentItemCanged`	Déclenché lors du changement d'éléments sélectionnés. Retourne l'élément précédemment sélectionné et l'élément nouvellement sélectionné.
`itemActivated`	Déclenché lors de la sélection d'un élément. Retourne l'élément sélectionné.
`itemClicked`	Déclenché lors du clic d'un élément. Retourne l'élément cliqué.
`itemDoubleClicked`	Déclenché lors du double-clic d'un élément. Retourne l'élément double-cliqué.

L'affichage en tableau `QTableWidget`

Il est possible d'afficher des données sous la forme d'une table. Chaque cellule contient une instance de la classe `QtWidgets.QTableWidgetItem`. Lors de sa création, on passe le nombre de lignes et de colonnes en argument du constructeur. Voici les méthodes usuelles :

Méthode	Description
`setItem(ligne, colonne, item)`	Définit la cellule spécifiée par sa ligne et sa colonne. L'item passé en argument est un objet `QTableWidgetItem(chaine).`
`setHorizontalHeaderLabels (liste)`	Modifie les en-têtes des colonnes de la table.
`setVerticalHeaderLabels (liste)`	Modifie les en-têtes des lignes de la table.
`setRowCount(nombre)`	Définit le nombre de lignes passées en argument.
`setColumnCount(nombre)`	Définit le nombre de colonnes passées en argument.
`rowCount()`	Retourne le nombre de lignes de la table.
`columnCount()`	Retourne le nombre de colonnes de la table.

Voici quelques signaux proposés :

Signal	Déclencheur
`currentItemCanged`	Déclenché lors du changement d'éléments sélectionnés. Retourne l'élément précédemment sélectionné et l'élément nouvellement sélectionné.
`itemActivated`	Déclenché lors de la sélection d'un élément. Retourne l'élément sélectionné.
`itemClicked`	Déclenché lors du clic d'un élément. Retourne l'élément cliqué.
`itemDoubleClicked`	Déclenché lors du double-clic d'un élément. Retourne l'élément double-cliqué.
`currentCellCanged`	Déclenché lors du changement d'éléments sélectionnés. Retourne les coordonnées de l'élément précédemment sélectionné et les coordonnées de l'élément nouvellement sélectionné.
`cellActivated`	Déclenché lors de la sélection d'un élément. Retourne les coordonnées de l'élément sélectionné.
`cellClicked`	Déclenché lors du clic d'un élément. Retourne les coordonnées de l'élément cliqué.
`cellDoubleClicked`	Déclenché lors du double-clic d'un élément. Retourne les coordonnées de l'élément double-cliqué.

L'affichage en arbre `QTreeWidget`

Il est possible de représenter les données sous la forme d'un arbre. Chaque élément de l'arbre est une instance de la classe `QtWidgets.QTreeWidgetItem` avec en argument la chaîne de caractères de l'élément. Pour ajouter un élément enfant, on utilise la méthode `addChild(item)` avec comme argument l'item enfant implémentant `QTreeWidgetItem`. On définit l'élément racine de l'arbre avec la méthode `addTopLevelItem(item)`. Les signaux sont identiques à la classe `QListWidget`.

La boîte de dialogue `QInputDialog`

Pour simplifier nos programmes, il existe la classe `QInputDialog` qui permet de demander une donnée à l'utilisateur. Cette boîte de dialogue comporte le champ à saisir, un titre, un message, un bouton pour valider et un bouton pour annuler. Cette classe s'utilise comme suit :

```
age = QtWidgets.QInputDialog.getInt(parent, "Votre âge", "Entrez votre âge
: ")
```

Les méthodes offertes permettent de déterminer le type de données à demander :

Méthode	Description
`getInt(parent, titre, message, valeur)`	Demande un entier à l'utilisateur.
`getDouble(parent, titre, message, valeur)`	Demande un réel à l'utilisateur.

`getItem(parent, titre, message, listeValeurs, editable)`	Demande un élément parmi la liste à l'utilisateur. Peut être modifiable si `editable=True`.
`getText(parent, titre, message)`	Demande une chaîne à l'utilisateur. Peut être caché si `echo=QtWidgets.QLineEdit.Password`.

Le sélectionneur de couleur `QColorDialog`

Cette boîte de dialogue permet de choisir une couleur parmi un nuancier. Voici les méthodes offertes par cette classe :

Méthode	Description
`selectedColor()`	Retourne la couleur choisie par l'utilisateur (classe `QColor`).
`setCurrentColor(couleur)`	Définit la couleur de la boîte de dialogue.
`getColor()`	Ouvre la boîte de dialogue pour choisir la couleur.

Voici les signaux proposés par cette classe :

Signal	Déclencheur
`colorSelected`	Déclenché lors de la sélection d'une couleur. Retourne la couleur sélectionnée.
`currentColorChanged`	Déclenché lors du changement de couleur choisie. Retourne la couleur sélectionnée.

Le sélectionneur de fontes `QFontDialog`

Cette boîte de dialogue permet de choisir une fonte parmi les polices installées sur le système. Voici les méthodes offertes par cette classe :

Méthode	Description
`selectedFont()`	Retourne la fonte choisie par l'utilisateur (classe `QFont`).
`setCurrentFont(fonte)`	Définit la fonte de la boîte de dialogue.
`getFont()`	Ouvre la boîte de dialogue pour choisir la fonte.

Voici les signaux proposés par cette classe :

Signal	Déclencheur
`fontSelected`	Déclenché lors de la sélection d'une fonte. Retourne la fonte sélectionnée.
`currentFontChanged`	Déclenché lors du changement de fonte choisie. Retourne la fonte sélectionnée.

Le sélectionneur de fichier `QFileDialog`

La boîte de dialogue `QFileDialog` permet de choisir un fichier ou un répertoire. Voici les méthodes permettant de créer ou choisir un fichier ou un répertoire. Toutes les méthodes présentées retournent le chemin du fichier et le filtre choisis dans le cas des fichiers :

Méthode	Description
`getExistingDirectory()`	Permet de sélectionner un répertoire.
`getOpenFileName()`	Permet de sélectionner un fichier à ouvrir.
`getOpenFileNames()`	Permet de sélectionner un ou plusieurs fichiers à ouvrir.
`getSaveFileName()`	Permet de sauvegarder un fichier.

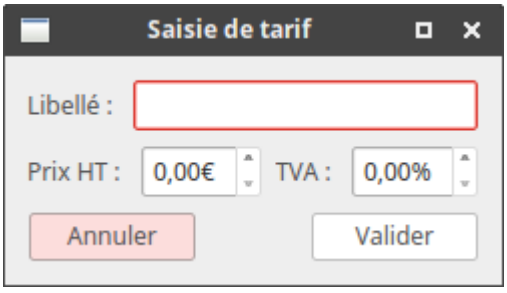
Les layouts

Les layouts permettent de placer les widgets dans les conteneurs (fenêtre, `QTabWidget`, …). Voici les layouts proposés par PySide.

Le placement sur une ligne `QHBoxLayout` et sur une colonne `QVBoxLayout`

Ces layouts simplifient la mise en place des widgets en les juxtaposant (verticalement avec `QVBoxLayout` et horizontalement avec `QHBoxLayout`) avec la méthode `addWidget(widget)`. Il est cependant possible d'ajouter un layout au sein du layout actuel avec la méthode `addLayout(layout)`. Il est enfin possible d'ajouter un espace élastique qui occupe tout l'espace restant lors du redimensionnement de la fenêtre avec `addStretch`.

Voici un exemple de mise en œuvre de ces layouts.



Voici le code source permettant d'obtenir ce résultat :

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class Dialog(QtWidgets.QDialog):
        def __init__(self, parent=None):
                QtWidgets.QDialog.__init__(self,parent)
                self.setWindowTitle("Saisie de tarif")
                self.__labelLibelle = QtWidgets.QLabel("Libellé : ")
                self.__champLibelle = QtWidgets.QLineEdit("")
                self.__layoutLibelle = QtWidgets.QHBoxLayout()
                self.__layoutLibelle.addWidget(self.__labelLibelle)
                self.__layoutLibelle.addWidget(self.__champLibelle)
                self.__labelPrixHT = QtWidgets.QLabel("Prix HT : ")
                self.__champPrixHT = QtWidgets.QDoubleSpinBox()
                self.__champPrixHT.setSuffix("€")
```

```
                self.__labelTauxTVA = QtWidgets.QLabel("TVA : ")
                self.__champTauxTVA = QtWidgets.QDoubleSpinBox()
                self.__champTauxTVA.setSuffix("%")
                self.__layoutPrix = QtWidgets.QHBoxLayout()
                self.__layoutPrix.addWidget(self.__labelPrixHT)
                self.__layoutPrix.addWidget(self.__champPrixHT)
                self.__layoutPrix.addWidget(self.__labelTauxTVA)
                self.__layoutPrix.addWidget(self.__champTauxTVA)
                self.__boutonAnnuler = QtWidgets.QPushButton("Annuler")
                self.__boutonValider = QtWidgets.QPushButton("Valider")
                self.__layoutBoutons = QtWidgets.QHBoxLayout()
                self.__layoutBoutons.addWidget(self.__boutonAnnuler)
                self.__layoutBoutons.addStretch()
                self.__layoutBoutons.addWidget(self.__boutonValider)
                self.__layoutPrincipal = QtWidgets.QVBoxLayout()
                self.__layoutPrincipal.addLayout(self.__layoutLibelle)
                self.__layoutPrincipal.addLayout(self.__layoutPrix)
                self.__layoutPrincipal.addLayout(self.__layoutBoutons)
                self.setLayout(self.__layoutPrincipal)
app = QtWidgets.QApplication(sys.argv)
dialog = Dialog()
dialog.exec_()
```

Le placement en formulaire `QFormLayout`

Le placement en formulaire permet de simplifier le code source de votre application en proposant un layout mettant en forme les widgets en formulaire. Cette mise en forme est divisée en deux colonnes, avec à gauche les labels associés aux widgets situés à droite.

Le layout possède la méthode `addRow(chaine, widget)` qui crée le label associé aux widgets avec comme texte la chaîne passée en argument.

Le placement en grille `QGridLayout`

Nous avons déjà vu ce type de layout au début de ce chapitre. Nous ajouterons comment faire en sorte qu'un widget ou un layout occupent plusieurs lignes ou colonnes. Pour cela, il faut ajouter deux arguments permettant de spécifier le nombres de lignes et de colonnes occupées.

Voici un exemple de mise en œuvre de cette fusion de cellules.

```
layout = QtWidgets.QGridLayout()
layout.addWidget(widget0, 0 ,0, 1, 2)
layout.addWidget(widget1, 0 ,2)
layout.addWidget(widget2, 1 ,0, 2, 1)
layout.addWidget(widget3, 1 ,1)
layout.addWidget(widget4, 1 ,2)
layout.addWidget(widget5, 2 ,1)
layout.addWidget(widget6, 2 ,2)
```

Code source

Widget 0		Widget 1
Widget 2	Widget 3	Widget 4
	Widget 5	Widget 6

Rendu

Les fenêtres principales

Les fenêtres des applications sont généralement construites avec la classe `QMainWindow` qui gère automatiquement les barres d'outils, les menus, les barres d'états …

Application : le bloc-notes

Nous allons étudier cette classe en créant un bloc-notes permettant d'ouvrir, d'éditer et d'enregistrer un fichier. Notre application s'articulera autour d'une zone de texte. Voici un schéma de la fenêtre à concevoir.

Nous allons instancier la classe `QMainWindow` qui est affichée en appelant la méthode `show()` ou la méthode `setVisible(bool)`. Dans cette partie, nous aborderons uniquement la construction de la fenêtre avec la barre d'outils et de menu.

```
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                # La fenêtre sera décrite ici
                self.show()
app = QtWidgets.QApplication(sys.argv)
fenetre = BlocNotes()
app.exec_()
```

Nous allons créer la zone de texte centrale et la définir comme widget central de la fenêtre :

```
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                self.setWindowTitle("Bloc-notes")
                self.__zoneTexte = QtWidgets.QTextEdit()
                self.setCentralWidget(self.__zoneTexte)
                # ...
```

Pour définir un layout comme widget central, créez une instance `QtWidgets.QWidget`, affectez votre layout à ce widget avec la méthode `setLayout` et définissez ce widget comme widget central.

Les actions `QAction`

Dans notre application, nous avons la barre de menu et la barre d'outils qui comportent les mêmes actions. La classe `QAction` permet de regrouper tous ces éléments graphiques et les associer à la même méthode, en y ajoutant une icône et un raccourci clavier. Cette classe peut être utilisée de trois manières différentes :

- `QAction(parent)`
- `QAction(chaine, parent)` : La chaîne décrit l'action. Elle sera utilisée dans les menus.
- `QAction(icone, chaine, parent)` : Ajoute une icône de la classe `QIcon` utilisée dans les menus et les barres d'outils.

Voici quelques méthodes offertes par cette classe :

Méthode	Description
`setStatusTip(chaine)`	Définit le texte affiché dans la barre d'actions des fenêtres.
`setShortcuts(raccourci)`	Définit le raccourci clavier. L'argument est une instance de la classe `QKeySequence`.

Les actions génèrent le signal `triggered` lorsqu'elles sont activées. Voici les actions utilisées pour notre application :

- Nouveau
- Ouvrir
- Enregistrer
- Enregistrer sous
- Quitter
- Annuler
- Refaire
- Couper
- Copier
- Coller

```
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                # ...
                self.__actionNew = QtWidgets.QAction(QtGui.QIcon("document-
new.svg"), "Nouveau", self)
                self.__actionNew.setShortcuts(QtGui.QKeySequence.New)
                self.__actionNew.setStatusTip("Nouveau document")
                self.__actionOpen = QtWidgets.QAction(QtGui.QIcon("document-
open.svg"), "Ouvrir", self)
                self.__actionOpen.setShortcuts(QtGui.QKeySequence.Open)
                self.__actionOpen.setStatusTip("Ouvrir un document
existant")
                self.__actionSave = QtWidgets.QAction(QtGui.QIcon("document-
save.svg"), "Enregistrer", self)
                self.__actionSave.setShortcuts(QtGui.QKeySequence.Save)
                self.__actionSave.setStatusTip("Enregistrer le document")
                self.__actionSaveAs =
QtWidgets.QAction(QtGui.QIcon("document-save-as.svg"), "Enregistrer sous",
self)
                self.__actionSaveAs.setShortcuts(QtGui.QKeySequence.SaveAs)
                self.__actionSaveAs.setStatusTip("Enregistrer le document
sous")
                self.__actionQuit =
QtWidgets.QAction(QtGui.QIcon("exit.svg"), "Quitter", self)
                self.__actionQuit.setShortcuts(QtGui.QKeySequence.Quit)
```

```
                self.__actionQuit.setStatusTip("Quitter l'application")
                self.__actionUndo =
QtWidgets.QAction(QtGui.QIcon("undo.svg"), "Annuler", self)
                self.__actionUndo.setShortcuts(QtGui.QKeySequence.Undo)
                self.__actionUndo.setStatusTip("Annuler la dernière
opération")
                self.__actionRedo =
QtWidgets.QAction(QtGui.QIcon("redo.svg"), "Refaire", self)
                self.__actionRedo.setShortcuts(QtGui.QKeySequence.Redo)
                self.__actionRedo.setStatusTip("Refaire la dernière
opération")
                self.__actionCut = QtWidgets.QAction(QtGui.QIcon("edit-
cut.svg"), "Couper", self)
                self.__actionCut.setShortcuts(QtGui.QKeySequence.Cut)
                self.__actionCut.setStatusTip("Couper le texte vers le
presse-papier")
                self.__actionCopy = QtWidgets.QAction(QtGui.QIcon("edit-
copy.svg"), "Copier", self)
                self.__actionCopy.setShortcuts(QtGui.QKeySequence.Copy)
                self.__actionCopy.setStatusTip("Copier le texte vers le
presse-papier")
                self.__actionPaste = QtWidgets.QAction(QtGui.QIcon("edit-
paste.svg"), "Coller", self)
                self.__actionPaste.setShortcuts(QtGui.QKeySequence.Paste)
                self.__actionPaste.setStatusTip("Coller le texte depuis le
presse-papier")
                # ...
```

Nous allons associer les actions aux méthodes créées (non décrites ici) :

```
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                # ...
                self.__actionNew.triggered.connect(self.newDocument)
                self.__actionOpen.triggered.connect(self.openDocument)
                self.__actionSave.triggered.connect(self.saveDocument)
                self.__actionSaveAs.triggered.connect(self.saveAsDocument)
                self.__actionQuit.triggered.connect(self.quit)
                self.__actionUndo.triggered.connect(self.undo)
                self.__actionRedo.triggered.connect(self.redo)
                self.__actionCut.triggered.connect(self.cut)
                self.__actionCopy.triggered.connect(self.copy)
                self.__actionPaste.triggered.connect(self.paste)
                # ...
```

Les barres de menu `QMenu`

Nous allons à présent utiliser les actions précédemment créées pour les insérer dans des menus. Pour cela, nous allons créer deux menus : le menu *Fichier* et le menu *Édition*. Pour créer un menu, on appelle la méthode `addMenu(nom)` de la fenêtre `QMainWindow`. On passe en argument le nom du menu.

Ce nouveau menu accepte deux méthodes, `addAction(action)` qui ajoute une action au menu, et la méthode `addSeparator()` qui ajoute un séparateur.

Voici la création de nos deux menus :

```
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                # ...
                self.__menuFile = self.menuBar().addMenu("Fichier")
                self.__menuFile.addAction(self.__actionNew)
                self.__menuFile.addAction(self.__actionOpen)
                self.__menuFile.addAction(self.__actionSave)
                self.__menuFile.addAction(self.__actionSaveAs)
                self.__menuFile.addSeparator()
                self.__menuFile.addAction(self.__actionQuit)
                self.__menuEdit = self.menuBar().addMenu("Édition")
                self.__menuEdit.addAction(self.__actionUndo)
                self.__menuEdit.addAction(self.__actionRedo)
                self.__menuEdit.addSeparator()
                self.__menuEdit.addAction(self.__actionCut)
                self.__menuEdit.addAction(self.__actionCopy)
                self.__menuEdit.addAction(self.__actionPaste)
                # ...
```

Les barres d'outils

À l'instar des barres de menu, la classe `QMainWindow` possède une méthode `addToolBar(nom)` avec le nom passé en argument. Ces barres d'outils possèdent deux méthodes `addAction(action)` qui ajoute une action au menu et la méthode `addSeparator()` qui ajoute un séparateur.

Voici la création de la barre d'outils :

```
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                # ...
                self.__barreFile = self.addToolBar("Fichier")
                self.__barreFile.addAction(self.__actionNew)
                self.__barreFile.addAction(self.__actionOpen)
                self.__barreFile.addAction(self.__actionSave)
                self.__barreEdit = self.addToolBar("Édition")
                self.__barreEdit.addAction(self.__actionUndo)
                self.__barreEdit.addAction(self.__actionRedo)
                self.__barreEdit.addAction(self.__actionCut)
                self.__barreEdit.addAction(self.__actionCopy)
                self.__barreEdit.addAction(self.__actionPaste)
                # ...
```

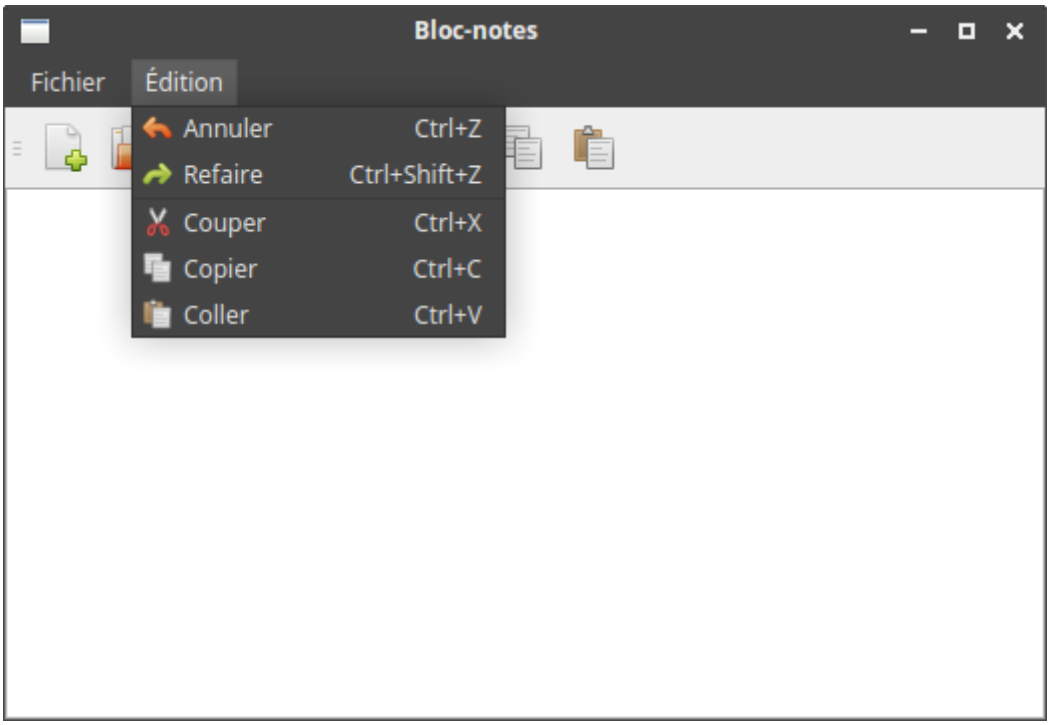
Voici le code source complet de notre application :

```
#!/usr/bin/env python3
import sys
from PySide2 import QtCore, QtGui, QtWidgets
class BlocNotes(QtWidgets.QMainWindow):
        def __init__(self, parent=None):
                QtWidgets.QMainWindow.__init__(self, parent)
                self.setWindowTitle("Bloc-notes")
                self.__zoneTexte = QtWidgets.QTextEdit()
                self.setCentralWidget(self.__zoneTexte)
                self.__actionNew = QtWidgets.QAction(QtGui.QIcon("document-
new.svg"), "Nouveau", self)
                self.__actionNew.setShortcuts(QtGui.QKeySequence.New)
                self.__actionNew.setStatusTip("Nouveau document")
```

```
                self.__actionOpen = QtWidgets.QAction(QtGui.QIcon("document-
open.svg"), "Ouvrir", self)
                self.__actionOpen.setShortcuts(QtGui.QKeySequence.Open)
                self.__actionOpen.setStatusTip("Ouvrir un document
existant")
                self.__actionSave = QtWidgets.QAction(QtGui.QIcon("document-
save.svg"), "Enregistrer", self)
                self.__actionSave.setShortcuts(QtGui.QKeySequence.Save)
                self.__actionSave.setStatusTip("Enregistrer le document")
                self.__actionSaveAs =
QtWidgets.QAction(QtGui.QIcon("document-save-as.svg"), "Enregistrer sous",
self)
                self.__actionSaveAs.setShortcuts(QtGui.QKeySequence.SaveAs)
                self.__actionSaveAs.setStatusTip("Enregistrer le document
sous")
                self.__actionQuit =
QtWidgets.QAction(QtGui.QIcon("exit.svg"), "Quitter", self)
                self.__actionQuit.setShortcuts(QtGui.QKeySequence.Quit)
                self.__actionQuit.setStatusTip("Quitter l'application")
                self.__actionUndo =
QtWidgets.QAction(QtGui.QIcon("undo.svg"), "Annuler", self)
                self.__actionUndo.setShortcuts(QtGui.QKeySequence.Undo)
                self.__actionUndo.setStatusTip("Annuler la dernière
opération")
                self.__actionRedo =
QtWidgets.QAction(QtGui.QIcon("redo.svg"), "Refaire", self)
                self.__actionRedo.setShortcuts(QtGui.QKeySequence.Redo)
                self.__actionRedo.setStatusTip("Refaire la dernière
opération")
                self.__actionCut = QtWidgets.QAction(QtGui.QIcon("edit-
cut.svg"), "Couper", self)
                self.__actionCut.setShortcuts(QtGui.QKeySequence.Cut)
                self.__actionCut.setStatusTip("Couper le texte vers le
presse-papier")
                self.__actionCopy = QtWidgets.QAction(QtGui.QIcon("edit-
copy.svg"), "Copier", self)
                self.__actionCopy.setShortcuts(QtGui.QKeySequence.Copy)
                self.__actionCopy.setStatusTip("Copier le texte vers le
presse-papier")
                self.__actionPaste = QtWidgets.QAction(QtGui.QIcon("edit-
paste.svg"), "Coller", self)
                self.__actionPaste.setShortcuts(QtGui.QKeySequence.Paste)
                self.__actionPaste.setStatusTip("Coller le texte depuis le
presse-papier")
                self.__actionNew.triggered.connect(self.newDocument)
                self.__actionOpen.triggered.connect(self.openDocument)
                self.__actionSave.triggered.connect(self.saveDocument)
                self.__actionSaveAs.triggered.connect(self.saveAsDocument)
                self.__actionQuit.triggered.connect(self.quit)
                self.__actionUndo.triggered.connect(self.undo)
                self.__actionRedo.triggered.connect(self.redo)
                self.__actionCut.triggered.connect(self.cut)
                self.__actionCopy.triggered.connect(self.copy)
                self.__actionPaste.triggered.connect(self.paste)
                self.__menuFile = self.menuBar().addMenu("Fichier")
                self.__menuFile.addAction(self.__actionNew)
                self.__menuFile.addAction(self.__actionOpen)
                self.__menuFile.addAction(self.__actionSave)
                self.__menuFile.addAction(self.__actionSaveAs)
                self.__menuFile.addSeparator()
                self.__menuFile.addAction(self.__actionQuit)
```

```
                self.__menuEdit = self.menuBar().addMenu("Édition")
                self.__menuEdit.addAction(self.__actionUndo)
                self.__menuEdit.addAction(self.__actionRedo)
                self.__menuEdit.addSeparator()
                self.__menuEdit.addAction(self.__actionCut)
                self.__menuEdit.addAction(self.__actionCopy)
                self.__menuEdit.addAction(self.__actionPaste)
                self.__barreFile = self.addToolBar("Fichier")
                self.__barreFile.addAction(self.__actionNew)
                self.__barreFile.addAction(self.__actionOpen)
                self.__barreFile.addAction(self.__actionSave)
                self.__barreEdit = self.addToolBar("Édition")
                self.__barreEdit.addAction(self.__actionUndo)
                self.__barreEdit.addAction(self.__actionRedo)
                self.__barreEdit.addAction(self.__actionCut)
                self.__barreEdit.addAction(self.__actionCopy)
                self.__barreEdit.addAction(self.__actionPaste)
                self.show()
        def fonct(self):
                pass
app = QtWidgets.QApplication(sys.argv)
fenetre = BlocNotes()
app.exec_()
```

Voici le rendu de notre fenêtre.



Exercices

Vous êtes nouvellement embauché dans une entreprise en bâtiment pour créer un programme permettant au secrétariat de saisir les estimations de travaux de l'entreprise. Votre application devra s'interfacer avec une base de données SQLite qui stockera le catalogue des prestations proposées par l'entreprise et leurs tarifs. Cette base de données contiendra également les estimations déjà réalisées.

Pour créer une estimation, il faut d'abord saisir les informations relatives au client (nom, prénom, adresse, code postal, ville, téléphone, courriel), le titre du chantier, puis choisir dans le catalogue les prestations à ajouter. Chaque prestation est dans une catégorie et comporte un texte la décrivant, un prix unitaire et une unité (définissant le prix unitaire). Voici un exemple :

Prestation	Prix unitaire (€/unité)	Unité
Cloison sur ossature métallique.	45	m²

Si la prestation n'existe pas, une boîte de dialogue permet d'en ajouter, de même pour les catégories. Lors de l'ajout d'une prestation à l'estimation, l'utilisateur doit choisir un taux de TVA (exprimé en %). Une fois la saisie de la prestation terminée, les totaux hors-taxes, de TVA et le total TTC (hors-taxes + TVA) sont automatiquement mis à jour.

Enfin, il sera possible d'exporter l'estimation au format texte suivant l'exemple ci-dessous :

```
Société Bati Plus
52 rue de Clairecombe
74930 Moulincourbe
                                                    Lionel Paulin
                                                    48 Ruelle de Locvaux
                                                    74019 Mivran
                                                    01 98 74 30 52

        lionel.paulin@exemple.com
Estimation numéro 524 réalisée le 10 avril 2017.
Chantier de plâtrerie
Prestation                      Prix unitaire   Quantité        Total HT
        TVA             Total TTC
Cloison sur ossature métallique         45 €/m²         17 m²           765€
                153€ (20%)      918€
Pose d'une porte                78 €/porte      1 porte         78€
        15,6€ (20%)     93,6€

Total HT : 843€
Total TVA : 168,6€
Total TTC : 1011,6€
```

Tous les totaux seront arrondis à deux décimales.