Programmation Python,

les fondamentaux

Sébastien Jeudy

Objectif et contenu

Ce cours généraliste vous initiera aux fondamentaux de la programmation en langage Python :

1) Techniques de base
2) Objets essentiels
3) Programmation Orientée Objet
4) Introduction aux interfaces graphiques Tkinter
5) Accès aux bases de données
6) Compléments

Public concerné

Développeurs souhaitant apprendre à programmer en langage Python (débutants en Python).

Prérequis

Connaissance d'un autre langage de programmation, si possible orienté objet.

Sommaire

1 TECHNIQUES DE BASE......11

1.1 Le langage Python......12
1.2 L'interpréteur de commandes Python......13
1.2.1 Accès......13
1.2.2 Saisie de nombres......14
1.2.3 Calculs et opérateurs......15
1.2.4 Commentaires......15
1.3 Un programme Python......16
1.3.1 Sur Windows......16
1.3.2 Sur Unix / Linux / macOS......18
1.3.3 Encodage des caractères......20
1.4 Variables......21
1.4.1 Création......21
1.4.2 Types de données......23
1.4.3 Chaînes de caractères......24
1.4.4 Opérations complémentaires......26
1.4.5 Fonctions utiles......29
1.5 Structures conditionnelles......32
1.5.1 Structures if, else, elif......32
1.5.2 Opérateurs de comparaison......35
1.5.3 Expressions booléennes......35
1.5.4 Mots-clés or, and, not......37
1.5.5 Structure match, case......38
1.6 Structures itératives......39

1.6.1 Boucle while......39
1.6.2 Boucle for......40
1.6.3 Mots-clés break et continue......42
1.7 Fonctions......43
1.7.1 Définition......43
1.7.2 Sans paramètres......44
1.7.3 Avec paramètres......45
1.7.4 Valeur par défaut des paramètres......47
1.7.5 Appels avec paramètres nommés......48
1.7.6 Signature......50
1.7.7 Renvoi de valeurs avec return......51
1.7.8 Portée des variables......52
1.7.9 Docstrings......53
1.7.10 Fonctions lambda......54
1.8 Modules......55
1.8.1 Définition......55
1.8.2 Importation classique......55
1.8.3 Espaces de noms......57
1.8.4 Importation avec espace de noms personnalisé (alias)......57
1.8.5 Importations spécifiques......58
1.8.6 Création......59
1.9 Packages......62
1.9.1 Définition......62
1.9.2 Création et importations......63
1.10 Exceptions......66
1.10.1 Définition......66
1.10.2 Interceptions de base......67
1.10.3 Mots-clés else et finally......69
1.10.4 Mot-clé pass......70
1.10.5 Assertions......71

1.10.6 Lever une exception........72

2 OBJETS ESSENTIELS........73

2.1 Notion d'objet........74
2.2 Chaînes de caractères........74
2.2.1 Classe str........74
2.2.2 Méthodes de mise en forme........76
2.2.3 Formatages........78
2.2.4 Concaténation classique........81
2.2.5 Parcours de chaînes........82
2.2.6 Découpage de chaînes........84
2.2.7 Compter, rechercher, remplacer........85
2.3 Listes et tuples........86
2.3.1 Classe list........86
2.3.2 Accès et modification des objets........87
2.3.3 Insertion d'objets........88
2.3.4 Concaténation de listes........90
2.3.5 Suppression d'objets........91
2.3.6 Parcours de listes........93
2.3.7 Tuples........96
2.3.8 Passage d'une chaîne à une liste et inversement........99
2.3.9 Fonction avec un nombre indéterminé de paramètres........100
2.3.10 Transformer une liste ou un tuple en paramètres de fonction........103
2.3.11 Compréhensions de liste........104
2.4 Dictionnaires et ensembles........106
2.4.1 Classe dict........106
2.4.2 Insertion, accès et modification des objets........107
2.4.3 Suppression d'objets........109
2.4.4 Application : dictionnaire de fonctions........110

2.4.5 Parcours de dictionnaires....112
2.4.6 Fonction avec un nombre indéterminé de paramètres nommés....116
2.4.7 Transformer un dictionnaire en paramètres nommés de fonction....118
2.4.8 Ensembles (sets)....119
2.5 Référence des objets....121
2.6 Fichiers....124
2.6.1 Module os....124
2.6.2 Répertoire courant....124
2.6.3 Ouverture d'un fichier....125
2.6.4 Fermeture d'un fichier....126
2.6.5 Lecture d'un fichier....127
2.6.6 Ecriture dans un fichier....129
2.6.7 Position dans le fichier....130
2.6.8 Ouverture avec with, as....132
2.6.9 Ecriture et lecture d'objets....133

3 PROGRAMMATION ORIENTÉE OBJET....135

3.1 Bref rappel....136
3.2 Définition d'une classe....137
3.2.1 Constructeur, attributs d'instances, instanciation....137
3.2.2 Attributs de classe....140
3.2.3 Méthodes d'instances (ou méthodes d'objets)....142
3.2.4 Méthodes de classe....144
3.2.5 Méthodes statiques....145
3.2.6 Introspection....146
3.2.7 Encapsulation et propriétés....147
3.3 Méthodes spéciales....152
3.3.1 Constructeur et destructeur....152
3.3.2 Représentation et affichage....154

3.3.3 Attributs....156
3.3.4 Conteneurs....161
3.3.5 Opérateurs....164
3.3.6 Sérialisation....168
3.4 Héritage....171
3.4.1 Définition....171
3.4.2 Héritage simple....172
3.4.3 Fonctions utiles....174
3.4.4 Héritage multiple....175
3.4.5 Exceptions personnalisées....176
3.5 Itérateurs....178
3.5.1 Utilisation....178
3.5.2 Création....180
3.6 Générateurs....182
3.6.1 Principe....182
3.6.2 Coroutines....186
3.7 Décorateurs....188
3.7.1 Définition....188
3.7.2 Modification de la fonction....190
3.7.3 Décorateur avec paramètres....191
3.7.4 Fonction avec paramètres....194
3.7.5 Appliquer plusieurs décorateurs....196
3.7.6 Appliquer un décorateur à une classe....197
3.8 Métaclasses....200
3.8.1 Processus d'instanciation....200
3.8.2 Créer une classe dynamiquement....202
3.8.3 Définition d'une métaclasse....206

4 INTRODUCTION AUX INTERFACES GRAPHIQUES TKINTER....209

4.1 Module tkinter......210
4.2 Exemple basique......210
4.3 Principaux widgets......212
4.3.1 Fenêtres et labels (complément)......212
4.3.2 Boutons......212
4.3.3 Zones de saisie......213
4.3.4 Cases à cocher......213
4.3.5 Boutons radio......214
4.3.6 Listes déroulantes......215
4.3.7 Combo boxes......216
4.3.8 Barres de défilement......217
4.3.9 Canevas......218
4.3.10 Boîtes de dialogue......219
4.3.11 Menus......220
4.4 Organisation des widgets......221
4.4.1 Redimensionnement......221
4.4.2 Positionnement......221
4.4.3 Regroupement......224
4.5 Accès aux fichiers......226
4.5.1 Ouvrir un fichier texte et l'afficher......226
4.5.2 Enregistrer un fichier texte......227
4.5.3 Ouvrir un fichier image et l'afficher......228
4.6 Evénements......229
4.6.1 Principe......229
4.6.2 Principaux événements du clavier......230
4.6.3 Principaux événements de la souris......231
4.6.4 Premier exemple......232
4.6.5 Deuxième exemple......233
4.6.6 Troisième exemple......234
4.6.7 Attributs de l'objet event......235

4.7 Classes d'interfaces......236

5 ACCÈS AUX BASES DE DONNÉES......241

5.1 API standard......242
5.2 PostgreSQL......243
5.2.1 Création de la base de données......243
5.2.2 Module psycopg2......244
5.2.3 Exécution de requêtes......245
5.3 MySQL......248
5.3.1 Création de la base de données......248
5.3.2 Module mysql.connector......249
5.3.3 Exécution de requêtes......250
5.4 SQLite......253
5.4.1 Spécificités......253
5.4.2 Création de la base de données......254
5.4.3 Module sqlite3......255
5.4.4 Exécution de requêtes......256

6 COMPLÉMENTS......259

6.1 Programmation système......260
6.1.1 Flux standards......260
6.1.2 Signaux......263
6.1.3 Arguments en ligne de commande......265
6.1.4 Vérifier le système d'exploitation......269
6.1.5 Commandes système......269
6.2 Expressions régulières......271
6.2.1 Définition......271
6.2.2 Regex courantes......271

6.2.3 Module re....272
6.2.4 Raw strings....273
6.2.5 Rechercher....274
6.2.6 Remplacer....276
6.2.7 Compiler....278
6.3 Temps, dates et heures....280
6.3.1 Module time....280
6.3.2 Epoch et timestamp....280
6.3.3 Informations sur une date et heure....282
6.3.4 Temporiser....283
6.3.5 Formater....284
6.3.6 Module datetime....285
6.4 Mathématiques....288
6.4.1 Module math....288
6.4.2 Module fractions....290
6.4.3 Module random....293
6.5 Mots de passe....295
6.5.1 Saisie....295
6.5.2 Hachage....296

1 Techniques de base

1.1 Le langage Python

Python est un **langage de programmation interprété**.

Date de première version : 20 février 1991
Auteur : Guido van Rossum (Néerlandais)
Développeurs actuels : Python Software Foundation / PSF (depuis 2001)

Paradigmes : impératif, structuré, fonctionnel et orienté objet
Typage : dynamique fort

Systèmes d'exploitation : multiplateformes
Licence : libre

Site Web : https://www.python.org ⇒ *téléchargements, documentations,...*

Versions majeures :

1991 : Python 0.9
1994 : Python 1.0
2000 : Python 2.0
2008 : Python 3.0 (attention : rupture de compatibilité avec Python 2) ⇒ ***ici est traité Python 3***

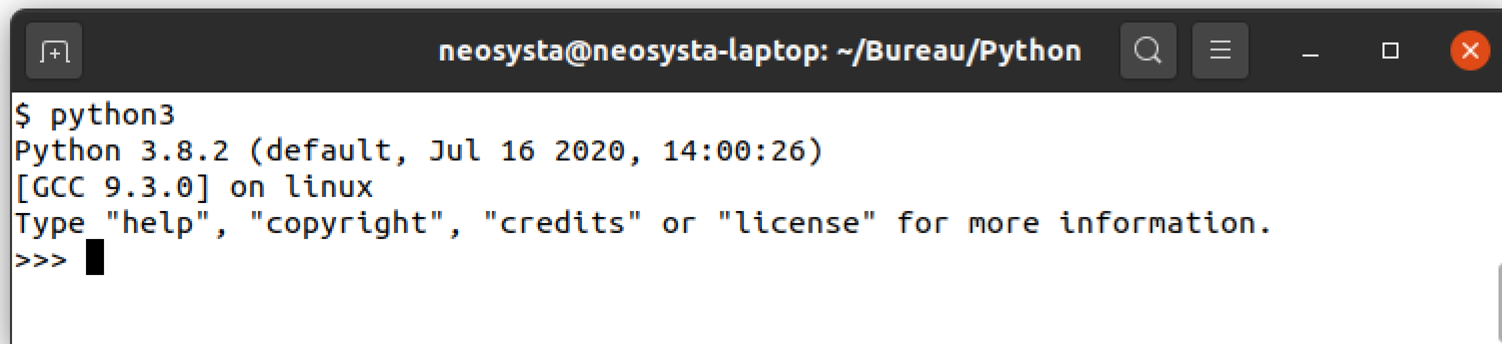
1.2 L'interpréteur de commandes Python

1.2.1 Accès

Les programmes en Python sont directement exécutés par l'**interpréteur Python** (abordé plus loin), mais ce dernier est également **utilisable en lignes de commandes :**

1) Ouvrir une **Console Windows** ou un **Terminal Unix / Linux / macOS**.

2) Saisir la **commande « python » sur Windows** ou **« python3 » sur Unix / Linux / macOS**.



⇒ *vérifier la version*

>>> correspond à l'invite de commande (prompt).

Dans l'interpréteur Python, **l'instruction « exit() » quitte l'interpréteur**. ⇒ *aussi les programmes*

Utiliser l'interpréteur Python en lignes de commandes est **pratique pour tester des instructions ou des séquences de codes** ("en live", sans sauvegarde).

1.2.2 Saisie de nombres

L'interpréteur Python en lignes de commandes **accepte la saisie directe de nombres :**

Interpréteur Python :

```
>>> 12
12
>>> -5.7
-5.7
```

⇒ *notation anglo-saxonne : le point correspond à la virgule*

1.2.3 Calculs et opérateurs

L'interpréteur Python en lignes de commandes **accepte aussi la saisie directe de calculs :**

Interpréteur Python :

```
>>> (5 + 4) / 3 + 2.3 * 10 - 6.5
19.5
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 7 % 3
1
```

⇒ *// correspond à la division entière et % au modulo (reste de la division entière)*

1.2.4 Commentaires

En Python, un commentaire **commence par un # (dièse)** et se termine par la fin de ligne.

1.3 Un programme Python

1.3.1 Sur Windows

1) **Accéder à un éditeur de texte brut** (Bloc-notes, Notepad++,…) **et écrire le programme :**

```
print("Hello, World!")
input()
```

⇒ *attention à la casse*

2) **Dans le dossier au choix, enregistrer le fichier sous le nom (par exemple) :** helloworld.py

⇒ *l'extension « .py » est obligatoire sur Windows pour que le fichier soit exécuté avec Python*

3) Avec l'Explorateur de fichiers, aller dans le dossier en question et **pour exécuter le programme double-cliquer sur l'icône du fichier** « helloworld.py ».

⇒ *le programme s'exécute dans une console et « input() » permet d'y rester jusqu'à [ENTRÉE]*

Autre solution d'exécution du programme :

1) **Ouvrir une Console Windows et à l'aide de la commande « cd » aller dans le dossier du fichier** « helloworld.py ».

2) **Pour exécuter le programme, saisir la commande :**

python helloworld.py

⇒ *en mode console, « input() » est inutile en fin de programme*

1.3.2 Sur Unix / Linux / macOS

1) **Ouvrir un Terminal Unix / Linux / macOS et à l'aide de la commande « cd » aller dans votre répertoire de travail**.

2) **Accéder à un éditeur de texte brut** (nano, vim, emacs,…) **pour éditer le fichier** « helloworld.py », par exemple :

```
nano helloworld.py
```

⇒ *l'extension « .py » n'est pas obligatoire sur Unix / Linux / macOS*

3) **Dans l'éditeur de texte, écrire le programme :**

```
#!/usr/bin/python3
print("Hello, World!")
```

⇒ *attention à la casse et à la version de Python (vérifier si nécessaire)*

4) **Enregistrer le fichier, retourner dans le Terminal et ajouter les permissions d'exécution au fichier** « helloworld.py » :

chmod ugo+x helloworld.py

5) **Pour exécuter le programme, saisir la commande :**

./helloworld.py

Autre solution d'exécution du programme (shebang et « chmod » inutiles mais hors normes) :

python3 helloworld.py

⇒ *attention à la version de Python (vérifier si nécessaire)*

1.3.3 Encodage des caractères

Par défaut, Python considère que ses fichiers sources sont encodés en UTF-8.

Idéalement, **pour éviter les problèmes d'encodage des caractères** des fichiers sources en Python, **ajouter la ligne suivante au début du code source :**

Par exemple sur Windows (à adapter si nécessaire) :

 # -*- coding: cp1252 -*-

⇒ *en 1ère ligne*

Normalement sur Unix / Linux / macOS :

 # -*- coding: utf-8 -*-

⇒ *sous le shebang, donc en 2ème ligne (mais logiquement inutile)*

1.4 Variables

1.4.1 Création

En Python, une variable existe à partir du moment où on lui affecte une valeur sous la forme :

nom_variable = valeur

C'est également au moment de cette affection qu'elle obtient son type (mais qui peut changer !).

Pour le nom des variables :

Lettres minuscules ou majuscules, chiffres et underscore « _ ».
Ne doit pas commencer par un chiffre.

⇒ *conseil de nommage : mon_age ou monAge*

Rappel : Python est sensible à la casse (différence entre minuscules et MAJUSCULES).

Python n'a pas besoin de point-virgule (;) pour terminer les instructions (en fin de ligne).

Les points-virgules sont uniquement utilisés pour séparer plusieurs instructions sur la même ligne.

Interpréteur Python :

```
>>> nombre1 = 100
>>> nombre1
100
```

Dans l'interpréteur de commandes, on peut afficher la valeur d'une variable en tapant seulement son nom (pas dans un programme Python !). **Autres opérations possibles (classiques) :**

Interpréteur Python :

```
>>> nombre1 = nombre1 + 50
>>> nombre1
150
>>> nombre2 = nombre1 / 3
>>> nombre2
50.0
```

<u>Attention</u> : ne pas utiliser les mots-clés de Python pour les noms créés (ils sont réservés).

1.4.2 Types de données

Python associe à chaque donnée un **type qui définit les opérations autorisées** sur celle-ci.

<u>Principaux types de données</u> : ⇒ *d'autres types existent*

int (entier), **float** (décimal), **str** (chaîne de caractères), **bool** (booléen, True ou False).

Une variable obtient un type à chaque affection, qui peut donc changer :

Interpréteur Python :

```
>>> variable = 10
>>> variable = "texte"
>>> variable
'texte'
```

1.4.3 Chaînes de caractères

Une chaîne de caractères **doit être entourée de délimiteurs :**

- guillemets : "exemple de chaîne de caractères"
- apostrophes : 'exemple de chaîne de caractères'
- triples guillemets : """exemple de chaîne de caractères"""
⇒ *il existe aussi les triples apostrophes (similaires aux triples guillemets)*

L'antislash **« \ » sert à échapper** notamment les apostrophes et les guillemets :

Interpréteur Python :

```
>>> texte = 'J'ai un problème'
  File "<stdin>", line 1
    texte = 'J'ai un problème'
               ^
SyntaxError: invalid syntax
>>> texte = 'Je n\'ai plus de problème'
>>> texte
"Je n'ai plus de problème"
```

<u>Précision</u> : pour écrire un antislash dans une chaîne, il faut l'échapper lui-même (« \\ »).

Les triples guillemets (ou les triples apostrophes) dispensent d'échapper les guillemets et apostrophes, et permettent d'écrire un texte sur plusieurs lignes :

Interpréteur Python :

```
>>> texte = """J'arrive
... à écrire
... des lignes"""
>>> texte
"J'arrive\nà écrire\ndes lignes"
```

⇒ *les retours à la ligne sont automatiquement remplacés par des « \n »*

1.4.4 Opérations complémentaires

Incréments :

Interpréteur Python :

```
>>> nombre = 1
>>> nombre
1
>>> nombre = nombre + 1
>>> nombre
2
>>> nombre += 1
>>> nombre
3
>>> nombre += 5
>>> nombre
8
```

⇒ *les opérateurs -=, *= et /= existent également (mais pas ++ et -- !)*

Python permet facilement d'**échanger les valeurs de deux variables :**

Interpréteur Python :

```
>>> i = 10
>>> j = 20
>>> i,j = j,i
>>> i
20
>>> j
10
```

Ou encore d'**affecter la même valeur à plusieurs variables :**

Interpréteur Python :

```
>>> i = j = k = 10
>>> i
10
>>> j
10
>>> k
10
```

On peut également **découper les longues instructions sur plusieurs lignes à l'aide de « \ » :**

Interpréteur Python :

```
>>> (5 + 30 - 6) \
... * (2 + 25 - 10) \
... / (13 - 15)
-246.5
```

⇒ *utile surtout dans un code source*

Enfin, **pour une amélioration de la rigueur de programmation :**

Python 3.6 a introduit l'annotation « nom: str = "DUPONT" ».

Mais ensuite tout de même possible « nom = 50 » !

⇒ *uniquement pour la lisibilité des programmes (aucuns contrôles)*

1.4.5 Fonctions utiles

Comme tous les langages, **Python dispose de fonctions intégrées et permet d'en définir** (abordé plus loin). Appel d'une fonction :

nom_fonction(parametre1, parametre2,…) ⇒ *sans paramètres : nom_fonction()*

La fonction « type » renvoie le type de ce qu'on lui passe en paramètre :

Interpréteur Python :

```
>>> nombre = 10
>>> type(nombre)
<class 'int'>
>>> type(1.23)
<class 'float'>
>>> type("texte")
<class 'str'>
>>> type(True)
<class 'bool'>
```

⇒ *en Python, les types de données sont définis sous forme de classes d'objets*

La fonction « print » affiche dans la console ce qu'on lui passe en paramètres (indispensable pour les affichages dans un programme) :

Interpréteur Python :

```
>>> nombre1 = 10
>>> nombre2 = 20
>>> print(nombre1)
10
>>> print("1er nombre =", nombre1, "et 2ème nombre =", nombre2)
1er nombre = 10 et 2ème nombre = 20
```

⇒ *Python affiche tous les paramètres transmis séparés par un espace*

Utilisation directe des **« \n » (retour à la ligne) et « \t » (tabulation) :**

Interpréteur Python :

```
>>> print("Bonjour\n\n\n\t\t\tà tous")
Bonjour



			à tous
```

La fonction « input » récupère une saisie au clavier :

input() ⇒ *attend la saisie de la touche [ENTRÉE]*

variable = input("Saisissez quelque chose : ") ⇒ *affiche le texte et la variable récupère la saisie*

Attention : **la saisie est récupérée sous forme de chaîne de caractères** (type « str »).

Si nécessaire, utiliser les **fonctions de conversion de type :** ⇒ *mêmes noms que les types*

Convertir en entier : **int(...)**
Convertir en décimal : **float(...)**
Convertir en booléen : **bool(...)**
Et convertir en texte : **str(...)**

Interpréteur Python :

```
>>> age = input("Saisissez votre âge : ")
Saisissez votre âge : 50
>>> type(age)
<class 'str'>
>>> age = int(age)
>>> type(age)
<class 'int'>
```

1.5 Structures conditionnelles

1.5.1 Structures if, else, elif

Code Python :

```
if age >= 18:
    print("Vous êtes majeur.")
```

Attention :

Les structures complexes en Python n'ont pas de début et de fin explicites, ni d'accolades qui pourraient marquer là où commence et se termine le bloc d'instructions.

Le seul délimiteur sont les deux points (:) et l'indentation du code lui-même (obligatoires).

La recommandation pour chaque niveau d'indentation est de 4 espaces.

Syntaxe également possible : if int(nombre) / 2 >= 50:

Code Python :

```
if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

Code Python :

```
if note >= 16 and note <= 20:
    print("Très Bien")
else:
    if note >= 14 and note < 16:
        print("Bien")
    else:
        if note >= 12 and note < 14:
            print("Assez Bien")
        else:
            if note >= 10 and note < 12:
                print("Passable")
            else:
                if note >= 0 and note < 10:
                    print("Ajourné")
                else:
                    print("La note doit être comprise entre 0 et 20 !")
```

Tous les « else: if » peuvent être remplacés par des « elif », simplifie la structure et les indentations :

Code Python :

```
if note >= 16 and note <= 20:
    print("Très Bien")
elif note >= 14 and note < 16:
    print("Bien")
elif note >= 12 and note < 14:
    print("Assez Bien")
elif note >= 10 and note < 12:
    print("Passable")
elif note >= 0 and note < 10:
    print("Ajourné")
else:
    print("La note doit être comprise entre 0 et 20 !")
```

⇒ *un seul « else: » à la fin (optionnel)*

Dans tous les cas, attention aux « : » et à l'indentation.

1.5.2 Opérateurs de comparaison

Pour les conditions de test :

<	**Strictement inférieur**
>	**Strictement supérieur**
<=	**Inférieur ou égal**
>=	**Supérieur ou égal**
==	**Égal**
!=	**Différent**

⇒ *« == » et non « = » (affectation)*

1.5.3 Expressions booléennes

Les conditions de test sont des expressions booléennes également appelées prédicats.

Leur résultat est un **booléen (type « bool ») : True** (vrai) **ou False** (faux).

Interpréteur Python :

```
>>> age = 15
>>> age == 18
False
>>> age != 18
True
>>> age < 18
True
```

Les valeurs booléennes **« True » et « False » peuvent être affectées à des variables :**

majeur = False

Une condition de test sur une variable non définie génère une erreur.

1.5.4 Mots-clés or, and, not

Comme dans tous les langages, **entre deux conditions de test :**

« or » est utilisé **quand l'une OU l'autre condition** doit être vraie (True).
« and » est utilisé **quand l'une ET l'autre condition** doivent être vraies (True).

⇒ *le résultat de ces opérateurs logiques est également un booléen*

Exemple :

```
if note >= 16 and note <= 20:
```

« not » inverse un prédicat : « not age == 18 » équivaut donc à « age != 18 ».

Pour tester un booléen, préférer « if majeur: » / « if not majeur: ».

(plutôt que « majeur == True » / « majeur != True » ou « majeur is True » / « majeur is not True »)

Les parenthèses ( ) permettent également de **forcer les priorités** (rappel : ET prioritaire sur OU).

1.5.5 Structure match, case

Python 3.10 a introduit la structure « match, case » (différents types d'évaluations existent) :

Code Python :

```
match jour:     # variable testée (ici un numéro de jour), suivie de chaque cas à évaluer
    case 1:
        print("Lundi")
    case 2:
        print("Mardi")
    case 3:
        print("Mercredi")
    case 4:
        print("Jeudi")
    case 5:
        print("Vendredi")
    case 6:
        print("Samedi")
    case 7:
        print("Dimanche")
    case _:     # autres cas
        print("Jour inconnu")
```

1.6 Structures itératives

1.6.1 Boucle while

Similaire aux autres langages, **répète un bloc d'instructions tant qu'une condition est vraie :**

Code Python :

```
mot = "pasfin"
while mot != "fin":
    mot = input("Tapez 'fin' pour arrêter : ")
```

Code Python :

```
nombre = 1

while nombre <= 100:
    print(nombre)
    nombre += 1
```

1.6.2 Boucle for

Différente des autres langages, **sert à parcourir une séquence de données :** ⇒ *très utile !*

for element in sequence:

La variable « element » prend successivement chacune des valeurs de la séquence parcourue.

⇒ *« foreach » dans d'autres langages*

Une chaîne de caractères est une séquence de caractères :

Code Python :

```
texte = "Chaîne de caractères"
for lettre in texte:
    print(lettre)
```

⇒ *affiche chaque lettre ligne par ligne (chaque « print » faisant un retour à la ligne)*

Fonctionne également :

```
for lettre in "Chaîne de caractères":
    print(lettre)
```

Le mot-clé « in » peut s'utiliser dans d'autres conditions :

Code Python :

```
texte = "exemple"
phrase = "Ceci est un exemple de phrase."
if texte in phrase:
    print(texte, "est présent dans :", phrase)
```

Pour boucler sur des nombres entiers : ⇒ *range(début, fin+1)*

Code Python :

```
for nombre in range(1, 101):
    print(nombre)
```

Et aussi : for nombre in range(100, 48, -2): ⇒ *nombres de 100 à 50 de -2 en -2*

1.6.3 Mots-clés break et continue

Pour les boucles while / for :

« break » arrête la boucle.

« continue » continue la boucle à l'itération suivante (sans exécuter la suite du bloc).

Code Python :

```
n = 0
while True:     # par défaut : boucle infinie
    n += 1
    if n <= 3:
        continue
    print(n)
    if n == 6:
        break
```

⇒ *affiche uniquement 4, 5, 6*

Eviter d'utiliser « break » et « continue » (comme ici, en général on peut s'en passer).

1.7 Fonctions

1.7.1 Définition

Comme tous les langages, Python permet de définir des **fonctions réalisant un traitement et appelées au besoin** (éventuellement avec paramètre(s)).

Si nécessaire, celles-ci **peuvent également renvoyer / retourner un résultat** à l'appelant.

⇒ *modularité : boîtes noires encapsulant les traitements récurrents*

(comme les fonctions intégrées type, print, input,...)

Syntaxe générale de définition (création) d'une fonction :

```
def nom_fonction(parametre1, parametre2,…):
    # bloc d'instructions (avec indentations)
```

⇒ *sans paramètres : def nom_fonction():*

⇒ *mêmes règles de nommage que les variables*
⇒ *ne pas reprendre le nom d'une variable et vice-versa, l'une redéfinirait (écraserait) l'autre !*

1.7.2 Sans paramètres

Code Python :

```
# Définition :

def boucler():
    nombre = 1
    while nombre <= 100:
        print(nombre)
        nombre += 1

# Appel :

boucler()
```

Précision :

Les variables définies dans la fonction sont locales à cette fonction, elles sont inconnues du programme appelant.

1.7.3 Avec paramètres

Code Python :

```
# Définition :

def boucler(debut, fin, increment):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Appel :

boucler(10, 20, 2)
```

⇒ *affiche les nombres de 10 à 20 de 2 en 2*

Lors de l'appel, les valeurs des paramètres peuvent être transmises "en dur" ou via des variables.

Sans nommer les paramètres (abordé plus loin) : **attention à l'ordre, au type et au nombre des paramètres transmis** par rapport à leur définition dans la fonction.

En Python, **la définition des paramètres dans « def » doit uniquement préciser leur nom sans type** (mais attention aux types lors de l'appel).

Précision :

Python 3.5 a introduit le « typing » (annotation des types dans « def »), par exemple :

```
def nom_fonction(parametre1: str, parametre2: float) -> str:
```

⇒ *uniquement pour la lisibilité des programmes (aucuns contrôles)*

Dans certains langages, il y a également la notion de passage de paramètres par valeur ou par référence (adresse) :

Les paramètres sont toujours passés par référence en Python.

Par contre, il y a des paramètres « muables / mutables » (dont on peut modifier le contenu ou la valeur ; les listes et les dictionnaires par exemple) **et des paramètres « immuables / immutables »** (dont le contenu ou la valeur est fixe / statique ; les nombres, les booléens et les chaînes de caractères par exemple ⇒ *équivalent à un passage par valeur*).

1.7.4 Valeur par défaut des paramètres

Dans « def », on peut également définir une valeur par défaut pour les paramètres.
⇒ *idéalement : définir une valeur par défaut pour tous les paramètres (voir paramètres nommés)*

Si un paramètre n'est pas précisé lors de l'appel, il prend sa valeur par défaut :

Code Python :

```
# Définition :

def boucler(fin, debut=1, increment=1):     # paramètres par défaut en dernier !
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Appel normal :

boucler(20, 10, 2)     # de 10 à 20 de 2 en 2

# Appel avec paramètres par défaut (pour debut et increment) :

boucler(10)     # de 1 à 10 de 1 en 1
```

1.7.5 Appels avec paramètres nommés

Lors de l'appel d'une fonction, on peut nommer les paramètres (pratique quand il y a des paramètres avec valeur par défaut) :

Interpréteur Python :

```
>>> def test(p1=1, p2=2, p3=3):
...         print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test()
p1 = 1 p2 = 2 p3 = 3
>>> test(10)
p1 = 10 p2 = 2 p3 = 3
>>> test(p2=20)
p1 = 1 p2 = 20 p3 = 3
>>> test(p2=20, p3=30)
p1 = 1 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, p1=10)
p1 = 10 p2 = 20 p3 = 30
```

<u>Mais aussi (avec paramètres sans valeur par défaut)</u> :

Interpréteur Python :

```
>>> def test(p1, p2, p3):
...         print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test(10, p3=30, p2=20)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, p1=10)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, 10)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Interpréteur Python :

```
>>> def test(p1, p2, p3=3):
...         print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test(10, p3=30, p2=20)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p1=10)
p1 = 10 p2 = 20 p3 = 3
```

Rappel : les valeurs transmises en paramètres peuvent être des variables.

1.7.6 Signature

La signature d'une fonction **c'est ce qui permet de l'identifier**.

Dans certains langages, c'est son nom et le type de ses paramètres.

En Python, les paramètres d'une fonction n'ayant pas de définition de type, **sa signature n'est que son nom**.

Dans le même programme, on peut redéfinir une fonction déjà définie auparavant, mais dans ce cas **c'est la dernière définition qui est prise en compte (la suivante écrase la précédente)**.

En Python, il n'y a donc pas de surcharges de fonctions (plusieurs fonctions du même nom avec paramètres différents).

⇒ *idéalement : définir les fonctions en début de programme (en évitant de les redéfinir ensuite)*

1.7.7 Renvoi de valeurs avec return

Une fonction peut également **renvoyer / retourner des valeurs à l'appelant grâce à l'instruction « return »** ; soit à une variable (stockage), soit à une autre fonction (utilisation).

En fonction des cas de traitement, **il peut y avoir plusieurs renvois / returns dans la même fonction**. Mais un renvoi / return provoque toujours la sortie (fin) de la fonction.

Interpréteur Python :

```
>>> def montant_ttc(montant_ht, taux_tva):
...         return montant_ht * (1 + taux_tva / 100)
...
>>> montant_facture = montant_ttc(10, 20)
>>> print(montant_facture)
12.0
>>> print(montant_ttc(10, 20))
12.0
```

On peut même retourner plusieurs valeurs en les séparant de virgules, qu'on peut récupérer dans des variables également séparées de virgules (abordé plus loin).

1.7.8 Portée des variables

Les variables extérieures qui ne sont que référencées (lues) à l'intérieur d'une fonction sont implicitement globales. Une variable qui se voit attribuer une valeur n'importe où dans le corps d'une fonction est par défaut locale, sauf si elle est déclarée auparavant avec « global » :

```
>>> variable = 10
>>> def test():
...     print(variable)     # uniquement lue : possède la valeur extérieure
>>> test()
10
>>> def test():
...     variable = 20     # définie en local : différente de la variable extérieure
>>> test()
>>> variable
10
>>> def test():
...     global variable     # définie en global : correspond à la variable extérieure
...     variable = 20
>>> test()
>>> variable
20
```

1.7.9 Docstrings

Une docstring est l'**explication (document d'aide) qu'on peut donner à une fonction** (ou un module, une classe, une méthode,…).

Elle **s'écrit entre triples guillemets** (ou triples apostrophes) **juste en-dessous de la ligne de définition**, tout en respectant l'indentation.

⇒ *avant la première ligne de code, après c'est un simple commentaire*

Elle **peut s'écrire sur une ou plusieurs lignes** (accepte les retours à la ligne).

On peut **ensuite l'afficher à l'aide de la fonction « help » :**

Interpréteur Python :

```
>>> def test():
...         """Fonction test"""
...         print("test")
...
>>> help(test)
```

La fonction « help » est **aussi utilisable pour tous les autres objets Python** (pensez-y).

1.7.10 Fonctions lambda

Une fonction lambda est une **fonction limitée à une seule instruction** (utile pour ce genre de cas simple).

Syntaxe générale de définition (création) d'une fonction lambda :

nom_fonction = lambda parametre1, parametre2,...: instruction de retour

Interpréteur Python :

```
>>> montant_ttc = lambda montant_ht, taux_tva: montant_ht * (1 + taux_tva / 100)
>>> montant_facture = montant_ttc(10, 20)
>>> print(montant_facture)
12.0
>>> print(montant_ttc(10, 20))
12.0
```

Attention aux « : ».

Les paramètres avec valeur par défaut et les paramètres nommés sont également possibles.

1.8 Modules

1.8.1 Définition

L'interpréteur Python intègre déjà un certain nombre de fonctions (et classes).

Mais **de nombreuses autres existent disponibles à travers des modules**, déjà fournis avec l'installation de Python ou téléchargeables en complément.

Un module est un **fichier contenant du code** (potentiellement n'importe quel code Python), généralement des **variables, fonctions ou classes liées par un sujet commun**.

Pour pouvoir utiliser les fonctionnalités d'un module, c'est-à-dire ses variables, fonctions, classes, **il suffit de l'importer auparavant** (dans l'interpréteur ou le programme concerné).

1.8.2 Importation classique

Pour importer un module (par exemple « math ») : **import math** ⇒ *aussi : import sys, os*

Puis, **pour utiliser une variable ou appeler une fonction du module (« math ») :**

Interpréteur Python :

```
>>> math.pi
3.141592653589793
>>> print(math.pi)
3.141592653589793
>>> math.sqrt(25)
5.0
>>> print(math.sqrt(25))
5.0
```

⇒ *on préfixe par le nom du module*

Pour connaître la liste des fonctionnalités du module (« math ») : **help(math)**

Pour lister uniquement le nom des variables et des fonctions du module (« math ») : **dir(math)**

Pour afficher l'aide d'une fonction du module (« math.sqrt ») : **help(math.sqrt)**

Attention : « math.sqrt » renvoie la référence de la fonction et « math.sqrt() » appelle et renvoie le résultat de la fonction.

1.8.3 Espaces de noms

Les variables et les fonctions (éventuellement les classes) **préfixées par le nom du module** (« math. ») **sont dans l'espace de noms du module** (« math »).

Ce qui évite d'avoir des conflits de noms entre ceux du module, ceux des autres modules et ceux du programme principal.

⇒ *chacun peut avoir une variable « pi » et une fonction « sqrt » sans conflits de noms*

1.8.4 Importation avec espace de noms personnalisé (alias)

On peut **personnaliser l'espace de noms d'un module (« math » en « mathematiques ») :**

Interpréteur Python :

```
>>> import math as mathematiques
>>> mathematiques.sqrt(25)
5.0
```

1.8.5 Importations spécifiques

On peut **importer une seule variable ou une seule fonction d'un module** (toujours « math ») :

Interpréteur Python :

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import sqrt
>>> sqrt(25)
5.0
```

⇒ *on ne préfixe pas par le nom du module (donc dans l'espace de noms du programme principal)*
⇒ *mais risques de conflits de noms (le dernier défini écrase le précédent) !*

De cette façon, on peut aussi **importer toutes les fonctionnalités du module (avec « * ») :**

Interpréteur Python :

```
>>> from math import *
>>> sqrt(25)
5.0
```

1.8.6 Création

Rappel :

Un module est un **fichier contenant du code** (potentiellement n'importe quel code Python), généralement des **variables, fonctions ou classes liées par un sujet commun**.

Exemple de création et d'utilisation d'un nouveau module :

1) **Créer le fichier du module « boucler.py » (contenant la fonction « fonc_boucler ») :**

Code Python :

```
"""Module boucler"""

def fonc_boucler(fin, debut=1, increment=1):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment
```

⇒ *module incluant une docstring (en 1ère ligne et entre triples guillemets) pour « help(boucler) »*

2) **Créer le fichier du programme « testboucler.py » (importo le module « boucler ») :**

⇒ *dans le même répertoire que le fichier du module « boucler.py »*

Code Python :

```
from boucler import *

# Appel de la fonction fonc_boucler (du module boucler)
fonc_boucler(20, 10, 2)
```

⇒ *import du module « boucler » : nom de son fichier « boucler.py » sans l'extension « .py »*

Précision :

Au moment de l'import du module, Python lit (ou crée si inexistant / modifié) son **fichier "précompilé" avec extension « .pyc » dans le dossier « __pycache__ »** (générés automatiquement).

⇒ *pour des gains de performance à l'exécution*

Un module étant un fichier de code Python avec variables / fonctions / classes (importées depuis d'autres programmes), **on peut également prévoir leurs tests dans le module lui-même et exécuter ce dernier comme tout programme Python**.

Mais pour éviter l'exécution de ces tests lors de l'import du module, **il faut les conditionner dans le fichier du module par la ligne « if __name__ == "__main__": »**.

⇒ « __name__ » est une variable de l'interpréteur qui vaut « __main__ » quand le fichier du module est directement exécuté (i.e. hors import)

Code Python :

```
"""Module boucler"""

def fonc_boucler(fin, debut=1, increment=1):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Test de la fonction fonc_boucler
if __name__ == "__main__":
    fonc_boucler(10)
```

1.9 Packages

1.9.1 Définition

Un module est un fichier de code Python avec variables / fonctions / classes (importées depuis d'autres programmes).

Un package regroupe des modules ou même d'autres packages.

Concrètement, c'est un **dossier contenant des fichiers de modules éventuellement organisés en sous-dossiers** correspondant à des "sous-packages".

Au final, **un package est une bibliothèque de fonctionnalités distribuables**. ⇒ *modularité*

On peut ensuite **importer un package ou l'un de ses "sous-packages", ou encore seulement l'un de ses modules / variables / fonctions / classes**.

On accède aux packages / "sous-packages" / modules en précisant leur chemin avec séparateurs « . » dans la hiérarchie des dossiers / sous-dossiers / fichiers de modules.

⇒ *hiérarchie d'espaces de noms évitant les conflits de noms*

1.9.2 Création et importations

Exemple d'une hiérarchie de packages (issu de https://docs.python.org) :

```
sound/                  Top-level package
    __init__.py         Initialize the sound package
    effects/            Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/            Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Chaque dossier de package doit contenir un fichier « __init__.py » pouvant être vide ou contenir un code exécuté au moment de l'import du package (initialisations de variables, importations d'autres packages,…).

Dans un programme, les imports se font par défaut depuis le répertoire courant du programme (la liste « sys.path » du module « sys » permet de définir le "path" des chemins parcourus).

Différentes syntaxes d'imports existent :

Créer cet échantillon de packages (fichiers « __init__.py » vides) :

```
sound/
    __init__.py
    effects/
        __init__.py
        echo.py
```

Avec le module « echo.py » :

Code Python :

```python
def echofilter():
    print("Fonction echofilter")
```

Puis tester ces **exemples d'imports et d'utilisations de fonctions :**

import sound.effects.echo ⇒ *import d'un module (« echo »)*
sound.effects.echo.echofilter()

from sound.effects import echo ⇒ *import d'un module (« echo »)*
echo.echofilter()

from sound.effects.echo import echofilter ⇒ *import d'une fonction (« echofilter »)*
echofilter()

Eviter d'utiliser « from package import * » : importe les noms de modules définis dans la liste « __all__ » initialisable dans « __init__.py » du package (mais lourd et ambigu).

from sound.effects import *
echo.echofilter()

⇒ *fonctionne uniquement avec « __all__ = ["echo"] » dans « __init__.py » du package « effects »*

1.10 Exceptions

1.10.1 Définition

Une exception est une **interruption du programme provoquée par une erreur** (mais pas que).

Quand une erreur se produit (syntaxe, opération,...), Python lève une exception :

Interpréteur Python :

```
>>> int("texte")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'texte'
```

En précisant :

- Les fichier / ligne / module concernés
- Le type d'exception (différents types existent) : « ValueError »
- Le message de l'erreur en question : « invalid literal for int() with base 10: 'texte' »

En Python, dès qu'une erreur / exception se produit le traitement s'arrête.

Mais on peut intercepter une exception : essayer un bloc de code et continuer s'il n'y a pas d'erreur ou exécuter un bloc particulier en cas d'erreur.

1.10.2 Interceptions de base

Syntaxe minimale de la structure « try, except » :

try:
 # bloc à essayer
except:
 # bloc exécuté en cas d'erreur

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except:
    print("Erreur de calcul.")
```

L'inconvénient de cette syntaxe minimale est que le traitement exécuté en cas d'erreur est le même quel que soit le type d'exception.

Mais on peut isoler et différencier les types d'exceptions possibles, avec la syntaxe :

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError:
    print("Montant HT ou taux de TVA non définis.")
except TypeError:
    print("Montant HT ou taux de TVA non numériques.")
```

Et aussi récupérer le message d'une exception dans une variable grâce au mot-clé « as » :

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError as message_exception:
    print(message_exception)
except TypeError as message_exception:
    print(message_exception)
```

Des exceptions ne sont pas des erreurs : CTRL+C (arrêt de programme) est une exception.

⇒ *donc éviter d'utiliser la syntaxe minimale (i.e. « except » sans préciser de types d'exceptions)*
⇒ *« except » sans type d'exception est surtout utile pour les autres cas (« except » en dernier)*
⇒ *l'interpréteur de commandes Python est aussi pratique pour tester les types d'exceptions*

1.10.3 Mots-clés else et finally

On peut également ajouter à la structure « try, except » les mots-clés « else » et « finally » :

« else » permet d'exécuter un traitement si aucune erreur ne survient dans le bloc « try ».

⇒ *par exemple pour mettre en forme le résultat*

« finally » permet d'exécuter un traitement après le bloc « try » quel que soit son résultat (avec ou sans erreur).

⇒ *« finally » est exécuté dans tous les cas, même avec un « return » dans un « except » !*

(ces deux mots-clés peuvent s'utiliser indépendamment)

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError:
    print("Montant HT ou taux de TVA non définis.")
except TypeError:
    print("Montant HT ou taux de TVA non numériques.")
else:
    print("Le montant TTC est :", mt_ttc)
finally:
    print("Fin de traitement du montant TTC.")
```

1.10.4 Mot-clé pass

On utilise « pass » **dans les blocs d'instructions sans code** (quels qu'ils soient) :

```
try:
    # bloc à tester
except type_exception:
    pass     # pas de traitement en cas d'erreur (rien ne se passe)
```

1.10.5 Assertions

Une assertion permet de **tester une condition avec le mot-clé « assert »**.

Si la condition est fausse (False), une exception de type « AssertionError » est levée :

Interpréteur Python :

```
>>> nombre = 10
>>> assert nombre == 10
>>> assert nombre == 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Utile dans les structures « try, except » :

Code Python :

```
try:
    assert mt_ht >= 0     # exception AssertionError si condition fausse
except AssertionError:
    print("Le montant HT est négatif !")
```

1.10.6 Lever une exception

On peut soi-même lever une exception **grâce au mot-clé « raise »**.

Syntaxe générale d'une levée d'exception :

raise type_exception("message affiché")

Interpréteur Python :

```
>>> raise ValueError("erreur de valeur")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: erreur de valeur
>>>
>>> raise TypeError("erreur de type")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: erreur de type
```

⇒ *peut être utile dans certains traitements*

2 Objets essentiels

2.1 Notion d'objet

Python est un **langage de programmation orienté objet**.

Un objet est avant tout une structure de données (en mémoire) :

La principale différence vient du fait que **l'objet regroupe les données** (variables sous forme d'attributs) **et les moyens de traiter ces données** (fonctions sous forme de méthodes).

Variables, fonctions,… **tout est objet en Python !**

Enfin, **un objet est créé (instancié) à partir d'une classe** qui est un "moule à objets" (modèle).

2.2 Chaînes de caractères

2.2.1 Classe str

Les chaînes de caractères sont des objets créés (instanciés) à partir de la classe « str ».

Quand on crée une variable contenant du texte c'est un objet de la classe « str » :

Interpréteur Python :

```
>>> texte = "Chaîne de caractères"
>>> type(texte)
<class 'str'>
```

Comme toute classe, on peut aussi instancier un objet "chaîne de caractères" avec la syntaxe :

Interpréteur Python :

```
>>> texte1 = str()     # créé vide, équivalent à : texte1 = ""
>>> type(texte1)
<class 'str'>
>>> print(texte1)

>>> texte2 = str("Chaîne de caractères")     # équivalent à : texte2 = "Chaîne de caractères"
>>> type(texte2)
<class 'str'>
>>> print(texte2)
Chaîne de caractères
```

La classe « str » dispose de nombreuses méthodes ("fonctions") accessibles par la syntaxe :

nom_objet.nom_methode(arguments) ⇒ *sans arguments : nom_objet.nom_methode()*

⇒ *fonctionnalités manipulant les chaînes de caractères*

2.2.2 Méthodes de mise en forme

Principales méthodes de mise en forme des chaînes de caractères :

lower : en minuscules
upper : en majuscules
capitalize : première lettre en majuscule, les autres en minuscules
strip : sans espaces ni « \n » au début et à la fin
center : centrée par rapport à une taille de caractères (passée entre parenthèses)
…

Liste complète des méthodes de la classe « str » : **help(str)** ⇒ *aussi : dir(str)*

Aide d'une méthode (exemple) : **help(str.lower)**

Interpréteur Python :

```
>>> texte = "EXEMPLE DE TEXTE"
>>> texte.lower()
'exemple de texte'
>>> texte = "exemple de texte"
>>> texte.upper()
'EXEMPLE DE TEXTE'
>>> texte.capitalize()
'Exemple de texte'
>>> texte = "    exemple de texte  "
>>> texte.strip()
'exemple de texte'
>>> texte.strip().center(30)
'       exemple de texte       '
```

Pour la dernière :

« texte.strip() » renvoie une chaîne de caractères, donc elle-même un objet de la classe « str » pour lequel on appelle à son tour la méthode « center(30) ».

Précision :

Ces méthodes renvoient une chaîne résultat mais **ne modifient pas la chaîne d'origine**.

2.2.3 Formatages

La méthode « format », également de la classe « str », **permet de formater une chaîne de caractères** à l'aide de plusieurs syntaxes.

Avec indices des paramètres à insérer : ⇒ *ordre indifférent*

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> print("{1} a {0} ans.".format(age, nom))     # directement affiché
Tom a 30 ans.
>>> chaine = "{1} a {0} ans.".format(age, nom)     # stocké dans une variable
>>> chaine
'Tom a 30 ans.'
```

⇒ *les indices commencent à 0 (zéro) : {0} correspond à « age » et {1} correspond à « nom »*

Sans indices des paramètres à insérer : ⇒ *ordre à respecter*

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> print("{} a {} ans.".format(nom.upper(), age))
TOM a 30 ans.
```

Avec noms et valeurs des paramètres à insérer : ⇒ *ordre indifférent et plus intuitif*

Interpréteur Python :

```
>>> print("{nom} a {age} ans.".format(age=30, nom="Tom"))
Tom a 30 ans.
>>> autrenom = "Bill"
>>> autreage = 40
>>> print("{nom} a {age} ans.".format(age=autreage, nom=autrenom.upper()))
BILL a 40 ans.
```

⇒ *les valeurs peuvent provenir d'autres variables ou de fonctions / méthodes*

On peut combiner indices et noms : print("{nom} a {0} ans.".format("30", nom="Tom"))

Il existe aussi de **nombreuses syntaxes de mise en forme des valeurs :**

Interpréteur Python :

```
>>> nb = 123.456789
>>> print("Le nombre vaut : {nombre:.3f}".format(nombre=nb))
Le nombre vaut : 123.457
>>> nb = 123
>>> print("Le nombre vaut : {nombre:10d}".format(nombre=nb))
Le nombre vaut :        123
>>> print("Le nombre vaut : {nombre:0>10d}".format(nombre=nb))
Le nombre vaut : 0000000123
>>> nom = "Tom"
>>> print("{0:10} complété par des espaces (sur 10 car.).".format(nom))
Tom        complété par des espaces (sur 10 car.).
```

Python 3.6 a également introduit les **chaînes littérales formatées ou « f-chaînes » (f-strings) :**

```
nom = "Tom" ; age = 30 ; chaine = f"{nom.upper()} a {age} ans." ; print(chaine)
nombre = 123.456789 ; print(f"Le nombre vaut : {nombre:.3f}")
nombre = 123 ; print(f"Le nombre vaut : {nombre:10d}")
nombre = 123 ; print(f"Le nombre vaut : {nombre:0>10d}")
nom = "Tom" ; print(f"{nom:10} complété par des espaces (sur 10 car.).")
```

2.2.4 Concaténation classique

Le signe **« + » sert également à concaténer des chaînes de caractères :**

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> chaine = nom + " a " + age + " ans."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
>>> chaine = nom + " a " + str(age) + " ans."
>>> print(chaine)
Tom a 30 ans.
```

Les autres types doivent être convertis en chaîne de caractères avec la méthode « str(…) ».

⇒ *Python est un langage fortement typé*

2.2.5 Parcours de chaînes

Rappel : une chaîne de caractères est une séquence de caractères (donc aussi de chaînes).

On peut directement la parcourir **avec une boucle « for » :**

Code Python :

```
chaine = "Chaîne de caractères"
for caractere in chaine:
    print(caractere)
```

On peut également **accéder à chaque caractère d'une chaîne par indice (commençant à 0) :**

Interpréteur Python :

```
>>> chaine = "Chaîne de caractères"
>>> chaine[0]     # premier caractère
'C'
>>> chaine[3]     # quatrième caractère
'î'
>>> chaine[-1]     # dernier caractère (accès par la fin avec indice négatif)
's'
```

Pour obtenir le nombre de caractères d'une chaîne (longueur) : **len(chaine)**

⇒ *ce n'est pas une méthode de la classe « str », mais une fonction pour différents types d'objets*

On peut donc aussi parcourir une chaîne de caractères **avec une boucle « while » :**

Code Python :

```
chaine = "Chaîne de caractères"
indice = 0
while indice < len(chaine):
    print(chaine[indice])
    indice += 1
```

⇒ *accéder à un indice inexistant lève une exception de type « IndexError »*

Mais on ne peut pas modifier un caractère de chaîne par indice : ⇒ *objet « immutable »*

Interpréteur Python :

```
>>> chaine[5] = "s"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

2.2.6 Découpage de chaînes

Le découpage / slicing des chaînes de caractères **se fait par une autre syntaxe entre crochets :**

[début:fin] ou **[début:fin:pas]** ⇒ *début commence à 0 et fin correspond au dernier élément*
⇒ *valeurs négatives et variables / fonctions possibles*

Interpréteur Python :

```
>>> chaine = "Chaîne de caractères"
>>> chaine[:3]     # ou : chaine[0:3]
'Cha'
>>> chaine[3:]     # ou : chaine[3:len(chaine)]
'îne de caractères'
>>> chaine[12:16]
'ract'
>>> chaine[-8:-4]     # par la fin
'ract'
>>> chaine[::2]     # 1 sur 2
'Can ecrcèe'
>>> chaine[4:17:3]     # avec 1 sur 3
'ndcaè'
>>> chaine[::-1]
'serètcarac ed enîahC'
```

2.2.7 Compter, rechercher, remplacer

count : compte le nombre d'occurrences de la sous-chaîne dans la chaîne

find : recherche la première occurrence de la sous-chaîne dans la chaîne

replace : remplace les occurrences de la sous-chaîne dans la chaîne

Interpréteur Python :

```
>>> chaine = "Exemple de texte"
>>> chaine.count("te")
2
>>> chaine.find("te")     # commence à 0, -1 si non trouvé
11
>>> chaine.replace("te", "pa")
'Exemple de paxpa'
```

⇒ *ne modifie pas la chaîne d'origine*

2.3 Listes et tuples

2.3.1 Classe list

Les listes sont aussi **des séquences. Des objets capables de contenir d'autres objets** de n'importe quel type (même d'autres listes) et sans limite de taille.

Elles sont **créées (instanciées) à partir de la classe « list » :**

Interpréteur Python :

```
>>> liste = list()     # créée vide, équivalent à : liste = []
>>> type(liste)
<class 'list'>
>>> liste
[]
>>> liste = ["chaîne", 123, 9.99, []]     # créée avec des objets (de tous types)
>>> print(liste)
['chaîne', 123, 9.99, []]
```

⇒ *les objets d'une liste doivent être entre crochets « [ ] » et séparés de virgules « , »*

2.3.2 Accès et modification des objets

On accède aux objets d'une liste avec leur indice entre crochets (commençant à 0) :

Interpréteur Python :

```
>>> liste = ["chaîne", 123, 9.99, []]     # créée avec des objets (de tous types)
>>> print(liste)
['chaîne', 123, 9.99, []]
>>> liste = [321, "texte", 9.99, "a", "b"]     # réaffectée / écrasée (recréée)
>>> print(liste)
[321, 'texte', 9.99, 'a', 'b']
>>> liste[0]
321
>>> liste[3]
'a'
>>> liste[3] = "aaa"
>>> liste
[321, 'texte', 9.99, 'aaa', 'b']
```

Contrairement aux chaînes de caractères, **les listes sont des objets « muables / mutables »**.

2.3.3 Insertion d'objets

Ne fonctionne pas :

Interpréteur Python :

```
>>> liste = ["chaîne", 123]
>>> liste[1]
123
>>> liste[2] = 9.99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Pour ajouter un objet à la fin d'une liste, il faut passer par la méthode « append » :

Interpréteur Python :

```
>>> liste.append(9.99)
>>> liste
['chaîne', 123, 9.99]
```

⇒ *« append » est une méthode de la classe « list » (voir « help(list) »)*

« append » modifie l'objet mais ne renvoie rien. ⇒ *contraire des chaînes de caractères*

Attention au comportement des méthodes des classes :

Interpréteur Python :

```
>>> chaine1 = "TEST"
>>> chaine2 = chaine1.lower()     # renvoie un résultat (nouvel objet)
>>> chaine1
'TEST'
>>> chaine2
'test'
>>> liste1 = ["TEST"]
>>> liste2 = liste1.append(123)     # ne renvoie rien (modifie l'objet lui-même)
>>> liste1
['TEST', 123]
>>> liste2
>>> print(liste2)
None
```

« None » correspond à l'objet vide (rien) en Python.

Rappel :
Les chaînes de caractères sont des objets « immutables » et les listes des objets « mutables ».

Pour insérer un objet dans une liste, il faut passer par la méthode « insert » :

Interpréteur Python :

```
>>> liste = ["chaîne", 123, 9.99]
>>> liste.insert(2, "aaa")
>>> liste
['chaîne', 123, 'aaa', 9.99]
```

⇒ *en précisant l'indice concerné (décale la suite)*

2.3.4 Concaténation de listes

Interpréteur Python :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = ["a", "b", "c"]
>>> liste1 += liste2     # avec la concaténation classique "+" (liste1 = liste1 + liste2)
>>> print(liste1)
[1, 2, 3, 'a', 'b', 'c']
```

Interpréteur Python :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = ["a", "b", "c"]
>>> liste1.extend(liste2)     # avec la méthode "extend" de la classe "list"
>>> print(liste1)
[1, 2, 3, 'a', 'b', 'c']
```

2.3.5 Suppression d'objets

Le mot-clé « del » permet de supprimer un objet, variable ou autre, exemple : **del variable**

⇒ *ce n'est pas une méthode de la classe « list », mais un mot-clé général (interne à l'interpréteur)*

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> liste
['chaîne', 123, 'aaa', 9.99]
>>> del liste[2]     # on précise l'indice concerné
>>> liste
['chaîne', 123, 9.99]
```

La méthode « remove » de la classe « list » permet aussi de **supprimer un objet d'une liste mais en précisant sa valeur :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99, 123]
>>> liste
['chaîne', 123, 'aaa', 9.99, 123]
>>> liste.remove(123)
>>> liste
['chaîne', 'aaa', 9.99, 123]
>>> liste.remove(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

⇒ *supprime uniquement la première occurrence trouvée*

2.3.6 Parcours de listes

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> for element in liste:
...     print(element)
...
chaîne
123
aaa
9.99
>>> indice = 0
>>> while indice < len(liste):
...     print(liste[indice])
...     indice += 1
...
chaîne
123
aaa
9.99
```

⇒ *équivalent aux chaînes de caractères ; « len » (longueur) fonctionne également sur les listes*

Dans une boucle « for / in », **« enumerate »** (c'est une classe) **prend en paramètre la liste parcourue et retourne pour chaque élément un tuple sous la forme « (indice, objet) » :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> for element in enumerate(liste):
...         print(element)
...
(0, 'chaîne')
(1, 123)
(2, 'aaa')
(3, 9.99)
```

⇒ *les tuples sont détaillés plus loin*

Leurs valeurs peuvent être récupérées dans des variables : ⇒ *parenthèses inutiles*

Code Python :

```
for indice, objet in enumerate(liste):
    print("Pour l'indice {} : {}".format(indice, objet))
```

Autres exemples de parcours de listes (qui incluent d'autres listes) :

Interpréteur Python :

```
>>> panier = [
... ["Pommes", 6, 1.50],
... ["Oeufs", 12, 3.00],
... ["Tomates", 10, 3.50]
... ]
>>> for produit, quantite, prix in panier:
...         print(f"{quantite} {produit} pour {prix:.2f} euros.")
...
6 Pommes pour 1.50 euros.
12 Oeufs pour 3.00 euros.
10 Tomates pour 3.50 euros.
```

Interpréteur Python :

```
>>> for element in enumerate(panier):
...         print(element)
...
(0, ['Pommes', 6, 1.5])
(1, ['Oeufs', 12, 3.0])
(2, ['Tomates', 10, 3.5])
```

2.3.7 Tuples

Les tuples sont des **listes (séquences) « immuables / immutables », c'est-à-dire non modifiables**. Ils sont créés (instanciés) à partir de la **classe « tuple » :**

Interpréteur Python :

```
>>> liste_tuple = tuple()     # créé vide (en soi inutile...), équivalent à : liste_tuple = ()
>>> type(liste_tuple)
<class 'tuple'>
>>> liste_tuple
()
>>> liste_tuple = ("chaîne", 123, 9.99)     # (re)créé avec des objets
>>> liste_tuple = (321, "texte", 9.99, "a", "b")     # réaffecté (recréé), non modifié !
>>> print(liste_tuple)
(321, 'texte', 9.99, 'a', 'b')
>>> liste_tuple[0]     # accès avec indice entre crochets (comme une liste)
321
>>> liste_tuple[3] = "aaa"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

⇒ *les objets d'un tuple doivent être entre parenthèses « ( ) » et séparés de virgules « , »*

Une fois créé, **on ne peut plus y ajouter d'objets ou en retirer, ni en modifier** (uniquement le réaffecter / recréer intégralement).

Pour le reste, un tuple s'utilise comme une liste (accès, parcours,…).

Aide : **help(tuple)**
Nombre d'éléments : **len(liste_tuple)**
Concaténation de tuples : **tuple1 += tuple2** ⇒ *ou : tuple1 = tuple1 + tuple2*
Suppression d'un tuple : **del liste_tuple** ⇒ *mais pas d'un élément !*

Interpréteur Python :

```
>>> liste_tuple = (321, "texte", 9.99, "a", "b")
>>> liste_tuple
(321, 'texte', 9.99, 'a', 'b')
>>> liste_tuple = list(liste_tuple)     # conversion en liste
>>> liste_tuple
[321, 'texte', 9.99, 'a', 'b']
>>> liste_tuple = tuple(liste_tuple)     # conversion en tuple
>>> liste_tuple
(321, 'texte', 9.99, 'a', 'b')
```

Les affectations multiples passent par des tuples :

Interpréteur Python :

```
>>> a, b, c = 3, 2, 1
>>> print(c, b, a)
1 2 3
>>> (a, b, c) = (3, 2, 1)
>>> print(c, b, a)
1 2 3
```

⇒ *les parenthèses sont optionnelles (et ensuite les variables sont bien modifiables)*

Interpréteur Python :

```
>>> def fonction():
...     a = 10 ; b = 20 ; c = 30
...     return a, b, c
...
>>> var1, var2, var3 = fonction()
>>> print(var1, var2, var3)
10 20 30
>>> retour = fonction()
>>> print(retour)
(10, 20, 30)
```

2.3.8 Passage d'une chaîne à une liste et inversement

Passage d'une chaîne à une liste avec la méthode « split » de la classe « str » :

Interpréteur Python :

```
>>> chaine = "Sébastien;MEYER;10, Avenue des Vosges;67000;STRASBOURG"
>>> chaine.split(";")
['Sébastien', 'MEYER', '10, Avenue des Vosges', '67000', 'STRASBOURG']
```

⇒ *« split » prend en paramètre le séparateur (par défaut : espaces, tabulations et sauts de ligne)*

Passage d'une liste à une chaîne avec la méthode « join » de la classe « str » :

Interpréteur Python :

```
>>> liste = ['Sébastien', 'MEYER', '10, Avenue des Vosges', '67000', 'STRASBOURG']
>>> ";".join(liste)
'Sébastien;MEYER;10, Avenue des Vosges;67000;STRASBOURG'
```

⇒ *« join » est appliquée à la chaîne servant de séparateur (peut être une variable)*

Et aussi :

Interpréteur Python :

```
>>> chaine = "a---b---c"
>>> var1, var2, var3 = chaine.split("---")
>>> print(var1, var2, var3)
a b c
>>> chaine = "***".join([var1, var2, var3])
>>> print(chaine)
a***b***c
```

2.3.9 Fonction avec un nombre indéterminé de paramètres

Comme la fonction « print ».

Pour définir une fonction avec un nombre indéterminé de paramètres, il suffit de **prévoir un paramètre précédé d'une étoile (*) qui recevra dans un tuple la liste des paramètres transmis :**

def nom_fonction(*parametres):

Interpréteur Python :

```
>>> def fonction(*parametres):
...     print("Paramètres transmis :", parametres)
...
>>> fonction()
Paramètres transmis : ()
>>> fonction(123)
Paramètres transmis : (123,)
>>> fonction(123, "aaa", 9.99)
Paramètres transmis : (123, 'aaa', 9.99)
>>> variable = 456
>>> fonction(variable, variable+1000, [1, 2, "a", "b"])
Paramètres transmis : (456, 1456, [1, 2, 'a', 'b'])
```

Les appels avec paramètres nommés ne sont pas gérés (abordés plus loin avec les dictionnaires).

On peut définir une **fonction avec paramètres obligatoires (classiques) suivis de paramètres variables (toujours après) :**

def nom_fonction(parametre1, parametre2,… , *parametres):

Interpréteur Python :

```
>>> def fonction(parametre1, parametre2, *parametres):
...         print("Paramètres transmis : {}, {} et {}".format(parametre1, parametre2, parametres))
...
>>> fonction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fonction() missing 2 required positional arguments: 'parametre1' and 'parametre2'
>>> fonction(123, "aaa")
Paramètres transmis : 123, aaa et ()
>>> fonction(123, "aaa", 9.99, "etc")
Paramètres transmis : 123, aaa et (9.99, 'etc')
```

Et si besoin, **les paramètres avec valeur par défaut doivent être définis en dernier :**

Interpréteur Python :

```
>>> def fonction(parametre1, parametre2, *parametres, defaut1="valeur1", defaut2="valeur2"):
...         print("Paramètres transmis : {}, {} et {}".format(parametre1, parametre2, parametres))
...         print("Et paramètres par défaut :", defaut1, "et", defaut2)
...
>>> fonction(123, "aaa", 9.99, "etc")
Paramètres transmis : 123, aaa et (9.99, 'etc')
Et paramètres par défaut : valeur1 et valeur2
```

2.3.10 Transformer une liste ou un tuple en paramètres de fonction

Lors d'un appel de fonction (ou de méthode), on peut **transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre**, et non en tant qu'objet liste (ou tuple), **en précédant son nom également d'une étoile (*) :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> print(liste)     # appel classique : un seul paramètre transmis (l'objet liste)
['chaîne', 123, 'aaa', 9.99]
>>> print(*liste)     # appel en transmettant les éléments de l'objet liste paramètre par paramètre
chaîne 123 aaa 9.99
```

⇒ *utile dans certains cas de traitements*

Conclusion :

Dans la définition d'une fonction : l'étoile (*) permet de recevoir des paramètres variables dans un seul paramètre sous forme de tuple.

Lors de l'appel d'une fonction, c'est l'inverse : l'étoile (*) permet de transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre.

2.3.11 Compréhensions de liste

Une compréhension de liste (« list comprehension » en anglais) **permet de générer une liste à partir du parcours d'une autre liste tout en la modifiant et/ou la filtrant**. ⇒ *très puissant*

Sa syntaxe est donc **entre crochets « [ ] »**.

Parcours simple avec modification :

Interpréteur Python :

```
>>> nombres = [-9, -2, 0, 3, 5, 7]
>>> [n*10 for n in nombres]
[-90, -20, 0, 30, 50, 70]
```

⇒ *la boucle « for » parcourt la liste « nombres » et pour chaque élément (n) génère le résultat « n*10 » renvoyé en liste*

(peut également parcourir un tuple)

Parcours avec filtrage et modification :

Interpréteur Python :

```
>>> nombres = [-9, -2, 0, 3, 5, 7]
>>> [n*10 for n in nombres if n >= 0]
[0, 30, 50, 70]
```

⇒ *dans la boucle « for », le « if » permet de conditionner (filtrer) les valeurs souhaitées*

(plus simple que de produire le résultat avec des structures classiques imbriquées)

Autre exemple (avec tri d'une liste de tuples) : ⇒ *fonction « sorted » ou méthode liste « sort »*

Interpréteur Python :

```
>>> panier = [("Pommes", 6, 1.50), ("Oeufs", 12, 3.00), ("Tomates", 10, 3.50)]
>>> panier_prix = [(pri, pro, qte) for pro, qte, pri in panier]
>>> panier_prix     # prix en premier pour le tri
[(1.5, 'Pommes', 6), (3.0, 'Oeufs', 12), (3.5, 'Tomates', 10)]
>>> panier = [(pro, qte, pri) for pri, pro, qte in sorted(panier_prix, reverse=True)]
>>> panier     # nouvelle liste triée d'après prix décroissant
[('Tomates', 10, 3.5), ('Oeufs', 12, 3.0), ('Pommes', 6, 1.5)]
```

2.4 Dictionnaires et ensembles

2.4.1 Classe dict

Comme les listes, les dictionnaires sont **des conteneurs. Des objets capables de contenir d'autres objets** de n'importe quel type et sans limite de taille.

Mais contrairement aux listes, ils **ne sont pas ordonnés et pour accéder aux objets on utilise des clés** (chaînes ou autres, voire indices entiers). ⇒ *couples « clés / objets »*

Ils sont **créés (instanciés) à partir de la classe « dict » :**

Interpréteur Python :

```
>>> dictionnaire = dict()     # créé vide, équivalent à : dictionnaire = {}
>>> type(dictionnaire)
<class 'dict'>
>>> dictionnaire
{}
```

Les dictionnaires sont des objets « muables / mutables ».

2.4.2 Insertion, accès et modification des objets

Pour référencer un objet dans un dictionnaire (existant), on précise le nom du dictionnaire avec la clé entre crochets « [ ] » :

Interpréteur Python :

```
>>> dictionnaire["nom"] = "Tom"     # affecte la valeur (objet) "Tom" à la clé "nom"
>>> dictionnaire["age"] = 30
>>> print(dictionnaire["nom"], ",", dictionnaire["age"])
Tom , 30
>>> dictionnaire["age"] = 40     # modifie la clé "age"
>>> dictionnaire["taille"] = 1.75     # ajoute la clé "taille"
>>> dictionnaire
{'nom': 'Tom', 'age': 40, 'taille': 1.75}
>>> dictionnaire = {"nom": "Tom", "age": 30}     # entièrement réaffecté (recréé)
>>> dictionnaire
{'nom': 'Tom', 'age': 30}
```

⇒ *les couples « clés / objets » d'un dictionnaire doivent être entre accolades « { } »*
⇒ *séparés de virgules « , » et écrits sous la forme « clé: objet »*
⇒ *on peut utiliser quasiment tous les types comme clés et tous les types comme objets*

Les clés peuvent être des tuples, par exemple pour une matrice de points binaires :

Interpréteur Python :

```
>>> matrice = {}
>>> matrice[(1, 'a')] = 0
>>> matrice[(1, 'b')] = 1
>>> matrice[(1, 'c')] = 1
>>> matrice[(2, 'a')] = 1
>>> matrice[(2, 'b')] = 0
>>> matrice[(2, 'c')] = 1
>>> matrice[(3, 'a')] = 1
>>> matrice[(3, 'b')] = 0
>>> matrice[(3, 'c')] = 1
>>> print(matrice)
{(1, 'a'): 0, (1, 'b'): 1, (1, 'c'): 1, (2, 'a'): 1, (2, 'b'): 0, (2, 'c'): 1, (3, 'a'): 1, (3, 'b'): 0, (3, 'c'): 1}
```

⇒ *les parenthèses « ( ) » des tuples entre crochets « [ ] » sont optionnelles (mais plus explicites)*

2.4.3 Suppression d'objets

Avec le mot-clé « del » (déjà vu) ou la méthode « pop » de la classe « dict » :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 30, "taille": 1.75, "métier": "Informaticien"}
>>> dictionnaire
{'nom': 'Tom', 'age': 30, 'taille': 1.75, 'métier': 'Informaticien'}
>>> del dictionnaire["taille"]     # on précise la clé
>>> dictionnaire
{'nom': 'Tom', 'age': 30, 'métier': 'Informaticien'}
>>> dictionnaire.pop("métier")     # on précise également la clé
'Informaticien'
>>> dictionnaire
{'nom': 'Tom', 'age': 30}
```

⇒ *la méthode « pop » renvoie la valeur de la clé supprimée (peut être utile)*

2.4.4 Application : dictionnaire de fonctions

Rappel : « print » correspond à la référence de la fonction et « print() » à son appel.

Interpréteur Python :

```
>>> print
<built-in function print>
>>> print()

>>>
```

On peut donc **stocker la référence d'une fonction dans une variable :**

Interpréteur Python :

```
>>> afficher = print
>>> afficher("Un texte")
Un texte
```

⇒ *la variable « afficher » possède la même référence que la fonction « print »*
⇒ *quand on utilise la variable « afficher », elle appelle donc la fonction « print »*

On peut aussi **stocker les références de fonctions dans n'importe quel objet conteneur, listes, dictionnaires, etc :**

Interpréteur Python :

```
>>> def fonction1():
...       print("Appel fonction1")
...
>>> def fonction2():
...       print("Appel fonction2")
...
>>> fonctions = {}
>>> fonctions["f1"] = fonction1
>>> fonctions["f2"] = fonction2
>>> fonctions["f1"]     # fait référence à fonction1
<function fonction1 at 0x7f1f38a063a0>
>>> fonctions["f1"]()     # fait appel à fonction1
Appel fonction1
>>> fonctions["f2"]     # fait référence à fonction2
<function fonction2 at 0x7f1f38a06430>
>>> fonctions["f2"]()     # fait appel à fonction2
Appel fonction2
```

2.4.5 Parcours de dictionnaires

Le parcours classique ne retourne que les clés du dictionnaire :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>> for element in dictionnaire:
...         print(element)
...
nom
age
taille
```

Précision :

Les dictionnaires n'étant pas ordonnés, les clés retournées ne sont pas forcément dans l'ordre de saisie (ou alphanumérique).

⇒ *dépend de la gestion interne des dictionnaires*

La méthode « keys » de la classe « dict » retourne également les clés : ⇒ *à privilégier*

Interpréteur Python :

```
>>> dictionnaire.keys()
dict_keys(['nom', 'age', 'taille'])
>>> for cle in dictionnaire.keys():
...     print(cle)
...
nom
age
taille
```

Et la méthode « values » de la classe « dict » retourne les valeurs (objets) :

Interpréteur Python :

```
>>> dictionnaire.values()
dict_values(['Tom', 40, 1.75])
>>> for valeur in dictionnaire.values():
...     print(valeur)
...
Tom
40
1.75
```

Utilisables également **pour les recherches de clés ou de valeurs / objets :**

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>> cle = "age"
>>> if cle in dictionnaire.keys():
...         print(cle, "est présent.")
...
age est présent.
>>>
>>> valeur = 1.75
>>> if valeur in dictionnaire.values():
...         print(valeur, "est présent.")
...
1.75 est présent.
```

Enfin, **la méthode « items » de la classe « dict » retourne les couples « clés / objets » :**

Interpréteur Python :

```
>>> dictionnaire.items()
dict_items([('nom', 'Tom'), ('age', 40), ('taille', 1.75)])
>>>
>>> for element in dictionnaire.items():
...         print(element)
...
('nom', 'Tom')
('age', 40)
('taille', 1.75)
>>>
>>> for cle, valeur in dictionnaire.items():     # parenthèses inutiles autour de cle, valeur
...         print(cle, "=", valeur)
...
nom = Tom
age = 40
taille = 1.75
```

⇒ *retourne pour chaque élément un tuple sous la forme « (clé, objet) »*

2.4.6 Fonction avec un nombre indéterminé de paramètres nommés

Rappel : dans la définition d'une fonction, l'étoile (*) permet de recevoir des paramètres variables dans un seul paramètre sous forme de tuple, sauf les paramètres nommés.

Deux étoiles (**) permettent de recevoir des paramètres nommés variables dans un seul paramètre sous forme de dictionnaire (sauf les paramètres non nommés) :

Interpréteur Python :

```
>>> def fonction(**parametres_nommes):
...     print("Paramètres nommés transmis :", parametres_nommes)
...
>>> fonction()
Paramètres nommés transmis : {}
>>> fonction(entier=123, chaine="aaa", decimal=9.99)
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
>>> fonction(456, "zzz", entier=123, chaine="aaa", decimal=9.99)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fonction() takes 0 positional arguments but 2 were given
```

Et pour recevoir des paramètres variables non nommés et nommés (attention à l'ordre) :

Interpréteur Python :

```
>>> def fonction(*parametres, **parametres_nommes):     # obligatoirement dans cet ordre
...         print("Paramètres non nommés transmis :", parametres)
...         print("Paramètres nommés transmis :", parametres_nommes)
...
>>> fonction()
Paramètres non nommés transmis : ()
Paramètres nommés transmis : {}
>>> fonction(456, "zzz")
Paramètres non nommés transmis : (456, 'zzz')
Paramètres nommés transmis : {}
>>> fonction(entier=123, chaine="aaa", decimal=9.99)
Paramètres non nommés transmis : ()
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
>>> fonction(456, "zzz", entier=123, chaine="aaa", decimal=9.99)
Paramètres non nommés transmis : (456, 'zzz')
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
```

2.4.7 Transformer un dictionnaire en paramètres nommés de fonction

Rappel : lors de l'appel d'une fonction, l'étoile (*) permet de transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre.

Deux étoiles (**) permettent de transmettre les éléments d'un dictionnaire paramètre nommé par paramètre nommé :

Interpréteur Python :

```
>>> dictionnaire = {"sep": " - ", "end": ".\n"}
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", dictionnaire)
dictionnaire en paramètres de fonction {'sep': ' - ', 'end': '.\n'}
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", **dictionnaire)
dictionnaire - en - paramètres - de - fonction.
>>>
>>> # Equivalent à :
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", sep=" - ", end=".\n")
dictionnaire - en - paramètres - de - fonction.
```

⇒ *la fonction « print » dispose des paramètres nommés :*
⇒ *« sep » (séparateur des paramètres affichés) et « end » (fin d'affichage)*

2.4.8 Ensembles (sets)

Cas particulier : ⇒ *voir « help(set) »*

Un ensemble est une **collection d'éléments non ordonnée, sans index (indices ou clés) et qui ne peut pas posséder d'éléments dupliqués** (il supprime automatiquement les doublons).

C'est également un **objet conteneur, de la classe « set », dont les objets sont entre accolades « { } » et séparés de virgules « , » :**

Interpréteur Python :

```
>>> ensemble = {"Tom", "Bill", "Bob", "Tom", "Tom", "Bill"}
>>> ensemble
{'Bob', 'Tom', 'Bill'}
>>> type(ensemble)
<class 'set'>
>>> for objet in ensemble:
...     print(objet)
...
Bob
Tom
Bill
```

Il n'est **pas possible d'accéder aux objets d'un ensemble avec l'opérateur « [ ] »**.
⇒ *mais on peut convertir un ensemble en liste avec « liste = list(ensemble) »*

Attention, **pour créer un ensemble vide, il faut utiliser la fonction « set() » :**

Interpréteur Python :

```
>>> ensemble = {}
>>> type(ensemble)
<class 'dict'>
>>> ensemble = set()
>>> type(ensemble)
<class 'set'>
```

Pour ajouter et supprimer des objets dans un ensemble (existant) : ⇒ *objet « mutable »*

Interpréteur Python :

```
>>> ensemble = {"Bob", "Tom", "Bill"}
>>> ensemble.add("Jim")
>>> ensemble
{'Bill', 'Tom', 'Jim', 'Bob'}
>>> ensemble.remove("Jim")
>>> ensemble
{'Bill', 'Tom', 'Bob'}
```

2.5 Référence des objets

Rappels et principes :

Les nombres, les chaînes de caractères,… sont des **objets « immuables / immutables » : leurs méthodes ne modifient pas l'objet** (mais renvoient un nouvel objet modifié).

Les listes, les dictionnaires,… sont des **objets « muables / mutables » : leurs méthodes modifient directement l'objet** (en fonction de l'objectif de la méthode).

Une variable est en fait un nom identifiant, pointant vers la référence de l'objet (sa position en mémoire Python ⇒ *"adresse"*).

La fonction intégrée **« id » retourne la référence de l'objet**.

L'opérateur **« == » compare les valeurs / contenus**.

Le mot-clé **« is » compare les références**.

Interpréteur Python :

```
>>> var1 = "essai"
>>> var2 = var1     # ont la même référence d'objet
>>> var1 ; id(var1)
'essai'
140227330927152
>>> var2 ; id(var2)
'essai'
140227330927152
>>> var1 == var2
True
>>> var1 is var2
True
>>> var1 += ", essai2"     # ne modifie pas l'objet, nouvel objet créé !
>>> var1 ; id(var1)
'essai, essai2'
140227330928048
>>> var2 ; id(var2)
'essai'
140227330927152
>>> var1 == var2
False
>>> var1 is var2
False
```

Interpréteur Python :

```
>>> liste1 = ["essai"]
>>> liste2 = liste1     # ont la même référence d'objet (mais pas : liste2 = list(liste1) )
>>> liste1 ; id(liste1)
['essai']
140586784186944
>>> liste2 ; id(liste2)
['essai']
140586784186944
>>> liste1 == liste2
True
>>> liste1 is liste2
True
>>> liste1.append("essai2")     # modifie directement l'objet
>>> liste1 ; id(liste1)
['essai', 'essai2']
140586784186944
>>> liste2 ; id(liste2)
['essai', 'essai2']
140586784186944
>>> liste1 == liste2
True
>>> liste1 is liste2
True
```

2.6 Fichiers

2.6.1 Module os

Le module « os » **permet d'effectuer des opérations courantes liées au système d'exploitation** (très riche en fonctionnalités, notamment en gestion de fichiers).

Il est **indépendant du système d'exploitation** et peut fonctionner sur n'importe quel OS.

Importer le module « os » : **import os**
Aide et fonctionnalités : **help(os)** ⇒ *liste : dir(os)*

2.6.2 Répertoire courant

Changer de répertoire courant : **os.chdir("*chemin*")** ⇒ *avec chemin absolu ou relatif*
⇒ *et des « / » (même sur Windows)*

Obtenir le répertoire courant : **os.getcwd()** ⇒ *depuis la racine (chemin absolu)*

2.6.3 Ouverture d'un fichier

La fonction intégrée **« open » (sans import) ouvre un fichier et le renvoie en tant qu'objet fichier de la classe « _io.TextIOWrapper »** pour lecture, écriture ou modification (à l'aide de méthodes dédiées).

Syntaxe générale :

open("*chemin/fichier*", mode) ⇒ *avec chemin absolu ou relatif*

Avec pour modes :

r (read) : ouvre le fichier en lecture (mode par défaut), erreur si inexistant
w (write) : ouvre le fichier en écriture, le crée si inexistant sinon l'écrase
x (create) : crée le fichier et l'ouvre en écriture, erreur si existant
a (append) : ouvre le fichier en ajout (à la fin), le crée si inexistant
b (binary) : après r/w/x/a ouvre le fichier en mode binaire
t (text) : après r/w/x/a ouvre le fichier en mode texte (mode par défaut)
+ : après r/w/x/a ouvre le fichier en mise à jour (lecture et écriture)

Ouvrir en lecture le fichier texte « fichier.txt » (existant dans le répertoire courant) :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> fichier
<_io.TextIOWrapper name='fichier.txt' mode='r' encoding='UTF-8'>
>>> type(fichier)
<class '_io.TextIOWrapper'>
```

2.6.4 Fermeture d'un fichier

Une fois utilisé, **un fichier doit être fermé à l'aide de la méthode « close » :**

Interpréteur Python :

```
>>> fichier.close()
```

⇒ *libère les ressources liées au fichier*
⇒ *en écriture, les modifications sont réellement écrites sur disque au moment du « close »*

2.6.5 Lecture d'un fichier

Par défaut, la méthode « read » de la classe « TextIOWrapper » lit l'intégralité d'un fichier :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> contenu = fichier.read()
>>> contenu
'Une ligne\nUne autre ligne\nEncore une ligne\n'
>>> print(contenu)
Une ligne
Une autre ligne
Encore une ligne

>>> fichier.close()
```

On peut tout faire avec la variable récupérant le contenu (ici du texte), par exemple la "splitter" en liste avec la méthode « split » afin de traiter ses lignes individuellement.

⇒ *la taille maximum est limitée par la mémoire*
⇒ *« read » peut prendre en paramètre la taille maximum lue (nombre de caractères / octets)*
⇒ *une boucle avec « fichier.read(1) » peut lire le fichier caractère par caractère*

La méthode « splitlines » permet de lire l'intégralité d'un fichier directement dans une liste :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> liste = fichier.read().splitlines()
>>> liste
['Une ligne', 'Une autre ligne', 'Encore une ligne']
>>> fichier.close()
```

Pour lire un fichier ligne par ligne : ⇒ *allège la mémoire*

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> for ligne in fichier:
...     ligne = ligne.strip()    # retire le retour à la ligne (\n)
...     print(ligne)
...
Une ligne
Une autre ligne
Encore une ligne
>>> fichier.close()
```

La méthode « readline » lit uniquement la ligne suivante du fichier :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> fichier.readline()
'Une ligne\n'
>>> fichier.readline().strip()     # retire le retour à la ligne (\n)
'Une autre ligne'
>>> fichier.readline().strip()
'Encore une ligne'
>>> fichier.readline().strip()
''
>>> fichier.close()
```

2.6.6 Ecriture dans un fichier

La méthode « write » de la classe « TextIOWrapper » permet d'écrire dans un fichier.

Elle **n'accepte en paramètre que des chaînes de caractères**. ⇒ *convertir les autres types* (ou du binaire avec le mode « b »)

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "w")
>>> fichier.write("Ligne une\n")
10
>>> fichier.write("Ligne deux\n")
11
>>> fichier.write("Ligne trois\n")
12
>>> fichier.close()
```

⇒ *« write » retourne le nombre de caractères écrits*
⇒ *le mode « w » écrase le fichier, le mode « a » ajoute à la fin du fichier*

2.6.7 Position dans le fichier

La méthode « seek » permet de changer la position dans le fichier.

La méthode « tell » permet de connaître la position dans le fichier.

⇒ *à partir de 0 (zéro)*

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r+")     # lecture avec mise à jour
>>> fichier.read()
'Ligne une\nLigne deux\nLigne trois\n'
>>> fichier.seek(16)
16
>>> fichier.tell()
16
>>> fichier.read(4)
'deux'
>>> fichier.tell()
20
>>> fichier.seek(16)
16
>>> fichier.write("XXXX")
4
>>> fichier.tell()
20
>>> fichier.seek(0)
0
>>> fichier.read()
'Ligne une\nLigne XXXX\nLigne trois\n'
>>> fichier.close()
```

2.6.8 Ouverture avec with, as

Ouvrir un fichier avec **« with, as » évite de le fermer avec la méthode « close »**.

Si une erreur se produit dans le bloc de traitement du fichier, l'exception se lève et le fichier est automatiquement fermé. ⇒ *fiabilise les traitements et plus court qu'un « try, finally »*

Interpréteur Python :

```
>>> with open("fichier.txt", "r") as fichier:
...         print(fichier.read())
...
Ligne une
Ligne XXXX
Ligne trois

>>> fichier.closed
True
>>> print(fichier.read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

2.6.9 Ecriture et lecture d'objets

Le module « pickle » dispose de fonctionnalités permettant d'écrire et de lire des objets dans des fichiers, à importer au préalable :

import pickle

Il possède les **classes « Pickler » pour écrire un objet et « Unpickler » pour lire un objet**.

Un objet de la classe « Pickler » permet d'écrire un objet dans un fichier à l'aide de la méthode « dump » :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>>
>>> with open("objet-dictionnaire", "wb") as fichier:     # écriture obligatoirement en binaire
...         mon_pickler = pickle.Pickler(fichier)     # crée l'objet pickler pour le fichier
...         mon_pickler.dump(dictionnaire)     # écrit l'objet dans le fichier grâce à l'objet pickler
```

⇒ *on peut écrire avec des « dump » plusieurs objets à la suite dans le fichier*

Un objet de la classe « Unpickler » permet de lire un objet d'un fichier à l'aide de la méthode « load » :

Interpréteur Python :

```
>>> with open("objet-dictionnaire", "rb") as fichier:     # lecture obligatoirement en binaire
...         mon_unpickler = pickle.Unpickler(fichier)     # crée l'objet unpickler pour le fichier
...         dictionnaire_lu = mon_unpickler.load()     # lit l'objet du fichier grâce à l'objet unpickler
...
>>> dictionnaire_lu
{'nom': 'Tom', 'age': 40, 'taille': 1.75}
```

⇒ *on peut lire avec des « load » plusieurs objets à la suite dans le fichier*

Le module « pickle » **implémente donc des protocoles binaires de sérialisation et dé-sérialisation d'objets Python**.

3 Programmation Orientée Objet

3.1 Bref rappel

Python est un **langage de programmation orienté objet**, avec sa propre philosophie (attention).

La Programmation Orientée Objet (POO) permet de modéliser les programmes sous forme d'objets comme ceux qui nous entourent, avec caractéristiques et fonctionnalités.

⇒ *programmation davantage modulable, moins linéaire*

<u>En programmation, un objet est avant tout une structure de données (en mémoire)</u> :

La principale différence vient du fait que **l'objet regroupe les données** (variables sous forme d'attributs) **et les moyens de traiter ces données** (fonctions sous forme de méthodes).

Variables, fonctions,… **tout est objet en Python !**

Enfin, **un objet est créé (instancié) à partir d'une classe** qui est un "moule à objets" (modèle).

⇒ *on commence par définir les classes desquelles on crée les objets de notre programme*

3.2 Définition d'une classe

3.2.1 Constructeur, attributs d'instances, instanciation

Au minimum, **pour définir une classe on utilise le mot-clé « class » suivi du nom de la classe et de deux points (:)**.

Avec la convention de nommage "Camel Case" : **NomDeLaClasse**

Comme toujours, **son bloc d'instructions doit être indenté**. Celui-ci contient alors la définition des attributs et des méthodes de la classe.

Un attribut d'instance et sa valeur sont propres à chaque objet instancié (créé).

En Python, les attributs d'instances **doivent être définis / initialisés dans le constructeur de la classe : méthode spéciale nommée « __init__ »** (entourée de 2 underscores « **__** ») ayant une définition similaire à une fonction.

Ce constructeur « __init__ » est systématiquement appelé lors de la création / instanciation d'un objet à partir de la classe (initialisant ainsi ses attributs d'instance).

Exemple basique de définition d'une classe :

Code Python :

```
class Personne:
    def __init__(self):     # self est obligatoire en paramètre (on pourrait le nommer autrement !)
        self.nom = "Tom"     # self est obligatoire pour un attribut d'instance (sinon variable locale !)
```

La classe « Personne » définit et initialise dans son constructeur « __init__ » un attribut d'instance nommé « nom » (initialisé ici avec une valeur en dur ⇒ *exemple basique*).

Le mot « self » sert à faire référence à l'objet concerné, celui créé par l'instanciation (plusieurs objets différents pouvant être créés à partir de la même classe). « self » est donc en paramètre du constructeur « __init__ » et reçoit la référence de l'objet instancié.

Exemple d'instanciation et d'utilisation d'un objet créé à partir de cette classe :

Interpréteur Python :

```
>>> personne1 = Personne()     # instancie l'objet "personne1" de la classe "Personne"
>>> personne1
<__main__.Personne object at 0x7f6691fdcaf0>
>>> personne1.nom
'Tom'
```

Et aussi :

Interpréteur Python :

```
>>> personne1.nom = "Bill"
>>> personne1.nom
'Bill'
```

Pour faire référence à un attribut d'instance d'un objet (en accès ou en affectation), on écrit :

nom_objet.nom_attribut

⇒ *la notion d'accesseur / mutateur est abordée plus loin*

On peut également **initialiser les attributs d'instance en précisant leur valeur directement au moment de l'instanciation de l'objet :** ⇒ *valeurs transmises en paramètres à « __init__ »*

Code Python :

```
class Personne:
    def __init__(self, nom, age):     # self en premier paramètre, suivi des paramètres transmis
        self.nom = nom     # affecte le paramètre transmis "nom" à l'attribut d'instance "self.nom"
        self.age = age
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
>>> personne2 = Personne("Bill", 40)
>>> personne1.nom
'Tom'
>>> personne1.age
30
>>> personne2.nom
'Bill'
>>> personne2.age
40
```

Les paramètres avec valeur par défaut et les paramètres nommés sont également possibles. De même qu'un nombre indéterminé de paramètres, nommés ou non. ⇒ *voir chapitres précédents*

3.2.2 Attributs de classe

Comme expliqué, **un attribut d'instance et sa valeur sont propres à chaque objet instancié**.

⇒ *référencé par « self » dans la classe (méthodes d'instances) et par le nom de l'objet à l'utilisation*

Un attribut de classe et sa valeur sont propres à la classe, donc communs à tous les objets.

⇒ *référencé par le nom de la classe, dans les méthodes d'instances et à l'utilisation des objets*
⇒ *utile pour partager une valeur entre les objets de la classe (comme compter les objets créés)*

Code Python :

```
class Personne:
    nb_pers = 0     # définition d'attribut de classe, sans préfixe et avant les méthodes
    def __init__(self, nom):
        self.nom = nom
        Personne.nb_pers += 1     # utilisation d'attribut de classe, préfixé par le nom de la classe
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom")
>>> personne2 = Personne("Bill")
>>> personne1.nom     # utilisation d'attribut d'instance, préfixé par le nom de l'objet
'Tom'
>>> personne2.nom
'Bill'
>>> Personne.nb_pers     # utilisation d'attribut de classe, préfixé par le nom de la classe
2
>>> personne1.nb_pers ; personne2.nb_pers     # également possible, mais impropre !
```

3.2.3 Méthodes d'instances (ou méthodes d'objets)

Une méthode d'instance **sert à réaliser un traitement sur un objet et ses attributs d'instance**. Elle se définit comme une fonction **avec pour premier paramètre « self »** (référence à l'objet).

Code Python :

```
class Compte:
    def __init__(self, numero, solde=0):     # solde initialisé à 0 si non fourni
        self.numero = numero
        self.solde = solde
    def crediter_debiter(self, montant):     # définition de méthode d'instance (self en premier)
        self.solde += montant
```

Interpréteur Python :

```
>>> compte1 = Compte("100200300")
>>> print("Le compte n°{} a pour solde {} €".format(compte1.numero, compte1.solde))
Le compte n°100200300 a pour solde 0 €
>>> compte1.crediter_debiter(1500)
>>> print("Le compte n°{} a pour solde {} €".format(compte1.numero, compte1.solde))
Le compte n°100200300 a pour solde 1500 €
```

Pour appeler une méthode d'instance, on écrit : **nom_objet.nom_methode(...)**

Précisions :

Comme expliqué, **le mot « self » sert à faire référence à l'objet concerné :** un attribut d'instance (ou attribut d'objet) et sa valeur sont propres à chaque objet instancié. ⇒ *contenus dans l'objet*

Une méthode d'instance est quant à elle contenue dans la classe, et non dans chaque objet.
Quand un objet appelle une méthode d'instance, c'est la classe qui appelle la méthode avec en paramètre l'objet concerné (grâce à « self ») :

Interpréteur Python :

```
>>> compte1.crediter_debiter
<bound method Compte.crediter_debiter of <__main__.Compte object at 0x7fbffc10db80>>
>>> # Donc possible aussi :
>>> Compte.crediter_debiter(compte1, -2000)
>>> compte1.solde
-500
```

⇒ *avec « def crediter_debiter(self, montant): »*
⇒ *« compte1.crediter_debiter(-2000) » équivalent à « Compte.crediter_debiter(compte1, -2000) »*

3.2.4 Méthodes de classe

Une méthode de classe **ne travaille pas sur une instance mais sur la classe elle-même**.

Elle se définit comme une méthode d'instance avec pour premier paramètre « cls » (contenant la classe de l'objet) à la place de « self » (contenant l'instance de l'objet).

⇒ *utile pour une fonctionnalité indépendante des instances d'objets (donc commune à la classe)*

Code Python :

```
class Personne:
    nb_pers = 0     # définition d'attribut de classe, sans préfixe et avant les méthodes
    def __init__(self, nom):
        self.nom = nom
        Personne.nb_pers += 1     # utilisation d'attribut de classe, préfixé par le nom de la classe
    @classmethod     # décorateur de méthode de classe
    def afficher_nb_pers(cls):     # définition de méthode de classe (cls en premier)
        print("Nombre de personnes créées :", cls.nb_pers)     # attribut de classe préfixé par cls
```

Avant la méthode, **« @classmethod » est un décorateur : transforme la méthode en méthode de classe**. ⇒ *ou après la méthode : « afficher_nb_pers = classmethod(afficher_nb_pers) »*

Interpréteur Python :

```
>>> personne1 = Personne("Tom")
>>> personne2 = Personne("Bill")
>>> Personne.afficher_nb_pers()    # appel méthode de classe, préfixé par le nom de la classe
Nombre de personnes créées : 2
>>> personne1.afficher_nb_pers()    # également possible, mais impropre !
Nombre de personnes créées : 2
```

3.2.5 Méthodes statiques

Une méthode statique **est totalement indépendante des instances et de la classe**, elle n'a ni le paramètre « self » ni le paramètre « cls ». ⇒ *utile pour une fonctionnalité annexe (sans lien direct)*

Code Python :

```
class Personne:
    def __init__(self, nom):
        self.nom = nom
    @staticmethod    # décorateur de méthode statique
    def afficher():    # définition de méthode statique (ni self, ni cls)
        print("Méthode statique, indépendante des instances et de la classe.")
```

Avant la méthode, **« @staticmethod » est un décorateur : transforme la méthode en méthode statique.** ⇒ *ou après la méthode : « afficher = staticmethod(afficher) »*

Interpréteur Python :

```
>>> Personne.afficher()
Méthode statique, indépendante des instances et de la classe.
>>> personne1 = Personne("Tom")
>>> Personne.afficher()
Méthode statique, indépendante des instances et de la classe.
>>> personne1.afficher()
Méthode statique, indépendante des instances et de la classe.
```

3.2.6 Introspection

Pour explorer un objet, connaître ses attributs et méthodes : ⇒ *penser aux docstrings*

La fonction « dir » retourne la liste des attributs et méthodes d'un objet : **dir(personne1)**
Un objet a l'attribut spécial « __dict__ », dictionnaire des attributs / valeurs : **personne1.__dict__**
La fonction « id » retourne la référence de l'objet : **id(personne1)**
Et aussi : **help(personne1) ; type(personne1)** Pour la classe : **help(Personne)**

3.2.7 Encapsulation et propriétés

Encapsulation en POO :

L'encapsulation est un **principe qui consiste à cacher ou protéger l'information contenue dans un objet** et de ne proposer que des méthodes de manipulation de cet objet.

L'utilisateur extérieur ne peut pas modifier directement l'information et risquer de mettre en péril les propriétés comportementales de l'objet. ⇒ *objet vu de l'extérieur comme une "boîte noire"*

Les principes de l'encapsulation sont notamment **appliqués à l'aide de niveaux de visibilité** (ou portée), comme en Java : **public, protégé, privé**.

On utilise alors des méthodes "publiques" d'accès et de modification (accesseur "Get" et mutateur "Set") définies en complément de l'attribut "privé" qui contient réellement la donnée.

Encapsulation en Python : ⇒ *autre philosophie*

En Python, les "getters" et les "setters" ne sont pas les mêmes que ceux des autres langages de programmation orientés objet : **il n'y a pas d'attributs (et méthodes) privés, tout est public**.

Cependant, on peut **définir des méthodes propriétés permettant de programmer les contrôles et les traitements à réaliser sur les attributs en accès, modification, suppression**.

Tout est public, mais on applique la convention : **un attribut, ou une méthode, ne devant être utilisé qu'à l'intérieur d'un objet doit avoir un nom commençant par un « _ »** (underscore).

Définition classique des propriétés :

Code Python :

```
class Personne:
    def __init__(self, nom):
        self._nom = nom     # attribut interne ("privé")
    def _get_nom(self):     # méthode propriété appelée en accès
        print("Retourne le nom.")
        return self._nom
    def _set_nom(self, nom):     # méthode propriété appelée en modification
        self._nom = nom
        print("Nom modifié.")
    def _del_nom(self):     # méthode propriété appelée en suppression
        del self._nom
        print("Nom supprimé.")
    # Définition de l'attribut "nom" comme propriété :
    nom = property(_get_nom, _set_nom, _del_nom, "Nom de personne.")
```

⇒ *property() possède les paramètres nommés fget, fset, fdel et doc*

```
>>> personne1 = Personne("Tom")
>>> personne2 = Personne("Bill")
>>> personne1.nom     # attribut "nom" défini comme propriété
Retourne le nom.
'Tom'
>>> personne1.nom = "Bob"
Nom modifié.
>>> personne1._nom     # également possible, mais impropre !
'Bob'
>>> personne1._get_nom()     # également possible, mais impropre !
Retourne le nom.
'Bob'
>>> del personne1.nom
Nom supprimé.
>>> personne1.nom
Retourne le nom.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in _get_nom
AttributeError: 'Personne' object has no attribute '_nom'
>>> personne2.nom
Retourne le nom.
'Bill'
>>> help(Personne.nom)
```

Définition des propriétés avec décorateurs :

Code Python :

```
class Personne:
    def __init__(self, nom):
        self._nom = nom     # attribut interne ("privé")
    @property     # décorateur de la méthode propriété "nom" appelée en accès
    def nom(self):
        """Nom de personne."""
        print("Retourne le nom.")
        return self._nom
    @nom.setter     # décorateur de la méthode propriété "nom" appelée en modification
    def nom(self, nom):
        self._nom = nom
        print("Nom modifié.")
    @nom.deleter     # décorateur de la méthode propriété "nom" appelée en suppression
    def nom(self):
        del self._nom
        print("Nom supprimé.")
```

⇒ *un décorateur sert à modifier le comportement d'une fonction ou d'une méthode*
⇒ *les méthodes d'une propriété doivent avoir le même nom (mais ce n'est pas de la surcharge)*

Interpréteur Python :

```
>>> personne1 = Personne("Tom")
>>> personne2 = Personne("Bill")
>>> personne1.nom     # attribut "nom" défini comme propriété
Retourne le nom.
'Tom'
>>> personne1.nom = "Bob"
Nom modifié.
>>> personne1._nom     # également possible, mais impropre !
'Bob'
>>> del personne1.nom
Nom supprimé.
>>> personne1.nom
Retourne le nom.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in nom
AttributeError: 'Personne' object has no attribute '_nom'
>>> personne2.nom
Retourne le nom.
'Bill'
>>> help(Personne.nom)
```

3.3 Méthodes spéciales

Tout ce qu'on crée en Python est **objet, créé / hérité à partir de classes internes à Python possédant des méthodes appelées « méthodes spéciales »** (et aussi des attributs spéciaux).

Ces méthodes spéciales **proposent des fonctionnalités particulières sur les objets**, en fonction des types d'objets créés et de leurs besoins, **et sont redéfinissables dans nos classes**.

Elles sont très riches et **possèdent toutes un nom entouré de 2 underscores « __ »**.

3.3.1 Constructeur et destructeur

Rappel (voir avant) :

Les attributs d'instances doivent être définis / initialisés dans le **constructeur de la classe : méthode spéciale nommée « __init__ »**.

Ce constructeur « __init__ » est systématiquement appelé lors de la création / instanciation d'un objet à partir de la classe (initialisant ainsi ses attributs d'instance).

Il y a également un **destructeur de la classe : méthode spéciale nommée « __del__ » appelée au moment de la destruction / suppression d'un objet**.

Un objet est détruit soit explicitement par le mot-clé « del », soit implicitement quand l'espace contenant l'objet est détruit (par exemple dans une fonction) ou encore en fin de programme.

Code Python :

```
class Personne:
    def __init__(self, nom):
        self.nom = nom
        print(self.nom, "a été créé.")
    def __del__(self):
        print(self.nom, "a été supprimé.")
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom")
Tom a été créé.
>>> del personne1
Tom a été supprimé.
```

⇒ *« __del__ » permet donc de réaliser un traitement lors de la suppression d'un objet*

Les méthodes spéciales ont un comportement par défaut si elles ne sont pas redéfinies.

3.3.2 Représentation et affichage

Par défaut, quand on appelle directement un objet dans l'interpréteur, ou qu'on l'affiche avec « print », celui-ci retourne des informations générales sur sa définition :

Interpréteur Python :

```
>>> personne1
<__main__.Personne object at 0x7f2ac995db50>
>>> print(personne1)
<__main__.Personne object at 0x7f2ac995db50>
```

La méthode spéciale « __repr__ » définit l'affichage quand on appelle l'objet par son nom :

Code Python :

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __repr__(self):     # uniquement self en paramètre
        return "Personne: nom={}, age={}.".format(self.nom, self.age)     # retourne une chaîne
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
>>> personne1
Personne: nom=Tom, age=30.
>>> repr(personne1)     # appelle aussi __repr__
'Personne: nom=Tom, age=30.'
```

La méthode spéciale « __str__ » définit l'affichage quand on affiche l'objet avec « print » :

Code Python :

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __str__(self):     # uniquement self en paramètre
        return "{} a {} ans.".format(self.nom, self.age)     # retourne une chaîne
```

⇒ *par défaut, si la méthode « __str__ » n'est pas redéfinie, Python appelle « __repr__ »*

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
>>> print(personne1)
Tom a 30 ans.
>>> str(personne1)     # appelle aussi __str__
'Tom a 30 ans.'
```

⇒ *rappel : « str(…) » convertit en chaîne de caractères*

3.3.3 Attributs

Quand on accède à un attribut avec « nom_objet.nom_attribut » (sauf en affectation), **si Python ne trouve pas l'attribut et si la méthode spéciale « __getattr__ » existe, il va l'appeler en lui transmettant en paramètre le nom de l'attribut** sous forme de chaîne de caractères.

Redéfinir cette méthode spéciale **« __getattr__ » permet donc de réaliser un traitement lors de l'accès à un attribut inexistant**.

Code Python :

```
class Personne:
    def __init__(self, nom):
        self.nom = nom
    def __getattr__(self, nom_attribut):
        print("L'attribut", nom_attribut, "n'existe pas !")
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom")
>>> personne1.nom
'Tom'
>>> personne1.age
L'attribut age n'existe pas !
```

Redéfinir la méthode spéciale « __setattr__ » permet de réaliser un traitement quand un attribut est affecté.

« nom_objet.nom_attribut = valeur » provoque l'appel
« nom_objet.__setattr__("nom_attribut", valeur) »
avec le nom de l'attribut transmis sous forme de chaîne de caractères.

Code Python :

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __setattr__(self, nom_attribut, valeur):
        object.__setattr__(self, nom_attribut, valeur)
        print("L'attribut", nom_attribut, "est affecté avec la valeur", valeur)
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
L'attribut nom est affecté avec la valeur Tom
L'attribut age est affecté avec la valeur 30
>>> personne1.nom = "Bill"
L'attribut nom est affecté avec la valeur Bill
>>> personne1.age = 40
L'attribut age est affecté avec la valeur 40
```

Concernant l'appel « object.__setattr__(self, nom_attribut, valeur) » : ⇒ *voir héritage*

En Python, **toutes les classes créées héritent de la classe de base interne « object » incluant les principales méthodes spéciales** (avec leur comportement par défaut). ⇒ *voir « help(object) »*
Quand on redéfinit « __setattr__ » dans notre classe (fille), il ne faut pas oublier d'appeler « __setattr__ » de la classe « object » (mère) sinon l'attribut ne sera pas affecté !

Redéfinir la méthode spéciale « __delattr__ » permet de réaliser un traitement quand un attribut est supprimé.

Par exemple une classe sans suppression de ses attributs :

Code Python :

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __delattr__(self, nom_attribut):     # prend en paramètre le nom de l'attribut à supprimer
        print("Suppression d'attribut impossible !")
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
>>> del personne1.nom
Suppression d'attribut impossible !
>>> personne1.nom
'Tom'
```

⇒ *ne pas oublier d'appeler « object.__delattr__(self, nom_attribut) » si l'attribut doit être supprimé*

Fonctions complémentaires :

Code Python :

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
>>> getattr(personne1, "nom")
'Tom'
>>> setattr(personne1, "age", 40)
>>> getattr(personne1, "age")
40
>>> delattr(personne1, "age")
>>> hasattr(personne1, "nom")
True
>>> hasattr(personne1, "age")
False
```

⇒ *utiles quand le nom des attributs doit être géré sous forme de chaînes de caractères*

3.3.4 Conteneurs

Pour rappel, les objets conteneurs sont les chaînes de caractères, les listes, les dictionnaires,… Ils contiennent d'autres objets auxquels on accède avec l'opérateur « [ ] ».

Il existe les **méthodes spéciales « __getitem__ », « __setitem__ » et « __delitem__ » appelées quand on accède aux éléments d'un conteneur et qu'on peut redéfinir si besoin**.

Code Python :

```
class MonDictionnaire:
    def __init__(self):
        self._dictionnaire = {}     # attribut interne ("privé")
    def __getitem__(self, index):
        return self._dictionnaire[index]
    def __setitem__(self, index, valeur):
        self._dictionnaire[index] = valeur
    def __delitem__(self, index):
        del self._dictionnaire[index]
```

⇒ *ce type de classe est une "classe enveloppe de dictionnaire"*

Interpréteur Python :

```
>>> dictionnaire1 = MonDictionnaire()
>>> type(dictionnaire1)
<class '__main__.MonDictionnaire'>
>>> dictionnaire1["cle1"] = "valeur1"
>>> dictionnaire1["cle2"] = "valeur2"
>>> dictionnaire1["cle1"]
'valeur1'
>>> dictionnaire1["cle2"]
'valeur2'
>>> dictionnaire1
<__main__.MonDictionnaire object at 0x7f03168e23a0>
>>> print(dictionnaire1)
<__main__.MonDictionnaire object at 0x7f03168e23a0>
>>> del dictionnaire1["cle2"]
>>> dictionnaire1["cle2"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __getitem__
KeyError: 'cle2'
```

⇒ *on pourrait également redéfinir les méthodes spéciales « __repr__ » et « __str__ » (voir avant)*

La méthode spéciale « __contains__ » est appelée par le mot-clé « in » (recherche dans un conteneur).

De même, **la méthode spéciale « __len__ » est appelée par la fonction « len »** (taille d'un conteneur).

Interpréteur Python :

```
>>> liste = [321, "texte", 9.99, "a", "b"]
>>> 9.99 in liste
True
>>> 10 in liste
False
>>> liste.__contains__(9.99)
True
>>> liste.__contains__(10)
False
>>> len(liste)
5
>>> liste.__len__()
5
```

⇒ *ces méthodes spéciales pourraient être aussi redéfinies dans la classe « MonDictionnaire »*

3.3.5 Opérateurs

Les opérateurs sont aussi orientés objet grâce à des **méthodes spéciales incluses aux types**.

⇒ *listes non exhaustives, voir documentation Python et « help(type de données) »*

Méthodes spéciales des opérateurs unaires :

__pos__(self) : +
__neg__(self) : -
__abs__(self) : abs()

Méthodes spéciales des opérateurs arithmétiques (objet à gauche) :

__add__(self, autre_objet) : +
__sub__(self, autre_objet) : -
__mul__(self, autre_objet) : *
__matmul__(self, autre_objet) : @
__truediv__(self, autre_objet) : /
__floordiv__(self, autre_objet) : //
__mod__(self, autre_objet) : %
__pow__(self, autre_objet, modulo) : **

Méthodes spéciales des opérateurs arithmétiques (objet à droite) :

__radd__(self, autre_objet) :	+
__rsub__(self, autre_objet) :	-
__rmul__(self, autre_objet) :	*
__rmatmul__(self, autre_objet) :	@
__rtruediv__(self, autre_objet) :	/
__rfloordiv__(self, autre_objet) :	//
__rmod__(self, autre_objet) :	%
__rpow__(self, autre_objet, modulo) :	**

Méthodes spéciales des opérateurs "in-place" :

__iadd__(self, autre_objet) :	+=
__isub__(self, autre_objet) :	-=
__imul__(self, autre_objet) :	*=
__imatmul__(self, autre_objet) :	@=
__itruediv__(self, autre_objet) :	/=
__ifloordiv__(self, autre_objet) :	//=
__imod__(self, autre_objet) :	%=
__ipow__(self, autre_objet, modulo) :	**=

Méthodes spéciales des opérateurs de comparaison :

__eq__(self, autre_objet) : ==
__ne__(self, autre_objet) : !=
__lt__(self, autre_objet) : <
__le__(self, autre_objet) : <=
__gt__(self, autre_objet) : >
__ge__(self, autre_objet) : >=

Interpréteur Python :

```
>>> nombre = 10
>>> -nombre ; nombre.__neg__()
-10
-10
>>> nombre + 20 ; nombre.__add__(20)
30
30
>>> 20 + nombre ; nombre.__radd__(20)
30
30
>>> nombre == 10 ; nombre.__eq__(10)
True
True
```

Code Python :

```
class Couleur:
    couleurs = ["rouge", "orange", "jaune", "vert", "bleu", "violet"]
    def __init__(self, ma_couleur):
        self.ma_couleur = ma_couleur
    def __eq__(self, autre_couleur):     # la méthode index retourne l'indice d'une valeur de liste
        return Couleur.couleurs.index(self.ma_couleur) == Couleur.couleurs.index(autre_couleur)
    def __ne__(self, autre_couleur):
        return Couleur.couleurs.index(self.ma_couleur) != Couleur.couleurs.index(autre_couleur)
    def __lt__(self, autre_couleur):
        return Couleur.couleurs.index(self.ma_couleur) < Couleur.couleurs.index(autre_couleur)
    def __le__(self, autre_couleur):
        return Couleur.couleurs.index(self.ma_couleur) <= Couleur.couleurs.index(autre_couleur)
    def __gt__(self, autre_couleur):
        return Couleur.couleurs.index(self.ma_couleur) > Couleur.couleurs.index(autre_couleur)
    def __ge__(self, autre_couleur):
        return Couleur.couleurs.index(self.ma_couleur) >= Couleur.couleurs.index(autre_couleur)
```

Interpréteur Python :

```
>>> couleur1 = Couleur("jaune")
>>> couleur1 < "bleu"
True
>>> couleur1 > "bleu"
False
```

3.3.6 Sérialisation

Le module « pickle » utilise les méthodes spéciales « __getstate__ » et « __setstate__ » :

« __getstate__ » est appelée juste avant la sérialisation (écriture) d'un objet dans un fichier.

Si « __getstate__ » n'est pas redéfinie, « pickle » enregistre le dictionnaire des attributs de l'objet « __dict__ ». Sinon, **« pickle » enregistre la valeur renvoyée par « __getstate__ »** (généralement un dictionnaire d'attributs modifié).

« __setstate__ » est appelée juste après la dé-sérialisation (lecture) d'un objet d'un fichier.

Si « __setstate__ » n'est pas redéfinie, « pickle » stocke le dictionnaire des attributs de l'objet lu dans « __dict__ ». Sinon, **« pickle » envoie ce dictionnaire des attributs à « __setstate__ » qui peut le modifier**.

Précision :

« __getstate__ » et « __setstate__ » permettent d'écrire et de lire autre chose que des dictionnaires d'attributs (« __getstate__ » peut renvoyer d'autres objets, par contre « __setstate__ » doit savoir en retour les interpréter).

<u>Exemple d'une classe qui sérialise / dé-sérialise un seul attribut sur deux attributs en doublon</u> :

Code Python :

```
import pickle

class Serialisation:
    def __init__(self, valeur):
        self.attribut1 = valeur     # deux attributs avec la même valeur
        self.attribut2 = valeur

    def __getstate__(self):
        print("Un seul attribut est enregistré.")
        return dict(attribut=self.attribut1)     # renvoie un dictionnaire d'un seul attribut

    def __setstate__(self, dictionnaire_lu):
        print("Un seul attribut est lu et les deux sont régénérés.")
        self.attribut1 = dictionnaire_lu["attribut"]
        self.attribut2 = dictionnaire_lu["attribut"]
```

Interpréteur Python :

```
>>> objet = Serialisation("test")
>>> objet.attribut1
'test'
>>> objet.attribut2
'test'
>>> objet.__dict__
{'attribut1': 'test', 'attribut2': 'test'}
>>> with open("objet-dictionnaire", "wb") as fichier:
...         pickle.Pickler(fichier).dump(objet)     # sérialisation de l'objet
...
Un seul attribut est enregistré.
>>> # vérification : consulter le fichier "objet-dictionnaire"
>>> with open("objet-dictionnaire", "rb") as fichier:
...         objet_lu = pickle.Unpickler(fichier).load()     # dé-sérialisation de l'objet
...
Un seul attribut est lu et les deux sont régénérés.
>>> objet_lu.attribut1
'test'
>>> objet_lu.attribut2
'test'
>>> objet_lu.__dict__
{'attribut1': 'test', 'attribut2': 'test'}
```

3.4 Héritage

3.4.1 Définition

Python reprend les règles traditionnelles de l'héritage :

L'héritage est une **fonctionnalité objet qui permet de déclarer une classe modelée sur une autre classe** appelée classe mère ou classe parente.

Quand une classe B hérite d'une classe A, **les objets créés sur la classe B (fille) ont accès aux attributs et aux méthodes de la classe A (mère)**.

La classe B ne se contente pas de reprendre les attributs et les méthodes de la classe A, **elle peut en définir d'autres : d'autres attributs et d'autres méthodes qui lui sont propres**, en plus des attributs et des méthodes de la classe A.

Elle peut également redéfinir les méthodes de la classe mère. ⇒ *c'est le polymorphisme*

Important : Python cherche d'abord une méthode dans la classe fille et s'il ne la trouve pas dans sa classe mère. Mais si une méthode est redéfinie, il n'appelle pas celle de la classe mère.

3.4.2 Héritage simple

Code Python :

```
class Vehicule:    # classe mère définie de façon classique
    def __init__(self, constructeur, modele):
        self.constructeur = constructeur
        self.modele = modele
    def avancer(self):
        print("Avance.")
    def vendre(self, prix):
        print("A vendre au prix de", prix, "euros.")

class Voiture(Vehicule):    # Voiture hérite de Vehicule
    def __init__(self, constructeur, modele, nb_portes):
        super().__init__(constructeur, modele)    # obligatoire car "__init__" est redéfinie
        self.nb_portes = nb_portes    # attribut spécifique à Voiture
    def avancer(self):    # redéfinition de la méthode "avancer" (polymorphisme)
        print("Roule.")
    def klaxonner(self):    # méthode spécifique à Voiture
        print("Klaxonne.")
```

La définition d'une classe fille héritant d'une classe mère s'écrit : **class ClasseFille(ClasseMere):**

La classe mère regroupe les caractéristiques communes et la classe fille les spécificités.

« super() » référence la classe mère. On aurait pu appeler « Vehicule.__init__(self, constructeur, modele) » mais « super() » est davantage généraliste et évolutif.

⇒ *« __init__ » est redéfinie dans la classe fille et les attributs « constructeur » et « modele » sont définis dans la classe mère, d'où la nécessité d'appeler « super().__init__(constructeur, modele) »*

Interpréteur Python :

```
>>> voiture1 = Voiture("Peugeot", "308", 5)
>>> print(voiture1.constructeur, voiture1.modele, voiture1.nb_portes, "portes.")
Peugeot 308 5 portes.
>>> voiture1.avancer()
Roule.
>>> voiture1.klaxonner()
Klaxonne.
>>> voiture1.vendre(25500)
A vendre au prix de 25500 euros.
```

⇒ *on pourrait également définir des classes « Avion » et « Bateau » héritant de « Vehicule »*

3.4.3 Fonctions utiles

La fonction « issubclass » vérifie si une classe est la sous-classe d'une autre classe :

```
>>> issubclass(Voiture, Vehicule)
True
>>> issubclass(Vehicule, Voiture)
False
>>> issubclass(Vehicule, object)     # les classes créées héritent de la classe interne object
True
>>> issubclass(Voiture, object)     # Voiture hérite de Vehicule qui hérite d'object
True
```

Et aussi : **« help(Vehicule) » / « help(Voiture) »**

La fonction « isinstance » vérifie si un objet est l'instance d'une classe ou classe mère :

```
>>> isinstance(voiture1, Voiture)
True
>>> isinstance(voiture1, Vehicule)
True
```

3.4.4 Héritage multiple

En Python, **une classe (fille) peut également hériter de plusieurs classes (mères)** avec une définition sous la forme :

class ClasseFille(ClasseMere1, ClasseMere2,…):

⇒ *différent d'une classe qui hérite d'une classe qui hérite elle-même d'une autre classe*
⇒ *utile pour les classes qui doivent hériter des fonctionnalités de plusieurs classes différentes*

Recherche des méthodes :

La recherche des méthodes se fait dans l'ordre de la définition de la classe.

D'abord dans la classe (fille) en question, puis **dans les classes mères de gauche à droite**.

Si Python ne trouve pas la méthode dans une classe mère donnée, il **remonte successivement dans ses propres classes mères**, et ainsi de suite.

⇒ *les héritages multiples complexes peuvent être difficiles à comprendre et à maintenir...*

3.4.5 Exceptions personnalisées

Les exceptions sont **aussi des classes, hiérarchisées d'après un héritage précis**.

Attention : le type d'exception précisé après « except » est intercepté ainsi que toutes les classes héritées de ce type.

Pour connaître l'héritage d'une exception, utiliser « help(type d'exception) » (ou voir la documentation Python).

Exemple d'un héritage d'exception : ⇒ *voir « help(NameError) »*

```
Help on class NameError in module builtins:

class NameError(Exception)
 |  Name not found globally.
 |
 |  Method resolution order:
 |      NameError
 |      Exception
 |      BaseException
 |      object
```

Les modules importés peuvent apporter d'autres exceptions et **on peut également créer nos propres exceptions (gérables ensuite avec « raise » et « except ») :**

Nos exceptions doivent hériter d'une exception de Python, une spécifique ou au pire de :

- Exception : classe mère de toutes les exceptions d'erreurs.
- BaseException : classe mère de toutes les exceptions, d'erreurs ou non.

Exemple d'une exception personnalisée :

Code Python :

```
class MonErreur(Exception):     # MonErreur hérite d'Exception
    def __init__(self, message):     # pour la construction de l'exception et son message
        self.message = message     # peut prendre d'autres paramètres à l'instanciation (si besoin)
    def __str__(self):     # pour l'affichage du message d'exception (mise en forme libre)
        return self.message
```

Interpréteur Python :

```
>>> raise MonErreur("Exemple d'exception d'erreur.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonErreur: Exemple d'exception d'erreur.
```

3.5 Itérateurs

3.5.1 Utilisation

Quand on parcourt un conteneur avec une boucle « for, in » c'est un mécanisme d'itérateur qui est appelé (en interne, la boucle s'arrête quand l'exception « StopIteration » est levée).

Un itérateur est un **objet chargé de parcourir un objet conteneur** (chaîne, liste, dictionnaire,...).

Il est **créé dans la méthode spéciale « __iter__ » de l'objet conteneur**.

À chaque itération, **la méthode spéciale « __next__ » de l'itérateur renvoie l'élément suivant** (ou lève l'exception « StopIteration » à la fin du parcours).

Enfin, **la fonction « iter » permet d'appeler la méthode spéciale « __iter__ » de l'objet conteneur** passé en paramètre.

Et **la fonction « next » permet d'appeler la méthode spéciale « __next__ » de l'itérateur** passé en paramètre.

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> iterateur_liste = iter(liste)
>>> iterateur_liste
<list_iterator object at 0x7f561c8e7d30>
>>> next(iterateur_liste)
'chaîne'
>>> next(iterateur_liste)
123
>>> next(iterateur_liste)
'aaa'
>>> next(iterateur_liste)
9.99
>>> next(iterateur_liste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

⇒ *voir aussi « help(liste) » et « help(iterateur_liste) »*

3.5.2 Création

On peut **créer nos propres itérateurs en redéfinissant les méthodes spéciales « __iter__ » et « __next__ »** dans les classes concernées.

Exemple d'itérateur parcourant une chaîne de caractères de la fin au début :

Code Python :

```
class InverseStr(str):     # InverseStr hérite de str
    def __iter__(self):     # seule méthode à redéfinir
        return IterateurInverseStr(self)     # crée et renvoie l'itérateur

class IterateurInverseStr:
    def __init__(self, chaine):
        self.chaine = chaine
        self.position = len(chaine)     # part de la fin
    def __next__(self):
        if self.position == 0:     # si arrive au début
            raise StopIteration     # fin du parcours
        self.position -= 1     # décrémente la position
        return self.chaine[self.position]     # renvoie l'élément suivant
```

La classe « InverseStr » - héritant de la classe « str » - est le conteneur avec redéfinition de la méthode spéciale « __iter__ » qui crée et renvoie l'itérateur permettant de parcourir la chaîne dans le sens inverse.

La classe « IterateurInverseStr » est l'itérateur avec redéfinition de la méthode spéciale « __next__ » qui renvoie l'élément suivant et lève l'exception « StopIteration » à la fin.

Interpréteur Python :

```
>>> chaine = InverseStr("Exemple")
>>> print(chaine)
Exemple
>>> for caractere in chaine:
...       print(caractere)
...
e
l
p
m
e
x
E
```

3.6 Générateurs

3.6.1 Principe

Les générateurs **simplifient la gestion des itérateurs grâce au mot-clé « yield »** (traduisible par "génère").

« yield » **ne peut s'utiliser que dans une fonction ou une méthode** (le générateur qui définit l'itération) et **sert à renvoyer la valeur de l'itération** en cours.

Fonctionnement :

Au premier « next », **la fonction s'exécute jusqu'au premier « yield » qui renvoie sa valeur et se met en pause**.

Au « next » suivant, la fonction reprend après le dernier « yield » jusqu'au « yield » suivant qui renvoie sa valeur et se met en pause, etc.

Cela jusqu'à la fin de la fonction où l'exception « StopIteration » est automatiquement levée.

```
>>> def generateur123():
...     yield 1
...     yield 2
...     yield 3
...
>>> iterateur123 = iter(generateur123())    # il faut appeler (exécuter) la fonction avec ()
>>> iterateur123
<generator object generateur123 at 0x7f02c5105120>
>>> next(iterateur123)
1
>>> next(iterateur123)
2
>>> next(iterateur123)
3
>>> next(iterateur123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>> for nombre in generateur123():    # il faut appeler (exécuter) la fonction avec ()
...     print(nombre)
1
2
3
```

Quand on exécute la fonction, celle-ci produit un générateur : objet créé par Python (de la classe « generator ») qui définit ses propres méthodes spéciales « __iter__ » et « __next__ » et donc son propre itérateur.

⇒ *voir « help(generateur123()) »*

Autre exemple :

Code Python :

```
def intervalle(min, max):
    while min <= max:
        yield min
        min += 1
```

Interpréteur Python :

```
>>> for nombre in intervalle(7, 9):
...         print(nombre)
...
7
8
9
```

« yield » peut aussi s'utiliser directement dans la méthode spéciale « __iter__ » d'une classe conteneur, produisant automatiquement les générateur et itérateur nécessaires :

Code Python :

```
class InverseStr(str):
    def __iter__(self):
        position = len(self)
        while position > 0:
            position -= 1
            yield self[position]
```

Interpréteur Python :

```
>>> chaine = InverseStr("Exemple")
>>> for caractere in chaine:
...         print(caractere)
...
e
l
p
m
e
x
E
```

3.6.2 Coroutines

Les générateurs possèdent également des méthodes permettant d'interagir avec eux pendant le parcours (fonctionnalités de « yield »).

⇒ *le traitement d'un générateur avec lequel on peut interagir est une coroutine*

Interrompre le générateur (exemple avec la précédente fonction « intervalle ») : ⇒ *« close »*

Interpréteur Python :

```
>>> generateur_intervalle = intervalle(5, 50)     # production du générateur
>>> for nombre in generateur_intervalle:     # parcours du générateur
...         if nombre <= 8:
...             print(nombre)
...         else:
...             generateur_intervalle.close()
...
5
6
7
8
```

<u>Envoyer des données au générateur</u> : ⇒ *« send »*

Le mot-clé « yield » renvoie la valeur de l'itération en cours **puis se met en pause. À ce moment-là, on peut envoyer une valeur au générateur**, d'où la syntaxe générale de « yield » :

valeur_recue = yield valeur_renvoyee

```
def intervalle(min, max):
    while min <= max:
        valeur_recue = yield min
        if valeur_recue is not None:
            min = valeur_recue
        else:
            min += 1
```

```
>>> generateur_intervalle = intervalle(5, 50)
>>> for nombre in generateur_intervalle:
...         print(nombre)
...         if nombre == 6:
...             generateur_intervalle.send(49)
5
6
49
50
```

3.7 Décorateurs

3.7.1 Définition

Un décorateur est une **fonction modifiant le comportement par défaut d'une autre fonction** (ou d'une méthode, ou même d'une classe).

⇒ *c'est de la métaprogrammation : un programme transformant un autre programme*

Concrètement, **la fonction d'un décorateur prend en paramètre une fonction (celle à modifier) et renvoie une fonction (la version modifiée)**.

Pour préciser qu'une fonction (ou une méthode) doit être modifiée par un décorateur, **il faut ajouter l'indication suivante avant la ligne de définition de la fonction :**

@nom_fonction_decorateur ⇒ *avec le nom de la fonction du décorateur*

Important :

Le décorateur s'exécute au moment de la définition de la fonction et non lors de son appel.

Exemple basique (sans modification de la fonction transmise au décorateur) :

```
>> def mon_decorateur(fonction_transmise):     # fonction décorateur avec fonction transmise
...       print("Décorateur appelé avec en paramètre la fonction :", fonction_transmise)
...       return fonction_transmise     # renvoie la fonction transmise (sans l'avoir modifiée)
...
>>> @mon_decorateur     # précise que ma_fonction doit être modifiée par mon_decorateur
...    def ma_fonction():
...         print("Appel de ma_fonction.")
...
Décorateur appelé avec en paramètre la fonction : <function ma_fonction at 0x7fd161b12430>
>>>
>>> ma_fonction()     # mon_decorateur n'est pas rappelée lors de l'appel de ma_fonction
Appel de ma_fonction.
```

⇒ *on voit que « mon_decorateur » est bien appelée juste après la définition de « ma_fonction »*

Produisent le même résultat :

@mon_decorateur
def ma_fonction():

def ma_fonction():
...
ma_fonction = mon_decorateur(ma_fonction) ⇒ *après la définition*

3.7.2 Modification de la fonction

Dans la fonction du décorateur, **on doit définir et renvoyer la fonction qui modifie le comportement de la fonction transmise au décorateur**. La fonction renvoyée remplace alors la fonction d'origine :

```
def mon_decorateur(fonction_transmise):     # fonction décorateur avec fonction transmise
    def fonction_modifiee():     # fonction modifiée qui sera renvoyée (fonction locale)
        print(fonction_transmise, "est modifiée !")
        return fonction_transmise()     # exécute et renvoie le résultat de la fonction d'origine
    return fonction_modifiee     # renvoie la fonction modifiée (juste sa référence)

@mon_decorateur
def ma_fonction():
    print("Appel de ma_fonction.")
```

```
>>> ma_fonction     # ma_fonction fait référence à fonction_modifiee (qui la remplace)
<function mon_decorateur.<locals>.fonction_modifiee at 0x7f1e0262e4c0>
>>>
>>> ma_fonction()     # fonction_modifiee est appelée avec son complément de ma_fonction
<function ma_fonction at 0x7f1e0262e430> est modifiée !
Appel de ma_fonction.
```

3.7.3 Décorateur avec paramètres

Le principe est basé sur le fonctionnement suivant :

Code Python :

```
def fonction1():
    def fonction2():    # fonction2 est locale à fonction1
        print("fonction2")
    return fonction2    # fonction1 renvoie la référence de fonction2
```

Interpréteur Python :

```
>>> fonction2()    # fonction2 est locale à fonction1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fonction2' is not defined
>>> fonction1()    # fonction1 renvoie la référence de fonction2
<function fonction1.<locals>.fonction2 at 0x7f9aab721430>
>>> fonction1()()    # appelle donc fonction2 !
fonction2
```

```
@mon_decorateur(parametre)
def ma_fonction():
```

est donc équivalent à :

```
def ma_fonction():
...
ma_fonction = mon_decorateur(parametre)(ma_fonction)
```

⇒ *sans paramètre(s), la fonction du décorateur prend en paramètre la fonction à modifier*

⇒ *avec paramètre(s), on doit définir comme décorateur une fonction qui prend le(s) paramètre(s) du décorateur et qui renvoie un décorateur à qui on transmet la fonction à modifier*

⇒ *on se retrouve alors avec un niveau supplémentaire dans notre fonction : la fonction du décorateur habituel est imbriquée dans la fonction recevant le(s) paramètre(s) du décorateur*

<u>Exemple de décorateur avec paramètre</u> :

Décorateur qui répète un certain nombre de fois la fonction "décorée" (page suivante).

Code Python :

```
def repeter(nombre):
    def decorateur_repeter(fonction_transmise):
        def fonction_modifiee():
            i = 1
            while i <= nombre:
                fonction_transmise()     # ici on ne gère pas une fonction retournant un résultat
                i += 1
        return fonction_modifiee
    return decorateur_repeter

@repeter(3)
def exemple():
    print("Exemple")
```

Interpréteur Python :

```
>>> exemple
<function repeter.<locals>.decorateur_repeter.<locals>.fonction_modifiee at 0x7f747993d550>
>>>
>>> exemple()
Exemple
Exemple
Exemple
```

3.7.4 Fonction avec paramètres

Pour qu'un décorateur puisse s'appliquer à des fonctions quels que soient leurs paramètres (aucuns, non nommés ou nommés), **il faut adapter la fonction modifiée ainsi :**

def fonction_modifiee(*parametres_non_nommes, **parametres_nommes):

et aussi l'appel de la fonction d'origine :

fonction_transmise(*parametres_non_nommes, **parametres_nommes)

Code Python :

```
def repeter(nombre):
    def decorateur_repeter(fonction_transmise):
        def fonction_modifiee(*parametres_non_nommes, **parametres_nommes):
            print("Paramètres non nommés transmis :", parametres_non_nommes)
            print("Paramètres nommés transmis :", parametres_nommes)
            i = 1
            while i <= nombre:
                fonction_transmise(*parametres_non_nommes, **parametres_nommes)
                i += 1
        return fonction_modifiee
    return decorateur_repeter
```

Interpréteur Python :

```
>>> @repeter(3)
...    def personne(nom, age):
...          print(nom, "a", age, "ans.")
...
>>> personne
<function repeter.<locals>.decorateur_repeter.<locals>.fonction_modifiee at 0x7fc48e96d550>
>>>
>>> personne("Tom", 30)
Paramètres non nommés transmis : ('Tom', 30)
Paramètres nommés transmis : {}
Tom a 30 ans.
Tom a 30 ans.
Tom a 30 ans.
>>>
>>> personne(age=40, nom="Bill")
Paramètres non nommés transmis : ()
Paramètres nommés transmis : {'age': 40, 'nom': 'Bill'}
Bill a 40 ans.
Bill a 40 ans.
Bill a 40 ans.
```

3.7.5 Appliquer plusieurs décorateurs

On peut appliquer plusieurs décorateurs à une définition de fonction :

```
@decorateur1
@decorateur2
def fonction():
```

est donc équivalent à :

```
def fonction():
...
fonction = decorateur1(decorateur2(fonction))
```

3.7.6 Appliquer un décorateur à une classe

On peut aussi appliquer un décorateur à une définition de classe : **au lieu de recevoir en paramètre la fonction, on reçoit la classe**.

<u>Exemple basique (sans modification de la classe transmise au décorateur)</u> :

Interpréteur Python :

```
>>> def mon_decorateur(classe_transmise):     # fonction décorateur avec classe transmise
...         print("Décorateur appelé avec en paramètre la classe :", classe_transmise)
...         return classe_transmise     # renvoie la classe transmise (sans l'avoir modifiée)
...
>>> @mon_decorateur     # précise que ma_classe doit être modifiée par mon_decorateur
...    class ma_classe:
...         pass     # sans caractéristiques
...
Décorateur appelé avec en paramètre la classe : <class '__main__.ma_classe'>
```

⇒ *on voit que « mon_decorateur » est bien appelée juste après la définition de « ma_classe »*

<u>Exemple concret avec une classe "singleton"</u> :

Une classe "singleton" est une classe qui ne peut être instanciée qu'une seule fois (on ne peut créer qu'un seul objet de cette classe).

Code Python :

```
def singleton(classe_transmise):
    instances = {}     # dictionnaire des instances créées
    def classe_modifiee():
        # Si la classe transmise n'a pas encore d'instance
        if classe_transmise not in instances:
            # Crée l'objet de la classe transmise
            # et ajoute son instance dans le dictionnaire (avec la clé classe_transmise)
            instances[classe_transmise] = classe_transmise()
            # Renvoie l'instance créée (de la classe transmise)
            return instances[classe_transmise]
        else:
            # Sinon erreur
            raise Exception("objet de la {} déjà créé".format(classe_transmise))
    return classe_modifiee
```

```
>>> @singleton
...   class Classe1:
...         pass
...
>>> @singleton
...   class Classe2:
...         pass
...
>>> Classe1
<function singleton.<locals>.classe_modifiee at 0x7facd97a8430>
>>> Classe2
<function singleton.<locals>.classe_modifiee at 0x7facd97a84c0>
>>> o1_classe1 = Classe1()
>>> o2_classe1 = Classe1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in classe_modifiee
Exception: objet de la <class '__main__.Classe1'> déjà créé
>>> o1_classe2 = Classe2()
>>> o2_classe2 = Classe2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in classe_modifiee
Exception: objet de la <class '__main__.Classe2'> déjà créé
```

3.8 Métaclasses

Les métaclasses sont aussi un concept de métaprogrammation : **des classes générées à partir d'autres classes**.

3.8.1 Processus d'instanciation

La méthode spéciale « __init__ » initialise l'objet (en affectant ses attributs), mais elle ne le crée pas.

Auparavant, **c'est la méthode spéciale « __new__ » qui se charge de créer l'objet** (attribuant « self »), méthode définie par la classe mère « object » qu'on peut redéfinir si besoin.

<u>Processus d'instanciation complet</u> :

1) On demande à créer un objet, en écrivant par exemple « Personne("Tom", 30) ».

2) La méthode « __new__ » de la classe (Personne) est appelée et se charge de construire un nouvel objet.

3) Si « __new__ » renvoie une instance de la classe, le constructeur « __init__ » est appelé avec en paramètres cette nouvelle instance (self) et les arguments passés lors de la création de l'objet.

La méthode spéciale « __new__ » est une méthode de classe, elle ne prend pas « self » en premier paramètre (son but étant de créer une nouvelle instance qui n'existe pas encore).

Elle prend la classe manipulée « cls » : quand on crée un objet de la classe « Personne », la méthode « __new__ » est appelée avec en premier paramètre la classe « Personne » elle-même.

Les autres paramètres passés à la méthode « __new__ » (valeurs initiales des attributs) sont ensuite transmis au constructeur « __init__ ».

Code Python :

```
class Personne:
    def __new__(cls, nom, age):     # redéfinition de la méthode __new__
        print("Appel de la méthode __new__ de la classe :", cls)
        return object.__new__(cls)     # pour que l'instance soit bien créée et renvoyée
        # ou : return super().__new__(cls)
    def __init__(self, nom, age):
        print("Appel de la méthode __init__ pour l'objet :", self)
        self.nom = nom
        self.age = age
```

Interpréteur Python :

```
>>> personne1 = Personne("Tom", 30)
Appel de la méthode __new__ de la classe : <class '__main__.Personne'>
Appel de la méthode __init__ pour l'objet : <__main__.Personne object at 0x7f956f1b83a0>
>>> personne1.nom
'Tom'
>>> personne1.age
30
```

⇒ redéfinir « __new__ » peut permettre de créer une instance d'une autre classe (Python l'utilise pour produire les types « immuables / immutables »), ou permettre la définition d'une métaclasse

3.8.2 Créer une classe dynamiquement

Tout est objet en Python, même les classes ! Elles sont donc aussi modelées sur une classe.

```
>>> type(123)
<class 'int'>
>>> type(int)
<class 'type'>
```

Par défaut, toutes nos classes sont modelées sur la classe « type » :

Quand on crée une nouvelle classe, Python appelle la méthode « __new__ » de la classe « type ».

Une fois la classe créée (en tant qu'objet), il appelle la méthode « __init__ » de la classe « type ».

⇒ *la classe « type » hérite de la classe « object »*
⇒ *voir aussi « type(Personne) » et « help(type) »*

Pour créer une classe dynamiquement sans le mot-clé « class » mais avec la classe « type » :

La classe « type » a besoin de trois paramètres pour sa construction :

- Le nom de la classe à créer

- Un tuple contenant les classes dont la nouvelle classe hérite

- Un dictionnaire contenant les attributs et méthodes de la classe ⇒ *ici attributs de classe*

Code Python :

```
# Fonctions des méthodes :

def initialiser_personne(self, nom, age):     # self prendra l'objet manipulé
    self.nom = nom
    self.age = age

def afficher_personne(self):     # self prendra l'objet manipulé
    print(self.nom, "a", self.age, "ans.")

# Dictionnaire des méthodes associées aux fonctions (ici pas d'attributs) :

methodes = {
    "__init__": initialiser_personne,
    "afficher": afficher_personne
}

# Création dynamique de la classe Personne (par une instanciation de la classe type) :

Personne = type("Personne", (), methodes)
```

Interpréteur Python :

```
>>> Personne
<class '__main__.Personne'>
>>>
>>> personne1 = Personne("Tom", 30)
>>> personne1
<__main__.Personne object at 0x7f0c81cefac0>
>>> personne2 = Personne("Bill", 40)
>>> personne2
<__main__.Personne object at 0x7f0c81c41640>
>>> personne1.nom
'Tom'
>>> personne1.age
30
>>> personne1.afficher()
Tom a 30 ans.
>>> personne2.nom
'Bill'
>>> personne2.age
40
>>> personne2.afficher()
Bill a 40 ans.
```

⇒ *cette alternative est là pour mieux comprendre le principe de métaclasse (mais ne pas l'utiliser)*

3.8.3 Définition d'une métaclasse

Une métaclasse est une **classe dont les instances sont des classes**. ⇒ *classe de classes*

« type » est la métaclasse par défaut de toutes les classes créées (voir avant).

Mais **une classe peut posséder une autre métaclasse que « type »**.

Construire une métaclasse se fait de la même façon qu'une classe, **elle doit cependant hériter de « type »**.

La méthode spéciale « __new__ » d'une classe métaclasse prend donc en paramètres :

- La métaclasse servant de modèle à la création de la nouvelle classe
- Le nom de la classe à créer
- Le tuple contenant les classes dont la nouvelle classe hérite
- Le dictionnaire contenant les attributs et méthodes de la classe

La méthode spéciale « __init__ » d'une classe métaclasse prend les mêmes paramètres que « __new__ », sauf le premier qui n'est plus la métaclasse servant de modèle mais la classe que l'on vient de créer.

Pour indiquer qu'une classe prend comme métaclasse autre chose que « type », la définition de la classe doit s'écrire : **class MaClasse(metaclass=MaMetaClasse):**

Exemple concret avec une métaclasse interdisant les héritages multiples :

Code Python :

```
# Définition de la métaclasse (héritant de type) :

class HeritageSimple(type):
    def __new__(metaclasse, nom_classe, classes_meres, attributs_methodes):
        # Si hérite de plusieurs classes mères alors erreur :
        if len(classes_meres) > 1:
            raise Exception("héritage multiple interdit")
        # Sinon exécute et renvoie __new__ de la classe mère (type) :
        return super().__new__(metaclasse, nom_classe, classes_meres, attributs_methodes)

# Définition d'une classe de base (mère) d'après la métaclasse HeritageSimple :

class ClasseBase(metaclass=HeritageSimple):
    pass
```

Interpréteur Python :

```
>>> class Classe1(ClasseBase):
...         pass
...
>>> class Classe2(ClasseBase):
...         pass
...
>>> class Classe3(ClasseBase, ClasseBase):
...         pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __new__
Exception: héritage multiple interdit
>>>
>>> class Classe4(Classe1, Classe2):
...         pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __new__
Exception: héritage multiple interdit
```

⇒ *la définition se répercute également dans toutes les classes filles*

4 Introduction aux interfaces graphiques Tkinter

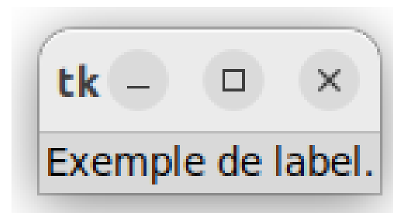
4.1 Module tkinter

Le module « tkinter » est le **module de base permettant de créer des interfaces graphiques**.

Importer le module « tkinter » : **import tkinter**
Aide et fonctionnalités : **help(tkinter)** ⇒ *liste : dir(tkinter)*

⇒ *il est très riche (notamment en propriétés graphiques) et commun aux différentes plateformes*

4.2 Exemple basique



Interpréteur Python :

```
>>> from tkinter import *
>>> fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
>>> label = Label(fenetre, text="Exemple de label.")     # création d'un objet label pour la fenêtre
>>> label.pack()     # place le label (l'affiche)
>>> fenetre.mainloop()     # boucle de la fenêtre jusqu'à fermeture (interactions avec l'utilisateur)
```

Les objets graphiques sont appelés **"widgets". Après leur création, on peut aussi afficher et modifier la valeur de leurs options** (gérées sous forme de dictionnaire propre à l'objet) :

```
>>> label["text"]
'Exemple de label.'
>>> label["text"] = "Autre label."     # sous la forme : widget["option"] = valeur
>>> label["text"]
'Autre label.'
>>> label.keys()
['activebackground', 'activeforeground', 'anchor', 'background', 'bd', 'bg', 'bitmap', 'borderwidth',
'compound', 'cursor', 'disabledforeground', 'fg', 'font', 'foreground', 'height', 'highlightbackground',
'highlightcolor', 'highlightthickness', 'image', 'justify', 'padx', 'pady', 'relief', 'state', 'takefocus',
'text', 'textvariable', 'underline', 'width', 'wraplength']
```

⇒ *à ne pas confondre avec les attributs et les méthodes de l'objet (voir « dir(label) »)*

La fenêtre est également un widget avec options : ⇒ *voir aussi « dir(fenetre) »*

```
>>> fenetre.keys()
['bd', 'borderwidth', 'class', 'menu', 'relief', 'screen', 'use', 'background', 'bg', 'colormap',
'container', 'cursor', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'padx',
'pady', 'takefocus', 'visual', 'width']
>>> fenetre["borderwidth"] = 50
```

4.3 Principaux widgets

4.3.1 Fenêtres et labels (complément)

Les fenêtres disposent des méthodes « title » et « geometry » pour leur titre et taille.

Les labels servent à afficher du texte (non modifiable par l'utilisateur), utiles pour donner une signification à un autre widget (comme une zone de saisie).

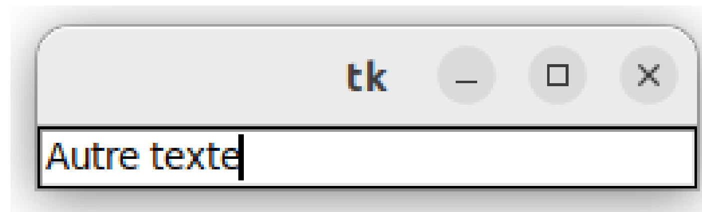
4.3.2 Boutons

Les boutons **permettent de proposer une action à l'utilisateur (appel d'une fonction ou méthode)**, comme quitter le traitement (« mainloop ») de la fenêtre (mais ne la ferme pas) :

```
>>> bouton_quitter = Button(fenetre, text="Quitter", command=fenetre.quit)     # définit le bouton
>>> bouton_quitter.pack()     # place le bouton (l'affiche)
>>> fenetre.mainloop()     # boucle de la fenêtre jusqu'à bouton "Quitter" ou fermeture
```

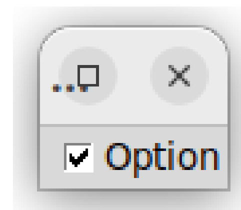
⇒ *ferme la fenêtre : command=fenetre.destroy*

4.3.3 Zones de saisie




```
>>> texte = StringVar()     # définit l'objet texte lié à la zone de saisie
>>> texte.set("Texte par défaut")     # lui affecte un texte
>>> zone_saisie = Entry(fenetre, textvariable=texte, width=30)     # définit la zone de saisie
>>> zone_saisie.pack()     # place la zone de saisie (l'affiche)
>>> texte.get()     # récupère la saisie en cours (ENTREE inutile)
'Autre texte'
```

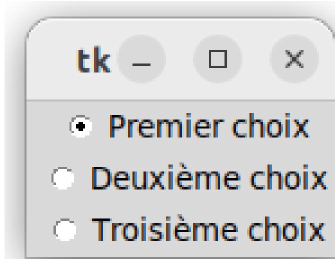
4.3.4 Cases à cocher




```
>>> case = BooleanVar()     # définit l'objet booléen lié à la case à cocher
>>> case_cocher = Checkbutton(fenetre, text="Option", variable=case)     # définit la case
>>> case_cocher.pack()     # place la case (l'affiche)
>>> case.get()     # récupère la sélection en cours (True ou False)
```

4.3.5 Boutons radio

Les boutons radio sont des sortes de **cases à cocher d'un groupe dans lequel seul un élément peut être sélectionné** à la fois. Pour créer un groupe de boutons radio, **il faut simplement qu'ils soient tous liés au même objet :**

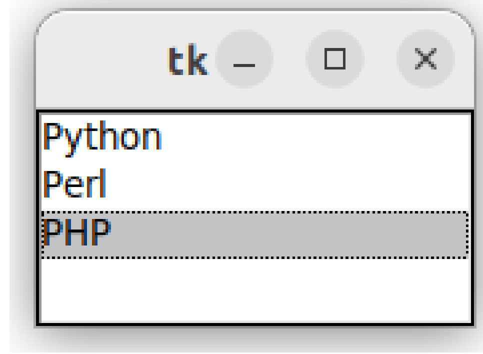


Interpréteur Python :

```
>>> choix_bouton = StringVar()     # définit l'objet texte lié aux boutons radio
>>> choix_bouton.set("choix1")     # lui affecte la valeur par défaut
>>> bouton1 = Radiobutton(fenetre, text="Premier choix", variable=choix_bouton,
value="choix1")     # définit le bouton1
>>> bouton2 = Radiobutton(fenetre, text="Deuxième choix", variable=choix_bouton,
value="choix2")     # définit le bouton2
>>> bouton3 = Radiobutton(fenetre, text="Troisième choix", variable=choix_bouton,
value="choix3")     # définit le bouton3
>>> bouton1.pack()     # place le bouton1 (l'affiche)
>>> bouton2.pack()     # place le bouton2 (l'affiche)
>>> bouton3.pack()     # place le bouton3 (l'affiche)
>>> choix_bouton.get()     # récupère la valeur de la sélection en cours
```

4.3.6 Listes déroulantes

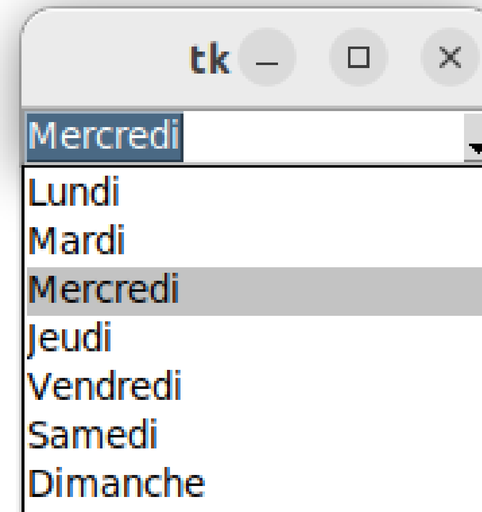
Les listes déroulantes **permettent de construire une liste dans laquelle on peut sélectionner un ou plusieurs éléments :**



Interpréteur Python :

```
>>> liste = Listbox(fenetre)     # définit la liste déroulante (et aussi : selectmode=MULTIPLE)
>>> liste.pack()     # place la liste déroulante (l'affiche)
>>> liste.insert(0, "Python")     # y insère le 1er élément (indice commence à 0, END pour fin)
>>> liste.insert(1, "Java")     # y insère le 2ème élément
>>> liste.insert(2, "PHP")     # y insère le 3ème élément
>>> liste.delete(1)     # y supprime le 2ème élément
>>> liste.insert(1, "Perl")     # y insère un autre 2ème élément (décale la suite)
>>> liste.curselection()     # récupère la sélection en cours (tuple avec l'indice)
(2,)
>>> liste.get(liste.curselection()[0])     # récupère la valeur de la sélection en cours
'PHP'
```

4.3.7 Combo boxes



Les combo boxes **réunissent une zone de saisie et une liste déroulante :**

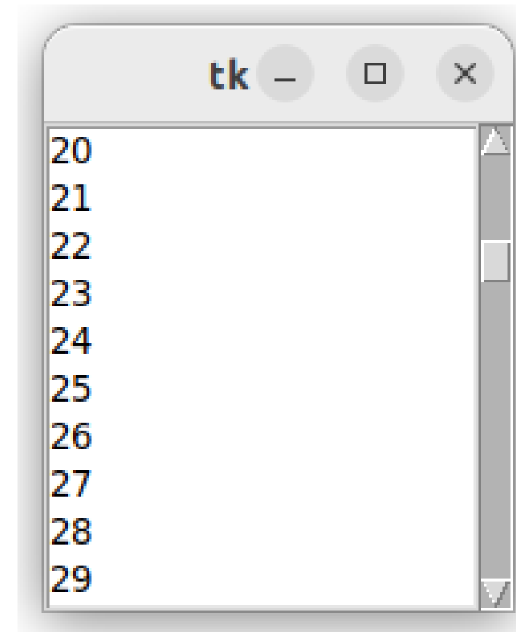
⇒ *nécessitent le sous-module « Ttk » des widgets à thèmes*

⇒ *attention : supplante (et altère) les widgets classiques*

Interpréteur Python :

```
>>> from tkinter.ttk import *
>>>
>>> jours = ("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche")
>>> jour = StringVar()     # définit l'objet texte lié à la combo box
>>> jour.set("Lundi")     # lui affecte la valeur par défaut
>>> combo_jours = Combobox(fenetre, textvariable=jour, values=jours)     # définit la combo
>>> combo_jours.pack()     # place la combo (l'affiche)
>>> jour.get()     # récupère la valeur de la sélection en cours
'Mercredi'
>>> jour.get()     # récupère aussi la saisie en cours (Autre texte)
'Autre texte'
```

4.3.8 Barres de défilement



Les barres de défilement **permettent de faire défiler les widgets listes :**

Interpréteur Python :

```
>>> liste = Listbox(fenetre)     # définit la liste déroulante
>>> liste.pack(side=LEFT)     # place la liste déroulante (l'affiche à gauche)
>>> barre = Scrollbar(fenetre)     # définit la barre de défilement
>>> barre.pack(side=RIGHT, fill=BOTH)     # place la barre de défilement (l'affiche à droite)
>>> for valeurs in range(100):     # remplit la liste déroulante de 100 valeurs
...        liste.insert(END, valeurs)
>>> liste.config(yscrollcommand=barre.set)     # rattache verticalement la barre à la liste
>>> barre.config(command=liste.yview)     # la barre fait défiler verticalement la liste
```

⇒ *horizontalement : « xscrollcommand » et « xview »*

4.3.9 Canevas

Les canevas sont des **zones dans lesquelles on peut dessiner ou écrire :**

Interpréteur Python :

```
>>> canevas = Canvas(fenetre, width=150, height=120, background="black")
>>> canevas.pack()
>>> ligne = canevas.create_line(10, 30, 120, 100, width=10, fill="green")
>>> ovale = canevas.create_oval(30, 50, 130, 90, width=5, outline="blue")
>>> texte = canevas.create_text(90, 20, text="Texte", font="Arial 20 bold italic", fill="orange")
```

Créations possibles :

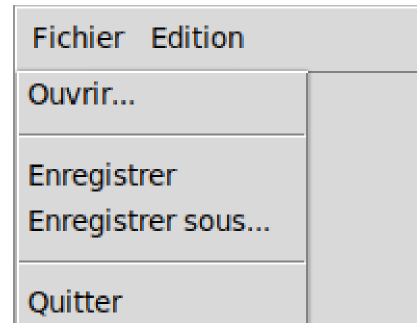
create_arc, create_bitmap, create_image, create_line, create_oval, create_polygon, create_rectangle, create_text, create_window.

4.3.10 Boîtes de dialogue

```
>>> from tkinter.messagebox import *     # sous-module boîtes de dialogue (d'autres existent)
>>>
>>> showinfo("Showinfo", "Texte du message")
'ok'
>>> showwarning("Showwarning", "Texte du message")
'ok'
>>> showerror("Showerror", "Texte du message")
'ok'
>>> askquestion("Askquestion", "Texte du message")
'yes'
>>> askokcancel("Askokcancel", "Texte du message")
False
>>> askretrycancel("Askretrycancel", "Texte du message")
True
>>> askyesno("Askyesno", "Texte du message")
False
>>> askyesnocancel("Askyesnocancel", "Texte du message")
True
```

⇒ *renvoient la réponse de l'utilisateur*

4.3.11 Menus



Interpréteur Python :

```
>>> def commande(): pass     # commande sans code (juste pour l'exemple)
...
>>> barre_menus = Menu(fenetre)     # barre de menus de la fenêtre
>>> fenetre.config(menu=barre_menus)     # l'associe à la fenêtre et l'affiche
>>> menu_fichier = Menu(barre_menus, tearoff=0)     # menu Fichier (sans détachement)
>>> barre_menus.add_cascade(label="Fichier", menu=menu_fichier)     # l'ajoute à la barre
>>> menu_fichier.add_command(label="Ouvrir...", command=commande)     # ajout des options
>>> menu_fichier.add_separator()     # ligne de séparation
>>> menu_fichier.add_command(label="Enregistrer", command=commande)
>>> menu_fichier.add_command(label="Enregistrer sous...", command=commande)
>>> menu_fichier.add_separator()     # ligne de séparation
>>> menu_fichier.add_command(label="Quitter", command=fenetre.destroy)
>>> menu_edition = Menu(barre_menus, tearoff=0)     # menu Edition (sans détachement)
>>> barre_menus.add_cascade(label="Edition", menu=menu_edition)     # l'ajoute à la barre
>>> menu_edition.add_command(label="Couper", command=commande)     # ajout des options
>>> menu_edition.add_command(label="Copier", command=commande)
>>> menu_edition.add_command(label="Coller", command=commande)
```

4.4 Organisation des widgets

4.4.1 Redimensionnement

En général, **les widgets disposent des options « width » et « height »** pour les redimensionner.

Leur unité dépend du widget (caractères, lignes, pixels ou autres).

4.4.2 Positionnement

La méthode « pack » permet de placer / afficher un widget dans son conteneur. Elle dispose de plusieurs options (liste non exhaustive) :

side :	empilement dans le conteneur (TOP par défaut, ou BOTTOM, LEFT, RIGHT)
padx, pady :	espace supplémentaire autour du widget (x pour horizontal, y pour vertical)
fill :	X remplit horizontalement, Y verticalement, BOTH les deux (NONE par défaut)
expand=TRUE :	se développe pour occuper un maximum d'espace dans le conteneur

<u>Attention</u> : avec la méthode « pack », le positionnement des widgets n'est pas totalement libre.

Idéalement, suivre ce modèle d'empilements :

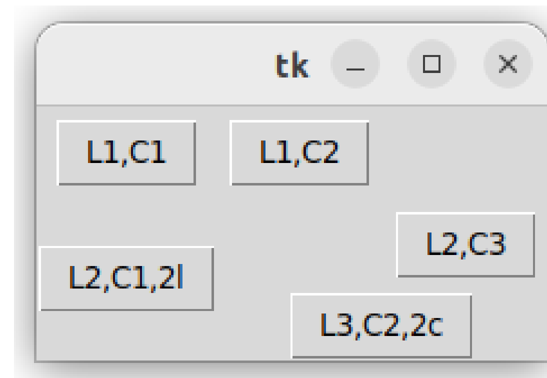
⇒ *exemple avec des canevas*

⇒ *tester en redimensionnant la fenêtre à la main*

Interpréteur Python :

```
>>> Canvas(fenetre, width=50, height=100, background="black").pack(side=LEFT, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=50, height=100, background="gray").pack(side=LEFT, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=50, height=100, background="black").pack(side=RIGHT, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=50, height=100, background="white").pack(side=RIGHT, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=200, height=50, background="black").pack(side=TOP, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=200, height=50, background="gray").pack(side=TOP, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=200, height=50, background="black").pack(side=BOTTOM, padx=5,
pady=5, fill=BOTH, expand=TRUE)
>>> Canvas(fenetre, width=200, height=50, background="white").pack(side=BOTTOM, padx=5,
pady=5, fill=BOTH, expand=TRUE)
```

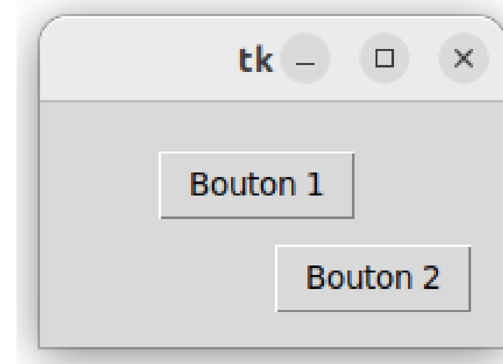
Plus pratique, **la méthode « grid » permet de placer / afficher un widget dans son conteneur par rapport à une grille de lignes / colonnes :**



Interpréteur Python :

```
>>> Button(fenetre, text="L1,C1").grid(row=1, column=1, padx=5, pady=5)
>>> Button(fenetre, text="L1,C2").grid(row=1, column=2, padx=5, pady=5)
>>> Button(fenetre, text="L2,C3").grid(row=2, column=3, padx=5, pady=5)
>>> Button(fenetre, text="L2,C1,2l").grid(row=2, column=1, rowspan=2)     # sur 2 lignes
>>> Button(fenetre, text="L3,C2,2c").grid(row=3, column=2, columnspan=2)     # sur 2 colonnes
```

Et la méthode « place » avec des coordonnées en pixels :



Interpréteur Python :

```
>>> Button(fenetre, text="Bouton 1").place(x=50, y=20)
>>> Button(fenetre, text="Bouton 2").place(x=100, y=60)
```

4.4.3 Regroupement

Les "frames" (cadres) sont des widgets conteneurs pouvant contenir d'autres widgets (ou d'autres "frames").

Ils **facilitent leur organisation** et la réutilisation de blocs de fonctionnalités.

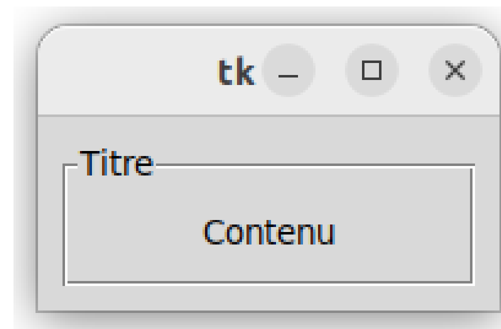
⇒ *souvent utilisés comme classes de base*

Interpréteur Python :

```
>>> cadre = Frame(fenetre, borderwidth=2, relief=GROOVE)
>>> cadre.pack(padx=10, pady=10)
>>> texte = Text(cadre, width=25, height=5)     # Text : autre widget de saisie de texte
>>> texte.pack(side=TOP, padx=5, pady=5)
>>> bouton_valider = Button(cadre, text="Valider")
>>> bouton_valider.pack(side=LEFT, padx=5, pady=5)
>>> bouton_quitter = Button(cadre, text="Quitter", command=fenetre.destroy)
>>> bouton_quitter.pack(side=RIGHT, padx=5, pady=5)
```

⇒ *récupérer le texte : texte.get(1.0, "end-1c")   # caractères : début (ligne.colonne-1), fin-1*

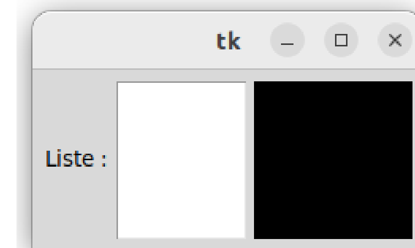
Il existe également **le widget « LabelFrame », un cadre avec un titre :**



Interpréteur Python :

```
>>> cadre = LabelFrame(fenetre, text="Titre")
>>> cadre.pack(padx=10, pady=10)
>>> Label(cadre, text="Contenu", width=20, height=2).pack()
```

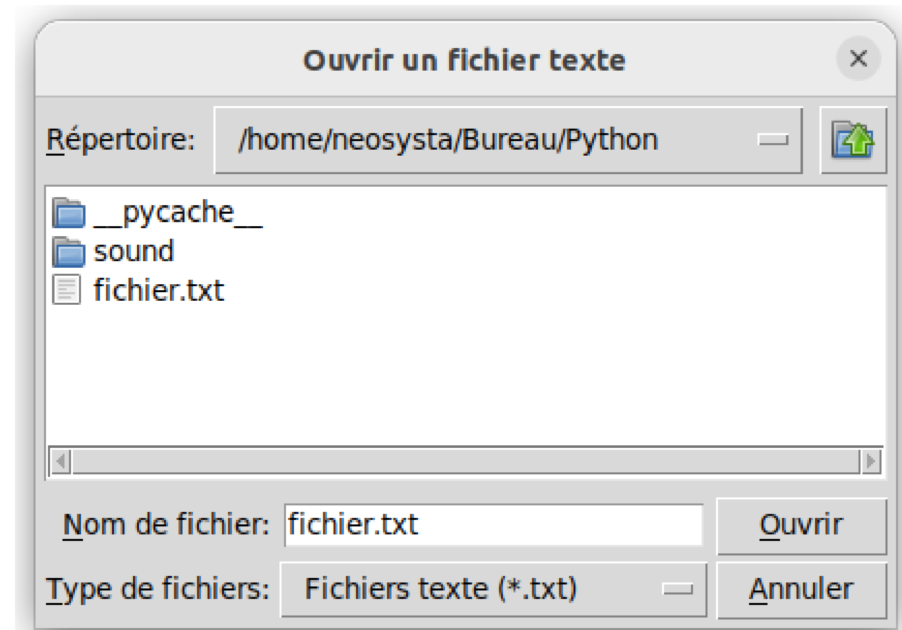
Et le widget « PanedWindow » permettant d'empiler des widgets puis de les redimensionner à la souris :



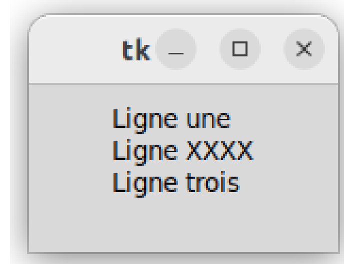
Interpréteur Python :

```
>>> panneaux = PanedWindow(fenetre, orient=HORIZONTAL)
>>> panneaux.pack(side=TOP, padx=5, pady=5, fill=BOTH, expand=TRUE)
>>> label = Label(panneaux, text="Liste :")
>>> panneaux.add(label)
>>> liste = Listbox(panneaux, width=10, height=5)
>>> panneaux.add(liste)
>>> canevas = Canvas(panneaux, width=100, height=100, background="black")
>>> panneaux.add(canevas)
```

4.5 Accès aux fichiers



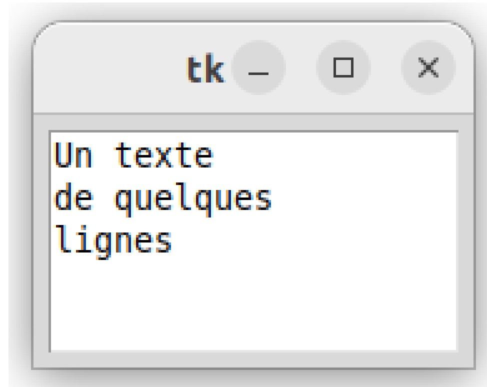
4.5.1 Ouvrir un fichier texte et l'afficher

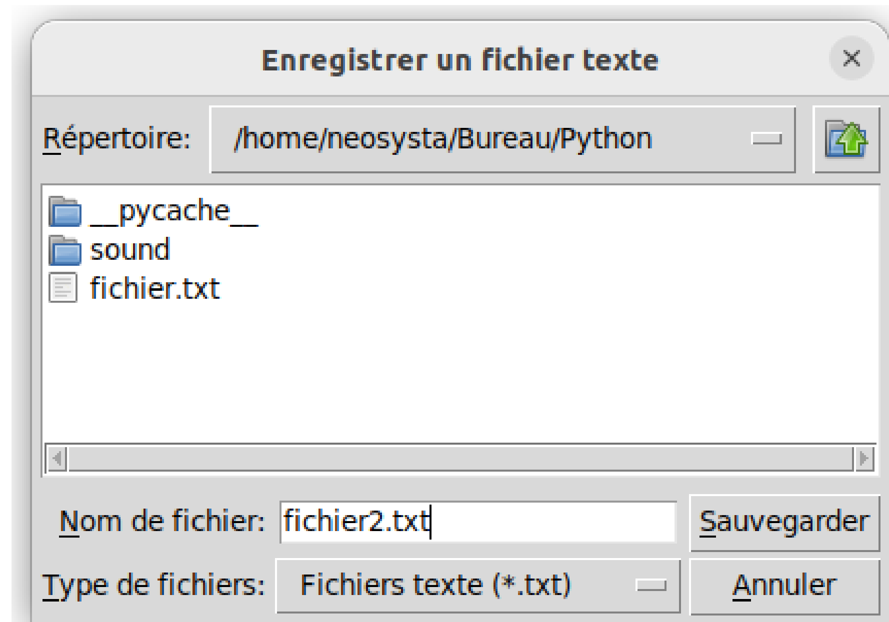


Interpréteur Python :

```
>>> from tkinter import *
>>> from tkinter.filedialog import *     # sous-module des boîtes de dialogue de fichiers
>>>
>>> fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
>>> chemin_fichier = askopenfilename(title="Ouvrir un fichier texte", filetypes=[("Fichiers texte",
".txt"), ("Tous les fichiers", ".*")])     # accès à la boîte de dialogue de sélection du fichier
>>> chemin_fichier
'/home/neosysta/Bureau/Python/fichier.txt'
>>> with open(chemin_fichier, "r") as fichier_texte:
...        contenu = fichier_texte.read()
...        Label(fenetre, text=contenu, justify=LEFT, width=20, height=5).pack()
```

4.5.2 Enregistrer un fichier texte

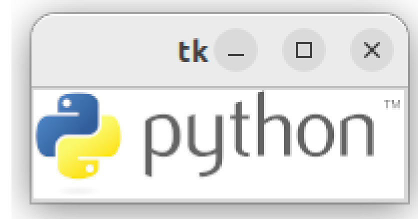


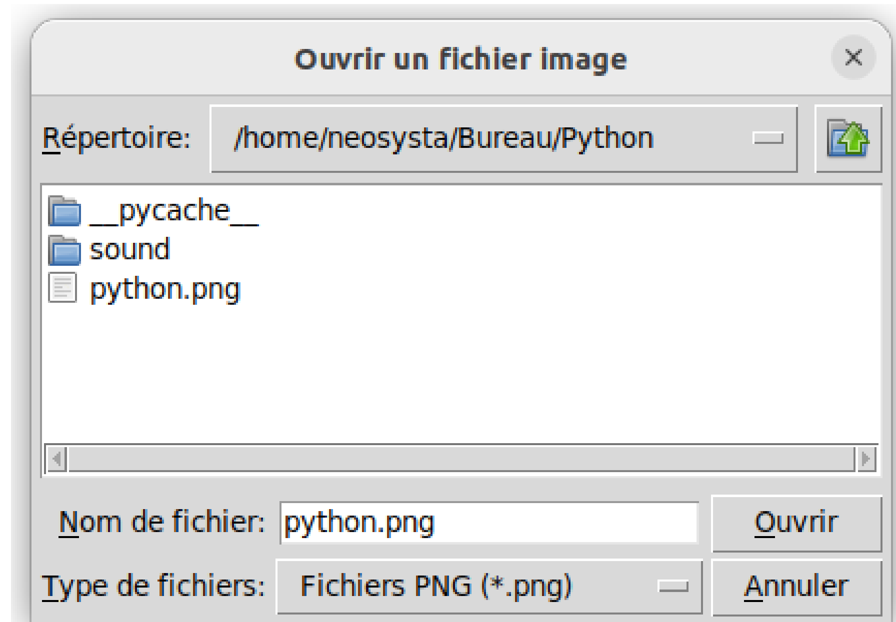


Interpréteur Python :

```
>>> from tkinter import *
>>> from tkinter.filedialog import *     # sous-module des boîtes de dialogue de fichiers
>>>
>>> fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
>>> texte = Text(fenetre, width=20, height=5)
>>> texte.pack(padx=5, pady=5)
>>> texte.insert(END, "Un texte\nde quelques\nlignes")
>>> chemin_fichier = asksaveasfilename(title="Enregistrer un fichier texte", filetypes=[("Fichiers texte", ".txt"), ("Tous les fichiers", ".*")])     # boîte de sauvegarde (prévient si fichier existant)
>>> with open(chemin_fichier, "w") as fichier_texte:
...         fichier_texte.write(texte.get(1.0, "end"))
```

4.5.3 Ouvrir un fichier image et l'afficher





Interpréteur Python :

```
>>> from tkinter import *
>>> from tkinter.filedialog import *     # sous-module des boîtes de dialogue de fichiers
>>>
>>> fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
>>> chemin_fichier = askopenfilename(title="Ouvrir un fichier image", filetypes=[("Fichiers
PNG", ".png"), ("Fichiers GIF", ".gif")])
>>> photo_image = PhotoImage(file=chemin_fichier)     # lit le fichier image
>>> canevas = Canvas(fenetre, width=photo_image.width(), height=photo_image.height())
>>> canevas.pack()
>>> canevas.create_image(0, 0, image=photo_image, anchor=NW)     # NW : en haut à gauche
```

⇒ *« PhotoImage » accepte les formats PGM, PPM, GIF et PNG (voir aussi la librairie « Pillow »)*

4.6 Evénements

4.6.1 Principe

La méthode « bind » permet de lier ("bind" en anglais) à des fenêtres ou des widgets des événements (clavier, souris, etc) déclenchant des actions via des fonctions ou des méthodes (appelées "callbacks").

<u>Sous la forme</u> :

widget.bind("<Evenement>", nom_fonction)

La fonction (ou la méthode) appelée reçoit alors l'événement pour l'exploiter.

⇒ *l'événement doit être précisé entre guillemets (ou apostrophes) et chevrons : "<Evenement>"*

L'événement peut aussi être une séquence d'événements simultanés : chaîne de caractères contenant les motifs d'événements correspondants (séparés par un tiret « - »).

⇒ *par exemple pour une combinaison de touches*

4.6.2 Principaux événements du clavier

L'utilisateur a pressé / relâché une touche quelconque : **Key ou KeyPress / KeyRelease**

L'utilisateur a pressé une touche spéciale :

- Touches de fonction : **F1, F2, ..., F12**
- Touches de déplacement du curseur : **Left, Right, Up, Down**
- Touches "Page Up", "Page Down" : **Prior, Next**
- Touches de début, tabulation, retour arrière : **Home, Tab, BackSpace**
- Touches "Suppr", "Fin", "Inser" : **Delete, End, Insert**
- Touches "Esc" ou "Echap", "Entrée" : **Escape, Return**
- Touches de verrouillage des majuscules, du pavé numérique : **Caps_Lock, Num_Lock**
- Touches "Alt", "Ctrl", "Shift" : **Alt_L, Alt_R, Control_L, Control_R, Shift_L, Shift_R**

L'utilisateur a pressé une touche normale :

- Le nom de l'événement a le même nom que la touche : **a, b, c, ...**
- Touche "Espace" : **space**

L'utilisateur a pressé une combinaison de touches :

- "Alt / Ctrl / Shift + autre touche" : **Alt-, Control-, Shift-**

4.6.3 Principaux événements de la souris

⇒ *1 : gauche ; 2 : milieu ; 3 : droite*

L'utilisateur a déplacé la souris : **Motion**

L'utilisateur a cliqué un bouton : **Button-1, Button-2, Button-3**

L'utilisateur a déplacé la souris en maintenant un bouton : **B1-Motion, B2-Motion, B3-Motion**

L'utilisateur a double-cliqué un bouton : **Double-Button-1, Double-Button-2, Double-Button-3**

L'utilisateur a relâché un bouton : **ButtonRelease-1, ButtonRelease-2, ButtonRelease-3**

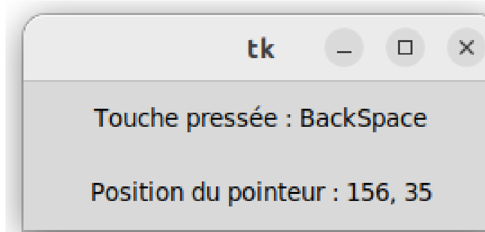
L'utilisateur a utilisé la molette de la souris sur Windows / macOS : **MouseWheel**

L'utilisateur a utilisé la molette de la souris sur Unix / Linux (haut et bas) : **Button-4, Button-5**

L'utilisateur a déplacé la souris et elle est entrée dans un widget : **Enter**

L'utilisateur a déplacé la souris et elle est sortie d'un widget : **Leave**

4.6.4 Premier exemple



Code Python :

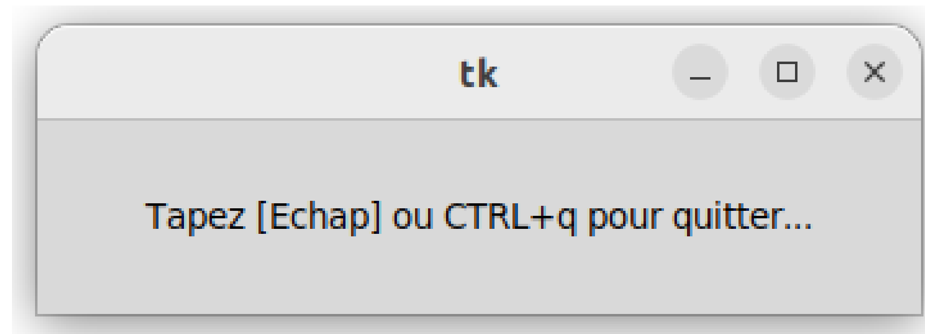
```
from tkinter import *

def clavier(event):     # la fonction reçoit l'événement en paramètre
    touche = event.keysym     # keysym contient le nom de la touche
    label_clavier["text"] = f"Touche pressée : {touche}"

def souris(event):     # la fonction reçoit l'événement en paramètre
    x = event.x     # position x du pointeur
    y = event.y     # position y du pointeur
    label_souris["text"] = f"Position du pointeur : {x}, {y}"

fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
label_clavier = Label(fenetre, width=30, height=1)
label_clavier.pack(padx=10, pady=10)
label_souris = Label(fenetre, width=30, height=1)
label_souris.pack(padx=10, pady=10)
fenetre.bind("<Key>", clavier)     # événement touche pressée
fenetre.bind("<Motion>", souris)     # événement mouvement souris
fenetre.mainloop()     # boucle de la fenêtre jusqu'à fermeture
```

4.6.5 Deuxième exemple



Code Python :

```
from tkinter import *

def quitter(event):     # la fonction reçoit l'événement en paramètre
    fenetre.destroy()

fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
label = Label(fenetre, text="Tapez [Echap] ou CTRL+q pour quitter...", width=40, height=3)
label.pack(padx=10, pady=10)

fenetre.bind("<Escape>", quitter)     # événement touche [Echap]
fenetre.bind("<Control-q>", quitter)     # événement combinaison de touches CTRL+q

fenetre.mainloop()     # boucle de la fenêtre jusqu'à fermeture
```

4.6.6 Troisième exemple

Code Python :

```
from tkinter import *

def dessiner(event):     # la fonction reçoit l'événement en paramètre
    canevas.create_oval(event.x-2, event.y-2, event.x+2, event.y+2, \
    outline="coral", fill="coral")

fenetre = Tk()     # création de l'objet fenetre (et l'affiche)
canevas = Canvas(fenetre, width=300, height=200, background="black")
canevas.pack(padx=10, pady=10)
canevas.bind("<B1-Motion>", dessiner)     # événement click gauche + mouvement souris
fenetre.mainloop()     # boucle de la fenêtre jusqu'à fermeture
```

4.6.7 Attributs de l'objet event

L'objet « event » de la fonction ou de la méthode "callback" contient des informations sur l'événement qui a été déclenché à travers différents attributs (liste non exhaustive) :

widget : instance du widget qui a déclenché l'événement

type : type de l'événement représenté par un nombre

keysym : nom de la touche (seulement pour un événement clavier)

keycode : code unique de la touche (seulement pour un événement clavier)

num : numéro du bouton (seulement pour un événement souris)

x, y : position de la souris par rapport au widget

x_root, y_root : position de la souris par rapport à l'écran

width, height : nouvelles largeur et hauteur du widget

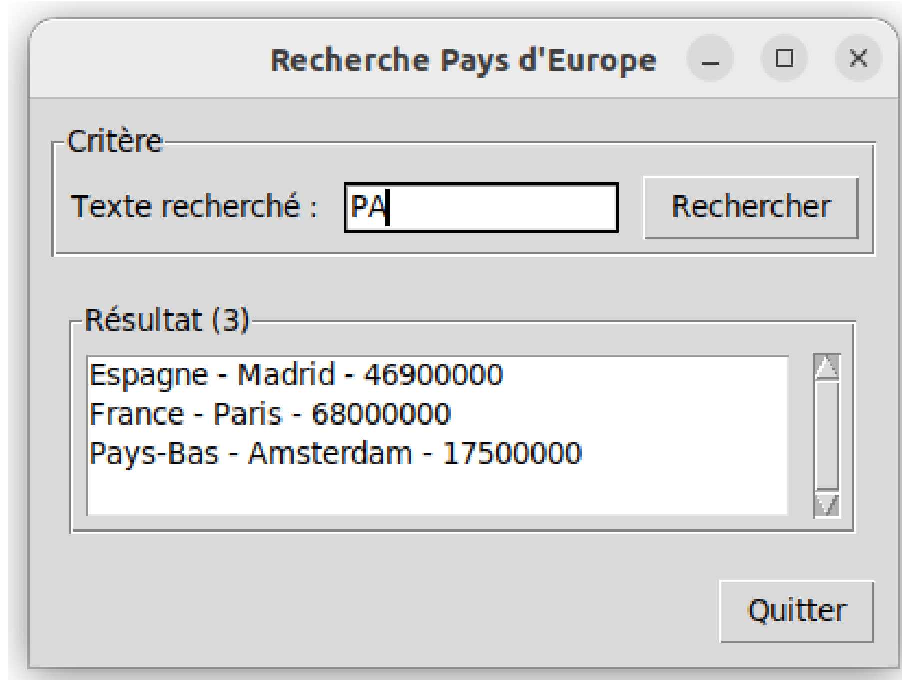
⇒ *utiles pour contrôler les actions à effectuer (voir exemples précédents)*

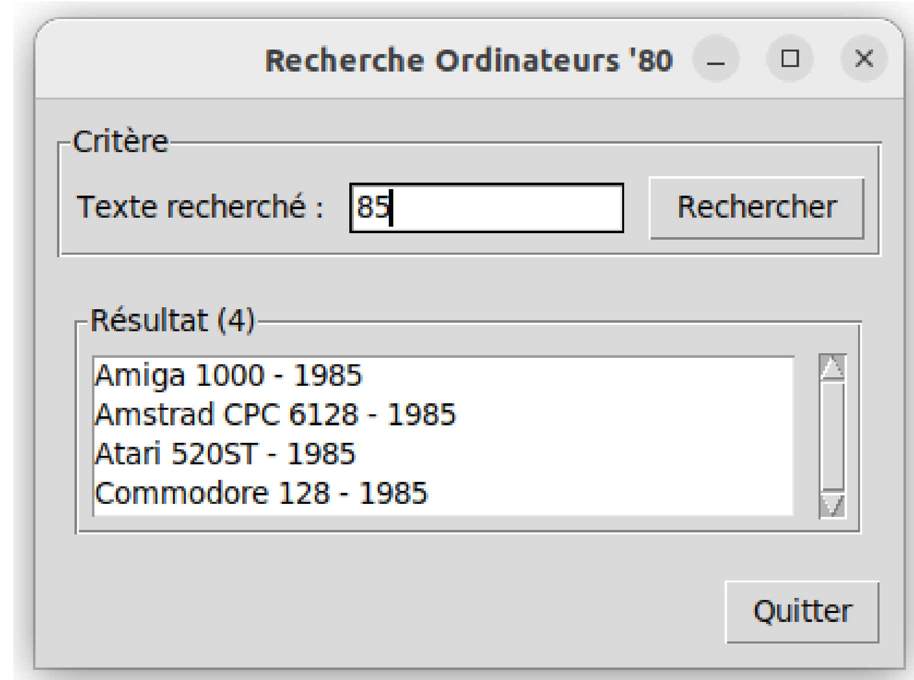
4.7 Classes d'interfaces

Idéalement, définir les interfaces (fenêtres, cadres / frames,…) dans des classes.

⇒ *réutilisation, héritage, polymorphisme,…*

Exemple de création et d'utilisation d'une classe fenêtre :





Code Python :

```
from tkinter import *

class FenetreRecherche(Tk):    # hérite de la classe Tk

    def __init__(self, donnees):
        super().__init__()    # appelle __init__ de la classe mère Tk car redéfini
        self.donnees = sorted(donnees)    # liste de données triée
        self.creer_widgets()    # crée les widgets de l'interface
        self.rechercher_donnees()    # préaffiche la liste résultat (recherche à vide)

    def creer_widgets(self):
        # Cadre "Critère" :
        self.cadre_critere = LabelFrame(self, text="Critère")
        self.cadre_critere.pack(padx=10, pady=10)
        self.label_recherche = Label(self.cadre_critere, text="Texte recherché :")
        self.label_recherche.pack(side=LEFT, padx=5, pady=5)
        self.texte_recherche = StringVar()
        self.zone_recherche = Entry(self.cadre_critere, textvariable=self.texte_recherche, \
        width=15)
        self.zone_recherche.pack(side=LEFT, padx=5, pady=5)
        self.zone_recherche.focus_set()
```

```
        self.bouton_rechercher = Button(self.cadre_critere, text="Rechercher", \
        command=self.rechercher_donnees)
        self.bouton_rechercher.pack(side=RIGHT, padx=5, pady=5)
        # Touche [Entrée] :
        self.bind("<Return>", self.rechercher_donnees)
        # Cadre "Résultat" :
        self.cadre_resultat = LabelFrame(self)
        self.cadre_resultat.pack(padx=10, pady=10)
        self.liste_resultat = Listbox(self.cadre_resultat, width=40, height=4)
        self.liste_resultat.pack(side=LEFT, padx=5, pady=5)
        self.barre_resultat = Scrollbar(self.cadre_resultat)
        self.barre_resultat.pack(side=RIGHT, padx=5, pady=5, fill=BOTH)
        self.liste_resultat.config(yscrollcommand=self.barre_resultat.set)
        self.barre_resultat.config(command=self.liste_resultat.yview)
        # Bouton "Quitter" :
        self.bouton_quitter = Button(self, text="Quitter", command=self.destroy)
        self.bouton_quitter.pack(side=RIGHT, padx=10, pady=10)

    def rechercher_donnees(self, event=None):     # event=None : pour les appels sans event
        # (Re)vide la liste résultat
        self.liste_resultat.delete(0, END)
        nombre = 0
```

```
        # Parcourt les lignes de données
        for ligne in self.donnees:
            # Convertit la ligne courante en texte (séparateur " - ")
            ligne_texte = " - ".join(map(str, ligne))     # map applique str à chaque élément
            # Si le texte recherché est dans la ligne courante
            if self.texte_recherche.get().lower() in ligne_texte.lower():
                # Ajoute la ligne dans la liste résultat
                self.liste_resultat.insert(END, ligne_texte)
                nombre += 1
        self.cadre_resultat["text"] = f"Résultat ({nombre})"

# Programme principal :

if __name__ == "__main__":     # permet de réutiliser ce programme comme module (import)

    # Fenêtre Recherche Pays d'Europe :
    pays_europe = [("France", "Paris", 68000000), ("Allemagne", "Berlin", 84000000), \
    ("Espagne", "Madrid", 46900000), ("Belgique", "Bruxelles", 11500000), \
    ("Portugal", "Lisbonne", 10300000), ("Italie", "Rome", 59000000), \
    ("Grèce", "Athènes", 10400000), ("Autriche", "Vienne", 8900000), \
    ("Pays-Bas", "Amsterdam", 17500000), ("Danemark", "Copenhague", 5900000)]
```

```
fenetre_recherche_pays_europe = FenetreRecherche(pays_europe)
fenetre_recherche_pays_europe.title("Recherche Pays d'Europe")
fenetre_recherche_pays_europe.mainloop()

# Fenêtre Recherche Ordinateurs '80 :
ordinateurs_80 = [("Amiga 1000", 1985), ("Commodore 64", 1982), ("Atari 520ST", 1985), \
("Amstrad CPC 464", 1984), ("Macintosh", 1984), ("Thomson MO5", 1984), \
("Oric-1", 1982), ("Amstrad CPC 6128", 1985), ("Amiga 500", 1987), \
("Commodore 128", 1985), ("Thomson TO7", 1982), ("ZX Spectrum", 1982)]

fenetre_recherche_ordinateurs_80 = FenetreRecherche(ordinateurs_80)
fenetre_recherche_ordinateurs_80.title("Recherche Ordinateurs '80")
fenetre_recherche_ordinateurs_80.mainloop()
```

⇒ *tester également les attributs et les méthodes des objets créés*

5 Accès aux bases de données

5.1 API standard

Python permet d'interagir facilement avec les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR).

Cette facilité est en partie **due à la définition d'une API standard d'accès aux bases de données** décrite par la PEP 249 « Python Database API Specification v2.0 » :

https://peps.python.org/pep-0249/

⇒ *API : Application Programming Interface / Interface de Programmation d'Application*
⇒ *PEP : Python Enhancement Proposal / Proposition d'Amélioration de Python*

Pour accéder à une base de données, **il suffit d'installer le paquet du module permettant de se connecter au SGBDR** concerné.

La procédure de connexion peut avoir des spécificités selon le SGBDR, mais **l'interaction avec la base de données reste globalement standard** (comme les SGBDR des pages suivantes).

⇒ *facilite le portage des applications d'un SGBDR à l'autre*

5.2 PostgreSQL

5.2.1 Création de la base de données

Dans PostgreSQL, créer la base de données utilisée à l'aide des requêtes SQL suivantes :

create database bibliotheque;

Puis dans la base de données « bibliotheque » (schéma « public » par défaut) :

create table auteurs(id serial primary key, nom varchar(30) not null, prenom varchar(30) not null, date_naissance date, date_deces date);

insert into auteurs (nom, prenom) values ('Verne', 'Jules');
insert into auteurs (nom, prenom) values ('Stevenson', 'Robert Louis');
insert into auteurs (nom, prenom) values ('De Saint-Exupéry', 'Antoine');
insert into auteurs (nom, prenom) values ('Zola', 'Emile');
insert into auteurs (nom, prenom) values ('Hemingway', 'Ernest');

Vérifier : select * from auteurs;

5.2.2 Module psycopg2

Le module « psycopg2 » **permet d'accéder à des bases de données PostgreSQL depuis Python**.

⇒ *pilote le plus populaire pour PostgreSQL (d'autres existent), voir : https://www.psycopg.org*

Pour télécharger et intégrer le module « psycopg2 » à l'environnement Python, saisir la commande suivante dans la Console Windows ou le Terminal Unix / Linux / macOS (normalement même commande pour tous) :

pip install psycopg2-binary ⇒ *version minimale (version complète : pip install psycopg2)*
⇒ *attention : installation par utilisateur système*

Importer le module « psycopg2 » : **import psycopg2**
Aide et fonctionnalités : **help(psycopg2)** ⇒ *liste : dir(psycopg2)*

Précision / rappel :

« pip » est un gestionnaire de paquets utilisé pour installer et gérer des paquets écrits en Python. De nombreux paquets peuvent être trouvés sur le dépôt Python Package Index : https://pypi.org

5.2.3 Exécution de requêtes

Créer l'objet connexion à la base de données :

Interpréteur Python :

```
>>> connexion = psycopg2.connect(user="utilisateur", password="utilisateur", host="127.0.0.1",
port="5432", database="bibliotheque")
>>> connexion.autocommit = True     # activation de l'autocommit (comme PostgreSQL)
```

⇒ *adapter les paramètres de connexion en fonction des cas*

Pour cet objet connexion, créer l'objet curseur servant aux requêtes :

Interpréteur Python :

```
>>> curseur = connexion.cursor()
```

Avec cet objet curseur, exécuter une requête de mise à jour : ⇒ *immédiate en autocommit*

Interpréteur Python :

```
>>> curseur.execute("insert into auteurs (nom, prenom) values ('Hugo', 'Victor');")
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer toutes ses lignes :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchall()     # récupère toutes les lignes retournées (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-Exupéry', 'Antoine', None, None), (4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest', None, None), (6, 'Hugo', 'Victor', None, None)]
>>> curseur.fetchall()     # plus rien à lire (déjà lu)
[]
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer ses lignes une par une :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchone()     # récupère la ligne retournée suivante (tuple)
(1, 'Verne', 'Jules', None, None)
>>> curseur.fetchone()     # récupère la ligne retournée suivante (tuple)
(2, 'Stevenson', 'Robert Louis', None, None)
```

⇒ *retourne « None » quand il n'y a plus de ligne à lire*

Avec cet objet curseur, exécuter une requête SELECT et récupérer un nombre de lignes :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-
Exupéry', 'Antoine', None, None)]
>>> curseur.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest', None, None), (6, 'Hugo', 'Victor',
None, None)]
>>> curseur.fetchmany(3)     # plus rien à lire
[]
```

⇒ *« curseur.rowcount » contient le nombre total de lignes retournées (ou mises à jour)*
⇒ *« curseur.rownumber » contient le numéro de la dernière ligne lue*

Quand terminé, fermer l'objet curseur puis l'objet connexion : ⇒ *désallocations*

Interpréteur Python :

```
>>> curseur.close()
>>> connexion.close()
```

5.3 MySQL

5.3.1 Création de la base de données

Dans MySQL, créer la base de données utilisée à l'aide des requêtes SQL suivantes :

create database bibliotheque;

Puis dans la base de données « bibliotheque » :

create table auteurs(id integer auto_increment primary key, nom varchar(30) not null, prenom varchar(30) not null, date_naissance date, date_deces date);

insert into auteurs (nom, prenom) values ('Verne', 'Jules');
insert into auteurs (nom, prenom) values ('Stevenson', 'Robert Louis');
insert into auteurs (nom, prenom) values ('De Saint-Exupéry', 'Antoine');
insert into auteurs (nom, prenom) values ('Zola', 'Emile');
insert into auteurs (nom, prenom) values ('Hemingway', 'Ernest');

Vérifier : select * from auteurs;

5.3.2 Module mysql.connector

Le module « mysql.connector » **permet d'accéder à des bases de données MySQL depuis Python**.

⇒ *pilote officiel de MySQL (d'autres existent)*
⇒ *voir : https://dev.mysql.com/doc/connector-python/en/*

Pour télécharger et intégrer le module « mysql.connector » à l'environnement Python, saisir la commande suivante dans la Console Windows ou le Terminal Unix / Linux / macOS (normalement même commande pour tous) :

pip install mysql-connector-python ⇒ *attention : installation par utilisateur système*

Importer le module « mysql.connector » : **import mysql.connector**
Aide et fonctionnalités : **help(mysql.connector)** ⇒ *liste : dir(mysql.connector)*

Précision / rappel :

« pip » est un gestionnaire de paquets utilisé pour installer et gérer des paquets écrits en Python. De nombreux paquets peuvent être trouvés sur le dépôt Python Package Index : https://pypi.org

5.3.3 Exécution de requêtes

Créer l'objet connexion à la base de données :

Interpréteur Python :

```
>>> connexion = mysql.connector.connect(user="utilisateur", password="utilisateur",
host="127.0.0.1", port="3306", database="bibliotheque")
>>> connexion.autocommit = True     # activation de l'autocommit (comme MySQL)
```

⇒ *adapter les paramètres de connexion en fonction des cas*

Pour cet objet connexion, créer l'objet curseur servant aux requêtes :

Interpréteur Python :

```
>>> curseur = connexion.cursor()
```

Avec cet objet curseur, exécuter une requête de mise à jour : ⇒ *immédiate en autocommit*

Interpréteur Python :

```
>>> curseur.execute("insert into auteurs (nom, prenom) values ('Hugo', 'Victor');")
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer toutes ses lignes :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchall()     # récupère toutes les lignes retournées (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-Exupéry', 'Antoine', None, None), (4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest', None, None), (6, 'Hugo', 'Victor', None, None)]
>>> curseur.fetchall()     # plus rien à lire (déjà lu)
[]
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer ses lignes une par une :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchone()     # récupère la ligne retournée suivante (tuple)
(1, 'Verne', 'Jules', None, None)
>>> curseur.fetchone()     # récupère la ligne retournée suivante (tuple)
(2, 'Stevenson', 'Robert Louis', None, None)
```

⇒ *retourne « None » quand il n'y a plus de ligne à lire*

Avec cet objet curseur, exécuter une requête SELECT et récupérer un nombre de lignes :

Interpréteur Python :

```
>>> curseur.execute("select * from auteurs;")     # exécute la requête
>>> curseur.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-
Exupéry', 'Antoine', None, None)]
>>> curseur.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest', None, None), (6, 'Hugo', 'Victor',
None, None)]
>>> curseur.fetchmany(3)     # plus rien à lire
[]
```

⇒ *« curseur.rowcount » contient le nombre cumulé de lignes lues (ou mises à jour)*
⇒ *« curseur.rownumber » est inexistant pour MySQL*

Quand terminé, fermer l'objet curseur puis l'objet connexion : ⇒ *désallocations*

Interpréteur Python :

```
>>> curseur.close()
True
>>> connexion.close()
```

5.4 SQLite

5.4.1 Spécificités

SQLite est un **système de gestion de bases de données relationnelles embarquées, libre et open source**.

Contrairement aux autres SGBDR, **il n'est pas orienté "client-serveur", ses bases de données sont uniquement exploitées en local** (idéal pour les logiciels devant embarquer la base de données, donc sans partage des données en réseau avec d'autres utilisateurs).

Pour cela, **il est également ultra-léger (environ 1 Mo)**.

Autre intérêt : **l'intégralité d'une base de données est stockée dans un fichier indépendant de la plateforme**. La gestion des droits d'accès et de modification des données se fait alors par le système de fichiers du système d'exploitation.

Intégré à de nombreux logiciels grand public, produits professionnels, bibliothèques standards, langages de programmation, systèmes embarqués, smartphones et tablettes, **SQLite est au final le moteur de bases de données le plus utilisé au monde**.

5.4.2 Création de la base de données

Dans la Console Windows ou le Terminal Unix / Linux / macOS, créer la base de données utilisée (normalement même commande pour tous) :

sqlite3 bibliotheque.db ⇒ *fichier créé dans le répertoire courant*

Puis dans SQLite et dans la base de données « bibliotheque.db » :

create table auteurs(id integer primary key not null, nom varchar(30) not null, prenom varchar(30) not null, date_naissance date, date_deces date);

insert into auteurs (nom, prenom) values ('Verne', 'Jules');
insert into auteurs (nom, prenom) values ('Stevenson', 'Robert Louis');
insert into auteurs (nom, prenom) values ('De Saint-Exupéry', 'Antoine');
insert into auteurs (nom, prenom) values ('Zola', 'Emile');
insert into auteurs (nom, prenom) values ('Hemingway', 'Ernest');

Vérifier : select * from auteurs;

5.4.3 Module sqlite3

Le module « sqlite3 » **permet d'accéder à des bases de données SQLite depuis Python**.

Fourni avec Python et respecte l'API standard de la PEP 249 « Python Database API Specification v2.0 ».

⇒ *voir : https://docs.python.org/3/library/sqlite3.html*

⇒ *souvent utilisé lors des développements avant de basculer vers le SGBDR final*

⇒ *d'autres pilotes existent*

Importer le module « sqlite3 » : **import sqlite3**
Aide et fonctionnalités : **help(sqlite3)** ⇒ *liste : dir(sqlite3)*

5.4.4 Exécution de requêtes

Créer l'objet connexion à la base de données :

Interpréteur Python :

```
>>> connexion = sqlite3.connect("bibliotheque.db", isolation_level=None)
```

⇒ *si le fichier « bibliotheque.db » existe dans le répertoire courant, sinon le crée*
⇒ *« isolation_level=None » active l'autocommit (comme SQLite)*
⇒ *avec SQLite, pas d'utilisateur ni d'adresse de serveur*

Pour cet objet connexion, créer l'objet curseur servant aux requêtes :

Interpréteur Python :

```
>>> curseur = connexion.cursor()
```

Avec cet objet curseur, exécuter une requête de mise à jour : ⇒ *immédiate en autocommit*

Interpréteur Python :

```
>>> resultat = curseur.execute("insert into auteurs (nom, prenom) values ('Hugo', 'Victor');")
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer toutes ses lignes :

Interpréteur Python :

```
>>> resultat = curseur.execute("select * from auteurs;")     # exécute la requête
>>> resultat.fetchall()     # récupère toutes les lignes retournées (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-
Exupéry', 'Antoine', None, None), (4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest',
None, None), (6, 'Hugo', 'Victor', None, None)]
>>> resultat.fetchall()     # plus rien à lire (déjà lu)
[]
```

Avec cet objet curseur, exécuter une requête SELECT et récupérer ses lignes une par une :

Interpréteur Python :

```
>>> resultat = curseur.execute("select * from auteurs;")     # exécute la requête
>>> resultat.fetchone()     # récupère la ligne retournée suivante (tuple)
(1, 'Verne', 'Jules', None, None)
>>> resultat.fetchone()     # récupère la ligne retournée suivante (tuple)
(2, 'Stevenson', 'Robert Louis', None, None)
```

⇒ *retourne « None » quand il n'y a plus de ligne à lire*

Avec cet objet curseur, exécuter une requête SELECT et récupérer un nombre de lignes :

Interpréteur Python :

```
>>> resultat = curseur.execute("select * from auteurs;")     # exécute la requête
>>> resultat.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(1, 'Verne', 'Jules', None, None), (2, 'Stevenson', 'Robert Louis', None, None), (3, 'De Saint-Exupéry', 'Antoine', None, None)]
>>> resultat.fetchmany(3)     # récupère les 3 lignes retournées suivantes (liste de tuples)
[(4, 'Zola', 'Emile', None, None), (5, 'Hemingway', 'Ernest', None, None), (6, 'Hugo', 'Victor', None, None)]
>>> resultat.fetchmany(3)     # plus rien à lire
[]
```

⇒ *l'attribut « rowcount » contient le nombre de lignes mises à jour ou « -1 » en SELECT*
⇒ *l'attribut « rownumber » est inexistant pour SQLite*

Quand terminé, fermer l'objet curseur puis l'objet connexion : ⇒ *désallocations*

Interpréteur Python :

```
>>> curseur.close()
>>> connexion.close()
```

6 Compléments

6.1 Programmation système

6.1.1 Flux standards

Les flux standards sont des **canaux de communication avec l'extérieur du programme, ils sont au nombre de trois :**

La sortie standard, sur laquelle sortent les messages normaux, par défaut redirigée vers l'écran (par exemple « print »).

La sortie d'erreurs standard, sur laquelle sortent les messages d'erreurs (exceptions), par défaut également redirigée vers l'écran.

L'entrée standard, sur laquelle arrivent les informations en entrée, par défaut reliée au clavier (par exemple « input »).

Le module « sys » permet notamment d'accéder aux objets représentant ces flux standards.

⇒ *le module « sys » dispose de diverses fonctionnalités permettant d'interagir avec le système*

Interpréteur Python :

```
>>> import sys
>>> sys.stdout     # sortie standard
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
>>> sys.stderr     # sortie d'erreurs standard
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>
>>> sys.stdin     # entrée standard
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
```

Ce sont des **objets de la même classe « _io.TextIOWrapper » que les fichiers** ouverts grâce à la fonction intégrée « open ».

On peut donc **écrire et lire dans les flux standards en utilisant les méthodes « write »** (pour les deux sorties stdout et stderr) **et « read »** (pour l'entrée stdin) :

Interpréteur Python :

```
>>> sys.stdout.write("Exemple\n")
Exemple
8
```

⇒ *le nombre retourné correspond au nombre de caractères écrits (incluant le retour à la ligne)*

Et pour rediriger les flux standards dans un fichier quelconque :

```
>>> fichier = open("fichier.txt", "w")
>>> sys.stdout = fichier
>>> print("Redirection de la sortie standard\n")
>>> fichier.close()
>>> sys.stdout = sys.__stdout__     # rétablit la sortie standard normale
```

```
>>> fichier = open("fichier.txt", "w")
>>> sys.stderr = fichier
>>> int("texte")
>>> fichier.close()
>>> sys.stderr = sys.__stderr__     # rétablit la sortie d'erreurs standard normale
```

```
>>> fichier = open("fichier.txt", "r")
>>> sys.stdin = fichier
>>> contenu = input()
>>> fichier.close()
>>> sys.stdin = sys.__stdin__     # rétablit l'entrée standard normale
>>> print(contenu)
```

6.1.2 Signaux

Dans un système informatique, un signal est un **message (action) envoyé à un programme (processus)**. C'est une **technique de communication inter-processus**.

Un signal **peut être intercepté dans un programme** (pour un traitement particulier).

⇒ *attention, tous les signaux ne sont pas présents dans tous les systèmes*

Le module « signal » dispose de fonctionnalités permettant de gérer les signaux, à importer au préalable :

import signal

Le signal 2 ou SIGINT correspond au CTRL+C (arrêt de programme) :

Interpréteur Python :

```
>>> signal.SIGINT     # référence du signal 2
<Signals.SIGINT: 2>
```

Exemple d'interception du signal 2 / SIGINT dans un programme :

Code Python :

```
import signal

def quitter(signal, frame):     # fonction appelée par le signal 2 / SIGINT
    # signal contient le signal concerné
    # frame contient le contexte du signal
    print("Programme quitté par CTRL+C.")
    exit(0)     # sortie du programme avec code retour 0 (sans erreur)

# Pour associer le signal à la fonction :
signal.signal(signal.SIGINT, quitter)

# Traitement (infini) du programme :
print("Programme lancé...")
while True:
    continue
```

⇒ *le programme s'arrête par CTRL+C qui appelle la fonction « quitter »*

⇒ *liste des signaux existants : voir « help(signal) »*

6.1.3 Arguments en ligne de commande

Le module « sys » dispose de la liste « argv » qui contient les arguments transmis à un programme lors de son appel en ligne de commande :

Code Python :

```
import sys

print(sys.argv)
```

Console ou Terminal :

```
$ python3 programme.py
['programme.py']
$ python3 programme.py un 2 trois "un argument avec espaces" etc
['programme.py', 'un', '2', 'trois', 'un argument avec espaces', 'etc']
$ python3 programme.py un 2 trois 'un argument avec espaces' etc
['programme.py', 'un', '2', 'trois', 'un argument avec espaces', 'etc']
```

⇒ *rappel : commande « python » sur Windows ou « python3 » sur Unix / Linux / macOS*

⇒ *le premier élément de la liste « sys.argv » est toujours le nom du programme*

Si les arguments transmis sont limités ou figés, ils peuvent être traités de façon classique.

Sinon, **le module « getopt » dispose de la fonction « getopt » qui permet la gestion des options courtes** (une lettre précédée de « - ») **et des options longues** (un mot précédé de « -- ») suivies ou non **d'un seul argument chacune**.

⇒ *format Unix / Linux / macOS*

La fonction « getopt.getopt » prend en paramètres :

- la liste des arguments à interpréter sans le nom du programme, donc « sys.argv[1:] »
- la chaîne des lettres des options courtes (sans « - »)
- la liste (optionnelle) des mots des options longues (sans « -- »)

⇒ *s'ils ont besoin d'un argument, une lettre d'option courte doit être suivie de « : » et un mot d'option longue de « = »*

Et retourne :

- une liste de couples (tuples) « option, argument »
- une liste des arguments non interprétés (si besoin, peuvent être traités séparément)

Exemple de programme (compteur.py) avec gestion d'options courtes et longues :

Code Python :

```
import sys, getopt

try:
    options, autres_args = getopt.getopt(sys.argv[1:], "hc:d:", ["help", "compte=", "decompte="])
except getopt.GetoptError as erreur:
    print(erreur)
    exit(1)

for option, argument in options:
    if option in ("-h", "--help"):
        print("Pour compter : -c 10, ou --compte=10. Pour décompter : -d 10, ou --decompte=10.", 
end=" ")
    elif option in ("-c", "--compte"):
        for nombre in range(1, int(argument)+1): print(nombre, end=" ")
    elif option in ("-d", "--decompte"):
        for nombre in range(int(argument), 0, -1): print(nombre, end=" ")
print()
```

Console ou Terminal :

```
$ python3 compteur.py -h
Pour compter : -c 10, ou --compte=10. Pour décompter : -d 10, ou --decompte=10.
$ python3 compteur.py --help
Pour compter : -c 10, ou --compte=10. Pour décompter : -d 10, ou --decompte=10.
$ python3 compteur.py -c 3
1 2 3
$ python3 compteur.py --compte=3
1 2 3
$ python3 compteur.py -d 3
3 2 1
$ python3 compteur.py --decompte=3
3 2 1
$ python3 compteur.py -a
option -a not recognized
$ python3 compteur.py --add
option --add not recognized
$ python3 compteur.py -c
option -c requires argument
$ python3 compteur.py --compte
option --compte requires argument
$ python3 compteur.py --decompte=5 -h -c 3 --help -d 4 --compte=2 abc 123 def 456
5 4 3 2 1 Pour compter : -c 10, ou --compte=10. Pour décompter : -d 10, ou --decompte=10. 1 2
3 Pour compter : -c 10, ou --compte=10. Pour décompter : -d 10, ou --decompte=10. 4 3 2 1 1 2
```

6.1.4 Vérifier le système d'exploitation

import sys
sys.platform ⇒ *retourne le système d'exploitation utilisé (win32, linux, darwin,...)*

6.1.5 Commandes système

Le module « os » dispose de la fonction « system » permettant d'exécuter une commande du système d'exploitation :

<u>Exemples sur Windows</u> :
os.system("cls") # efface l'écran de la console
os.system("dir") # liste le répertoire courant

<u>Exemples sur Unix / Linux / macOS</u> :
os.system("clear") # efface l'écran du terminal
os.system("ls") # liste le répertoire courant

⇒ *affiche le résultat de la commande et renvoie son code de retour*
⇒ *le code retour est récupérable mais pas l'affichage de la commande*

Le module « os » dispose également de la classe « popen » permettant d'exécuter une commande du système d'exploitation et d'en récupérer l'affichage :

Interpréteur Python :

```
>>> import os
>>>
>>> commande = os.popen("echo TEXTE")
>>> commande.read()
'TEXTE\n'
>>> commande.read()
''
>>> commande = os.popen("dir")     # ou "ls"
>>> resultat = commande.read()
>>> print(resultat)
boucler.py   donnees      fichier.txt          __pycache__     signal2.py      testspackages.py
client.py    fcttests.py  helloworld.py        serveur1client.py   sound       tests.py
compteur.py  fenetre.py   objet-dictionnaire  serveurNclients.py  testboucler.py

>>> commande.read()
''
```

⇒ *la méthode « read » lit l'affichage de la commande exécutée par « popen »*
⇒ *bloque le programme jusqu'à la fin de la commande puis retourne son affichage*

6.2 Expressions régulières

6.2.1 Définition

Une expression régulière est un ensemble de caractères particuliers (formant le langage d'expressions régulières), appelé un format (ou motif, "pattern" en anglais), **qui permet de décrire un ensemble de chaînes de caractères à reconnaître** (rechercher).

⇒ *"regular expressions" en anglais, souvent abrégées en "regex"*
⇒ *standardisées et incluses dans de nombreux outils et langages informatiques*

Voir : https://fr.wikipedia.org/wiki/Expression_régulière

6.2.2 Regex courantes

chaîne : recherche la chaîne en question
. : un caractère et un seul (quel qu'il soit)
expr1|expr2 : expression1 ou expression2

^expr : expression en début de chaîne ou de ligne
expr$: expression en fin de chaîne ou de ligne
[caract] : un des caractères entre crochets (intervalles possibles : [A-Za-z0-9])
[^caract] : un caractère n'étant pas entre crochets
expr? : expression qui précède présente zéro ou une fois
expr+ : expression qui précède répétée une ou plusieurs fois
expr* : expression qui précède répétée zéro ou plusieurs fois
expr{n} : exactement n occurrences de l'expression qui précède
expr{n,m} : entre n et m occurrences de l'expression qui précède
expr{n,} : au moins n occurrences de l'expression qui précède
(expr) : groupement de l'expression entre parenthèses (traitée comme élément unitaire)

⇒ *les expressions régulières peuvent se combiner (pour une expression complexe)*

6.2.3 Module re

Le module « re » a été **spécialement conçu pour travailler avec les expressions régulières**.

Importer le module « re » : **import re**
Aide et fonctionnalités : **help(re)** ⇒ *liste : dir(re)*

6.2.4 Raw strings

Important :

Pour que les caractères spéciaux comme « \n » ne soient pas interprétés dans les expressions régulières, il faut les échapper avec un antislash « \ » (donc « \\n »).

Le préfixe « r » ("raw string" / chaîne brute) devant une chaîne d'expression régulière force la non-interprétation de tous ses caractères spéciaux. ⇒ *plus pratique*

La fonction **« repr » convertit également une chaîne en "raw string"**.

Interpréteur Python :

```
>>> r"\n\t"
'\\n\\t'
>>> chaine = "\n\t"
>>> chaine_brute = repr(chaine)
>>> chaine_brute
"'\\n\\t'"
```

⇒ *dans tous les cas, un symbole d'expression régulière doit être échappé par un « \ » pour ne pas être interprété (caractère brut)*

6.2.5 Rechercher

Principales fonctions de recherche du module « re » :

match : détermine si l'expression régulière correspond dès le début de la chaîne

search : analyse la chaîne à la recherche d'une position où l'expression régulière correspond

findall : trouve toutes les sous-chaînes qui correspondent à l'expression régulière et les renvoie sous forme de liste

⇒ *« match » et « search » renvoient « None » si aucune correspondance n'est trouvée*

⇒ *si elles trouvent une correspondance, un objet de la classe « re.Match » est renvoyé contenant les informations relatives à la correspondance : position de début et de fin, la sous-chaîne qui correspond et d'autres informations*

⇒ *voir « help(re.Match) »*

```
>>> chaine = "email1: tom@mail.com, email2: bill.NONAME@autremail.fr, email3:
BOB@SONMAIL.COM"
>>> resultat = re.match("[a-zA-Z0-9.]+@[a-zA-Z0-9.]+", chaine)
>>> print(resultat)
None
>>> resultat = re.match("email.: [a-zA-Z0-9.]+@[a-zA-Z0-9.]+", chaine)
>>> print(resultat)
<re.Match object; span=(0, 20), match='email1: tom@mail.com'>
>>> if resultat:
...     print("Trouvé :", resultat.group(), "Début :", resultat.start(), "Fin :", resultat.end())
...
Trouvé : email1: tom@mail.com Début : 0 Fin : 20
>>>
>>> resultat = re.search("[a-zA-Z0-9.]+@[a-zA-Z0-9.]+", chaine)
>>> print(resultat)
<re.Match object; span=(8, 20), match='tom@mail.com'>
>>> if resultat:
...     print("Trouvé :", resultat.group(), "Début :", resultat.start(), "Fin :", resultat.end())
...
Trouvé : tom@mail.com Début : 8 Fin : 20
>>>
>>> resultat = re.findall("[a-zA-Z0-9.]+@[a-zA-Z0-9.]+", chaine)
>>> print(resultat)
['tom@mail.com', 'bill.NONAME@autremail.fr', 'BOB@SONMAIL.COM']
```

La méthode « group » de la classe « re.Match » permet également d'extraire les groupes correspondant aux expressions régulières « (expr) », en précisant leur numéro :

Interpréteur Python :

```
>>> resultat = re.match("(email.: )([a-zA-Z0-9.]+@[a-zA-Z0-9.]+)", chaine)
>>> resultat.group()     # ensemble trouvé
'email1: tom@mail.com'
>>> resultat.group(0)     # ensemble trouvé
'email1: tom@mail.com'
>>> resultat.group(1)     # premier groupe
'email1: '
>>> resultat.group(2)     # deuxième groupe
'tom@mail.com'
```

6.2.6 Remplacer

Principales fonctions de remplacement du module « re » :

sub : prend trois paramètres – l'expression à rechercher, par quoi remplacer cette expression, la chaîne d'origine – et renvoie la chaîne modifiée

subn : réalise la même opération que « sub » mais renvoie un tuple (chaîne modifiée, nombre d'expressions remplacées)

Pour les remplacements, on utilise l'expression régulière de groupement « (expr) » :

Dans l'expression à rechercher, on regroupe les parties à extraire entre parenthèses puis dans l'expression de remplacement, on fait référence à un groupe en précédant son numéro d'un « \ ».

```
>>> chaine = "email1: tom@mail.com, email2: bill.NONAME@autremail.fr, email3:
BOB@SONMAIL.COM"
>>> expression = "email(.): ([a-zA-Z0-9.]+)@([a-zA-Z0-9.]+)"
>>>
>>> re.sub(expression, "adresse\\1: nom=\\2 et domaine=\\3", chaine)
'adresse1: nom=tom et domaine=mail.com, adresse2: nom=bill.NONAME et
domaine=autremail.fr, adresse3: nom=BOB et domaine=SONMAIL.COM'
>>>
>>> re.sub(expression, r"adresse\1: nom=\2 et domaine=\3", chaine)
'adresse1: nom=tom et domaine=mail.com, adresse2: nom=bill.NONAME et
domaine=autremail.fr, adresse3: nom=BOB et domaine=SONMAIL.COM'
>>>
>>> re.subn(expression, r"adresse\1: nom=\2 et domaine=\3", chaine)
('adresse1: nom=tom et domaine=mail.com, adresse2: nom=bill.NONAME et
domaine=autremail.fr, adresse3: nom=BOB et domaine=SONMAIL.COM', 3)
```

On peut également nommer les groupes, en précisant :

Après la parenthèse ouvrante du groupe : **?P<nom_groupe>**
Et dans l'expression de remplacement : **\g<nom_groupe>**

Interpréteur Python :

```
>>> chaine = "email1: tom@mail.com, email2: bill.NONAME@autremail.fr, email3:
BOB@SONMAIL.COM"
>>> expression = "email(?P<num>.): (?P<nom>[a-zA-Z0-9.]+)@(?P<domaine>[a-zA-Z0-9.]+)"
>>>
>>> re.sub(expression, r"adresse\g<num>: nom=\g<nom> et domaine=\g<domaine>", chaine)
'adresse1: nom=tom et domaine=mail.com, adresse2: nom=bill.NONAME et
domaine=autremail.fr, adresse3: nom=BOB et domaine=SONMAIL.COM'
```

6.2.7 Compiler

Avec la fonction « compile » du module « re », on peut compiler une expression régulière dans un objet en mémoire (mise en cache pour un gain de performance).

⇒ *utile pour une expression utilisée fréquemment*

L'objet d'une expression régulière compilée dispose ensuite de l'essentiel des fonctions (méthodes) vues précédemment, mais celles-ci ne prennent pas en premier paramètre l'expression (étant directement dans l'objet).

Interpréteur Python :

```
>>> chaine = "email1: tom@mail.com, email2: bill.NONAME@autremail.fr, email3:
BOB@SONMAIL.COM"
>>> expression_compilee = re.compile("email(.): ([a-zA-Z0-9.]+)@([a-zA-Z0-9.]+)")
>>> help(expression_compilee)

>>> expression_compilee.sub(r"adresse\1: nom=\2 et domaine=\3", chaine)
'adresse1: nom=tom et domaine=mail.com, adresse2: nom=bill.NONAME et
domaine=autremail.fr, adresse3: nom=BOB et domaine=SONMAIL.COM'
```

6.3 Temps, dates et heures

6.3.1 Module time

Le module « time » **fournit différentes fonctions liées au temps**.

Importer le module « time » : **import time**
Aide et fonctionnalités : **help(time)** ⇒ *liste : dir(time)*

Attention :

Ce module est toujours présent, mais **toutes les fonctions ne sont pas disponibles sur toutes les plateformes ou peuvent varier d'un système à l'autre**.

6.3.2 Epoch et timestamp

L'epoch est la date de départ du temps et dépend de la plateforme. **Par exemple sur Unix/Linux et en Python, epoch est le 1er janvier 1970 à 00:00:00 (UTC / GMT)**. ⇒ *voir « time.gmtime(0) »*

Le timestamp est le nombre de secondes écoulées depuis le 1er janvier 1970 à 00:00:00, retourné par la fonction « time » du module « time » :

Interpréteur Python :

```
>>> time.time()
1658242809.5653145
>>> debut = time.time()
>>> fin = time.time()
>>> print("Début :", debut, "Fin :", fin)
Début : 1658242812.73855 Fin : 1658242817.307421
>>> debut < fin
True
>>> print("Durée :", fin - debut, "secondes")
Durée : 4.568871021270752 secondes
```

⇒ *pratique pour des comparaisons ou des calculs*

6.3.3 Informations sur une date et heure

La fonction « localtime » du module « time » retourne toutes les informations relatives à une date et heure, sous forme d'un objet avec attributs :

Interpréteur Python :

```
>>> time.localtime()     # date et heure actuelles
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=15, tm_min=19,
tm_sec=30, tm_wday=2, tm_yday=201, tm_isdst=1)
>>> dateheure = time.localtime()
>>> dateheure
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=15, tm_min=19,
tm_sec=39, tm_wday=2, tm_yday=201, tm_isdst=1)
>>> dateheure.tm_year
2022
>>> time.localtime().tm_year
2022
>>> time.localtime(3600*24*365*50)     # prend en paramètre un timestamp
time.struct_time(tm_year=2019, tm_mon=12, tm_mday=20, tm_hour=1, tm_min=0, tm_sec=0,
tm_wday=4, tm_yday=354, tm_isdst=0)
```

La fonction « mktime » du module « time » retourne le timestamp d'un objet date et heure :

Interpréteur Python :

```
>>> dateheure = time.localtime()
>>> dateheure
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=15, tm_min=55,
tm_sec=55, tm_wday=2, tm_yday=201, tm_isdst=1)
>>> timestamp_dateheure = time.mktime(dateheure)
>>> timestamp_dateheure
1658325355.0
>>> time.localtime(timestamp_dateheure)
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=15, tm_min=55,
tm_sec=55, tm_wday=2, tm_yday=201, tm_isdst=1)
```

6.3.4 Temporiser

Temporiser 5 secondes : **time.sleep(5)**

6.3.5 Formater

La fonction « strftime » du module « time » permet de formater un objet date et heure à l'aide de caractères spéciaux (liste non exhaustive) :

%d : jour de 01 à 31
%m : mois de 01 à 12
%y : année sur 2 chiffres
%Y : année sur 4 chiffres
%H : heure de 00 à 23
%M : minutes de 00 à 59
%S : secondes de 00 à 59

Interpréteur Python :

```
>>> time.strftime("%H:%M:%S le %d/%m/%Y.")     # par défaut date et heure actuelles
'19:51:23 le 20/07/2022.'
>>> time.localtime(3600*24*365*50)
time.struct_time(tm_year=2019, tm_mon=12, tm_mday=20, tm_hour=1, tm_min=0, tm_sec=0,
tm_wday=4, tm_yday=354, tm_isdst=0)
>>> time.strftime("%d/%m/%Y", time.localtime(3600*24*365*50))     # avec objet date et heure
'20/12/2019'
```

6.3.6 Module datetime

Le module « datetime » **fournit plusieurs classes pour représenter les dates et heures** (orienté objet).

Importer le module « datetime » : **import datetime**
Aide et fonctionnalités : **help(datetime)** ⇒ *liste : dir(datetime)*

Ce module « datetime » dispose de la classe « date » avec les attributs year, month, day :

Interpréteur Python :

```
>>> madate = datetime.date(2022, 7, 25)
>>> madate
datetime.date(2022, 7, 25)
>>> print(madate)
2022-07-25
>>> print("Année:", madate.year, "Mois:", madate.month, "Jour:", madate.day)
Année: 2022 Mois: 7 Jour: 25
```

La classe « date » du module « datetime » possède notamment les deux méthodes de classe :

date.today() : renvoie la date en cours
date.fromtimestamp(timestamp) : renvoie la date du timestamp passé en paramètre

Interpréteur Python :

```
>>> date1 = datetime.date.today()
>>> print(date1)
2022-07-25
>>> date2 = datetime.date.fromtimestamp(3600*24*365*50)
>>> print(date2)
2019-12-20
>>> date1 > date2
True
```

Le module « datetime » dispose également de la classe « time » :

Interpréteur Python :

```
>>> monheure = datetime.time(13, 45, 10)     # attributs hour, minute, second
>>> print(monheure)
13:45:10
>>> datetime.time.fromisoformat("04:23:01")
datetime.time(4, 23, 1)
```

Et de la classe « datetime » :

```
>>> # Attributs year, month, day, hour, minute, second
>>> dateheure = datetime.datetime(2022, 7, 25, 15, 10, 32)
>>> dateheure
datetime.datetime(2022, 7, 25, 15, 10, 32)
>>> print(dateheure)
2022-07-25 15:10:32
>>> print(f"{dateheure.hour}:{dateheure.minute}:{dateheure.second} le
{dateheure.day}/{dateheure.month}/{dateheure.year}.")
15:10:32 le 25/7/2022.
>>> print(f"{dateheure.time()} le {dateheure.date()}.")
15:10:32 le 2022-07-25.
>>>
>>> datetime.datetime.fromisoformat("2022-07-25 15:10:32")
datetime.datetime(2022, 7, 25, 15, 10, 32)
>>> dateheure1 = datetime.datetime.now()     # date et heure actuelles
>>> print(dateheure1)
2022-07-25 15:57:03.707384
>>> dateheure2 = datetime.datetime.fromtimestamp(3600*24*365*50)
>>> print(dateheure2)
2019-12-20 01:00:00
>>> dateheure1 > dateheure2
True
```

6.4 Mathématiques

6.4.1 Module math

Le module « math » est le **module de base incluant l'essentiel des fonctions mathématiques**.

Importer le module « math » : **import math**
Aide et fonctionnalités : **help(math)** ⇒ *liste : dir(math)*

<u>Quelques précisions</u> :

Fonctions classiques :

Interpréteur Python :

```
>>> math.pow(2, 3)     # puissance (équivalent à : 2 ** 3, existe aussi : pow(2, 3) )
8.0
>>> math.sqrt(10)     # racine carrée
3.1622776601683795
>>> math.fabs(-5.123)     # valeur absolue (existe aussi : abs(-5.123) )
5.123
```

Trigonométrie :

⇒ *inclut les fonctions trigonométriques usuelles (voir aide)*

Attention en Python, par défaut les angles sont en radians (rad).

```
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
>>> math.degrees(math.pi)     # convertit en degrés
180.0
>>> math.radians(180)     # convertit en radians
3.141592653589793
```

Arrondis :

```
>>> math.ceil(5.1)     # arrondit à l'entier supérieur
6
>>> math.floor(5.9)     # arrondit à l'entier inférieur
5
>>> math.trunc(5.9)     # tronque
5
```

Et aussi :

```
>>> round(5.49)
5
>>> round(5.51)
6
>>> round(5.5)
6
```

6.4.2 Module fractions

Le module « fractions » **dispose de fonctionnalités manipulant les fractions**.

Importer le module « fractions » : **import fractions**
Aide et fonctionnalités : **help(fractions)** ⇒ *liste : dir(fractions)*

Sa principale classe est « Fraction » :

from fractions import Fraction
help(Fraction)

Création de fractions :

Interpréteur Python :

```
>>> un_demi = Fraction(1, 2)
>>> un_demi
Fraction(1, 2)
>>>
>>> un_quart = Fraction('1/4')
>>> un_quart
Fraction(1, 4)
>>>
>>> fraction_simplifiee = Fraction(-7, 21)
>>> fraction_simplifiee
Fraction(-1, 3)
>>>
>>> Fraction.from_float(0.25)
Fraction(1, 4)
>>>
>>> float(un_quart)
0.25
```

Manipulation de fractions :

Interpréteur Python :

```
>>> un_cinquieme = Fraction(1, 5)
>>> un_cinquieme + un_cinquieme
Fraction(2, 5)
>>>
>>> un_cinquieme * un_demi
Fraction(1, 10)
>>>
>>> un_demi + 1
Fraction(3, 2)
>>>
>>> 2 / un_demi
Fraction(4, 1)
>>>
>>> un_cinquieme / un_demi
Fraction(2, 5)
```

6.4.3 Module random

Le module « random » **dispose de fonctionnalités générant des séquences aléatoires**.

Importer le module « random » : **import random**
Aide et fonctionnalités : **help(random)** ⇒ *liste : dir(random)*

La fonction « random » génère un nombre aléatoire entre 0 et 1 :

Interpréteur Python :

```
>>> random.random()
0.39750654672798014
```

La fonction « randrange » génère un nombre aléatoire entre « début, fin-1, intervalle » :

Interpréteur Python :

```
>>> random.randrange(3, 10, 3)     # valeurs possibles : 3, 6, 9
6
```

La fonction « randint » génère un nombre aléatoire entre « début, fin » (inclus) :

Interpréteur Python :

```
>>> random.randint(5, 10)     # valeur entre 5 et 10
8
```

La fonction « choice » renvoie au hasard un élément d'une séquence :

Interpréteur Python :

```
>>> random.choice(["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"])
'mercredi'
```

La fonction « shuffle » mélange au hasard une séquence :

Interpréteur Python :

```
>>> lettres = ["P", "Y", "T", "H", "O", "N"]
>>> random.shuffle(lettres)
>>> lettres
['T', 'N', 'O', 'H', 'P', 'Y']
```

6.5 Mots de passe

6.5.1 Saisie

Le module « getpass » dispose de la fonction « getpass » permettant la saisie d'un mot de passe en le masquant :

Interpréteur Python :

```
>>> from getpass import getpass
>>> mot_de_passe = getpass()
Password:
>>> mot_de_passe
'Azerty-123'
>>>
>>> mot_de_passe = getpass("Saisissez votre mot de passe : ")
Saisissez votre mot de passe :
>>> mot_de_passe
'Qwerty-987'
```

⇒ *utilisation similaire à la fonction « input »*

6.5.2 Hachage

Le module « hashlib » propose une interface commune à divers algorithmes de hachage.

Importer le module « hashlib » : **import hashlib**
Aide et fonctionnalités : **help(hashlib)** ⇒ *liste : dir(hashlib)*

Il dispose des listes (ensembles) :

algorithms_guaranteed : algorithmes garantis d'une plateforme à l'autre
algorithms_available : algorithmes garantis et ceux propres à la plateforme

Interpréteur Python :

```
>>> hashlib.algorithms_guaranteed
{'sha256', 'blake2s', 'sha3_384', 'sha512', 'sha1', 'shake_256', 'md5', 'sha384', 'sha224',
'blake2b', 'sha3_224', 'sha3_256', 'shake_128', 'sha3_512'}
>>>
>>> hashlib.algorithms_available
{'sha256', 'blake2s', 'sha1', 'sha512_224', 'sha224', 'sm3', 'md5-sha1', 'ripemd160', 'sha3_384',
'sha512_256', 'shake_128', 'shake_256', 'sha3_224', 'sha3_256', 'sha512', 'whirlpool', 'md5',
'sha384', 'md4', 'blake2b', 'sha3_512'}
```

Exemple avec l'algorithme « sha1 » :

⇒ *attention, le constructeur de la classe « sha1 » prend en paramètre une chaîne de bytes (octets)*

Pour créer un objet de la classe « sha1 » d'après la chaîne à hacher :

Interpréteur Python :

```
>>> mot_de_passe = hashlib.sha1(b"Mot_de_Passe")     # préfixe 'b' pour convertir en bytes
>>> mot_de_passe
<sha1 _hashlib.HASH object @ 0x7f9a10a2d390>
```

La méthode « digest » hache en bytes.
La méthode « hexdigest » hache en chaîne de caractères hexadécimaux.

Interpréteur Python :

```
>>> mot_de_passe.digest()
b'P\\\xd1\xc8\x9c\xd4*\xf3\xdf\xbc%\xc1\xc0\x9eCo\x1dM\xa0\x17'
>>> type(mot_de_passe.digest())
<class 'bytes'>
>>> mot_de_passe.hexdigest()
'505cd1c89cd42af3dfbc25c1c09e436f1d4da017'
>>> type(mot_de_passe.hexdigest())
<class 'str'>
```

Exemple de contrôle de saisie d'un mot de passe :

Code Python :

```
from getpass import getpass
import hashlib

mot_de_passe_hash = hashlib.sha1(b"Azerty-123").hexdigest()

verrouille = True
while verrouille:
    mot_de_passe_saisi = getpass("Saisissez votre mot de passe : ")
    mot_de_passe_saisi_hash = hashlib.sha1(mot_de_passe_saisi.encode()).hexdigest()
    if mot_de_passe_saisi_hash == mot_de_passe_hash:
        verrouille = False
    else:
        print("Mot de passe incorrect.")
print("\nMot de passe accepté.")
```

⇒ *on ne décrypte pas un mot de passe haché, on compare les hachages*

⇒ *la méthode « encode » convertit également une chaîne de caractères en bytes*