LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°86
France MÉTRO. : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

MÉMO PYTHON

75 RECETTES POUR ACCÉLÉRER VOS DÉVELOPPEMENTS

Compatible Raspberry Pi / Windows / Mac OS / Linux

TYPES DE BASE
Exploitez pleinement les listes, dictionnaires et autres types Python

STRUCTURE
Organisez votre code de manière à le rendre efficace et maintenable

DONNÉES & INTERFACES
Interagissez avec vos données et les utilisateurs

OUTILS
Utilisez les bons outils pour développer, débugger et distribuer votre code

Édité par Les Éditions Diamond
L 15066 - 86 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Conception graphique : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

Préface

Comment lire un fichier **XML** ? Comment créer un itérateur ?

Lorsque l'on ne programme pas en **Python** tous les jours, les solutions aux problèmes les plus triviaux ne nous sont pas toujours directement accessibles, enfouies dans un recoin de notre cerveau. Des efforts longs et coûteux en temps et en énergie seront alors nécessaires pour tenter de se les remémorer. Et parfois, malgré des efforts intenses, de nombreux tests et des apports énergétiques salutaires (tablette de chocolat, bonbons divers, etc.), la solution ne vient toujours pas. Il faut alors plonger dans les méandres du Web... avec plus ou moins de réussite. N'existerait-il pas une autre alternative ?

Nous vous proposons ce mois-ci un hors-série au format un peu particulier : un « mémo » Python pour vous présenter de manière synthétique un certain nombre de problèmes simples et plus complexes auxquels vous pourriez être confronté au cours de vos développements. Les problèmes sont groupés en grandes familles dans lesquelles vous pourrez trouver :

- « Les types de base » qui regroupe des problèmes portant sur les chaînes de caractères, les nombres, les listes, les dictionnaires, etc. ;
- « La structuration du code » qui aborde tout ce qui est architecture : fonctions, boucles, programmation orientée objet, etc. ;
- « Données et interfaces » qui traite de l'accès à une base de données (ici **SQLite3**, mais la syntaxe est la même pour **MySQL**, **MariaDB** ou encore **PostgreSQL**), de la manière d'interagir avec le Shell et d'effectuer des calculs en *multi-threading*, de la façon de manipuler des données web et de créer des serveurs ainsi que de la création d'interfaces textuelles ou graphiques ;
- « Les outils » qui expose l'utilisation des outils de l'environnement Python.

Au sein de cette arborescence, la présentation de chaque problème suit la même organisation :

- exposé du problème en partant d'un exemple concret ;
- solution du problème ;
- discussion sur la solution présentée.

L'objectif de ce hors-série est donc clairement de devenir un compagnon de développement Python, un kit de secours à utiliser en cas de blocage. Bien entendu, il est impossible de recenser au sein d'un seul hors-série l'ensemble des cas pouvant se présenter et il a fallu faire un choix. Il est donc possible que vous rencontriez un problème qui n'est pas exposé dans ce « mémo »... il le sera alors peut-être dans un futur hors-série. En attendant cela, j'espère déjà que le présent guide vous aidera dans vos développements !

Tristan Colombo

SOMMAIRE

1 Recettes 1-28

EXPLOITEZ LES TYPES DE BASE

Chercher des sous-chaînes à l'aide d'une expression régulière p.10
Lire des données au clavier p.11
Internationaliser un texte p.12
Utiliser un template pour générer du texte p.14
Enregistrer et charger des matrices p.18
Utiliser des nombres aléatoires p.19
Vérifier la validité d'une date p.20
Lire un fichier texte ligne à ligne p.21
Accéder à un endroit précis d'un fichier p.22
Créer un fichier xml p.24
Lire un fichier xml p.26
Créer un fichier json p.27
Lire un fichier json p.28
Créer un fichier ini p.29
Lire un fichier ini p.30
Créer un fichier hdf5 p.31
Lire un fichier hdf5 p.33
Créer un fichier odt p.34
Lire un fichier odt p.36
Parcourir une liste p.38
Trier une liste d'entiers lue sous forme de chaîne de caractères p.39
Trier avec fct de tri personnalisée p.40
Supprimer les doublons d'une liste p.42
Trier les valeurs d'un dictionnaire p.43

Conserver l'ordre des éléments dans un dictionnaire p.44
Créer un dictionnaire qui soit la fusion de deux dictionnaires p.46
Créer une liste en utilisant la compréhension de listes p.47
Créer un dictionnaire en utilisant la compréhension de dictionnaires p.48

2 Recettes 29-48

ORGANISEZ VOTRE CODE

Créer une fonction ayant un nombre de paramètres infini p.52
Créer une fonction anonyme p.53
Créer une fonction qui renvoie une fonction p.54
Connaître l'état de sortie d'une boucle for p.55
Affecter une variable de manière conditionnelle p.56
Creer une fonction ayant un comportement différent suivant le type de ses paramètres p.58
Créer un décorateur de log p.59
Créer un décorateur indiquant les types attendus pour les paramètres d'une fonction p.60
Créer un itérateur p.61
Créer un itérateur à l'aide d'un générateur p.62
Créer un paquetage de modules p.64
Créer des constantes p.66
Créer un attribut de classe p.67

Créer une méthode de classe p.68
Créer un singleton p.69
Surcharger un opérateur p.70
Rendre un objet callable p.71
Ajouter un attribut à un objet sans le dégrader p.72
Ajouter une méthode à une classe existante p.73
Créer un sélecteur de méthode p.74

3 Recettes 49-67

INTERAGISSEZ AVEC VOS DONNÉES ET LES UTILISATEURS

Créer une base SQLite3 p.78
Insérer des données dans une base SQLite3 p.79
Lire les données d'une base SQLite3 p.80
Appeler une commande shell p.81
Récupérer les arguments de la ligne de commande p.78
Effectuer un calcul en utilisant plusieurs CPUs p.85
Charger une page html p.87
Traiter des données JSON p.88

Envoyer un email p.89
Créer un serveur XMLRPC p.90
Créer un serveur CGI p.92
Télécharger un fichier p.94
Créer une communication entre deux machines p.96
Écrire en couleur dans un terminal p.98
Modifier la position du curseur p.100
Créer un menu en mode texte p.102
Tracer et annoter une courbe p.103
Créer une animation d'attente en mode terminal p.104
Créer une interface graphique p.107

4 Recettes 68-75

UTILISEZ LES BONS OUTILS

Distribuer les fichiers d'un projet dans un seul fichier exécutable p.112
Gérer les modules avec pip p.113
Travailler avec un environnement virtuel p.114
Mettre en place des tests unitaires p.116
Débugger un programme p.118
Obtenir une trace détaillée des erreurs p.120
Afficher des messages de débogage à l'aide d'un logger p.122
Mesurer le temps d'exécution d'une commande p.124

1 EXPLOITEZ LES TYPES DE BASE

À découvrir dans cette partie...

• LES CHAÎNES DE CARACTÈRES

Manipuler des chaînes de caractères est une opération courante. Pourtant certains cas peuvent poser quelques difficultés. Les recettes de cette partie devraient vous permettre de manipuler aisément ce type de données. (Recettes 1 - 4)

• LES NOMBRES

Dans l'utilisation des nombres, le module permettant de manipuler les nombres aléatoires est souvent méconnu. Il est possible d'obtenir les résultats attendus, mais au prix d'un travail parfois inutile. De même, la manipulation de matrices n'est pas complexe, mais peut être effectuée de deux manières différentes, à choisir en fonction des besoins. (Recettes 5 - 6)

• LES DATES ET LE TEMPS

Effectuer des opérations en additionnant ou retranchant des dates est chose possible en Python, de même que la vérification qu'une date donnée est correcte et qu'elle appartient bien à notre calendrier. (Recette 7)

• LES FICHIERS

Devant la multitude de formats de fichiers disponibles, il n'est pas toujours évident de savoir comment lire ou écrire un fichier. Dans cette partie, nous nous attacherons au traitement des fichiers textes et de quelques formats parmi les plus utilisés tels que xml, json, ini, hdf5 et odt. (Recettes 8 - 19)

• LES LISTES ET LES DICTIONNAIRES

Il existe une multitude d'opérations pouvant être réalisées sur les listes et les dictionnaires. De par la taille de ces types de données, il est important de bien savoir les manipuler de manière à ce que les traitements soient réalisés de manière efficace. (Recettes 20 - 28)

NOS RECETTES POUR...

1 Chercher des sous-chaînes à l'aide d'une expression régulière p.10
2 Lire des données au clavier p.11
3 Internationaliser un texte p.12
4 Utiliser un template pour générer du texte p.14
5 Enregistrer et charger des matrices p.18
6 Utiliser des nombres aléatoires p.19
7 Vérifier la validité d'une date p.20
8 Lire un fichier texte ligne à ligne p.21
9 Accéder à un endroit précis d'un fichier p.22
10 Créer un fichier xml p.24
11 Lire un fichier xml p.26
12 Créer un fichier json p.27
13 Lire un fichier json p.28
14 Créer un fichier ini p.29
15 Lire un fichier ini p.30
16 Créer un fichier hdf5 p.31
17 Lire un fichier hdf5 p.33
18 Créer un fichier odt p.34
19 Lire un fichier odt p.36
20 Parcourir une liste p.38
21 Trier une liste d'entiers lue sous forme de chaîne de caractères p.39
22 Trier avec fct de tri personnalisée p.40
23 Supprimer les doublons d'une liste p.42
24 Trier les valeurs d'un dictionnaire p.43
25 Conserver l'ordre des éléments dans un dictionnaire p.44
26 Créer un dictionnaire qui soit la fusion de deux dictionnaires p.46
27 Créer une liste en utilisant la compréhension de listes p.47
28 Créer un dictionnaire en utilisant la compréhension de dictionnaires p.48

RECETTE 1

CHERCHER DES SOUS-CHAÎNES à l'aide d'une expression régulière

L'OBJECTIF

Nous disposons d'une chaîne de caractères comportant des personnages de séries sous la forme `nom_serie:personnage`. Tous les personnages sont séparés par un point-virgule et on souhaite retrouver facilement l'ensemble des personnages d'une série donnée.

LA SOLUTION

```
01: import re
02:
03: series = 'Miss Fisher:Phrany Fisher;GOT:Daenerys Targaryen;X-
Files:Dana Scully;Miss Fisher:Jack Robinson; Miss Fisher:Dorothy
Williams;GOT:Jon Snow;GOT:Tyrion Lannister;X-Files:Fox Mulder'
04:
05: pattern = r':([A-Za-z ]+)[;|$]'
06: match = re.findall(r'GOT' + pattern, series)
07: print(match)
```

COMMENTAIRES

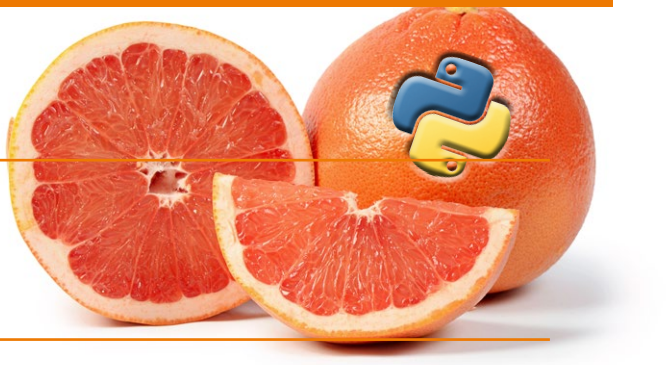
Le module `re` fournit toutes les fonctions permettant de travailler avec des expressions régulières. Dans notre cas, nous n'aurons besoin que de `findall()` pour trouver toutes les sous-chaînes recherchées sous forme d'une liste. Le programme retourne le résultat suivant :

```
['Daenerys Targaryen', 'Jon Snow', 'Tyrion Lannister']
```

La chaîne de caractères définissant le motif de recherche (ligne 5) est préfixée par un `r` signifiant *raw string* : les caractères spéciaux tels que `\n` ne sont pas évalués. Le motif que l'on souhaite récupérer est indiqué entre parenthèses et comme le nom d'un personnage se termine soit par un point-virgule, soit par une fin de chaîne, on emploie `[;|$]` pour indiquer ces deux possibilités.
La structure du programme permet de chercher les personnages de différentes séries en modifiant à la volée l'expression régulière. Ainsi, en ligne 6 nous préfixons l'expression régulière par `GOT`, mais nous pourrions plus tard utiliser `X-Files` ou `Miss Fisher`. Si votre expression régulière ne doit pas varier, alors vous pouvez la compiler. Les lignes 5 et 6 deviendraient alors :

```
...
05: pattern = re.compile(r'GOT:([A-Za-z ]+)[;|$]')
06: match = pattern.findall(series)
...
```

RECETTE 2 — UTILISER UN TEMPLATE pour générer du texte



L'OBJECTIF

Avoir un modèle (*template*) permettant de générer un document en lui transmettant des valeurs.

LA SOLUTION

Il faut installer le module **jinja2** :

```
$ sudo pip3 install jinja2
```

Un *template* ressemblera au fichier `modele.tpl` :

```
01: Série {{ name | upper }}
02:
03: Personnages :{% for perso in personnages %}
04:   - {{ perso }}{% endfor %}
```

Le script utilisant ce *template* :

```
01: import jinja2
02:
03: env = jinja2.Environment(loader=jinja2.FileSystemLoader('.'))
04:
05: template = env.get_template('modele.tpl')
06:
07: data = {
08:     'name': 'Terra Nova',
09:     'personnages': ['Jim Shannon', 'Nathaniel Taylor', '...']
10: }
11: print(template.render(data))
```

À l'exécution, les données seront placées dans le *template* :

```
$ python3 utiliser_template.py
Série TERRA NOVA

Personnages :
  - Jim Shannon
  - Nathaniel Taylor
  - ...
```

COMMENTAIRES

Dans le *template*, les éléments entre doubles accolades seront remplacés par une valeur transmise depuis un dictionnaire et dont la clé portera le même nom que celui indiqué. Pour `{{ name }}`, il faudra transmettre un dictionnaire contenant `{ 'name': 'value', ... }`. Il est possible d'appliquer des filtres sur les données transmises à l'aide d'un pipe. En ligne 1 par exemple, la donnée transmise pour `name` sera convertie en majuscules par le filtre `upper`. Les éléments compris entre `{%` et `%}` déterminent un bloc. Dans les lignes 3 et 4, il s'agit d'une boucle : `personnages` est traité comme une liste, donc la clé `personnages` du dictionnaire qui est transmis au modèle devra être une liste. Pour utiliser un *template*, il suffit de le charger (lignes 3 et 5 du script Python), de définir les données (lignes 7 à 10) et de les insérer dans le modèle (ligne 11). ▮

RECETTE 3

LIRE DES DONNÉES au clavier

L'OBJECTIF

Lire des données saisies au clavier par l'utilisateur :

- un entier ;
- une chaîne de caractères ;
- des réels.

LA SOLUTION

```
01: error = True
02: while error:
03:     try:
04:         entier = int(input('Donnez un entier : '))
05:         error = False
06:     except:
07:         print('Une valeur entière est attendue!')
08: print('Entier : {}'.format(entier))
09:
10: chaine = input('Donnez une phrase : ')
11: print('Chaîne : {}'.format(chaine))
12:
13: error = True
14: while error:
15:     try:
16:         reels = list(map(float, input('Donnez des réels séparés
par une virgule : ').split(',')))
17:         error = False
18:     except:
19:         print('Erreur dans la saisie!')
20: print('Liste de réels : {}'.format(reels))
```

COMMENTAIRES

1 Conversion des données

En Python 3, il n'y a qu'une seule fonction permettant de récupérer une saisie utilisateur : `input()`. Cette fonction renvoie une chaîne de caractères (voir lignes 10 et 11) et il faut donc convertir les données si un autre type est attendu.

Dans les lignes 1 à 8, nous mettons en place un mécanisme qui va boucler tant que l'utilisateur n'aura pas saisi un entier (ou appuyé sur <Ctrl> + <C> bien sûr...). Si la conversion par `int()` de la ligne 4 échoue alors une exception est levée :

- la ligne 5 n'est pas exécutée et la variable `error` reste à `True` ;
- on se branche en ligne 6 et le message « Une valeur entière est attendue ! » est affiché ;
- on retourne en ligne 2.

Le même mécanisme est utilisé dans les lignes 13 à 20 pour lire une liste de réels. Notez ici que la conversion s'effectue en plusieurs temps :

- un appel à `split(',')` pour découper la chaîne suivant les virgules et obtenir une liste de chaînes de caractères ;
- un appel à `map()` pour appliquer la fonction `float()` à l'ensemble des éléments de la liste.

Pour une meilleure compréhension, la ligne 16 pourrait s'écrire :

```
reels = list(map(float, input('Donnez des réels séparés par une
virgule : ').split(',')))
saisie = input('Donnez des réels séparés par une virgule : ')
saisie_liste = saisie.split(',')
liste_reels = map(float, saisie_liste)
reels = list(liste_reels)
```

Le passage de `liste_reels` à `reels` est dû au fait que la fonction `map()` renvoie un générateur qu'il faut donc convertir sous forme de liste à l'aide de `list()`.

Voici un exemple d'exécution du code :

```
Donnez un entier : et
Une valeur entière est attendue!
Donnez un entier : 4
Entier : 4
Donnez une phrase : glmf et lp
Chaîne : glmf et lp
Donnez des réels séparés par une virgule : 3, er ,"r
Erreur dans la saisie!
Donnez des réels séparés par une virgule : 3.4, 5, 23.6
Liste de réels : [3.4, 5.0, 23.6]
```

2 Le input() de Python 2.x

Si vous utilisez encore des programmes en Python 2.x, l'utilisation de `input()` est fortement déconseillée ! En effet, cette fonction effectue en fait un `eval(raw_input())` où `raw_input()` est l'équivalent du `input()` de Python 3 et renvoie donc une chaîne de caractères. Cela constitue une faille permettant d'effectuer une injection de code :

```
> d = input('Saisissez un entier : ')
Saisissez un entier : __import__('os').system('ls')
```

Ce code va exécuter un `ls` au niveau du système. Je vous laisse imaginer le résultat avec un `__import__('os').system('\rm -R *')` ... 

RECETTE 4

INTERNATIONALISER une application

L'OBJECTIF

Une application contenant du texte doit pouvoir être utilisée par des personnes parlant uniquement français, ou allemand, ou anglais, ou ... Nous partirons d'un script `internationaliser.py` affichant simplement trois chaînes de caractères (nous avons besoin d'une phrase au singulier et une autre au pluriel pour mettre en évidence le mécanisme de traduction des chaînes au pluriel) :

```
01: print('Salut à tous !')
02: print('1 erreur')
03: print('2 erreurs')
```

LA SOLUTION

Nous devons structurer notre solution en plusieurs étapes.

1 Architecture des fichiers

Nous créons les répertoires suivants :

```
$ mkdir -p i18n/fr/LC_MESSAGES
$ mkdir -p i18n/en/LC_MESSAGES
$ mkdir -p i18n/de/LC_MESSAGES
$ tree
.
├── i18n
│   ├── de
│   │   └── LC_MESSAGES
│   ├── en
│   │   └── LC_MESSAGES
│   └── fr
│       └── LC_MESSAGES
└── internationaliser.py
```

2 Préparer l'application à l'internationalisation

```
01: import gettext
02: gettext.install('internationaliser', localedir='i18n')
03:
04: print(_('salut'))
05: print('1 {}'.format(_('erreur')))
06: print('2 {}'.format(_('erreurs')))
```

3 Générer le fichier modèle de traduction

Le fichier modèle de traduction sera `i18n/internationaliser.pot` :

```
$ xgettext --language=Python --keyword=_ --output=./i18n/
internationaliser.pot ./internationaliser.py
```

Complétez ensuite les informations nécessaires :

```
01: # Internationalisation d'une application
02: # Copyleft GNU GPL v3
03: # This file is distributed under the same license as the
'internationaliser' package.
04: # Tristan Colombo <tristan@gnulinuxmag.com>, 2016.
05: #
06: #, fuzzy
07: msgid ""
08: msgstr ""
09: "Project-Id-Version: 1.0\n"
10: "Report-Msgid-Bugs-To: tristan@gnulinuxmag.com\n"
11: "POT-Creation-Date: 2016-04-22 10:05+0200\n"
12: "PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
13: "Last-Translator: Tristan Colombo <tristan@gnulinuxmag.com>\n"
14: "Language-Team: French\n"
15: "Language: \n"
16: "MIME-Version: 1.0\n"
17: "Content-Type: text/plain; charset=UTF-8\n"
18: "Content-Transfer-Encoding: 8bit\n"
...
```

4 Création des fichiers de traduction

On utilise le modèle `i18n/internationaliser.pot` pour créer les fichiers de traduction de chaque langue supportée. D'abord en français :

```
$ msginit --input=./i18n/internationaliser.pot --output=./i18n/fr/
LC_MESSAGES/internationaliser.po
Le nouveau catalogue de messages devrait contenir votre adresse
de courriel afin que les utilisateurs puissent vous envoyer leurs
commentaires ...
...
Which is your email address?
1 tristan@gnulinuxmag.com
...
Please choose the number, or enter your email address.
1
Récupération de http://translationproject.org/team/index.html...
terminé.
A translation team for your language (fr) does not exist yet.
...
```

Puis il faut lancer les mêmes commandes pour les autres langues :

```
$ msginit --input=./i18n/internationaliser.pot --output=./i18n/en/
LC_MESSAGES/internationaliser.po
...
$ msginit --input=./i18n/internationaliser.pot --output=./i18n/de/
LC_MESSAGES/internationaliser.po
...
```

5 Traduire les chaînes de caractères

Il faut éditer chaque fichier po et donner la traduction de chaque chaîne de caractères. Par exemple, voici le fichier `i18n/en/LC_MESSAGES/internationaliser.po` :

```
...
20: #: internationaliser.py:4
21: msgid "Salut à tous !"
22: msgstr "Hi everybody !"
23:
24: #: internationaliser.py:5
25: msgid "1 erreur"
26: msgstr "1 error"
27:
28: #: internationaliser.py:6
29: msgid "2 erreurs"
30: msgstr "2 errors"
```

6 Compiler les fichiers de traduction

Chaque fichier de traduction doit être compilé sous la forme d'un fichier `.mo` :

```
$ msgfmt ./i18n/en/LC_MESSAGES/internationaliser.po --output-file
./i18n/en/LC_MESSAGES/internationaliser.mo
```

7 Lancer l'application

L'application utilisera le contenu de la variable `LC_ALL` pour déterminer la langue à employer. Au lancement de l'application, vous pouvez modifier temporairement la valeur de `LC_ALL` pour afficher les chaînes de caractères de l'application dans la langue souhaitée :

```
$ LC_ALL=en python3 internationaliser.py
Hi everybody !
1 error
2 errors
```

NOTE

Mise à jour des chaînes à traduire

Si vous ajoutez de nouvelles chaînes de caractères dans votre application, il vous faudra mettre à jour les différents fichiers... sans effacer leur contenu ! Pour cela :

1. Regénérez le fichier modèle :

```
$ xgettext --language=Python --keyword=_ --output=./i18n/
internationaliser.pot ./internationaliser.py
```

2. Mettez à jour les fichiers de traduction (ici exemple avec le fichier de langue anglaise) :

```
$ msgmerge --update --no-fuzzy-matching --backup=off ./i18n/en/
LC_MESSAGES/internationaliser.po ./i18n/internationaliser.pot
```

3. Traduisez les nouvelles chaînes de caractères.

4. Compilez les fichiers de traduction :

```
$ msgfmt ./i18n/en/LC_MESSAGES/internationaliser.po --output-file
./i18n/en/LC_MESSAGES/internationaliser.mo
...
```

COMMENTAIRES

1 Préparer l'application à l'internationalisation

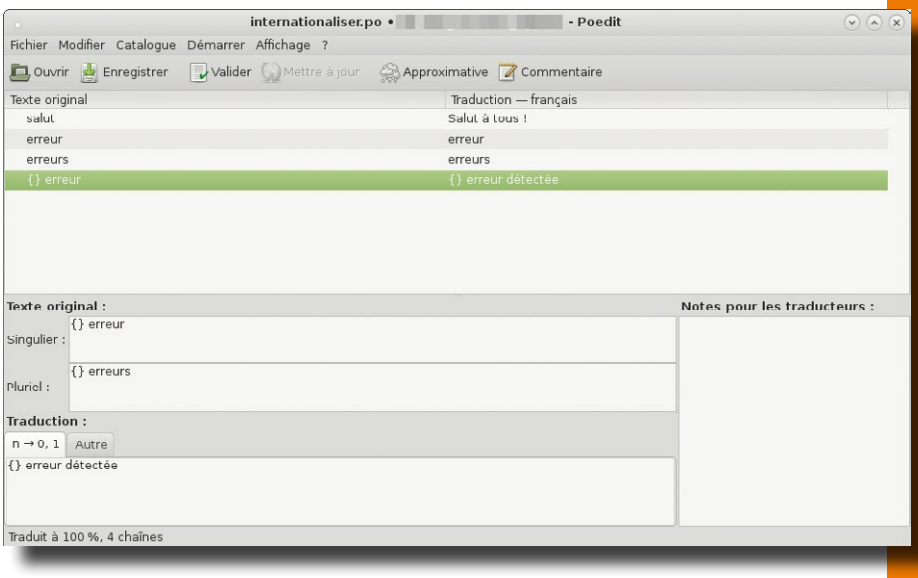
Notez que la commande `xgettext` n'a pas été conçue pour fonctionner avec des polices accentuées : les identifiants de chaînes de caractères utilisées dans `_()` ne doivent donc pas comporter d'accents (écrivez vos chaînes en anglais).

2 Logiciels de traduction

Notez qu'il existe des logiciels de traduction permettant de travailler sur les fichiers `.po` dans une interface graphique (les traducteurs ne sont pas forcément développeurs !).

Par exemple, le programme **poedit** (voir figure suivante), disponible dans les dépôts des distributions basées sur Debian remplit très bien cette tâche. Il s'installe par :

```
$ sudo apt install poedit
```



3 À propos des chaînes de caractères au pluriel

La fonction `ngettext()` permet de proposer deux chaînes de caractères, l'une au singulier et l'autre au pluriel. En fonction de la valeur d'un troisième paramètre, le programme affichera la forme au singulier ou l'une des formes au pluriel (certaines langues ont plusieurs pluriels - généralement 1 indiquera le singulier et 2 le pluriel).

Voici comment employer `ngettext()` :

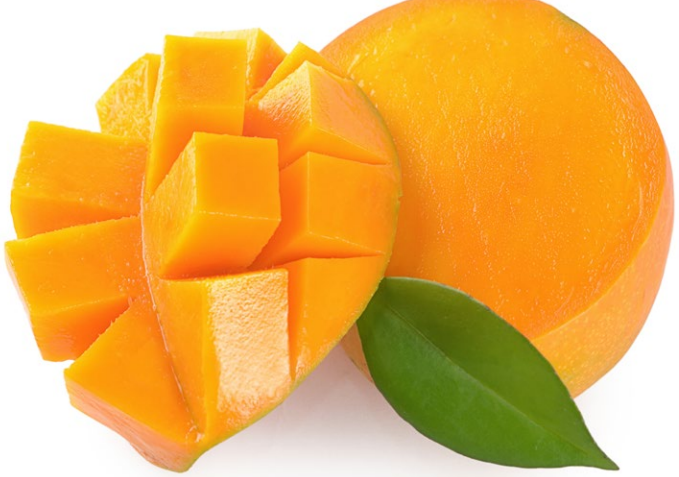
```
01: import gettext
02: gettext.install('internationaliser', localedir='i18n',
names=('ngettext',))
03:
04: print(_('salut'))
05: print('1 {}'.format(_('erreur')))
06: print('2 {}'.format(_('erreurs')))
07:
08: print(ngettext('{} erreur', '{} erreurs', 1).format(1))
09: print(ngettext('{} erreur', '{} erreurs', 2).format(5))
```

Il faut ensuite générer le modèle et mettre à jour les fichiers de traduction puis ajouter les traductions. Voici un exemple pour le fichier `i18n/fr/LC_MESSAGES/internationaliser.po` :

```
...
33: #: internationaliser.py:8 internationaliser.py:9
34: msgid "{} erreur"
35: msgid_plural "{} erreurs"
36: msgstr[0] "{} erreur détectée"
37: msgstr[1] "{} erreurs détectées"
```

Après compilation, les chaînes de caractères seront affichées au singulier ou au pluriel. ■

RECETTE 5



ENREGISTRER ET CHARGER des matrices

L'OBJECTIF

Travailler sur des objets de type `matrix` et pouvoir les enregistrer.

LA SOLUTION

Il faut avoir installé **scipy** (pré-installé dans **Python(x,y)** sous Windows).

```
01: from scipy.io import loadmat, savemat
02: import scipy
03:
04: A = scipy.matrix([[1, 0, 2], [0, 3, 0]])
05: B = scipy.matrix([[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]])
06: savemat('ma_matrice.mtx', {'A': A, 'B': B})
07:
08: m = loadmat('ma_matrice.mtx')
09: print(m['A'])
10: print(m['B'])
```

COMMENTAIRES

Cette méthode de sauvegarde et de lecture des matrices est compatible avec les fichiers **Matlab** versions 6, et 7 à 7.2.

Il est possible d'utiliser un format Python :

```
01: from scipy.io import mmread, mmwrite
02: import scipy
03:
04: A = scipy.matrix([[1, 0, 2], [0, 3, 0]])
05: mmwrite('ma_matrice.mtx', A)
06:
07: B = mmread('ma_matrice.mtx')
08: print(B)
```

Avec ce format, on ne peut sauvegarder et lire qu'une seule matrice à la fois alors qu'avec le format Matlab on travaille sur un dictionnaire de matrices. ■

RECETTE 6

UTILISER DES NOMBRES aléatoires

L'OBJECTIF

Générer aléatoirement des entiers entre 0 et 10 et des réels entre 5 et 25. Pendant la phase de développement, les nombres générés devront toujours être les mêmes (pour pouvoir détecter plus facilement les erreurs). À la mise en production, les séquences devront être différentes à chaque exécution.

LA SOLUTION

```
01: import random
02:
03: DEBUG = True
04:
05: if DEBUG:
06:     random.seed(1)
07:
08: for i in range(5):
09:     print(random.randint(0, 10))
10:
11: for i in range(5):
12:     print(random.uniform(5, 25))
```

COMMENTAIRES

C'est le module `random` qui fournit les fonctions permettant de travailler avec le générateur aléatoire :

- `seed()` initialise la « graine » du générateur : avec une même graine, vous obtiendrez toujours la même suite de nombres ! Ce mécanisme est utilisé dans les lignes 3 à 6 : si `DEBUG` passe à `False`, alors on arrête l'initialisation avec la valeur `1` et on repasse à l'initialisation par défaut qui est l'heure système (au moment du chargement du module). En utilisant `seed()` sans paramètre, on demande une initialisation avec l'heure du système ;
- `randint(a, b)` retourne un entier compris entre `a` et `b` ;
- `uniform(a, b)` retourne un réel compris entre `a` et `b`. ▮

RECETTE 7

VÉRIFIER LA VALIDITÉ d'une date

L'OBJECTIF

Une date a été fournie par un utilisateur et l'on souhaite s'assurer qu'il s'agit bien d'une date présente dans le calendrier.

LA SOLUTION

```
01: import datetime
02:
03: date = input('Indiquez un jour au format (jj/mm/aaaa): ')
04: try:
05:     (day, month, year) = map(int, date.split('/'))
06:     datetime = datetime.datetime(year, month, day)
07: except ValueError:
08:     print('La date n\'est pas valide !')
```

COMMENTAIRES

Une fois les données acquises (ici par un `input()` en ligne 3 puis un découpage suivant le caractère `/` et une conversion de l'ensemble des éléments de la liste obtenue sous forme d'entiers), nous essayons de créer un objet `datetime`. Si le format de la date est invalide (nombre du mois supérieur à `12` par exemple), une exception sera déclenchée. De même, si la date saisie n'existe pas dans le calendrier (le 29 février 2015 par exemple), une exception sera déclenchée.

Le module `datetime` permet de réaliser de nombreuses autres opérations sur les dates. On peut ainsi effectuer des calculs ! Partons ainsi du jour courant fourni par `date.today()` et ajoutons ou retranchons des jours :

```
> ce_jour = datetime.date.today()
> jours = [datetime.timedelta(days=jour) for jour in range(11)]
> jours
[datetime.timedelta(0), datetime.timedelta(1), datetime.timedelta(2),
datetime.timedelta(3), datetime.timedelta(4), datetime.timedelta(5),
datetime.timedelta(6), datetime.timedelta(7), datetime.timedelta(8),
datetime.timedelta(9), datetime.timedelta(10)]
> ce_jour + jours[1]
datetime.date(2016, 8, 25)
> ce_jour + jours[10]
datetime.date(2016, 9, 3)
> ce_jour - jours[3]
datetime.date(2016, 8, 21)
```

`timedelta()` permet de spécifier un intervalle de temps qui sera utilisé lors d'une opération sur les dates. Cette fonction permet de spécifier un intervalle précis composé de `days`, `hours`, `minutes`, `seconds`, `milliseconds` et même `microseconds`. ▮

ACCÉDER À UN ENDROIT PRÉCIS d'un fichier

L'OBJECTIF

On veut lire les caractères 7 à 10 (4 caractères) et 21 à 25 (5 caractères) d'un fichier. Exemple avec le fichier `fichier.txt` :

```
Hello GLMF !
Fichier texte
```

LA SOLUTION

```
01: try:
02:     with open('fichier.txt', 'r') as fic:
03:         fic.seek(6)
04:         print(fic.read(4))
05:         fic.seek(21)
06:         print(fic.read(5))
07: except FileNotFoundError as e:
08:     print('Le fichier {} n\'existe pas !'.format(e.filename))
09:     exit(1)
10: # gestions des autres exceptions
```

COMMENTAIRES

Le premier caractère d'un fichier se trouve en position **1**, mais pour que la tête de lecture puisse le lire, celle-ci doit être positionnée en **0** (de manière à « passer dessus »). C'est pour cela que pour lire à partir du caractère 7 il faut positionner la tête de lecture en 6 (ligne 3). Toutefois, en ligne 5 nous positionnons la tête en **21** pour lire le caractère **21**... En fait ce caractère est à la position **22**, car nous n'avions pas compté le caractère de saut à la ligne se trouvant après le point d'exclamation. ▮

Attention : Le caractère de saut à la ligne se trouvant à la fin de chaque ligne d'un fichier est invisible, mais il faut penser à le compter comme un caractère !

LIRE UN FICHIER TEXTE ligne à ligne

L'OBJECTIF

Ouvrir un fichier texte en lecture en gérant les erreurs possibles et afficher son contenu en le lisant ligne à ligne.

LA SOLUTION

```
01: try:
02:     with open('fichier.txt', 'r') as fic:
03:         for line in fic:
04:             print(line, end='')
05: except FileNotFoundError as e:
06:     print('Le fichier {} n\'existe pas !'.format(e.filename))
07:     exit(1)
08: except PermissionError as e:
09:     print('Droit de lecture absent sur le fichier {}
!'.format(e.filename))
10:     exit(2)
11: except Exception as e:
12:     print('Une erreur a empêché l\'ouverture du fichier :
{}'.format(e.strerror))
13:     exit(3)
```

COMMENTAIRES

1 La structure with

L'emploi de la structure `with` permet de ne définir le descripteur de fichier `fic` que dans le bloc qui suit. Ainsi, en quittant le bloc à la fin de la ligne 4, avant que la variable ne soit détruite le descripteur de fichier est fermé par un appel implicite à `fic.close()`. On s'assure ainsi de toujours quitter proprement le fichier… en évitant d'écrire une ligne supplémentaire.

2 La fonction open()

Il existe deux fonctions `open()` en Python : la fonction intégrée que nous venons d'utiliser et celle proposée par le module `os` pour les traitements de bas niveau. Comme ces fonctions portent le même nom, prenez garde de ne pas utiliser la commande `from os import *`. En effet, cette dernière écraserait la définition de la fonction

`open()` de haut niveau et remplacerait celle-ci par le `open()` de bas niveau. Vous obtiendrez alors des messages d'erreur que vous ne comprendriez pas, les deux fonctions n'acceptant pas les mêmes paramètres.

3 Le traitement des exceptions

Nous traitons trois types d'exceptions :

- `FileNotFoundError` qui indique que le fichier n'a pas pu être trouvé ;
- `PermissionError` qui, dans notre cas, est levée si le fichier n'est pas accessible en lecture ;
- `Exception`, la classe mère des exceptions, qui permet de traiter tous les autres cas d'erreur sans traitement spécifique. Attention : ce traitement doit toujours être positionné en dernier sous peine de masquer le traitement des autres exceptions.

4 La fonction exit()

La fonction `exit()` permet de sortir du programme en retournant un code d'erreur compris entre `1` et `255` ou `0` si tout s'est déroulé correctement. Ce code peut être lu en accédant à la valeur de `$?` dans le shell sous GNU/Linux ou Mac OS X ou en accédant à la valeur de `%Errorlevel%` dans un l'invite de commandes sous Windows.

5 Fin de fichier

La marque de fin de fichier en Python est la chaîne de caractères vide `''`. Ainsi, si vous souhaitez tester manuellement si vous êtes parvenu à la fin d'un fichier, il vous faudra tester si la chaîne lue est vide.

Attention : ne confondez pas chaîne vide et ligne vide : une ligne vide dans un fichier contient un retour à la ligne `'\n'`, elle est donc différente de la chaîne vide `''`.

NOTE

Il est parfois utile de récupérer les informations caractère par caractère. Le code sera alors sensiblement le même que précédemment :

```
...
    with open('fichier.txt', 'r') as fic:
        while True :
            car = fic.read(1)
            if car is '':
                print('Fin du fichier atteinte')
                break
...
```

Le cas de la lecture caractère par caractère se présente souvent lorsque l'on souhaite comparer au moins deux fichiers. Voici comment ouvrir les descripteurs de fichier :

```
with open('fichier_1.txt', 'r') as fic_1, ...,
open('fichier_n', 'r') as fic_n:
    ...
```

RECETTE 10

CRÉER UN FICHIER xml

L'OBJECTIF

Créer un fichier xml correctement structuré. Notre fichier sera de la forme :

```
<series>
  <nom titre="Docteur_Who" lang="fr">
    <personnage>Docteur Who</personnage>
    <personnage>Rose Tyler</personnage>
    <personnage>Mickey Smith</personnage>
  </nom>
  <nom titre="Game_Of_Throne" lang="en">
    <personnage etat="mort">Eddard Stark</personnage>
    <personnage etat="mort">Joffrey Baratheon</personnage>
    ...
    <personnage etat="mort">Margaery Tyrell</personnage>
  </nom>
</series>
```

LA SOLUTION

```
01: from lxml import etree
02:
03: root = etree.Element('series')
04:
05: serie = etree.SubElement(root, 'nom')
06: serie.set('lang', 'fr')
07: serie.set('titre', 'Docteur_who')
08:
09: personnages = ['Docteur Who', 'Rose Tyler', 'Mickey Smith']
10:
11: for perso in personnages:
12:     personnage = etree.SubElement(serie, 'personnage')
13:     personnage.text = perso
14:
15: try:
16:     with open('series.xml', 'w') as fic:
17:         fic.write(etree.tostring(root, pretty_print=True).
decode('utf-8'))
18: except IOError:
19:     print('Problème rencontré lors de l\'écriture...')
20:     exit(1)
```

COMMENTAIRES

1 Le module lxml

Pour employer le module `lxml`, vous devez effectuer plusieurs installations :

```
$ sudo aptitude install libxml2-dev libxslt1-dev
```

Puis :

```
$ sudo pip3 install lxml
```

Pour Windows, vous devrez télécharger et installer le fichier `libxml2-2.7.8.win32.zip` disponible sur **ftp://ftp.zlatkovic.com/libxml/**.

Pour Mac OS X, vous pourrez consulter les instructions d'installation sur **http://lxml.de/build.html#building-lxml-on-macos-x**.

Pourquoi utiliser ce module alors que le module `xml.etree.ElementTree` permettant de manipuler le xml est nativement présent dans Python ? Le module `lxml` est basé sur les bibliothèques `libxml2` et `libxslt` écrites en C donc réputées plus rapides.

2 La structure

On commence par créer l'élément racine en ligne 3. Dans les lignes 5 à 7, nous créons un élément (le nom) contenant un attribut `lang` et un attribut `titre`. Dans les lignes 11 à 13, `etree.SubElement()` permet d'ajouter des éléments dans `serie`. Ensuite, pour exporter le fichier xml il n'y a plus qu'à en obtenir la représentation sous forme de chaîne de caractères et à placer le résultat dans un fichier (ligne 17). Remarquez que la fonction `etree.tostring()` renvoie des `bytes` et qu'il faut donc effectuer une conversion par la méthode `decode('utf-8')` pour obtenir une `str`.

NOTE

N'oubliez pas qu'un document XML est bien formé et valide à condition qu'il vérifie les règles exprimées dans son DTD (*Document Type Definition*) également appelé *doctype*. Notre fichier précédent pourrait déclarer utiliser un DTD personnalisé à l'aide des lignes suivantes placées dans l'en-tête du document :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE series SYSTEM "series.dtd">
...
```

Le fichier `series.dtd` est alors un ensemble de règles définissant la structure du document. On aurait pu écrire :

```
<!DOCTYPE series
[
<!ELEMENT series (nom)>
<!ELEMENT nom (personnage)*>
    <!ATTLIST nom
        titre CDATA #REQUIRED
        lang (en | fr) #REQUIRED>
<!ELEMENT personnage (#PCDATA)>
    <!ATTLIST personnage
        etat (mort | vivant | blessé)>
]>
```

RECETTE 11

LIRE UN FICHIER xml

L'OBJECTIF

Ouvrir un fichier xml et en lire le contenu.
Nous prendrons pour exemple le fichier :

```
<series>
  <nom titre="Docteur_Who" lang="fr">
    <personnage>Docteur Who</personnage>
    <personnage>Rose Tyler</personnage>
    <personnage>Mickey Smith</personnage>
  </nom>
  ...
</series>
```

LA SOLUTION

```
01: from lxml import etree
02:
03: tree = etree.parse('series.xml')
04:
05: for node in tree.xpath('//series/nom'):
06:     print('Série {} en {}:'.format(node.get('titre'), node.
get('lang')))
07:     for perso in node.xpath('personnage'):
08:         print('  {}'.format(perso.text))
```

COMMENTAIRES

On utilise les XPath pour naviguer dans le document XLM. Un XPath est un chemin indiquant le déplacement au sein de l'arborescence XML. Ainsi `//series/nom` fait référence au nœud `nom` dont le parent est `series` (la racine).

Le contenu d'un nœud est accessible via l'attribut `text` (voir ligne 8) et pour obtenir la valeur d'un attribut (au sens xml), il faut utiliser la méthode `get()` (voir ligne 6 pour les attributs `titre` et `lang`).

NOTE

D'autres actions sont possibles comme :

- récupérer le doctype du document : `doctype = tree.docinfo.doctype`
- récupérer le nœud racine : `root = tree.getroot()`
- accéder à la liste des attributs ou des valeurs d'un tag :

```
for node in tree.xpath("//personnage"):
    print(node.items()) # Liste des attributs
    print(node.text) # Liste des valeurs
```

RECETTE 12 CRÉER UN FICHIER json

L'OBJECTIF

Un fichier json a une structure de dictionnaire. La création depuis Python est donc très simple si l'on s'appuie sur cette structure. Nous allons créer le fichier json suivant :

```
{
  "series": {
    "Docteur Who": {
      "personnages" : [
        "Docteur Who",
        "Rose Tyler",
        "Mickey Smith"
      ]
    }
  }
}
```

LA SOLUTION

```
01: import json
02:
03: data = {
04:   'series': {
05:     'Docteur Who': {
06:       'personnages': [
07:         'Docteur Who',
08:         'Rose Tyler',
09:         'Mickey Smith'
10:       ]
11:     }
12:   }
13: }
14:
15: try:
16:     with open('series.json', 'w') as fic:
17:         fic.write(json.dumps(data, indent=4))
18: except:
19:     print('Erreur lors de la création du fichier')
20:     exit(1)
```

COMMENTAIRES

La fonction `dumps()` du module `json` va sérialiser le dictionnaire qui lui est fourni et il ne restera alors plus qu'à écrire la chaîne de caractères retournée dans un fichier. Le paramètre `indent` permet de définir l'indentation à employer pour obtenir un rendu plus lisible. ▮

RECETTE 13

LIRE UN FICHIER json

L'OBJECTIF

Lire un fichier json de manière à stocker toutes les données dans un dictionnaire.

LA SOLUTION

```
01: try:
02:     with open('series.json', 'r') as fic:
03:         series = json.load(fic)
04: except:
05:     print('Erreur lors de la lecture du fichier')
06:     exit(1)
07:
08: print(series)
```

COMMENTAIRES

La fonction `load()` (sans « s » à la fin) du module `json` permet de charger les données d'un fichier json dans un dictionnaire (ici `series`).

NOTE

La fonction `loads()` (avec un « s » cette fois-ci) du module `json` permet de désérialiser la chaîne de caractères qui lui est transmise en paramètre (voir ligne 11 suivante). Nous nous retrouvons là encore avec un dictionnaire classique dans lequel nous pouvons naviguer à l'aide des différentes clés.

```
01: import json
02:
03: try:
04:     with open('series.json', 'r') as fic:
05:         data = fic.read()
06: except:
07:     print('Erreur lors de la lecture du fichier')
08:     exit(1)
09:
10:
11: series = json.loads(data)
12: print(series['series']['Docteur Who'])
```

RECETTE 14 CRÉER UN FICHIER ini

L'OBJECTIF

Créer un fichier de configuration ini ayant la structure suivante :

```
[personnages]
amicalement_votre = Brett Sinclair, Danny Wilde
chapeau_melon = John Steed, Emma Peel

[saisons]
amicalement_votre = 1
chapeau_melon = 6
```

LA SOLUTION

```
01: import configparser
02:
03: config = configparser.ConfigParser()
04: config['personnages'] = {'chapeau_melon': 'John Steed, Emma
Peel', 'amicalement_votre': 'Brett Sinclair, Danny Wilde'}
05: config['saisons'] = {'chapeau_Melon': 6, 'amicalement_votre': 1}
06:
07: with open('series.ini', 'w') as fic:
08:     config.write(fic)
```

COMMENTAIRES

Une fois l'objet `ConfigParser` (ligne 3) créé, il suffit d'ajouter des éléments comme s'il s'agissait d'un simple dictionnaire (lignes 4 et 5).

L'écriture du fichier se fait à l'aide de la méthode `write()` à laquelle on transmet un descripteur de fichier (ligne 8).

NOTE

Les fichiers ini sont particulièrement utilisés en tant que fichiers de configuration. Ils sont divisés en sections (signalées à l'aide de crochets) et chaque section comporte une liste de définitions de paramètres sous la forme `nom_parametre = valeur`.

On peut y insérer des commentaires en faisant précéder un texte du caractère « point-virgule » : tout ce qui suivra jusqu'à la fin de la ligne sera considéré comme commentaire. Par exemple :

```
chapeau_melon = 6 ; nombre de saisons de chapeau melon et
bottes de cuir
```

Comme nous l'avons fait dans notre exemple ce format de fichier peut également être utilisé pour stocker simplement des données faiblement structurées.

RECETTE 15

LIRE UN FICHIER ini

L'OBJECTIF

Lire les données issues d'un fichier `series.ini`.

LA SOLUTION

```
01: import configparser
02:
03: config = configparser.ConfigParser()
04: config.read('series.ini')
05:
06: print('Les données sont:')
07: for section in config.sections():
08:     print('Section {}'.format(section))
09:     for key in config[section]:
10:         print('   {} => {}'.format(key, config[section][key]))
```

COMMENTAIRES

La lecture du fichier ini se fait directement à l'aide de la méthode `read()` sur un objet `ConfigParser` (ligne 4). Les données du fichier se trouvent dans l'instance `config` sous la forme d'un dictionnaire où la première clé est le nom de la section et la seconde clé le nom de l'option (`config.sections()` renvoie la liste de toutes les sections disponibles dans le fichier lu).

Si nous prenons le fichier `series.ini` utilisé dans la recette 13, nous obtenons :

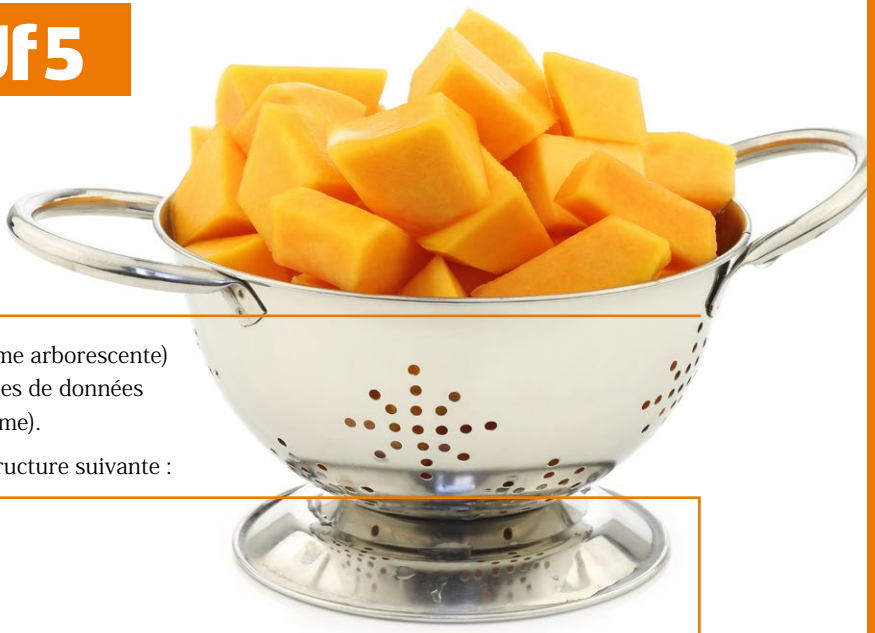
```
Les données sont:
Section personnages
  amicalement_votre => Brett Sinclair, Danny Wilde
  chapeau_melon => John Steed, Emma Peel
Section saisons
  amicalement_votre => 1
  chapeau_melon => 6
```

NOTE

Si l'une des valeurs de vos paramètres est censée être un booléen, vous pouvez utiliser la méthode `getboolean()` qui lèvera une exception `ValueError` en cas d'erreur. Il est également possible de paramétrer le comportement de cette fonction à l'aide de l'attribut `BOOLEAN_STATES`. En effet, par défaut, `getboolean()` considère les valeurs `'1'`, `'yes'`, `'true'`, `'on'` comme étant vraies (`True`) et `'0'`, `'no'`, `'false'`, `'off'` comme étant fausses (`False`). Voici un exemple :

```
config.BOOLEAN_STATES = {'vrai' : True, 'faux' : False,
'disabled' : False, '0' : False, '6' : True}
if not config['saisons'].getboolean('chapeau_melon'):
    print('Vide !')
```

RECETTE 16 CRÉER UN FICHIER hdf5



L'OBJECTIF

Un fichier hdf5 est un fichier structuré (forme arborescente) qui permet de stocker de très grands volumes de données (la seule limitation de taille est liée au système).

Nous allons créer le fichier hdf5 ayant la structure suivante :

```
series
> friends
  >> personnages
  >> saison
    [nombre] = 10
    [debut] = 1994
    [fin] = 2004
  >> data
    [data_friends] = [0, ..., 99]
> les mystères de l'ouest
  >> personnages
  >> saison
    [nombre] = 1
    [debut] = 1965
    [fin] = 1969
  >> data
    [data_mysteres] = [2, ..., 198]
```

LA SOLUTION

Il faut installer le module **h5py** (paquet **python3-h5py** sous Debian) :

```
$ sudo pip3 install h5py
```

Le code est le suivant :

```
01: import h5py
02: import numpy as np
03:
04: try:
05:     with h5py.File('data.hdf5', 'w') as fic:
06:         racine = fic.create_group('series')
07:         racine_friends = racine.create_group('friends')
08:         racine_friends_persos = racine_friends.create_
group('personnages')
09:         racine_friends_saisons = racine_friends.create_group('saison')
10:         racine_friends_data = racine_friends.create_group('data')
11:
12:         racine_friends_saisons.attrs['nombre'] = '10'
13:         racine_friends_saisons.attrs['debut'] = '1994'
```

```
14:         racine_friends_saisons.attrs['fin'] = '2004'
15:         dset_friends = racine_friends_data.create_
dataset('data_friends', (100,), dtype = 'i')
16:         dset_friends[...] = np.arange(100)
17:
18:         racine_mysteres = racine.create_group('les mystères
de l\'ouest')
19:         racine_mysteres_persos = racine_mysteres.create_
group('personnages')
20:         racine_mysteres_saisons = racine_mysteres.create_
group('saison')
21:         racine_mysteres_data = racine_mysteres.create_
group('data')
22:
23:         racine_mysteres_saisons.attrs['nombre'] = '1'
24:         racine_mysteres_saisons.attrs['debut'] = '1965'
25:         racine_mysteres_saisons.attrs['fin'] = '1969'
26:         dset_mysteres = racine_mysteres_data.create_
dataset('data_mysteres', (100,), dtype = 'i')
27:         dset_mysteres[...] = 2 * np.arange(100)
28: # Gestion des exceptions
29: except IOError:
...
```

COMMENTAIRES

L'arborescence est créée à l'aide de groupes (méthode `create_group()`). Les données peuvent être stockées de deux manières différentes :

- soit sous forme d'attributs attachés à un groupe (liste `attrs` comme dans les lignes 12 à 14 et 23 à 25) ;
- soit sous forme de *dataset* (voir encadré).

Notez que l'ellipsis (les points de suspension) de `dset_friends[...]` désigne toutes colonnes.

NOTE

Les datasets

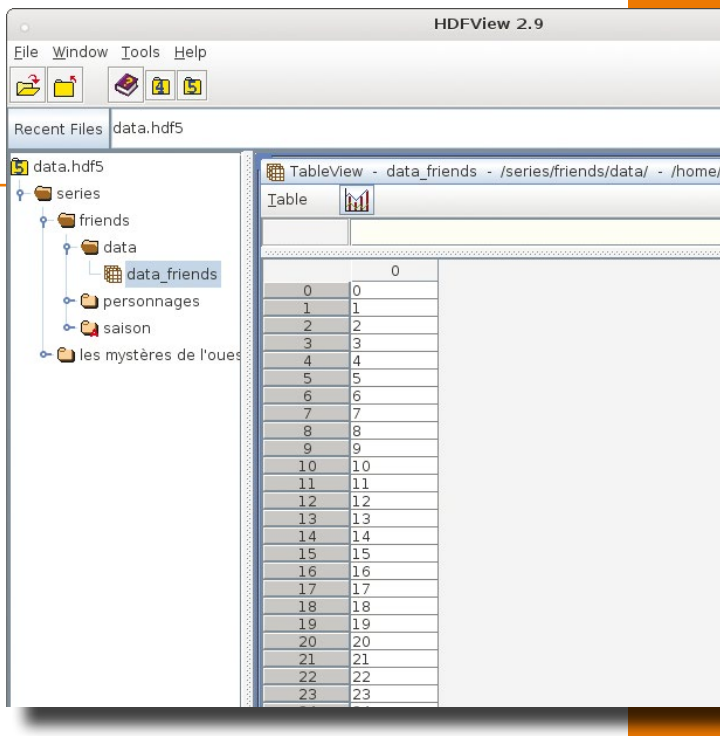
Un dataset est défini par la méthode `create_dataset()` appliquée sur un descripteur de fichier hdf5 avec en paramètres un nom, un *dataspace* et un *datatype* :

```
descripteur.create_dataset(nom, (nb_
lignes, nb_colonnes), dtype=format)
```

Le format des données (valable pour toutes les cellules du tableau) doit être indiqué en utilisant l'écriture des `dtype` de `numpy` (`i` pour des entiers, `f` pour des réels, etc.). Pour créer un format composé, il faut associer à chaque cellule un *dtype* composé (par exemple `f, i`) : chaque cellule du tableau contiendra alors plusieurs éléments.

La taille du tableau de données, ou *dataspace*, est déterminée par un tuple `(nb_lignes, nb_colonnes)`.

LIRE UN FICHIER hdf5



L'OBJECTIF

Lire un fichier hdf5 ayant la structure suivante :

```
series
> friends
  >> personnages
  >> saison
    [nombre] = 10
    [debut] = 1994
    [fin] = 2004
  >> data
    [data_friends] = [0, ..., 99]
> les mystères de l'ouest
  >> personnages
  >> saison
    [nombre] = 1
    [debut] = 1965
    [fin] = 1969
  >> data
    [data_mysteres] = [2, ..., 198]
```

LA SOLUTION

Il faut installer le module **h5py** (paquet **python3-h5py** sous Debian) :

```
$ sudo pip3 install h5py
```

Le code est le suivant :

```
01: import h5py
02:
03: try:
04:     with h5py.File('data.hdf5', 'r') as fic:
05:         print('Friends')
06:         print("Nombre de saisons : {}".format(fic['/series/
friends/saison'].attrs['nombre']))
07:         tab = fic['/series/friends/data']['data_friends'][...]
08:         print("Type du tableau : {}".format(type(tab)))
09:         print(tab)
10: # Gestion des exceptions
11: except IOError:
...
```

COMMENTAIRES

Il est très simple d'accéder aux données en indiquant le chemin à suivre pour y accéder. Ainsi, pour lire le contenu de l'attribut **nombre** se trouvant dans **/series/friends/saison**, il faut utiliser le descripteur de fichier de la manière suivante :

```
fic['/series/friends/saison'].attrs['nombre']
```

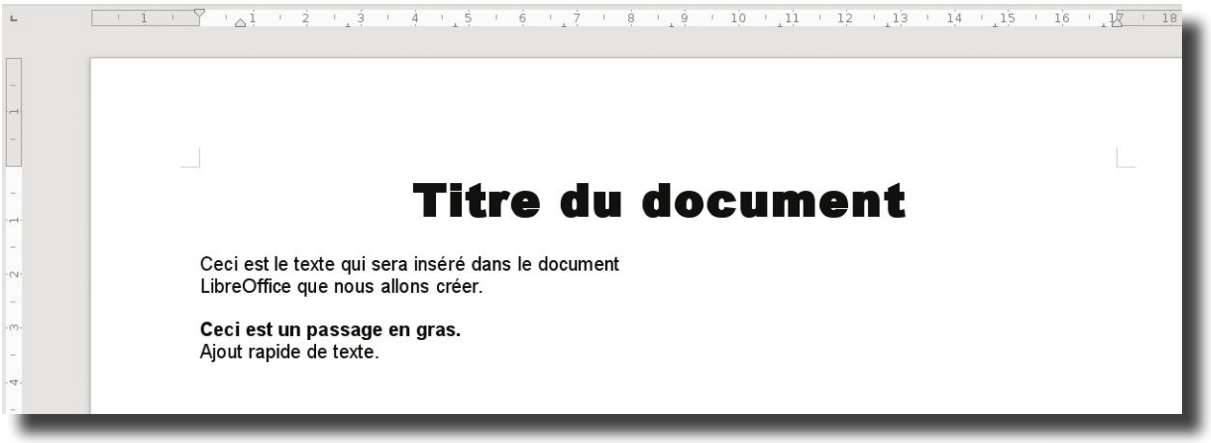
Notez qu'il existe des logiciels tels que **hdfview** permettant de lire des fichiers hdf5 et de naviguer dans leur arborescence de manière graphique. ■

RECETTE 18

CRÉER UN FICHIER LibreOffice Writer

L'OBJECTIF

Créer un petit fichier .odt ayant l'apparence suivante :



LA SOLUTION

Il faut installer le module **odfpy** :

```
$ sudo pip3 install odfpy
```

Le code est ensuite le suivant :

```
01: from odf.opendocument import OpenDocumentText
02: from odf.style import Style, TextProperties, ParagraphProperties
03: from odf.text import H, P, Span
04: from odf import teletype
05:
06: document = OpenDocumentText()
07:
08: # Styles
09: ## Titre principal
10: h1Style = Style(name='Titre principal', family='paragraph')
11: h1Style.addElement(ParagraphProperties(attributes={'textalign':
'center'}))
```

```
12: h1Style.addElement(TextProperties(attributes={'fontfamily':
'Arial Black', 'fontsize': '24pt', 'fontweight': 'bold'}))
13:
14: ## Gras
15: boldStyle = Style(name='Gras', family='text')
16: boldStyle.addElement(TextProperties(fontweight='bold'))
17:
18: ## Normal
19: stdStyle = Style(name='Normal', family='paragraph')
20: stdStyle.addElement(TextProperties(attributes={'fontfamily':
'Arial', 'fontsize': '10pt'}))
21:
22: # Ajout des styles
23: s = document.styles
24: s.addElement(h1Style)
25: s.addElement(boldStyle)
26: s.addElement(stdStyle)
27:
28:
29: # Document
30: title = H(outlinelevel=1, stylename=h1Style, text='Titre du
document')
31: document.text.addElement(title)
32:
33: paragraph = P(stylename=stdStyle)
34: text = """
35: Ceci est le texte qui sera inséré dans le document
36: LibreOffice que nous allons créer.
37: """
38: teletype.addTextToElement(paragraph, text)
39:
40: boldpart = Span(stylename=boldStyle)
41: boldText = """
42: Ceci est un passage en gras.
43: """
44: teletype.addTextToElement(boldpart, boldText)
45: paragraph.addElement(boldpart)
46:
47: paragraph.addText('Ajout rapide de texte.')
48:
49: document.text.addElement(paragraph)
50:
51: document.save('mon_document.odt')
```

COMMENTAIRES

La création d'un document odt se fait en plusieurs étapes :

1 - La création de l'objet document proprement dit (ligne 6) ;

2 - La définition des styles qui seront employés dans le document (lignes 8 à 26). Il y a deux types de styles : les styles **paragraph** et les styles **text** (comme leur nom l'indique, ces styles portent soit sur tout un paragraphe, soit seulement sur un morceau de texte). On peut définir pour ces styles des propriétés de paragraphe (**ParagraphProperties**) ou des propriétés de texte (**TextProperties**) ;

3 - Une fois les styles définis, il faut les attacher au document (lignes 22 à 26) ;

4 - On peut alors définir les différents éléments constitutifs du document : titre (lignes 30 et 31) et paragraphes (lignes 33 à 49) ;

5 - Pour finir, on enregistre le document (ligne 51). ■

RECETTE 19

LIRE UN FICHIER LibreOffice Writer

L'OBJECTIF

Accéder au contenu textuel d'un fichier odt.

LA SOLUTION

Il faut installer le module **odfpy** :

```
$ sudo pip3 install odfpy
```

Le code est ensuite le suivant :

```
01: from odf.opendocument import load
02: from odf.text import P
03:
04: doc = load('mon_document.odt')
05: for paragraph in doc.getElementsByType(P):
06:     print(paragraph)
```

COMMENTAIRES

Un document odt possède une structure de type XML et il est donc possible de naviguer dans son DOM (*Document Object Model*) pour récupérer des éléments précis comme on le fait en JavaScript. Ici l'utilisation de la méthode `getElementsType()` sur le type `P` permet de récupérer les paragraphes, mais nous aurions pu utiliser `H` pour les titres, etc.

NOTE

Un fichier odt est un fichier compressé ayant la structure suivante :

- `META-INF` (répertoire) ;
- `content.xml` ;
- `meta.xml` ;
- `mimetype` ;
- `styles.xml`.

Le fichier `content.xml` correspondant au fichier `mon_document.odt` créé précédemment contient :

```
<office:document-content office:version="1.2">
    <office:automatic-styles/>
    <office:body>
        <office:text>
            <text:h text:outline-level="1" text:style-
name="Titre_20_principal">Titre du document</text:h>
            <text:p text:style-name="Normal">
                <text:line-break/>
                Ceci est le texte qui sera inséré dans le document
                <text:line-break/>
                LibreOffice que nous allons créer.
                <text:line-break/>
                <text:span text:style-name="Gras">
                    <text:line-break/>
                    Ceci est un passage en gras.
                    <text:line-break/>
                </text:span>
                Ajout rapide de texte.
            </text:p>
        </office:text>
    </office:body>
</office:document-content>
```

Lorsque dans notre code nous demandons de récupérer les éléments de type `P` (`getElementsByType(P)`), il s'agit du passage du fichier `content.xml` ci-dessus qui a été surligné. Il n'y en a qu'un seul dans notre document et vous pouvez le vérifier en affichant un compteur devant chaque paragraphe affiché en modifiant le code de la manière suivante :

```
...
05: for num, paragraph in enumerate(doc.getElementsByType(P)):
06:     print(num, paragraph)
```

Le fichier `styles.xml` est de la forme :

```
<?xml version='1.0' encoding='UTF-8'?>
<office:document-styles ... office:version="1.2">
    <office:styles>
        <style:style style:display-name="Titre principal"
style:name="Titre_20_principal" style:family="paragraph">
            ...
    </office:styles>
</office:document-styles>
```

RECETTE 20

PARCOURIR une liste

L'OBJECTIF

Parcourir une liste et afficher les éléments en les préfixant par leur position dans la liste.

LA SOLUTION

```
01: ma_liste = ['Jarod', 'Miss Parker', 'Sydney', 'Broots',
'Raines']
02:
03: for num, name in enumerate(ma_liste):
04:     print(num, name)
```

COMMENTAIRES

La fonction `enumerate()` renvoie à chaque itération un tuple composé de la position de l'élément dans la liste et de l'élément lui-même. On évite ainsi une écriture beaucoup plus lourde du style :

```
01: ma_liste = ['Jarod', 'Miss Parker', 'Sydney', 'Broots',
'Raines']
02:
03: num = 0
04: for name in ma_liste:
05:     print(num, name)
06:     num += 1
```

NOTE

L'utilisation de la fonction `index()` permettant d'obtenir l'index d'un élément passé en paramètre, bien qu'autorisant l'économie de l'usage d'une variable, serait bien trop lourde !

```
01: ...
02: for name in ma_liste:
03:     print(ma_liste.index(name), name)
```

RECETTE 21 — TRIER UNE LISTE D'ENTIERS lus sous forme d'une chaîne de caractères

L'OBJECTIF

Une chaîne de caractères contient des entiers séparés par un point-virgule. Cette chaîne peut contenir des caractères parasites invisibles (espaces, tabulations, sauts de ligne et retours chariot). Nous souhaitons obtenir une liste de ces entiers triés par ordre croissant.

LA SOLUTION

```
01: liste = '5;12 ; 24; 6; 1    ;8 ;33\n; 21\r; 4'
02: liste_triee = sorted(map(int, liste.split(';')))
03: print(list(liste_triee))
```

COMMENTAIRES

1 Construire une liste depuis une chaîne de caractères

Pour obtenir une liste depuis la chaîne `liste`, nous avons employé la méthode `split()` qui découpe une chaîne suivant une sous-chaîne passée en paramètre. Ici, nous avons découpé la chaîne initiale suivant les caractères point-virgule. On obtient alors une liste de chaînes de caractères :

```
> print(liste.split(';'))
['5', '12\t', ' 24', ' 6', ' 1   ', '\t8 ', '33\n', ' 21\r', ' 4']
```

2 Transformer la liste de chaînes de caractères en liste d'entiers

La fonction `map()` permet d'appliquer une fonction à l'ensemble des éléments d'une liste. En appliquant la fonction `int()` aux éléments de `list.split(';')`, nous obtenons une liste d'entiers (comme `map()` renvoie un itérateur, nous appliquons la fonction `list()` sur ce dernier pour pouvoir l'afficher) :

```
> print(list(map(int, liste.split(';'))))
[5, 12, 24, 6, 1, 8, 33, 21, 4]
```

3 Le tri

Le tri est très simple une fois que l'on dispose de la liste d'entiers : nous appliquons la fonction `sorted()` nous renvoyant la liste triée. Si nous avions voulu utiliser la méthode `sort()`, il aurait fallu convertir l'itérateur retourné par `map()` en liste. Mais cela serait aberrant, car `sort()` modifie la liste sur laquelle il est appliqué et renvoie `None`... donc nous ne pouvons pas utiliser la liste qui a été générée de cette manière : `list(map(int, liste.split(';'))).sort()`. La seule solution ici est d'employer la fonction `sorted()`. ▮

RECETTE 22

TRIER UNE LISTE avec une fonction de tri personnalisée

L'OBJECTIF

Trier une liste en utilisant une fonction de tri définie par nos soins permettant ainsi de travailler sur une structure complexe. Nous prendrons pour exemple une liste de tuples contenant chacun une chaîne de caractères et un entier. Nous voulons trier ces tuples par ordre d'entiers décroissants.

LA SOLUTION

```
01: ma_liste = [('Joey', 4), ('Chandler', 2), ('Rachel', 5), ('Ross',
1), ('Phoebe', 3), ('Monica', 6)]
02:
03: ma_liste.sort(ma_liste, key = lambda elt : elt[1], reverse = True)
04:
05: print(ma_liste)
```

COMMENTAIRES

1 Le paramètre key

Pour indiquer une fonction de tri personnalisée, il faut employer le paramètre **key** et lui fournir un pointeur vers une fonction. Nous avons utilisé une fonction anonyme, mais le code suivant aurait eut exactement le même effet :

```
...
03: def compare_function(elt):
04:     return elt[1]
05:
06: ma_liste.sort(key = compare_function, reverse = True)
...
```

La fonction de comparaison renvoie l'élément qui doit être comparé. Cette méthode permet de travailler très simplement sur des objets en utilisant l'un de leur attribut en guise de clé de tri :

```
01: class Perso:
02:     def __init__(self, name, rank):
03:         self.name = name
04:         self.rank = rank
05:
```

```
06: ma_liste = [Perso('Joey', 4), Perso('Chandler', 2), Perso('Rachel', 5),
Perso('Ross', 1), Perso('Phoebe', 3), Perso('Monica', 6)]
07:
08: ma_liste.sort(key = lambda perso : perso.rank, reverse = True)
09:
10: for perso in ma_liste:
11:     print(perso.rank, perso.name)
```

Je n'ai volontairement pas défini de méthode `__str__()` dans la classe `Perso` de manière à ce que l'on puisse voir en affichant directement la liste `ma_liste` que celle-ci contient bien des objets de type `Perso`. Pour le mettre en évidence vous pouvez effectuer un affichage de `perso` dans la boucle :

```
...
12:     print(perso)
```

Vous obtiendrez un affichage du style :

```
<__main__.Perso object at 0x7fea43b9ef98>
```

En cas de besoin, la déclaration de la méthode `__str__()` pourrait ressembler à :

```
01: class Perso:
...
06:     def __str__():
07:         return "{} {}".format(self.rank, self.name)
...
```

Pour rappel, l'utilisation de `__str__()` est totalement transparente : c'est l'appel de l'objet dans un contexte de chaîne de caractères qui provoquera la recherche de `__str__()` et l'emploi de celle-ci. `print(perso)` n'affichera donc plus l'adresse d'un objet de type `Perso`, mais la chaîne définie dans la méthode `__str__()` (il est obligatoire que cette méthode retourne une chaîne de caractères).

2 Le paramètre reverse

Pour obtenir un tri en ordre décroissant, il faut indiquer un ordre « inverse », le tri par défaut se faisant de manière croissante. Pour cela, on utilise le paramètre `reverse` à qui l'on donne la valeur `True` (`False` pour un tri croissant, mais comme il s'agit de la valeur par défaut, en général on l'omet).

NOTE

Le tri personnalisé de listes a changé entre Python 2 et Python 3. Si vous utilisez un « vieux » code Python 2, la logique sera différente et la syntaxe sera :

```
...
03: def compare_function(elt1, elt2):
04:     if elt1[1] < elt2[1]:
05:         return -1
06:     elif elt1[1] > elt2[1]:
07:         return 1
08:     else:
09:         return 0
10:
11: ma_liste.sort(cmp = compare_function, reverse = True)
12:
13: print(ma_liste)
```

RECETTE 23

SUPPRIMER LES DOUBLONS dans une liste

L'OBJECTIF

Une liste d'entiers contient des doublons et l'on souhaite les supprimer pour ne conserver qu'une seule occurrence de chaque entier.

LA SOLUTION

```
01: ma_liste = [1, 1, 2, 4, 4, 5, 3, 2, 3, 6, 7, 6, 6, 7, 7]
02:
03: ma_liste_sans_doublon = list(set(ma_liste))
04: print(ma_liste_sans_doublon)
```

On obtient bien une liste sans doublon :

```
$ python3 supprimer_doublons.py
[1, 2, 3, 4, 5, 6, 7]
```

COMMENTAIRES

Un ensemble (`set`) ne peut pas contenir de doublon. La solution consiste donc à convertir la liste en ensemble puis à nouveau en liste. Bien entendu, il aurait été beaucoup plus efficace de travailler directement avec des ensembles :

```
01: ma_liste = set((1, 1, 2, 4, 4, 5, 3, 2, 3, 6, 7, 6, 6, 7, 7))
02:
03: print(ma_liste)
```

Et pour être complet, l'ajout des éléments se fait certainement un à un (sinon nous aurions supprimé les doublons directement). Voici comment faire sur un objet de type `set` :

```
01: ma_liste = set()
02: ma_liste.add(1)
03: ma_liste.add(1)
04: ...
05: ma_liste.add(7)
06:
07: print(ma_liste)
```

TRIER LES VALEURS d'un dictionnaire

L'OBJECTIF

Afficher toutes les valeurs contenues dans un dictionnaire en les triant par ordre alphabétique.

LA SOLUTION

```
01: dico = {}
02: dico['David'] = 'Ziva'
03: dico['Gibbs'] = 'Leroy Jethro'
04: dico['DiNozzo'] = 'Anthony'
05: dico['Sciuto'] = 'Abigail'
06: dico['McGee'] = 'Timothy'
07:
08: for firstname in sorted(dico.values()):
09:     print(firstname)
```

COMMENTAIRES

1 La méthode values()

Pour obtenir une liste des valeurs d'un dictionnaire, on utilise la méthode `values()` :

```
> print(dico.values())
dict_values(['Anthony', 'Ziva', 'Leroy Jethro', 'Abigail', 'Timothy'])
```

2 La fonction sorted()

Comme nous sommes dans une boucle `for` (ligne 8), nous voulons itérer sur une liste et nous ne pouvons pas utiliser la méthode `sort()` qui modifie la liste sur laquelle elle est appliquée et renvoie la valeur `None`. La fonction `sorted()` effectue exactement la même tâche que `sort()`, mais renvoie la liste triée plutôt que de la modifier. Dans l'exemple suivant, on voit bien que la liste `val` est modifiée :

```
> val = list(dico.values())
> print(val)
['Ziva', 'Abigail', 'Anthony', 'Leroy Jethro', 'Timothy']
> print(val.sort())
None
> print(val)
['Abigail', 'Anthony', 'Leroy Jethro', 'Timothy', 'Ziva']
```

En utilisant `sorted()`, la fonction renvoie bien une nouvelle liste triée, mais ne modifie pas la liste de départ :

```
> val = list(dico.values())
> print(val)
['Leroy Jethro', 'Ziva', 'Abigail', 'Timothy', 'Anthony']
> print(sorted(val))
['Abigail', 'Anthony', 'Leroy Jethro', 'Timothy', 'Ziva']
> print(val)
['Leroy Jethro', 'Ziva', 'Abigail', 'Timothy', 'Anthony']
```

RECETTE 25

CONSERVER L'ORDRE DES ÉLÉMENTS dans un dictionnaire

L'OBJECTIF

Par définition, un dictionnaire est une structure non ordonnée : vous ne pourrez pas lire ses éléments dans l'ordre dans lequel ils ont été insérés. Nous allons créer un dictionnaire ordonné grâce à l'objet `OrderedDict`.

LA SOLUTION

```
01: import collections
02:
03: dico = collections.OrderedDict()
04: dico['Castle'] = 'Richard'
05: dico['Beckett'] = 'Kate'
06: dico['Esposito'] = 'Javier'
07: dico['Ryan'] = 'Kevin'
08:
09: for name, firstname in dico.items():
10:     print(firstname, name)
```

COMMENTAIRES

1 Résultats

À l'exécution, vous obtiendrez la liste suivante :

```
Richard Castle
Kate Beckett
Javier Esposito
Kevin Ryan
```

Le dictionnaire est ordonné.

Si en ligne 3 nous avions utilisé un dictionnaire classique en utilisant `dico = {}`, le résultat aurait été quelque chose semblable à :

```
Kevin Ryan
Javier Esposito
Richard Castle
Kate Beckett
```

On vérifie bien qu'un dictionnaire classique n'est pas ordonné.

2 Méthodes dignes d'intérêt

Il faut noter l'existence de deux méthodes particulièrement intéressantes :

- `popitem()` : renvoie le dernier élément inséré dans le dictionnaire et le supprime de celui-ci. En passant en paramètre `last=False`, renvoie le premier élément et le supprime du dictionnaire.
- `move_to_end()` : déplace un élément du dictionnaire à la fin (ou au début de celui-ci si l'on spécifie l'attribut `last=False`). Il faut indiquer la clé de l'élément à déplacer et gérer une exception `KeyError` si l'élément n'existe pas.

```
try:
    elt = dico.move_to_end('Castle')
    print('Déplacement effectué')
except KeyError:
    print('Action impossible')
```

NOTE

Pour bien voir l'impact de l'objet `OrderedDict`, on peut comparer les deux types de dictionnaires :

```
01: import collections
02:
03: def createDico(dico):
04:     dico['Castle'] = 'Richard'
05:     dico['Beckett'] = 'Kate'
06:     dico['Esposito'] = 'Javier'
07:     dico['Ryan'] = 'Kevin'
08:     return dico
09:
10: dico = createDico({})
11: dico_ord = createDico(collections.OrderedDict())
12:
13: print('Standard:')
14: for name, firstname in dico.items():
15:     print(firstname, name)
16:
17: print('\nOrderedDict:')
18: for name, firstname in dico_ord.items():
19:     print(firstname, name)
20:
21: print('\ndico == dico_ord ? {}'.format(dico == dico_ord))
```

La sortie nous montre bien l'impact d'`OrderedDict` sur l'insertion des données et le fait que les deux dictionnaires contiennent les mêmes informations :

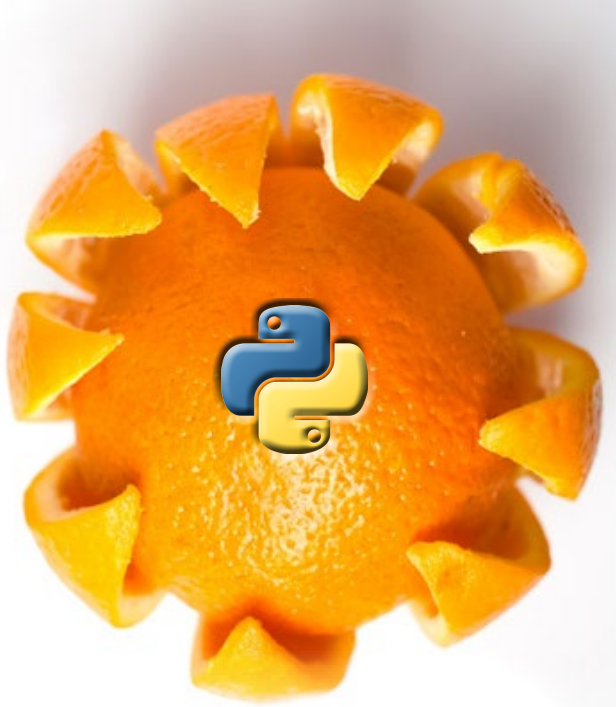
```
Standard:
Kevin Ryan
Kate Beckett
Richard Castle
Javier Esposito

OrderedDict:
Richard Castle
Kate Beckett
Javier Esposito
Kevin Ryan

dico == dico_ord ? True
```

RECETTE 26

CRÉER UN DICTIONNAIRE qui soit la fusion de deux dictionnaires



L'OBJECTIF

Étant donnés deux dictionnaires, nous souhaitons créer un nouveau dictionnaire contenant toutes les données des deux dictionnaires précédents.

LA SOLUTION

```
01: dico_1 = {'Nathaniel': 'Taylor', 'Malcom': 'Wallace', 'Lucas':
'Taylor'}
02: dico_2 = {'Jim': 'Shannon', 'Elisabeth': 'Shannon', 'Maddy':
'Shannon'}
03:
04: new_dico = dico_1.copy()
05: new_dico.update(dico_2)
06: print(new_dico)
```

COMMENTAIRES

Il est obligatoire de commencer par effectuer une copie du premier dictionnaire (ligne 4) puis de mettre à jour les données avec les informations contenues dans le second dictionnaire (ligne 5).

Notez que si `dico_1` et `dico_2` ont des clés en commun, ce seront les informations contenues par `dico_2` qui remplaceront les informations obtenues depuis `dico_1` :

```
> dico_1 = {'Nathaniel': 'Taylor', 'Malcom': 'Wallace', 'Lucas':
'Taylor'}
> dico_2 = {'Jim': 'Shannon', 'Elisabeth': 'Shannon', 'Maddy':
'Shannon', 'Lucas': ''}
> new_dico = dico_1.copy()
> new_dico.update(dico_2)
> new_dico
{'Elisabeth': 'Shannon', 'Jim': 'Shannon', 'Malcom': 'Wallace',
'Nathaniel': 'Taylor', 'Lucas': '', 'Maddy': 'Shannon'}
```

Cette solution n'est pas très élégante. Si vous utilisez Python 3.5 ou supérieur, vous pouvez remplacer les lignes 4 et 5 par :

```
new_dico = {**dico_1, **dico2}
```

RECETTE 27 — CRÉER UNE LISTE en utilisant la compréhension de liste

L'OBJECTIF

Nous allons créer deux listes :

- la première contient des tuples avec une valeur, son carré et son cube ;
- la seconde contient toutes les coordonnées entières de (0, 0) à (10, 10).

LA SOLUTION

```
01: ma_liste = [(num, num ** 2, num ** 3) for num in (1, 4, 7, 12)]
02:
03: ma_liste_2 = [(x, y) for x in range(11) for y in range(11)]
```

COMMENTAIRES

1 Sans compréhension de liste

Si nous n'avions pas utilisé la compréhension de liste notre code aurait été le suivant :

```
01: ma_liste = []
02: for num in (1, 4, 7, 12):
03:     ma_liste.append((num, num ** 2, num ** 3))
04:
05: ma_liste_2 = []
06: for x in range(11):
07:     for y in range(11):
08:         ma_liste_2.append((x, y))
```

Comme vous le voyez, le code est beaucoup plus long...

2 Utilisation de l'opérateur **

L'opérateur `**` permet d'élever un nombre à une puissance donnée. Ainsi, `n ** m` signifie n^m.

3 Précautions d'emploi

La compréhension de liste est un mécanisme très puissant permettant de réaliser des choses complexes en peu de lignes. Tant que ces lignes sont compréhensibles, il est judicieux de l'employer... mais à partir du moment où il faudrait passer un long moment pour comprendre ce que fait la ligne, il vaut mieux utiliser une structuration classique. Une compréhension de liste avec un ou deux `for` est en général aisément compréhensible, avec trois c'est moins sûr et au-delà il vaut mieux éviter de l'employer ! ▮

RECETTE 28

CRÉER UN DICTIONNAIRE en utilisant la compréhension de dictionnaire

L'OBJECTIF

À partir d'une liste alternant des noms et des prénoms en minuscule, nous voulons créer un dictionnaire dont la clé sera le nom en majuscules et la valeur le prénom avec la première lettre en majuscule.

LA SOLUTION

```
01: ma_liste = ['murdoch', 'william', 'ogden', 'julia',
'brackenreid', 'thomas', 'crabtree', 'george']
02:
03: dico = { name.upper() : firstname.capitalize() for name,
firstname in zip(ma_liste[::2], ma_liste[1::2])}
04:
05: print(dico)
```

COMMENTAIRES

1 Le slicing

Dans un premier temps, il faut être capable de détecter les noms puis les prénoms. Nous savons que les noms ont pour indice 0, 2, 4, etc. alors que les prénoms ont pour indice 1, 3, 5, etc. Le *slicing* va nous permettre de ne récupérer que les noms ou que les prénoms en indiquant l'indice du caractère de départ (`0` par défaut s'il est omis), l'indice du caractère de fin non compris (la fin du mot par défaut), et le pas (1 par défaut). Avec ce mécanisme, on récupère les noms ainsi :

```
> ma_liste[::2]
['murdoch', 'ogden', 'brackenreid', 'crabtree']
```

Pour les prénoms, nous gardons un pas de 2, mais en décalant d'un élément le début du parcours de la liste :

```
> ma_liste[1::2]
['william', 'julia', 'thomas', 'george']
```

2 La fonction zip

Il est possible d'associer les éléments en indice `i` de plusieurs listes au sein d'un tuple et de construire une liste avec tous ces éléments grâce à la fonction `zip()`. C'est ce qui nous permet d'associer les noms et les prénoms :

```
> print(list(zip(ma_liste[::2], ma_liste[1::2])))
[('murdoch', 'william'), ('ogden', 'julia'), ('brackenreid',
'thomas'), ('crabtree', 'george')]
```

3 La compréhension de dictionnaire

Une fois que l'on a réussi à associer nom et prénom de chaque personnage, il suffit de parcourir la liste et d'utiliser les valeurs des tuples comme clés et valeurs du dictionnaire :

```
dico = { name : firstname for name, firstname in zip(ma_liste[::2],
ma_liste[1::2])}
```

4 Les méthodes upper et capitalize

Pour passer le nom en majuscules, nous utilisons la méthode `upper()` et pour obtenir les prénoms avec la première lettre en majuscule, nous utilisons la méthode `capitalize()`.

Sans ces méthodes, l'écriture aurait été beaucoup plus complexe :

```
01: ma_liste = ['murdoch', 'wIlliam1', 'ogden', 'julia',
'brackenreid', 'thomas', 'crabtree', 'george']
02:
03: def my_upper(string):
04:     result = ''
05:     for car in string:
06:         if ord(car) >= 97 and ord(car) <= 122:
07:             result += chr(ord(car) - 32)
08:         else:
09:             result += car
10:     return result
11:
12: def my_capitalize(string):
13:     if ord(string[0]) >= 97 and ord(string[0]) <= 122:
14:         result = chr(ord(string[0]) - 32)
15:     else:
16:         result = string[0]
17:     for car in string[1:]:
18:         if ord(car) >= 65 and ord(car) <= 90:
19:             result += chr(ord(car) + 32)
20:         else:
21:             result += car
22:     return result
23:
24: dico2 = { my_upper(name) : my_capitalize(firstname)  for name,
firstname in zip(ma_liste[::2], ma_liste[1::2])}
25: print(dico2)
```

Il aurait ainsi fallu avoir recours aux fonctions `ord()` et `chr()` permettant respectivement d'obtenir le code ASCII d'un caractère et d'obtenir le caractère associé à un code ASCII. ▮

2 ORGANISEZ VOTRE CODE

À découvrir dans cette partie...

• LES FONCTIONS

Le premier élément de structuration d'un code consiste à créer des fonctions et à pouvoir agir sur les valeurs de retour ou celles qui seront passées en paramètres. Certaines structures, plus particulières, permettent même de créer des fonctions qui ne seront rattachées à aucun nom. (Recettes 29 - 34)

• DÉCORATEURS, ITÉRATEURS ET GÉNÉRATEURS

Il est fréquent de constater chez nombre de développeurs Python une confusion entre les trois structures que sont les décorateurs, les itérateurs et les générateurs qui sont peu voire sous employées. Les recettes de cette partie devraient vous permettre de les employer plus fréquemment. (Recettes 35 - 39)

• PROGRAMMATION ORIENTÉE OBJET

En programmation orientée objet, on utilise souvent la même structuration, mais certains éléments, moins employés que d'autres doivent parfois être rappelés. Ainsi, dans cette partie sont traités les attributs et méthodes de classes, mais également des structures plus « exotiques ». (Recettes 40 - 48)

NOS RECETTES POUR...

29 Créer une fonction ayant un nombre de paramètres infini p.52
30 Créer une fonction anonyme p.53
31 Créer une fonction qui renvoie une fonction p.54
32 Connaître l'état de sortie d'une boucle for p.55
33 Affecter une variable de manière conditionnelle p.56
34 Creer une fonction ayant un comportement différent suivant le type de ses paramètres p.58
35 Créer un décorateur de log p.59
36 Créer un décorateur indiquant les types attendus pour les paramètres d'une fonction p.60
37 Créer un itérateur p.61
38 Créer un itérateur à l'aide d'un générateur p.62
39 Créer un paquetage de modules p.64
40 Créer des constantes p.66
41 Créer un attribut de classe p.67
42 Créer une méthode de classe p.68
43 Créer un singleton p.69
44 Surcharger un opérateur p.70
45 Rendre un objet callable p.71
46 Ajouter un attribut à un objet sans le dégrader p.72
47 Ajouter une méthode à une classe existante p.73
48 Créer un sélecteur de méthode p.74

RECETTE 29

CRÉER UNE FONCTION ayant un nombre de paramètres non fixé

L'OBJECTIF

Créer un équivalent de la fonction réalisant la somme du carré de tous les entiers qui lui sont passés en paramètre.

LA SOLUTION

```
01: def sq_sum(*args):
02:     result = 0
03:     for elt in args:
04:         result += elt ** 2
05:     return result
06:
07: print(sq_sum(1, 4, 5, 6, 2, 8))
```

COMMENTAIRES

Le paramètre `*args` (seule l'étoile est importante, vous pouvez changer le nom du paramètre si vous le souhaitez) récupère une liste de paramètres. `args` est donc une liste que l'on peut parcourir pour effectuer diverses actions.

Ici, en appelant `sq_sum(1, 4, 5, 6, 2, 8)`, `args` a pour valeur la liste `[1, 4, 5, 6, 2, 8]`.

NOTE

Pour les paramètres nommés, on utilise `**kwargs` qui est alors un dictionnaire où la clé est le nom du paramètre et la valeur... sa valeur :

```
> def fct(**kwargs):
      print(kwargs)
> fct(a=1, b='hello')
{'a': 1, 'b': 'hello'}
```

Attention : dans le cas où vous utiliseriez à la fois les paramètres `*args` et `**kwargs`, une fois que vous commencez à nommer des paramètres vous ne pouvez plus utiliser des paramètres « anonymes » :

```
> def fct(*args, **kwargs):
    print(args)
    print(kwargs)
> fct(1, 2, a=5, 'hello')
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

RECETTE 30 — CRÉER UNE FONCTION anonyme

L'OBJECTIF

Créer et utiliser une fonction qui sera détruite immédiatement après emploi. Nous utiliserons comme exemple une liste d'entiers (sous forme de chaîne de caractères) dont nous voulons obtenir les carrés.

LA SOLUTION

```
01: ma_liste = ['1', '4', '7', '9']
02:
03: print(list(map(lambda x: int(x) ** 2, ma_liste)))
```

COMMENTAIRES

Les fonctions anonymes sont également appelées lambda fonctions. Cette écriture issue de la programmation fonctionnelle permet de passer en paramètre à une fonction une autre fonction que l'on ne souhaite pas conserver en mémoire. Dans notre exemple, la fonction `map()` va appliquer à tous les éléments de la liste qui lui est passée en second paramètre (`ma_liste`) la fonction qui constitue son premier paramètre. Cette fonction est une fonction anonyme : ces paramètres sont déclarés après le mot-clé `lambda` et le corps de la fonction se trouve après le caractère deux-points. Ainsi, la fonction anonyme `lambda x: int(x) ** 2` est équivalente à :

```
def compute(x) :
    return int(x) ** 2
```

Bien qu'il soit possible de conserver en mémoire une fonction lambda, cette pratique n'est pas recommandée s'il ne s'agit que d'écrire une fonction sur un minimum de lignes :

```
> f = lambda x : x ** 2
> f(2)
4
```

NOTE

Pour écrire une fonction anonyme acceptant plusieurs paramètres, il suffit de les lister après le mot-clé `lambda` :

```
> g = lambda x, y: x**2 + y**2
> g(2, 3)
13
```

RECETTE 31

CRÉER UNE FONCTION qui renvoie une fonction

L'OBJECTIF

Créer une fonction qui, en fonction des paramètres qui lui seront passés, créera une autre fonction.

Nous prendrons pour exemple une fonction qui renverra une liste constituée de l'ensemble des paramètres qui lui auront été transmis et qui seront élevés à une puissance passée en paramètre de la fonction de génération. Sur l'exemple ce sera plus compréhensible...

LA SOLUTION

```
01: def genere_fonction(exposant):
02:     return lambda *e : [x ** exposant for x in e]
03:
04: carre_liste = genere_fonction(2)
05: print(carre_liste(1, 2, 3, 4, 5))
06: cube_liste = genere_fonction(3)
07: print(cube_liste(1, 2, 3, 4, 5))
```

COMMENTAIRES

On utilise ici une fonction anonyme pour renvoyer la fonction (ligne 2). `*e` fait référence à une liste de paramètres (taille non déterminée) qui sera transmise à la fonction. Par exemple, avec l'appel de la ligne 5 ou 6, `e` contient le tuple `(1, 2, 3, 4, 5)`. La fonction `genere_fonction()` crée donc une fonction anonyme qu'elle renvoie (parcourt des paramètres puis création d'une liste où les paramètres sont élevés à la puissance `exposant`).

Voici le résultat obtenu :

```
$ python3 genere_fct.py
[1, 4, 9, 16, 25]
[1, 8, 27, 64, 125]
```

NOTE

Si la fonction renvoyée avait été plus complexe, nous aurions pu choisir une autre architecture telle que la suivante :

```
01: def liste_exp(exposant, *args):
02:     return [x ** exposant for x in args]
03:
04: def genere_fonction(exposant):
05:     return lambda *e : liste_exp(exposant, *e)
...
```

RECETTE 32 CONNAÎTRE L'ÉTAT de sortie d'une boucle for

L'OBJECTIF

Savoir si une boucle `for` a été interrompue par un `break` ou non. Nous utiliserons comme exemple l'analyse d'une liste d'entiers pour savoir si cette dernière contient au moins un nombre pair.

LA SOLUTION

```
01: def detect_even_in_list(liste):
02:     for elt in liste:
03:         if elt % 2 == 0:
04:             print('La liste contient au moins un nombre pair')
05:             break
06:     else:
07:         print('La liste ne contient pas de nombre pair')
08:
09: detect_even_in_list([1, 3, 5, 6, 7, 9])
```

COMMENTAIRES

La structure `for`/`else` n'est pas très employée. Pourtant, elle permet d'exécuter du code seulement si l'on est sorti de la boucle « normalement », sans utiliser de `break`. On économise ainsi une variable booléenne et quelques lignes. Le code précédent sans `for`/`else` serait en effet :

```
01: def detect_even_in_list(liste):
02:     is_even_in_list = False
03:     for elt in liste:
04:         if elt % 2 == 0:
05:             is_even_in_list = True
06:             break
07:
08:     if is_even_in_list:
09:         print('La liste contient au moins un nombre pair')
10:     else:
11:         print('La liste ne contient pas de nombre pair')
12:
13: detect_even_in_list([1, 3, 5, 6, 7, 9])
```

Notez bien que cet exemple a été choisi pour montrer le comportement du `for`/`else`. Il est bien évident que dans ce cas l'usage de `return` est bien plus indiqué :

```
01: def detect_even_in_list(liste):
02:     for elt in liste:
03:         if elt % 2 == 0:
04:             return 'La liste contient au moins un nombre pair'
05:     return 'La liste ne contient pas de nombre pair'
06:
07: print(detect_even_in_list([1, 3, 5, 6, 7, 9]))
```

Pour aller encore plus loin, la fonction ne devrait renvoyer qu'un booléen pour indiquer la présence ou l'absence d'un nombre pair. La structure `for`/`else` est donc très intéressante, mais assurez-vous que l'on ne puisse pas faire mieux sans l'utiliser... ▮

RECETTE 33

CRÉER UNE FONCTION ayant un comportement différent suivant le type de ses paramètres

L'OBJECTIF

Nous souhaitons créer une fonction réagissant différemment suivant que l'on utilise des entiers ou des chaînes de caractères en paramètre. Pour cela, nous allons définir une fonction permettant d'ajouter soit deux entiers et le résultat sera la somme, soit deux chaînes de caractères et le résultat sera la somme des valeurs ASCII des caractères des deux chaînes.

LA SOLUTION

```
01: from functools import singledispatch
02:
03: @singledispatch
04: def add(a, b):
05:     raise NotImplementedError('Les types fournis ne sont pas
supportés!')
06:
07: @add.register(int)
08: def _(a, b):
09:     return a + b
10:
11: @add.register(str)
12: def _(a, b):
13:     return sum(list(map(ord, a)) + list(map(ord, b)))
14:
15: print(add(2, 3))
16: print(add('GLMF', 'LP'))
```

COMMENTAIRES

1 Singledispatch

Le décorateur `singledispatch` est fourni par le module `functools` (seulement pour Python 3.4 est supérieur). Il suffit de décorer la fonction souhaitée en indiquant que, par défaut, les types des paramètres ne sont pas supportés (voir lignes 3 à 5). C'est seulement ensuite que l'on indique les types autorisés. Notez bien qu'en fait seul le type du premier paramètre est vérifié (d'où le nom de *single dispatching*).

Pour indiquer le comportement de la fonction décorée en fonction du type du premier paramètre, on utilise un décorateur qui sera de la forme `@nom_fonction.register(type_paramètre)`. Lors de la définition de

la fonction proprement dite, le `_` fera référence au nom de fonction utilisé dans le décorateur (voir lignes 7-8 et 11-12). Ainsi, si le premier paramètre est de type `int` la fonction des lignes 8 et 9 sera exécutée, si le type est `str` ce seront les lignes 12 et 13 et sinon ce seront les lignes 4 et 5.

2 La somme des éléments des deux listes de codes ASCII

Pour obtenir le code ASCII d'un caractère, il faut utiliser la fonction `ord()` :

```
> ord('A')
65
```

Une chaîne de caractères est un tuple de caractères (liste non modifiable). Nous pouvons donc utiliser la fonction `map()` qui va appliquer la fonction `ord()` à tous les éléments de la chaîne. Pour finir, nous convertissons le résultat en liste avec `list()` :

```
> list(map(ord, 'GLMF'))
[71, 76, 77, 70]
```

Pour ajouter les éléments de deux listes, il suffit d'utiliser l'opérateur `+` :

```
> ['G', 'L', 'M', 'F'] + ['L', 'P']
['G', 'L', 'M', 'F', 'L', 'P']
```

Enfin, pour effectuer la somme de tous les éléments d'une liste, il faut utiliser la fonction `sum()` :

```
> sum([1, 2, 3, 4])
10
```

La ligne 13 applique toutes ces fonctions pour calculer la somme des codes ASCII des caractères des deux chaînes passées en paramètre à la fonction `add()`.

3 Et pour des paramètres de types différents ?

Le décorateur `singledispatch` ne pouvant pas être utilisé dans ce cas, il va falloir installer un module permettant le *multiple dispatching* :

```
$ sudo pip3 install multipledispatch
```

Le fonctionnement est très proche de `singledispatch`. Imaginons l'ajout d'un entier et d'une chaîne qui sera l'addition de cet entier avec la somme des valeurs des codes ASCII des caractères de la chaîne :

```
01: from multipledispatch import dispatch
02:
03: @dispatch(int, int)
04: def add(a, b):
05:     return a + b
06:
07: @dispatch(int, str)
08: def add(a, b):
09:     return a + sum(list(map(ord, b)))
10:
11: print(add(2, 3))
12: print(add(2, 'GLMF'))
```

Ici, l'exception `NotImplementedError` est directement incluse :

```
> print(add('GLMF', 'LP'))
...
NotImplementedError: Could not find signature for add: <str, str>
```

Notez également que `@dispatch(int, str)` est bien entendu différent de `@dispatch(str, int)`.

RECETTE 34

AFFECTER UNE VARIABLE de manière conditionnelle

L'OBJECTIF

Affecter une valeur à une variable en fonction d'une autre variable sur une seule ligne.

LA SOLUTION

```
01: valeur = int(input('Saisissez un entier : '))
02:
03: ma_variable = True if valeur == 10 else False
04: print(ma_variable)
```

COMMENTAIRES

Dans cet exemple, l'utilisateur doit saisir un entier (nous n'avons pas géré les cas d'erreur) et en fonction de la valeur saisie, la variable `ma_variable` vaudra `True` (si `valeur` vaut `10`) ou `False`. L'intérêt réside ici dans le fait que l'on ait pu écrire notre déclaration sur une seule ligne au lieu de :

```
if valeur == 10 :
    ma_variable = True
else:
    ma_variable = False
```

NOTE

Il est possible de réaliser des traitements plus complexes en enchaînant les conditions :

```
...
03: ma_variable = 'A' if valeur == 10 else 'B' if valeur == 11 else 'C'
...
```

Prenez garde toutefois que votre code reste lisible ! L'écriture précédente, est équivalente à l'écriture suivante :

```
...
03: if valeur == 10 :
04:     ma_variable = 'A'
05: elif valeur == 11 :
06:     ma_variable = 'B'
07: else :
08:     ma_variable = 'C'
...
```

RECETTE 35 CRÉER UN DÉCORATEUR de log

L'OBJECTIF

Créer un décorateur affichant les paramètres transmis à une fonction.

LA SOLUTION

```
01: def logger(function):
02:     def intern(*args, **kwargs):
03:         print('Arguments transmis :')
04:         print('  args   : {}'.format(args))
05:         print('  kwargs : {}'.format(kwargs))
06:         return function(*args, **kwargs)
07:     return intern
08:
09: @logger
10: def test_fct(val1, val2, val3):
11:    print('Juste un test')
12:
13: test_fct(1, 'GLMF', val3=12)
```

COMMENTAIRES

1 Le résultat

En exécutant le code précédent, vous obtiendrez le résultat suivant :

```
Arguments transmis :
  args   : (1, 'GLMF')
  kwargs : {'val3': 12}
Juste un test
```

2 Explications

La fonction `logger()` utilisée comme décorateur de la fonction `test_fct()` dans les lignes 9 et 10 va être appelée avec pour paramètre ladite fonction. Nous pouvons le montrer en ajoutant la ligne suivante :

```
01: def logger(function):
02:     def intern(*args, **kwargs):
...
06:         return function(*args, **kwargs)
07:     print(function.__name__)
08:     return intern
...
```

L'appel à `return intern` va ensuite exécuter la fonction interne à `logger()` et justement nommée `intern`. Les paramètres `*args` et `**kwargs` sont ceux qui ont été utilisés avec la fonction passée en paramètre de `logger()`. Ils peuvent donc être utilisés (voir lignes 3 à 5), puis on effectue un `return` sur `function(*args, **kwargs)` de manière à rendre la main à la fonction décorée et que celle-ci puisse effectuer son travail. ▮

RECETTE 36

CRÉER un itérateur

L'OBJECTIF

Créer un objet itérable fournissant aléatoirement des entiers compris entre 1 et 6 (comme un dé) et qui s'arrête lorsque l'on tire un 6.

LA SOLUTION

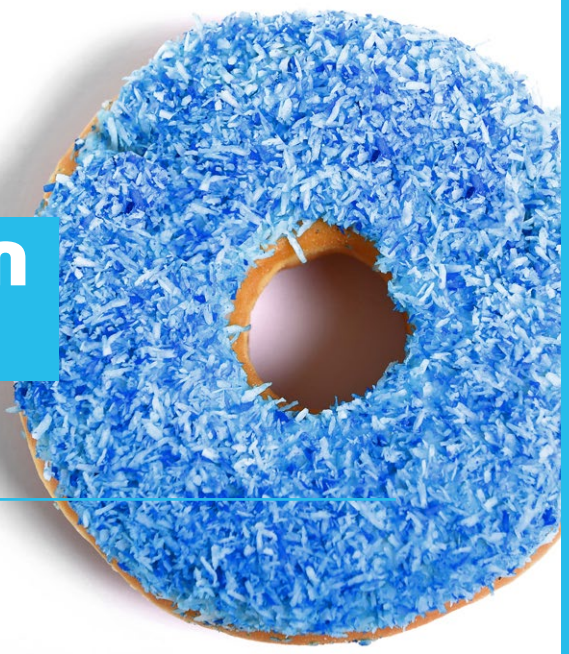
```
01: import random
02:
03: class Dice:
04:     def __iter__(self):
05:         return self
06:
07:     def __next__(self):
08:         n = random.randint(1, 6)
09:         if n >= 6:
10:             raise StopIteration
11:         return n
12:
13: if __name__ == '__main__':
14:     dice = Dice()
15:     for number in dice:
16:         print(number)
```

COMMENTAIRES

Pour définir un objet itérable, il faut au moins deux méthodes :

- `__iter__()` : renvoie l'objet. Cette méthode est appelée à l'aide de la fonction `iter()` à laquelle l'instance de l'objet est passée en paramètre. On s'en sert pour remettre éventuellement des paramètres à zéro pour pouvoir relancer une itération (l'instruction `for` exécute cela automatiquement) ;
- `__next__()` : renvoie l'élément suivant. C'est la méthode qui est appelée dans une boucle `for`. Il faut penser à lever une exception `StopIteration` pour indiquer que toutes les valeurs ont été fournies (l'instruction `for` traite cette exception de manière à arrêter la boucle). ▮

CRÉER UN ITÉRATEUR à l'aide d'un générateur



L'OBJECTIF

Créer un objet itérable fournissant aléatoirement des entiers compris entre 1 et 6 (comme un dé) et qui s'arrête lorsque l'on tire un 6.

LA SOLUTION

```
01: import random
02:
03: def dice():
04:     while True:
05:         n = random.randint(1, 6)
06:         if n >= 6:
07:             raise StopIteration
08:         else:
09:             yield n
10:
11:
12: if __name__ == '__main__':
13:     for number in dice():
14:         print(number)
```

COMMENTAIRES

Contrairement à la définition manuelle d'un itérateur, nous ne sommes pas obligés d'utiliser de la programmation orientée objet. Une simple fonction utilisant `yield` au lieu de `return` deviendra un générateur nous permettant d'itérer sur les valeurs renvoyées. `yield` fonctionne comme un `return` qui conserve le contexte d'exécution au sein de la fonction. Considérons que le code de la fonction `generator()` soit le suivant :

```
def generator():
    for i in range(0, 10):
        yield i
```

À chaque appel à la méthode `__next__()`, nous obtiendrons la valeur suivante de `i` : la variable `i` au sein de la fonction `generator()` n'est pas effacée. Voici un exemple d'appel :

```
> g = generator()
> g.__next__()
0
> g.__next__()
1
...
```

Le fonctionnement est le même avec `dice()`. ■

RECETTE 38

CRÉER UN DÉCORATEUR indiquant les types attendus pour les paramètres d'une fonction

L'OBJECTIF

Créer un décorateur affichant les types attendus par les paramètres d'une fonction et le type de la valeur retournée. Pour indiquer les types des paramètres et de la valeur de retour, nous utiliserons les annotations de fonctions.

LA SOLUTION

```
01: def helper(function):
02:     def intern(*args, **kwargs):
03:         print('Helper on function {}'.format(function.__name__))
04:         for arg, argType in function.__annotations__.items():
05:             if arg != 'return':
06:                 print('  {} : {}'.format(arg, argType))
07:         print('  Return type : {}'.format(function.
__annotations__['return']))
08:         return function(*args, **kwargs)
09:     return intern
10:
11: @helper
12: def test_fct(val1 : int, val2 : str, val3 : int) -> bool:
13:    print('Juste un test')
14:    return True
15:
16: test_fct(1, 'GLMF', val3=12)
```

COMMENTAIRES

1 Le résultat

En exécutant le code précédant, vous obtiendrez le résultat suivant :

```
Helper on function test_fct
  val2 : <class 'str'>
  val1 : <class 'int'>
  val3 : <class 'int'>
  Return type : <class 'bool'>
Juste un test
```

2 Le décorateur

Pour l'explication sur le décorateur, vous pouvez vous reporter à *« Créer un décorateur de log »*.

3 Les annotations

Les annotations d'une fonction permettent de donner une indication sur le type attendu pour les paramètres et le type de la valeur retournée. Pour les paramètres, le type est donné après le paramètre et un caractère deux-points et pour le type de retour de la fonction, on utilise une flèche `->` (voir ligne 12).

L'accès aux informations d'annotation se fait à l'aide du dictionnaire `__annotations__` :

```
> def fct(val1 : int, val2 : str, val3 : int) -> bool :
...
> print(fct.__annotations__)
{'return': <class 'bool'>, 'val2': <class 'str'>, 'val3': <class
'int'>, 'val1': <class 'int'>}
```

NOTE

Notez bien que les annotations n'ont aucun caractère restrictif : il ne s'agit que de commentaires insérés dans le code que l'on peut ensuite utiliser comme bon nous semble. Par exemple, le fait d'avoir inscrit `def fct(val1 : int, ...)` ne rend pas le type entier obligatoire pour le paramètre `val1` ! Nous aurions pu tout aussi bien écrire `def fct(val1 : 'ceci est mon commentaire', ...)`. `fct.__annotations__` contiendrait alors `{'val1': 'ceci est mon commentaire', ...}`. Voici un court exemple illustrant le fait que sans traitement particulier spécifié par le développeur, les annotations n'ont aucun impact sur le code :

```
> def fct(val1 : int, val2 : 'le deuxieme parametre', val3 :
12) -> 'ma fonction':
...     print(val1, val2, val3)
...     return
...
>>> fct(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fct() missing 2 required positional arguments:
'val2' and 'val3'
>>> type(fct('hello', 12, 3))
hello 12 3
<class 'NoneType'>
```

On voit bien que les annotations ne constituent pas des valeurs par défaut et ne contraignent pas le type des paramètres.

RECETTE 39

CRÉER UN PAQUETAGE de modules

L'OBJECTIF

Organiser des modules dans un paquetage contenant des sous-paquetages. Pour cela, nous utiliserons la structure suivante :

```
paquet
├── module_X.py
├── sous_paquet_1
│   ├── module_a.py
│   └── module_b.py
└── sous_paquet_2
    ├── module_1.py
    └── module_2.py
```

LA SOLUTION

Tout d'abord, nous devons ajouter des fichiers `__init__.py` dans chaque répertoire de manière à les rendre accessibles pour l'import de modules :

```
paquet
├── __init__.py
├── module_X.py
├── sous_paquet_1
│   ├── __init__.py
│   ├── module_a.py
│   └── module_b.py
└── sous_paquet_2
    ├── __init__.py
    ├── module_1.py
    └── module_2.py
```

Cette solution est déjà fonctionnelle pour un import du type :

```
from . import sous_paquet_1
from . import sous_paquet_2
```

Par contre, si nous souhaitons importer `paquet` et avoir accès à tous les modules, ça ne marche pas ! Il faut modifier les fichiers `__init__.py` pour qu'ils importent les différents modules :

- `paquet/__init__.py` :

```
from . import sous_paquet_1
from . import sous_paquet_2
```

- `paquet/sous_paquet_1/__init__.py` :

```
from . import module_a
from . import module_b
```

- `paquet/sous_paquet_2/__init__.py` :

```
from . import module_1
from . import module_2
```

Il est alors possible d'utiliser par exemple une fonction `display()` définie dans `module_a` par :

```
import paquet
paquet.sous_paquet_1.module_a.display()
```

COMMENTAIRES

1 Le fichier __init__.py

La présence d'un fichier `__init__.py` rend le répertoire accessible pour la recherche et l'import de modules. Il est possible d'y insérer des instructions qui seront exécutées lors de l'import du nom du répertoire auquel il appartient. C'est le cas de `paquet/__init__.py` qui contient des lignes d'import qui seront exécutées lors de l'appel de `import paquet`.

2 Les erreurs d'import à éviter

Les imports circulaires

Si vous créez deux modules qui s'importent l'un l'autre, vous risquez d'avoir beaucoup de mal à détecter l'erreur. En effet, l'exception que Python lèvera sera une `AttributeError` qui ne vous éclairera pas forcément... Évitez donc les codes semblables au suivant :

- Fichier `mod_a.py` :

```
01: import mod_b
02:
03: def a_display() :
04:     print('Hello ', end='')
05:     mod_b.b_display()
06:
07: a_display()
```

- Fichier `mod_b.py` :

```
01: import mod_a
02:
03: def b_display() :
04:     print('World !')
05:     mod_a.a_display()
06:
07: b_display()
```

Les imports masqués

Si vous donnez à l'un de vos fichiers le nom d'un module existant et qu'il se trouve dans le chemin de recherche des modules, celui-ci sera chargé à la place du module standard. Prenons l'exemple du fichier `math.py` suivant :

```
01: import math
02:
03: print(math.randint(1, 20))
```

Vous obtiendrez l'erreur :

```
AttributeError: module 'math' has no attribute 'randint'
```

Ce n'est pas le module `math` que Python a chargé, mais votre fichier `math.py`. Le message d'erreur ne vous aidera pas forcément... Ce type d'erreur peut également survenir si vous ne conservez pas l'espace de noms lors d'un import. Voici l'exemple de la fonction `open()` utilisée pour ouvrir un fichier. Si l'on importe le contenu du module `os` par `from os import *`, et que l'on essaye de créer un fichier de manière classique avec `open('fichier.txt', 'w')`, on obtient le message :

```
TypeError: an integer is required (got type str)
```

En effet, le module `os` contient lui aussi une fonction `open()` qui est venue écraser la fonction `open()` du module standard et cette fonction n'accepte pas les mêmes paramètres... là encore il sera assez complexe de trouver l'erreur !

RECETTE 40

CRÉER une constante

L'OBJECTIF

Créer une constante, c'est-à-dire stocker une valeur qui ne pourra pas être modifiée.

LA SOLUTION

```
CONSTANTE = 12
```

COMMENTAIRES

Les constantes n'existent pas en Python ! En suivant les bonnes pratiques, il faut simplement nommer une variable en majuscules pour que le développeur sache qu'il s'agit d'une constante. Attention : le fait de nommer une variable en majuscules ne la protégera pas :

```
> VARIABLE = 15
> print(VARIABLE)
15
> VARIABLE = 2
> print(VARIABLE)
2
```

De plus, cette convention n'est malheureusement pas toujours suivie, comme c'est le cas de `pi` dans le module **math** :

```
> import math
> print(math.pi)
> print(math.pi)
3.141592653589793
```

Il est tout à fait possible de modifier `math.pi` :

```
> math.pi = 12
> print(math.pi)
12
```

CRÉER un attribut de classe

L'OBJECTIF

Créer un attribut qui sera commun à l'ensemble des instances d'une classe (chaque élément créé à partir de cette classe pourra lire et écrire dans un attribut commun).

LA SOLUTION

```
01: class Serie:
02:     nb = 0
03:
04:     def __init__(self, name):
05:         self.name = name
06:         Serie.nb += 1
07:
08:     def stock(self):
09:         print('Nombre total de séries définies :', Serie.nb)
10:
11:
12: if __name__ == '__main__':
13:     lost = Serie('Lost')
14:     got = Serie('Game of Throne')
15:     lost.stock()
16:     got.stock()
```

COMMENTAIRES

Un attribut de classe se définit à l'extérieur de toute méthode. Ici, `nb` en ligne 2 est un attribut de classe. Pour y accéder, il faut préfixer son nom par le nom de la classe (voir lignes 6 et 9).

La méthode `stock()` des lignes 8 et 9 n'utilise aucune autre méthode de la classe ni aucun attribut « standard » : elle aurait pu être définie en tant que méthode de classe.

NOTE

Il est possible d'accéder directement à l'attribut de la classe `Serie` depuis le programme principal en utilisant `Serie.nb` comme dans la ligne suivante :

```
...
17:     print(Serie.nb)
```

RECETTE 42

CRÉER UNE MÉTHODE de classe

L'OBJECTIF

Créer une méthode qui sera commune à l'ensemble des instances d'une classe et qui pourra être appelée même si aucune instance de cette classe n'a été créée.

LA SOLUTION

```
01: class Serie:
02:     nb = 0
03:
04:     def __init__(self, name):
05:         self.name = name
06:         Serie.nb += 1
07:
08:     @staticmethod
09:     def stock():
10:         print('Nombre total de séries définies :', Serie.nb)
11:
12:
13: if __name__ == '__main__':
14:     lost = Serie('Lost')
15:     got = Serie('Game of Throne')
16:     Serie.stock()
```

COMMENTAIRES

C'est le décorateur `@staticmethod` qui définit la méthode `stock()` comme méthode de classe. Notez l'absence du paramètre `self` en ligne 9. Cette méthode ne peut avoir accès qu'aux attributs de classe (contrairement à une méthode « standard » qui a accès à la fois aux attributs de classe et aux attributs « standards »).

Pour appeler une méthode de classe, il faut la préfixer par le nom de la classe (voir ligne 16).

NOTE

Il est également possible d'utiliser le décorateur `@classmethod`, mais il faudra utiliser en paramètre le nom de la classe (l'appel sera transparent et s'effectuera comme avec `@staticmethod`) :

```
...
08:     @classmethod
09:     def stock(cls):
10:         print('Nombre total de séries définies :', cls.nb)
...
```

CRÉER un singleton

L'OBJECTIF

Nous définissons une classe dont une seule instance peut être créée grâce au patron de conception (*design pattern*) Singleton.

LA SOLUTION

Une classe générique dans `Singleton.py` :

```
01: class Singleton(type):
02:     __instances = {}
03:     def __call__(cls, *args, **kwargs):
04:         if cls not in cls.__instances:
05:             cls.__instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
06:             return cls.__instances[cls]
07:         else:
08:             raise Exception('Il existe déjà une instance de cet objet!')
```

Une classe dans `MySingleton.py` qui définit un singleton contenant une valeur `value` :

```
01: from Singleton import Singleton
02:
03: class MySingleton(metaclass=Singleton):
04:     def __init__(self, val):
05:         self.value = val
06:
07: if __name__ == '__main__':
08:     try:
09:         single = MySingleton(5)
10:         print(single.value)
11:         single2 = MySingleton(7)
12:     except Exception as e:
13:         print(e)
```

COMMENTAIRES

Cette solution est générique dans la mesure où il vous est possible de créer des classes singleton à partir de n'importe quelle autre classe en ajoutant la meta-classe `Singleton` lors de la définition d'une nouvelle classe (voir ligne 3 de `MySingleton`). La classe `Singleton` hérite de `type` de manière à être une méta-classe. L'attribut de classe `__instances` permet de stocker les instances des classes créées. Si une instance est déjà présente dans ce dictionnaire, une exception est levée (ligne 8). Notez qu'il est possible d'obtenir un fonctionnement non bloquant en modifiant le code de `Singleton` comme suit :

```
...
04:         if cls not in cls.__instances:
05:             cls.__instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
06:         return cls.__instances[cls]
```

À ce moment-là, la création de plusieurs instances d'une même classe semblera fonctionner même si en réalité toutes les instances pointeront vers le même objet :

```
> from Singleton import Singleton
> single = MySingleton(5)
> single2 = MySingleton(7)
> print(single is single2)
True
> print(single2.value)
5
```

RECETTE 44

SURCHARGER un opérateur

L'OBJECTIF

Créer une pile d'entiers dans laquelle nous pourrons ajouter des éléments à l'aide de l'opérateur `+`.

LA SOLUTION

```
01: class Pile:
02:     def __init__(self):
03:         self.list = []
04:
05:     def __add__(self, value):
06:         self.list.append(value)
07:
08:     def __str__(self):
09:         return str(self.list)
10:
11: if __name__ == '__main__':
12:     p = Pile()
13:     p + 2
14:     p + 4
15:     print(p)
```

COMMENTAIRES

Pour surcharger les opérateurs d'un objet, il faut utiliser les méthodes prédéfinies : `__add__` pour `+`, `__mult__` pour `*`, `__lt__` pour `<`, etc. Ici nous avons pris quelques libertés avec l'opérateur `+`. En effet, celui-ci renvoie normalement le résultat d'une opération or notre méthode ne retourne aucune valeur et ne permet donc pas une utilisation pour initialiser/modifier une variable. Nous aurions pu utiliser l'opérateur de décalage à droite `>>` pour cette opération, celui-ci étant moins souvent employé, cela aurait été un peu moins choquant (mais tout autant étrange du point de vue de la signification des opérateurs). Méfiez-vous donc de la surcharge ! Faites en sorte que les opérateurs restent compréhensibles : à minima les opérateurs de tests restent des opérateurs de tests, etc. Si vous en décidez autrement, il faut documenter précisément votre code et surtout vous interroger sur le bienfondé de votre démarche, les cas semblables à la pile sont rares.

L'appel `p + 2` de la ligne 13 est interprété en machine comme `p.__add__(2)` qui appelle donc les lignes 5 et 6. Toute surcharge d'opérateur fonctionnera sur le même schéma. ▮

RECETTE 45

RENDRE un objet callable

L'OBJECTIF

Nous voulons pouvoir appeler une instance d'un objet comme s'il s'agissait d'une fonction et que cette dernière fournisse un résultat. Ici, la fonction renverra la somme d'un attribut de l'objet avec une valeur passée en paramètre lors de l'appel.

LA SOLUTION

```
01: class MonObjet:
02:     def __init__(self, valeur):
03:         self.valeur = valeur
04:
05:     def __call__(self, val):
06:         print('Objet appelé comme une fonction')
07:         print('Contenu de valeur :', self.valeur)
08:         return val + self.valeur
09:
10: if __name__ == '__main__':
11:     obj = MonObjet(10)
12:     somme = obj(5)
13:     print(somme)
```

COMMENTAIRES

C'est la méthode `__call__()` des lignes 5 à 8 qui rend l'objet *callable*, c'est-à-dire que l'instance `obj` créée en ligne 11 peut être appelée comme une fonction en ligne 12. Le résultat de ce code est le suivant :

```
Objet appelé comme une fonction
Contenu de valeur : 10
15
```

RECETTE 46

AJOUTER UN ATTRIBUT à une liste sans dégrader son comportement

L'OBJECTIF

Nous souhaitons ajouter un attribut permettant de nommer une liste (ou de la décrire, ou...) mais nous ne voulons surtout pas perdre le comportement de la liste.

LA SOLUTION

```
01: class List(list):
02:     def __new__(self, *args, **kwargs):
03:         return super(List, self).__new__(self, args, kwargs)
04:
05:     def __init__(self, *args, **kwargs):
06:         if len(args) == 1 and hasattr(args[0], '__iter__'):
07:             list.__init__(self, args[0])
08:         else:
09:             list.__init__(self, args)
10:         self.__dict__.update(kwargs)
11:
12:     def __call__(self, **kwargs):
13:         self.__dict__.update(kwargs)
14:         return self
15:
16: l = List(2, 3, 4, 5)
17: print(l)
18: l.name = 'ma_suite'
19: print(l)
20: print(l.name)
21: l(description='numbers')
22: print(l.description)
```

COMMENTAIRES

Nous créons une nouvelle classe de liste héritant de `list`. La méthode `__new__()`, rarement définie sauf pour les métaclasses comme c'est le cas ici, est le véritable constructeur d'une classe : c'est elle qui crée la liste en ligne 3. L'objet créé est automatiquement transmis à `__init__()` qui appelle la méthode `__init__()` de `list` en ligne 7 si un seul paramètre itérable est transmis ou en ligne 9 sinon. Ensuite, l'attribut `__dict__` est mis à jour avec les valeurs du dictionnaire `kwargs` (c'est ce qui permet de conserver d'éventuels attributs).

La méthode `__call__()` des lignes 12 à 14 permet de rendre l'objet *callable*, c'est-à-dire que s'il est employé comme une fonction alors la méthode `__call__()` sera appelée et renverra une valeur. C'est grâce à ce mécanisme que l'on peut écrire la ligne 21. ▮

RECETTE 47 CRÉER UN SÉLECTEUR de méthode

L'OBJECTIF

Nous voulons créer une méthode qui appelle une autre méthode d'après un paramètre qui lui est transmis.

LA SOLUTION

```
01: class Resume:
02:     def __init__(self, text):
03:         self.text = text
04:
05:     def display(self, format="text"):
06:         return getattr(self, 'output_' + format)()
07:
08:     def output_text(self):
09:         return self.text
10:
11:     def output_html(self):
12:         return '<p>' + self.text + '</p>'
13:
14: if __name__ == '__main__':
15:     t = Resume('Salut à tous!')
16:     print(t.display(format='html'))
```

COMMENTAIRES

Cette solution est possible grâce à la fonction `getattr()` qui appelle une méthode passée en second paramètre sur l'objet passé en premier paramètre (ligne 6). Notez la présence des parenthèses en fin de ligne ; sans celles-ci nous ne faisons que créer un pointeur sur fonction comme le montrent les quelques lignes suivantes :

```
> ma_liste = []
> ajoute = getattr(ma_liste, 'append')
> ma_liste
[]
> ajoute(2)
> ma_liste
[2]
```

Sans `getattr()` et en considérant l'ajout de nombreux autres formats, la solution aurait été beaucoup plus lourde et moins lisible avec l'utilisation d'une cascade de `if` :

```
...
05: def display(self, format="text"):
06:         if format == 'text':
07:             return self.ouput_text()
08:         elif format == 'html':
09:             return '<p>' + self.text + '</p>'
...
```

RECETTE 48

AJOUTER UNE MÉTHODE à une classe existante

L'OBJECTIF

Nous souhaitons ajouter une méthode à une classe existante sans avoir recours à l'héritage.
Nous prendrons pour exemple de classe existante la classe suivante contenue dans `Objet.py` :

```
01: class Objet:
02:     chaine = 'Classe Objet'
03:     def __init__(self, val):
04:         self.value = val
```

LA SOLUTION AVEC HÉRITAGE

```
01: from Objet import Objet
02:
03: class Objet_modifie(Objet):
04:     def __init__(self, val):
05:         super().__init__(val)
06:
07:     def display(self):
08:         print('Je suis une méthode!')
09:         print('J\'ai accès aux attributs : {}'.format(self.
value))
10:         print('Et aux attributs de classe : {}'.format(Objet.
chaine))
11:
12: if __name__ == '__main__':
13:     obj = Objet_modifie(5)
14:     obj.display()
```

LA SOLUTION SANS HÉRITAGE

```
01: from Objet import Objet
02:
03: def ma_methode(self):
```

```
04:     print('Je suis une méthode!')
05:     print('J\'ai accès aux attributs : {}'.format(self.value))
06:     print('Et aux attributs de classe : {}'.format(Objet.
chaine))
07:
08: Objet.display = ma_methode
09:
10: obj = Objet(5)
11: obj.display()
```

COMMENTAIRES

1 Avec héritage

On définit une nouvelle classe `Objet_modifie` dans `Objet_modifie.py`. Il faut penser à indiquer la relation d'héritage avec `Objet` (ligne 1) et à appeler le constructeur de la classe mère (ligne 5). Notez qu'en Python 2.7, la ligne 5 doit s'écrire :

```
Objet.__init__(self, val)
```

Il suffit ensuite de définir la nouvelle méthode (lignes 7 à 10) et d'utiliser non plus `Objet`, mais `Objet_modifie` (ligne 14).

2 Sans héritage

Il suffit de définir une fonction dont le premier paramètre sera l'élément `self` (l'objet courant) et d'écrire son contenu comme si nous nous trouvions dans la classe. Il faut ensuite lier cette méthode à la classe à l'aide d'un pointeur sur fonction (ligne 8), ce qui permet éventuellement de lui attribuer un autre nom.

L'utilisation de cette nouvelle méthode sera alors totalement transparente. Si vous souhaitez pouvoir utiliser votre `Objet` modifié dans plusieurs programmes, il faudra isoler dans un fichier votre modification, importer ce module plutôt que la classe. Voici comment faire sur la base de l'exemple précédent :

1. On crée un fichier `Objet_modif.py` :

```
01: from Objet import Objet
02:
03: def ma_methode(self):
04:     print('Je suis une méthode!')
05:     print('J\'ai accès aux attributs : {}'.format(self.value))
06:     print('Et aux attributs de classe : {}'.format(Objet.
chaine))
07:
08: Objet.display = ma_methode
```

2. On utilise notre `Objet` modifié :

```
01: import Objet_modif
02:
03: mon_objet = Objet_modif.Objet(5)
04: mon_objet.display()
```

3 INTERAGISSEZ AVEC VOS DONNÉES ET LES UTILISATEURS

À découvrir dans cette partie...

• BASE DE DONNÉES

Comment travailler avec une base de données SQLite3 : connexion, création d'une base, création d'une table puis lecture des données. Ces recettes pourront ensuite être utilisées sur d'autres SGBD tels que MySQL/MariaDB ou PostgreSQL, la technique restant la même. (Recettes 49 - 51)

• SYSTÈME

Appeler une commande shell depuis un script Python ou encore utiliser en Python des paramètres transmis depuis l'invite de commandes du shell, voici les recettes abordées dans cette partie. Et si vos codes effectuent de nombreux calculs, vous serez désormais capables de les faire exécuter en parallèle. (Recettes 52 - 54)

• WEB

Le Web est un élément omniprésent de notre société. Il est donc important de pouvoir y accéder pour récupérer et traiter des données ou bien fournir des services qui seront accessibles simplement à partir de différentes URL. (Recettes 55 - 61)

• INTERFACES

L'interface est l'élément qui va permettre à l'utilisateur de communiquer avec le programme. Cette interface peut être graphique ou simplement en mode console, mais son élaboration doit être toujours réfléchie pour simplifier au maximum les actions de l'utilisateur. (Recettes 62 - 67)

NOS RECETTES POUR...

49 Créer une base SQLite3 p.78
50 Insérer des données dans une base SQLite3 p.79
51 Lire les données d'une base SQLite3 p.80
52 Appeler une commande shell p.81
53 Récupérer les arguments de la ligne de commande p.82
54 Effectuer un calcul en utilisant plusieurs CPUs p.85
55 Charger une page html p.87
56 Traiter des données JSON p.88
57 Envoyer un email p.89
58 Créer un serveur XMLRPC p.90
59 Créer un serveur CGI p.92
60 Télécharger un fichier p.94
61 Créer une communication entre deux machines p.96
62 Écrire en couleur dans un terminal p.98
63 Modifier la position du curseur p.100
64 Créer un menu en mode texte p.102
65 Tracer et annoter une courbe p.103
66 Créer une animation d'attente en mode terminal p.104
67 Créer une interface graphique p.107

RECETTE 49

CRÉER UNE BASE SQLite3

L'OBJECTIF

Créer une base SQLite3 contenant la table suivante **Personnage** incluant les champs **Nom** et **Prenom**.

LA SOLUTION

```
01: import sqlite3
02:
03: try:
04:     db = sqlite3.connect('seriesBase.db')
05:
06:     cursor = db.cursor()
07:
08:     cursor.execute('''create table personnage (
09:         id integer primary key,
10:         nom text,
11:         prenom text)''')
12:
13:     db.commit()
14:
15:     cursor.close()
16:     db.close()
17: except:
18:     print('Une erreur est survenue lors de la création de la base')
19:     exit(1)
20:
21: print('La base de données a été créée')
```

COMMENTAIRES

L'accès aux bases de données suit toujours le même schéma :

1. Connexion à la base (ligne 4) ;
2. Création d'un curseur (ligne 6) ;
3. Utilisation du curseur pour les requêtes (lignes 8 à 11) ;
4. Fermeture du curseur (ligne 15) ;
5. Déconnexion de la base (ligne 16).

Ici, un **try**/**except** permet de gérer les erreurs de création (par exemple, si la base et la table existent déjà).

Notez que si, en ligne 4, vous indiquez la chaîne **':memory:'** en lieu et place d'un nom de fichier, alors votre base sera stockée en mémoire vive (les données ne seront pas stockées sur le disque). ▮

RECETTE 50 INSÉRER DES DONNÉES dans une base SQLite3

L'OBJECTIF

Insérer des données dans une base SQLite3 contenant la table **personnage** incluant les champs **nom** et **prenom**.

LA SOLUTION

```
01: import sqlite3
02:
03: persos = [('Skywalker', 'Luke'), ('Skywalker', 'Anakin'),
('Solo', 'Han')]
04:
05: try:
06:     db = sqlite3.connect('seriesBase.db')
07:
08:     cursor = db.cursor()
09:
10:     for data in persos:
11:         cursor.execute('insert into personnage (nom, prenom)
values (?, ?)', data)
12:
13:     db.commit()
14:
15:     cursor.close()
16:     db.close()
17: except:
18:     print('Une erreur est survenue lors de l\'insertion dans la
base')
19:     exit(1)
20:
21: print('Insertion effectuée')
```

COMMENTAIRES

Voir « Créer une base SQLite3 » pour les explications générales.

L'insertion est effectuée dans les lignes 10 et 11 avec l'ajout des données directement dans la fonction **execute()** (remplacement des **?** par les données de **data**) permettant de se protéger contre les injections SQL.

RECETTE 51

LIRE dans une base SQLite3

L'OBJECTIF

Lire des données dans une base SQLite3 contenant la table **personnage** incluant les champs **nom** et **prenom**.

LA SOLUTION

```
01: import sqlite3
02:
03: try:
04:     db = sqlite3.connect('seriesBase.db')
05:
06:     cursor = db.cursor()
07:
08:     cursor.execute('select * from personnage')
09:     for row in cursor:
10:         print('{}|{:15}|{:15}'.format(*row))
11:
12:     cursor.close()
13:     db.close()
14: except:
15:     print('Une erreur est survenue lors de la lecture dans la base')
16:     exit(1)
```

COMMENTAIRES

Voir « Créer une base SQLite3 » pour les explications générales. La lecture s'effectue en ligne 8 puis on parcourt les résultats contenus dans **cursor** dans les lignes 9 et 10. **row** est un tuple contenant les données lues dans la base (donc **id**, **nom** et **prenom**).

NOTE

Il est également possible d'avoir accès aux données en procédant de la manière suivante :

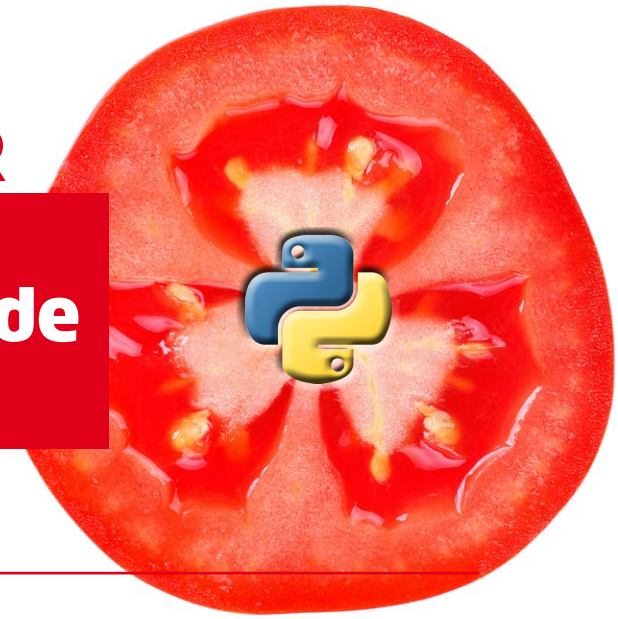
```
...
    while True:
        data = cursor.fetchone()
        print(data[1])
...
```

On obtient alors une liste qui contiendra ici en index 0 l'**id**, en index 1 le **nom** et en index 2 le **prenom**.

Une autre écriture similaire pourrait être :

```
...
    data = cursor.fetchall()
    for d in data:
        print(d)
...
```

APPELER une commande shell



L'OBJECTIF

Exécuter depuis un script Python une commande shell :

1 - une commande ne retournant rien (exemple avec **espeak**, le synthétiseur vocal) ;

2 - une commande retournant un résultat que l'on souhaite traiter en Python (exemple avec **ls -l** pour afficher la liste des fichiers).

LA SOLUTION

1 Appel d'une commande sans valeur de retour

```
01: import subprocess
02:
03: subprocess.call(['espeak', '-len', 'Hello'])
```

2 Traitement des données d'une commande

```
01: import subprocess
02:
03: pipe = subprocess.Popen("ls -l", shell=True, stdout=subprocess.
PIPE)
04:
05: while pipe.poll() is None:
06:     line = pipe.stdout.readline().decode('utf-8')
07:     print(line, end='')
```

COMMENTAIRES

subprocess.call() ne pourra pas fournir la valeur de retour de la commande appelée, mais permet de poursuivre l'exécution du script Python une fois la commande achevée.

subprocess.Popen() permet de lire les résultats de la commande sous la forme d'un flux. Il faut employer la méthode **poll()** pour déterminer si le processus fils est achevé et **stdout.readline()** pour lire les lignes de résultat qui sont retournées sous la forme d'une liste de **bytes** (d'où la conversion en **string** par **decode('utf-8')**). ▮

RECETTE 53

RÉCUPÉRER LES ARGUMENTS de la ligne de commandes

L'OBJECTIF

Récupérer les arguments transmis à un programme Python via la ligne de commandes. Nous supposerons que notre programme attend :

- `--output=filename` : le nom d'un fichier de sortie (optionnel) ;
- `-v` ou `--verbose` : indique le passage en mode verbeux (optionnel) ;
- `filename` : le nom d'un fichier en entrée.

LA SOLUTION

1 Récupération « manuelle »

```
01: import sys
02:
03: def syntax():
04:     print('Syntax: recuperer_arguments filename [-v|--
verbose] [--output=filename]')
05:     print('    filename : fichier en entrée')
06:     print('    -v | --verbose : mode verbeux')
07:     print('    --output=filename : fichier en sortie')
08:     exit(1)
09:
10: def cmd_line(args):
11:     nb_args = len(args)
12:     arguments = { 'input': None, 'verbose': False, 'output':
'default.txt' }
13:
14:     if nb_args < 2 or nb_args > 4:
15:         syntax()
16:
17:     for i in range(1, nb_args):
18:         if args[i] == '-v' or args[i] == '--verbose':
```

```
19:             arguments['verbose'] = True
20:         elif args[i][0] != '-':
21:             arguments['input'] = args[i]
22:         else:
23:             output = args[i].split('=')
24:             if output[0] != '--output':
25:                 syntax()
26:             arguments['output'] = output[1]
27:
28:     if arguments['input'] is None:
29:         syntax
30:
31:     return arguments
32:
33:
34: if __name__ == '__main__':
35:     args = cmd_line(sys.argv)
36:     print(args)
```

Exemple d'utilisation :

```
$ python3 recuperer_arguments.py mon_fichier.txt
{'output': 'default.txt', 'input': 'mon_fichier.txt', 'verbose': False}
$ python3 recuperer_arguments.py mon_fichier.txt --coucou
Syntax: recuperer_arguments filename [-v|--verbose] [--output=filename]
    filename : fichier en entrée
    -v | --verbose : mode verbeux
    --output=filename : fichier en sortie
$ python3 recuperer_arguments.py mon_fichier.txt -v --output=sortie.txt
{'verbose': True, 'input': 'mon_fichier.txt', 'output': 'sortie.txt'}
```

2 Récupération avec argparse

```
01: import sys
02: import argparse
03:
04: def syntax():
05:     print('Syntax: recuperer_arguments filename [-v|--
verbose] [--output=filename]')
06:     print('    filename : fichier en entrée')
07:     print('    -v | --verbose : mode verbeux')
08:     print('    --output=filename : fichier en sortie')
09:     exit(1)
10:
11: def cmd_line(args):
12:     arguments = { 'input': None, 'verbose': False, 'output':
'default.txt' }
13:     parser = argparse.ArgumentParser(description='Récupérer les
arguments de la ligne de commande')
14:     parser.add_argument('-v', '--verbose', dest='verbose',
default=False, action='store_true', help='mode verbeux')
15:     parser.add_argument('input', action='store', help='fichier
en entrée')
16:     parser.add_argument('--output', dest='output',
default='default.txt', help='fichier en sortie')
```

```
17:
18:     return parser.parse_args()
19:
20:
21: if __name__ == '__main__':
22:     args = cmd_line(sys.argv)
23:     print(args)
```

Exemple d'utilisation :

```
$ python3 recuperer_arguments.py mon_fichier.txt
Namespace(input='mon_fichier.txt', output='default.txt', verbose=False)
$ python3 recuperer_arguments.py mon_fichier.txt --coucou
usage: recuperer_arguments.py [-h] [-v] [--output OUTPUT] input
recuperer_arguments.py: error: unrecognized arguments: --coucou
$ python3 recuperer_arguments.py -h
usage: recuperer_arguments.py [-h] [-v] [--output OUTPUT] input

Récupérer les arguments de la ligne de commande

positional arguments:
  input            fichier en entrée

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbose    mode verbeux
  --output OUTPUT  fichier en sortie
$ python3 recuperer_arguments.py mon_fichier.txt -v --output=sortie.txt
Namespace(input='mon_fichier.txt', output='sortie.txt', verbose=True)
```

COMMENTAIRES

1 Récupération « manuelle »

Cette solution n'est normalement à employer que lorsque l'on souhaite traiter un petit nombre d'arguments sans options. L'exemple présenté montre qu'il est possible de manipuler des options, mais la solution n'est guère élégante et maintenable difficilement. Il faut en effet traiter manuellement les données de la liste `sys.argv` contenant les arguments de la ligne de commandes.

2 Récupération avec argparse

Argparse est le module standard de lecture des arguments de la ligne de commandes. Vous pourrez trouver bien d'autres modules plus ou moins exotiques dont le but est de simplifier la récupération des arguments.

Le fonctionnement d'argparse se décompose en trois étapes :

1 - Création du *parser* (ligne 13) ;

2 - Ajout des arguments à l'aide de `argparse.add_argument()` (lignes 14 à 16) ;

3 - Traitement des arguments à l'aide du *parser* en invoquant `parse_args()` sur ce dernier. ■

RECETTE 54

EFFECTUER UN CALCUL en utilisant plusieurs CPUs

L'OBJECTIF

Transmettre un calcul à une fonction et faire exécuter ce calcul à plusieurs CPUs. Nous utiliserons comme exemple la somme des carrés des entiers de `0` à `10 000 000`.

LA SOLUTION

Si vous n'avez pas installé `numpy` :

```
$ sudo pip3 install numpy
```

Le code est ensuite le suivant :

```
01: from multiprocessing import Process, Queue
02: from numpy import arange
03:
04:
05: def partsum(istart, istop, pile):
06:     psum = sum([i**2 for i in arange(istart,istop)])
07:     pile.put(psum)
08:
09:
10: if __name__ == "__main__":
11:     n = 10000000.
12:     parts = 2
13:     pile = Queue()
14:     threads = []
15:     sum_tot = 0
16:
17:     # Création des threads
18:     for i in range(parts):
19:         threads.append(Process(target=partsum, args = (i * n /
parts, (i + 1) * n / parts, pile)))
20:
21:     # Lancement des threads
22:     for i in range(parts):
23:         threads[i].start( )
24:
25:     # Attente de la fin du calcul
26:     for i in range(parts):
```

```
27:          threads[i].join( )
28:
29:      # Calcul du résultat
30:      for i in range(parts):
31:          sum_tot += pile.get()
32:
33:      print('Valeur finale de la somme : {}'.format(sum_tot))
```

COMMENTAIRES

La fonction **partsum()** des lignes 5 à 7 effectue le calcul de la somme des carrés entre **istart** et **istop** en créant une liste des carrés de **istart** à **istop** par compréhension de liste puis en sommant les éléments de la liste obtenue. Le résultat est placé dans une pile qui est un objet **Queue** (la fonction sera appelée par plusieurs CPUs et il faut stocker les résultats intermédiaires).

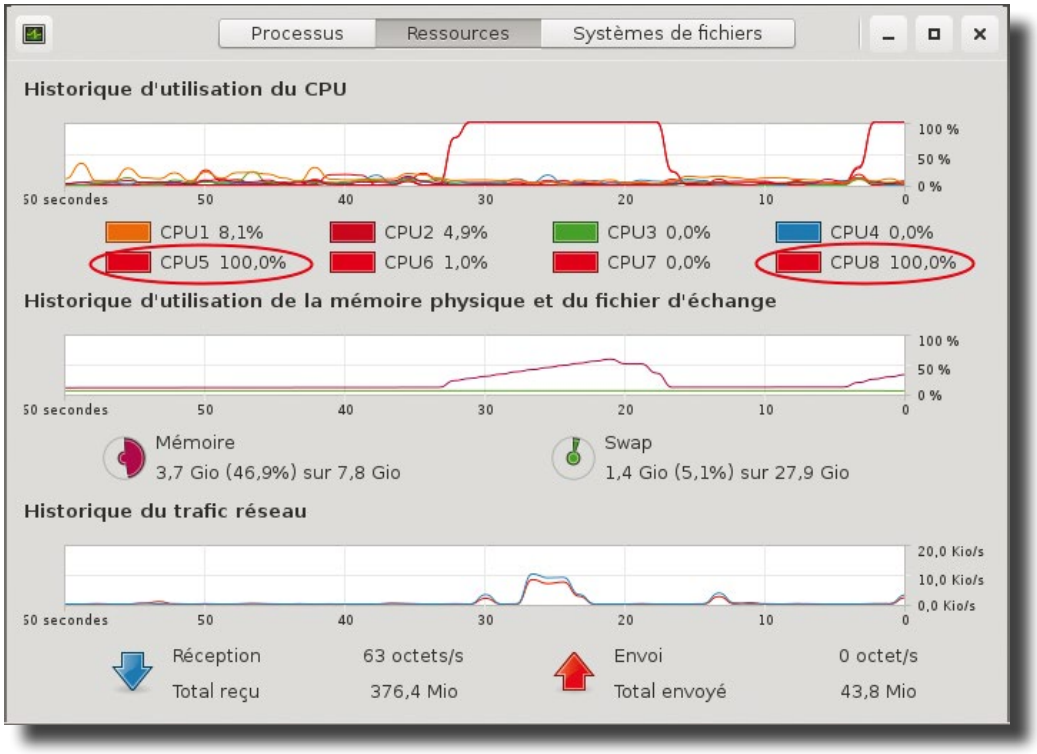
C'est en ligne 13 que la pile est créée. Le nombre de *threads* souhaités est défini en ligne 12, les *threads* seront stockés dans une liste **threads** et le résultat final sera calculé dans **sum_tot**.

Dans les lignes 18 et 19, on crée les *threads* en fonction du nombre (**parts**) indiqué. C'est la fonction **partsum** qui sera appelée (**target**) avec les paramètres **i*n/parts**, **(i+1)*n/parts** et **pile**.

Dans les lignes 22 et 23, les *threads* de la liste **threads** sont lancés à l'aide de la méthode **start()**.

Dans les lignes 26 et 27, on attend la fin des calculs, puis on calcule le résultat final en récupérant les données de la pile par **pile.get()** (lignes 30 et 31).

Sur la capture suivante vous pouvez voir l'effet d'un calcul lancé sur 2 CPUs. ■



RECETTE 55

CHARGER une page html

L'OBJECTIF

Charger sous forme de chaîne de caractères une page html depuis une URL donnée.

LA SOLUTION

```
01: from urllib.request import urlopen
02:
03: def getPage(url):
04:     with urlopen(url) as fic:
05:         data = fic.read().decode('utf-8')
06:     return data
07:
08: print(getPage('http://www.gnulinuxmag.com'))
```

COMMENTAIRES

La fonction **urlopen()** du module **urllib.request** ouvre la page passée en paramètre en créant une instance d'un objet **HTTPResponse** qui se comporte comme un descripteur de fichier. Il n'y a qu'à lire cet élément pour obtenir le contenu de la page.

Tout comme avec les descripteurs de fichier, les objets **HTTPResponse** disposent des méthodes **readline()** et **readlines()** pour lire une ligne ou l'ensemble des lignes présentées dans une liste. Ainsi, pour lire une seule ligne, il faut écrire :

```
data = fic.readline().decode('utf-8')
```

NOTE

Il faut signaler trois méthodes pouvant être utilisées sur un descripteur ouvert par **urlopen()** (dans notes exemple **fic**) :

- **getcode()** qui retourne le code HTTP de la réponse ;
- **geturl()** qui renvoie la véritable url ayant fourni l'information (utile en cas de redirection) ;
- **info()** qui retourne les en-têtes de la page.

RECETTE 56

TRAITER LES DONNÉES au format json

L'OBJECTIF

Un document **json** est structuré sous la forme d'un ensemble de paires clé/valeur à la manière d'un dictionnaire Python. Mais contrairement aux dictionnaires, un document json est une simple chaîne de caractères (une sérialisation d'un dictionnaire). Nous voulons passer d'une donnée « plate » json à un dictionnaire exploitable en Python.

Nous considérons que nos données json sont les suivantes (ces données sont en général obtenues par le biais d'un service web) :

```
{"series":{"Fargo":{"personnages": "Lorne Malvo, Lester Nygaard,
Molly Solverson, Gus Grimly"}, "Walking dead":{"personnages":
"Shane Walsh, Andrea Harrison, Carl Grimes, Rick Grimes, Glenn
Rhee"}}}
```

LA SOLUTION

```
01: import json
02:
03: data = '{"series":{"Fargo":{"personnages": "Lorne Malvo,
Lester Nygaard,  Molly Solverson, Gus Grimly"}, "Walking
dead":{"personnages": "Shane Walsh, Andrea Harrison, Carl Grimes,
Rick Grimes, Glenn Rhee"}}}'
04:
05: parsed_json = json.loads(data)
06: print(parsed_json['series']['Walking dead']['personnages'])
```

COMMENTAIRES

La fonction `loads()` du module json prend simplement une chaîne de caractères (au format JSON correct) et renvoie le dictionnaire qui lui est associé. Il ne vous reste plus qu'à manipuler le dictionnaire pour en extraire les données souhaitées. ▮

RECETTE 57

ENVOYER UN E-MAIL à partir d'un serveur SMTP distant

L'OBJECTIF

Envoyer un mail au format texte en utilisant un serveur SMTP distant.

LA SOLUTION

```
01: import smtplib
02:
03: from email.mime.text import MIMEText
04: from email.header import Header
05:
06: email = MIMEText('Ceci est un email', _charset='utf-8')
07:
08: email['Subject'] = Header('Un nouveau message !', 'utf-8')
09: email['From'] = 'adresse_emetteur@gnulinuxmag.com'
10: email['To'] = 'adresse_destinataire@gnulinuxmag.com'
11:
12: smtpserver = smtplib.SMTP('smtp.ma_box.com')
13: smtpserver.send_message(email)
14: smtpserver.quit()
```

COMMENTAIRES

Si les chaînes de caractères contiennent des caractères accentués, il faut penser à préciser l'encodage dans l'objet `MIMEText` et dans l'objet `Header` (lignes 6 et 8). Notez que la fonction `set_debuglevel(level)` (où `level` vaut `1` ou `2`) permet de déterminer le niveau de débuggage : `1` affichage des messages de debug pour tout message envoyé et reçu et `2` idem, mais avec ajout de l'horodatage.

Pour les connexions en SSL, il faudra employer `SMTP_SSL()` en lieu et place de `SMTP()`.

RECETTE 58

CRÉER UN SERVEUR XML-RPC

L'OBJECTIF

Créer un serveur XML-RPC proposant deux services :

- **hello** : pour s'assurer que le serveur est actif ;
- **getTime** : pour obtenir l'heure et la date.

LA SOLUTION

Le serveur se trouve dans **server.py** :

```
01: from xmlrpc.server import SimpleXMLRPCServer
02: import time
03:
04: server = SimpleXMLRPCServer(('localhost', 8000), logRequests=True)
05:
06: def hello():
07:     return 'Hello !'
08: server.register_function(hello)
09:
10: def getTime():
11:     return time.strftime('%H:%M:%S (%a %d %b %Y)', time.localtime())
12: server.register_function(getTime)
13:
14: try:
15:     print('Serveur actif sur le port 8000')
16:     server.serve_forever()
17: except KeyboardInterrupt:
18:     print('Arrêt du serveur')
```

Le client est dans **client.py** :

```
01: from xmlrpc.client import ServerProxy
02:
03: server = ServerProxy('http://localhost:8000')
04:
05: print(server.hello())
06: print(server.getTime())
```

Lancez le serveur :

```
$ python3 server.py
Serveur actif sur le port 8000
```

Exécutez ensuite **client.py** depuis un autre terminal.

```
$ python3 client.py
Hello !
16:00:20 (Mon 01 Aug 2016)
```

Pour interrompre le serveur, il faut appuyer sur <Ctrl> + <c>.

COMMENTAIRES

XML-RPC est un protocole RPC (*Remote procedure call*) permettant de définir simplement un service web (mise à disposition d'un ensemble de fonctions accessibles à travers le réseau quel que soit le système d'exploitation).

Le point important ici est de bien penser à enregistrer les fonctions auprès du serveur une fois que celles-ci ont été définies (méthode **register_function()** de **SimpleXMLRPCServer**) sous peine de voir apparaître des messages d'exceptions de la forme :

```
xmlrpc.client.Fault: <Fault 1: '<class \'Exception\'>:method
"portNawak" is not supported'>
```

Si vous le souhaitez, vous pouvez traiter ces erreurs proprement de la manière suivante :

```
...
02: import xmlrpc.client
...
12: try:
13:     print(server.portNawak())
14: except xmlrpc.client.Fault as e:
15:     print('Erreur !')
16:     print('  Code    : {}'.format(e.faultCode))
17:     print('  Message : {}'.format(e.faultString))
```

Le résultat est alors :

```
Hello !
06:59:51 (Fri 02 Sep 2016)
Erreur !
  Code    : 1
  Message : <class 'Exception'>:method "portNawak" is not supported
```

NOTE

Il est également possible de servir et recevoir des objets binaires. Par exemple, nous pourrions ajouter à **server.py** une méthode servant une image sous la forme d'un fichier png :

```
...
03: import xmlrpc.client
...
15: def getTux():
16:     with open('tux.png', 'rb') as image:
17:         return xmlrpc.client.Binary(image.read())
18: server.register_function(getTux)
...
```

La partie **client.py** associée permettant de charger et sauvegarder l'image sera :

```
...
08: with open('my_tux.png', 'wb') as image:
09:     image.write(server.getTux().data)
```

Ici le fichier **tux.png** sera lu, envoyé, reçu, et enfin stocké dans le fichier **my_tux.png**.

RECETTE 59

CRÉER UN SERVEUR CGI

L'OBJECTIF

Créer rapidement un petit serveur CGI capable de servir des documents html.

LA SOLUTION

Le serveur se trouve dans **server.py** :

```
01: import http.server
02:
03: PORT = 8000
04: server_address = ("", PORT)
05:
06: server = http.server.HTTPServer
07: handler = http.server.CGIHTTPRequestHandler
08: handler.cgi_directories = ["/"]
09: print("Serveur actif sur le port :", PORT)
10:
11: httpd = server(server_address, handler)
12: httpd.serve_forever()
```

Le programme de la page est dans **index.py** :

```
01: #!/usr/bin/python3
02:
03: import cgi
04:
05: form = cgi.FieldStorage()
06: print('Content-type: text/html; charset=utf-8\n')
07:
08: data = form.getvalue('serie')
09:
10: if data is not None:
11:     print('<div>Vous avez saisi la série : {}</div>'.format(data))
12:     if data.lower() == 'like-moi!':
```

```
13:         print('Bon choix!')
14:
15: file = open('index.html', 'r')
16: html = file.read()
17: file.close()
18:
19: print(html)
```

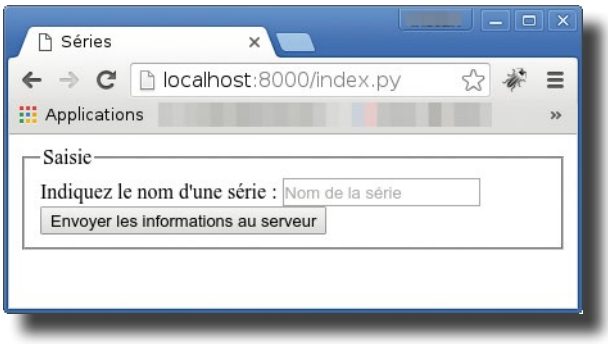
Et enfin le *template* dans **index.html** :

```
01: <!doctype html>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Séries</title>
06:   </head>
07:
08:   <body>
09:     <form action="/index.py" method="post">
10:       <fieldset>
11:         <legend>Saisie</legend>
12:         <label for="serie">Indiquez le nom d'une série :</label>
13:         <input type="text" name="serie" id="serie"
placeholder="Nom de la série" /></br>
14:         <input type="submit" name="send" value="Envoyer les
informations au serveur">
15:       </filedset>
16:     </form>
17:   </body>
18: </html>
```

Lancez le serveur :

```
$ python3 server.py
Serveur actif sur le port : 8000
```

Ouvrez ensuite l'URL **localhost:8000/index.py** dans un navigateur.



COMMENTAIRES

Prenez bien garde de rendre le fichier **index.py** exécutable et d'indiquer sur sa première ligne le chemin vers l'interpréteur (qui n'est pas forcément le même sur votre système que dans l'exemple donné ci-dessus). Sous Windows, cette ligne, nommée *shebang*, sera la suivante :

```
#!python3
```

Si le fichier **index.py** n'est pas exécutable, vous verrez apparaître une erreur 503 dans votre navigateur. C'est ce fichier qui lit **index.html** et l'affiche (lignes 15 à 19). Les données de la ligne **<input type="text" name="serie" id="serie" placeholder="Nom de la série" />** sont récupérées à l'aide de la méthode **getvalue('serie')** en ligne 8 où **serie** correspond au nom donné au champ dans le document html.

RECETTE 60

TÉLÉCHARGER UN FICHIER depuis une URL

L'OBJECTIF

Télécharger un fichier sur Internet en connaissant son URL.

LA SOLUTION

```
01: from urllib.request import urlretrieve
02:
03: def progress(blk_reads, blk_size, total_size):
04:     if not blk_reads:
05:         print('Début du téléchargement')
06:         return None
07:
08:     if total_size < 0:
09:         # Taille totale du fichier inconnue
10:         print('{:8d} / ??? octets téléchargés'.format(blk_
reads * blk_size))
11:     else:
12:         # Taille totale du fichier connue
13:         print('{:8d} / {:8d} octets téléchargés'.format(blk_
reads * blk_size, total_size))
14:     return None
15:
16: if __name__ == '__main__':
17:     filename, headers = urlretrieve('http://le_serveur.com/un_
exemple_d_image.png', reporthook=progress)
18:     print('Fin du téléchargement')
19:     print('Le fichier se trouve dans :', filename)
```

COMMENTAIRES

La fonction **`urlretrieve()`** d'**`urllib.request`** permet de récupérer un fichier depuis une URL donnée. En indiquant un pointeur vers une fonction au paramètre **`reporthook`**, il est possible de suivre l'évolution du téléchargement. Cette fonction doit nécessairement accepter trois paramètres qui seront le nombre de blocs lus, la taille des blocs en octets et la taille totale du fichier.

Il est possible que la taille totale du fichier soit inconnue. Dans ce cas, le troisième paramètre de la fonction de rappel du **`reporthook`** contiendra une valeur négative (les lignes 8 à 10 traitent ce cas).

Voici un exemple de résultat obtenu en téléchargeant une image :

```
Début du téléchargement
    8192 /   185487 octets téléchargés
   16384 /   185487 octets téléchargés
   24576 /   185487 octets téléchargés
   32768 /   185487 octets téléchargés
   40960 /   185487 octets téléchargés
   49152 /   185487 octets téléchargés
   57344 /   185487 octets téléchargés
   65536 /   185487 octets téléchargés
   73728 /   185487 octets téléchargés
   81920 /   185487 octets téléchargés
   90112 /   185487 octets téléchargés
   98304 /   185487 octets téléchargés
  106496 /   185487 octets téléchargés
  114688 /   185487 octets téléchargés
  122880 /   185487 octets téléchargés
  131072 /   185487 octets téléchargés
  139264 /   185487 octets téléchargés
  147456 /   185487 octets téléchargés
  155648 /   185487 octets téléchargés
  163840 /   185487 octets téléchargés
  172032 /   185487 octets téléchargés
  180224 /   185487 octets téléchargés
  188416 /   185487 octets téléchargés
Fin du téléchargement
Le fichier se trouve dans : /tmp/tmpsjxijym2
```

NOTE

Si vous souhaitez accéder aux en-têtes du fichier transféré, il vous suffit d'utiliser la seconde valeur renvoyée par `urlretrieve()`. Dans notre exemple, il s'agit de la variable `headers` de la ligne 17. Voici ce qu'elle contient dans le cas de **http://www.gnu.org/licenses/gpl.html** :

```
Date: Mon, 04 July 2016 12:30:59 GMT
Server: Apache/2.4.7
Content-Location: gpl-3.0.html
Vary: negotiate,accept,accept-language,Accept-Encoding
TCN: choice
Access-Control-Allow-Origin: (null)
Accept-Ranges: bytes
Cache-Control: max-age=0
Expires: Mon, 04 July 2016 12:30:59 GMT
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
Content-Language: en
```

RECETTE 61

FAIRE COMMUNIQUER deux machines

L'OBJECTIF

Établir une communication entre deux machines (un serveur et un client) à l'aide des *sockets*.

LA SOLUTION

Il y a deux éléments : le serveur et le client. Voici **server.py** :

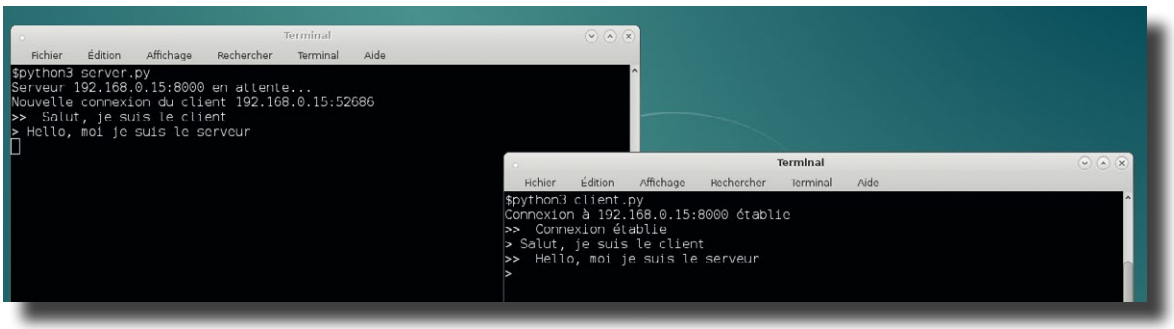
```
01: import socket
02:
03: HOST = '192.168.0.15'
04: PORT = 8000
05:
06: server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
07:
08: try:
09:     server.bind((HOST, PORT))
10: except socket.error:
11:     print('Impossible de lier le socket à l'adresse {}:{}'.format(HOST, PORT))
12:     exit(1)
13:
14: print('Serveur {}:{} en attente...'.format(HOST, PORT))
15:
16: while True:
17:     server.listen(2)
18:     connexion, address = server.accept()
19:     print('Nouvelle connexion du client {}:{}'.format(address[0], address[1]))
20:
21:     connexion.send('Connexion établie'.encode('utf-8'))
22:     while True:
23:         msg = connexion.recv(1024).decode('utf-8')
24:         if msg.upper() == 'QUIT':
25:             break
26:         print('>> ', msg)
27:         msgToSend = input('> ')
28:         connexion.send(msgToSend.encode('utf-8'))
29:
30:     connexion.send('quit'.encode('utf-8'))
31:     connexion.close()
32:     print('Fermeture de la connexion')
33:     print('Attente d'une nouvelle connexion...')
```

Le code du client **client.py** est le suivant :

```
01: import socket
02:
03: HOST = '192.168.0.15'
04: PORT = 8000
```

```
05:
06: connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
07:
08: try:
09:     connexion.connect((HOST, PORT))
10: except socket.error:
11:     print('Connexion impossible à {}:{}'.format(HOST, PORT))
12:     exit(1)
13:
14: print('Connexion à {}:{} établie'.format(HOST, PORT))
15:
16: while True:
17:     msg = connexion.recv(1024).decode('utf-8')
18:     if msg.upper() == 'QUIT':
19:         break
20:     print('>> ', msg)
21:     msgToSend = input('> ')
22:     connexion.send(msgToSend.encode('utf-8'))
23:
24: print('Fermeture de la connexion')
25: connexion.close()
```

En lançant **server.py** et **client.py** dans deux terminaux distincts, on peut dialoguer comme le montre la figure suivante :



COMMENTAIRES

1 Le serveur

Le serveur effectue les actions suivantes :

1 - Création du *socket* en ligne 6 ;

2 - Liaison du socket sur **HOST**:**PORT** dans les lignes 8 à 12 ;

3 - Attente de connexion d'un client en ligne 17 ;

4 - Création de la connexion avec le client dans les lignes 18 et 19 ;

5 - Communication avec le client par suite de réception/émission de messages dans les lignes 16 à 22 ;

6 - Fermeture de la connexion dans les lignes 30 à 32.

Le serveur ne s'arrête que par <Ctrl> + <c>, car nous avons fait le choix d'attendre une nouvelle connexion après chaque communication (ligne 33 et bloc `while True`).

2 Le client

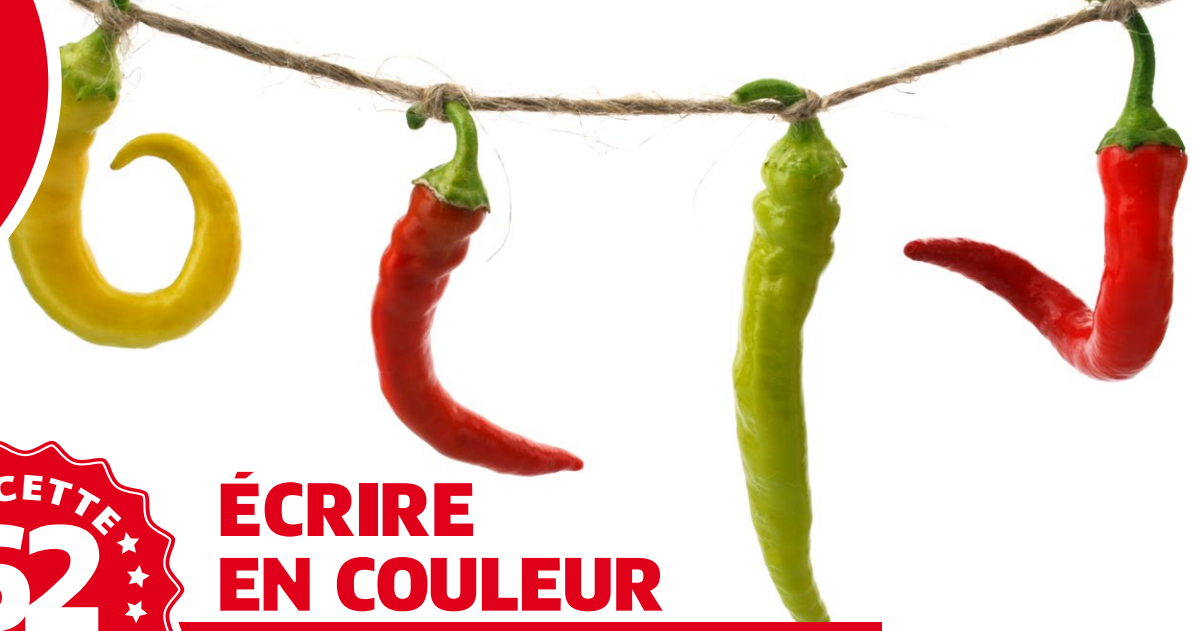
Le client à un comportement similaire au serveur :

1 - Création du *socket* en ligne 6 ;

2 - Connexion au serveur dans les lignes 8 à 12 ;

3 - Communication avec le serveur par suite de réception/émission de messages dans les lignes 16 à 22 ;

4 - Fermeture de la connexion en ligne 25.



RECETTE 62

ÉCRIRE EN COULEUR dans un terminal

L'OBJECTIF

Écrire des informations en couleur sur le terminal.

LA SOLUTION

```
01: BLACK   = 0
02: RED     = 1
03: GREEN   = 2
04: DEFAULT = 9
05:
06: def color(code, background=False):
07:     if background:
08:         print('\033[4{}m'.format(code), end='')
09:     else:
10:         print('\033[3{}m'.format(code), end='')
11:
12: color(GREEN)
13: print('En vert')
14: color(BLACK)
15: color(GREEN, background=True)
16: print('En noir sur fond vert')
17: color(DEFAULT)
18: color(DEFAULT, background=True)
19: print('Retour à la normale')
```

COMMENTAIRES

1 Les séquences d'échappement ANSI

Les terminaux sont capables d'interpréter des séquences de caractères appelées séquences d'échappement ou séquences ANSI. Ces séquences sont toutes construites de la même manière, à savoir :

- une séquence d'initialisation du code appelée CSI pour *Control Sequence Introducer* (Séquence de Contrôle Initiale). C'est le `\033[` utilisé dans le code précédent ;
- un ou plusieurs paramètres. Il s'agit de codes comme ceux employés dans notre cas avec des codes SGR (pour *Select Graphic Rendition*) ;
- une commande qui va utiliser les paramètres précédents et qui va terminer la séquence. Nous avons utilisé ici la commande `m` qui permet de régler l'affichage.

2 Les codes SGR

Nous avons employé deux types de codes SGR : un pour changer la couleur du texte et un autre pour changer la couleur du fond. On peut noter également un code pour écrire en gras et un autre en surligné comme le montre le tableau récapitulatif suivant :

Code	Effet
\033[0m	Remise à zéro de tous les paramètres (retour au style initial)
\033[1m	Augmente l'intensité des couleurs
\033[2m	Diminue l'intensité des couleurs
\033[4m	Affichage en surligné
\033[3...m	Changement de la couleur d'affichage (les points de suspension indiquent un code couleur)
\033[4...m	Changement de la couleur de fond du texte (les points de suspension indiquent un code couleur)

3 Les codes de couleur

Pour les commandes de changement de couleur, le tableau suivant résume les possibles :

Code	Couleur
0	Noir
1	Rouge
2	Vert
3	Jaune
4	Bleu
5	Magenta
6	Cyan
7	Blanc
9	Couleur par défaut

NOTE

Pour effacer le terminal, vous pouvez employer le code `\033[2J`.

Notez également que `\033` correspond au caractère ASCII de code `27`. Vous pouvez donc écrire vos appels aux codes SGR sous la forme :

```
print(chr(27) + '[2J')
```

RECETTE 63

MODIFIER LA POSITION DU CURSEUR dans un terminal

L'OBJECTIF

Écrire une phrase et corriger un mot en écrivant par-dessus un autre mot.

LA SOLUTION

```
01: UP    = 'A'
02: DOWN  = 'B'
03: RIGHT = 'C'
04: LEFT  = 'D'
05:
06: def move(distance, direction):
07:     print('\033[{}{}'.format(distance, direction), end='')
08:
09: print('Il fait mauvais !', end='')
10: move(9, LEFT)
11: print('beau !   ')
```

COMMENTAIRES

1 Les séquences d'échappement ANSI

Ici encore, nous utilisons les séquences d'échappement ANSI (voir « Écrire en couleur dans un terminal »). Les commandes **nA**, **nB**, **nC**, et **nD** permettent respectivement de déplacer le curseur de **n** caractères vers le haut, le bas, la droite (avant) ou la gauche (arrière).

2 Les sauts de lignes

Prenez garde de bien désactiver le retour à la ligne (`end=''`) pour effectuer un retour arrière ! Sinon vous serez sur une nouvelle ligne et il vous serait impossible de reculer.

3 Penser à effacer les caractères non masqués

Avec un retour arrière les caractères ne sont pas effacés, donc si votre modification comporte moins de caractères, il faudra ajouter des espaces pour les masquer (voir ligne 11). Une autre solution est d'utiliser la commande `\033[K` qui efface les caractères depuis le curseur jusqu'à la fin de la ligne.

4 Commandes supplémentaires

Il faut savoir qu'il est également possible de déplacer directement le curseur en ligne `n`, colonne `m` par `\033[n;mH`, et que l'on peut enregistrer la position courante du curseur par `\033[s` et la restituer par `\033[u`. Notez que ces deux derniers codes ne fonctionnent malheureusement pas dans bon nombre de terminaux.

NOTE

La création de fonctions permettant de manipuler les différents codes est loin d'être superflue ! Elle permet de :

- structurer le code ;
- obtenir une meilleure maintenabilité (voire une maintenabilité du code tout court) ;
- gérer plus simplement les différents codes ;
- réaliser des appels simples et lisibles.

5 Résumé des commandes de manipulation de la position du curseur

Commande	Effet
`\033[nA`	Déplacement de `n` caractères vers le haut
`\033[nB`	Déplacement de `n` caractères vers le bas
`\033[nC`	Déplacement de `n` caractères vers la droite
`\033[nD`	Déplacement de `n` caractères vers la gauche
`\033[n;mH`	Positionnement du curseur en ligne `n`, colonne `m`
`\033[s`	Enregistrement de la position courante (quand disponible)
`\033[u`	Restitution de la position courante (quand disponible)

RECETTE 64

CRÉER UN MENU en mode texte

L'OBJECTIF

Afficher un menu à l'écran et renvoyer le choix de l'utilisateur si ce choix est autorisé. Dans le cas contraire, gérer proprement les erreurs et redemander à l'utilisateur de sélectionner une entrée.

LA SOLUTION

```
01: def menu(entries):
02:     max = len(entries) + 1
03:     escape = False
04:     while not escape:
05:         print('MENU :')
06:         for num, entry in enumerate(entries):
07:             print(' {:2}. {}'.format(num + 1, entry))
08:         try:
09:             choice = int(input('Votre choix : '))
10:         except ValueError:
11:             print('Vous devez saisir un entier !')
12:         except KeyboardInterrupt:
13:             print('\nAbandon')
14:             exit(1)
15:         else:
16:             if choice < 1 or choice >= max:
17:                 print('Entrée inconnue !')
18:             else:
19:                 escape = True
20:     return choice
21:
22:
23: c = menu(('Démarrer', 'Exécuter une commande', 'Quitter'))
```

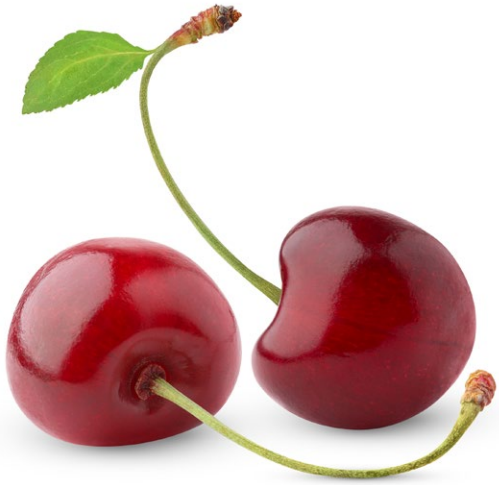
COMMENTAIRES

1 L'affichage du menu

Pour afficher le menu, nous parcourons une liste d'entrées fournie en paramètre (ligne 6). Grâce à **`enumerate`**, nous associons une valeur entière à chaque entrée (nous ajouterons **`1`** pour ne pas commencer à **`0`**). Le formatage de la ligne 7 permet de présenter proprement les informations. Par exemple, **`{:2}`** réserve deux caractères pour écrire le nombre correspondant aux entrées : même s'il y a plus de neuf entrées, les lignes seront alignées.

2 La gestion des exceptions

La conversion en entier de la ligne 9 peut provoquer une exception du type **`ValueError`** si la conversion est impossible. Ce cas est traité dans les lignes 10 et 11. Si l'utilisateur interrompt le traitement en appuyant sur <Ctrl> + <C>, une exception **`KeyboardInterrupt`** est lancée. Nous traitons ce cas dans les lignes 12 à 14. Enfin, si aucune erreur n'est survenue, nous exécutons le code des lignes 15 à 19 permettant de tester si le choix de l'utilisateur est valide. Si c'est le cas, la variable **`escape`** passe à **`True`** (ligne 19) mettant fin à la boucle **`while`** commençant en ligne 4, affichant le menu et questionnant l'utilisateur tant que celui-ci n'a pas fourni de réponse valide. ▮



RECETTE 65 TRACER ET ANNOTER une courbe

L'OBJECTIF

Tracer la courbe de $\sin(2\pi x)e^{-x}$ et ses asymptotes, donner sa légende et annoter le point `(5, 0)`.

LA SOLUTION

Il faut installer **matplotlib** et **numpy** (paquets `python3-matplotlib` et `python3-numpy` sous Debian) :

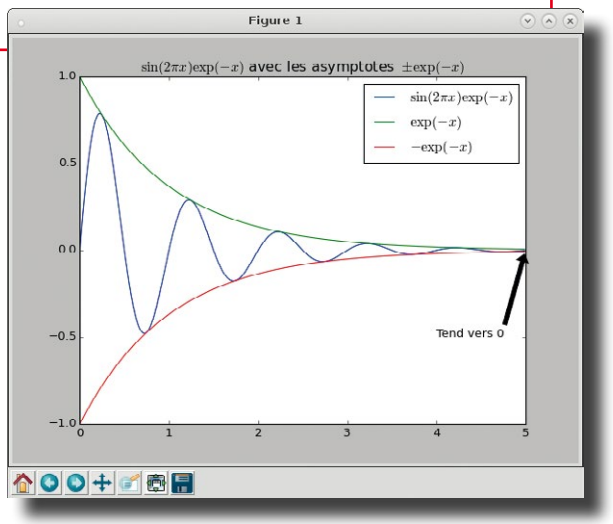
```
$ sudo pip3 install pylab
```

Sous Windows, vous pouvez utiliser le dépôt non officiel **http://www.lfd.uci.edu/~gohlke/pythonlibs/**. Si vous utilisez beaucoup de modules scientifiques, le plus simple est d'installer **Python(x,y)** qui comprend déjà de nombreux modules de calcul : **http://python-xy.github.io/**.

Le code répondant au problème est le suivant :

```
01: import pylab as pl
02: import numpy as np
03:
04: x = np.arange(0., 5., .01)
05: y = [np.sin(2*np.pi*xx) * np.exp(-xx) for xx in x]
06:
07: pl.plot(x, y, label=r'$\sin(2\pi x)\exp(-x)$')
08: pl.plot(x, np.exp(-x), label=r'$\exp(-x)$')
09: pl.plot(x, -np.exp(-x), label=r'$-\exp(-x)$')
10: pl.title(r'$\sin(2\pi x)\exp(-x)$ avec les asymptotes $\pm\exp(-x)$')
11: pl.legend()
12: pl.annotate('Tend vers 0', xy=(5, 0), xytext=(4,-.5), arrowprops=dict
(facecolor='black', shrink=0.05))
13:
14: pl.show()
```

On obtient la figure suivante ci-contre.



COMMENTAIRES

Matplotlib permet d'intégrer du code **LaTeX** dans les chaînes de caractères qui sont employées dans un graphe. Pour cela, on définit une chaîne « brute » (ou *rawstring*) en la préfixant par la lettre `r` : les caractères spéciaux ne seront ainsi plus évalués (`\n` pourrait poser problème avec un élément LaTeX dont le nom commencerait par `n`...). C'est la méthode `annotate()` qui permet de définir une annotation sous la forme d'une flèche pointant vers un point donné. Notez que les propriétés de cette flèche (ligne 12) sont données sous la forme d'un dictionnaire `dict(facecolor='black', shrink=0.05)`, mais que l'écriture `{'facecolor': 'black', 'shrink': 0.05}` est équivalente. ■

RECETTE 66

CRÉER UNE ANIMATION D'ATTENTE en mode console

L'OBJECTIF

Lorsqu'une tâche s'exécute, elle peut être plus ou moins longue et il faut donc signaler à l'utilisateur que le programme est toujours « vivant ». Pour cela, on peut afficher une petite animation (une barre qui tournoie) ou bien une barre de progression.

LA SOLUTION SIMPLE

```
01: import time
02:
03: class Waiting:
04:     states = ('/', '-', '\\', '|')
05:     step = 0
06:     def __init__(self):
07:         self.next()
08:
09:     def next(self):
10:         print('Work in progress... {}'.format(
11:                   Waiting.states[Waiting.step % 4]), end='\r')
12:         Waiting.step += 1
13:
14: if __name__ == '__main__':
15:     progressBar = Waiting()
16:     for i in range(10):
17:         progressBar.next()
18:         time.sleep(1)
19:
20:     print()
```

COMMENTAIRES SUR LA SOLUTION SIMPLE

1 La classe Waiting

Nous définissons une classe **Waiting** comportant deux attributs de classes (ou attributs statiques) :

- **states** : contient la liste des caractères à afficher pour donner l'illusion d'une barre tournoyante ;
- **step** : représente le nombre de fois où la classe est appelée via la méthode **next()**.

La méthode **next()** affiche une phrase et la barre d'attente. C'est l'utilisation de **\r** qui permet de rester sur la même ligne (voir lignes 10 et 11).

2 Utilisation

Pour utiliser cette barre, il suffit de créer une instance de **Waiting** et d'appeler régulièrement la méthode **next()**. Ici, pour simuler un temps de travail de la machine, nous avons utilisé la fonction **sleep()** du module **time** permettant d'attendre une seconde avant de poursuivre le traitement (voir ligne 18). Au lancement, on obtient :

```
Work in progress... /
```

LA SOLUTION PLUS COMPLÈTE

```
01: import time
02:
03: class ProgressBar:
04:     def __init__(self, msg = 'Work in progress'):
05:         self.current = 0
06:         self.msg = msg
07:
08:     def add(self, increment):
09:         self.current += increment
10:         if self.current > 100:
11:             self.current = 100
12:
13:     def show(self):
14:         text = '{} : [{}%]'.format(self.msg, self.current)
15:         if self.isFinished():
16:             text += '\n'
17:         else:
18:             text += '\r'
19:         print(text, end='')
20:
21:     def isFinished(self):
22:         return self.current == 100
23:
24:
25: if __name__ == '__main__':
```

```
26:     bar = ProgressBar()
27:
28:     while not bar.isFinished():
29:         bar.show()
30:         time.sleep(1)
31:         bar.add(5)
```

COMMENTAIRES SUR LA SOLUTION PLUS COMPLÈTE

1 La classe ProgressBar

La classe **ProgressBar** contient un attribut **current** (initialisé à **0**) permettant de conserver le pourcentage de la tâche effectuée (donc sa valeur ne peut excéder **100**) et un attribut **msg** autorisant la modification du texte à afficher. La méthode **add()** permet de faire progresser la barre en indiquant un pourcentage de la tâche qui a été réalisé (en plus de celui existant). Les lignes 10 et 11 interdisent tout dépassement de la valeur **100** pour l'attribut **current**.

Pour afficher la barre, il faudra employer la méthode **show()** des lignes 13 à 19 qui utilise comme précédemment le caractère **\r** pour rester sur la même ligne dans le terminal.

Enfin, la méthode **isFinished()** indique si la tâche est achevée (**current** vaut alors **100**). Nous aurions pu écrire cette méthode sous une forme plus développée :

```
def isFinished(self):
    if self.current == 100:
        return True
    else:
        return False
```

Les deux codes fournissent exactement le même résultat. **self.current == 100** est un test qui renvoie une valeur booléenne : **True** si **self.current** vaut **100** et **False** sinon. On peut donc directement retourner cette valeur sans effectuer un test qui renverra… la même chose.

2 Utilisation

Il faut créer une instance de **ProgressBar** puis incrémenter le pourcentage de la tâche effectué grâce à la méthode **add()**, l'affichage se faisant par **show()** (voir lignes 25 à 31). Au lancement, on obtient :

```
Work in progress : [10%]
```

RECETTE 67

CRÉER UNE INTERFACE GRAPHIQUE en Tk

L'OBJECTIF

Créer une petite calculatrice graphique acceptant deux flottants et permettant de réaliser l'une des quatre opérations élémentaires.

LA SOLUTION

```
01: from tkinter import *
02:
03: class App:
04:
05:     def operation(self, sign):
06:         self.label_sign_op.config(text=sign)
07:         try:
08:             result = eval('a' + sign + 'b', {'a' :self.var_a.
get(), 'b' :  self.var_b.get()})
09:             self.label_result.config(text=str(result))
10:         except Exception as e:
11:             error = Toplevel()
12:             label_error = Label(error, text='Error : {}'.
format(e))
13:             label_error.pack(fill=X)
14:             btn_quit = Button(error, text='Ok', command=error.
destroy)
15:             btn_quit.pack(fill=X)
16:
17:     def __init__(self, master_ui):
18:         master_ui.title('Calculatrice')
19:
20:         label_a = Label(master_ui, text='A')
21:         label_a.grid(sticky=W, column=0, row=0, columnspan=3)
22:
23:         label_b = Label(master_ui, text='B')
```

```
24:         label_b.grid(sticky=W, column=0, row=1, columnspan=3)
25:
26:         label_a_op = Label(master_ui, text='A')
27:         label_a_op.grid(sticky=W, column=0, row=2)
28:
29:         self.label_sign_op = Label(master_ui, text=' ')
30:         self.label_sign_op.grid(sticky=W, column=1, row=2)
31:
32:         label_b_op = Label(master_ui, text='B')
33:         label_b_op.grid(sticky=W, column=2, row=2)
34:
35:         self.var_a = DoubleVar()
36:         self.entry_a = Entry(master_ui, textvariable=self.
var_a)
37:         self.entry_a.grid(sticky=W+E, column=3, row=0)
38:
39:         self.var_b = DoubleVar()
40:         self.entry_b = Entry(master_ui, textvariable=self.
var_b)
41:         self.entry_b.grid(sticky=W+E, column=3, row=1)
42:
43:         label_action = Label(master_ui, text='Action')
44:         label_action.grid(sticky=N+S+W+E, column=4, row=0,
columnspan=2)
45:
46:         self.label_result = Label(master_ui, text='')
47:         self.label_result.grid(sticky=E, column=3, row=2)
48:
49:         btn_plus = Button(master_ui, text='+', command=lambda
sign='+' : self.operation(sign))
50:         btn_plus.grid(sticky=N+S+W+E, column=4, row=1)
51:
52:         btn_minus = Button(master_ui, text='-', command=lambda
sign='-' : self.operation(sign))
53:         btn_minus.grid(sticky=N+S+W+E, column=5, row=1)
54:
55:         btn_multiply = Button(master_ui, text='*',
command=lambda sign='*' : self.operation(sign))
56:         btn_multiply.grid(sticky=N+S+W+E, column=4, row=2)
57:
58:         btn_divide = Button(master_ui, text='/', command=lambda
sign='/' : self.operation(sign))
59:         btn_divide.grid(sticky=N+S+W+E, column=5, row=2)
60:
61: if __name__ == '__main__':
62:     root = Tk()
63:     app = App(root)
64:     root.mainloop()
```

Le résultat est visible sur la figure suivante.

COMMENTAIRES

NOTE

Il est possible que vous obteniez l'erreur suivante sous GNU/Linux :

```
    import _tkinter # If this fails your Python may not be
configured for Tk
ImportError: No module named '_tkinter'
```

Cela signifie qu'il vous manque le paquet **python3-tk** que l'on peut installer sur les systèmes basés sur Debian par :

```
$ sudo apt install python3-tk
```

On importe tout le module **tkinter** en ligne 1 par **from tkinter import *** : les bonnes pratiques Python nous indiquent de préférer un import conservant l'espace de nom, mais il y a ici de nombreux objets donc nous utilisons cette syntaxe en toute connaissance de cause. La classe App va définir le comportement de notre fenêtre graphique qui est lancée dans les lignes 62 à 64 en créant une instance de Tk, en y liant notre interface puis en déclenchant une boucle infinie permettant de récupérer les différents événements.

Le constructeur de la classe **App** dans les lignes 17 à 59 définit la fenêtre graphique :

- le titre en ligne 18 ;
- les différents labels dans les lignes 20 à 33 (le positionnement des éléments dans la fenêtre graphique se fait à l'aide d'une grille et le positionnement à l'intérieur d'une cellule est déterminé par la valeur de **sticky**) ;
- les champs de texte dans les lignes 35 à 41 (il faut définir un objet spécial pour récupérer la saisie utilisateur - ici un **DoubleVar()**, car nous travaillons avec des nombres réels) ;
- les quatre boutons permettant de réaliser les opérations dans les lignes 49 à 59 (**command** indique la fonction à exécuter en cas de clic ; pour fournir un paramètre à cette fonction, nous devons utiliser des fonctions anonymes (ou *lambda function*).

C'est la méthode **operation()** des lignes 5 à 15 qui est chargée de calculer le résultat en fonction du signe qui lui est transmis par le bouton qui l'a appelée et de la valeur des deux attributs **var_a** et **var_b**. ▮

4 UTILISEZ LES BONS OUTILS

À découvrir dans cette partie...

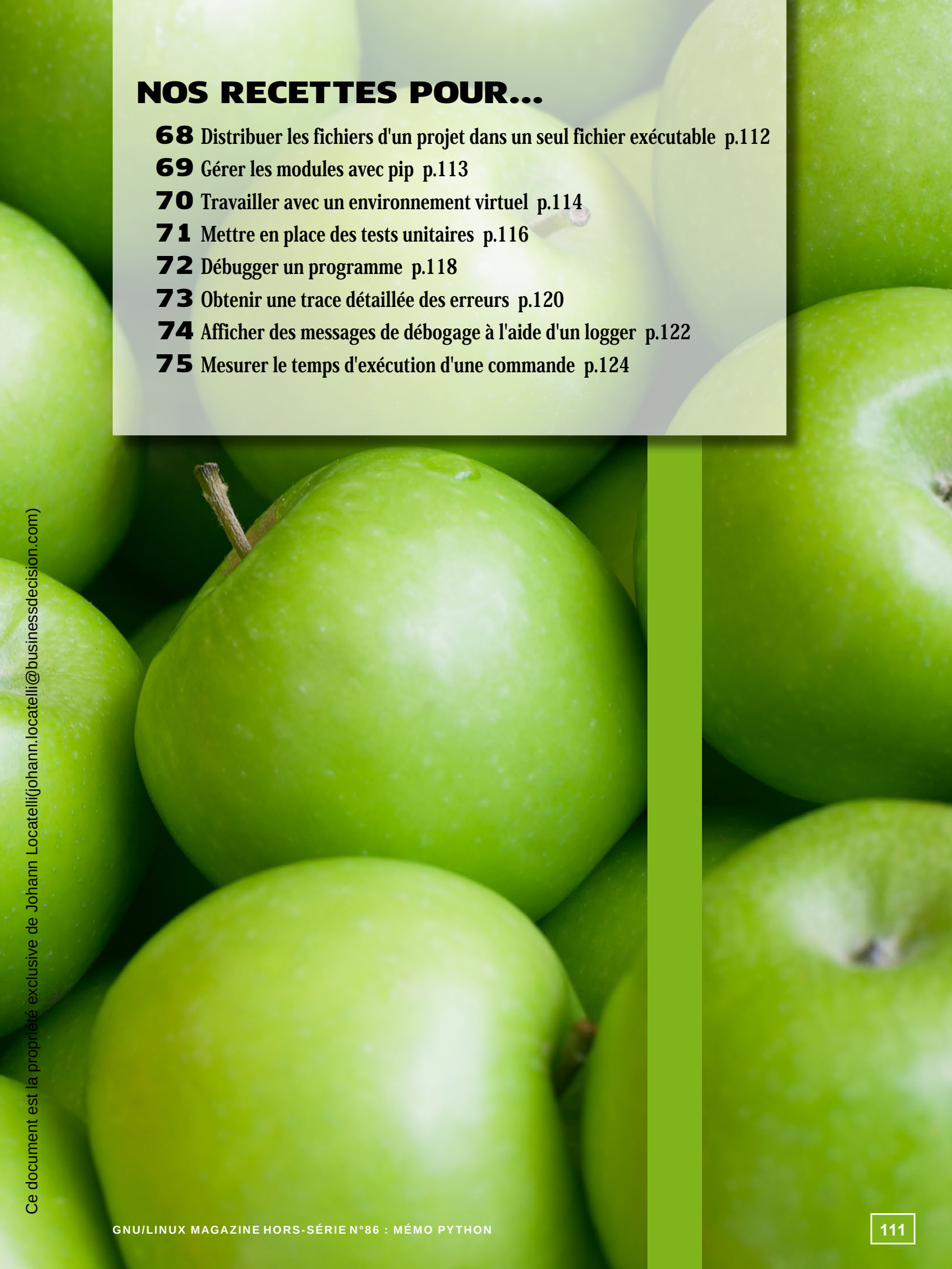
L'écosystème Python nous permet de bénéficier de nombreux outils touchant différents domaines tels que les tests unitaires, le débuggage ou encore l'environnement de développement.

La gestion des fichiers d'un projet et la configuration de l'environnement sera abordée dans trois recettes (recettes 68 à 70), la mise en place de tests unitaires fera l'objet de la recette 71, le débuggage interviendra dans les recettes 72 à 74 et enfin, l'étude des performances du code sera le sujet de la recette 75.

Des outils à utiliser tous les jours pour optimiser son travail...

NOS RECETTES POUR...

68 Distribuer les fichiers d'un projet dans un seul fichier exécutable p.112
69 Gérer les modules avec pip p.113
70 Travailler avec un environnement virtuel p.114
71 Mettre en place des tests unitaires p.116
72 Débugger un programme p.118
73 Obtenir une trace détaillée des erreurs p.120
74 Afficher des messages de débogage à l'aide d'un logger p.122
75 Mesurer le temps d'exécution d'une commande p.124



RECETTE 68

DISTRIBUER LES FICHIERS D'UN PROJET dans un seul fichier exécutable

L'OBJECTIF

Regrouper au sein d'un seul fichier compressé exécutable l'ensemble des fichiers d'un projet.

LA SOLUTION

Il faut structurer le projet à l'intérieur d'un répertoire que nous nommerons ici `projet`. Ce répertoire doit contenir un fichier `__init__.py` (pour qu'il soit accessible) et un fichier `__main__.py` qui sera le fichier exécuté au démarrage. Nous ajouterons ici un fichier `lib.py` qui contiendra une fonction :

```
01: def affiche():
02:     print('Exécution de la fonction affiche()')
```

Le fichier `__main__.py` contiendra ici :

```
01: import lib
02:
03: lib.affiche()
```

Pour compresser le projet, il faudra lancer la commande suivante dans un terminal :

```
$ python3 -m zipapp projet
```

Pour l'exécuter, ce sera simplement un appel de Python sur le fichier compressé (sous GNU/Linux, l'extension sera `.pyz` et `.pyzw` sous Windows) :

```
$ python3 projet.pyz
```

COMMENTAIRES

Notez que si votre exécutable doit démarrer simplement sur une fonction de l'un des modules (un peu comme l'exemple présenté avec l'exécution de `lib.affiche()`), vous pouvez vous affranchir de la création d'un fichier `__main__.py` en lançant la commande suivante qui indique qu'il faudra lancer la fonction `affiche` du module `lib` :

```
$ python3 -m zipapp projet -m "lib:affiche"
```

Sous GNU/Linux, l'option `-p` permet d'ajouter une ligne de *shebang* autorisant l'exécution du fichier sans appeler l'interpréteur manuellement :

```
$ python3 -m zipapp projet -p /usr/local/bin/python3
```

RECETTE 69 GÉRER LES PAQUETS avec pip

L'OBJECTIF

Utiliser `pip` pour installer et supprimer des paquets Python.

LA SOLUTION

L'installation se fait par la sous-commande `install` et on peut éventuellement spécifier un numéro de version :

```
$ sudo pip3 install click
$ sudo pip3 install click==6.5
```

Il est possible d'installer un paquet pour l'utilisateur courant avec l'option `--user` :

```
$ pip3 install click --user
```

Pour supprimer un paquet, on utilise `uninstall` :

```
$ sudo pip3 uninstall click
```

La commande `pip3 freeze` permet de lister l'ensemble des paquets installés. Très utile dans le cadre de l'utilisation d'un environnement virtuel, cela permet ensuite de réinstaller l'ensemble des paquets en une seule commande :

```
$ pip3 freeze > requirements.txt
$ pip3 install -r requirements.txt
```

NOTE

La commande `show` permet d'afficher des informations sur un paquet installé précédemment :

```
$ pip3 show numpy
---
Metadata-Version: 2.0
Name: numpy
Version: 1.11.0
Summary: NumPy: array processing for numbers, strings, records,
and objects.
Home-page: http://www.numpy.org
Author: NumPy Developers
Author-email: numpy-discussion@scipy.org
Installer: pip
License: BSD
Location: /usr/local/lib/python3.5/site-packages
Requires:
Classifiers:
  Development Status :: 5 - Production/Stable
...
```

LES BONS OUTILS

RECETTE 70

TRAVAILLER avec un environnement virtuel

L'OBJECTIF

Créer un environnement virtuel pour installer ses dépendances sans polluer le système de la machine de développement. Notre environnement s'appellera `env_test`.

LA SOLUTION

1 Méthode simplifiée depuis Python 3.3

```
$ pyvenv env_test
```

Pour activer l'environnement :

```
$ source env_test/bin/activate
(env_test) $
```

Sous Windows, ce sera :

```
> env_test/Scripts/activate.bat
(env_test) >
```

Pour quitter l'environnement virtuel :

```
(env_test) $ deactivate
```

2 Virtualenvwrapper

Il faut installer **virtualenvwrapper** :

```
$ sudo pip3 install virtualenvwrapper
```

Vous pouvez ensuite définir une variable d'environnement `WORKON_HOME` pour déterminer le lieu de stockage de vos environnements virtuels et lancer le script `virtualenvwrapper.sh` (vous pouvez placer ces lignes dans le fichier de configuration de votre shell - `~/.bashrc` si vous utilisez le `bash`) :

```
$ export WORKON_HOME=~/.virtualenvs
$ source /usr/local/bin/virtualenvwrapper.sh
```

Pour Windows, il faut installer **virtualenvwrapper-win** :

```
> pip3 install virtualenvwrapper-win
```

Et la variable `WORKON_HOME` est définie par défaut à `%USERPROFILE%\Envs` (mais vous pouvez bien entendu changer sa valeur). Pour créer un environnement, on utilise la commande `mkvirtualenv` suivie du nom et éventuellement de la version de Python :

```
$ mkvirtualenv env_test -p /usr/bin/python3.4
```

Pour lister les environnements virtuels disponibles :

```
$ lsvirtualenv
downfall
========
env_test
========
...
```

Pour activer l'environnement :

```
$ workon env_test
(env_test) $
```

Pour quitter l'environnement virtuel :

```
(env_test) $ deactivate
```

COMMENTAIRES

1 Méthode simplifiée depuis Python 3.3

Si vous avez plusieurs versions de Python disponibles sur votre machine, vous pouvez sélectionner celle utilisée en utilisant la commande `pyvenv-3.x` où `x` est le numéro de la version que vous souhaitez employer. Si vous ne faites pas cela, ce sera la version de Python la plus récente qui sera employée pour créer l'environnement virtuel. Sous Windows, la commande à lancer pour activer l'environnement virtuel est :

```
> env_test/Scripts/activate
```

Lors de la création d'un environnement virtuel, l'ajout de l'option `--system-site-packages` permettra d'avoir accès aux modules du système (ce qui n'est pas le cas par défaut puisque l'intérêt d'un environnement virtuel est justement d'isoler l'environnement de développement du système).

2 Virtualenvwrapper

Lors de l'exécution de `virtualenvwrapper.sh`, il est possible que vous obteniez le message suivant :

```
/usr/bin/python: No module named virtualenvwrapper
virtualenvwrapper.sh: There was a problem running the initialization hooks.
If Python could not import the module virtualenvwrapper.hook_loader,
check that virtualenvwrapper has been installed for
VIRTUALENVWRAPPER_PYTHON=/usr/bin/python and that PATH is set properly.
```

Déterminez quelle version de `pip` a été employée pour installer virtualenvwrapper :

```
$ pip3 --version
pip 8.1.1 from /usr/local/lib/python3.5/site-packages (python 3.5)
```

Récupérez le chemin vers cet interpréteur :

```
$ which python3.5
/usr/local/bin/python3.5
```

Définissez la variable d'environnement `VIRTUALENVWRAPPER_PYTHON` dans votre `.bashrc` avant d'exécuter `virtualenvwrapper.sh` :

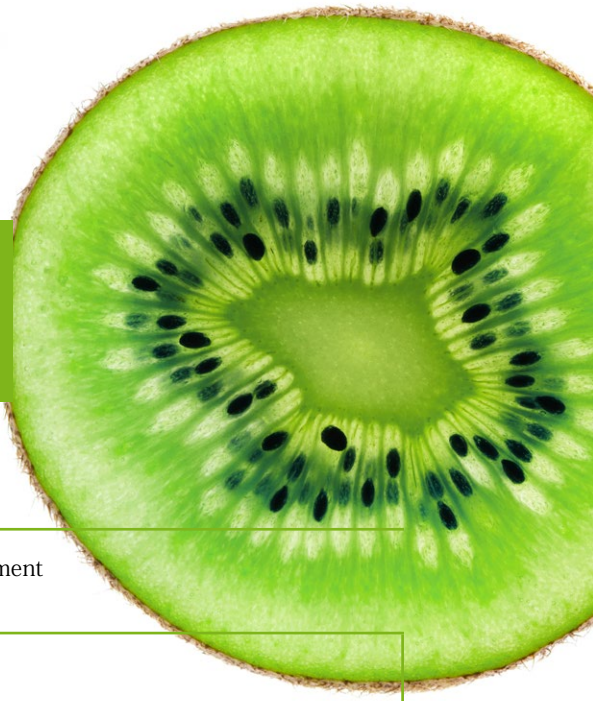
```
...
# Configuration de Virtualenvwrapper
export WORKON_HOME=~/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.5
source /usr/local/bin/virtualenvwrapper.sh
...
```

Pour détruire un environnement virtuel, il suffit de supprimer son répertoire. Virtualenvwrapper propose une commande permettant de simplifier l'opération, `rmvirtualenv` :

```
$ rmvirtualenv env_test
Removing env_test...
$
```

RECETTE 71

METTRE EN PLACE des tests unitaires



L'OBJECTIF

Créer un fichier de tests unitaires permettant de s'assurer du fonctionnement d'une classe. Nous prendrons pour exemple la classe suivante :

```
01: class Operation:
02:
03:     def __init__(self, valueA, valueB):
04:         self.__a = valueA
05:         self.__b = valueB
06:
07:     def add(self):
08:         return self.__a + self.__b
09:
10:     def div(self):
11:         return self.__a / self.__b
12:
13: if __name__ == '__main__':
14:     op = Operation(4, 2)
15:     print('Valeurs : 4 et 2')
16:     print('Addition :', op.add())
17:     print('Division :', op.div())
```

LA SOLUTION

```
01: import unittest
02: from Operation import Operation
03:
04: class Operationtest(unittest.TestCase):
05:     def setUp(self):
06:         print('Lancement setUp()')
07:         self.fixture = Operation(2, 3)
08:
09:     def tearDown(self):
10:         print('Lancement tearDown()')
11:
12:     def test_simple_add(self):
13:         self.obj = Operation(2, 3)
14:         self.assertEqual(self.obj.add(), 2 + 3)
15:
```

```
16:     def test_negatif_add(self):
17:         self.obj = Operation(-2, -3)
18:         self.assertEqual(self.obj.add(), 2 + -3)
19:
20: if __name__ == '__main__':
21:     unittest.main()
```

COMMENTAIRES

Les méthodes de tests sont préfixées par `test_` suivi du nom de la méthode ou du nom du test puis de celui de la méthode. Par exemple, la méthode `add()` dispose du test `test_simple` (lignes 12 à 14) et `test_negatif` (lignes 16 à 18).

La méthode `setUp()` permet de créer l'attribut `fixture` qui est une instance d'`Operation` et qui sera employé dans les tests (cela évite d'avoir à écrire la création d'une instance dans chaque test). La méthode `tearDown()` est exécutée après chaque test. Ici, elle ne sert à rien d'autre qu'à afficher un message indiquant qu'elle a été exécutée. La présence de cette méthode n'est pas obligatoire si vous ne vous en servez pas.

Voici maintenant le résultat de l'exécution des tests. Nous considérons que le fichier de tests s'appelle `test_Operation.py` :

```
$ python3 test_Operation.py
Lancement setUp()
Lancement tearDown()
FLancement setUp()
Lancement tearDown()
.

======================================================================
FAIL: test_negatif_add (__main__.Operationtest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_Operation.py", line 18, in test_negatif_add
    self.assertEqual(self.obj.add(), 2 + -3)
AssertionError: -5 != -1


----------------------------------------------------------------------
Ran 2 tests in 0.001s


FAILED (failures=1)
```

Un des tests a échoué.

Pour lancer tous les tests (différents fichiers) d'un répertoire, si vous avez pris soin de nommer tous vos fichiers en utilisant le préfixe `test_`, vous pouvez utiliser la commande :

```
$ python3 -m unittest
```

RECETTE 72

DÉBUGGER un programme avec pdb

L'OBJECTIF

Nous voulons débugger le programme `factorielle.py` suivant qui devrait calculer la factorielle d'un entier tiré au hasard entre `0` et `10`. Malheureusement, ce code renvoie toujours `0`...

```
01: import random
02:
03: def fact():
04:     n = random.randint(0, 10)
05:     result = 0
06:
07:     for i in range(n):
08:         result += result * i
09:
10:     return n, result
11:
12: if __name__ == '__main__':
13:     n, r = fact()
14:     print('{}! = {}'.format(n, r))
```

LA SOLUTION

Nous allons utiliser pdb, le débuggeur Python fourni avec le langage. Il faut le lancer sur le programme de la manière suivante :

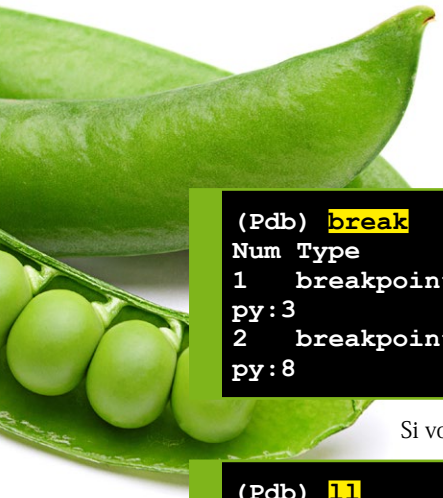
```
$ python3 -m pdb factorielle.py
> /home/tristan/python/factorielle.py(1)<module>()
-> import random
(Pdb)
```

Le prompt a changé : `(Pdb)` montre que le débuggeur attend maintenant une commande (la liste de celles-ci est accessible en tapant `help` ou `h`).

La ligne `-> import random` indique la prochaine ligne qui sera exécutée. Nous allons utiliser quelques-unes des commandes pour trouver notre erreur :

- nous plaçons un point d'arrêt au début de la fonction `fact()` et en ligne 8 :

```
(Pdb) b fact
Breakpoint 1 at /home/tristan/python/factorielle.py:3
(Pdb) b 8
Breakpoint 2 at /home/tristan/python/factorielle.py:8
```



- nous vérifions la présence de nos points d'arrêt :

```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /home/tristan/python/factorielle.
py:3
2   breakpoint   keep yes   at /home/tristan/python/factorielle.
py:8
```

Si vous le préférez, vous pouvez aussi afficher ces points d'arrêt dans leur contexte :

```
(Pdb) ll
  1  -> import random
  2
  3 B   def fact():
  4         n = random.randint(0, 10)
  5         result = 0
  6
  7         for i in range(n):
  8 B           result += result * i
  9
 10         return n, result
 11
 12     if __name__ == "__main__":
 13         n, r = fact()
 14         print('{}! = {}'.format(n, r))
```

- nous exécutons le code jusqu'au premier point d'arrêt :

```
(Pdb) c
> /home/tristan/python/factorielle.py(4)fact()
-> n = random.randint(0, 10)
```

- on peut exécuter plusieurs itérations et afficher à chaque fois les valeurs des variables `i`, `n` et `result` :

```
(Pdb) c
> /home/tristan/python/factorielle.py(8)fact()
-> result += result * i
(Pdb) display(i, n, result)
display (i, n, result): (0, 2, 0)
(Pdb) c
> /home/tristan/python/factorielle.py(8)fact()
-> result += result * i
display (i, n, result): (1, 2, 0)  [old: (0, 2, 0)]
(Pdb) c
2! = 0
The program finished and will be restarted
```

La variable `result` reste à `0`... Le problème vient de la ligne 5 où il faut initialiser `result` à `1` (neutre pour la multiplication).

COMMENTAIRES

Il existe d'autres débuggeurs pour Python. On peut citer **pdb-clone** qui permet le *remote-debugging*, **pudb** qui propose une interface CLI, et **ipdb** qui est la version de pdb pour Ipython. ▮

RECETTE 73

OBTENIR UNE TRACE DÉTAILLÉE des erreurs

L'OBJECTIF

Un programme renvoie une erreur et l'on souhaite obtenir plus de précisions sur cette erreur.

LA SOLUTION

Ajouter en début de programme les lignes suivantes :

```
01: import cgitb
02: cgitb.enable(format='text')
...
```

COMMENTAIRES

Pour bien voir l'influence de ces deux lignes, considérons le programme suivant :

```
01: import cgitb
02: cgitb.enable(format='text')
03:
04: def div(a, b):
05:     return a / b
06:
07: div(1, 0)
```

Sans les lignes 1 et 2, nous obtiendrons seulement l'affichage d'une exception :

```
$ python3 trace_detaillee.py
Traceback (most recent call last):
  File "trace_detaillee.py", line 4, in <module>
    div(1, 0)
  File "trace_detaillee.py", line 2, in div
    return a / b
ZeroDivisionError: division by zero
```

Bien entendu, ce message d'erreur devrait être suffisant pour corriger l'erreur. Admettons que ce ne soit pas le cas et que nous lancions alors notre programme en utilisant le module **cgitb** :

```
$ python3 trace_detaillee.py
ZeroDivisionError
Python 3.5.1: /usr/local/bin/python3
Mon May  2 17:15:19 2016


A problem occurred in a Python script.  Here is the sequence of
function calls leading up to the error, in the order they occurred.


trace_detaillee.py in <module>()
    3
    4 def div(a, b):
    5     return a / b
    6
    7 div(1, 0)
div = <function div>


trace_detaillee.py in div(a=1, b=0)
    3
    4 def div(a, b):
    5     return a / b
    6
    7 div(1, 0)
a = 1
b = 0
ZeroDivisionError: division by zero
    __cause__ = None
    __class__ = <class 'ZeroDivisionError'>
...
    __traceback__ = <traceback object>
    args = ('division by zero',)
    with_traceback = <built-in method with_traceback of
ZeroDivisionError object>


The above is a description of an error in a Python program.  Here
is the original traceback:


Traceback (most recent call last):
  File "trace_detaillee.py", line 7, in <module>
    div(1, 0)
  File "trace_detaillee.py", line 5, in div
    return a / b
ZeroDivisionError: division by zero
```

Avec cet affichage, on peut suivre plus simplement les valeurs des différentes variables. ▮

LES BONS OUTILS

RECETTE 74

AFFICHER DES MESSAGES de débogage à l'aide d'un logger

L'OBJECTIF

Disposer d'un fichier de log permettant de suivre l'exécution d'un programme et pouvoir ainsi le déboguer plus simplement. Les informations seront affichées à l'écran et également dans un fichier, mais avec un niveau de détail différent.

LA SOLUTION

Le fichier de configuration du *logger*, `mon_log.conf` :

```
01: [loggers]
02: keys = root,test_log
03:
04: [handlers]
05: keys = filehandler,consolehandler
06:
07: [formatters]
08: keys = completeFormatter,partialFormatter
09:
10: [logger_root]
11: level = NOTSET
12: handlers =
13:
14: [logger_test_log]
15: handlers = filehandler,consolehandler
16: qualname = test_log
17:
18: [handler_filehandler]
19: class = handlers.RotatingFileHandler
20: args = ("monAppli.log", 2048, 5)
21: level = INFO
22: formatter = completeFormatter
23:
24: [handler_consolehandler]
25: class = StreamHandler
26: args = (sys.stdout,)
27: level = ERROR
28: formatter = partialFormatter
29:
30: [formatter_completeFormatter]
31: format = %(asctime)s - %(name)s - %(filename)s - %(funcName)s
(%(lineno)d) - %(levelname)s : %(message)s
32: datefmt = %d/%m/%Y %H:%M:%S
33:
34: [formatter_partialFormatter]
35: format = %(levelname)s : %(message)s
```

Le programme utilisant le fichier de configuration :

```
01: import logging
02: import logging.config
```

```
03:
04: if __name__ == "__main__":
05:     logging.config.fileConfig("mon_log.conf")
06:     main_logger = logging.getLogger("test_log")
07:
08:     main_logger.critical("Ceci est une erreur critique !")
09:     main_logger.warning("Ceci est un message de débogage !")
```

Le résultat à l'écran :

```
$ python3 logger.py
CRITICAL : Ceci est une erreur critique !
```

Le contenu du fichier `monAppli.log` :

```
27/08/2016 17:29:58 - test_log - logger.py - <module> (8) -
CRITICAL : Ceci est une erreur critique !
27/08/2016 17:29:58 - test_log - logger.py - <module> (9) -
WARNING : Ceci est un message de débogage !
```

COMMENTAIRES

Le fichier de configuration permet de définir le nombre de loggers à créer (lignes 1 et 2). Ici, il s'agira de `root` (obligatoire même s'il n'est pas utilisé comme dans cet exemple) et de `test_log`. La section `[handlers]` des lignes 4 et 5 indique que nous allons définir deux *handlers* qui sont les types de logs. La section `[formatters]` des lignes 7 et 8 permet d'indiquer que deux types de formats seront définis (`completeFormatter` et `partialFormatter`). Ces premières lignes sont en quelque sorte l'en-tête de notre *logger* (nos *loggers* pour être plus exact).

Dans les lignes 10 à 12, le *logger* `root` est désactivé (`level` à `NOTSET`). Dans les lignes 14 à 16, on indique que le *logger* `test_log` utilisera deux *handlers* nommés `filehandler` et `consolehandler` et que son nom sera... `test_log` (option `qualname` de la ligne 16). Ces deux sections définissant les *loggers* doivent avoir un nom commençant par `handler_`.

Les lignes 18 à 28 permettent de définir les deux *handlers* (les noms des sections commencent nécessairement par `handler_`). Pour le premier, il s'agira d'écrire les messages dans des fichiers `monAppli.log` sur lesquels une rotation sera appliquée : taille maximale de chaque fichier 2Kio (`2048` bits) et conservation des fichiers `monAppli.log`, `monAppli.log.1`, ..., `monAppli.log.5` (le `5` de `args = ("monAppli.log", 2048, 5)` en ligne 20). Les messages seront de niveau `INFO` et le format employé pour les afficher sera `completeFormatter`. Le second *handler* affichera les messages sur la console, il sera de niveau `ERROR` et utilisera le format d'affichage `partialFormatter`.

Les deux dernières sections des lignes 30 à 35 définissent les formats d'affichage. Les noms de ces sections doivent commencer par `formatter_`.

Les niveaux d'erreurs

Il existe six niveaux d'erreurs classés de manière hiérarchique : `CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET`.

La signification de ces niveaux d'erreurs est la suivante :

- `CRITICAL` : erreurs critiques ;
- `ERROR` : erreurs ;
- `WARNING` : messages d'avertissement ;
- `INFO` : messages d'information ;
- `DEBUG` : tests et débogage ;
- `NOTSET` : non défini (utilisé uniquement pour désactiver un *logger*).

Cette classification implique qu'un *logger* de niveau `INFO` affichera les messages `INFO`, `WARNING`, `ERROR` et `CRITICAL`. Un *logger* affiche toujours les messages de son niveau et des niveaux supérieurs.

RECETTE 75

MESURER LE TEMPS D'EXÉCUTION d'une commande

L'OBJECTIF

Nous disposons de deux implémentations d'une fonction factorielle et nous souhaitons déterminer laquelle est la plus efficace.

```
01: def fact_it(n):
02:     result = 1
03:
04:     for i in range(n):
05:         result += result * i
06:
07:     return result
08:
09: def fact_rec(n):
10:     if n == 0 or n == 1:
11:         return 1
12:     else:
13:         return n * fact_rec(n - 1)
14:
15: if __name__ == '__main__':
16:     print('{}! = {}'.format(6, fact_it(6)))
17:     print('{}! = {}'.format(6, fact_rec(6)))
```

LA SOLUTION

```
01: import timeit
...
17: if __name__ == '__main__':
...
20:     tps_fact_it = timeit.repeat('fact_it(32)', 'from __main__
import fact_it', repeat=3, number=10000)
21:     tps_fact_rec = timeit.repeat('fact_rec(32)', 'from __main__
import fact_rec', repeat=3, number=10000)
22:     print('Tps pour fact_it : {}'.format(min(tps_fact_it)))
23:     print('Tps pour fact_rec : {}'.format(min(tps_fact_rec)))
```

COMMENTAIRES

Le module **timeit** fournit des fonctions pour évaluer le temps d'exécution d'un code. `repeat()` permet d'exécuter `repeat` mesures de la commande passée en premier paramètre et exécutée `number` fois. Le résultat de l'exécution de notre code est le suivant :

```
6! = 720
6! = 720
Tps pour fact_it : 0.040213363001385005
Tps pour fact_rec : 0.05365261499900953
```

NOTE

Le deuxième paramètre de `repeat()` permet d'importer la fonction que nous voulons tester et qui a été définie dans l'espace de nom courant (nommé `__main__`). Ce paramètre est inutile si l'on désire tester des fonctions standards de Python :

```
timeit.repeat('liste = [2, 3, 4]', repeat=3, number=10000)
```

Vous pouvez également lancer directement la commande suivante :

```
$ python -m timeit 'liste = [2, 3, 4]'
```

Vous pouvez également lancer directement la commande suivante :

```
$ python3 -m timeit 'liste = [2, 3, 4]'
```

Les options disponibles sont :

- `-u` permettant de spécifier l'unité de temps (µsec (`usec`), `msec` ou `sec`). Par exemple :

```
$ python3 -m timeit -u usec 'liste = [2, 3, 4]'
10000000 loops, best of 3: 0.0582 usec per loop
$ python3 -m timeit -u msec 'liste = [2, 3, 4]'
10000000 loops, best of 3: 5.81e-05 msec per loop
```

- `-s` pour indiquer un traitement à exécuter une seule fois au départ de la série de mesures. Par exemple :

```
$ python3 -m timeit -s 'f=lambda x : x**2'  'f(10)'
1000000 loops, best of 3: 0.33 usec per loop
```

- `-n` pour déterminer le nombre de fois où le traitement devra être exécuté. Par exemple :

```
$ python3 -m timeit -n 100 'liste = [2, 3, 4]'
100 loops, best of 3: 0.119 usec per loop
```

- `-r` qui permet d'indiquer le nombre de fois où la série de mesures doit être lancée. Par exemple :

```
$ python3 -m timeit -r 4 'liste = [2, 3, 4]'
10000000 loops, best of 4: 0.0588 usec per loop
```

Si vous utilisez l'interpréteur IPython, la commande magique `%timeit` permet d'évaluer le temps d'exécution d'une commande directement depuis le shell IPython :

```
$ ipython
In [1]: %timeit liste=[1, 2, 3]
The slowest run took 19.07 times longer than the fastest. This
could mean that an intermediate result is being cached.
10000000 loops, best of 3: 68.9 ns per loop
```

TOUJOURS DISPONIBLE !

LE 1ER HORS-SÉRIE DE HACKABLE !

VOUS UTILISEZ DÉJÀ LA RASPBERRY PI ?
METTEZ-VOUS À L'ARDUINO !

RENDEZ-VOUS SUR :
www.ed-diamond.com

VISITEZ NOTRE NOUVELLE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE

GNU LINUX PRATIQUE — COMPRENDRE, UTILISER ET ADMINISTRER LINUX — ESSENTIEL

MISC — Multi-System & Internet Security Cookbook

HACKABLE MAGAZINE

Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

« Moi, je les lis en version **PAPIER** ! »

« Moi, je les lis en version **PDF** ! »

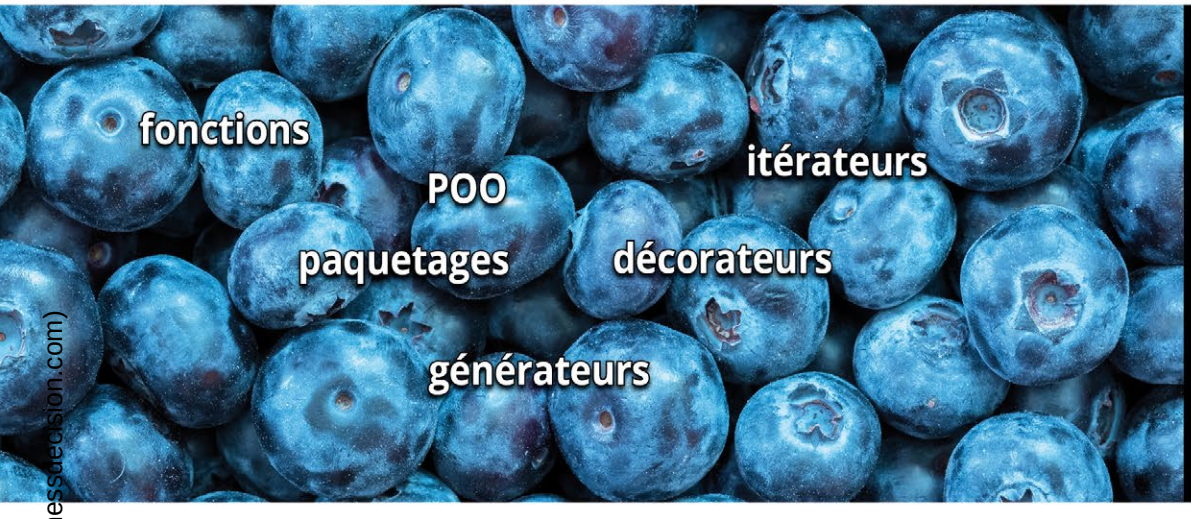
« Moi, je consulte la **BASE DOCUMENTAIRE** ! »

RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !

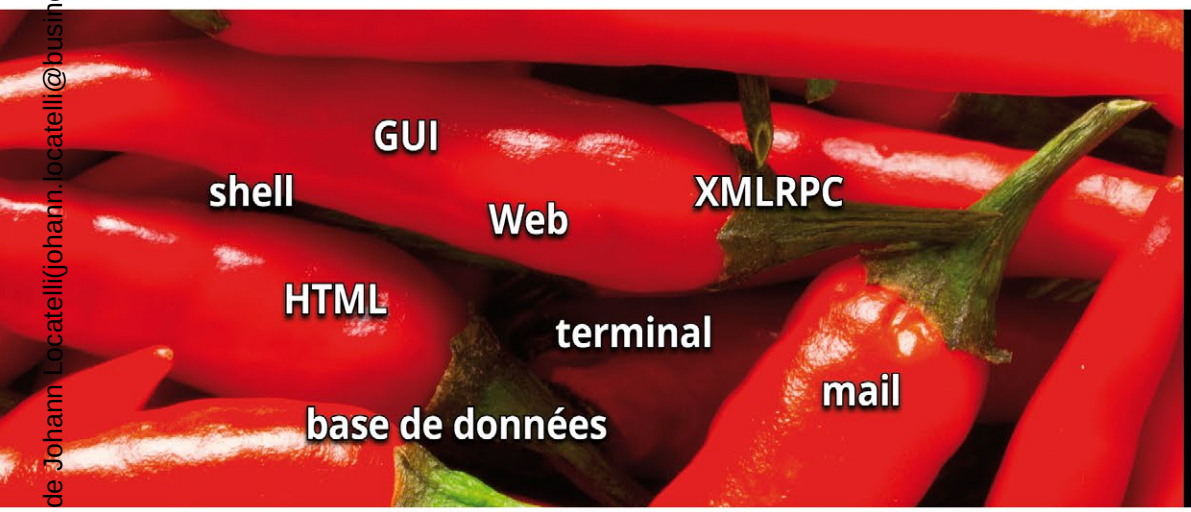
TYPES DE BASE

Exploitez pleinement les listes, dictionnaires et autres types Python



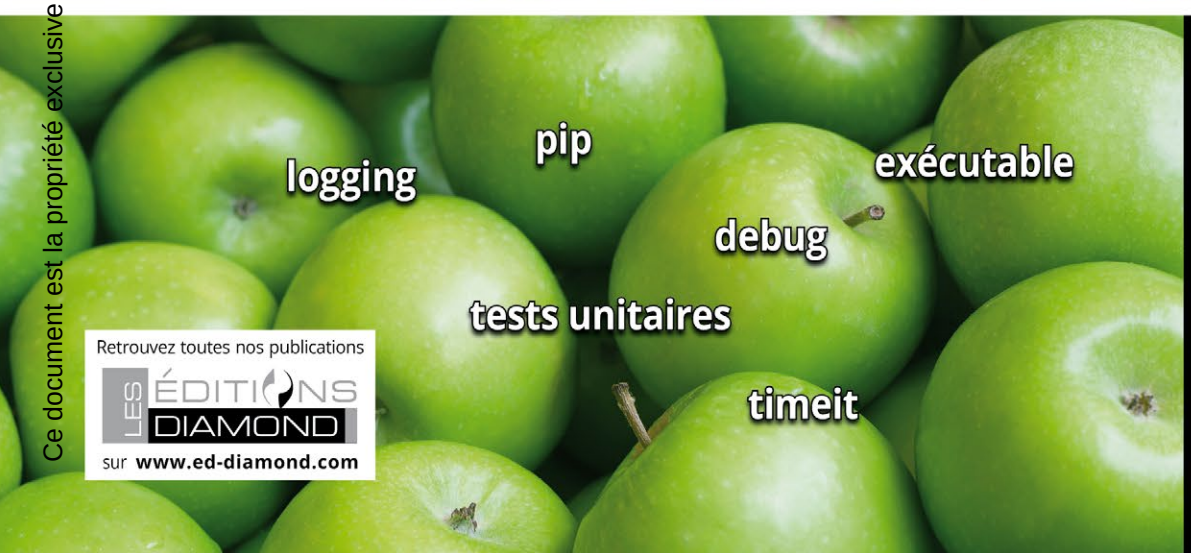
STRUCTURE

Organisez votre code de manière à le rendre efficace et maintenable



DONNÉES & INTERFACES

Interagissez avec vos données et les utilisateurs



OUTILS

Utilisez les bons outils pour développer, débugger et distribuer votre code