2

Python pour quels usages ?

For tiny projects (100 lines or fewer) that involve a lot of text pattern matching, I am still more likely to tinker up a Perl-regexp-based solution [...] For anything larger or more complex, I have come to prefer the subtle virtues of Python -- and I think you will, too.

« Pour les petits projets de moins de cent lignes qui nécessitent beaucoup de recherche de texte, je préfère encore la solution Perl et ses outils d'expressions régulières. Pour tout projet plus grand ou plus complexe, j'opte à présent pour les vertus de Python, et je pense que vous y viendrez aussi. »

Eric Raymond

Le langage C pour l'embarqué, Ada pour les systèmes critiques, Perl pour les expressions régulières, etc. Chaque langage a ses sujets de prédilection, que ce soit pour des raisons historiques ou parce qu'il offre de réels avantages dans le domaine.

Ce chapitre décrit les différents domaines dans lesquels Python est le plus utilisé, au travers d'exemples concrets, à savoir :

- l'administration système ;
- le prototypage rapide d'applications ;
- la recherche et le calcul scientifique ;
- les applications de gestion ;
- les applications web.

Cette liste n'est certainement pas exhaustive mais représente les domaines les plus fréquemment cités.

Administration système

Les administrateurs système ont souvent besoin de concevoir des petits programmes pour automatiser certaines tâches. Ils utilisent généralement l'interpréteur de commandes, qui offre une syntaxe basique pour concevoir des séquences d'opérations.

Toutefois, ce système est très limité et n'offre que des fonctionnalités de très haut niveau : certaines opérations sur le système ne sont pas possibles sans appels à des programmes annexes.

Utiliser des langages de plus bas niveau comme le C permet de lever ces limitations, mais la conception des scripts devient vite fastidieuse et délicate.

Python, conçu à l'origine pour ce cas de figure, s'intercale entre l'interpréteur de commandes et le C, en proposant un niveau intermédiaire, c'est-à-dire un shell surpuissant, et dans le même temps un langage de programmation plus simple et plus direct.

Bien que ce genre de besoin soit plus fréquent sur les systèmes Unices (les systèmes de la famille Unix), il n'est plus rare de rencontrer des administrateurs Windows qui aient adopté Python pour la conception de leurs scripts système.

Des API simples et efficaces

Un langage de manipulation d'un système d'exploitation doit permettre de travailler avec ce dernier de manière pertinente et concise. Manipuler un système consiste notamment à :

- manipuler des fichiers et des dossiers ;
- manipuler des programmes ;
- envoyer et recevoir des e-mails ;
- échanger des informations avec d'autres systèmes.

Manipuler des fichiers et des dossiers

La manipulation du système de fichiers est triviale et puissante en Python. Prenons l'exemple d'un script dont l'objectif est de rechercher sur le système tous les fichiers d'extension `.log` dont la taille dépasse 1 Mo pour générer un rapport.

Recherche de gros fichiers d'extension .log

```
#!/usr/bin/python
# -*- coding: ISO-8859-15 -*-
import os

UN_MEGA = 1024*1024
```

```
def scan_rep(repertoire, extension):
    """ scan le répertoire courant à la recherche
    de fichiers de plus de 1mo """
    for racine, reps, fichiers in os.walk(repertoire, topdown=True):
        for fichier in fichiers:
            if fichier.endswith('.%s' % extension):
                nom_complet = os.path.join(racine, fichier)
                taille = os.path.getsize(nom_complet)
                if taille >= UN_MEGA:
                    print '%s : %.2f Mo' % \
                          (nom_complet, taille/float(UN_MEGA))

if __name__ == '__main__':
    scan_rep('.', 'log')
```

Ce petit script multi-plate-forme utilise pour parcourir le contenu du disque une API nommée `walk` qui gère tous les aspects inhérents au système de fichiers comme les problématiques de droits d'accès ou encore les liens symboliques qui risqueraient de faire partir le programme dans une boucle infinie.

Il est bien sûr perfectible (certains répertoires du système n'ont pas besoin d'être scannés), mais témoigne du confort fourni par les API système de Python.

Manipuler des programmes

Imaginons qu'un administrateur rencontre un problème avec son serveur web *Apache*, qui s'arrête plusieurs fois par jour sans raison apparente. Ce problème ne se retrouve malheureusement que sur le serveur de production. Il faut donc réussir à le régler tout en maintenant le service. L'administrateur souhaite concevoir un petit script qui procède à une série de tests avant de relancer Apache.

Sans rentrer dans les détails des tests opérés, voici à quoi pourrait ressembler le script en question :

Script de surveillance d'Apache

```
import os
from outils import runAudit

cmd_status = '/etc/init.d/httpd status'
cmd_restart = '/etc/init.d/httpd restart'

def checkApache():
    """ surveille l'état du daemon Apache """
    status = os.popen(cmd_status).readlines()
```

```
    # récupère la deuxième ligne et retire le saut de ligne
    status = status[1].strip()

    if status ==  'Apache is *not* running.':
        # Tests sur le système
        runAudit()

        # Apache doit être relancé
        os.popen(cmd_restart).read()
        status = os.popen(cmd_status).readlines()
        status = status[1].strip()

        if status == 'Apache is running.':
            print 'Apache relancé avec succès'
        else:
            print 'Impossible de relancer Apache'
    else:
        print 'État OK : %s' % status

checkApache()
```

L'API `popen` permet de lancer une commande et de récupérer le résultat sous forme d'une liste de lignes. Ce script est facilement portable sur tout autre système compatible avec Python si les chemins vers les commandes utilisées et la lecture des résultats de ces dernières sont adaptés.

Envoyer et recevoir des courriers électroniques

Souvent, l'e-mail est le seul lien entre l'administrateur et l'ensemble des utilisateurs, ou entre l'administrateur et ses serveurs. Maîtriser l'envoi d'e-mail est donc relativement important. Cette action est triviale au niveau du shell et l'intérêt d'un langage de script dans ce cas reste donc limité.

La réception et le traitement d'e-mails de structures complexes est en revanche une opération beaucoup plus délicate.

Prenons un exemple concret : l'administrateur souhaite automatiser la mise en place des *clés SSH* (voir encadré) des utilisateurs sur le serveur. Il propose à ces derniers de lui envoyer un e-mail contenant l'identifiant de l'utilisateur et la clé en pièce attachée à une adresse e-mail prévue à cet effet.

Le script à réaliser doit automatiquement récupérer ces e-mails, placer la clé sur le serveur et envoyer un accusé de réception à l'utilisateur. Les e-mails traités sont ensuite archivés dans un répertoire `Traités` de l'adresse e-mail dédiée.

Mise en place automatique des clés SSH

```
from imaplib import IMAP4
from smtplib import SMTP
from email import message_from_string
from email.MIMEText import MIMEText

def setupKey(contenu_nom, contenu_cle):
    """ met en place la clé sur le système """
    [...]

# mise en place des connecteurs
reception = IMAP4('localhost')
reception.login('cles@localhost', 'motdepass')
reception.create('INBOX.Traités')
envoi = SMTP('localhost')

# mise en place de l'accusé de réception
mail_envoye = MIMEText('Votre clé SSH est activée')
mail_envoye['From'] = 'administrateur <root@localhost>'
mail_envoye['Subject'] = 'Clé SSH activée'
reception.select('INBOX')

# lecture des messages
ids_mails = reception.search(None, 'ALL')[1]

for id_mail in ids_mails:

    if id_mail.strip() == '':
        continue

    contenu = reception.fetch(id_mail, '(RFC822)')[1][0][1]
    mail_recu = message_from_string(contenu)

    if mail_recu.is_multipart():
        envoye_par = mail_recu['From']

        # lecture nom
        nom = mail_recu.get_payload(0)
        contenu_nom = nom.get_payload().strip()

        # récupération clé
        cle = mail_recu.get_payload(1)
        contenu_cle = cle.get_payload().strip()
```

```
        # déplacement message sur serveur dans sous-dossier "Traités"
        reception.copy('INBOX.Traités', id_mail)
        reception.store(id_mail, 'FLAGS', '(\Deleted)')

        # place la clé sur le système
        setupKey(contenu_nom, contenu_cle)

        # accusé de réception
        mail_envoye['To'] = envoye_par
        envoi.sendmail('administrateur <root@localhost>', envoye_par,
                       mail_envoye.as_string())

    else:
        # mauvaise structure, le mail
        # devrait être composé de deux parties
        pass

# fermeture des connecteurs
reception.expunge()
reception.close()
reception.logout()
envoi.quit()
```

Moins de cent lignes sont nécessaires pour mettre en place ce processus relativement complexe, grâce à la simplicité d'utilisation des modules en charge des échanges avec le serveur de courriers.

CULTURE Le SSH en deux mots

Le SSH (Secure Shell) est un shell sécurisé par lequel les utilisateurs peuvent se connecter au serveur. Tous les échanges sont chiffrés.
Pour qu'un serveur reconnaisse automatiquement un utilisateur au moment d'une connexion SSH, il est possible d'utiliser des clés. Les clés SSH sont un couple de fichiers texte que l'utilisateur génère sur son poste par le biais d'un petit utilitaire. Un des deux fichiers (la clé dite privée) reste sur le poste de l'utilisateur et l'autre (la clé publique) est placé sur le serveur. Ces deux clés, de la même manière qu'avec le logiciel *GnuPG*, sont confrontées au moment de la connexion pour authentifier l'utilisateur.
Ce moyen de connexion est souvent le plus sûr et parfois la seule voie proposée par l'administrateur pour se connecter à un système.

Échanger des informations avec d'autres systèmes

Toujours dans l'idée d'automatiser les dialogues entre le serveur et d'autres acteurs du système, maîtriser les différents protocoles directs d'échanges de données doit être aussi simple que l'envoi d'e-mails.

Prenons l'exemple des mises à jour système dans un parc de serveurs. La règle instaurée est qu'une machine de l'Intranet met à disposition par le biais d'un serveur FTP tous les patchs que les serveurs doivent télécharger et exécuter. Le parc de machines est relativement homogène, constitué de serveurs GNU/Linux sous distribution Debian et de serveurs Windows 2000. Sur le serveur FTP, un répertoire pour chacune des plates-formes contient les derniers patchs à récupérer et exécuter.

Chaque serveur est responsable de sa mise à jour. Le script à composer, qui doit pouvoir s'exécuter sur n'importe quelle plate-forme du parc doit donc :

- récupérer les bons patchs ;
- les télécharger ;
- les exécuter ;
- les archiver.

La dernière étape ne consiste qu'à conserver les fichiers téléchargés.

Mise à jour centralisée automatique

```
import os
from StringIO import StringIO
from ftplib import FTP

patches_done = os.listdir(os.curdir)
patches_todo = []
_result = StringIO()

# fonctions de récupération des flux ftp
def callback(line):
    _result.write(line)

def callbacktext(line):
    _result.write('%s\n' % line)

def readresults(text=False):
    content = _result.getvalue()
    _result.buf = ''
    return content

# code principal
ftp = FTP('localhost')
ftp.login('root', 'motdepasse')
try:
    ftp.cwd(os.name)
    ftp.dir(callbacktext)
    patches = readresults().split('\n')
```

```
    # tous les fichiers téléchargés sont binaires
    ftp.voidcmd('TYPE I')
    for patch in patches:
        line = patch.split()
        if len(line) == 0:
            continue
        filename = line[-1]
        if filename not in patches_done:
            ftp.retrbinary('RETR %s' % filename, callback)
            filecontent = readresults()
            file = open(filename, 'w')
            file.write(filecontent)
            file.close()
            os.chmod(filename, 467)
            patch_file = os.path.join(os.curdir, filename)
            patches_todo.append(patch_file)
finally:
    ftp.close()

for patch in patches_todo:
    # le patch est auto-exécutable
    print 'application du patch %s...' % patch
    log = os.popen(patch)
    print '\n'.join(log)
```

Les autres protocoles sont rarement plus complexes à implémenter, sauf lorsqu'il est nécessaire de procéder en entrée et en sortie à des traitements de données plus poussés.

> **À SAVOIR Lancement automatique des scripts**
>
> Les exemples précédents et ceux qui suivront dans ce chapitre ont tous été conçus pour être exécutés par le système de manière automatique et régulière, que ce soit par le biais des *tâches cron* sur les systèmes de type Unix ou par une nouvelle entrée dans le gestionnaire de tâches sur les plates-formes Windows.

Le match Perl-Python

La partie concernant l'administration système serait incomplète sans parler de Perl. Le langage Perl est souvent le langage préféré des administrateurs et a remplacé dans beaucoup de cas le shell. Perl est très puissant, possède une énorme bibliothèque de modules facilement accessible (CPAN) et une communauté très active.

Ce langage souffre cependant de défauts qui peuvent peser lourd lors de la conception d'applications conséquentes, comme une syntaxe pas très lisible, de l'aveu même

de Larry Wall, son créateur et de structures de données difficiles à construire et manipuler. Perl reste cependant très puissant pour les manipulations de texte.

« *Perl is worse than Python because people wanted it worse* »

— Larry Wall

Syntaxe

Prenons l'exemple d'un script en charge de préparer le répertoire web personnel d'un utilisateur lorsqu'il est ajouté à un système GNU/Linux. Le programme doit remplir les tâches suivantes :

- création d'une page web personnelle ;
- ajout dans le serveur Apache d'un *Virtual Directory ;*
- envoi d'un e-mail de notification au nouvel utilisateur.

La page web créée permet à l'utilisateur d'avoir des liens personnalisés vers les applicatifs du groupware de l'entreprise comme le Webmail.

Sans entrer dans les détails du programme, nous allons simplement présenter ici la partie qui consiste à créer la page web personnelle. Cette section du programme peut elle-même être découpée en trois étapes :

1. Chargement d'un modèle de page web.
2. Personnalisation du modèle en fonction de l'utilisateur.
   Les occurrences de <NOM> et <PRENOM> sont remplacées par des valeurs réelles.
3. Création du fichier dans le répertoire web de l'utilisateur.

Version en Python

```
import os

def creation_page(nom, prenom, modele, chemin):
    """ création de la page web """
    contenu_modele = open(modele, 'r')
    page = contenu_modele.readlines()
    page = '\r\n'.join(page)
    contenu_modele.close()

    page = page.replace('<NOM>', nom)
    page = page.replace('<PRENOM>', prenom)

    nom_fichier = os.path.join(chemin, 'index.html')
    fichier = open(nom_fichier, 'w')
    fichier.write(page)
    fichier.close()
```

La version Perl est très similaire en terme de facilité de mise en œuvre et de longueur de code, mais beaucoup moins lisible.

La version Perl

```
use strict;
use warnings;

sub creation_page
{
    my ($nom, $prenom, $modele, $chemin) = (@_);
    open I, "<", $modele;
    my $page = do { local $/; <I> };
    close(I);

    $page =~ s/<NOM>/$nom/g;
    $page =~ s/<PRENOM>/$nom/g;

    open O, ">", "$chemin/index.html";
    print O $page;
    close(O);
}
```

Structures de données

La création et la manipulation de structures de données en Perl est relativement lourde. Dans l'exemple ci-dessous, la création d'une simple classe, sans aucun contenu, nécessite cinq fois plus de code en Perl qu'en Python :

Définition d'une classe en Perl et Python

```
# Version Perl
package MaClasse;
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class
    $self->initialize();  # do initialization here
    return $self;
}

# Version Python
class MaClasse:
    pass
```

Cette syntaxe verbeuse de Perl, qui se confirme dans toutes les définitions de structure, peut être pesante dans la conception d'applications de grande taille, et augmente proportionnellement les risques de bogues.

Manipulation de texte

En terme de manipulation de texte, les outils disponibles pour Perl sont à l'heure actuelle beaucoup plus puissants que pour Python.

À titre d'exemple, les expressions régulières sous Python sont un portage de ce qui existait à l'époque pour Perl 5, et n'ont plus évolué depuis.

La possibilité d'étendre le moteur d'expressions régulières sous Perl est inexistante sous Python.

Extension du moteur regexp sous Perl

```
# exemple tiré de l'aide en ligne de Perl
# permet d'ajouter '\Y|' au moteur
# qui est un raccourci pour (?=\S)(?<!\S)|(?!\S)(?<=\S)
package customre;
use overload;

sub import {
    shift;
    die "No argument to customre::import allowed" if @_;
    overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

my %rules = ( '\\' => '\\',
                'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
    my $re = shift;
    $re =~ s{
            \\ ( \\ | Y . )
            }
            { $rules{$1} or invalid($re,$1) }sgex;
    return $re;
}
```

Conclusion

Perl reste supérieur pour la conception de petits scripts de moins de 100 lignes, par la puissance de ses outils et par son intégration poussée des expressions régulières. Python devient un meilleur choix pour de plus grosses applications.

Prototypage rapide d'applications

Dans certains types de projets, les premières étapes d'implémentation consistent de manière quasi-systématique à effectuer un certain nombre d'allers-retours entre la définition théorique du cahier des charges et une maquette plus ou moins fonctionnelle.

Objectif d'une maquette

Concevoir une maquette permet à l'architecte d'un logiciel de prendre du recul et de réduire la marge entre ce qu'il a imaginé et ce qu'il faut réellement implémenter. Armé de ce prototype, il va déceler très vite certaines problématiques de logique d'implémentation, mais également fournir au client un véritable jouet.

Si ce dernier est lui-même technicien, il pourra faire évoluer la maquette pour exprimer ses besoins de manière plus directe. Ce logiciel pâte à modeler doit donc être facile à créer et à modifier.

Les maquettes peuvent être des maquettes d'interfaces ou plus simplement des maquettes de code.

Maquette d'interfaces

Pour les logiciels dotés d'une interface graphique, la maquette est constituée d'une série d'écrans liés entre eux par des menus et des boutons. C'est avant tout l'*ergonomie* de l'interface et la logique des enchaînements qui priment, car ils sont bien souvent très proches des *processus métier* souhaités par le client.

> **CULTURE Définition de l'ergonomie**
>
> L'ergonomie consiste à améliorer l'interface homme-machine, en rendant l'outil le plus simple et le plus logique possible aux yeux d'un utilisateur. Un programme ergonomique est en général utilisable sans avoir à se référer à l'aide en ligne et diminue au maximum le nombre d'étapes nécessaires à l'utilisateur pour obtenir le résultat recherché.

Il existe plusieurs méthodes pour créer des interfaces avec Python. La plus intéressante pour les exercices de maquettage consiste à utiliser les Environnements de Développement Intégré (*EDI*) qui proposent des éditeurs visuels d'interfaces. Certains n'ont pas forcément de liens avec Python et se contentent de générer des fichiers pour chaque fenêtre dessinée. Ceux-ci peuvent ensuite être chargés et interprétés par un programme Python par le biais de bibliothèques spécialisées. Le programme associe alors une portion de code à chaque événement provoqué par l'utilisateur, selon le principe de la *programmation événementielle*.

On peut citer comme *EDI* pour Python :

- Glade, qui permet de construire des interfaces Gnome/GTK+ sauvegardées dans des fichiers XML, pouvant être interprétés par une bibliothèque Python spécifique.
- BoaConstructor, inspiré des principes des composants VCL de l'outil *Delphi* de *Borland*, et manipulant *wxPython*, bibliothèque au-dessus de *wxWindows*.
- QtDesigner, sur le même principe que *BoaConstructor* mais pour les bibliothèques *Qt*.

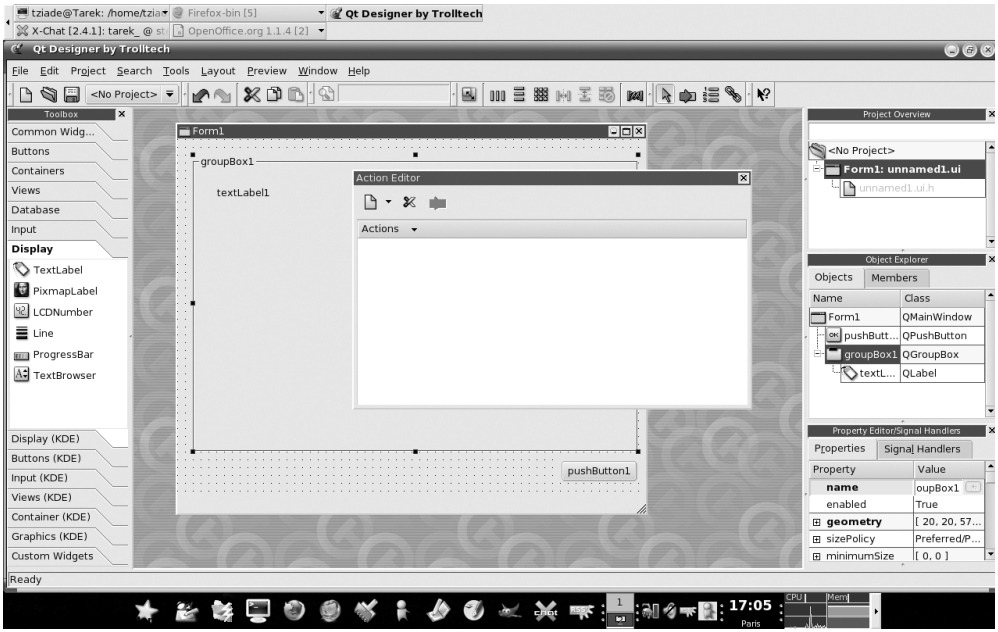

Figure 2–1 QTDesigner à l'œuvre

CULTURE Programmation événementielle

La programmation événementielle, utilisée pour les applications à interface graphique, associe à chaque événement de l'utilisateur une portion de code.
Un événement peut être par exemple l'action de cliquer sur un bouton d'une fenêtre. Le programme lorsqu'il est exécuté, entre dans une boucle infinie qui attend qu'un événement se produise. Lorsque c'est le cas, l'appel est transmis à la portion de code définie pour cet événement, si elle existe, puis la boucle repasse en attente d'un autre événement.
Ce type de programmation est plus détaillé au chapitre 10, dans la partie consacrée à Tkinter.

Maquette de bibliothèque ou Fake

Un autre type de maquette beaucoup moins utilisé mais très pratique est la maquette de bibliothèque. Complément des maquettes d'interfaces, ce genre de prototype permet de simuler un service qui n'a pas encore été développé.

Exemple de prototype de bibliothèque

Prenons l'exemple d'un module de pilotage d'un appareil électronique que l'on souhaite interfacer avec une application graphique.

La mise au point de ce module peut être relativement complexe car elle nécessite l'élaboration de protocoles d'échanges avec l'appareil par le biais du port *IEEE*. De plus, les personnes en charge de développer le reste de l'applicatif n'ont pas à leur disposition ce genre d'appareil et doivent pourtant continuer le développement de l'application comme s'ils en disposaient.

Les méthodes qui seront accessibles aux programmes qui piloteront l'appareil sont quant à elles très simples :

- *start()* : initialise l'appareil ;
- *stop()* : met l'appareil hors-tension ;
- *run(commande)* : lance une commande.

Chacune de ces méthodes renvoie `vrai` lorsque la commande a fonctionné.

Une maquette pour cette bibliothèque pourra se contenter de fournir ces méthodes et de toujours renvoyer un résultat positif sans que l'appareil réel n'ait été appelé :

Prototype

```
import time

class Appareil(object):
    def __init__(self):
        self.started = False

    def start(self):
        self.started = True
        time.sleep(5)
        return 'OK - Listening'

    def stop(self):
        time.sleep(5)
        self.started = False
        return 'OK - Closed'
```

```
def runCommand(self, command, *args):
    time.sleep(2)
    return 'OK %s' % command
```

Ce *Fake* pourra suffire dans un premier temps à construire le reste de l'application en se basant sur l'interface fournie.

PROGRAMMATION Simuler des serveurs à l'aide des Fakes

Les applications qui interagissent avec des serveurs tiers utilisent souvent cet artifice pour simuler leurs présences dans des contextes particuliers comme lors de l'exécution de tests unitaires. Une application de gestion d'e-mails peut implémenter dans ce genre de contexte un faux serveur *IMAP*.

Recherche et calcul scientifique

Certains domaines de recherche sont devenus totalement dépendants de l'informatique. Il existe quantités de logiciels dédiés pour chacun de ces domaines, mais dès lors que le chercheur souhaite sortir des sentiers battus il doit programmer lui-même ses outils.

Dans cet exercice, il cherche un outil de programmation simple à maîtriser, qui permette de manipuler facilement quantité de données et utiliser des bibliothèques de calcul tierces.

Les tableurs comme Excel, qui proposent des fonctionnalités de scripting, sont les outils les plus répandus dans les laboratoires de recherche, car ils permettent de manipuler très simplement les données et de modéliser rapidement des calculs. Mais dès lors que les traitements se complexifient ou qu'il est nécessaire de mettre en place des protocoles particuliers, les tableurs atteignent leurs limites.

Facilité de prise en main

Contrairement aux langages de plus bas niveau comme le C, qui nécessitent un certain bagage technique informatique, Python est beaucoup plus simple à maîtriser pour un chercheur qui ne connaît pas la programmation. La gestion de la mémoire, l'utilisation de pointeurs, le typage des variables, et tous les détails de l'implémentation d'un programme sont autant de contraintes qui sont loin des préoccupations premières d'un chercheur, et doivent le rester.

Parallèlement, la facilité avec laquelle une bibliothèque de traitement peut être intégrée au langage comme extension fait de Python un outil de script de choix dans ce domaine.

Création ou utilisation d'outils spécialisés

Prenons l'exemple de la biologie moléculaire. Si le chercheur souhaite confronter des séquences d'ADN à des séquences connues, répertoriées dans un dépôt centralisé comme le dépôt *GenBank*, il doit mettre en place un outil d'accès au serveur distant pour être en mesure de l'interroger puis d'interpréter les fichiers.

Nous avons vu dans les chapitres précédents que le langage Python disposait d'une bibliothèque d'accès FTP simple d'usage. Construire une bibliothèque d'accès aux dépôts *GenBank* n'est pas plus compliqué. Une fois mise au point, cette bibliothèque offre au chercheur la possibilité d'utiliser et de réutiliser ce genre de système dans ses programmes.

En l'occurrence, la bibliothèque d'accès au dépôt *GenBank* et de lecture des fichiers existe déjà : elle fait partie d'un ensemble d'outils Python dédiés à la bio-informatique nommé *Biopython*, créé par des chercheurs en biologie moléculaire. Toujours dans l'esprit des logiciels libres, ces outils sont mis à disposition de tous sur Internet.

> **RECHERCHE Le projet GenBank**
>
> La base de données GenBank (http://www.ncbi.nlm.nih.gov/) est un projet international de regroupement de séquences de *nucléotides* et leur traduction en *protéines.* Ces données sont fournies par des centres de séquençages du monde entier et sont librement consultables en ligne.

Applications de gestion

Les applications de gestion peuvent être définies comme des logiciels qui traitent un problème métier particulier, comme :

- la gestion de stocks ;
- la gestion de la relation client ;
- la gestion financière ;
- etc.

Ces logiciels se caractérisent en général par :

- une interface utilisateur pour saisir, visualiser et manipuler des données ;
- un besoin de stockage de données qui peut parfois être assez conséquent en taille ;
- une standardisation des flux d'entrées et de sorties pour intégrer le programme au parc applicatif existant.

Interface utilisateur

Outre la conception et l'enchaînement d'écrans décrits dans la partie concernant le prototypage, une application de gestion a un besoin fondamental d'*ergonomie*. Lorsque de simples maquettes peuvent se contenter dans la plupart des cas des composants visuels de base, il s'avère souvent nécessaire de créer ses propres composants pour de véritables applications. En pratique, la création d'une interface en adéquation avec les besoins métier et la nature des données peut peser très lourd dans la balance lorsque l'utilisateur teste l'outil.

Prenons l'exemple de l'application *GRAMPS* (http://gramps-project.org/). Ce logiciel de gestion de généalogie, écrit en Python, offre une interface de visualisation des liens de parenté entre des personnes. Cette fonctionnalité prend tout son sens grâce au composant spécifiquement développé pour afficher des arbres généalogiques.

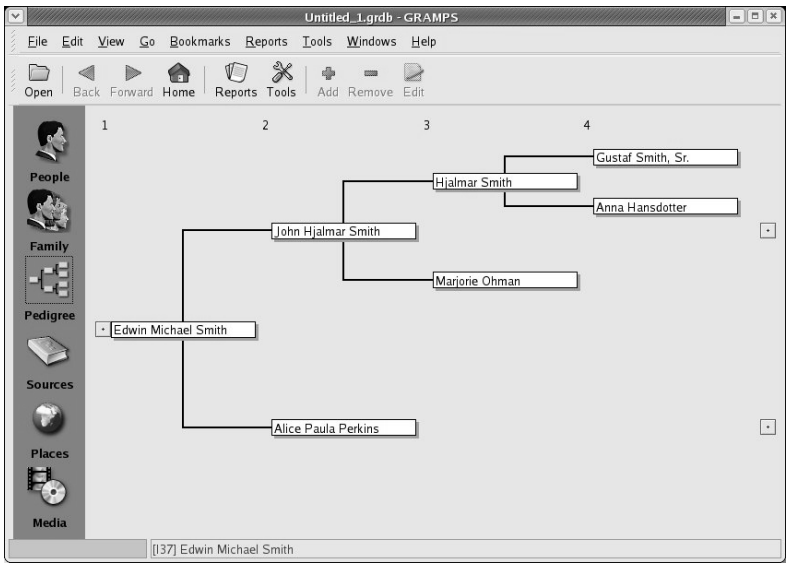

Figure 2–2
Visualisation des liens de parenté avec GRAMPS

Tous les *kits* à disposition du développeur Python fournissent un *framework* de création de nouveaux *widgets*.

Stockage de données

Le stockage de données, appelé aussi *persistance*, peut prendre différentes formes en fonction des besoins et des contraintes du programme. Il peut parfois s'agir d'un simple besoin de sauvegarde de paramètres de fonctionnement. Dans ce cas de figure, les *fichiers INI* ou autres *fichiers XML* font l'affaire. Mais lorsque les besoins

de stockage s'étendent, d'autres outils plus en adéquation avec la quantité et la *granularité* des données manipulées doivent prendre le relais.

Sérialisation des objets

Python fournit des fonctionnalités de *sérialisation* des objets intéressantes. La *sérialisation* consiste à sauvegarder sur le système de fichiers l'état d'un objet stocké en mémoire. Cette mécanique peut être par exemple utilisée pour mémoriser l'état d'un objet lorsque l'application se termine, pour pouvoir le restaurer au prochain démarrage. Le principe de sérialisation est aussi très utile dans des *programmes distribués*. Ce mécanisme fonctionne pour tous les objets Python à quelques exceptions, comme nous le verrons dans le chapitre 9.

Exemple de sérialisation d'un objet

```
import cPickle

class MaClasse:
    valeur_1 ='1'
    valeur_2 =5

# création d'un objet
exemple = MaClasse()
exemple.valeur_1= 'je suis modifié'

# sauvegarde
fichier = open('MaClasse.sav', 'wb')
try:
    cPickle.dump(exemple, fichier, 1)
finally:
    fichier.close()

# rechargement
fichier = open('MaClasse.sav', 'rb')
try:
    nouvel_exemple = cPickle.load(fichier)
finally:
    fichier.close()

# vérification des valeurs
print nouvel_exemple.valeur_1
print nouvel_exemple.valeur_2
```

Ce système peut être utilisé pour les besoins de sauvegarde de tout type de programme, mais il impose un certain nombre de contraintes au développeur. Une des problématiques les plus importantes est que ce fonctionnement introduit une dépendance forte entre le code et les données : si ce système est utilisé pour des sauvegardes durables, toute modification des attributs d'une classe rend les sauvegardes précédentes caduques. Les évolutions du code sont donc plus complexes à gérer.

Une application de gestion qui travaille avec des données qui peuvent parfois provenir d'autres sources et dont le format est imposé, optera pour un stockage plus classique.

Les bases de données relationnelles

Outre tous les connecteurs existants pour la quasi-totalité des bases de données du marché, il existe un SGBD léger codé en Python nommé *GadflyB5* (http://gadfly.sourceforge.net/). Capable de supporter une petite partie de *la norme ODBC 2.0* et utilisé par le biais du langage SQL, Gadfly est parfois très pratique pour mettre en œuvre des programmes client-serveur aux besoins de stockage limités.

Exemple d'utilisation de Gadfly

```
import gadfly

# création d'une base
connection = gadfly.gadfly()
connection.startup('base.gdf', '/home/tziade')
cursor = connection.cursor()

# création d'un table "personne"
cursor.execute('create table personne (nom varchar, prenom varchar)')

# ajout d'une entrée
cursor.execute("insert into personne(nom, prenom) values ('Ziadé',
'Tarek')")
connection.commit()

# lecture
cursor.execute('select * from personne')
print cursor.fetchall()

connection.close()
```

Comme Gadfly respecte ODBC 2.0 et la norme SQL, il reste possible, si les besoins augmentent, de changer le moteur de base utilisé sans modifier le code.

Applications web

Les applications web sont des applications qui mettent en jeu la quasi-totalité des technologies informatiques actuelles.

La conception d'un Intranet nécessite couramment la mise en œuvre :

- d'annuaires LDAP ;
- de gestion de flux de données variés ;
- de systèmes distribués ;
- d'un système de publication web avancé ;
- etc.

Une application web est bien souvent la brique centrale d'un système d'information, et doit offrir aux développeurs des outils souples et modulaires pour implémenter toutes les fonctionnalités nécessaires, s'intégrer à un parc applicatif, et s'interfacer avec des applications tierces qui participent aux services fournis par l'applicatif.

Les deux frameworks Python majeurs, qui tentent de réunir ces qualités en offrant un maximum d'outils de construction d'applicatifs web, sont Zope et Twisted.

Le framework Zope est un des plus gros projets Open Source Python, et de nombreuses évolutions et innovations du langage sont issues de ce framework et de sa communauté très active.

De nouveaux frameworks émergent également, comme :

- Spyce ;
- Quixote ;
- Turbogears ;
- Django.

Tous ces frameworks sont très actifs et propulsent Python sur le devant de la scène en matière de développement web.

En un mot...

Même si Python est beaucoup plus à l'aise dans certains domaines vus dans ce chapitre, comme la programmation système ou le prototypage, ses facultés d'extension et son ouverture lui permettent de s'adapter relativement facilement à de nouveaux contextes.

Il n'est plus rare par exemple de rencontrer dans le monde industriel des applications critiques dont les couches supérieures sont codées en Python.

Le prochain chapitre présente l'installation de Python et son paramétrage, ainsi qu'un tour d'horizon de quelques éditeurs de code.