Chapitre 1-4
Installer son environnement de travail

1. Introduction

Il ne s'agit ici que de CPython, l'implémentation de référence de Python, et non de PyPy ou Jython.
Quel que soit votre système d'exploitation, vous pouvez installer Python en lisant ce chapitre puis, dans un second temps, installer des bibliothèques tierces au gré de vos besoins (cf. section Installer une bibliothèque tierce) et vous pourrez créer des environnements virtuels (cf. section Créer un environnement virtuel).
Si vous souhaitez installer d'un seul coup Python ainsi que Jupyter (anciennement IPython) et la plupart des bibliothèques scientifiques ou d'analyse de données, vous pouvez aller directement à la section Installer Anaconda, pour installer celui-ci en lieu et place de Python. Vous disposerez alors d'autres méthodes pour gérer les environnements virtuels et pour installer des bibliothèques tierces.

2. Installer Python

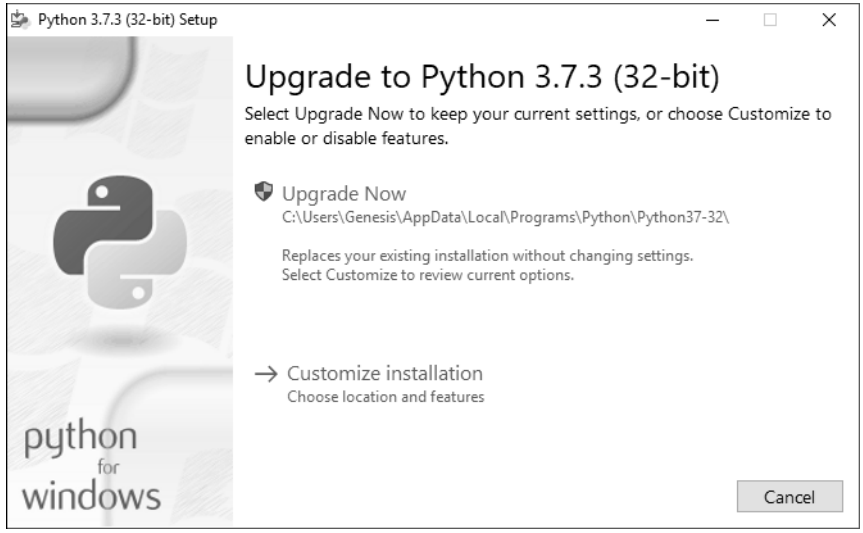
2.1 Pour Windows

Le système d'exploitation Windows requiert usuellement l'utilisation d'un installateur pour pouvoir installer un logiciel quel qu'il soit. Si vous disposez de Windows, vous devriez en avoir l'habitude. Python ne déroge pas à la règle.
Pour installer Python, vous devez donc aller sur le site officiel (https://www.python.org/downloads/) pour télécharger l'installateur adéquat. Comme vous pourrez le constater, on vous met en avant un accès rapide à la dernière version (au moment où ces lignes sont écrites, la 3.11.0), puis un accès aux dernières versions encore actives (actuellement la version 3.10 qui reçoit encore des corrections d'anomalies, puis les versions 3.9 à 3.7 qui reçoivent des corrections de sécurité uniquement).

Il est également possible de télécharger la toute dernière version de la branche 2.7 qui est en fin de vie (elle n'est plus mise à jour), car il existe encore de nombreux projets n'ayant pas encore migré.
Le support correctif dure 2 ans après la première sortie de la version et le support de sécurité dure 5 ans.
Pour notre part, nous vous conseillons la dernière 3.x, mais vous êtes libre d'installer celle que vous souhaitez ou même d'en installer plusieurs suivant vos contraintes, il n'y a pas d'objection à cela.
Une fois le téléchargement effectué, vous devez lancer l'installateur (et éventuellement passer quelques protections de votre système qui vous demande d'accorder votre confiance à cet installateur), pour observer l'écran suivant :

Comme vous pouvez le constater, il est possible de personnaliser l'installation en choisissant le chemin d'installation du logiciel ou en choisissant de ne pas sélectionner quelques fonctionnalités, mais nous ne le conseillons pas.
Nous vous recommandons en revanche de cocher la case **Add python.exe to PATH** afin de configurer la variable PATH du terminal pour rendre Python accessible plus facilement.

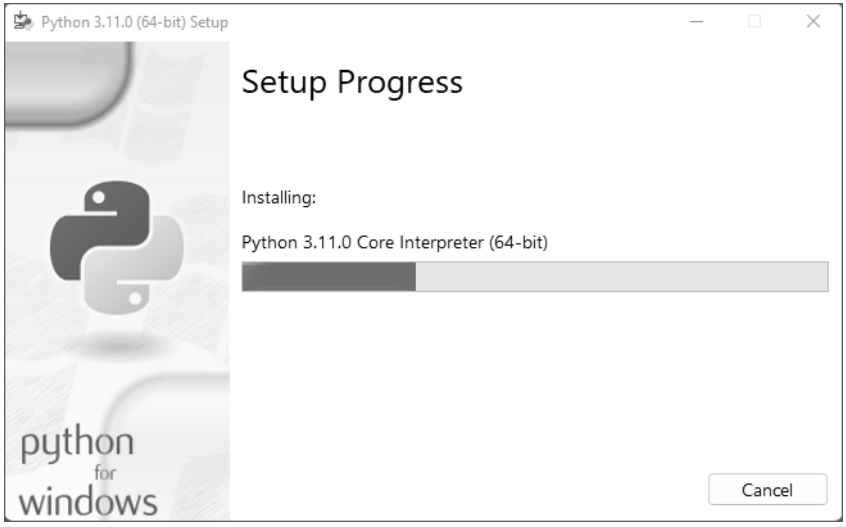
Si vous avez déjà une ancienne version de Python installée de la même branche (dans cet exemple, Python 3.7.2 est déjà installé), vous pourrez la mettre à jour à l'aide du même installeur :



Par contre, si vous avez déjà la version 3.7.1 et que vous installez la version 3.11, cette dernière ne viendra pas remplacer la précédente, mais s'installera à côté. Si vous souhaitez remplacer, il vous faudra donc désinstaller proprement la toute dernière version installée, ce qu'il est possible de faire en relançant l'installeur d'origine.

Nous vous encourageons à garder les installeurs sur votre PC, car ils pourraient devenir indisponibles au téléchargement si trop vieux.

Quel que soit le scénario, vous arriverez devant un écran vous montrant la progression de l'installation et vous n'aurez qu'à fermer la fenêtre une fois celle-ci terminée :



Vous êtes maintenant prêt à utiliser Python.

2.2 Pour Mac

Il faut savoir qu'une version de Python est déjà préinstallée sur Mac, car Mac OS X l'utilise pour ses propres besoins et Python est intégré à son propre cycle de développement. Cependant, si vous souhaitez une version différente de celle qui est déjà présente, vous pouvez l'installer, sachant qu' il n'y a pas de contre-indication à posséder plusieurs versions de Python sur la même machine.

Pour installer Python sur Mac OS X, la procédure à suivre est similaire à celle pour Windows. Il faut donc se rendre sur le site officiel (https://www.python.org/downloads/mac-osx/), télécharger un installateur correspondant à sa configuration et suivre les étapes.

Pour les utilisateurs de Mac, sachez que Python dispose d'une bonne intégration de ses spécificités, en particulier vis-à-vis de Objective-C, le langage de programmation avec lequel est développé Mac OS X, et Cocoa, interface de programmation de Mac OS X.

2.3 Pour GNU/Linux et BSD

Les différentes distributions libres utilisent nativement Python, notamment pour des parties sensibles. Python y est donc tout naturellement déjà installé, généralement sous la dernière version de la branche 2.x. Cependant, ici comme ailleurs, il n'y a pas d'objections à utiliser plusieurs versions de Python.
Le plus simple reste d'utiliser votre gestionnaire de paquets, ce qui peut se faire via un outil graphique, comme Synaptic pour Debian :

Il suffit alors de faire une recherche sur le mot-clé **`python`** pour voir les différentes versions (sur une ancienne Debian, ce sont Python 2.6, 2.7 et 3.2).
Par contre, tous les paquets python3-xxxxx que vous pouvez voir ici sont des bibliothèques tierces et non Python lui-même. Nous en parlerons plus tard dans ce chapitre.
Une fois les paquets souhaités sélectionnés, il ne manque plus qu'à les installer en cliquant sur le bouton **Appliquer**.
Notez que tout ceci peut se faire par la simple ligne de commande, toujours en utilisant votre gestionnaire de paquets qui peut être apt-get, aptitude, yum, emerge, pkg_add ou autre.
Voici par exemple pour une distribution Debian ou Ubuntu :

```
$ sudo aptitude install python3
```

Ceci ne permet cependant pas de choisir la version que l'on souhaite, à moins d'aller trouver des sources alternatives. Si l'on veut avoir la toute dernière version de Python, il faudra la plupart du temps passer par la compilation.

2.4 Par la compilation

Compiler Python n'est pas en soi une tâche très complexe. C'est par contre souvent une tâche imposée lorsque l'on ne travaille pas avec des conteneurs. En effet, en entreprise, on développe souvent des applications qui sont destinées à être hébergées. Il est alors impératif de travailler sur votre propre poste avec une version de Python qui soit la même que celle existante sur la machine de production.

Sous GNU/Linux, mais aussi sous d'autres systèmes, il est possible de compiler la version de Python que l'on souhaite. Après tout, Python n'est rien d'autre qu'un programme écrit en C. Pour ce faire, il faut aller télécharger le code source (https://www.python.org/downloads/source/), qui prend la forme d'une archive, puis décompresser celle-ci, se placer dans le répertoire ainsi obtenu et taper ces quelques commandes :

```
$ ./configure –prefix=/path/to/my/python/directory
$ make
$ sudo make altinstall
```

Notez que dans cette dernière ligne, nous n'utilisons pas la commande **`make install`**, qui aurait pour effet de remplacer votre Python système par le Python que vous compilez, ce qui pourrait avoir des conséquences indésirables voire désastreuses.

Notez également que vous choisissez lors de la configuration le chemin dans lequel vous placerez vos bibliothèques Python. En général, l'usage veut que l'on utilise **`/opt`**, mais il n'y a pas de règle, tout dépend des pratiques de votre entreprise ou votre expérience en la matière.

Si vous venez d'installer Python 3.5 par cette méthode, vous aurez alors maintenant accès à ce programme en l'appelant ainsi, depuis votre terminal :

```
$ python3.5
```

Par cette même méthode, vous pouvez installer les dernières versions (https://www.python.org/download/pre-releases/) de Python qui ne sont pas encore sorties (alphas ou betas), ce qui vous permet de les tester en avant-première !

Notons que, par cette méthode, toutes les bibliothèques de Python ne fonctionneront pas. En effet, lorsqu'elles ont besoin d'autres bibliothèques C, il faut effectuer des compilations croisées et utiliser les différents en-têtes de ces bibliothèques. C'est le cas par exemple pour faire fonctionner Curses, ReportLab (génération de fichiers PDF) ou encore PyUSB (accès aux ports matériels USB).

Dans ce cas-là, la commande **`./configure`** devra recevoir des arguments supplémentaires et vous devrez trouver un tutoriel en ligne pour vous indiquer la démarche, laquelle peut être plus ou moins complexe.

2.5 Pour un smartphone

Installer une machine virtuelle Python sur un smartphone est possible. Pour Android, la procédure est assez simple puisqu'il existe un produit dédié (http://qpython.com/), tout comme sur Windows Phone (https://apps.microsoft.com/store/detail/python-39/9P7QFQMJRFP7). Pour iOS, c'est une autre paire de manches (https://github.com/linusyang/python-for-ios) étant donné que l'utilisateur est enfermé dans un système sur lequel il n'a aucun contrôle.

3. Installer une bibliothèque tierce

Remarque

Si vous abhorrez le terminal, sachez que vous pouvez installer une bibliothèque tierce depuis votre IDE, ce qui sera probablement plus aisé pour vous.

3.1 À partir de Python 3.4

Pour installer une bibliothèque tierce, vous devez simplement connaître son nom. Celui-ci est généralement assez intuitif. Par exemple, la bibliothèque permettant de communiquer avec un serveur Redis s'appelle redis.
Il peut y avoir des variations. Par exemple, la bibliothèque de référence pour traiter du XML est lxml et, plus complexe, celle pour BeautifulSoup est bs4. En recherchant comment répondre à un besoin sur le Net ou sur PyPi (https://pypi.python.org/pypi), vous trouverez rapidement une bibliothèque de référence.
Sur des sujets plus confidentiels, il vous arrivera de trouver plusieurs petites bibliothèques. Vous pourrez alors les tester et choisir celle que vous utiliserez pour votre projet.
Sachez que vous pouvez aussi conduire une recherche directement depuis votre terminal :

```
$ pip search xml
$ pip search soup
```

Cela vous donnera une liste de bibliothèques accompagnée d'une courte description, à la manière de ce que font les gestionnaires de paquets sous Linux (lesquels sont écrits en Python, au passage).
Sachez que **`pip`** existe quel que soit votre système d'exploitation (vous devez être familier avec le terminal de votre système, cependant) et que depuis la version 3.4 de Python, il est installé automatiquement avec celui-ci. Si ce n'est pas votre cas, consultez la section suivante : Pour une version inférieure à Python 3.4.
`pip` est un outil formidable. Si vous utilisez une version de Python qui est celle du système, vous utiliserez alors la commande **`pip`** pour gérer les bibliothèques. Si vous utilisez une autre version, telle que Python 3.5, alors vous utiliserez la commande **`pip-3.5`**. Pour Python 3.3, ce sera **`pip-3.3`**. Dans les exemples suivants, il vous faudra prendre en compte cette particularité.
Cet outil vous permettra d'installer une bibliothèque à sa dernière version ainsi que toutes les bibliothèques dépendantes. En effet, il n'est pas rare qu'une bibliothèque de Python ait besoin d'une autre bibliothèque (ou de plusieurs) pour fonctionner. Par exemple, l'installation de redis se fait par cette commande :

```
$ pip install redis
```

On peut aussi choisir la version à installer :

```
$ pip install -Iv redis==2.10.5
```

Ou mettre à jour la bibliothèque à une version précise :

```
$ pip install -U redis==2.10.5
```

Ou à la dernière version :

```
$ pip install -U redis
```

Et on peut la désinstaller :

```
$ pip uninstall redis
```

Une fonctionnalité très importante permet d'obtenir la liste des bibliothèques déjà installées (quelle que soit la manière dont elles ont été installées) :

```
$ pip freeze
```

Ce que l'on peut mettre dans un fichier :

```
$ pip freeze > requirements.txt
```

Pour installer tous les paquets ainsi listés, il faut procéder ainsi :

```
$ pip install -r requirements/base.txt
```

Cette méthode est particulièrement utile dans le cadre d'un environnement virtuel ; nous y reviendrons.
Il est possible de retrouver des informations sur un paquet déjà installé :

```
$ pip show django-redis
---
Name: django-redis
Version: 4.3.0
Location: /path/to/my/env/lib/python3.4/site-packages
Requires: redis
```

On voit ici que le paquet **`django-redis`** a une dépendance vers **`redis`** : en l'installant, on installe automatiquement **`redis`**.
Mettre à jour ce paquet met à jour automatiquement les dépendances :

```
$ pip install -U django-redis
```

Si on ne veut pas mettre à jour les dépendances, on peut procéder ainsi :

```
$ pip install -U --no-deps django-redis
```

On peut aussi installer plusieurs bibliothèques en même temps :

```
$ pip install django-redis==4.3.0 bs4 lxml
```

Cette commande installera donc automatiquement redis s'il n'est pas installé, car il est déclaré comme dépendance.
Cette commande a cependant des limites. En effet, si vous installez une bibliothèque tierce qui utilise une bibliothèque C, vous devrez disposer des en-têtes C correspondants (paquets **`dev`** pour Debian ou **`devel`** pour Fedora). Il faut donc avoir un peu de pratique dans ce genre de situation pour savoir déjouer ces pièges.

Chapitre 3
Gestion des styles et apparences en PyQt6

1. Introduction

PyQt6 apporte quelques changements par rapport à PyQt5, notamment au niveau de certains modules, méthodes, et classes. Cependant, les principes de gestion des styles restent similaires, offrant la flexibilité de personnaliser l'apparence des applications.

Dans ce chapitre, nous approfondirons l'utilisation des polices de caractères, des images et des icônes pour enrichir visuellement nos applications PyQt. Nous explorerons plus largement la gestion des **ressources** dans PyQt. Nous détaillerons également l'utilisation de `QPalette`, `QBrush` et des **feuilles de style** en PyQt, en découvrant notamment les *Qt Style Sheets* (QSS). Enfin, nous apprendrons à créer nos propres styles personnalisés pour PyQt.

Pour assimiler progressivement les concepts abordés, nous développerons une application qui servira de fil conducteur tout au long du chapitre. Il s'agit d'un formulaire très simple permettant de saisir un nom, un prénom (dans des zones de texte), un loisir préféré (dans une *combobox*), ainsi que la possession (ou non) d'un vélo (précision à l'aide d'une *checkbox*).

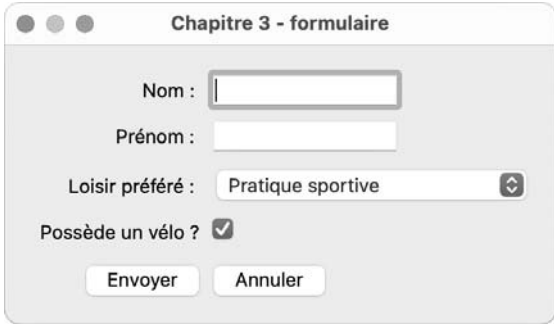
Le code de cette application est le suivant :

```
import sys
from PyQt6.QtWidgets import (
   QApplication,
```

```
    QWidget,
    QPushButton,
    QCheckBox,
    QFormLayout,
    QLabel,
    QLineEdit,
    QComboBox
)
from PyQt6.QtCore import Qt

class Fenetre(QWidget):
    def __init__(self):
        super().__init__()

        self.setGeometry(100, 100, 250, 100)
        self.setWindowTitle("Chapitre 3 - formulaire")

        self.disposition = QFormLayout()

        # Champ "Nom"
        self.nomLabel = QLabel("Nom : ")
        self.nom = QLineEdit()
        self.disposition.addRow(self.nomLabel, self.nom)

        # Champ "Prénom"
        self.prenomLabel = QLabel("Prénom : ")
        self.prenom = QLineEdit()
        self.disposition.addRow(self.prenomLabel, self.prenom)

        # Champ "Loisir préféré"
        self.loisirLabel = QLabel("Loisir préféré : ")
        self.loisir = QComboBox()
        self.loisir.addItems([
            "Pratique sportive",
            "Pratique artistique",
            "Programmation informatique",
            "Voyage"
        ])
        self.disposition.addRow(self.loisirLabel, self.loisir)

        # Checkbox "Possède un vélo ?"
        self.veloLabel = QLabel("Possède un vélo ?")
        self.velo = QCheckBox()
        self.velo.setChecked(True)
```

```
        self.disposition.addRow(self.veloLabel, self.velo)

        # Boutons "Envoyer" et "Annuler"
        self.boutonEnvoyer = QPushButton("Envoyer")
        self.boutonAnnuler = QPushButton("Annuler")
        self.disposition.addRow(self.boutonEnvoyer,
self.boutonAnnuler)

        # Définir la disposition
        self.setLayout(self.disposition)

if __name__ == "__main__":
   app = QApplication(sys.argv)
   fenetre = Fenetre()
   fenetre.show()
   sys.exit(app.exec())
```

Graphiquement cette petite application ressemble à ceci :



Fenêtre de formulaire

À titre informatif voici le code équivalent en **PyQt5** ; si les différences sont effectives, on peut se rendre compte que les différences restent ténues et se limitent à quelques aspects déclaratifs.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton,
QCheckBox from PyQt5.QtWidgets import QFormLayout, QLabel,
QLineEdit, QComboBox


class Fenetre(QWidget): def__init__(self):
```

```
super().__ init__ ()

self.setGeometry(100, 100, 250, 100)
self.setWindowTitle("Chapitre 3 - formulaire")
self.disposition = QFormLayout()

self.nomLabel = QLabel("Nom : ") self.nom = QLineEdit()
self.disposition.addRow(self.nomLabel, self.nom)

self.prenomLabel = QLabel("Prénom : ") self.prenom = QLineEdit()
self.disposition.addRow(self.prenomLabel, self.prenom)

self.loisirLabel = QLabel("Loisir préféré : ")
self.loisir = QComboBox()

self.loisir.addItems(["Pratique sportive", "Pratique artistique",
"Programmation informatique", "Voyage"])
self.disposition.addRow(self.loisirLabel, self.loisir)

self.veloLabel = QLabel("Possède un vélo ?") self.velo =
QCheckBox() self.velo.setChecked(True)
self.disposition.addRow(self.veloLabel, self.velo)

self.boutonEnvoyer = QPushButton("Envoyer")
self.boutonAnnuler = QPushButton("Annuler")


self.disposition.addRow(self.boutonEnvoyer, self.boutonAnnuler)
self.setLayout(self.disposition)
```

Entrons dans le vif du sujet, en commençant par la gestion des polices et donc par `QFont`.

2. Manipulation des polices et l'objet QFont

2.1 Utilisation de QFont

L'objet `QFont` permet d'encapsuler le nom de la police choisie, sa taille, son style éventuel (gras, italique, etc.), son étirement, la dimension des espaces entre chaque caractère et bien d'autres paramètres encore.

Pour l'utiliser, on commence par déclarer l'import adéquat.

```
from PyQt6.QtGui import QFont
```

On instancie ensuite l'objet de type `QFont`, par exemple ci-dessous dans lequel on déclare la police de nom **Chalkduster**, en taille **18** et avec une stylisation en **gras**. On active également la mise en **italique**.

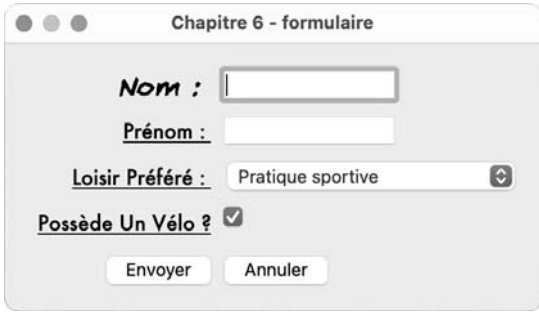
```
maFont = QFont('Chalkduster', 18, QFont.Weight.Bold)
maFont.setItalic(True)
```

Une fois notre police PyQt créée, on peut l'utiliser sur nos différents widgets grâce à la méthode `setFont`. Ainsi, en reprenant l'exemple précédent, on change la police de différents labels de l'interface.

```
import sys
from PyQt6.QtWidgets import (
   QApplication, QWidget, QPushButton, QCheckBox,
   QFormLayout, QLabel, QLineEdit, QComboBox
)
from PyQt6.QtGui import QFont
from PyQt6.QtCore import Qt


# Première police
maFont = QFont('Chalkduster', 18, QFont.Weight.Bold)
maFont.setItalic(True)

# Deuxième police
maFont2 = QFont()
maFont2.setFamily('Futura')
maFont2.setPointSize(15)
maFont2.setCapitalization(QFont.Capitalization.Capitalize)
maFont2.setWeight(QFont.Weight.Medium)
```

```
maFont2.setUnderline(True)


class Fenetre(QWidget):
   def __init__(self):
       super().__init__()

       self.setGeometry(100, 100, 250, 100)
       self.setWindowTitle("Chapitre 3 - formulaire")
       self.disposition = QFormLayout()

       self.nomLabel = QLabel("Nom : ")
       self.nom = QLineEdit()
       self.disposition.addRow(self.nomLabel, self.nom)

       self.prenomLabel = QLabel("Prénom : ")
       self.prenom = QLineEdit()
       self.disposition.addRow(self.prenomLabel, self.prenom)

       self.loisirLabel = QLabel("Loisir préféré : ")
       self.loisir = QComboBox()
       self.loisir.addItems([
           "Pratique sportive", "Pratique artistique",
           "Programmation informatique", "Voyage"
       ])
       self.disposition.addRow(self.loisirLabel, self.loisir)

       self.veloLabel = QLabel("Possède un vélo ?")
       self.velo = QCheckBox()
       self.velo.setChecked(True)
       self.disposition.addRow(self.veloLabel, self.velo)

       self.boutonEnvoyer = QPushButton("Envoyer")
       self.boutonAnnuler = QPushButton("Annuler")
       self.disposition.addRow(self.boutonEnvoyer,
self.boutonAnnuler)

       self.setLayout(self.disposition)

       # Appliquer les polices
       self.nomLabel.setFont(maFont)
       self.prenomLabel.setFont(maFont2)
       self.loisirLabel.setFont(maFont2)
       self.veloLabel.setFont(maFont2)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    fenetre = Fenetre()
    fenetre.show()
    sys.exit(app.exec())
```



Rendu de l'exemple d'usage de QFont

2.2 La question des polices de caractères non installées

Les polices de caractères utilisées dans vos documents ou définies programmatiquement via PyQt sont généralement préinstallées sur l'ordinateur cible. Elles proviennent soit du système d'exploitation, soit d'une installation manuelle.

Cependant, il est difficile de garantir qu'une police donnée sera disponible sur toutes les machines où votre application PyQt sera exécutée. Il est donc recommandé d'utiliser des polices courantes et d'anticiper l'absence d'une police spécifique sur la machine cible.

Lorsque la police demandée n'est pas disponible, PyQt utilise une police par défaut. Pour comprendre ce comportement et identifier la police effectivement utilisée, on peut utiliser la classe `QFontInfo`.

Exemple pratique avec une police inexistante

On commence par créer un objet `QFont` avec une famille de polices fictive, par exemple `"LA_POLICE_INCONNUE"`, qui n'existe sur aucune machine. Voici comment définir cette police :

```
maFont = QFont()
maFont.setFamily('LA_POLICE_INCONNUE')
maFont.setPointSize(15)
maFont.setCapitalization(QFont.Capitalization.Capitalize)
maFont.setWeight(QFont.Weight.Medium)
```

Lorsque la police n'est pas trouvée, **PyQt** remplace automatiquement cette dernière par une police par défaut. Le code complet pour tester ce comportement est le suivant.

```
import sys
from PyQt6.QtWidgets import (
   QApplication, QWidget, QPushButton, QCheckBox,
   QFormLayout, QLabel, QLineEdit, QComboBox
)
from PyQt6.QtGui import QFont, QFontInfo

# Création de la police inexistante
maFont = QFont()
maFont.setFamily('LA_POLICE_INCONNUE')  # Police inexistante
maFont.setPointSize(15)
maFont.setCapitalization(QFont.Capitalization.Capitalize)
maFont.setWeight(QFont.Weight.Medium)

class Fenetre(QWidget):
   def __init__(self):
       super().__init__()

       self.setGeometry(100, 100, 350, 150)
       self.setWindowTitle("Chapitre 3 - QFontInfo")
       self.disposition = QFormLayout()

       # Création des champs du formulaire
       self.nomLabel = QLabel("Nom : ")
       self.nom = QLineEdit()
       self.nomLabel.setFont(maFont)
# Application de la police inexistante
       self.disposition.addRow(self.nomLabel, self.nom)
```

```
        self.prenomLabel = QLabel("Prénom : ")
        self.prenom = QLineEdit()
        self.disposition.addRow(self.prenomLabel, self.prenom)

        self.loisirLabel = QLabel("Loisir préféré : ")
        self.loisir = QComboBox()
        self.loisir.addItems([
            "Pratique sportive", "Pratique artistique",
            "Programmation informatique", "Voyage"
        ])
        self.disposition.addRow(self.loisirLabel, self.loisir)

        self.veloLabel = QLabel("Possède un vélo ?")
        self.velo = QCheckBox()
        self.velo.setChecked(True)
        self.disposition.addRow(self.veloLabel, self.velo)

        # Boutons
        self.boutonEnvoyer = QPushButton("Envoyer")
        self.boutonAnnuler = QPushButton("Annuler")
        self.disposition.addRow(self.boutonEnvoyer,
self.boutonAnnuler)

        self.setLayout(self.disposition)

        # Identification de la police effectivement utilisée
        testFont = QFontInfo(maFont)
        print('Police utilisée par défaut :', testFont.family())

if __name__ == "__main__":
   app = QApplication(sys.argv)
   fenetre = Fenetre()
   fenetre.show()
   sys.exit(app.exec())
```

Quand on exécute ce programme dans le terminal, ici sur une machine macOS, on obtient le retour suivant (en plus de l'affichage de la fenêtre utilisant la police par défaut, ici `.AppleSystemUIFont`).

```
> python3 QFontInfo_Police_non_existante.py
qt.qpa.fonts: Populating font family aliases took 48 ms. Replace
uses of missing font family "LA_POLICE_INCONNUE" with one that
exists to avoid this cost.
```