Découvrez le framework Django

Un site de réservation pour disquaire

Plus précisément, la boutique aimerait gérer son stock dans une interface d'administration et offrir à sa clientèle un catalogue affichant les produits disponibles.

Le catalogue en ligne aura les **fonctionnalités suivantes** :

* Affichage des derniers CD ajoutés en base dans la page d'accueil,
* Visualisation d'un CD en cliquant sur sa pochette ou sur son titre,
* Recherche par auteur,
* Demande de réservation par un formulaire présent sur chaque page de CD.

L'**interface d'administration** sera **privée** et accessible par un seul utilisateur. Voici les fonctionnalités demandées :

* Ajout de disques, modification et suppression.
* Recherche d'un disque par titre d'album ou référence.
* Liste des demandes de réservation.
* Possibilité d'indiquer qu'un album est indisponible. Dans ce cas, il n'apparaît plus dans le catalogue public.

De quoi aurez-vous besoin pour réaliser ce projet ?

* Une **base de données** : vous devrez conserver de nombreuses données ! Les albums, les artistes, les réservations... Afin de les gérer efficacement, vous aurez besoin d'une base de données.
* Un service qui vous permet d'**interagir avec la base de données** de manière sécurisée.
* Un **serveur web** dont le rôle va être de recevoir des requêtes HTTP, de les traiter et de renvoyer une réponse HTTP conforme.
* Une **interface d'administration** pour gérer le contenu du site.
* **Bien d'autres éléments** que nous verrons au fur et à mesure.

Le but principal de Django, qui a présidé à sa création, est de **simplifier la création de sites web complexes** utilisant une base de données

*Quel genre de site puis-je réaliser avec Django ?*

Vous pouvez aller très loin avec ce framework ! Par exemple, saviez-vous que les sites suivants l'utilisent ? [Instagram](https://www.instagram.com/), [Pinterest](https://fr.pinterest.com/), [National Geographic](https://www.nationalgeographic.com/), [OpenStack](https://www.openstack.org/), [AutoLib](https://www.autolib.eu/fr/), [Liberation](http://www.liberation.fr/), ...

### Installer Django

Créez un nouveau projet puis, si ce n'est déjà fait, initialisez un environnement virtuel.

sudo install virtualenv ou sudo pip install virtualenv

$ mkdir disquaire

$ cd disquaire

$ virtualenv env -p python3

$ . env/bin/activate

Puis installez Django via PyPI :

$ pip install django

Quand l'installation est terminée, vérifiez que la librairie est bien disponible en tapant la commande suivante qui renvoie la version de Django :

$ python -m django --version

1.11.3

Utilisez la commande django-admin startproject projectname\_project pour créer la structure du projet :

$ django-admin startproject disquaire\_project

Rien ne s'affiche mais c'est normal ! Si vous listez vos fichiers, vous verrez qu'un nouveau dossier a fait son apparition : disquaire\_project.

En voici le contenu :

disquaire\_project/

manage.py

disquaire\_project/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

### Travailler avec une base de données PostgreSQL

### Installer la librairie psycopg2

Afin d'utiliser une base PostgreSQL dans un projet Django, il faut installer la librairie Psycopg2 :

$ pip install psycopg2

#### Installer PostgreSQL

Utilisateurs de Mac : vous pouvez installer PostgreSQL via [HomeBrew](https://brew.sh/) en tapant la commande brew install postgresql

# Linux downloads (Debian)

PostgreSQL is available in all Debian versions by default. However, the stable versions of Debians "snapshot" a specific version of PostgreSQL that is then supported throughout the lifetime of that Debian version. The PostgreSQL project also maintains an [apt repository](https://www.postgresql.org/download/linux/debian/#apt) with all supported of PostgreSQL available.

## PostgreSQL Apt Repository

If the version included in your version of Debian is not the one you want, you can use the [PostgreSQL Apt Repository](https://apt.postgresql.org). This repository will integrate with your normal systems and patch management, and provide automatic updates for all supported versions of PostgreSQL throughout the support [lifetime](https://www.postgresql.org/support/versioning/) of PostgreSQL.

The PostgreSQL apt repository supports the currently supported stable versions of Debian:

* Stretch (9.x)
* Jessie (8.x)
* Wheezy (7.x)

Create the file /etc/apt/sources.list.d/pgdg.list, and add a line for the repository:

deb http://apt.postgresql.org/pub/repos/apt/ YOUR\_DEBIAN\_VERSION\_HERE-pgdg main

Import the repository signing key, and update the package lists:

wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

sudo apt-get update

apt-get install postgresql-10

Vérifiez que l'installation s'est bien effectuée en tapant la commande psql dans un terminal.

Pour en sortir, tapez la commande \q

Pour interagir avec une base PostgreSQL, il faut donc au préalable démarrer les serveurs. Tapez la commande suivante :

$ postgres -D /usr/local/pgsql/data

Pour les utilisateurs de Mac ayant installé PostgreSQL avec Homebrew : vous pouvez lancer le serveur en tapant la commande brew services start postgresql. Pour le quitter, tapez brew services stop postgresql.

#### Créer une base de données

Bien ! Les serveurs sont lancés et PostgreSQL attend vos instructions ! Commencez par créer une nouvelle base de données grâce à la commande createdb -O yourusername dbname :

$ createdb -O celinems disquaire

* psql lance la console PostgreSQL
* \l liste les bases présentes dans notre ordinateur

#### Déclarer une base de données dans un projet Django

La dernière étape est d'informer Django que nous souhaitons utiliser la base de données que nous venons juste de créer. Après tout, il ne le sait pas encore !

Ouvrez le fichier disquaire\_project/settings.py. Remplacez le contenu de la constante DATABASES par les lignes suivantes :

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql', # on utilise l'adaptateur postgresql

'NAME': 'disquaire', # le nom de notre base de donnees creee precedemment

'USER': 'celinems', # attention : remplacez par votre nom d'utilisateur

'PASSWORD': '',

'HOST': '',

'PORT': '5432',

}

}

Puis lancez le serveur de Django en tapant la commande suivante :

$ ./manage.py runserver

Django vous informe que le projet ne fonctionnera peut-être pas correctement sans les migrations. Tapez ctrl + c pour arrêter le serveur. Puis tapez la commande suivante :

$ ./manage.py migrate

Cette commande effectue des modifications dans la base de données nécessaires pour faire tourner le projet.

Ouvrez un navigateur et tapez l'URL contenue dans le message ci-dessus. Vous devriez voir apparaître cette belle page :

### Organisation d'un projet Django

Le répertoire disquaire\_project rassemble tous les réglages de notre projet. Pour l'instant, il contient les quatre fichiers suivants :

* \_\_init\_\_.py : Déclare ce répertoire comme étant un module. Cela vous permettra d'importer les réglages ailleurs.
* settings.py : Les réglages généraux du projet. Nous détaillerons ce fichier juste après.
* urls.py : Les routes utilisées dans votre projet. Nous expliciterons ce concept plus tard dans le cours mais retenez que tous les "chemins" que vous entrez dans votre navigateur pour accéder à une page de votre application sont regroupés ici.
* wsgi.py : Réglages du serveur web.

Ouvrez le fichier settings.py et lisez-le attentivement. Parmi les réglages qui y sont listés, j'aimerais attirer votre attention sur ceux-ci :

* DEBUG = True : si votre application rencontre une erreur, le navigateur n'affichera pas une page blanche mais des informations sur le bug rencontré. C'est extrêmement pratique en développement !
* LANGUAGE\_CODE = "en-us" : La langue utilisée dans le projet, notamment dans l'interface d'administration. Remplacez-la dès maintenant par fr.
* TIME\_ZONE = 'UTC' : le fuseau horaire que doit suivre votre projet. Remplacez 'UTC' par 'Europe/Paris'. Pour en savoir plus, consultez la [liste des fuseaux horaires](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones/).

### Installer une application

Je vous propose d'installer [Django Debug Toolbar](https://github.com/jazzband/django-debug-toolbar). Que fait-elle ? Cette application ajoute de nombreuses informations dans une barre de navigation à droite de l'écran :

pip install django-debug-toolbar

Lorsque l'installation est terminée, ouvrez le fichier settings.py pour indiquer à Django que vous souhaitez utiliser cette application :

INSTALLED\_APPS = [

# ...

'django.contrib.staticfiles',

'debug\_toolbar',

]

C'est tout ce que vous avez à faire pour installer une nouvelle application !

Néanmoins, chaque librairie est différente. Certaines ont besoin d'accéder aux tables de la base de données pour y lire certains réglages ! Django Debug Toolbar n'échappe pas à la règle.

*settings.py*

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

# ...,

'debug\_toolbar.middleware.DebugToolbarMiddleware',

]

# ...

INTERNAL\_IPS = ['127.0.0.1']

Puis ouvrez le fichier *urls.py* et modifiez-le ainsi :

from django.conf import settings

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

]

if settings.DEBUG:

import debug\_toolbar

urlpatterns = [

url(r'^\_\_debug\_\_/', include(debug\_toolbar.urls)),

] + urlpatterns

Relancez le serveur et rendez-vous de nouveau sur la page d'accueil. Elle a changé ! Vous visualisez désormais une page d'erreur (code 404) et une belle barre de débogage a fait son apparition à droite ! Bravo ! 👏

Avez-vous l'impression que c'est un peu magique ? Oui ? C'est normal ! Vous venez tout juste d'ajouter une librairie à votre projet. Il est temps de démystifier tout cela ! Créez votre première application : store

### Créer une application

Cette application sera responsable des réservations et du catalogue à afficher sur le site.

Comment créer une nouvelle application ? Django contient une commande dédiée :

 django-admin startapp app\_name

$ django-admin startapp store

Un nouveau répertoire vient d'être créé au même niveau que disquaire\_project : store. Passons en revue les fichiers qu'il contient :

store/

migrations/

\_\_init\_\_.py

admin.py

apps.py

models.py

tests.py

views.py

\_\_init\_\_.py

Voici donc à quoi ils correspondent :

* migrations/ : fichiers qui modifient la structure de la base de données.
* \_\_init\_\_.py : déclare que ce répertoire est une librairie.
* admin.py : configuration de l'espace d'administration.
* apps.py : configuration de l'application (vous n'avez pas besoin de le modifier).
* models.py : modèles de l'application (nous y reviendrons).
* tests.py : fichier qui contiendra les tests.
* views.py : vues de l'application (nous y reviendrons).

Mettez à jour settings.py pour en déclarant votre application :

settings.py

INSTALLED\_APPS = [

'store.apps.StoreConfig',

# ...

]

## Découvrez l'architecture MVT

### Le MVT en théorie

Le MVT représente une **architecture** orientée autour de trois pôles : le **modèle**, la **vue** et le **template**. Elle s'inspire de l'[architecture MVC](https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur), très répandue dans les frameworks web. Son objectif est de séparer les responsabilités de chaque pôle afin que chacun se concentre sur ses tâches.

#### Modèle

Le modèle interagit avec la base de données. Sa mission est de chercher dans une base de donnée les items correspondant à une requête et de renvoyer une réponse facilement exploitable par le programme.

Les modèles s'appuient sur un ORM (Object Relational Mapping, ou Mapping objet-relationnel en français). A quoi cela sert-il ?

La consultation d'une base de données relationnelle se réalise par un langage appelé SQL (Structured Query Language). Sa syntaxe est très différente de Python !

Un **ORM traduit les résultats d'une requête SQL en objets Python** avec lesquels vous pouvez interagir. De même, il permet d'écrire une requête SQL en Python. Un peu comme un traducteur automatique !

Par exemple, la requête suivante renverra tous les items de la tablecontent : Content.objects.all()

Dans un projet Django, chaque application contient un document models.py qui réunit les différents modèles utilisés.

#### Template

Un template est un fichier HTML qui peut recevoir des objets Python et qui est lié à une vue (nous y reviendrons). Il est placé dans le dossier templates.

Concrètement, un template peut interpréter des variables et les afficher. Par exemple, nous pouvons "donner" la variable tom="Tom" au template index.html et ce dernier l'affichera à la place du prénom.

Les templates sont très communs dans le développement web mais également dans d'autres disciplines. Avez-vous déjà utilisé des thèmes dans Power Point ou Wordpress ? Ou bien les gabarits dans InDesign ? La logique est exactement la même. D'ailleurs, la traduction française du mot "template" est "gabarit", même si le premier terme est plus utilisé dans le langage courant. C'est pourquoi j'utiliserai les deux sans distinction.

#### Vue

La vue joue un rôle central dans un projet structuré en MVT : sa responsabilité est de recevoir une requête HTTP et d'y répondre de manière intelligible par le navigateur.

La vue réalise également toutes les actions nécessaires pour répondre à la requête :

* si une interaction avec la base de données est requise, la vue appelle un modèle et récupère les objets renvoyés par ce dernier.
* si un gabarit est nécessaire, la vue l'appelle.

Dans un projet Django, les vues de chaque application sont regroupées dans le document views.py.

Chaque vue est associée à une url.

Les urls d'un projet sont regroupées dans le fichier urls.py.

Une **vue** est une méthode qui génère le contenu à renvoyer à une requête. Il s'agit d'une méthode qui renvoie une réponse à une requête HTTP.

On appelle **route** un schéma d'url correspondant à une vue. A savoir également qu'une vue peut répondre à plusieurs URL.

### Exemple dans Django

Et si je vous disais que Django adopte un processus similaire ?

Imaginons à présent que Michel affronte une crise de flegmatite aiguë. Au lieu de se déplacer au centre commercial, il se rend sur le site "www.chapelier-fou.com".

Sur la page d'accueil, il voit un champ de recherche. Il entre "vert trèfles velours" et appuie sur entrée.

La page de résultat affiche le chapeau de ses rêves. Il le paye puis se rendra chez le chapelier pour le récupérer (il aime bien discuter avec ce chapelier fou).

Analysons la requête effectuée sur le site :

* Michel appuie sur "entrée". **Cela génère une requête HTTP**.
* la requête arrive sur le serveur. C'est à Django de jouer ! Il cherche dans le fichier urls.py **le schéma correspondant** à l'url. Dans le monde physique, cela correspond au moment auquel Michel s'est présenté à l'accueil.
* Chaque schéma est associé à une **vue**. La requête passe alors dans le fichier views.py, plus précisément dans la méthode associée au schéma. Dans le monde réel, cela correspond à l'arrivée de Michel dans la boutique et à son échange avec le chapelier.
* La vue exécute ce qui est demandé. Elle interroge le **modèle** Hat à la recherche d'un chapeau correspondant aux caractéristiques demandées. Cette étape correspond au moment auquel le chapelier se rend dans l'arrière-boutique à la recherche du chapeau demandé.
* La vue continue d'exécuter les instructions. Elle fait appel au **gabarit** adéquat pour formuler sa réponse. Cela correspond au moment auquel le chapelier met le chapeau dans la boite.
* Enfin, la vue renvoie une **réponse HTTP** correcte contenant le code HTML.

## Ajoutez une vue

Dans ce chapitre vous découvrirez comment afficher le message "Salut tout le monde !" à tout visiteur qui arrive sur la page d'accueil, nous ajouterons également une seconde route pour afficher les albums de l'application.

### Structurer une application en suivant le MVT

Par défaut, une application Django contient les fichiers suivants :

* models.py : regroupe les modèles,
* views.py : concentre les vues.

#### Les gabarits

L'application ne comprend également aucun gabarits. Empressons-nous d'en créer un !

Les templates sont, par convention, regroupés dans le répertoire templates d'une application. Créez-le :

$ mkdir store/templates

Pour plus de clarté, ajoutez un sous-répertoire au nom de l'application.

$ mkdir store/templates/store

#### Les URLs

Une URL étant le premier point d'entrée d'une requête, il est logique que le fichier se situe dans le répertoire consacré à la configuration du **projet**. Pourtant, vous y ajouterez de nombreuses autres routes à mesure que le projet croîtra. Il est donc important de regrouper les routes ! C'est pourquoi une bonne pratique consiste à créer un fichier urls.py **par application** et de l'importer dans les routes du projet.

Créez un nouveau document dans l'application :

$ touch store/urls.py

Puis importez-en le contenu dans disquaire\_project/urls.py :

disquaire\_project/urls.py

urlpatterns = [

url(r'^store/', include('store.urls')),

url(r'^admin/', admin.site.urls)

]

La méthode url() associe un schéma de route à un ensemble de vue. Plus précisément, elle prend en premier paramètre une expression régulière représentant un **schéma** et en second paramètre la **vue** à associer.

L'ajout de la méthode url(r'^store/', include('store.urls')), ajoute toutes les routes de store.urls aux routes du projet en les préfixant par store/

Il est impératif que vous sachiez bien utiliser les expressions régulières pour créer de nouvelles routes.

Si vous lancez le serveur, vous remarquerez qu'une erreur apparaît :

$ ./manage.py runserver

Django vous indique qu'il n'existe aucune route dans le fichier que vous essayez d'importer. C'est normal ! Votre fichier est vide à ce stade. Vous le remplirez sous peu.

### **Créer une vue**

#### Ajouter une vue

Votre première vue affichera le message "Salut tout le monde !" lorsque l'utilisateur entrera l'URL http://127.0.0.1:8000/store/

Ouvrez le fichier *views.py* et ajoutez la première vue :

*views.py*

def index(request):

message = "Salut tout le monde !"

return message

Néanmoins, il y a un petit problème : une vue doit retourner une **réponse valide** pouvant être interprétée par un navigateur.

Alors, comment faire ?

Django contient l'objet HttpResponse qui fait justement cela pour vous ! Importez-le et utilisez-le dans la vue :

*views.py*

from django.http import HttpResponse

def index(request):

message = "Salut tout le monde !"

return HttpResponse(message)

#### Ajouter une route

Parfait, la vue est créée ! Ajoutez la route permettant d'y accéder :

store/urls.py

from django.conf.urls import url

from . import views # import views so we can use them in urls.

urlpatterns = [

url(r'^$', views.index), # "/store" will call the method "index" in "views.py"

]

Pas de magie ! La méthode url() associe l'url / à la méthode index du fichier views.py.

Néanmoins, si vous lancez votre serveur et entrez l'URL   http://127.0.0.1:8000/, une page d'erreur apparaîtra ! Avez-vous trouvé pourquoi ? En effet, les routes contenues dans store/urls.py sont appelées dans disquaire\_project/urls.py avec le préfixe store/. Naviguez donc à l'url  http://127.0.0.1:8000/store   et vous verrez apparaitre le magnifique message entré précédemment dans la vue :

### Créer la page d'accueil

Comment faire pour la page d'accueil ? Vous allez procéder légèrement différemment. En effet, vous ne souhaitez pas utiliser de préfixe !

Vous allez donc directement appeler la vue index dans le fichier disquaire\_project/urls.py et réserver la route /store pour une autre action.

Modifiez disquaire\_project/urls.py comme suit :

from store import views

urlpatterns = [

url(r'^$', views.index),

url(r'^store/', include('store.urls')),

url(r'^admin/', admin.site.urls)

]

Pensez à supprimer la route url(r'^$', views.index) du fichier store/urls.py sinon la route sera définie deux fois à deux URLs différentes.

### Lister les albums

Ajoutez une nouvelle vue qui listera tous les albums de votre application ! Par souci de simplification, nous n'utiliserons pas la base de données pour le moment. Ajoutez simplement un jeu de données en pur Python dans models.py :

models.py

# from django.db import models

ARTISTS = {

'francis-cabrel': {'name': 'Francis Cabrel'},

'lej': {'name': 'Elijay'},

'rosana': {'name': 'Rosana'},

'maria-dolores-pradera': {'name': 'María Dolores Pradera'},

}

ALBUMS = [

{'name': 'Sarbacane', 'artists': [ARTISTS['francis-cabrel']]},

{'name': 'La Dalle', 'artists': [ARTISTS['lej']]},

{'name': 'Luna Nueva', 'artists': [ARTISTS['rosana'], ARTISTS['maria-dolores-pradera']]}

]

Suivez la même procédure pour ajouter une nouvelle vue mais avec une étape supplémentaire :

* ajout d'une url dans store/urls.py,
* ajout d'une vue dans views.py
* récupération des données du modèle

urls.py

urlpatterns = [

url(r'^$', views.listing),

]

views.py

from .models import ALBUMS

...

def listing(request):

albums = ["<li>{}</li>".format(album['name']) for album in ALBUMS]

message = """<ul>{}</ul>""".format("\n".join(albums))

return HttpResponse(message)

Naviguez à l'url /store... et la liste des albums s'affiche !

### Une architecture basée sur un protocole que l'on connaît bien : HTTP

Nous allons faire quelques rappels concernant HTTP : il s'agit d'un protocole d'échange entre deux machines. Une API n'est rien d'autre qu'une application capable de recevoir une requête HTTP et rendre une réponse HTTP. Ce qui change d'un site web "classique" est le fait qu'il s'agit de faire des actions plus atomiques, contrairement à une page web HTML. Une API va se charger de la gestion des utilisateurs (ajout, suppression…) ou d'une gestion de produits, ou tout autre ressource.

Revoyons ce que contient une requête et une réponse HTTP.

#### Requête HTTP

Une requête HTTP émane d'un client (tout logiciel dans la capacité de forger une requête). Une requête est constituée des éléments suivants :

1. La première ligne (request line) doit contenir :
   * la **méthode HTTP** ( GET ,  POST ,  PUT ,  PATCH ,  DELETE ,  OPTIONS ,  CONNECT ,  HEAD ou  TRACE )
   * l'**URI**, c'est-à -dire ce qu'il y a après le nom de domaine (exemple :  /users/1 )
   * la **version du protocole** (exemple :  HTTP/1.1 )
2. Les **entêtes** (headers), une entête par ligne, chaque ligne finie par le caractère spécial "retour à la ligne" (CRLF)
3. Le **contenu de la requête** (body), doit être séparé de deux caractères spéciaux "retour à la ligne" ( CRLF CRLF ) - optionnel

#### Réponse HTTP

Une réponse HTTP émane d'un serveur (tout logiciel dans la capacité de forger une réponse HTTP). Une réponse est constituée des éléments suivants :

1. La première ligne (status line) doit contenir :
   * la **version du protocole** utilisée
   * le **code status**
   * l'**équivalent textuel du code status**
2. Les entêtes (headers), une entête par ligne, chaque ligne finie par le caractère spécial "retour à la ligne" (CRLF)
3. Le contenu de la réponse (body), doit être séparé de deux caractères spéciaux "retour à la ligne (CRLFCRLF) - optionnel.

**Perfectionnez-vous en Python - Travaillez dans un environnement virtuel**

Télécharger le dichier dans data :

<https://github.com/oc-courses/perfectionnez_vous_en_python/tree/master>

Créer un fichier parite.py avec ceci :

#! /usr/bin/env python3

# coding: utf-8

def main():

pass

if \_\_name\_\_ == '\_\_main\_\_':

main()

L'objectif des environnements virtuels : créer un environnement comprenant une certaine version de Python et les librairies que vous souhaitez.

Installation de virtual env

pip install virtualenv

Puis déplacez-vous dans votre dossier de travail.

#### Créer et activer un environnement virtuel

Commencez par créer un environnement virtuel en lançant la commande suivante :

virtualenv -p python3 env

Elle crée un dossier env utilisant Python3 dans le répertoire courant.

Pour activer l'environnement, écrivez la commande suivante :

source env/bin/activate

À partir de maintenant, l'interpréteur Python utilisé n'est plus celui de votre système mais bien celui de l'environnement virtuel.

#### Désactiver un environnement virtuel

Pour désactiver un environnement virtuel, tapez simplement deactivate.

#### Désactiver le suivi Git

Si vous utilisez Git (et j'espère bien que oui ! :)), il faut impérativement désactiver le suivi de modifications de votre environnement virtuel. En effet, vous ne voulez pas envoyer sur Github tout votre environnement virtuel !

Pour cela, créez un fichier .gitignore et ajoutez à l'intérieur la ligne suivante :

env/

Elle indique à Git d'ignorer le dossier env.

### Un fichier de dépendances

Les librairies qui ne sont pas standards, qui doivent par conséquent être installées, sont ce que nous appelons des **dépendances**. En effet, le projet est dépendant de leur installation et ne peut pas fonctionner sans.

Imaginons a présent que le programme soit fini. Vous le publiez sur Github afin que toute personne intéressée puisse y accéder.

Comment ces dernières peuvent-elles connaître les différentes librairies utilisées dans le projet ? Elles n'auront pas accès à votre environnement virtuel. Bien sûr, elles peuvent regarder en haut des différents fichiers et chercher les imports. Mais cela est fastidieux et, surtout, il sera impossible de connaître les versions utilisées.

Afin de remédier à ce souci, il existe une convention : créer un fichier requirements.txt qui liste les différentes librairies utilisées ainsi que leur version.

#### Créer un fichier de dépendances

Pypi peut le faire automatiquement pour vous ! Pour cela, exécutez la commande suivante.

pip freeze > requirements.txt

Cette commande "gèle" les librairies utilisées ainsi que leur version en les listant dans un document.

#### Installer des dépendances

Si un fichier requirements.txt existe déjà, vous pouvez facilement en installer toutes les librairies en exécutant la commande suivante :

pip install -r requirements.txt

## Organisez un projet en modules

Créez un dossier data dans le dossier courant. Déplacez-y les fichiers que vous avez téléchargés au premier chapitre de ce cours.

### Organisation du projet

Les deux fichiers ont des fomats différents : CSV et XML. Afin de différencier les analyses, je vous propose de créer un script d'analyse par fichier :

* csv\_analysis.py : analyse du fichier current\_mps.csv
* xml\_analysis.py : analyse du fichier SyceronBrut.xml

Le fichier parite.py, qui existe déjà, servira à lancer les analyses.

Ouvrez csv\_analysis.py et commençons à analyser le fichier CSV.

### Analyse d'un fichier CSV

Nous allons utiliser [la librairie standard  os](https://docs.python.org/3.6/library/os.html)  afin d'interagir avec le système. Elle vous permet d'utiliser des fonctionnalités qui dépendent du système d'exploitation, par exemple écrire ou retrouver le chemin d'un fichier.

Importez-la en haut de votre fichier :

import os

Puis créez une fonction main() , comme précédemment :

def main():

pass

if \_\_name\_\_ == "\_\_main\_\_":

main()

#### Lire un fichier

C'est très simple ! Utilisez la méthode open() qui prend deux paramètres :

* le chemin vers le fichier à lire
* le mode d'ouverture (optionnel) : r ouvre un fichier en lecture, w  vous donne également le droit de le modifie

Le chemin correspond à l'endroit d'où la commande est exécutée. Par exemple, si vous exécutez python csv\_analysis.py , le chemin sera csv\_analysis.py.

Pour l'instant, les scripts sont tous au même niveau. Mais que se passe-t-il si vous créez un dossier analysis qui contient analysis\_cvs.py?

Si vous exécutez la commande python analysis/analysis\_csv.py, le chemin sera analysis/analysis\_csv.py. Mais si vous exécutez la commande directement dans le dossier analysis, le chemin sera alors de nouveau analysis\_csv.py.

Comment indiquer le chemin vers un fichier à l'intérieur d'un script ? Encore une fois, Python vient à notre rescousse ! Vous pouvez utiliser la variable \_\_file\_\_ dont la valeur correspond au chemin vers un fichier. Puis nous pourrons retrouver très facilement le chemin du fichier que nous souhaitons utiliser. Voici comment faire :

*csv\_analysis.py*

import os

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

pass # read first line

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('current\_mps.csv')

Vous pouvez ensuite utiliser la méthode readline() pour lire la première ligne du fichier. Démonstration :

def launch\_analysis(data\_file):

directory = os.path.dirname(\_\_file\_\_) # we get the right path.

path\_to\_file = os.path.join(directory, "data", data\_file) # with this path, we go inside the folder `data` and get the file.

with open(path\_to\_file,"r") as f:

preview = f.readline()

print("Yeah! We managed to read the file. Here is a preview: {}".format(preview))

if \_\_name\_\_ == "\_\_main\_\_":

launch\_analysis('current\_mps.csv')

Vous pouvez à présent lire le fichier. Mais il serait plus intéressant de lancer l'analyse non pas à partir de ce fichier mais à partir de parite.py afin de pouvoir choisir, plus tard, le format de fichier adéquat.

"Transformons" le fichier csv\_analysis.py en module.

### Créer un module

hh