Documentation interne

Projet : Conception d'un outil d'expérimentation de scénarios et de génération d'emploi du temps

Arbaut Jean-Baptiste

Deschamps Kylian

Duez-Faurie Valentine

Eramil Kadir

Pilloud Aubry

Tropel Célia

Université Grenoble Alpes

Documentation interne

Conception d'un outil d'expérimentation de scénarios et de génération d'emploi du temps

Les informations d'identification du document :

Référence du document :	
Version du document :	1.3
Date du document :	18/06/2025
Auteur(s) :	Arbaut Jean-Baptiste Deschamps Kylian Duez-Faurie Valentine Eramil Kadir Pilloud Aubry Tropel Célia

Les éléments de vérification du document :

Validé par :	Aurélie Landry
Validé le :	18/06/2025
Soumis le :	13/06/2025
Type de diffusion :	Document électronique (.pdf)
Confidentialité :	/

Les éléments d'authentification :

Maître d'ouvrage:	Aurélie Landry	**Chef de projet :**	/
Date / Signature :	18/06/2025 	**Date / Signature :**	/

Sommaire

1. Introduction

Ce document constitue la documentation interne du projet de conception d'un outil d'expérimentation de scénarios et de génération automatique d'emplois du temps à destination des établissements scolaires type collèges. Il décrit l'architecture générale de l'application, les modules qui la composent, les structures de données manipulées ainsi que les interactions entre les composants de l'interface utilisateur (développée avec Dash) et le solveur d'optimisation (basé sur OR-Tools).
Ce document s'adresse à toute personne amenée à contribuer au développement, à la maintenance ou à l'évolution de l'application, qu'il s'agisse de développeurs ou d'encadrants du projet.
L'objectif est d'offrir une vision claire et structurée du fonctionnement du système, en facilitant la prise en main rapide de chaque module, la compréhension des formats de données, ainsi que le diagnostic en cas de comportement inattendu du solveur.

1.1 Objectifs et méthodes

L'application développée s'inscrit dans le cadre d'un projet de TER (Travail d'Étude et de Recherche) et vise à proposer une solution complète et accessible pour la génération automatique d'emplois du temps dans les établissements scolaires de type collège. Cette application a pour objectif de réduire significativement le temps et la complexité liés à la création manuelle des plannings hebdomadaires, tout en garantissant la prise en compte de l'ensemble des contraintes pédagogiques, matérielles et humaines propres à chaque établissement.
Le projet repose sur deux composantes principales :

- Une interface web interactive, conçue avec le framework Dash (Python), permettant à l'utilisateur de renseigner ou importer les données nécessaires : enseignants, classes, disciplines, salles, contraintes horaires, options pédagogiques… Cette interface se veut claire, et accessible à des utilisateurs sans compétences en programmation. Elle permet également de visualiser, modifier et exporter les résultats.
- Un moteur de résolution de contraintes, basé sur la bibliothèque OR-Tools développée par Google, appelé "solver", qui modélise le problème de l'emploi du temps sous forme d'un problème d'optimisation à contraintes. Ce solver prend en compte les règles définies (volumes horaires, incompatibilités, préférences, simultanéités, etc.) et produit des plannings faisables et équilibrés, en minimisant les conflits.

Le développement a suivi une approche itérative en plusieurs étapes :

1. Recueil des besoins auprès d'exemples réels d'emplois du temps scolaires et définition des types de contraintes à prendre en charge.
2. Modélisation du problème et du plan de développement.
3. Implémentation du solver avec OR-Tools, incluant la gestion des sous-groupes, des salles, des volumes horaires par matière, et des conflits potentiels. Conception en parallèle de l'interface Dash en modules fonctionnels : saisie des données, gestion des

contraintes, visualisation des résultats et export. Avec des fonctionnalités telles que le système de sauvegarde automatique.

4. Validation continue par des jeux de données réalistes et ajustements selon la faisabilité et la lisibilité des emplois du temps générés.
5. Création des tests et de la documentation associée.
6. Documentation : les commentaires du code source ainsi que la documentation interne ont été réalisés tout le long du projet. Le rapport final et les guides ont été rédigés en fin de projet.

L'ensemble du système est conçu pour fonctionner localement, sans dépendre d'un serveur distant, et pour pouvoir être adapté selon les spécificités des établissements. Le projet met l'accent sur la modularité, la fiabilité des résultats, et l'accessibilité pour des non-informaticiens. L'un des objectifs était de fournir un export compatible avec Monoposte d'Index Education, ce qui n'a pas pu être effectué.

1.2 Documents de référence

Les documents suivants ont été utilisés comme support de conception, d'analyse ou de vérification pour la réalisation du projet et ainsi qu présent document :

- Cahier des charges.
- Documentation officielle de la bibliothèque OR-Tools.
- Documentation de Dash (Plotly).
- Guide d'utilisation de l'application.
- L'ensemble des fichiers .py et .json du projet.

2. Guide de lecture

Ce document a été conçu pour répondre aux besoins de différents profils de lecteurs impliqués dans la compréhension, la modification ou l'évolution du projet. Chaque section de la documentation correspond à une couche fonctionnelle de l'application (interface, données, solveur) et peut être abordée indépendamment selon les objectifs du lecteur.

2.1 Contributeur technique ou développeur (maître d'œuvre)

Objectif : Comprendre, adapter ou étendre le code source (interface ou solveur).
À lire en priorité :

- Section 4 : description détaillée des modules Python (interface, solveur, styles, constantes).
- Section 3 : concepts clés pour appréhender l'architecture (Dash, JSON, OR-Tools).
- Section 6 : documents de référence et bibliothèques utilisées.

Le développeur doit être à l'aise avec la structure multi-modules du projet, comprendre les dépendances entre interface et moteur d'optimisation, et maîtriser les formats de données.

2.2 Utilisateur final

Objectif : Utiliser l'interface pour configurer les données, lancer les calculs et consulter les résultats.
À lire en priorité :

- Section 1.1 : objectifs et méthode générale de l'application.
- Sections 4.3 à 4.8 : fonctionnement des pages de l'interface (saisie, contraintes, résultats).
- Section 5 : glossaire des termes utilisés dans l'interface.

L'utilisateur doit savoir comment saisir les données, interagir avec les pages de contraintes, lancer une génération et comprendre les résultats affichés dans l'interface.

2.3 Maître d'ouvrage

Objectif : Suivre l'avancement, évaluer la pertinence technique du projet.
À lire en priorité :

- Section 1 : introduction générale au projet.
- Section 4.15 : fonctionnement du solveur, contraintes gérées, stratégie de résolution.
- Sections 4.3 à 4.8 : fonctionnement des pages de l'interface (saisie, contraintes, résultats).
- Section 5 : glossaire des termes utilisés dans l'interface.

Ce lecteur doit savoir comment saisir les données, interagir avec les pages de contraintes, lancer une génération et comprendre les résultats affichés dans l'interface. Il doit aussi

pouvoir évaluer la robustesse du système, sa capacité à intégrer de nouvelles contraintes, et son adéquation avec les besoins réels.

3. Concepts de base

Cette section présente les notions essentielles à connaître pour comprendre le fonctionnement de l'application et suivre efficacement les étapes décrites dans ce document. Elle permet d'aborder la suite du manuel avec une vision claire des composants et mécanismes du système.

L'interface utilisateur de l'application est développée en Dash, un framework Python conçu pour créer des interfaces web interactives. Lorsqu'on lance l'application, elle s'ouvre automatiquement dans un navigateur. L'utilisateur peut alors accéder à toutes les fonctionnalités : saisie des données, consultation des contraintes, lancement du solveur, affichage des résultats. Aucun paramétrage manuel de fichiers ou de code n'est requis pour l'usage courant.

Les données manipulées par l'application sont enregistrées dans des fichiers au format JSON (.json). Ce format standard permet de structurer l'information sous forme de paires clé / valeur facilement interprétables par le programme. Il est utilisé pour stocker les enseignants, les classes, les matières, les contraintes, ainsi que les résultats de génération. L'utilisateur n'a pas à intervenir directement sur ces fichiers : l'interface se charge de leur lecture, modification et sauvegarde de façon transparente.

Le moteur de calcul, appelé "solver", est basé sur OR-Tools, c'est une bibliothèque d'optimisation développée par Google. Elle permet d'exprimer un ensemble de contraintes obligatoires (par exemple : "Un enseignant ne peut pas avoir deux cours en même temps.") et optionnelles (comme : "Éviter les journées trop chargées."), puis de trouver automatiquement une solution cohérente. Lorsqu'un calcul est lancé, l'application génère plusieurs emplois du temps, compare les solutions obtenues et affiche la plus satisfaisante. L'utilisateur est informé de la progression via une barre de chargement et un message estimant le temps restant.

Enfin, l'ensemble du système fonctionne en local, c'est-à-dire directement sur l'ordinateur de l'utilisateur. Aucune connexion à Internet ni à une base de données externe n'est nécessaire. L'intégralité des traitements (interface, calculs, enregistrement, export) est réalisée sur l'ordinateur de l'utilisateur.

4. Description des modules du projet

Dans cette partie, nous allons voir les objectifs, la structure ou encore les dépendances de chaque module du projet.

4.1 Présentation générale des modules

4.1.1 Présentation générale des modules “Pages”

L’application est structurée autour d’une page centrale qui joue le rôle de contrôleur principal. Cette page appelle, pour chaque vue, un module spécifique chargé de gérer le contenu, la forme et les interactions propres à une page précise. Ce découpage permet d’isoler la gestion des pages suivantes dans des modules dédiés :

- La page d’accueil.
- La page de saisie des informations.
- La page des contraintes.
- La page des contraintes optionnelles.
- La page de calculs.
- La page des résultats.
- La page de présentation.

Chacune de ces pages est définie dans un fichier Python distinct, suivant une architecture commune :

- Chaque module (de page) expose une fonction nommée layout_<nom_de_page>() qui construit et retourne la structure visuelle (layout) de la page avec ses composants interactifs.
- Chaque module (de page) regroupe également les callbacks associés, qui gèrent l’interactivité et la logique métier liée à la page (réactions aux clics, saisies, mises à jour dynamiques, etc.).

Par ailleurs, chaque page peut faire appel à des données partagées, à des fonctions utilitaires (regroupées dans des fichiers dédiés), ainsi qu’à des fichiers de styles pour garantir une présentation homogène.
Ce fonctionnement unifié garantit une intégration cohérente et une organisation claire de chaque page dans l’application principale.
L’ensemble de ces modules est orchestré par le fichier app.py, qui centralise l’appel aux fonctions de layout et la gestion de la navigation entre les pages. Le routage s’effectue via les URLs associées à chaque page, permettant à l’utilisateur de circuler librement dans l’application à l’aide de la barre de navigation.

4.1.2 Présentation des modules techniques

À ces modules de page s'ajoutent plusieurs modules techniques qui assurent la configuration, le style et les traitements de fond de l'application :

- **textes.py :** centralise les textes affichés dans l'interface, notamment les messages, titres, descriptions, etc.
- **constantes.py :** contient les constantes partagées dans l'application (valeurs fixes, noms de colonnes, seuils par défaut…).
- **fonctions.py :** regroupe les fonctions utilitaires utilisées pour le traitement des données, la validation ou la transformation de l'information.
- **styles.py :** centralise les styles visuels (polices, marges, couleurs, espacements…) pour garantir une cohérence graphique.
- **app.py :** point d'entrée de l'application Dash et script principal du projet, il assure la configuration des routes et fait le lien entre l'objet app, les différents layouts.
- **main_dash.py :** instancie l'objet app pour le reste de l'application.

4.1.3 Présentation des structures des fichiers de données

L'application repose sur plusieurs fichiers de données, chacun ayant un rôle spécifique dans le fonctionnement global du système. Ces fichiers sont stockés dans le dossier data/ et utilisés à différentes étapes, de la saisie des informations jusqu'à la génération finale de l'emploi du temps.

4.1.3.1 Un fichier de données pour l'interface

C'est le fichier principal de l'application. Il stocke toutes les données saisies par l'utilisateur dans l'interface :

- Les paramètres de l'établissement (horaires, jours de cours).
- Les ressources disponibles (langues, options, classes, salles, professeurs).
- Les contraintes obligatoires et optionnelles.

Ce fichier est modifié dynamiquement à chaque interaction dans l'interface, via les callbacks Dash. Il sert également de point de référence lors de la reprise de session.

4.1.3.2 Un fichier de données pour le solver

Ce fichier est une version transformée de data_interface.json, destinée à être directement utilisée par le solveur OR-Tools. Il contient uniquement les informations structurées et nettoyées nécessaires à l'optimisation, sans éléments visuels ou intermédiaires.
Il est généré automatiquement depuis data_interface.json lors du lancement du calcul.

4.1.3.3 Un fichier de données des résultats du solver

Ce fichier contient le résultat du solveur, à savoir : les emplois du temps générés pour les classes, les enseignants, les salles et les horaires attribués à chaque cours.
Il sert à alimenter les pages d'affichage et d'exportation de l'emploi du temps dans l'interface.

4.1.3.4 Un fichier de données des statistiques de la solution

Ce fichier contient les statistiques de résolution : le taux de satisfaction des contraintes obligatoires et optionnelles et les éventuelles erreurs ou conflits détectés.
Ces statistiques sont affichées dans la page des résultats pour évaluer la qualité de la solution produite.

Nous allons maintenant voir plus spécifiquement les caractéristiques des différents modules.

4.2 Module : app.py - Gestion du routage et de la structure globale

4.2.1 Objectifs

Ce module constitue la structure centrale de l'application multi-pages et est le script principal du projet. Il définit :

- le routage entre les pages (dcc.Location).
- Assurer le lien entre l'objet app et les différents layouts.
- l'appel dynamique des fonctions layout_X de chaque module de page (ex. : layout_informations, layout_contraintes).

4.2.2 Relations

1. Modules utilisés par app.py :
    - main_dash : pour récupérer l'objet app initialisé.
    - constantes.py : pour initialiser
    - styles.py : application des styles CSS aux composants.
    - data/data_interface.json : fichier de stockage des données saisies ou importées.
    - Modules de pages : page_accueil, page_informations, page_contraintes, page_contraintes_optionnelles, page_calculs, page_resultats, page_presentation.
2. Modules utilisant app.py :
    - Module principal exécuté à la racine du projet pour lancer l'application (commande : python app.py).

4.2.3 Types et attributs

Eléments :

- server : exposé pour le déploiement avec un serveur WSGI.
- Barre de navigation : boutons Dash permettant de naviguer vers chaque page.
- app.layout : structure principale de l'application (layout global).

Callbacks :

- callback_affichage_page : callback central gérant la logique d'affichage selon l'URL.

4.2.4 Procédures externes

- app.py expose l'objet server (via app.server) pour les pages externes.

4.3 Module : page_accueil.py - Accueil de l'application

4.3.1 Objectifs

Ce module définit la page d'accueil de l'application. Il présente brièvement le projet et propose un bouton pour accéder à la suite de l'application. C'est une page principalement statique, centrée sur l'information : grâce à la courte description du projet et au guide d'utilisation présent sur la page.

4.3.2 Relations d'utilisation

1. Modules utilisés par page_accueil.py :
    - style.py : pour les styles de la page.
2. Modules utilisant page_accueil.py :
    - app.py : pour appeler la fonction layout_accueil().

4.3.3 Types

- layout_accueil() : cette fonction retourne le composant html.Div représentant toute la structure de la page.

4.3.4 Procédures externes

- app.py utilise le module page_accueil.py pour appeler la fonction layout_accueil().

4.4 Module : page_informations.py - Saisie des données de l'établissement

4.4.1 Objectifs

Ce module gère la saisie, l'affichage et la validation des informations pédagogiques de l'établissement scolaire, incluant :

- Les journées et horaires de cours.
- Les langues vivantes proposées (LV1 à LV4).
- Les options disponibles.
- Les classes, les salles et les professeurs.

La page permet à l'utilisateur :

- De saisir ou modifier ces informations via des composants interactifs.
- D'importer et exporter des données au format CSV ou Excel pour les classes, salles et professeurs.
- De valider en temps réel les données et de les sauvegarder dans le fichier data/data_interface.json.

4.4.2 Relations

1. Modules utilisés par page_informations.py :
    - fonctions.py : lecture et écriture du fichier JSON, validation de saisies, sauvegarde centralisée.
    - constantes.py : liste des niveaux (6e à 3e), unités par défaut, et autres constantes partagées.
    - styles.py : application des styles CSS aux composants.
    - data/data_interface.json : fichier de stockage des données saisies ou importées.
2. Modules utilisant page_informations.py :
    - app.py : appelle la fonction layout_informations() pour intégrer cette page dans l'interface principale.

4.4.3 Types et attributs

Definitions :

- layout_informations() : cette fonction retourne le composant html.Div représentant toute la structure de la page.

Callbacks :

- gerer_reset_ou_import() : gère l'import d'un fichier JSON ou la réinitialisation des données.
- Les fonctions dont l'intitulé contient "prefill", "charger" ou "init" gèrent l'affichage des données présentes dans le JSON au lancement de l'application (ou lors de son rafraîchissement).

- Les fonctions dont l'intitulé contient le mot "afficher" renvoient un style CSS permettant d'afficher un composant (un Div HTML), ou bien ajoutent dynamiquement un bloc HTML dans un Div.
- Les fonctions dont l'intitulé contient "sauvegarder", "MAJ" ou "save" enregistrent les données saisies en direct dans le fichier de données.
- Les fonctions dont l'intitulé contient "valider" ont pour objectif d'appeler une fonction qui vérifie la validité d'une saisie, en prenant cette saisie en entrée.
- Les fonctions dont l'intitulé contient le mot "gérer" s'occupent dynamiquement des actions possibles pour les saisies "autres" (ex : ajout ou suppression de champs personnalisés).
- Les fonctions reset_slider permettent de réinitialiser les sélecteurs de niveaux à leurs valeurs par défaut.
- Les fonctions contenant "traiter_import" et "appliquer" concernent la partie 3 de la page d'informations. Si le tableau de données est vide, les données peuvent être insérées directement ; sinon, une boîte de dialogue s'ouvre afin de demander à l'utilisateur s'il souhaite fusionner ou remplacer les données existantes. La structure du fichier importé est vérifiée avant l'ajout.
- Les fonctions contenant "telecharger" permettent à l'utilisateur de télécharger des fichiers d'exemple au format .xlsx ou .csv. Ces fichiers contiennent quelques lignes d'exemple et les colonnes attendues, ce qui permet à l'utilisateur de remplir ses données dans Excel avant de les importer dans l'outil.
- Les fonctions contenant "ajouter" ont pour but, dans la partie 3 de la page, d'ajouter une ligne vierge aux tableaux des classes, salles ou professeurs.
- La fonction naviguer() permet d'appeler des fonctions de traitement pour préparer les données à l'affichage dans la page des contraintes. Elle gère également la navigation entre les pages via les boutons en bas de page.
- La fonction desactiver_freeze() réactive les callbacks après une réinitialisation complète.

4.4.4 Procédures externes

- app.py utilise le module page_informations.py pour appeler la fonction layout_informations().
- naviguer_page_contraintes() permet de passer à la page précédente (accueil) ou suivante (contraintes).

4.5 Module : page_contraintes.py - Saisie de contraintes

4.5.1 Objectifs

Ce module gère la saisie, l'affichage et la validation des contraintes de l'établissement scolaire.

Ce module permet de définir les contraintes horaires obligatoires associées :

- Aux professeurs (ex : indisponibilités).
- Aux groupes/classes (ex. indisponibilités ou créneaux réservés).
- Aux salles (ex. indisponibilités ou utilisations prioritaires).

La page se compose de plusieurs sections, qui utilisent toutes la même structure :

- Un menu de sélection (prof, classe ou salle).
- Un tableau emploi du temps interactif.
- Un panneau d'ajout de contraintes (jour, heure, type de contrainte).
- Un tableau récapitulatif des contraintes ajoutées ou supprimées.

Elle contient aussi une section 3.4 dédiée aux contraintes de planning, qui permet de définir :

- Un nombre d'heures maximal consécutives pour une matière.
- Des cours à obligation ou interdiction à certains créneaux.
- Des enchaînements autorisés ou interdits entre deux cours avec un minimum requis.

4.5.2 Relations

- Modules utilisés par page_contraintes.py :
    - fonctions.py : pour l'écriture centralisée dans le JSON.
    - constantes.py : pour récupérer les niveaux, matières, etc.
    - styles.py : pour styliser les tableaux et composants.
    - data/data_interface.json : le fichier JSON est utilisé pour lire les horaires, les entités et sauvegarder les contraintes.
- Modules utilisant page_contraintes.py :
    - app.py : appelle layout_contraintes() pour inclure la page dans la navigation principale.

4.5.3 Types et attributs

<u>Définitions :</u>

- layout_contraintes() : cette fonction retourne le composant html.Div représentant toute la structure de la page, en générant dynamiquement les sections pour chaque type d'entité (prof, groupe, salle) et la partie « contraintes planning ».

<u>Callbacks Majeurs:</u>

- make_callback_contraintes() : gère toutes les actions sur les tableaux emploi du temps interactifs pour chaque entité (professeur, groupe, salle) : coloration des cellules, gestion des clics, ouverture du panneau latéral d'ajout de contrainte, suppression de contraintes, et synchronisation du store des contraintes.
  Cette fonction garantit la cohérence entre l'interface et les données sauvegardées.
- make_callback_update_table_styles_on_selection() : recharge dynamiquement les styles des cellules dans chaque tableau emploi du temps dès qu'une nouvelle entité est sélectionnée (ex : changement de prof). Les cellules sont colorées en fonction des contraintes déjà posées, pour une lecture rapide de la disponibilité.

- make_disable_options_callback() : désactive automatiquement, dans le menu déroulant de sélection d'entité (prof, groupe, salle), les options incompatibles avec la sélection courante. Une entité est grisée si ses contraintes ne correspondent pas à celles de l'entité active, évitant ainsi des incohérences lors de la saisie.
- make_callback_contrainte_3_4() : fabrique dynamiquement les callbacks génériques nécessaires à la section 3.4 (contraintes de planning). Cette fonction permet d'appliquer et de réinitialiser les contraintes selon la sous-section concernée :
    - Nombre d'heures maximum (stockée dans store-planning-data).
    - Cours-planning (stockée dans store-cours-planning-data).
    - Enchaînement de cours (stockée dans store-enchainement-data).

Callbacks mineurs :

- apply_logic_planning(), apply_logic_cours(), apply_logic_enchainement() : servent respectivement à appliquer une contrainte.
- reset_logic_cours(), reset_enchainement(), reset_logic_planning() : servent respectivement à supprimer des contraintes.
- init_table_recap_enchainement(), init_table_recap_planning(), init_table_recap_cours() : initialisent les tableaux récapitulatifs des différentes sous-sections de contraintes à partir des données du store, à chaque chargement ou modification.
- maj_options_doubles(), update_enchainement_options(), update_minimum_dropdown() : actualisent dynamiquement les menus déroulants selon les sélections en cours, pour garantir la cohérence des choix possibles .

Autres fonctions :

Il existe 16 autres fonctions utilisées dans ce module, qui ne jouent pas un rôle central : elles ne sont pas associées directement à un callback, mais sont appelées en interne au sein des callbacks principaux. Elles ne sont pas séparées du reste du code, car leur logique est souvent très spécifique à cette page (elles portent parfois le même nom) et ne sont pas réutilisées ailleurs dans l'application, ce qui justifie de ne pas les intégrer dans le fichier fonctions.py.

4.5.4 Procédures externes

- app.py utilise le module page_contraintes.py pour appeler la fonction layout_informations().
- naviguer_page_contraintes() permet de passer à la page précédente (informations) ou suivante (contraintes optionnelles).

4.6 Module : page_contraintes_optionnelles.py - Saisie de contraintes optionnelles et ordonnancement

4.6.1 Objectifs

Ce module permet à l'utilisateur de définir les contraintes non obligatoires (dites « optionnelles ») dans le cadre de la génération de l'emploi du temps. Ces contraintes ne bloquent pas la génération d'un emploi du temps mais influencent la solution.
Sur cette page on peut saisir les contraintes de :

- Poids du matériel scolaire par matière et par niveau (avec une tolérance de 5% qui est appliquée par le générateur).
- Réfectoire et permanence (créneaux interdits selon les niveaux).

Une troisième section permet de classer par ordre d'importances ses contraintes optionnelles. Si la contraintes n'est pas saisie elle n'est pas présente dans la liste, au maximum il peut y avoir 5 éléments :

- Contraintes non obligatoires de salles : les indisponibilités partielles.
- Contraintes non obligatoires d'enseignants : les indisponibilités partielles.
- Contraintes non obligatoires des classes : les indisponibilité partielles.
- Contraintes de poids du matériel scolaire.
- Contraintes de réfectoire et permanence.

4.6.2 Relations

1. Modules utilisés par page_contraintes_optionnelles.py :
    - fonctions.py : pour sauvegarder les données saisies dans le fichier JSON.
    - styles.py : pour l'application de styles visuels sur les composants.
    - constantes.py : pour récupérer les niveaux, les matières, etc.
    - data/data_interface.json : toutes les contraintes optionnelles sont stockées dans la section contraintes_optionnelles.
    - textes.py : pour récuper du texte et l'importer sur la page.
2. Modules utilisant page_contraintes_optionnelles.py :
    - app.py : appelle la fonction layout_contraintes_optionnelles() pour inclure cette page dans l'application.

4.6.3 Types et attributs

Définitions :

- layout_informations() : cette fonction retourne le composant html.Div représentant toute la structure de la page.

Callbacks :

- Les fonctions dont l'intitulé contient "make_section" construisent chaque volet de saisie (sous forme d'accordéon / de bloc) pour chaque type de contrainte optionnelle. Chaque section comprend :

    - Des menus déroulants dynamiques,
    - Des zones de saisie spécifiques (ex : pourcentage, tolérance, jours),
    - Un tableau récapitulatif (DataTable) des contraintes enregistrées,
    - Des boutons d'application et de suppression.
- Les fonctions dont l'intitulé se compose de "prefill" ou "charger" ou init gèrent l'affichage des données qui sont dans le JSON au lancement de l'application (ou au rafraîchissement).
- Les fonctions dont l'intitulé se compose du mot "afficher" renvoie un style css permettant d'afficher un composant (un Div HTML) ou ajoute un bloc html dans un Div HTML.
- Les fonctions dont l'intitulé se compose de "sauvegarder" ou "MAJ" ou "save" enregistrent les données saisies en direct dans le fichier de données.
- La fonction "modifier_ordre" permet de déplacer une contrainte vers le haut ou le bas dans le tableau d'ordre.
- La fonction "reset_si_desactivation_poids" réinitialise les données liées au poids du matériel scolaire si l'utilisateur désactive cette contrainte.
- La fonction de navigation permettant d'accéder à la page suivante ou précédente suivant le bouton cliqué.

4.6.4 Procédures externes

- app.py utilise le module page_contraintes_optionnelles.py pour appeller la fonction layout_contraintes_optionnelles().
- La fonction de navigation permet d'accéder à la page précédente contraintes.py ou à la page suivante calculs.py.

4.7 Module : page_calculs.py - Lancement du solver

4.7.1 Objectifs

Ce module fournit l'interface permettant de lancer la génération automatique des emplois du temps à partir des données saisies dans l'interface. L'utilisateur choisit le nombre de solutions à comparer (nombre de runs) avant d'initier le calcul.
Le module gère l'affichage dynamique de la progression du calcul, l'animation de chargement, l'estimation du temps restant.

4.7.2 Relations

Modules utilisés par page_calculs.py :

- styles.py : pour styliser les tableaux et composants.
- textes.py : pour appeler du texte sur la page.

- fonctions.py et solver.py : pour charger les données et exécuter le solveur.
- data/data_interface.json : le fichier JSON est utilisé pour obtenir les informations de l'établissement.

Modules utilisant page_calculs.py :

- app.py : pour intégrer la page via la fonction layout_calculs().

4.7.3 Types et attributs

Définitions :

- layout_calculs() : cette fonction retourne le composant html.Div représentant toute la structure de la page.
- lancer_solver_thread() : Lance le solveur d'emploi du temps dans un thread séparé pour éviter des conflits avec dash.
-

Callbacks :

- demarrer_calcul() : Démarre le processus de calcul d'emploi du temps. Callback déclenché lors du clic sur le bouton "Lancer le calcul". Initialise l'interface de suivi en affichant le bloc de chargement, activant l'intervalle de mise à jour, et préparant les variables de suivi.
- aller_aux_resultats() : Redirige vers la page des résultats. Callback déclenché lors du clic sur le bouton "Voir les résultats" après la finalisation du calcul.
- progression() : Met à jour l'interface de progression du calcul en temps réel. Callback principal pour le suivi de l'avancement du calcul d'emploi du temps. Gère le démarrage du solver, la mise à jour de tous les éléments d'interface (barre de progression, temps estimé, messages de statut, taux de réussite) et la finalisation avec récupération des résultats.

4.7.4 Procédures externes

- app.py utilise le module page_calculs.py pour appeller la fonction layout_calculs().
- Les callbacks sont destinés à fonctionner avec executer_runs_avec_suivi(...) dans solver.py : exécution du solveur OR-Tools avec les données préparées.
- La fonction de navigation permet d'accéder à la page précédente contraintes_optionnelles.py ou à la page suivante page_resultats.py.

4.8 Module : page_resultats.py - visualisation des résultats

4.8.1 Objectifs

Ce module gère l'affichage, l'analyse, l'édition, le déplacement et l'exportation des emplois du temps de l'établissement scolaire, incluant :

- Statistiques de complétion : Affichage des indicateurs clés (volume horaire, contraintes obligatoires/optionnelles, taux global, violations) via des cartes colorées (vert si conforme, rouge sinon) et un tableau détaillé des contraintes avec une section repliable pour les violations.
- Affichage : Visualisation des emplois du temps par classe, professeur ou salle, avec distinction entre les semaines A et B, dans un tableau interactif à cellules cliquables.
- Édition : Modification des créneaux horaires (matière, professeur, salle, classe) via un panneau latéral, avec options pour ajouter ou supprimer des cours.
- Déplacement : Déplacement ou échange de cours entre créneaux horaires avec options dynamiques.
- Exportation : Génération de fichiers PDF pour l'emploi du temps affiché ou toutes les entités (classes, professeurs, salles), regroupés dans un fichier ZIP.

La page permet à l'utilisateur :

- Analyser les performances de l'emploi du temps via les statistiques (taux de respect des contraintes, détails des violations).
- Interagir avec les emplois du temps via des sélecteurs (vue, entité), un tableau split (Semaine A/B) et un panneau latéral.
- Modifier les informations des cours (matière, professeur, salle, classe).
- Exporter les emplois du temps en PDF (entité affichée ou toutes les entités d'un type).
- Sauvegarder les modifications dans le fichier data/emploi_du_temps_global.json.

4.8.2 Relations

Modules utilisés par page_resultat.py :

- app.py : fournit l'instance de l'application Dash (app) pour définir les callbacks et le layout.
- styles.py : contient les styles CSS centralisés pour les composants de l'interface.
- fonctions.py : Fournit des fonctions utilitaires pour charger, sauvegarder et manipuler les données des emplois du temps et des statistiques.
- data/emploi_du_temps_global.json : fichier de stockage des données des emplois du temps.
- data/tous_rapports_contraintes.json : fichier de stockage des statistiques des emplois du temps.

Modules utilisant page_resultat.py :

- app.py : appelle la fonction layout_resultats() pour intégrer cette page dans l'interface principale.

4.8.3 Types et attributs

Définitions :

- layout_resultats() : retourne le composant html.Div représentant la structure complète de la page, incluant les statistiques, les sélecteurs, le tableau des emplois du temps, le panneau latéral, et les composants de stockage et d'action.

Callbacks :

- maj_liste_entites() : Met à jour les options du menu déroulant des entités (classes, professeurs, salles) en fonction du mode de vue sélectionné, et définit l'entité par défaut.
- update_table_split() : génère le tableau des emplois du temps en fonction de la vue, de l'entité, du mode (affichage, édition, déplacement, exportation) et des sélections en cours.
- global_callback() : gère les interactions principales, telles que la sélection des créneaux pour l'édition ou le déplacement, l'enregistrement des modifications, la suppression des cours, l'annulation des actions, et l'exportation des PDF.
- update_sidepanel() : met à jour le contenu du panneau latéral en fonction du mode (affichage, édition, déplacement, exportation) et des sélections (créneau source, destination, ou options de transformation).
- hide_sidepanel() : contrôle la visibilité du panneau latéral en fonction du mode (caché en mode affichage, visible sinon).
- reset_selection_on_mode_change() : réinitialise les sélections (source, destination, action en attente) lors du changement de mode.
- apply_move() : applique les transformations sélectionnées (échange, fusion, déplacement) entre deux créneaux horaires.
- single_checkbox() : garantit qu'une seule option de transformation est sélectionnée dans le panneau de déplacement.
- sync_save_edit_visible(), sync_delete_visible(), sync_cancel_edit_visible(), sync_delete_a_visible(), sync_delete_b_visible(), sync_export_panel_visible() : Synchronisent les clics des boutons visibles avec leurs stores respectifs.
- trigger_save_edit(), trigger_delete(), trigger_cancel_edit(), trigger_delete_a(), trigger_delete_b(), trigger_export_panel() : Déclenchent les actions des boutons cachés à partir des stores.
- sync_matiere_a(), sync_prof_a(), sync_salle_a(), sync_classe_a(), sync_matiere_b(), sync_prof_b(), sync_salle_b(), sync_classe_b() : Synchronisent les valeurs des champs d'entrée visibles (Semaine A/B) avec les champs cachés.
- sync_export_options() : Synchronise les options d'exportation entre les panneaux visibles et cachés.

4.8.4 Procédures externes

- app.py utilise le module page_contraintes.py pour appeller la fonction layout_contraintes_optionnelles().
- naviguer_page_contraintes() permet de passer à la page précédente (contraintes_optionnelles) ou suivante (resultats).

4.9 Module : styles.py – Styles CSS centralisés de l'application

4.9.1 Objectifs

Le module styles.py centralise l'ensemble des styles CSS utilisés dans l'interface Dash. Il permet une gestion cohérente et maintenable de l'apparence visuelle de l'application. En définissant tous les styles dans un seul fichier, il facilite la réutilisation des composants et la modification rapide de l'apparence globale.

Ce fichier contient des dictionnaires Python de propriétés CSS pour des composants Dash/HTML, parmi lesquels :

- Les boutons (importation, génération, navigation, réinitialisation, etc.).
- Les tableaux (cellules, en-têtes, conteneurs, ordonnancement).
- Les textes (titres, messages, explications).
- Les composants interactifs (dropdowns, sliders, inputs).
- La structure des pages (marges, alignements, sections).

4.9.2 Relations

1. Modules utilisés par styles.py :
    - Aucun module n'est importé ici. Ce fichier contient exclusivement des définitions de dictionnaires de styles.
2. Modules utilisant styles.py :
    - page_accueil.py : applique les styles pour les titres, images, le texte.
    - page_informations.py : applique les styles pour les blocs de saisie, les tableaux, les boutons, les messages et les zones d'importation.
    - page_contraintes.py : applique les styles pour les tableaux emploi du temps, les boutons de disponibilité, les panneaux latéraux, et les récapitulatifs de contraintes.
    - page_contraintes_optionnelles.py : utilise les styles pour les dropdowns, les zones d'ajout de contraintes, les tableaux de recap, et les composants de navigation.
    - page_calculs.py : applique les styles pour les boutons de lancement, les textes explicatifs, les barres de progression et les indicateurs visuels de chargement.

- page_resultats.py : utilise les styles pour l'affichage des résultats, les résumés statistiques, les taux de satisfaction, les tableaux détaillés et les éléments de navigation.

4.9.3 Types et attributs

Ce module contient uniquement des variables globales, définies comme des dictionnaires Python (ex. : style_btn_suivant, style_table, style_input, etc.).

4.9.4 Procédures externes

Le module styles.py est importé dans toutes les pages principales de l'application (accueil, informations, contraintes, contraintes optionnelles, résultats, etc.) afin d'assurer une cohérence visuelle dans toute l'interface utilisateur.

4.10 Module : constantes.py - Constantes globales de l'application

4.10.1 Objectifs

Ce module regroupe toutes les constantes générales nécessaires au bon fonctionnement de l'application Dash de génération d'emploi du temps. Il centralise les données de base communes à plusieurs modules, afin de garantir une cohérence dans l'interface et les traitements.

Les objectifs principaux sont :

- Définir les listes de niveaux scolaires utilisés dans l'application (6e, 5e, 4e, 3e).
- Lister les jours de la semaine et leur ordre logique.
- Définir les catégories d'options disponibles dans les établissements (langues régionales, artistiques, sportives, etc.).
- Fournir des exemples de données par défaut pour les classes, les professeurs et les salles (utilisés pour pré-remplir des tableaux ou générer des fichiers d'exemple).
- Mettre à disposition des DataFrame utilisables dans les fonctions d'import/export (CSV, Excel).
- Définir les styles de contraintes utilisés dans la page contraintes.

4.10.2 Relations

Modules utilisés par constantes.py :

- fonctions.py : pour appeler la fonction initialiser_fichier_si_absent() en fin de script.

Modules utilisant constantes.py :

- app.py : pour appeler la fonction INITIALISATION_FICHIER sous la forme du constante.
- page_informations.py : pour utiliser les niveaux, données par défaut et constantes partagées.
- page_contraintes.py : pour les styles de contraintes (Disponible, Indisponibilité partielle, Indisponibilité totale).
- page_contraintes_optionnelles.py : pour l'accès à certaines valeurs de configuration globales (niveaux, horaires, options).

4.10.3 Types et attributs

- NIVEAUX : liste des niveaux proposés dans le collège (["6e", "5e", "4e", "3e"])
- JOURS_SEMAINE : liste des jours présentés avec des labels utilisables dans des composants Dash (Dropdown, Table).
- ORDRE_JOURS : ordre logique des jours, utilisé pour le tri et l'affichage.
- HORAIRES_LABELS : liste des libellés servant à la configuration des horaires de cours.
- CATEGORIES_OPTIONS : dictionnaire regroupant les catégories d'options (langues anciennes, régionales, artistiques, sportives, technologiques), sous forme de paires label/value prêtes à être injectées dans des composants Dash.
- OUI_NON : liste simple oui/non utilisée dans les sélecteurs binaires.
- CLASSES_DEFAUT, PROFS_DEFAUT, SALLES_DEFAUT : listes de dictionnaires servant d'exemples pour les tableaux de saisie ou d'import.
- DF_CLASSES, DF_PROFS, DF_SALLES : pandas.DataFrame contenant les mêmes données que ci-dessus mais au format exploitable directement dans les fonctions send_data_frame().
- CONSTRAINTS : dictionnaire contenant les styles de fond et de texte associés aux contraintes dans la page de contraintes.
- INITIALISATION_FICHIER : résultat de l'appel à initialiser_fichier_si_absent() garantissant la présence du fichier JSON de configuration.

4.10.4 Procédures externes

Aucune fonction n'est exposée directement par ce module à l'exception du déclenchement de initialiser_fichier_si_absent(). Ce module sert de référentiel de constantes importable tel quel dans tous les autres fichiers de l'application.

4.11 Module : textes.py - Textes d'accompagnement de l'interface

4.11.1 Objectifs

Ce module centralise l'ensemble des textes statiques utilisés dans l'interface de l'application. Il a pour objectif :

- D'assurer une cohérence éditoriale entre les différentes pages (accueil, informations, contraintes, calculs, résultats).
- De faciliter la modification des textes affichés dans les éléments Dash (html.P, html.H5, etc.).
- De séparer le contenu (textes) de la structure pour alléger le code des layouts.

Les textes peuvent correspondre à :

- Des consignes de saisie ou d'usage.
- Des paragraphes introductifs ou explicatifs.
- Des descriptions accompagnant les sections ou tableaux.

4.11.2 Relations

1. Modules utilisés par textes.py :
    - Ce module ne dépend d'aucun autre.
2. Modules utilisant textes.py :
    - page_informations.py : explications sur les horaires, langues, options, ressources, etc.
    - page_contraintes.py : instructions sur la saisie des contraintes et leur logique.
    - page_contraintes_optionnelles.py : textes explicatifs pour le poids du matériel, le réfectoire et les permanences, l'ordre des contraintes.
    - page_calculs.py : paragraphes expliquant le fonctionnement du solveur et la génération de plusieurs solutions.
    - page_resultats.py : description de l'affichage des résultats et des statistiques.

4.11.3 Types et attributs

Le module définit uniquement des chaînes de caractères (type str) associées à des variables, par exemple : texte_entete relatif au texte de présentation de la page des informations.

4.11.4 Procédures externes

Les variables de texte sont importées dans les modules des pages pour un affichage dans l'interface.

4.12 Module : fonctions.py - Fonctions de l'interface et de transformation des données

4.12.1 Objectifs

Ce module regroupe toutes les fonctions utilitaires nécessaires au fonctionnement global de l'interface Dash de génération d'emplois du temps. Il centralise la lecture, l'écriture et la mise à jour du fichier data_interface.json, gère la transformation de ces données en une configuration exploitable par le solveur OR-Tools, et propose divers outils pour l'affichage, la validation, et le traitement des contraintes.

4.12.2 Relations

1. Modules utilisés par fonctions.py :
    - styles.py : pour appliquer des styles aux éléments HTML.
    - textes.py : pour insérer du texte dans les composants de saisie.
    - main_dash.py : pour accéder à l'instance globale de l'application Dash.
2. Modules utilisant fonctions.py :
    - Toutes les pages de l'interface utilisent ce module pour la gestion des données : page_informations.py, page_contraintes.py, page_contraintes_optionnelles.py, page_calculs.py, page_resultats.py en dépendent fortement.

4.12.3 Types et attributs

Définitions :

- generer_squelette_json_interface() : crée la structure initiale complète du fichier data_interface.json.
- initialiser_fichier_si_absent() : vérifie et crée ce fichier au besoin.
- charger_donnees_interface() / sauvegarder_donnees_interface() : lecture et écriture complètes du fichier.
- mettre_a_jour_section_interface() : met à jour une section précise du fichier sans écraser le reste.
- merge_dicts() : fusion récursive de dictionnaires imbriqués.
- formater_options() : ajoute du style personnalisé aux options d'un menu déroulant.
- calculer_horaires_affichage() : construit une liste de créneaux horaires formatés.
- generer_professeurs_affichage(), generer_salles_affichage() : génèrent les noms visibles à l'écran.
- generer_volume_horaire() : assemble le volume horaire par matière, classe, langue ou option.
- transformer_interface_vers_config() : génère le fichier config.json lisible par le solveur.

- charger_data_interface_et_modeles() : prépare toutes les variables et objets à transmettre au solveur.
- charger_contraintes_interface(), save_constraints_interface() : lecture et écriture des indisponibilités et filtre en fonction de la présence ou non dans la partie affichage du json.
- build_styles() : applique visuellement les contraintes sur les tableaux.
- construire_recapitulatif() : transforme les contraintes horaires en tableau lisible.
- generate_lang_selection() : génère dynamiquement un champ de sélection de langues.
- valider_format_horaire() : assure que les horaires sont valides (HH:MM).
- maj_donnees_programme_par_niveau() : met à jour les volumes horaires et la présence de l'EMC.
- charger_statistiques_contraintes() : extrait et résume les statistiques de respect des contraintes.
- creer_table_edt() : initialise un DataFrame représentant un emploi du temps vide.
- make_section_contraintes : construit un composant AccordionItem pour la saisie interactive des contraintes (professeurs, groupes, salles) avec sélection, boutons d'état, tableau EDT et panneau récapitulatif.
- maj_contrainte_nb_heures : ajoute ou met à jour une contrainte sur le nombre d'heures consécutives pour une ou plusieurs classes/matières, puis sauvegarde la configuration.
- charger_donnees_edt : charge le fichier principal d'emploi du temps (emploi_du_temps_global.json) et en extrait la liste de tous les horaires utilisés.
- sauvegarder_edt_global : enregistre l'emploi du temps global dans le fichier principal au format JSON.
- get_entites : retourne la liste de toutes les entités (classes, professeurs, salles) présentes pour une vue donnée, toutes semaines confondues.
- fmt : formate le contenu d'une cellule de l'emploi du temps pour l'affichage HTML selon la vue (classe, prof, salle).
- fmt_cours_simple : formate simplement le nom de la matière d'un créneau, ou 'vide' si la cellule est vide.
- is_cours_plein : indique si un créneau est un cours « plein », c'est-à-dire identique en semaine A et B.
- has_content : indique si un créneau contient au moins un cours en semaine A ou B.
- get_cours_type : détermine le type de cours pour un créneau : 'plein', 'demi' (différent entre semaines), ou 'vide'.
- apply_transformation : applique une transformation (déplacement, échange, fusion, etc.) sur deux créneaux d'un emploi du temps et retourne la nouvelle structure résultante.
- fmt_pdf : formate le contenu d'une cellule pour l'export PDF, avec un style adapté selon la vue.
- make_pdf : génère un PDF stylisé à partir d'un tableau d'emploi du temps (inclut titres, couleurs, type de cellule…).
- safe_join : formate une valeur (liste ou chaîne) pour l'affichage ou la saisie (retourne une chaîne propre).

- generate_cell_content : génère le composant Dash affichant le contenu visuel d'une cellule EDT (cours plein, demi, état sélectionné/source…).
- build_split_table : construit le tableau principal d'emploi du temps pour une entité (classe, prof, salle), avec stylisation dynamique selon le type de chaque cellule.
- get_configuration_options : retourne la liste des transformations possibles (swap, move, overwrite, etc.) entre deux créneaux donnés selon leur type.
- build_option_preview_dash : génère un mini-tableau de prévisualisation Dash montrant l'effet d'une transformation appliquée.
- get_table_matrix_pdf : génère la matrice de données et types de cellules pour préparer l'export PDF.
- build_preview_table : construit un mini-tableau Dash compact pour une prévisualisation rapide de l'EDT d'une entité.
- build_edit_panel : génère le panneau d'édition interactif pour un créneau horaire (avec champs adaptés à la vue : classe, prof, salle…).
- harmoniser_intitule : Met en forme l'intitulé d'une matière.
- normaliser_matiere_groupe : Normalise les intitulés de la colonne 'MatiereGroupe' pour qu'ils soient uniquement l'une des valeurs autorisées : Option, LV1, LV2, LV3, LV4.

4.12.4 Procédures externes

Toutes les pages de l'interface (informations, contraintes, contraintes optionnelles, résultats) utilisent ce module pour charger les données, les valider, les enregistrer ou les transformer.

4.13 Module : main_dash.py

4.13.1 Objectifs

Ce module initialise l'application Dash avec les feuilles de style externes (thèmes Bootstrap et icônes).
Il crée également l'objet app, qui sert de point d'ancrage central à toute l'application et sur lequel viennent se greffer tous les layouts et callbacks. Cette séparation permet d'éviter les références circulaires entre le fichier appy.py (initialisation de app)et les modules des autres pages se reposant sur la présence de l'objet app.

4.13.2 Relations

1. Modules utilisant main_dash.py :
    - app.py : importe l'objet app pour le routage et le serveur.

4.13.3 Types et attributs

- external_stylesheets : liste des feuilles de style CSS externes à charger.
- app : objet principal Dash, accessible globalement pour l'ensemble des pages.

4.13.4 Procédures externes

- app.py utilise le module main_dash.py pour utiliser l'objet app.

4.15 Module : solver.py

Le module solver.py se décompose en plusieurs sous parties distinctes. Nous allons donc voir les objectifs, relations, types / attributs et procédures externes de chaque sous-partie.

4.14.1 Initialisation des données

4.14.1.1 Objectif

Préparer l'ensemble des structures de données nécessaires aux phases suivantes :

- Transformation du fichier provenant de l'interface en un fichier "config.json" permettant de récupérer les paramètres.
- Lecture des paramètres (volumes horaires, contraintes pédagogiques).
- Organisation des entités (semaines, jours, créneaux, matières, professeurs, salles, sous-groupes).
- Construction de mappings et listes facilitant la modélisation.

4.14.1.2 Modules

- json : lecture du fichier de configuration.
- collections.defaultdict : création de dictionnaires à valeurs par défaut.

4.14.1.3 Relations

La fonction **charger_relations** lit le fichier de configuration config.json (généré par transformer_interface_vers_config) et en extrait toutes les données utiles sous forme de variables Python, qu'elle regroupe ensuite dans un unique dictionnaire de sortie.

1. Lecture du fichier
    - Ouvre et parse le JSON config.json pour obtenir le dict brut config.

2. Extraction des listes principales
    - SEMAINES : récupère la liste des semaines (typiquement ["Semaine A", "Semaine B"]).
    - JOURS : liste des jours de la semaine (config["jours"]).
    - HEURES : liste des créneaux horaires normalisés (config["heures"]).
    - MATIÈRES : liste des matières (config["matieres"]).
3. Extraction des dictionnaires de ressources
    - PROFESSEURS : structure par matière et par niveau (config["professeurs"]).
    - CAPACITÉS_SALLES : capacité de chaque salle (config["capacites_salles"]).
4. Constitution des contraintes et préférences
    - INDISPONIBILITÉS
      • regroupe indisponibilites_profs et indisponibilites_salles sous une même clé.
    - PRÉFÉRENCES
      • regroupe preferences_profs et preferences_salle_professeur.
5. Structures hiérarchiques de classes
    - CLASSES_BASE : liste des classes « pures » sans sous-groupes (config["classes_base"]).
    - SOUS_GROUPES : pour chaque entrée de config["sous_groupes_config"], concatène le nom de chaque classe de base avec son suffixe pour former le nom du sous-groupe.
    - CLASSES : union de CLASSES_BASE et SOUS_GROUPES.
6. Assemblage final
    - Retourne un dictionnaire unique regroupant :
      SEMAINES, JOURS, HEURES, MATIÈRES, PROFESSEURS, CAPACITÉS_SALLES,
      INDISPONIBILITÉS, PRÉFÉRENCES, CLASSES_BASE, SOUS_GROUPES, CLASSES.

4.14.1.4 Types et attributs des variables

Objet retourné	Type	Description
SEMAINES	Liste	Contient « A » et « B » pour l'alternance entre les semaines
JOURS	Liste	Contient les jours où les élèves ont cours
HEURES	Liste	Contient les créneaux horaire de cours
MATIERES	Liste	Contient les matières pour tous les niveaux
PROFESEURS	Dictionnaire	Contient les caractéristiques de chaque enseignants (nom, matière et niveau enseigné)
VOLUME_HORAIRE	Dictionnaire	Contient le volume horaire de chaque matière pour chaque niveau
INDISPONIBILITES_PROFS/SALLES	Dictionnaire	Contient les créneaux pour lesquels un enseignant ou une salle ne peut pas être affecté
PREFERENCES_PROFS/SALLES	Dictionnaire	Contient les créneaux préférentielles des professeurs et les salles dans lesquels ils préfèrent avoir cours
CAPACITES_CLASSES/SALLES	Dictionnaire	Contient le nombre d'élève dans une classe et le nombre de place disponible dans une salle
NIVEAUX	Liste	Contient les niveaux enseignés
AFFECTATION_MATIERE_SALLE	Dictionnaire	Contient les matière affectées aux salles spéciales
CLASSES_BASE	Liste	Contient le nom des classes (ex : 6e1,6e2…)
SALLES_GENERALES	Liste	Contient le nom des salles
JOURS_SANS_APRES_MIDI	Liste	Contient le nom du jour où il n'y a pas cours l'après midi
MATIERE_SOUSGROUPES	Dictionnaire	Contient les différents groupes de matières
MATIERE_SCIENTIFIQUE/LANGUES/ARTISTIQUES	Liste	Contient le nom des matières pour chaque sous-groupe
SOUS_GROUPES_CONFIG	Dictionnaire	Contient les éléments permettant la configuration des sous-groupes
SOUS_GROUPES/SOUS_GROUPES_SUFFIXE/SOUS_GROUPES_TYPES	Dictionnaire	Contient le nom des sous-groupes ainsi que leurs suffixes permettant de les différencient des matières classes, le type sert à différencier les langues vivantes et les options
MAX_HEURE_PAR_ETENDUE	Dictionnaire	Contient les paramètres de la contrainte c'est-à-dire : le niveau, le nom de la matière, le nombre d'heure et l'étendue (journée ou demi-journée)
CLASSES	Liste	Contient l'aggrégation entre CLASSES_BASE et SOUS_GROUPES

4.14.1.5 Procédures externes

- On utilise open(path, encoding="utf-8") et json.load(file) pour lire et utiliser le fichier json.

4.14.2 Construction du modèle

4.14.2.1 Objectif

L'objectif de cette partie est d'assembler un modèle complet comprenant :

1. **Variables :**
    - Affectation matière par créneau en respectant le volume horaire.
    - Attribution de salle par cours.
    - Assignation de professeurs aux cours selon leurs préférences.
2. **Contraintes :**
    - Non-chevauchement : aucune salle ni enseignant ne peut être doublement affecté.
    - Volumes horaires : respect des heures allouées par matière/niveau.
    - Pédagogie : jours sans après-midi, alternance A/B, contraintes de suite de matières, respect des indisponibilités des professeurs et des salles.
    - Ressources : cantine, permanence, capacités de salles.
3. **Objectif :**
    - Minimisation des pénalités liées aux violations de contraintes optionnelles.
    - Application des contraintes obligatoires.

4.14.2.2 Modules

- ortools.sat.python.cp_model : CpModel, CpSolver, types de variables, méthodes de contrainte.
- random : sélection aléatoire de seeds pour diversification des runs.
- copy, time, tabulate : duplication de structures pour les semaines A et B, mesure de temps pour l'estimation.

4.14.2.3 Relations

- Repose sur les variables globales issues de init_donnees.
- Produit en sortie :
    - model (CpModel).
    - emploi_du_temps (Dict[(semaine, classe, jour, heure) → IntVar]).
    - emploi_du_temps_salles, emploi_du_temps_profs (mêmes clés → IntVar/BoolVar).

4.14.2.4 Types et attributs des variables

Voici la description des fonctions utilisées dans la fonction créer_modele :

1. **matExcluSuite**(pMatiere1, pMatiere2, pContrainte, pClasse) :
    a. **Objectifs :**
    Garantir qu'une occurrence de la matière pMatiere2 ne suive **jamais** immédiatement une occurrence de pMatiere1 dans un même créneau horaire, selon la force de contrainte.
    b. **Paramètres :**
        - pMatiere1 (str) : matière interdite en premier.
        - pMatiere2 (str) : matière interdite en second (au créneau suivant).
        - pContrainte (str) : "forte" (implication stricte) ou "faible" (au moins une des deux positions doit changer).
        - pClasses (list ou str) : liste de classes ciblées (ou préfixe pour filtrer).
    c. **Logique interne :**
        1. Récupère l'index CP-SAT (matdex_1, matdex_2) pour chacun des deux codes matières.
        2. Filtre les classes à traiter (soit CLASSES_BASE, soit celles dont le nom contient pClasses).
        3. Pour chaque créneau (classe, jour, heure) **sauf le dernier de la journée**, crée deux BoolVar :
            - excluSuite_a = 1 si matiere1 est affectée ici.
            - excluSuite_b = 1 si matiere2 est affectée au créneau suivant.
        4. Lie ces BoolVar aux IntVar matières via des OnlyEnforceIf.
        5. Selon pContrainte :
            - **forte** : AddImplication(excluSuite_a → ¬excluSuite_b)
            - **faible** : AddBoolOr([¬excluSuite_a, ¬excluSuite_b])
    d. **Effets :**
    Empêche systématiquement (pContrainte = forte) ou fortement (pContrainte = faible) la succession interdite de deux matières dans l'emploi du temps.

2. **matIncluSuite**(pMatiere1, pMatiere2, pContrainte, pNBFois=1, pClasses=CLASSES_BASE) :
    e. **Objectifs :**
    Imposer que la matière pMatiere1 soit **suivie** de pMatiere2 un certain nombre de fois (pNBFois) dans la semaine, avec une force de contrainte ajustable.
    f. **Paramètres :**
        - pMatiere1, pMatiere2 (str) : matières à enchaîner.
        - pContrainte (str) : "forte", "moyenne" (≥ pNBFois) ou "faible" (≤ pNBFois).
        - pNBFois (int) : nombre minimal ou maximal de suivis selon la force.
        - pClasses (list ou str) : classes ciblées.
    g. **Logique interne :**
        1. Convertit en indices CP-SAT (matdex_1, matdex_2).

2. Sélectionne les classes comme pour matExcluSuite.
3. Pour chaque paire de créneaux consécutifs (j, h) et (j, h+1), crée :
    - BoolVar détectant pMatiere1 à (j,h)
    - BoolVar détectant pMatiere2 à (j,h+1)
    - BoolVar suivi_var = AND des deux précédents.
4. Selon pContrainte :
    - **forte** : AddImplication(var_mat1 → var_mat2) sur chaque paire.
    - **moyenne** : Add(sum(sequence_Suivi) ≥ pNBFois).
    - **faible** : Add(sum(sequence_Suivi) ≤ pNBFois).

h. Effets :

Garantit un enchaînement régulier (au moins, au plus, ou systématiquement) de deux matières.

3. **memNivMemCours**(pNiveau, pMatiere) :
    a. Objectifs :

    Assurer que **toutes** les classes d'un même niveau aient **simultanément** la même matière sur chaque créneau où elle est planifiée, et atteindre le volume horaire cible global bi-hebdo.

    b. Paramètres :
    - pNiveau (str) : ex. "6e".
    - pMatiere (str) : matière concernée.

    c. Logique interne :
    1. Ajoute (pNiveau, pMatiere) à la liste MEMNIV_ACTIVES.
    2. Pour chaque créneau (j,h) :
        - Crée un BoolVar var qui vaudra 1 si **toutes** les classes du niveau ont pMatiere à (j,h).
        - Lie chacun des IntVar matières (emploi_du_temps) à var via OnlyEnforceIf.
    3. Calcule le volume cible : (VOLUME_HORAIRE_BIHEBDO[pNiveau][pMatiere][semaine]) et impose
       Add(sum(positionMatiere) == volume_cible).

    d. Effets :

    Force la synchronisation totale des emplois du temps au sein d'un même niveau pour une matière donnée.

4. **matHorairDonneV2**(pClasses, pMatiere, pJour, pHorairMin, pHorairMax=None, pNBFois=None) :
    a. Objectifs :

    Contrôler qu'une ou plusieurs matière(s) figure(nt) un nombre précis de fois (pNBFois) dans un intervalle horaire sur un jour donné, pour certaines classes.

    b. Paramètres :

- pClasses (str ou list) : préfixe de classe ou liste de classes.
- pMatiere (str ou list) : nom de matière, groupe de matières, ou liste mixte.
- pJour (str) : jour ciblé.
- pHorairMin, pHorairMax (str) : bornes de l'intervalle (ex. "13h", "17h").
- pNBFois (int) : nombre d'occurrences attendues (par défaut longueur de l'intervalle).

c. Logique interne :

1. Traduit pJour en index, cherche dans HEURES les indices Hordex_Min, Hordex_Max.
2. Détermine la liste de matières réelles (filtre MATIERES).
3. Pour chaque classe, chaque créneau dans l'intervalle :
   - Crée un BoolVar indiquant la présence de la matière (emploi_du_temps == code_matiere).
4. Imposes soit == pNBFois (cas où on veut un nombre fixe), soit adapte pNBFois au volume disponible.

d. Effets :

Garantit qu'une matière (ou groupe) soit programmée un **nombre exact** d'heures sur un créneau ou un intervalle horaire dans la semaine.

5. **fusionner_groupes_vers_classes**(emploi_du_temps, emploi_du_temps_salles, emploi_du_temps_profs, solver, semaine) :

a. **Objectifs :**

Construire un dictionnaire textuel décrivant pour chaque créneau la **matière**, le **professeur** et la **salle**, tant pour les classes de base que pour leurs sous-groupes.

b. **Paramètres :**

- emploi_du_temps, emploi_du_temps_salles, emploi_du_temps_profs : les trois dicts de variables.
- solver : instance CpSolver déjà exécutée.
- semaine : "Semaine A" ou "Semaine B".

c. **Logique interne :**

1. Pour chaque classe de base et pour chaque créneau (j,h) :
   - Récupère mat_idx = solver.Value(emploi_du_temps[semaine][(cl,j,h)]).
   - Traduit en libellé matière + sélectionne le professeur (cas dict, liste ou string) via emploi_du_temps_profs.
   - Récupère l'indice de salle et déduit le nom + capacité.
   - Concatène "<matière>\n<prof>\n[<salle> – <cap> places]".
2. Répète pour chaque sous-groupe, puis assemble toutes ces chaînes dans un unique dict fusion_data[(classe, j, h)] → string.

d. **Effets :**
Produit la vue utilisateur finale de l'emploi du temps, prête à l'affichage ou à l'export JSON.

Ainsi, voici les 4 objets retourné par la fonction creer_modele :

1. **model :**
C'est une instance de CpModel (de ortools.sat.python.cp_model). Elle contient toutes les variables (IntVar et BoolVar) définies pour représenter l'affectation des matières, des salles, des professeurs, ainsi que les variables auxiliaires pour les compteurs d'équité, de permanence et de pénalités. Elle contient aussi l'ensemble des contraintes (non-chevauchement, volumes horaires, cantine, jours sans après-midi, suites de matières, surcharge de cartable, etc.) et l'objectif de minimisation (total des heures de permanence + poids des pénalités).
2. **emploi_du_temps :**
C'est un dictionnaire à deux niveaux : d'abord la clé "Semaine A" ou "Semaine B", puis le couple (classe, indice_jour, indice_créneau) pointant vers une variable IntVar. Chaque IntVar prend la valeur 0 si aucun cours n'est programmé, ou k ($1 \leq k \leq$ nombre de matières) pour indiquer la matière d'indice k–1 dans la liste MATIERES. Cette structure permet de récupérer, pour chaque classe et chaque créneau, quelle matière a été assignée.
3. **emploi_du_temps_salles :**
Même structure que pour emploi_du_temps (deux clés "Semaine A"/"Semaine B" et (classe, jour, créneau)), mais chaque IntVar vaut de 0 à NOMBRE_DE_SALLES. La valeur 0 signifie qu'aucun cours n'a lieu, et i ($1 \leq i \leq$ NOMBRE_DE_SALLES) signifie que la salle d'indice i–1 dans la liste SALLES_GENERALES a été assignée. On s'en sert pour garantir qu'une salle ne sert pas à deux classes en même temps et pour vérifier capacités et indisponibilités.
4. **emploi_du_temps_profs :**
Dictionnaire similaire, mais ne rempli que les clés pour les matières où plusieurs professeurs peuvent intervenir. La clé est le tuple (classe, indice_jour, indice_créneau, matière, "Semaine A"/"Semaine B"). La valeur est un IntVar dont le domaine va de 0 à n–1, avec n le nombre de professeurs possibles pour cette matière et ce niveau. La valeur indique l'indice du professeur choisi parmi la liste PROFESEURS[…]. Grâce à ces variables, on impose l'équité entre professeurs, l'interdiction de doubler un prof sur deux cours simultanés, et on peut, en post-traitement, déterminer précisément quel enseignant donne chaque cours.

4.14.2.5 Procédures externes

- Utilisation de OR-Tools pour créer le modèle.

4.14.3 Exécution de plusieurs runs

4.14.3.1 Objectif

Comparer plusieurs solutions obtenues en lançant le même modèle CP-SAT avec différentes seeds, afin de sélectionner la meilleure sur la base du taux de respect global des contraintes.

4.14.3.2 Modules

- time : chronométrage et estimation du temps restant.
- copy.deepcopy : sauvegarde de la meilleure solution.
- Fonctions de vérification (voir section 4.14.3.4).

4.14.3.3 Relations

- Boucle sur seed de 1 à NOMBRE_DE_RUNS.
- Appel à solve_et_verifie(...) pour chaque seed.
- Agrégation dans :
    - fusions_par_run: détails de l'emploi du temps pour chaque run.
    - taux_par_run: pourcentage de contraintes respectées.
- Identification et conservation de la meilleure run.

4.14.3.4 Types et attributs des variables

Entrées de la fonction solve_et_verifie :

- model (CpModel) : modèle complet construit par creer_modele.
- emploi_du_temps (Dict[(semaine, classe, jour, heure), IntVar]) : variables IntVar indiquant la matière planifiée sur chaque créneau.
- emploi_du_temps_salles (Dict[(semaine, classe, jour, heure), IntVar]) : variables IntVar assignant une salle à chaque créneau.
- emploi_du_temps_profs (Dict[(semaine, classe, jour, heure), BoolVar/IntVar]) : variables BoolVar/IntVar signalant la présence d'un professeur sur chaque créneau.
- seed (int) : graine aléatoire utilisée pour initialiser CpSolver, garantissant la reproductibilité et la diversité des solutions.
- JOURS, HEURES, MATIERES (List[str]) : listes définissant l'univers des créneaux et des matières.
- PROFESSEURS (Dict[str, Any]) : profil complet de chaque enseignant (indisponibilités, préférences, volumes horaires).
- CLASSES, CLASSES_BASE, SOUS_GROUPES_SUFFIXES (List[str]) : hiérarchie pédagogique pour gérer classes de base et sous-groupes.
- CAPACITES_CLASSES (Dict[str, int]) : capacité maximale de chaque groupe de base.
- INDISPONIBILITES_PROFS, INDISPONIBILITES_SALLES (Dict[str, List[(jour, heure)]]) : créneaux interdits pour professeurs et salles.

- config (Dict[str, Any]) : options globales (cantine, permanence, poids des pénalités optionnelles, etc.).
- AFFECTATION_MATIERE_SALLE (Dict[str, List[str]]) : liste de salles éligibles.
- fusionner_groupes_vers_classes (Callable) : fonction utilitaire pour agréger les sous-groupes en classes de base lors de l'export.

Sorties de la fonction solve_et_verifie :

- fusion_par_semaine (Dict[(semaine, classe, jour, heure), str]) : pour chaque créneau, libellé final combinant matière, salle et professeur retenus.
- taux (float) : pourcentage global de contraintes satisfaites.

Entrées de la fonction executer_runs_avec_suivi :

- NOMBRE_DE_RUNS (int) : nombre d'itérations à tester.
- model (CpModel) : modèle construit par creer_modele.
- emploi_du_temps (dict de IntVar) : pour chaque tuple (semaine, classe, jour, heure), la variable CP-SAT représentant la matière planifiée.
- emploi_du_temps_salles (dict de IntVar) : pour chaque créneau, la variable assignant la salle.
- emploi_du_temps_profs (dict de BoolVar/IntVar) : pour chaque créneau, la variable indiquant la présence d'un professeur.
- Constantes :
    - JOURS, HEURES, MATIERES (listes de chaînes).
    - PROFESSEURS (dict décrivant indisponibilités et préférences).
    - CLASSES, CLASSES_BASE, SOUS_GROUPES_SUFFIXES (listes).
    - CAPACITES_CLASSES (dict de capacités numériques).
    - INDISPONIBILITES_PROFS et INDISPONIBILITES_SALLES (dicts de listes de créneaux interdits).
    - config (dict brut de configuration : cantine, permanence, poids des contraintes optionnelles…).
    - AFFECTATION_MATIERE_SALLE (dict matière → liste de salles éligibles).
    - fusionner_groupes_vers_classes (callable pour regrouper sous-groupes avant export).
    - solve_et_verifie (callable qui lance la résolution CP-SAT et effectue les vérifications).

Sorties de la fonction executer_runs_avec_suivi :

- fusions_par_run (liste de triples) : pour chaque seed testé, un tuple contenant :
    - le seed (int).
    - la fusion détaillée de l'emploi du temps sous forme d'un mapping (semaine, classe, jour, heure) → "matière–salle–prof".
    - le dictionnaire resultats_contraintes décrivant, pour chaque catégorie de règle, le total de cas, le nombre de cas respectés et la liste des violations.
- taux_par_run (liste de couples) : pour chaque seed, le taux global de contraintes satisfaites (float entre 0 et 1).

- meilleur_seed (int) : la graine qui a produit la meilleure solution selon les critères de taux et de temps.
- meilleure_fusion (dict) : fusion de l'emploi du temps correspondant à meilleur_seed.
- meilleurs_resultats (dict) : détail des résultats de vérification pour la meilleure run.

Structure du dictionnaire resultats_contraintes :
Pour chaque catégorie (par exemple volume_horaire, non_chevauchement_salles, jours_sans_apres_midi, permanence, matExcluSuite, etc.), on trouve :

- total (int) : nombre de cas vérifiés pour cette contrainte.
- respectées (int) : nombre de cas conformes.
- violations (liste) : items détaillant chaque violation (par exemple le nom de la classe, la matière et l'écart constaté).

Processus interne de comparaison :

1. Initialisation de variables :
    - meilleur_taux à 0
    - meilleur_temps à l'infini
2. Pour chaque seed de 1 à NOMBRE_DE_RUNS :
    a. Appel de solve_et_verifie().
    b. Mesure de la durée d'exécution.
    c. Récupération de (fusion, taux, résultats).
    d. Si taux > meilleur_taux ou (taux == meilleur_taux et durée < meilleur_temps) : mise à jour de meilleur_seed, meilleure_fusion, meilleurs_resultats, meilleur_temps.
    e. Ajout des données à fusions_par_run et taux_par_run.
3. Retour des objets finaux : fusions_par_run, taux_par_run, meilleur_seed, meilleure_fusion, meilleurs_resultats.

4.14.3.5 Procédures externes :

- time.time() :nPour mesurer le temps de début et de fin de chaque run, calculer la durée d'exécution et estimer le temps restant.
- copy.deepcopy(obj) : Pour cloner entièrement la fusion (mapping de l'emploi du temps, résultats de contraintes, etc.) dès qu'une run bat le meilleur taux, afin de conserver cette solution "à froid" sans risque de mutation ultérieure.
- ortools.sat.python.cp_model.CpSolver() : Instanciation du solveur CP-SAT, paramétrage (avec la graine seed) et appel à solver.Solve(model) pour résoudre le modèle.
- tabulate.tabulate(data, headers, tablefmt=…) : Pour formater de façon lisible (Markdown/ASCII) la synthèse des performances (taux par seed, comparaisons, etc.) et faciliter le suivi visuel.

4.14.4 Génération de rapports & export

4.14.4.1 Objectif

Produire en console un aperçu complet de l'emploi du temps (élèves, professeurs, salles), puis exporter deux fichiers JSON prêts pour l'interface :
• emploi_du_temps_global.json : les deux semaines d'EDT structurées pour classes, professeurs et salles.
• tous_rapports_contraintes.json : le détail des vérifications de contraintes par run, avec statuts et pourcentage global.

4.14.4.2 Modules

- json : sérialisation des objets Python en JSON.
- tabulate : mise en forme des tableaux en console.

4.14.4.3 Relations

Prend en entrée :

- fusion_choisie : dictionnaire {semaine → {(classe, jour, heure) → libellé}}
- fusions_par_run : liste de tuples (seed, fusion, resultats_contraintes)
- resultats_choisis : statistiques d'une run retenue (total, respectées, détails)
- constraint_status : mapping catégorie → "obligatoire"|"optionnelle"
- dictProfs, dictSalles : structures vides mises à jour par les fonctions d'attribution.

Appelle dans l'ordre :

- afficher_rapport_contraintes(resultats_choisis)
- afficher_edts_elèves(fusion_choisie, CLASSES, SEMAINES, JOURS, HEURES)
- afficher_edts_profs_salles(fusion_choisie, SEMAINES, CLASSES, JOURS, HEURES, dictProfs, dictSalles, attributProfsCours, attributEDTSalles, affichCoursProfs, affichEDTSalles)
- generer_json_edt(fusion_choisie, SEMAINES, CLASSES_BASE, JOURS, HEURES, MATIERES)
- generer_json_rapports(fusions_par_run, constraint_status)

4.14.4.4 Types et attributs des variables

afficher_edts_elèves(fusion, CLASSES, SEMAINES, JOURS, HEURES)

- **Entrées**
  - fusion : dict {semaine(str) → { (classe(str), jour_idx(int), heure_idx(int)) → libellé(str) }}
  - CLASSES, SEMAINES, JOURS, HEURES : listes de chaînes

- **Sortie**
    - affichage console de l'emploi du temps des classes dans la semaine

afficher_edts_profs_salles(fusion, SEMAINES, CLASSES, JOURS, HEURES,dictProfs, dictSalles,attributProfsCours, attributEDTSalles,affichCoursProfs, affichEDTSalles)

- **Entrées**
    - fusion, SEMAINES, CLASSES, JOURS, HEURES : mêmes types que ci-dessus
    - dictProfs, dictSalles : dicts initialisés pour accumuler les EDT des profs et salles
    - attributProfsCours, attributEDTSalles : fonctions de mise à jour de ces dicts
    - affichCoursProfs, affichEDTSalles : fonctions d'affichage terminal
- **Sortie**
    - Reconstruit les pTable, met à jour dictProfs/dictSalles, puis affiche l'emploi du temps des professeurs et des salles dans la console

generer_json_edt(fusion, SEMAINES, CLASSES_BASE, JOURS, HEURES, MATIERES)

- **Entrées**
    - fusion : même structure que précédemment
    - SEMAINES, JOURS, HEURES : listes de chaînes
    - CLASSES_BASE : liste de classes de base (sans sous-groupes)
    - MATIERES : liste de chaînes
- **Sortie**
    - None (écrit le fichier emploi_du_temps_global.json contenant trois objets :
        - edt_classe
        - edt_prof
        - edt_salle)

generer_json_rapports(fusions_par_run, constraint_status)

- **Entrées**
    - fusions_par_run : liste de tuples (seed(int), fusion(dict), resultats_contraintes(dict))
    - constraint_status : dict { nom_contrainte(str) → statut("obligatoire"|"optionnelle") }
- **Sortie**
    - Écrit le fichier tous_rapports_contraintes.json avec, pour chaque seed, total, respectées, détails et le pourcentage global)

4.14.4.5 Procédures externes :

- open() : Ouvre un flux de fichier en écriture (mode "w") pour émettre les JSON finaux.
- json.dump() : Sérialise en JSON.
- tabulate.tabulate() : Mise en forme tabulaire pour l'affichage console du rapport, en générant des tableaux Markdown ou ASCII.

5. Glossaire

Ce glossaire regroupe les principaux termes techniques utilisés dans ce document. Il vise à faciliter la compréhension de celui-ci, en particulier pour les nouveaux contributeurs ou utilisateurs.

Terme	Définition
1. Application locale	Application qui fonctionne entièrement sur l'ordinateur de l'utilisateur, sans connexion à un serveur distant.
2. Appuyer sur "CTRL + C"	Raccourci clavier permettant d'interrompre un programme en cours d'exécution dans le terminal.
3. Archive .ZIP	Fichier compressé contenant l'ensemble des fichiers d'un projet, peut être extrait sur n'importe quel système.
4. Bibliothèque (ou package)	Ensemble de fonctions et outils Python préprogrammés, installables via pip, utilisés pour étendre les capacités d'un projet.
5. Clé/valeur	Format structuré dans lequel chaque "clé" (ex : "nom") est associée à une "valeur" (ex : "Mme Dupont"), utilisé notamment dans les fichiers JSON.
6. Constraints (contraintes)	Règles définies pour générer un emploi du temps valide (ex : "aucun cours après 17H", "un prof ne peut enseigner à deux classes en même temps").
7. Dépendance	Bibliothèque externe nécessaire au bon fonctionnement du programme (ex : Dash, OR-Tools).
8. Dash	Framework Python utilisé pour créer des interfaces web interactives et réactives.
9. Emploi du temps (EDT)	Organisation hebdomadaire des cours par classe, professeur, matière, salle et créneau horaire.
10. Export (PDF)	Fonction permettant de sauvegarder les résultats générés (emplois du temps) dans un format consultable ou imprimable.
11. Fichier JSON (.json)	Format texte structuré utilisé pour stocker les données de manière hiérarchique (enseignants, matières, contraintes…).

12. Framework	Ensemble cohérent d'outils permettant de structurer et développer plus facilement une application (ex : Dash pour l'interface).
13. Interface web	Interface utilisateur accessible via un navigateur internet, permettant de gérer l'application sans connaissance technique.
14. Invite de commande	Voir Terminal. Terme utilisé sous Windows.
15. JSON (JavaScript Object Notation)	Voir Fichier JSON.
16. Lancer l'application	Action qui consiste à exécuter le fichier principal (app.py) pour démarrer l'interface web localement.
17. Navigateur web	Logiciel permettant d'accéder à l'interface utilisateur (ex : Chrome, Firefox, Edge…).
18. Optimisation	Processus informatique consistant à chercher la meilleure solution possible dans un ensemble de contraintes.
19. OR-Tools	Bibliothèque d'optimisation développée par Google, utilisée pour résoudre des problèmes complexes comme la génération d'un emploi du temps.
20. Paramètre / Configuration	Valeurs personnalisées saisies par l'utilisateur pour adapter l'application à son contexte (heures maximales, salles disponibles…).
21. PATH (variable d'environnement)	Chemin d'accès utilisé par le système pour exécuter des programmes ; ajouter Python au PATH permet d'utiliser python dans le terminal.
22. PDF	Formats de fichier permettant d'exporter les emplois du temps : PDF pour l'impression, HTML pour la consultation en ligne.
23. pip	Gestionnaire de paquets Python permettant d'installer des bibliothèques (ex : "pip install dash").
24. Port (localhost:8050)	Adresse locale sur laquelle l'application Dash est accessible dans un navigateur.
25. Projet TER	Travail d'Étude et de Recherche réalisé dans le cadre d'un cursus universitaire.

26. Python	Langage de programmation utilisé pour le développement de l'application. Version 3.10 ou supérieure recommandée.
27. Résolution de contraintes	Méthode consistant à satisfaire un ensemble de conditions logiques, ici appliquée à la construction d'un emploi du temps.
28. Serveur local	Programme lancé sur l'ordinateur de l'utilisateur, permettant d'afficher l'application dans un navigateur via une adresse comme : "http://127.0.0.1:8050".
29. Système d'exploitation (OS)	Logiciel principal d'un ordinateur : Windows, macOS ou Linux.
30. Terminal	Interface textuelle permettant d'interagir avec le système d'exploitation. Également appelée Invite de commandes sous Windows.
31. venv (environnement virtuel)	Environnement Python isolé dans lequel les bibliothèques sont installées indépendamment du reste du système.
32. Visualisation	Affichage graphique des résultats (emplois du temps), souvent accompagné de couleurs, de tableaux ou d'aperçus interactifs.

6. Références

Cette section recense les documents et sources ayant servi de support à la conception de ce document.

- Cahier des charges initial : définissant les objectifs fonctionnels et techniques attendus de l'outil.
- Comptes rendus de réunions projet : échanges réguliers avec la maîtresse d'ouvrage Mme. Landry et M. Pellier entre janvier et mai 2025.
- Manuel d'installation : décrivant la procédure de mise en place du projet en environnement local ou serveur.
- Documentation de Google du solver OR-Tools.
- Documentation du framework d'interface : Dash Plotly.

7. Index

Voici la liste les mots-clés du document et où les trouver dans celui-ci :