ENTWANNE

Geste de savoir

Ordonner des horaires de train

17 juillet 2021

Table des matières

Introduction

Avez-vous déjà essayé de rejoindre Besançon depuis Nancy en train, ou plus précisément les gares de Nancy-Ville et Besançon-Viotte ? Ces deux gares, distantes de 160km, ne bénéficient pas d'une ligne ferroviaire directe.

La faute à la chaîne des Vosges qui limite les traversées possibles entre Lorraine et Franche-Comté et à l'enclavement de la Haute-Saône qui ne dispose que de peu de lignes encore actives.
En effet, les grandes lignes autour (projets LGV Est et LGV Rhin-Rhône) ne font que contourner ce département qui se retrouve isolé et mal desservi.

Plusieurs possibilités se présentent alors :

— Le tracé le plus direct, avec correspondances à Épinal et Belfort, en un peu moins de 4h de trajet pour les plus courts.
— Le plus rapide, qui peut se faire en 3h30 en contournant par l'ouest avec correspondance à Dijon.
— Le plus confortable, un TGV direct de Nancy à Besançon TGV (puis correspondance TER), bénéficiant de la LGV à partir de Strasbourg qui permet de faire le trajet en 4h.
— Quelques variantes à ces précédents choix (passage par Metz, correspondance à Strasbourg et Mulhouse, liaison Dijon - Besançon TGV)

J'ai volontairement omis les trajets qui proposent de passer par Paris.

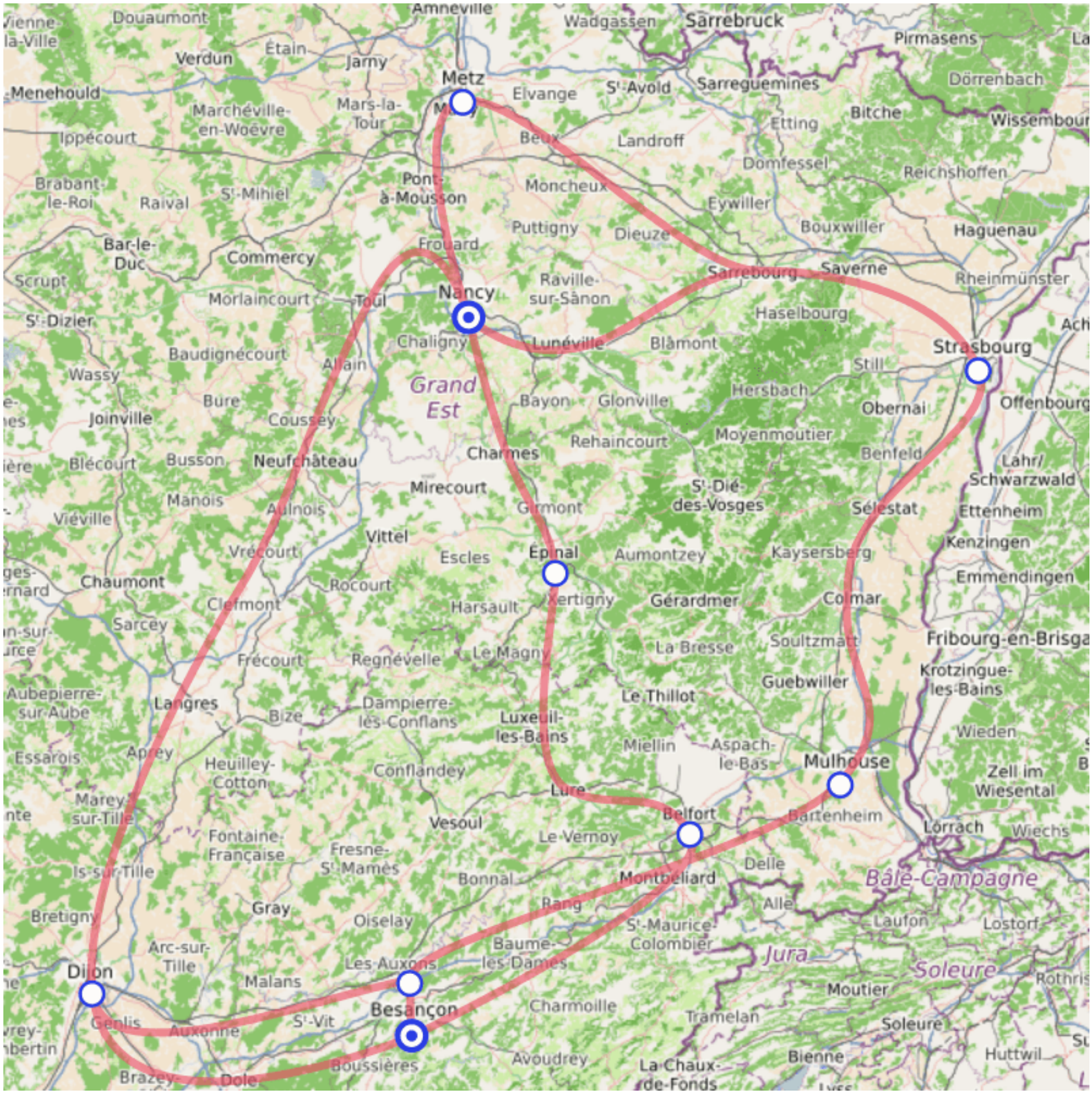

FIGURE 0.1. – Carte des différents lignes.

Bref, c'est assez compliqué et ça emprunte différents types de trains, difficile alors de savoir synthétiquement quels peuvent en être les horaires.
Mon objectif était donc de pouvoir générer une fiche horaire Nancy-Besançon me présentant ces différentes possibilités.

1. Passer ma route

Pour arriver à générer mes fiches horaires idéales, j'avais donc quelques contraintes à remplir.

Il fallait d'abord que je puisse agréger les données de plusieurs trains, la partie assez ennuyante du travail. Je ne me suis pas intéressé aux API proposées, j'ai «simplement» utilisé le site oui.sncf pour explorer les trajets possibles suivant les jours et les reporter dans un fichier CSV. Je notais donc une ligne pour chaque train (ex : *TER 835013*), avec gares et heures de départ/arrivée, ainsi que les jours de circulation.

Ce fichier comprenait alors les données brutes des trains, à partir desquelles je voulais élaborer moi-même les trajets, en groupant entre-elles les correspondances et en les ordonnant par heure de départ.
Il est aussi nécessaire de les filtrer par jours, pour n'avoir que des trajets cohérents selon le jour de la semaine.

Je voulais aussi garder une certaine liberté de choix et ainsi ne pas imposer une correspondance en TGV quand un TER était possible sur le même trajet dans les mêmes heures. Ça faisait surtout partie du travail de sélection des trains à mettre dans le fichier CSV, mais c'était aussi une contrainte à prendre en compte : plusieurs trains pour un même trajet peuvent faire partie d'un même groupe de correspondances.

Enfin, je souhaitais obtenir une «belle» présentation. Alors entendons-nous sur le mot : il s'agit juste d'avoir les données correctement représentées, sous la forme d'un tableau HTML ou markdown, afin que les fiches soient lisibles.
Donc une colonne correspondant à chaque arrêt et les arrêts proprement ordonnés.

2. Parsons vite

J'aurais pu faire appel à une API pour exécuter les requêtes et agréger directement les données liées aux différents trajets… J'aurais pu.

Mais ce n'était pas mon but, je ne voulais pas commencer à me perdre dans ces documentations et ne jamais venir à bout de mon projet. Alors j'allais utiliser une technique plus artisanale : faire une recherche pour chaque jour de la semaine sur chacun des trajets, et inscrire tout ça dans un fichier CSV.
Il me fallait connaître les gares et heures de départ et d'arrivée, mais aussi les intermédiaires (au cas où d'autres correspondances seraient possibles), ainsi que les jours de la semaine où le train circulait. J'enregistrais aussi le type de train et son numéro pour me permettre de le retrouver ensuite.

J'avais choisi d'ignorer toute contrainte liée aux jours fériés, aux vacances scolaires ou au travaux, je considérais qu'un train roulerait toujours de la même manière tous les lundis de l'année par exemple.

Je me suis donc retrouvé avec un fichier CSV assez conséquent, où j'avais déjà pré-filtré les trajets ne menant à rien (je n'y faisais pas figurer tous les Nancy-Épinal en sachant qu'il n'y avait pas de correspondance derrière) dont voici un extrait :

```
```

La suite consistait donc à parser ce fichier et à construire les données liées au train. J'implémentais pour ça un type `Train` reprenant toutes les caractéristiques listées au-dessus, qui représentait

une liste d'arrêts avec horodatage.
J'y avais ajouté quelques raccourcis pour accéder directement au départ et au terminus.

> Contenu masqué n°1

Le parsing en lui-même était assez trivial : module `csv`, format de l'heure et correspondance des jours.

> Contenu masqué n°2

Mais maintenant j'avais sous la main une liste de trains exploitables avec leurs arrêts.

3. C'est quand qu'on va où ?

Une fiche horaire présente les trains sous la forme d'un tableau, et dispose donc d'une liste d'arrêts. Cette liste est facile à obtenir dans le cas d'une unique ligne puisque les arrêts sont déjà linéaires, mais ce n'est pas du tout le cas de la carte que je présente en introduction.

Comment linéariser les différents arrêts entre des lignes de train qui partent dans différentes directions ?

L'idée qui me vint était de trier les arrêts selon leur «distance» par rapport au point de départ. Le point de départ était facilement identifiable : c'est le seul arrêt où n'arrive aucun train (ils ne peuvent qu'en partir).
Partant de là, je calculais la distance comme le temps minimum (le nombre d'arrêts et le temps total) pour rejoindre une gare depuis le point de départ, de proche en proche.

Je procédais pour ça en deux temps. D'abord en identifiant pour chaque gare le trajet minimal qui permettait de s'y rendre depuis une autre gare (quelle qu'elle soit). Puis je normalisais cela pour calculer la distance depuis le point de départ plutôt que depuis la gare précédente. Je ne tenais pas compte des temps de correspondance.

Il fallait faire attention aux temps d'arrivée qui pouvaient parfois être le lendemain matin, et donc à une heure antérieure à celle de départ. Cela se gérait bien avec une condition sur ce cas particulier.

J'obtenais donc une liste triée de mes arrêts mais je remarquais vite un petit problème : les listes pour le trajet aller et retour n'étaient pas cohérentes. J'aurais pensé que l'une serait simplement l'inverse de l'autre, mais c'aurait justement été trop simple.

J'ai assez vite identifié le problème, il s'agissait des temps de trajet pour Dijon et Mulhouse. En effet, il faut moins de temps depuis Nancy pour se rendre à Dijon qu'à Mulhouse… mais c'est vrai aussi depuis Besançon !
Mulhouse se retrouvait donc systématiquement après Dijon dans la liste des arrêts, à l'aller comme au retour.

Pour corriger le problème, j'ai donc calculé les «distances» minimales jusqu'à l'arrivée en plus des distances depuis le départ, et je combinais les deux en les soustrayant. Le point d'arrivée était lui aussi facilement identifiable puisqu'aucun train n'en partait.

Contenu masqué n°3

Cette fois-ci, j'avais une liste ordonnée et cohérente[1] footnote:1 des arrêts !

Nancy
Metz
Épinal
Strasbourg
Belfort Ville
Mulhouse
Dijon
Besancon TGV
Besancon Viotte

TABLE 3.2. – Liste des arrêts.

4. Colore le monde

Je pouvais maintenant entrer dans le vif du sujet, à savoir grouper les trajets entre-eux pour obtenir des itinéraires NancyBesançon.
Je me souvenais pour cela d'un algorithme de construction de labyrinthe qui consistait à colorer les cases puis à les fusionner en groupes au fur et à mesure que des murs étaient ouverts : https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe#Fusion_al%C3%A9atoire_de_chemins

FIGURE 4.2. – Coloration de labyrinthe, crédits Yann Langlais .

Je pourrais utiliser le même principe pour mes trains, en les fusionnant par groupe avec les autres trains environnants dans les mêmes gares.
Il s'agissait en fait de l'algorithme Union-Find dont j'ai fait une implémentation naïve. Je voulais néanmoins en faire quelque chose de générique que je pourrais réutiliser dans un autre

1. [2] footnote:1 Cohérente dans le sens où elle reste similaire dans un sens et dans l'autre. On remarque toujours quelques «bizarreries» comme le fait que Belfort apparaisse avant Mulhouse.

contexte, et j'ai donc mis en place un objet `grouper`.
Cet objet, je pourrais le manipuler pour ajouter de nouveaux éléments (chaque élément appartement à un nouveau groupe) et pour fusionner des groupes. Je pourrais aussi lui demander la liste des groupes et les éléments présents dans chaque groupe.
J'allais pour ça me heurter aux limites de Python car je voulais pouvoir stocker tous types d'objets (même des mutables) dans des ensembles. Il me fallait alors mettre en place un *wrapper* (`HashInstance`) pour rendre *hashable* tout objet.

Contenu masqué n°4

Je pouvais ensuite attribuer des «couleurs» à chaque train et les fusionner avec les trains précédent/suivant (dans chaque gare) pour former des groupe de correspondances. Les trains ne pourraient être mis en correspondance qu'avec des trains du même groupe.
Dans chaque groupe, les trains étaient ordonnés selon leurs heures d'arrivée et de départ, et les groupes entre-eux étaient triés selon leurs heures de départ. La clé d'un groupe correspondant à l'heure de départ minimale dans ce groupe.

Contenu masqué n°5

5. Chaque jour de plus

Mais je n'avais pas fini de grouper. Après avoir groupé par correspondances je devais grouper par jours.
En effet, les trains peuvent circuler certains jours et pas les autres et je voulais avoir un affichage condensé de tout ça. J'allais alors identifier les groupes de jours, c'est-à-dire les jours pour lesquels les trains seraient identiques.
Je calculais donc toutes les intersections possibles entre les jours de circulation des différents trains, et j'en déduisais les groupes distincts de jours identiques.

Contenu masqué n°6

6. J'entends siffler le train

Avec tout ça bout à bout, je les avais mes horaires ! Je n'avais plus qu'à afficher un beau tableau, gérer quelques arguments et coder la glue autour.
Pour le tableau, je choisissais de pouvoir gérer à la fois un export HTML et un export Markdown et j'implémentais mes fonctions par rapport à ça, sans vouloir les rendre trop spécifiques pour un format particulier.
C'est pourquoi je gardais une fonction `iter_table` à part qui ne ferait que produire les lignes du tableau sous forme de listes.
Le choix du format de sortie se retrouvait aussi côté arguments de la ligne de commande où il pouvait être précisé. J'ajoutais une autre option pour unifier les jours, c'est-à-dire présenter sous un même tableau tous les trains quels que soient leurs jours de circulation.

Contenu masqué n°7

L'ensemble du code de ce projet peut être trouvé sur le dépôt suivant : https://github.com/entwanne/horaires_trains

7. Ne partez pas sans moi

Après tout ça, je peux maintenant vous montrer les belles fiches horaires que j'ai obtenues. Et comme on le voit, ça laisse assez peu de possibilités pour faire le trajet rapidement.

7.1. Nancy-Ville → Besançon-Viotte

Contenu masqué n°8

7.2. Besançon-Viotte → Nancy-Ville

Contenu masqué n°9

8. On avance, on avance, on avance

Est-ce que ce projet est fini ? Je dirais que oui parce que je n'ai plus l'intention d'y toucher, d'où ce billet.
Est-ce que j'aurais pu aller plus loin ? Oui aussi.
Déjà il faudrait encore déboguer un coup, je peux avoir sur certains jours des trains qui apparaissent sans correspondances, parce qu'ils figurent dans un groupe dont les correspondances ne sont disponibles que pour d'autres jours.
Ensuite, il faudrait aussi gérer correctement les horaires qui s'étalent sur plusieurs jours. Le problème ne se pose pas trop car il n'y a pas beaucoup de trains qui roulent autour de minuit, mais j'en ai tout de même un dans mes résultats, et son bon fonctionnement est plus dû à un fix un peu crade qu'autre chose.
Enfin, je pourrais utiliser une API pour connaître les itinéraires et télécharger les horaires plutôt que d'avoir à faire ça manuellement, tout en filtrant les trains inutiles (sans correspondance possible).
Mais bon, j'ai eu ce que je voulais et je m'en contente. Ça me fait de beaux tableaux.

Conclusion

Comme on le constate, ordonner correctement des horaires de train n'est pas quelque chose d'aussi facile qu'on aurait pu le penser.
La SNCF elle-même a parfois quelques loupés.

Plouaret-Trégor	-	06.27
LANNION	**-**	**06.04**
Plounérin	-	06.34
Plouigneau	-	06.43
MORLAIX	**05.16**	**06.50**
BREST	**05.52**	**07.35**

FIGURE 8.3. – Fiche horaire de la ligne St Brieuc - Lannion - Morlaix - Brest

FIGURE 8.4. – Panneau d'affichage en gare de Tours—Merci @ache

---

Ressources complémentaires :

- Lignes ferroviaires de Haute-Saône, anciennes et actuelles :
  - Ligne de Besançon-Viotte à Vesoul ;
  - Ligne de Blainville - Damelevières à Lure ;
  - Ligne d'Aillevillers à Port-d'Atelier-Amance ;
- Railway Routing pour visualiser les lignes de train :
  - Ligne Nancy—Nice (carte) ;
  - Ligne Luxembourg—Marseille (carte).

Contenu masqué

Contenu masqué n°1

```python
class Train:
    """
    Representation of a train with multiple stops
    """

    def __init__(self, type, id, days=frozenset(), **stops):
        if len(stops) < 2:
            raise ValueError('At least 2 stops are needed')
        self.type = type
        self.id = id
        self.days = days
        self.stops = stops

    def __iter__(self):
        "Iterate over all (stop, stop_time) couples"
        return iter(self.stops.items())

    def iter_parts(self):
        ⌋
            "Iterable over all segments of the trip (couples of stop items)"
        it = iter(self)
        src_item = next(it)
        for dst_item in it:
            yield src_item, dst_item
            src_item = dst_item

    @property
    def departure(self):
        return next(iter(self.stops.keys()))

    @property
    def departure_time(self):
        return next(iter(self.stops.values()))

    @property
    def arrival(self):
        return next(reversed(self.stops.keys()))

    @property
    def arrival_time(self):
        return next(reversed(self.stops.values()))
```

Listing 1 – train.py

Retourner au texte.

Contenu masqué n°2

```python
import csv
import datetime
import sys

from .train import Train


time_fmt = '%H:%M'
day_names = ('L', 'Ma', 'Me', 'J', 'V', 'S', 'D')
days_mapping = {name: i for i, name in enumerate(day_names)}


def load(f=sys.stdin):
    "Load trains from a CSV file"
    reader = csv.reader(f)
    return map(load_train, reader)


def load_train(row):
    """
    Load train from a row

    ⌋
        load_train(['T', '#1', 'SD', 'viridian', '12:30', 'pewter', '13:00', '
    ⌋
        => Train('T', '#1', {5, 6}, viridian=time(12, 30), pewter=time(13, 0),
    """
    train_type, train_id, days, *stops = row
    days = {i for d, i in days_mapping.items() if d in days}
    stops = zip(stops[::2], stops[1::2])
    stops = {
        stop: datetime.datetime.strptime(time, time_fmt).time()
        for stop, time in stops
    }
    return Train(train_type, train_id, days, **stops)
```

Listing 2 – parse.py

Retourner au texte.

Contenu masqué n°3

```python
import datetime


def diff_time(t1, t2):
```

Contenu masqué

```
    "Compute difference between two time objects"
    t1 = datetime.datetime.combine(datetime.date.min, t1)
    t2 = datetime.datetime.combine(datetime.date.min, t2)
    # t2 in the next day
    if t2 < t1:
        t2 += datetime.timedelta(days=1)
    return (t2 - t1).total_seconds()


def compute_distances(trains):
    """
    Compute minimal distances to neighboring stops (previous & next)
    for all stops
    Return two distances dicts (minimal distances to stop & from stop)

    compute_distances([
        Train(viridian=time(10), pewter=time(20)),
        Train(viridian=time(10), pewter=time(30)),
        Train(pewter=time(50), cerulean=time(80)),
    ])
    => (
        {
            'viridian': (0, 0, None),
            'pewter': (10, 1, 'viridian'),
            'cerulean': (30, 1, 'pewter'),
        },
        {
            'cerulean': (0, 0, None),
            'pewter': (30, 1, 'cerulean'),
            'viridian': (10, 1, 'pewter'),
        },
    )
    """
    distances_to = {}
    distances_from = {}
    default = (0, 0, None)

    for train in trains:
        for (src, time_src), (dst, time_dst) in train.iter_parts():
            dist = diff_time(time_src, time_dst)
            distances_to.setdefault(src, default)
            distances_from.setdefault(dst, default)

            old_dist, _, _ = distances_to.get(dst, default)
            if not old_dist or dist < old_dist:
                distances_to[dst] = dist, 1, src

            old_dist, _, _ = distances_from.get(src, default)
```

Contenu masqué

```
            if not old_dist or dist < old_dist:
                distances_from[src] = dist, 1, dst

    return distances_to, distances_from


def normalize_distances(distances):
    """
    Normalize a distances dict
⌋
        Re-compute the distances to make them absolute from departure.arrival

    normalize_distances({
        'viridian': (0, 0, None),
        'pewter': (10, 1, 'viridian'),
        'cerulean': (30, 1, 'pewter'),
    })
    => {
        'viridian': (0, 0, None),
        'pewter': (10, 1, None),
        'cerulean': (40, 2, None),
    }
    """
    def set_absolute_dist(stop):
        # Walk recursively through stops to update min_dist &
            n_stops
        dist, n, src = distances[stop]

        if src is not None:
            d, i, src = set_absolute_dist(src)
            dist += d
            n += i
            distances[stop] = dist, n, src

        return dist, n, src

    for stop in list(distances):
        set_absolute_dist(stop)

    return distances


def get_sorted_stops(trains):
    """
    Get a sorted list of all stops by computing distances

    get_sorted_stops([
        Train(viridian=time(10), pewter=time(20)),
        Train(viridian=time(10), pewter=time(30)),
        Train(pewter=time(50), cerulean=time(80)),
```

```
    ])
    => ['viridian', 'pewter', 'cerulean']
    """
    distances_to, distances_from = compute_distances(trains)
    normalize_distances(distances_to)
    normalize_distances(distances_from)

    def sort_key(key):
        # Stops are sorted by closest from departure & furthest
            from arrival
        dist1, n1, _ = distances_to[key]
        dist2, n2, _, = distances_from[key]
        return dist1 - dist2, n1 - n2

    return sorted(distances_to, key=sort_key)
```

Listing 3 – sort.py

Retourner au texte.

Contenu masqué n°4

```
from contextlib import contextmanager


class _Group:
    "Identify a mergeable group"
    def __init__(self, key):
        self.key = key
        self.set = {self}

    def merge(self, rhs, merge_key_func=None):
        # Merge = union of both groups sets
        self.set.update(rhs.set)
        if merge_key_func:
            self.key = merge_key_func(self.key, rhs.key)
        for item in rhs.set:
            item.key = self.key
            item.set = self.set

    def clear(self):
        self.set.clear()


class HashInstance:
    "Wrap any Python object to make it hashable"
    def __init__(self, obj):
        self.obj = obj
```

```

    def __eq__(self, rhs):
        if not isinstance(rhs, __class__):
            return NotImplemented
        return self.obj is rhs.obj

    def __hash__(self):
        return object.__hash__(self.obj)


class Grouper:
    "Abstraction to manipulate groups over common objects"
    def __init__(self):
        self._groups = set()
        self._groups_by_object = {}

    def add(self, obj, key=None):
        "Create a new group for an object"
        group = _Group(key)
        # Register the object in the group
        group.obj = obj
        self._groups.add(group)
        obj = HashInstance(obj)
        self._groups_by_object[obj] = group

    def _obj_group(self, obj):
        "Get the group associated to an object"
        objh = HashInstance(obj)
        return self._groups_by_object[objh]

    def get_key(self, obj):
        return self._obj_group(obj).key

    def equal(self, lhs, rhs):
        "Compare groups of two objects"
        lgroup = self._obj_group(lhs)
        rgroup = self._obj_group(rhs)
        return lgroup.set is rgroup.set

    def merge(self, lhs, rhs, merge_key=None):
        "Merge groups of two objects"
        lgroup = self._obj_group(lhs)
        rgroup = self._obj_group(rhs)
        lgroup.merge(rgroup, merge_key_func=merge_key)

    def groups(self):
        "Iterate over objects by group"
        for group in self._groups:
            yield group.key, [g.obj for g in group.set]

```

```
    def clear(self):
        "Clear all groups"
        for group in self._groups:
            del group.obj
            group.clear()
        self._groups.clear()
        self._groups_by_object.clear()


@contextmanager
def grouper():
    "Create & clear a Grouper object"
    g = Grouper()
    try:
        yield g
    finally:
        g.clear()
```

Listing 4 – utils/grouping.py

Retourner au texte.

Contenu masqué n°5

```
from .utils import grouper


def get_next_train(train, trains):
    "Get direct/earliest connection after a train"
    trains = [t for t in trains if t.departure_time >
        train.arrival_time]
    if trains:
        return min(trains, key=lambda t: t.departure_time)


def get_prev_train(train, trains):
    "Get direct/latest connection before a train"
    trains = [t for t in trains if t.arrival_time <
        train.departure_time]
    if trains:
        return max(trains, key=lambda t: t.arrival_time)


def group_trains(trains, sorted_stops):
    """
    Group trains by connections
    All trains that are connected belong to a same group
    Return sorted groups as lists of trains
```

Contenu masqué

```
23
24      group_trains([
25          Train('T', '#1', viridian=time(12, 30), pewter=time(13, 0)),
26          Train('T', '#2', viridian=time(16, 0), pewter=time(16, 30)),
27          Train('T', '#3', pewter=time(13, 30), cerulean=time(14, 30)),
28          Train('T', '#4', pewter=time(17, 0), cerulean=time(18, 0)),
29      ], ['viridian', 'pewter', 'cerulean'])
30      => [
31          [Train('T', '#1', ...), Train('T', '#3', ...)],
32          [Train('T', '#2', ...), Train('T', '#4', ...)],
33      ]
34      """
35      trains_from_stop = {stop: [] for stop in sorted_stops}
36      trains_to_stop = {stop: [] for stop in sorted_stops}
37
38      with grouper() as g:
39          for train in trains:
40              # Groups are associated to a departure time
41              # when merged, groups get associated to earlier
                    departure times
42              g.add(train, train.departure_time)
43              trains_from_stop[train.departure].append(train)
44              trains_to_stop[train.arrival].append(train)
45
46          for stop in sorted_stops:
47              # Connect each train with next one
48              for train in trains_from_stop[stop]:
49                  next_train = get_next_train(train,
                        trains_from_stop[train.arrival])
50                  if next_train:
51                      g.merge(train, next_train, merge_key=min)
52
53              # Connect each train with previous one
54              for train in trains_to_stop[stop]:
55                  prev_train = get_prev_train(train,
                        trains_to_stop[train.departure])
56                  if prev_train:
57                      g.merge(train, prev_train, merge_key=min)
58
59          # Sort trains in groups by time of arrival & departure
60          grouped_trains = {
61              time: sorted(trains, key=lambda t: (t.departure_time,
                        t.arrival_time))
62              for time, trains in g.groups()
63          }
64          # Sort groups by departure time of the first train
```

```
65          return [trains for _, trains in
                sorted(grouped_trains.items())]
```

Listing 5 – group.py

Retourner au texte.

Contenu masqué n°6

```
1  def get_day_groups(trains):
2      """
3      Get a list of all distinctive sets of days
4
5      get_day_groups([
6          Train(days={1, 2, 3}),
7          Train(days={0, 1}),
8          Train(days={4}),
9      ])
10     => [{0}, {1}, {2, 3}, {4}]
11     """
12     day_sets = {frozenset(train.days) for train in trains}
13     all_days = frozenset.union(*day_sets)
14     groups = {all_days}
15
16     for day_set in day_sets:
17         couples = ((group & day_set, group - day_set) for group in
               groups)
18         groups = {group for couple in couples for group in couple
               if group}
19
20     return sorted(groups, key=tuple)
```

Listing 6 – group.py

Retourner au texte.

Contenu masqué n°7

```
1  day_names = ('L', 'Ma', 'Me', 'J', 'V', 'S', 'D')
2  day_full_names = (
3      'Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi',
           'Dimanche',
4  )
5
6
7  def dump_days(days, full=False):
8      "Get printable format for days set"
```

```
    if full:
        return ', '.join(day_full_names[i] for i in sorted(days))
    else:
        return ''.join(day_names[i] for i in sorted(days))
```

Listing 7 – parse.py

```
from .parse import dump_days


def iter_table(sorted_stops, grouped_trains, filter_days=set(), *,
    print_days=True):
    header = ['Train']
    if print_days:
        header.append('Jours')
    header.extend(sorted_stops)
    yield header

    for group in grouped_trains:
        yield
        for train in group:
            if not (train.days >= filter_days):
                continue
            row = [f'{train.type} {train.id}']
            if print_days:
                row.append(dump_days(train.days))
            stops = (train.stops.get(stop) for stop in
                sorted_stops)
            row.extend(time.strftime(time_fmt) if time else '' for
                time in stops)
            yield row
```

Listing 8 – print.py

Retourner au texte.

Contenu masqué n°8

8.0.1. Lundi, Mardi, Mercredi, Jeudi, Vendredi

Train	nancy	metz	epinal	stras-bourg	bel-fort_ville	mul-house	dijon	besan-con_tgv	besan-con_viotte
TER 835013	07:14			08:42					
TGV 9877				09:05		10:01		10:46	
TER 894518								11:04	11:19

TER 836380	07:54						10:29		
TER 894213							11:09		12:05
TER 835015	08:14			09:41					
TER 832361				10:21		11:14			
TGV 6704						11:58		12:43	
TER 894566								13:39	13:54
TER 835755	08:55		09:53						
TER 894609			09:59		11:25				
TER 894026					11:36				12:46
TER 839161	11:00			12:33					
TGV 9879				13:03		13:59		14:44	
TER 894528								14:54	15:09
TGV 5537	12:10			13:36		14:51		15:39	
TER 894575								15:48	16:01
TER 834024	12:55		13:53						
TGV 2571	14:05		14:47						
TER 894619			14:59		16:25				
TER 894040					16:36				17:46

Contenu masqué

TER 835021	14:15			15:41					
TGV 9580				16:14		17:09		17:55	
TER 894538								18:07	18:22
TER 835775	17:55		18:53						
TER 894627			18:59		20:25				
TER 894062					20:36				21:49
TER 836382	16:54						19:27		
TGV 9896							19:46	20:12	
TER 894267							19:50		20:55
TER 894560								20:22	20:37

8.0.2. Samedi

Train	nancy	metz	epinal	strasbourg	belfort_ville	mulhouse	dijon	besancon_tgv	besancon_viotte
TER 88503	06:50	07:27							
TGV 9877		07:58		09:05		10:01		10:46	
TER 894518								11:04	11:19
TER 836380	07:58						10:29		
TER 894213							11:09		12:05
TGV 6741							11:36	12:09	12:20

TGV 5537	12:10			13:36		14:51		15:39	
TER 894575								15:48	16:01
TER 835819	13:20		14:18						
TER 894619			14:59		16:25				
TER 894034					17:04				18:28
TER 835771	16:20		17:18						
TER 894627			18:59		20:25				
TER 894062					20:36				21:49

8.0.3. Dimanche

Train	nancy	metz	epinal	strasbourg	belfort_ville	mulhouse	dijon	besancon_tgv	besancon_viotte
TER 835041	08:16			09:41					
TER 96217				10:51		11:44			
TGV 6704						12:01		12:47	
TER 894566								13:39	13:54
TER 835043	11:16			12:41					
TGV 9879				13:03		13:59		14:44	
TER 894528								14:54	15:09

TGV 5537	12:27			13:47		14:51		15:39	
TER 894575								15:48	16:01
TER 835045	14:15			15:43					
TGV 9580				16:14		17:09		17:55	
TER 894538								18:07	18:22
TER 835775	17:55		18:53						
TER 894627			18:59		20:25				
TER 894062					20:36				21:49
TER 836382	16:54						19:27		
TGV 9896							19:46	20:12	
TER 894267							19:50		20:55
TER 894560								20:20	20:35

Retourner au texte.

Contenu masqué n°9

8.0.4. Lundi, Mardi, Mercredi, Jeudi, Vendredi

Train	besan-con__viotte	besan-con__tgv	dijon	mul-house	bel-fort__ville	stras-bourg	epinal	metz	nancy
TER 894517	09:55	10:10							
TGV 9898		10:34		11:23		12:27		13:24	
TER 88526								13:32	14:11

TER 894208	09:56		10:50						
TER 836385			11:00						13:29
TER 894521	11:40	11:52							
TGV 9583		12:03		12:55		13:43			
TER 835020						14:18			15:44
TER 894563	13:38	13:55							
TGV 5516		14:11		15:01		16:00			17:30
TER 894031	15:11				16:24				
TER 894624					17:05		18:34		
TER 834026							18:43		19:40
TER 894226	18:56		19:50						
TER 836389			20:05						22:31

8.0.5. Samedi

Train	besan-con_viotte	besan-con_tgv	dijon	mul-house	bel-fort_ville	stras-bourg	epinal	metz	nancy
TER 894517	10:12	10:27							
TGV 9898		10:34		11:23		12:27		13:24	
TER 88526								13:32	14:11

Contenu masqué

TER 894208	09:56		10:50						
TER 836385			11:00						13:29
TER 894521	11:40	11:52							
TGV 9583		12:03		12:55		13:43			
TER 835034						15:19			16:44
TER 894563	13:38	13:55							
TGV 5516		14:11		15:01		16:00			17:16
TER 894535	17:35	17:48							
TGV 5500		18:23		19:12		20:23		21:38	
TER 88616								22:34	23:11
TER 894559	19:28	19:41							
TGV 9896		20:15		21:03		22:00		22:57	
BUS 35441								23:39	01:16

8.0.6. Dimanche

Train	besan-con_viotte	besan-con_tgv	dijon	mul-house	bel-fort_ville	stras-bourg	epinal	metz	nancy
TER 894521	11:40	11:52							
TGV 9583		12:03		12:55		13:43			

Contenu masqué

TER 894569	12:23	12:36							
TGV 6705		13:31		14:17					
TER 832326				14:34		15:39			
TGV 2588						15:50			17:12
TER 839174						16:19			17:45
TER 894563	13:38	13:55							
TGV 5516		14:11		15:01		16:00			17:33
TER 894264	18:12		19:15						
TGV 6764	18:36	18:52	19:22						
TER 836389			20:05						22:31
TER 894559	19:28	19:41							
TGV 9896		20:15		21:03		22:00		22:57	
BUS 35441								23:39	01:16

Retourner au texte.