

Visualisation Python: design, matplotlib et open source

Dataviz Universe - 2024



Barbier-Darnal Joseph
Master IREF - ERDS
Note technique - Stage

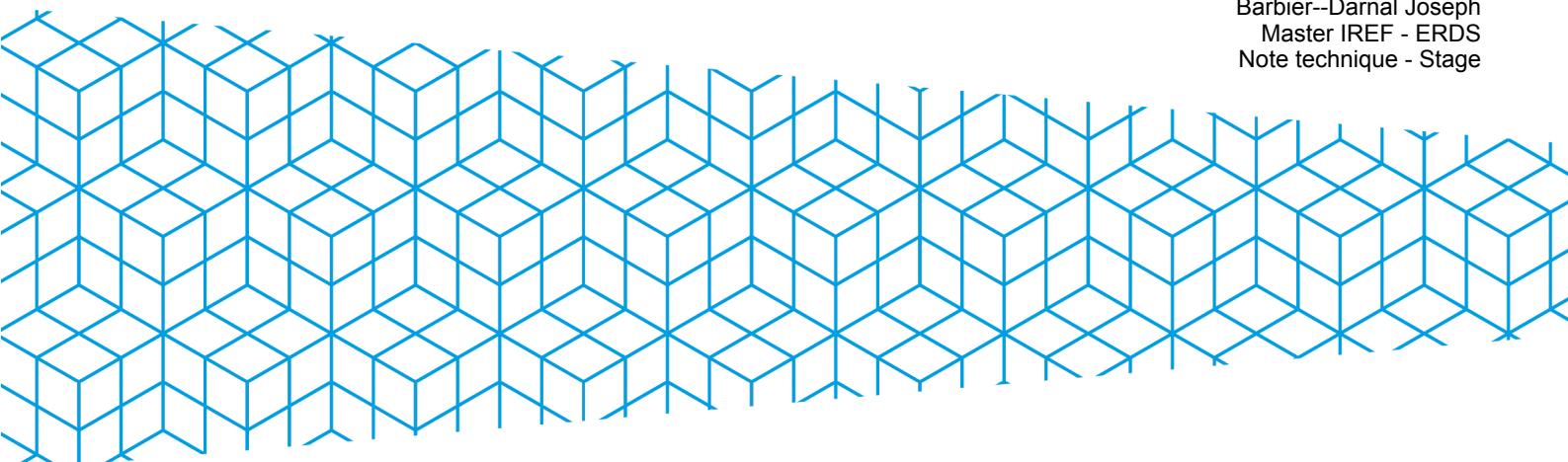


Table des matières

Partie I. Visualisation de données via Python	7
I/ Contexte	7
II/ De la donnée brute à la donnée traitée	7
III/ De la donnée traitée à l'histoire	7
Comprendre les données	8
Trouver une histoire	8
III/ De l'histoire au design	9
Choix du graphique	9
IV/ Du design au code	10
V/ Vue d'ensemble de graphiques.....	11
Partie II. Développement d'outils open source pour Matplotlib	12
I/ Matplotlib et ses extensions	12
II/ PyPalettes	12
Paletteer	12
Création de PyPalettes.....	13
Publication et communication.....	14
III/ PyFonts	14
Gestion des polices	14
Création de PyFonts.....	15
Futur du projet	15
IV/ DrawArrow	15
Tracer des flèches avec matplotlib	16
Création de DrawArrow	16
Futur du projet	17
Conclusion.....	18

Remerciements

J'aimerais exprimer ma gratitude à Yan, sans qui cette expérience n'aurait pas été possible. Notre collaboration a débuté durant l'été 2023, alors qu'il cherchait quelqu'un pour contribuer à son projet Python Graph Gallery. Pendant six mois, nous avons travaillé ensemble quelques heures par semaine.

Au fil de nos échanges, Yan m'a proposé de le rejoindre à temps plein pour mon stage, une opportunité qui m'a amené à m'installer à Montpellier. Je me considère très chanceux de pouvoir travailler à ses côtés. Grâce à lui, ma vision de beaucoup de choses a changé, notamment dans la façon de résoudre les problèmes, qu'ils soient techniques ou non. Je n'aurais jamais pensé faire autant de progrès et décupler mon plaisir au travail.

Pour toutes ces raisons, je tiens à remercier sincèrement Yan pour m'avoir permis de travailler avec lui, pour son honnêteté intellectuelle et pour avoir partagé avec moi sa passion pour la visualisation de données.

Introduction

Ce rapport de stage de fin d'études relate mon expérience professionnelle au sein de Dataviz Universe, un ensemble de plateformes de référence dédiées à la visualisation de données et à la science des données. Durant une période de six mois, j'ai eu l'opportunité d'intégrer cette entreprise située à Montpellier, en France, sous la direction de Yan Holtz, un expert reconnu dans le domaine. En tant qu'étudiant en Master 2 de mathématiques appliquées à l'Université de Bordeaux, ce stage représentait une étape importante de mon parcours académique, offrant un terrain idéal pour appliquer les concepts théoriques acquis et pour enrichir mes compétences techniques et analytiques.

L'immersion dans l'environnement de Dataviz Universe m'a permis de participer activement à divers projets innovants, renforçant non seulement mes capacités techniques mais aussi ma compréhension des concepts complexes liés à la visualisation de données. Le stage a été une occasion unique de collaborer avec des professionnels de haut niveau, de développer des solutions concrètes répondant aux besoins de l'entreprise et de contribuer au rayonnement d'une plateforme utilisée par une communauté mondiale de data scientists.

La richesse de l'expérience acquise au cours de ces six mois dépasse largement le simple cadre de la formation académique. J'ai pu explorer en profondeur des outils de visualisation avancés, participer au développement de nouvelles bibliothèques open source, et m'impliquer dans des projets de recherche appliquée en collaboration avec des chercheurs du CNRS. Ce rapport vise à détailler les différentes facettes de cette expérience, les objectifs fixés et les réalisations accomplies, tout en mettant en lumière les enseignements personnels et professionnels qui en découlent.

Contexte: à propos de Dataviz Universe

Dataviz Universe rassemble des sites créés par Yan Holtz en lien avec la visualisation de données et la science des données en général. Ces sites comprennent :

- > **R/Python/React/D3 Graph gallery** : 4 sites comprenant un large éventail de tutoriels sur la façon de créer n'importe quel type de graphique dans différents langages de programmation.
- > **Data to Viz** : un arbre de décision qui aide à définir quel type de graphique créer en fonction du type de données, avec des explications sur chaque type, les avantages et les inconvénients de chacun, et des cas d'utilisation spécifiques basés sur des concepts de visualisation de données.
- > **Dataviz Inspiration** : une collection des meilleures visualisations disponibles sur Internet, classées par type de graphique, format, thème et avec une description du projet à l'origine de la visualisation.
- > **Productive R Workflow** : un cours en ligne consacré à l'amélioration de la productivité lors de l'utilisation de R, qui met l'accent sur la manière d'écrire un code de meilleur qualité, de créer un site web personnalisé pour ses rapports avec Quarto, de versionner son code avec Git et Github, et d'autres sujets connexes.

Ensemble, ces sites attirent plus d'un million de visiteurs par mois. Ce type de plateforme réduit le gap entre une question d'entreprise ou de client et l'aspect technique, en permettant à quiconque d'accéder rapidement et aux aspects techniques et aux conceptuels des outils de visualisation de données.

Objectifs

Les objectifs de ce stage étaient nombreux et variés, et peuvent être énumérés comme suit.

- 1. créer des exemples de graphiques avancés avec Matplotlib
- 2. amélioration générale et ajout de contenu aux Gallery Python et R
- 3. développement d'un package R avec 2 chercheurs du CNRS sur la modélisation de l'histoire de vie
- 4. création de bibliothèques Python open source pour la visualisation de données
- 5. autres tâches mineures

Ce document ne détaille et n'explore que les objectifs 1 et 4.

Plan du document

Visualisation de données via Python

La première partie de ce document traite de la visualisation de données à l'aide de Python, en particulier Matplotlib. Nous examinerons le contexte de ces visualisations, ce que j'ai fait pour les créer et ce que j'en ai retiré d'un point de vue personnel.

Développement d'outils open source pour matplotlib

La deuxième partie de ce document se concentrera sur les outils open source développés pendant le stage. En effet, 3 bibliothèques ont été créées durant cette période, dont le but était de simplifier certains aspects du processus de création de visualisation avec Matplotlib. Ces outils traitent respectivement des couleurs, des polices et des flèches.

Partie I. Visualisation de données via Python

I/ Contexte

Toutes les visualisations que j'ai produites étaient destinées à un article sur le site web Python Graph Gallery. L'un des objectifs de Yan était que ce site contienne un grand nombre de visualisations avancées réalisées en Python. Bien que des millions de personnes créent des graphiques en Python, peu d'entre elles essaient malheureusement de les rendre attrayants.

Pour me challenger sur la création de graphiques, Yan m'a conseillé de participer au défi TidyTuesday. Cet événement, organisé par la Data Science Learning Community, présente chaque semaine un nouveau jeu de données, dans le but de permettre à tous ceux qui le souhaitent de créer un graphique, une application web ou tout autre projet de science des données, et de le partager avec d'autres. Cet événement existe depuis 6 ans maintenant, et chaque semaine rassemble un grand nombre de participants. C'était donc pour moi un bon moyen de me lancer dans l'exercice de la réalisation de graphiques dont le but est d'être partagés.

II/ De la donnée brute à la donnée traitée

La première chose à faire dans ce type d'exercice est de rendre les données utilisables. Les données proposées pour TidyTuesday peuvent porter sur n'importe quel sujet et provenir de n'importe quelle source. Il est donc fréquent que les formats ne soient pas standardisés, que les unités changent ou, plus généralement, qu'un travail de nettoyage soit nécessaire au préalable.

Cependant, cette étape est rarement complexe, car aujourd'hui des bibliothèques telles que pandas ou polars permettent d'effectuer 99% des tâches requises de manière simple et rapide. pandas a été mon outil de prédilection.

III/ De la donnée traitée à l'histoire

Une fois les données exploitables, vient l'étape la plus complexe : trouver le message que je veux faire passer.

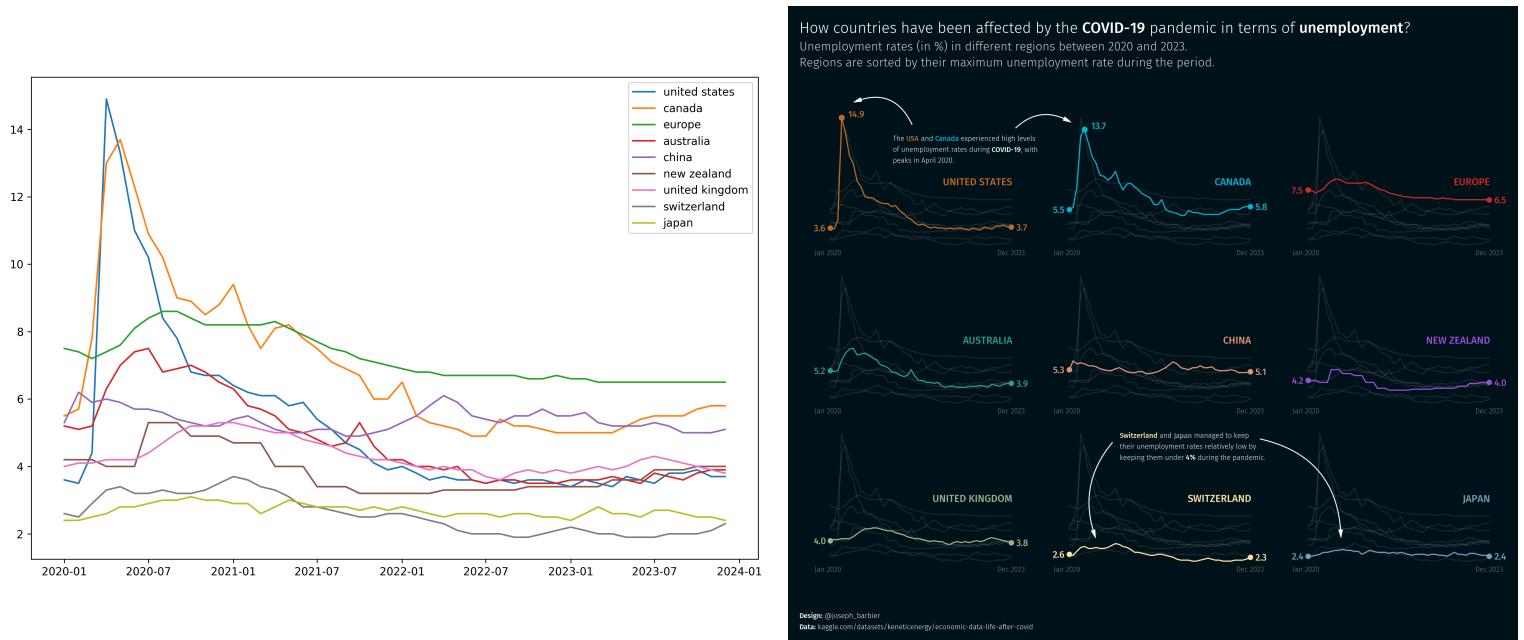
Comprendre les données

Le premier TidyTuesday que j'ai fait concernait le Fiscal Sponsor Directory, un outil qui permet à des associations/projets de recevoir des fonds ou des dons déductibles d'impôts. Cela sert principalement à des mettre des entités qui recherchent des aides fiscales et administratifs avec des entités qui cherchent à soutenir des projets.

Cet exemple illustre un point important : il est complexe d'explorer des données et d'en tirer des conclusions sans compréhension du sujet de fond et des enjeux qui l'accompagnent. Un travail important fut donc de comprendre de quoi étaient les données étaient faites, comment elles sont organisées par rapport au sujet, etc.

Trouver une histoire

Maintenant que le contexte des données est clair, je dois déterminer ce que je veux dire avec mon graphique. Pour que mon graphique ait un impact, il doit contenir un message clair, rapide et facile à comprendre. L'information seule est loin d'être suffisante.



Comparaison d'un graphique "par défaut" et d'un graphique avec un message clair. Ces deux graphiques représentent exactement la même information et utilisent les mêmes données.

Dans les données des graphiques ci-dessus, le message était facile à déterminer parce que le pattern est très clair. Quelques calculs et graphiques exploratoires permettent de se faire une idée des différences de taux de chômage entre les régions étudiées.

Cependant, cette étape est généralement beaucoup plus complexe, et ce pour plusieurs raisons :

- le sujet est spécifique ou de niche
- il n'y a pas de garantie que les données aient un pattern quelconque
- même s'il existe un pattern, il peut être :
 - hasardeux
 - trop spécifique et donc peu intéressant à promouvoir
 - trop difficile à rendre compréhensible

Pour toutes ces raisons, cette étape est la plus difficile et la plus longue, mais aussi la plus importante, car elle déterminera la suite des événements. Nous verrons qu'il est beaucoup plus facile de concevoir un graphique si l'on sait déjà quel message on veut faire passer.

III/ De l'histoire au design

Une fois que j'ai une idée plus ou moins concrète du message que je veux transmettre, je dois réfléchir à la manière dont je vais présenter ces informations.

Choix du graphique

Même si je sais quelle information je veux donner, cela ne me dit rien sur le type de graphique à utiliser. Le même type de graphique peut être utilisé dans des contextes très différents.

Si mon graphique porte sur une évolution dans le temps, je peux utiliser :

- un line chart
- un area chart
- un ridgeline plot
- un bar chart
- une heatmap
- un lollipop plot
- un waffle chart
- ou tout autre graphique au format vidéo

C'est pourquoi mon approche consiste généralement à procéder à plusieurs itérations, en passant à chaque fois un peu de temps avec Yan pour obtenir un feedback sur les directions à explorer ou à éviter.

Dans cette boucle de feedback, l'un des grands avantages est que Yan ne s'intéresse pas à la façon dont le code fonctionne ou à sa complexité. Je veux dire par là que Yan imposera des contraintes sur le design qu'il souhaite voir, me forçant à trouver un moyen technique de le respecter. Je pense que c'est l'une des choses qui m'a fait le plus progresser dans ma compréhension de Matplotlib parce que j'ai été forcé de trouver une solution pour réaliser une certaine chose, et donc d'explorer la documentation et la bibliothèque en général.

L'avantage d'utiliser Matplotlib est que l'on se rend vite compte que tout est possible si l'on prend le temps de chercher.

IV/ Du design au code

Lorsque j'ai réalisé mes premiers graphiques, j'avais des compétences techniques limitées : j'avais une connaissance superficielle de Matplotlib, une intuition assez faible de la façon de faire les choses et je passais beaucoup de temps sur les détails.

Par exemple, un jour Yan m'a suggéré d'ajouter des flèches avec des points d'inflexion pour servir de légende à un graphique. Je n'avais aucune idée de la manière de procéder et c'était effectivement assez complexe.

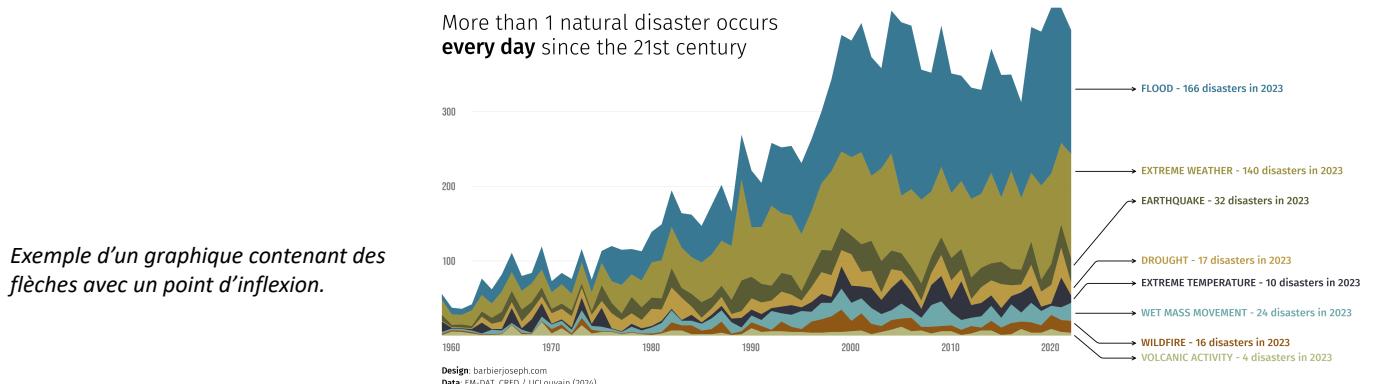
```
def arrow_inflexion(
    ax,
    start, end,
    angleA, angleB,
    radius=0,
    color="black",
    transform=None
):
    # get the coordinates
    x2, y2 = start
    x1, y1 = end

    # avoid division by zero
    epsilon = 1e-6
    if x2 == x1:
        x2 += epsilon
    if y2 == y1:
        y2 += epsilon

    # select right coordinates
    if transform is None:
        transform = ax.transData

    # add the arrow
    connectionstyle = f"angle,angleA={angleA},angleB={angleB},rad={radius}"
    ax.annotate(
        "",
        xy=(x1, y1), xycoords=transform,
        xytext=(x2, y2), textcoords=transform,
        arrowprops=dict(
            color=color, arrowstyle="->",
            shrinkA=5, shrinkB=5,
            patchA=None, patchB=None,
            connectionstyle=connectionstyle,
        ),
    )
```

Une des manières les plus simples de tracer une flèche avec un point d'inflexion avec Matplotlib



Exemple d'un graphique contenant des flèches avec un point d'inflexion.

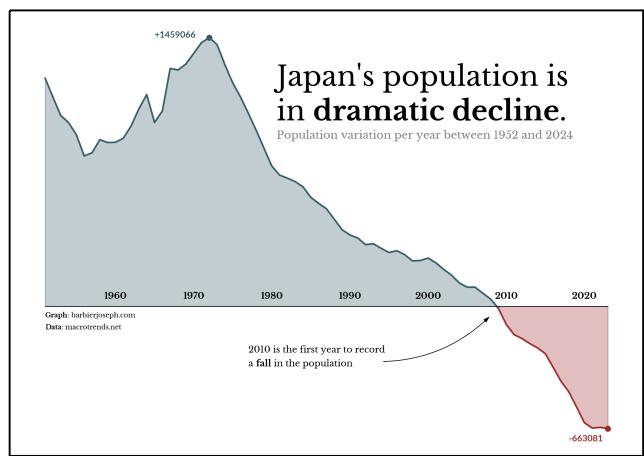
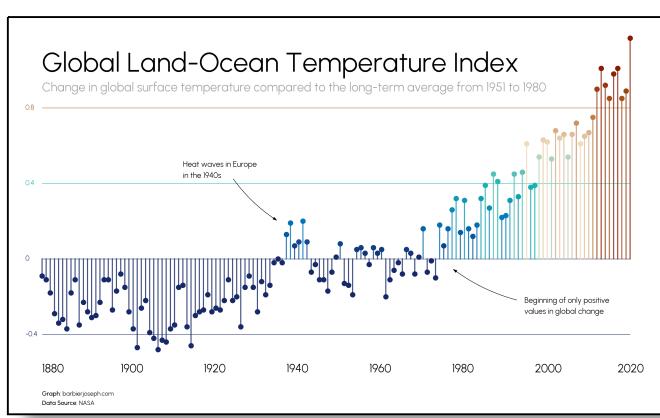
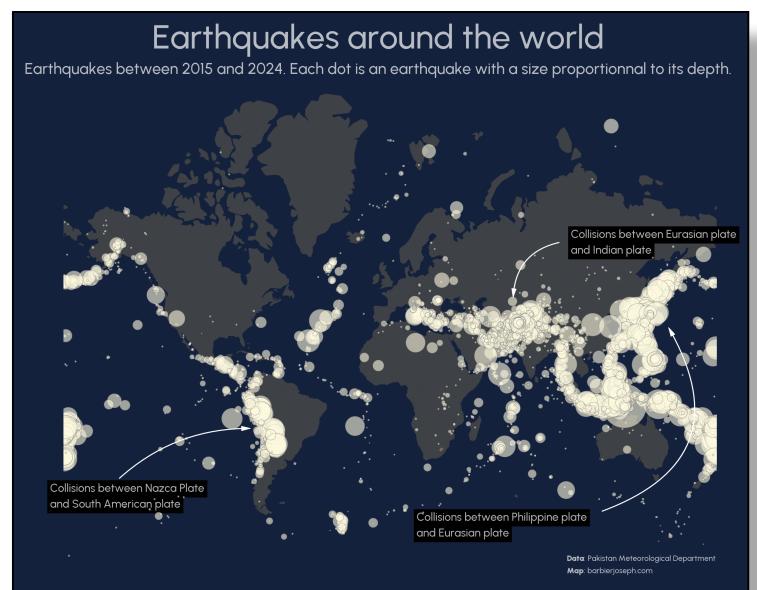
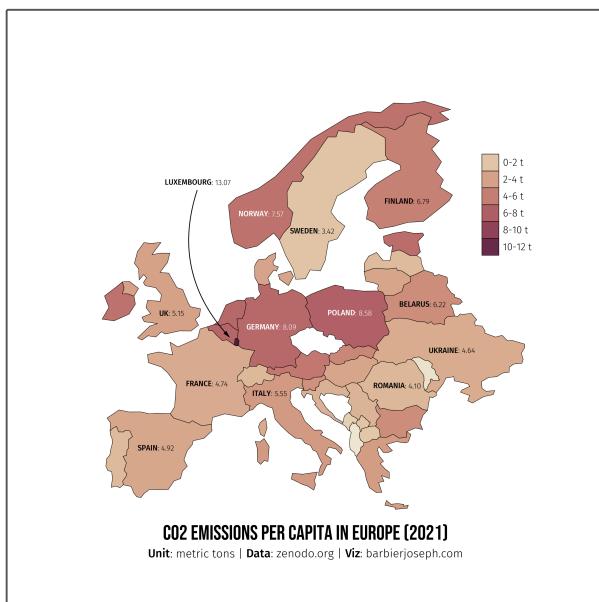
Tracer ce type de flèche nécessite une certaine compréhension de comment Matplotlib est construit et fonctionne en arrière plan, notamment des notions de transformation et de patch. Nous verrons d'ailleurs par la suite un outil que nous avons créé pour remédier à ce problème.

Cet exemple n'est qu'un cas parmi un grand nombre d'autres, où d'autres types d'éléments ou de customisation sont en fait peu intuitifs à créer. Fort heureusement, Matplotlib regorge de beaucoup de moyens pour simplifier ce travail.

Étant donné que Yan souhaitait créer une newsletter sur les meilleures astuces Matplotlib, il m'a demandé de répertorier entre 5 et 10 astuces qui permettent facilement d'améliorer un graphique fait avec Matplotlib. Nous avons donc écrit ensemble cette newsletter qui compte aujourd'hui plusieurs milliers de personnes à travers le monde.

V/ Vue d'ensemble de graphiques

Ci-dessous est joint une liste non-exhaustive des graphiques que j'ai pu réalisé durant ce stage.



Partie II. Développement d'outils open source pour Matplotlib

I/ Matplotlib et ses extensions

Matplotlib est la plus célèbre bibliothèque dédiée à la visualisation de données en Python. Crée il y a plus de 20 ans, elle est aujourd'hui largement utilisée et s'accompagne d'un grand nombre d'extensions.

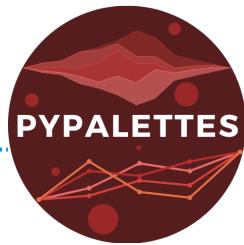
Bien que Matplotlib permette de créer des graphiques en relativement peu de lignes de code, elle sert également de boîte à outils pour de nombreuses autres bibliothèques écrites par-dessus d'elle. Les plus connues sont Seaborn, une bibliothèque qui reprend le cœur de Matplotlib mais fournit un accès rapide et intuitif à des graphiques complexes, et Plotnine, une bibliothèque qui implémente la grammaire graphique (principalement connue de ggplot2 en R) uniquement à partir de Matplotlib pour une utilisation en Python.

Cependant, ces deux exemples ne représentent qu'une infime partie de tout ce que la communauté open source de Python a créé autour de matplotlib. En fait, la production d'un graphique d'une qualité suffisante pour être publié nécessite une grande quantité de code et une certaine complexité si l'on utilise uniquement Matplotlib.

Même après plus de 20 ans d'existence, Matplotlib a toujours besoin de ses extensions et des personnes qui les créent. En créant des graphiques avec Matplotlib, j'ai remarqué qu'une grande partie de mon code était redondante, facile à généraliser à des cas d'utilisation plus larges et complexe à documenter. Pour cette raison, Yan et moi avons créé 3 extensions (principalement mais pas exclusivement) pour Matplotlib, qui simplifient le processus de création de graphiques.

II/ PyPalettes

Paletteer



Dans le langage R, la bibliothèque la plus populaire pour gérer les couleurs dans les graphiques est Paletteer, une agrégation de palettes provenant de la plupart des paquets R

dédiés aux couleurs. Cette agrégation donne accès à plus de 2500 palettes de couleurs pré-désignées, accessibles via une API simple d'utilisation.

L'idée était de trouver un moyen de rendre ces palettes disponibles dans une interface Python. Etant donné que les palettes en question sont sous licence libre, j'ai écrit un script Python qui scrappe dans la documentation de Paletteer toutes les valeurs hexadécimales de chaque couleur et le nom de chaque palette pour créer un fichier avec toutes les palettes en question.

Création de PyPalettes

J'ai ensuite ajouté toutes les palettes de couleurs déjà présentes dans Matplotlib et Seaborn pour rendre l'outil le plus exhaustif possible. J'ai ensuite créé une API aussi simple que possible pour les utilisateurs, avec une fonction principale : `load_cmap()`. Cette fonction prend un nom de palette et quelques autres arguments supplémentaires (pour des cas d'utilisation plus spécifiques) et renvoie un objet matplotlib dédiés aux palettes : une colormap.

Dans Matplotlib, les colormaps sont des objets similaires à des listes de couleurs, avec des propriétés et des fonctions spécifiques. Par exemple, elles ont l'avantage d'être détectées et "mappées" automatiquement lors de la création d'une légende ou de n'importe quel graphique nécessitant de mettre en correspondance une valeur et une couleur (tel que les cartes choroplèthes).

Nous avons procédé à plusieurs itérations avant d'arriver à un résultat satisfaisant, la complexité résidant dans le fait que l'utilisateur n'a pas à se préoccuper du fonctionnement en arrière-plan. PyPalettes est maintenant opérationnel.

```
import matplotlib.pyplot as plt
from pypalettes import load_cmap
import random

cmap = load_cmap('FridaKahlo', cmap_type='continuous')
data = [[random.random() for _ in range(12)] for _ in range(10)]

fig, ax = plt.subplots(dpi=300)
ax.imshow(data, cmap=cmap)
plt.show()
```

Exemple d'une heatmap avec la colormap "FridaKahlo"

Note : PyPalettes possède également d'autres fonctions, notamment pour récupérer des palettes dans d'autres formats que les colormaps décrites ci-dessus. En particulier, il est possible de les récupérer sous la forme d'une liste de chaînes de caractères avec les valeurs hexadécimales ou via une liste de valeurs RGB.

Publication et communication

Pour communiquer sur cet outil le plus efficacement possible, Yan a créé une application web qui permet de choisir une palette parmi toutes celles existantes et de voir automatiquement et à quoi elle ressemble sur plusieurs types de graphiques Python. L'application permet de visualiser très rapidement un grand nombre de palettes en simulant le style des graphiques matplotlib à l'aide d'outils JavaScript, accessibles via une simple url dans un navigateur dans la Gallery Python.

Nous avons beaucoup communiqué sur LinkedIn et Twitter, ainsi que sur divers blogs, pour attirer les gens vers le projet. Cela a donné lieu à plusieurs articles de blogs, de notre part ou non, une vidéo Lyndon Walker sur Youtube, et plusieurs milliers de likes en combiné sur les réseaux sociaux de Yan. Aujourd'hui, PyPalettes est installé entre 15 et 25 fois par jour et compte environ 200 favoris sur le projet Github associé contenant le code source.

L'étape finale a été de publier cette bibliothèque sur PyPi, le gestionnaire de paquets Python le plus populaire, et est donc installable via un simple:

```
pip install pypalettes
```

III/ PyFonts

Gestion des polices



La gestion des polices de caractères est étonnamment complexe et, par conséquent, la gestion des polices de caractères dans les logiciels en est héritée. Bien que Matplotlib fournit nativement un ensemble de polices, leur nombre est très limité. Cependant, Matplotlib donne surtout accès à un gestionnaire de polices, qui permet d'utiliser n'importe quelle police à condition d'avoir un accès local aux fichiers binaires associés à la police désirée.

Les problèmes posés par ce type de fonctionnement sont les suivants :

- nécessite le téléchargement de la police en amont sur l'ordinateur
- rend le code non reproduitible en :
 - nécessitant d'écrire en clair le chemin d'accès à un dossier spécifique à l'utilisateur
 - exigeant que tout autre utilisateur du code dispose des fichiers de police

```
from matplotlib.font_manager import FontProperties  
personal_path = '/Users/josephbarbier/Library/Fonts/'
```

```
font_path = personal_path + 'FiraSans-Regular.ttf'  
font = FontProperties(fname=font_path)
```

Comment charger une police avec matplotlib

Création de PyFonts

J'ai donc cherché un moyen d'accéder à n'importe quelle police sans avoir à installer quoi que ce soit. Etant donné que l'écrasante majorité des polices sous licence libre sont listées dans des projets Github, et que Github permet d'accéder facilement à une url vers la version « brute » d'un fichier binaire, j'ai entrepris de créer un moyen de récupérer des fichiers à partir de leur source même.

La première version actuelle de PyFonts donne accès à une fonction `load_font()`, qui, à partir d'une url vers n'importe quelle police, crée un objet `FontProperties` issu du gestionnaire de polices de Matplotlib. En pratique, la fonction crée un fichier temporaire local du fichier binaire, l'utilisera pour initialiser l'objet `FontProperties` et supprime ensuite le fichier. De cette manière, le code est court (entre 3 et 4 lignes avec les outils traditionnels de Matplotlib contre seulement 1 ligne avec PyFonts), simple (appel d'une fonction avec une url comme argument) et indépendant de l'ordinateur (chargement depuis le net).

```
from pyfonts import load_font  
  
font = load_font(  
    font_url="https://github.com/google/fonts/raw/main/apache/ultra/Ultra-Regular.ttf"  
)
```

Exemple d'utilisation de PyFonts avec la police Ultra en "regular"

Futur du projet

Les prochaines étapes consistent à simplifier encore plus le processus en créant un algorithme qui recherchera automatiquement dans Google Font (la plus grande base de données de polices au monde) la police en question simplement par son nom. Cette étape est encore en cours de développement et nécessite une certaine quantité de travail, mais PyFonts est maintenant disponible sur PyPi et est pleinement fonctionnel avec les fonctionnalités décrites précédemment. Il est installable via :

```
pip install pyfonts
```

IV/ DrawArrow



Tracer des flèches avec matplotlib

Matplotlib offre une large gamme d'outils pour créer des flèches de toutes sortes : droites, courbes, avec un ou plusieurs points d'inflexion, simples ou doubles, avec certaines formes ou patterns, etc. Le problème de ces outils réside principalement dans la conception de leur API.

Les méthodes existantes pour ajouter une flèche dans matplotlib sont radicalement différentes selon le type de flèche, ce qui rend l'API peu intuitive et inconsistante.

```
import matplotlib.pyplot as plt
from matplotlib.patches import FancyArrowPatch
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(15,5))

ax1.annotate('', xy=(1, 1), xytext=(0, 0), arrowprops=dict(arrowstyle='->'))
ax2.arrow(0, 0, 1, 1, head_width=0.05, head_length=0.1)
arrow = FancyArrowPatch((0, 0), (1, 1), arrowstyle='->')
ax3.add_patch(arrow)

plt.show()
```

3 manières différentes de créer une flèche via Matplotlib

Même avec des cas d'utilisation minimalistes, le code n'est pas facile à comprendre, et surtout, il n'est pas consistant. De plus, la complexité et la taille du code augmentent très rapidement pour des exemples plus complexes tels que des flèches avec des points d'inflexion, ce qui peut conduire à ce qu'une grande partie du code total d'un graphique soit dédiée aux seules flèches.

Création de DrawArrow

J'ai donc commencé à chercher un moyen d'unifier les fonctions présentes dans matplotlib autour d'une API et d'une syntaxe uniques.

L'un des premiers problèmes à résoudre a été de déterminer dans quel système de coordonnées la position des flèches serait définie : entre 0 et 1 ? Le système de données ? En pixels ? Autres ? Ma solution réside dans la création de 2 fonctions, qui respectent la consistance de l'API orientée objet de Matplotlib : une fonction qui utilise les positions des données du graphique spécifié (ou `Axes`) et une fonction qui utilise les positions relatives du graphique « global » (ou `Figure`), c'est-à-dire entre 0 et 1 sur les axes x et y. Ce principe est également présent dans la bibliothèque `highlight_text`, qui facilite la mise en forme des annotations dans Matplotlib.

Le second et principal problème était de réussir à écrire un code permettant de créer n'importe quel type de flèche tout en permettant à l'utilisateur de n'utiliser que les 2 fonctions mentionnées ci-dessus. Pour la première version de DrawArrow, j'ai décidé de me concentrer sur un nombre limité de fonctions, mais avec un code clair, modifiable et testable. J'ai donc créé une API qui n'utilise que le `FancyArrowPatch` vu ci-dessus, car il permet un haut niveau de personnalisation ainsi que la création de flèches très simples.

```
import matplotlib.pyplot as plt
from drawarrow import fig_arrow

fig, ax = plt.subplots(dpi=300)
fig_arrow(
    tail_position=(0.3, 0.3),
    head_position=(0.8, 0.8),
    color="#2a9d8f",
    tail_width=2,
    head_length=20,
    head_width=10,
    linewidth=2,
    radius=0.7,
    fig=fig
)
plt.show()
```

Exemple de création d'une flèche avec drawarrow

La syntaxe de drawarrow fournit un outil avec des arguments rapidement compréhensibles et une syntaxe qui respecte les principes de Matplotlib. En utilisant drawarrow, vous pouvez réduire considérablement la taille et la complexité de votre code tout en conservant toutes les fonctionnalités de Matplotlib.

Futur du projet

Drawarrow est maintenant entièrement fonctionnel et disponible sur PyPi. Cependant, cette bibliothèque ne permet pas de dessiner des flèches avec des points d'infexion, ainsi que d'autres types de flèches plus spécifiques. Cela nécessite une utilisation très différente de l'API de Matplotlib (voir précédemment) et est donc difficile à unifier avec les fonctions existantes de Drawarrow. Ceci est donc prévu et constitue l'un des prochains développements du projet.

Aujourd'hui DrawArrow est utilisable après installation :

```
pip install drawarrow
```



Université
de BORDEAUX