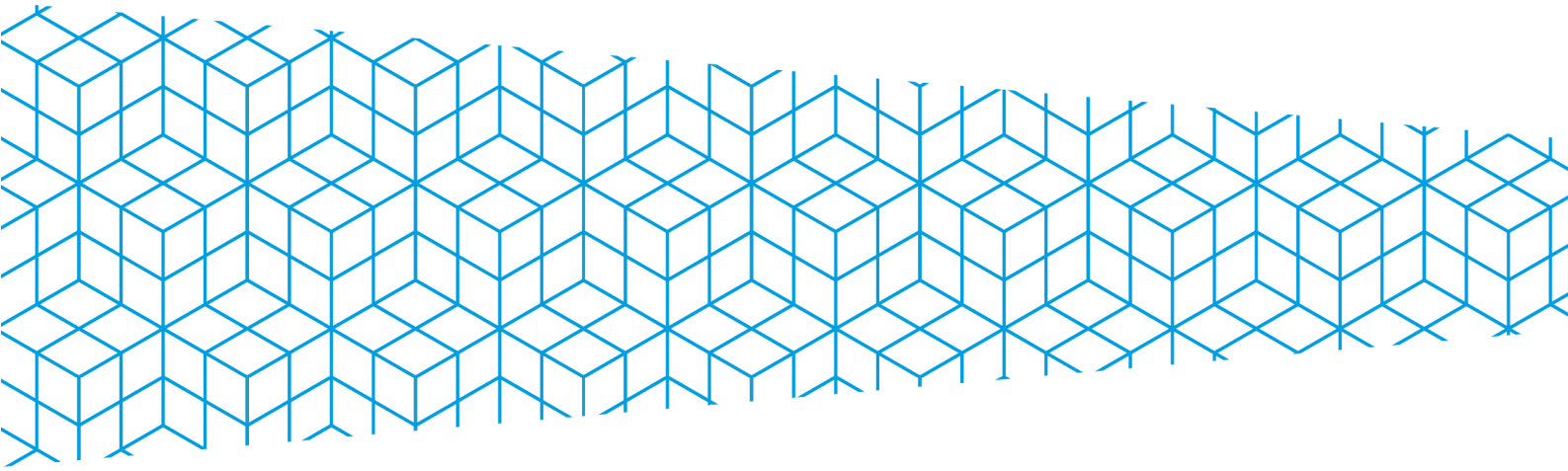


université de BORDEAUX

Table des matières ou plan

Complément de texte
Texte complémentaire

Visualisation Python: design, matplotlib et open source

Dataviz Universe - 2024

Introduction

Texte courant corps 11. Ro vellessed erum dolupienis nime et optatiisit quis pro inctusa ndistium eatempos accus incitas dia de vitatis sinum quid que litam voluptaecum laceped ea nimi, sit, sequam, ipicias utemo mincia quo mi, ea eum eniment ullorit earum sum nost, cum quis poria quo veruntisi doluptatur, ommostrupta secumque conecto modist, oditet dolo et verum ipsanda eceate doluptatur, te vero quatur as rempos aspid eum harchic iaersped que preperiberum auda si omnihillessi volo cupitatus ex etus senti alitia con re vendem quid eat pa veles nonsere none rem simillа dolecto volum quiscias eos archici enihicimet est, offic tem rectatem et, occus essima vel maior a nis auditatia exerum assunt, quo doluptatem ad moluptiore voluptatum dolenis qui cus aut reruptint, simi, sint.

Contexte

Texte courant en corps 11. Ro vellessed erum dolupienis nime et optatiisit quis pro inctusa ndistium[1] eatempos accus incitas dia de vitatis sinum quid que litam voluptaecum laceped ea quis poria quo veruntisi doluptatur, ommostrupta secumque conecto modist, oditet dolo et.

Tem rectatem et, occus essima vel maior a nis auditatia exerum assunt, quo doluptatem ad moluptiore voluptatum dolenis qui cus aut reruptint, simi, sint :

- Optis iditiberum aut venihit aspelit estem ;
- La doluptatia sunt ipsum nonsequundel mos architem nonsequo ;
- Dupide destotatur soluptat voluptas parum fugiae mint late nis doluptumque eostiosam nis auta deles simet experehent ;

Objectifs

Texte courant en corps 11. Ro vellessed erum dolupienis nime et optatiisit quis pro inctusa ndistium[2] eatempos accus incitas dia de vitatis sinum quid que litam voluptaecum laceped ea quis poria quo veruntisi doluptatur, ommostrupta secumque conecto modist, oditet dolo et.

Tem rectatem et, occus essima vel maior a nis auditatia exerum assunt, quo doluptatem ad moluptiore voluptatum dolenis qui cus aut reruptint, simi, sint

[1] Loi, référence etc.

[2] Loi, référence etc.

Plan du document (si nécessaire)

Texte courant corps 11. Ro vellessed erum dolupienis nime et optatiisit quis pro inctusa ndistium eatempos accus incitas dia de vitatis sinum quid que litam voluptaecum laceped ea nimi, sit, sequam, ipicias utemo mincia quo mi, ea eum eniment ullorit earum sum nost, cum quis poria quo veruntisi doluptatur, ommostrupta secumque conecto modist, oditet dolo et verum ipsanda eceate doluptatur, te vero quatur as rempos aspid eum harchic iaersped que preperiberum auda si omnihillessi volo cupitatus ex etus senti alitia con re vendem quid eat pa veles nonsere none rem simillа dolecto volum quiscias eos archici enihicimet est, offic tem rectatem et, occus essima vel maior a nis auditatia exerum assunt, quo doluptatem ad moluptiore voluptatum dolenis qui cus aut reruptint, simi, sint.

Partie I. Optis iditiberum aut venihit aspelit estem a doluptatia sunt ipsum nonsequundel mos architem nonsequo cupide destotatur soluptat voluptas parum fugiae mint late nis doluptumque eostiosam nis auta deles simet experehent.

Partie II. Archite nderit, sus et quaturem qui antiusc ienienis dolupta cum alicia secearciae premolorpos nossin nihicil igenti autae vitistis nam earuntio int officia dolorae modi volorent. Aqui des dolorati optio veni tem fugitet ut landipsus expeliqui quatias aut officie nditas atiur, nest autem aut del iumende aliquid itiore quatet audaepelis si dolupta dolest, sandiste nimenti busamus, sum et molesequi ilicima gnatqui beation eum dolore, est eumqui.

Partie I. Visualisation de données via Python

I/ Contexte

L'entièreté des visualisations que j'ai réalisées avaient pour finalité un article sur le site Python Graph Gallery. Un des objectifs de Yan était que ce site ait un large nombre de visualisations complexes réalisées en Python. En effet, même si des millions de personnes font des graphiques en Python, rares sont ceux qui essaient de les rendre attractifs.

Pour me challenger sur la création des graphiques, Yan m'a conseillé de faire le TidyTuesday challenge. Cet évènement géré par Data Science Learning Community propose chaque semaine un nouveau dataset avec comme objet que quiconque le souhaite fasse un graphique, une application web ou n'importe quel autre projet de data science, et le partage aux autres. Cela fait maintenant 6 ans que cet évènement a lieu et regroupe chaque semaine un nombre important de participant. Ce fut donc pour moi un bon moyen de me donner à l'exercice de faire des 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 le TidyTuesday peuvent concerner tous les sujets possibles et venant de n'importe quelle source. Il est donc assez courant que les formats ne soient pas standardisés, les unités changeantes ou plus généralement complexes.

Cela est rarement une étape complexe puisque aujourd'hui des librairies comme pandas ou polars permettent d'effectuer 99% des tâches requises de manière simple et rapide. Pandas fut mon outil de choix.

III/ De la donnée traitée à l'histoire

Une fois les données exploitables, l'étape la plus complexe vient : celle de trouver quel est le message que je souhaite faire passer.

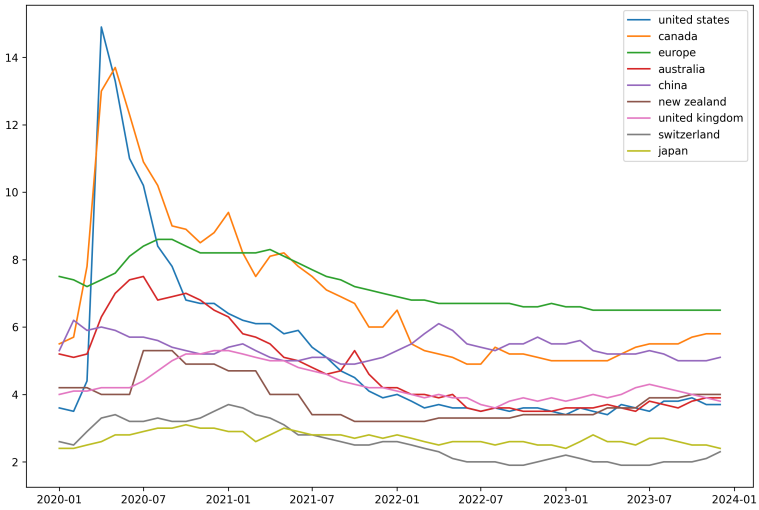
Comprendre les données

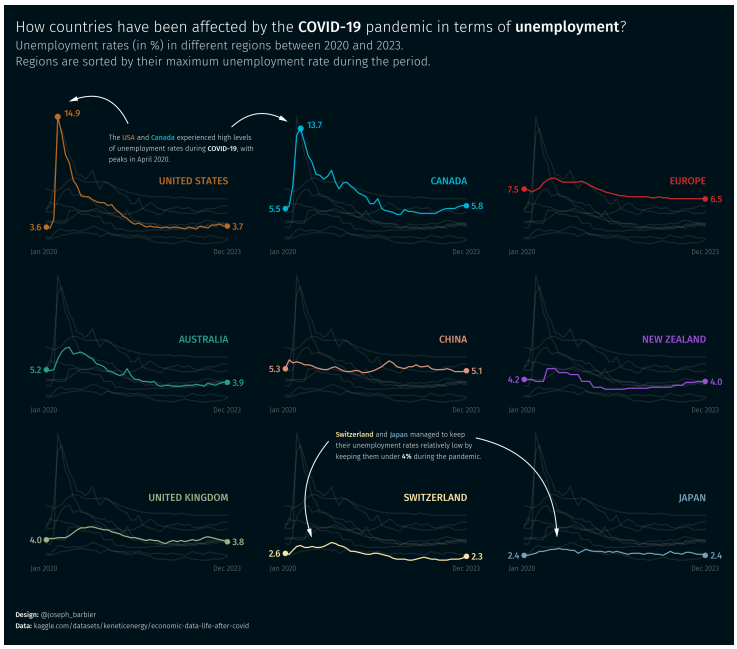
Le premier TidyTuesday que j'ai fait concernait le Fiscal Sponsor Directory, un outil qui permet à des associations/projets de recevoir des fonds ou des dons déductibles d'impôts. Cela sert principalement à des mettre des entités qui recherchent des aides fiscales et administratifs avec des entités qui 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 travaille 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, il faut trouver ce que je souhaite raconter avec mon graphique. En effet, pour que mon graphique ait un quelconque impact il se doit d'avoir 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 fut simple à déterminer car le pattern est très clair. Quelques calculs et graphiques exploratoires permettent de se rendre des comptes des écarts de taux de chômage entre les régions étudiées.

Cependant, cette étape est généralement bien plus complexe, et ce pour plusieurs raisons :
- le sujet de fond est spécifique ou niche
- il n'y a aucune garantie que les données aient un pattern
- même s'il existe un pattern il est peut-être
    - hasardeux
    - trop spécifique et donc peu intéressant à mettre en avant
    - trop difficile à rendre compréhensible

Pour toutes raisons, cette étape est la plus difficile, mais aussi la plus importante car c'est elle qui va déterminer la suite. En effet, nous verrons qu'il est bien plus facile de designer un graphique avec un message clair.

III/ De l'histoire au design

Maintenant que j’ai une idée à peu près concrète du graphique que je veux faire. Avant cette expérience je ne soupçonnais pas à quel point designer une visualisation était important et que cela est complexe.

IV/ Du design au code

Texte courant corps 11. Ro vellessed erum dolupienis nime et optatiisit quis pro inctusa ndistium eatempos accus incitas dia de vitatis sinum quid que litam voluptaecum laceped ea nimi, sit, sequam, ipicias utemo mincia quo mi, ea eum eniment ullorit earum sum nost. Optis iditiberum aut venihit aspelit estem a doluptatia sunt ipsum nonsequundel mos architem nonsequo cupide destotatur soluptat voluptas parum fugiae mint late nis.

Partie II. Développement d'outils open source pour matplotlib

I/ Matplotlib et ses extensions

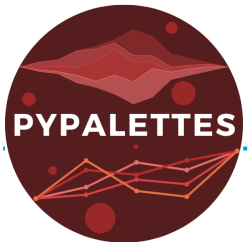
Matplotlib est la plus célèbre librairie dédiée à la visualisation de données en Python. Créée il y a plus de 20 ans, elle possède aujourd'hui une utilisation massive et est accompagnée d'un large nombre d'extensions.

Malgré le fait que Matplotlib permet la création de graphiques en relativement peu de lignes de code, elle sert également de boîte à outil pour de nombreuses autres librairies écrites par dessus elle. Les plus connues sont Seaborn, un librairie qui reprend le coeur de matplotlib mais permettant un accès rapide et intuitif à des graphiques complexes, ou bien Plotnine, une librairie qui implémente uniquement à partir de Matplotlib la grammaire des graphiques (principalement connue via ggplot2) pour un usage en Python.

Ces 2 exemples ne représentent pourtant qu'une infime partie de tout ce que la communauté open source Python a créé autour de matplotlib. En effet, réaliser un graphique avec une qualité publiable dans un média grand publique requiert une quantité importante de code ainsi qu'une certaine complexité, si l'on utilise uniquement Matplotlib.

Même après plus de 20 d'existence, Matplotlib a toujours besoin de ses extensions et des gens qui les créent. A force de créer des graphiques avec matplotlib j'ai remarqué que : beaucoup de mon code est redondant, facile à généraliser à des cas d'usages plus larges et complexe à documenter. Pour cette raison, Yan et moi avons créé 3 extensions pour (principalement mais pas uniquement) Matplotlib, simple d'utilisations et open sources.

II/ PyPalettes



Paletteer

En R, la librairie la plus populaire pour la gestion des couleurs dans un graphique est Paletteer, une agrégation de palettes de la plupart des packages R dédiés aux couleurs. Ce package donne accès à plus de 2500 palettes de couleur pré-faites accessibles via une simple API.

L'idée fut donc de trouver un moyen de rendre ces même palettes accessibles dans une interface Python. Etant donné que les palettes en question sont sous des licences libres, j'ai écris un script Python qui scrappe la documentation de Paletteer avec toutes les valeurs hexadécimales et le nom de chaque palette pour créer un fichier avec toutes les palettes.

Création de PyPalettes

Puis j'ai ajouté toutes les palette de couleurs déjà présentes dans Matplotlib et Seaborn afin d'avoir un outil le plus exhaustif possible. J'ai ensuite créé une API la plus simple possible pour les utilisateurs, avec une principale fonction : `load_cmap()`. Cette fonction prend un nom de palette et quelques autres arguments additionnels (pour des cas d'usages plus spécifiques) et retourne un object matplotlib dédié au palette : une colormap.

Nous avons effectué plusieurs itérations avant d'atteindre un résultat satisfaisant, la complexité résidant dans le fait que l'utilisateur n'est pas à se préoccuper de la complexité en arrière plan. PyPalettes est maintenant créé et fonctionnel.

```
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

Publication et communication

Afin de communiquer au mieux sur cet outil, Yan a créé une application web qui permet de choisir une palette parmi la large liste et de voir automatiquement à quoi elle ressemble sur plusieurs types de graphiques Python. L'application permet une visualisation très rapide d'un grand nombre de palettes en simulant le style des graphiques matplotlib via des outils JavaScript, accessible via une simple url dans un navigateur.

L'étape finale fut alors de faire un publication de cette librairie sur PyPi, le gestionnaire de package Python le plus populaire. Cet outil est peu amené à évoluer fondamentalement, en dehors de simples corrections de bugs et d'ajout de palettes.

Nous avons communiqué fortement sur LinkedIn et Twitter, ainsi que sur divers blogs afin d'attirer les gens vers le projet. Aujourd'hui PyPalettes est installé entre 10 et 15 fois par jour et a environ 200 favoris sur le projet Github associé contenant le code source.

III/ PyFonts

Gestion des polices

La gestion des polices de caractères est surprenamment complexe, et par conséquent, leur gestion dans un logiciel en hérite. Même si Matplotlib fournit un set de polices nativement, leur nombre reste très limitant. Cependant, Matplotlib donne surtout accès à un gestionnaire

de police (ou “font manager”), qui permet l’utilisation de n’importe quel police à condition d’avoir un accès local aux fichiers binaires associés à la police souhaité.

Les problèmes de ce type de fonctionnement sont les suivants :
- nécessite le téléchargement en amont de la police sur son ordinateur
- rend le code non-reproductible en :
    - nécessitant l’écriture en clair d’un chemin de dossier propre à un utilisateur
    - obligeant tout autre utilisateur du code d’avoir les 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

Je me suis donc lancé dans la recherche d’un moyen d’accéder à n’importe quelle police sans installation quelconque. Dans la mesure où l’écrasant majorité des polices sous license libres sont répertoriées dans des projets Github, et ce que ce dernier permet facilement d’accéder à une url vers la version “brute” (ou “raw”) d’un fichier binaire, je me suis lancé dans la création d’un moyen d’aller chercher les fichiers à leur source même.

La première version actuelle de PyFonts permet d’accéder à une fonction load_font(), qui à partir d’une url vers n’importe quel police de charger un objet FontProperties via le gestionnaire de police de Matplotlib. Dans la pratique, la fonction va créer un fichier temporaire local du fichier binaire, l’utiliser pour initialiser l’objet FontProperties et ensuite supprimer le fichier. De cette manière le code est court (entre 3 et 4 lignes avec les outils Matplotlib traditionnels contre seulement 1 ligne avec PyFonts), simple (appel d’une fonction avec une url comme argument) et indépendant de l’ordinateur (charge depuis le net).

```
import matplotlib.pyplot as plt
from pyfonts import load_font

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

fig, ax = plt.subplots(dpi=300)
ax.text(
    x=0.5,
    y=0.5,
    s=f"What an easy way to load fonts,\nisn't it?",
    font=font,
    fontsize=15,
    ha="center",
)
plt.show()
```

Exemple d’utilisation de PyFonts avec la police Ultra en “regular”

Futur du projet

Les prochaines étapes sont de simplifier encore plus le processus créant un algorithme qui va automatiquement chercher au sein de Google Font (plus grande base de données de police) la police voulue en question uniquement via son nom. Cette étape est encore en

développement, mais PyFonts est aujourd'hui disponible sur PyPi et parfaitement fonctionnel.

IV/ DrawArrow

Tracer des flèches avec matplotlib

Matplotlib offre une large vague d'outils pour créer des flèches en tout genre : droites, courbées, avec un ou plusieurs points d'inflexions, simples ou doubles, avec certaines formes ou dessins, etc. Le problème de ces outils réside principalement dans le design de leur API.

En effet, les manières existantes d'ajouter une flèche dans matplotlib sont radicalement différentes en fonction du type de flèche, rendant une API non-intuitive pour un utilisateur lambda.

```python
import matplotlib.pyplot as plt
from matplotlib.patches import FancyArrowPatch
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(15,5), dpi=300)

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 la même flèche via Matplotlib

Même avec des cas d'usages minimalistes, le code n'est pas évident à comprendre, et surtout, il n'est pas consistent. Egalement, la complexité ainsi que la taille du code grimpent très rapidement pour des exemples plus complexes comme les flèches avec point d'inflexion, pouvant amener à dédier une part importante du code total d'un graphique uniquement pour les flèches.

Création de DrawArrow

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

Un des premiers problèmes à résoudre fut de déterminer dans quel système de coordonnées la position des flèches sera définit : entre 0 et 1 ? Celui des données ? En pixels ? Autre ? Ma solution réside dans la création de 2 fonctions, qui respectent la consistence de l'API orientée objet de Matplotlib : une fonction qui utilise les positions des données du graphique (ou Axes) spécifié et une fonction qui utilise les positions relatives du graphique "global" (ou Figure), donc entre 0 et 1 sur l'axe des x et y. Ce principe est également présent dans la librairie `highlight_text`, qui permet de formatter les annotations plus facilement dans Matplotlib.

Le second et principal problème est de réussir à écrire un code qui permet la réalisation de n'importe quel type de flèche tout en permettant à l'utilisateur de n'utiliser que les 2 fonctions citées ci-dessus. Pour la première version de DrawArrow j'ai décidé de me concentrer sur un nombre de fonctionnalité restreint, mais avec un code clair, changeable et testable. J'ai donc créer une API qui n'utilise que le FancyArrowPatch vu précédemment, puisque ce dernier permet un haut niveau de customisation 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 permet donc d'avoir un outil avec des arguments rapidement compréhensibles et une syntaxe respectant les principes de Matplotlib. En utilisant drawarrow, on peut désormais largement réduire la taille et la complexité de son code tout en gardant les fonctionnalités de Matplotlib.

Futur du projet

Drawarrow est aujourd'hui parfaitement fonctionnel et disponible sur PyPi. Cependant cette librairie manque d'un moyen de tracer des flèches avec des points d'inflexions. En effet, cela nécessite une utilisation très différentes de l'API de Matplotlib et est donc difficile à unifier avec les fonctions déjà existantes de drawarrow. Cela est donc prévu et fait partie des prochaines évolutions du projet.

Contact
Joseph Barbier--Darnal
Joseph.barbier-darnal@u-bordeaux.fr

En savoir +

www.u-bordeaux.fr

université de BORDEAUX