

Rapport Projet Python 2022

Sujet 1 : Analyse de données Airbnb



**HUANG Cécile
JSEM Yoan
LIU Alice
ZHOU Jingyi**

SOMMAIRE

Guide d'utilisation.....	3
Introduction.....	6
Présentation des parties	7
Partie 1 : Analyse descriptive des bases et visualisation	7
Partie 2 : Moteur de recherche.....	15
Partie 3 : Interface	28
Partie 4 : Représentation graphique de données cartographiques.....	35
Prolongements et applications possibles.....	37
Ce que nous avons appris	38

Guide d'utilisation

Etape 1 :

- Télécharger notre dossier : **sujet_1_HUANG_JSEM_LIU_ZHOU**

Etape 2 : Analyse descriptive

- A partir du notebook de Jupyter lancer le fichier **MainFinal.ipynb**
- Exécuter les cellules dans l'ordre pour la partie 1

Ce **MainFinal** répond aux questions de la partie 1 et 2.

Etape 3 : Moteur de recherche

- Rester sur le fichier **MainFinal.ipynb**
- Exécuter les cellules une par une si vous voulez tester les fonctions une par une, sinon passer directement à la dernière cellule avec la fonction **recherche()**. (Elle regroupe toutes les autres)
- **recherche()** vous laisse le choix de sélectionner les options que vous souhaitez personnaliser

Attention si vous avez choisi l'option "installation" et que vous voulez plusieurs installations, veuillez à bien espacer les différentes installations comme dans l'image qui suit:

What type of amenities do you need ? (give an answer separated by spaces)

tv wifi clim

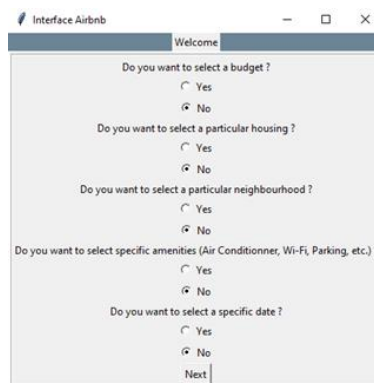
L'ensemble des fonctions ont été placées dans un script **subPart1_2.py**

Etape 4 : Interface

- Importer le script de l'interface dans une nouvelle cellule (par exemple dans le notebook **MainFinal**) pour lancer notre interface (écrire : **import InterfacePart3**)

L'ensemble des codes réalisés pour cette interface a été placé dans un script.

Une fois notre interface appelée, une première page de fenêtre Tkinter va apparaître.



Interface Airbnb

Welcome

Do you want to select a budget ?

☐ Yes

☒ No

Do you want to select a particular housing ?

☐ Yes

☒ No

Do you want to select a particular neighbourhood ?

☐ Yes

☒ No

Do you want to select specific amenities (Air Conditionner, Wi-Fi, Parking, etc.) ?

☐ Yes

☒ No

Do you want to select a specific date ?

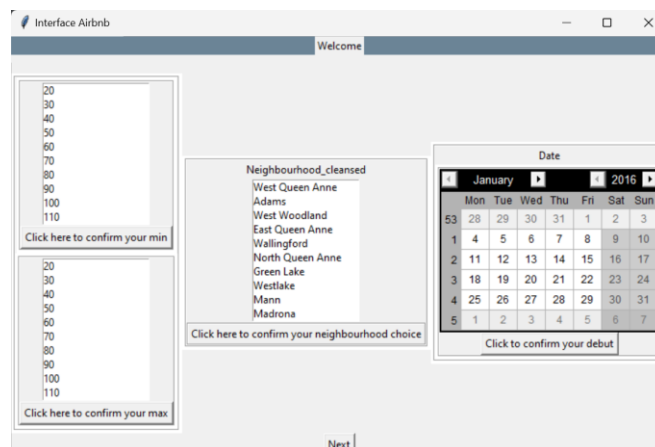
☐ Yes

☒ No

Next

- Cette page vous permet de choisir les options désirées : faites votre sélection
- Ensuite appuyer sur **Next**
- Une deuxième page va apparaître, cette page permet de préciser vos critères. (veuillez sélectionner)

Attention si vous avez choisis les options **budget**, **quartier** et **date** il faudra bien appuyer sur le bouton qui se trouve en dessous après chaque sélection.



Interface Airbnb

Welcome

20
30
40
50
60
70
80
90
100
110

Click here to confirm your min

20
30
40
50
60
70
80
90
100
110

Click here to confirm your max

Neighbourhood_cleansed

West Queen Anne
Adams
West Woodland
East Queen Anne
Wallingford
North Queen Anne
Green Lake
Westlake
Mann
Madrona

Click here to confirm your neighbourhood choice

Date

January 2016

Mon	Tue	Wed	Thu	Fri	Sat	Sun
53	28	29	30	31	1	2
1	4	5	6	7	8	9
2	11	12	13	14	15	16
3	18	19	20	21	22	23
4	25	26	27	28	29	30
5	1	2	3	4	5	6

Click to confirm your debut

Next

- Lorsque vos critères ont tous été sélectionné, appuyez sur le bouton **Next**.

Une troisième fenêtre apparaît, cette fenêtre affiche le résultat de la recherche avec à gauche la liste détaillée des locations correspondant aux critères sélectionnés et à droite une liste des id des logements répondant aux critères souhaités.

- Pour afficher le ou les images des logements, sélectionnez dans cette liste déroulante et appuyez sur le bouton en dessous pour confirmer.
- Pour lancer une nouvelle recherche, il suffit de cliquer sur le bouton **New Search**. Dans le cas où vous auriez fermé toutes les fenêtres Tkinter et que vous voulez faire une nouvelles recherche, il suffit d'appeler la fonction `reset()` du module `InterfacePart3` (exemple : `InterfacePart3.reset()`).

Etape 5 : Carte

- Dans notre dossier « sujet_1_HUANG_JSEM_LIU_ZHOU », ouvrir **MapFinal.ipynb** puis lancer les cellules une par une
- La carte interactive a un temps de chargement lent : une fois apparue vous pouvez vous déplacer avec la souris sur la carte et trouver les informations de chaque logement en navigant avec votre curseur sur les points.

Introduction

Dans le cadre du projet Python nous avons choisi le sujet 1 : Analyse de données Airbnb. Ce sujet semblait être plus intéressant et plus challengeant à traiter à première lecture. Le but d'un projet étant de progresser, nous avons décidé de joindre l'utile à l'agréable, un sujet intéressant et une occasion de progresser en programmation.

Dans le cadre du sujet nous utilisons deux fichiers, non nettoyés, avec des valeurs manquantes, un nombre d'observations et de colonnes différentes. Le but sera d'exploiter ces deux dataframes.

Le premier fichier, **Calendar**, regroupe le calendrier des disponibilités de chaque logement pour chaque jour de l'année. Nous avons donc pour chacun des logements leur identifiant, leur disponibilité sur les 365 jours de l'année avec leur prix.

Le deuxième fichier, **Listings**, est composé d'informations sur chacun des logements tel que le prix, le type de logement, le nom de propriétaires etc...

En ce qui concerne l'organisation nous ne nous sommes pas divisés en fonction des parties mais plutôt en fonction des questions. Pour la partie 1, nous étions tous sur cette même partie mais sur des questions différentes. Pour la partie 2, les premières questions ne représentant pas une grande difficulté, le travail d'une personne a suffi. Nous avons traité les questions restantes (plus techniques) par groupe de deux. La partie 3 a demandé une organisation différente, nous nous sommes répartis sur différents objectifs : l'implémentation des options, la réécriture des fonctions pour les options et la création de l'interface. La partie 4 comporte deux cartes, un groupe travaillait sur la carte basique et l'autre sur la carte interactive.

Le travail était assez libre, nous avons tous travaillé en groupe mais aussi de notre côté lorsque nous n'avions pas les mêmes disponibilités. Travailler par partie plutôt que de se répartir les tâches peut sembler contre-productif mais en réalité c'est l'inverse, de cette manière chacun a pu combler les difficultés d'un autre. Dès qu'une difficulté se présente, quelqu'un qui avait un problème similaire ou une idée pouvait venir donner son avis car nous travaillons sur la même partie. Le travail personnel hors groupe concernait souvent des détails, des petites corrections comme une manière plus optimisée de rédiger, une fonction ou encore des soucis de présentation. Ce travail était décisif puisqu'il permettait lors des séances de groupe de partir d'un code propre et faciliter l'avancée. Un code propre et sans lignes superflues (surchargé de commentaires) permet une meilleure visibilité et progression.

Présentation des parties

Partie 1 : Analyse descriptive des bases et visualisation

Afin d'effectuer une description classique des données, il nous faudrait importer et lire des données des fichiers csv. On utilise d'abord la fonction `getcwd()` du module `os` qui renvoie le répertoire actuel (c'est-à-dire où l'on se trouve) sous forme de chaîne de caractères.

Pour lire un fichier csv avec le module Pandas, on utilise la fonction `read_csv()` qui prend en argument obligatoire le chemin et nom du fichier. On assigne une variable au résultat de cette fonction, le fichier lu sera stocké en tant que data frame dans la variable créée.

Dans un premier temps, nous avons effectué une description classique des données pour chaque fichier (`calendar.csv` et `listings.csv`) :

- Pour avoir une description statistique d'un fichier, on a utilisé la fonction `describe()` qui nous permet d'avoir toutes les informations statistiques sous format d'un tableau (count, mean, std, min, max, etc). Cependant, on ne l'a appliqué que pour le data frame Listings car l'instruction `calendar.describe()` ne donne que les statistiques descriptives pour la variable "`listings_id`".
- Nombre d'observations = `calendar.shape[0]` ou `listings.shape[0]`
- Nombre de variables = `calendar.shape[1]` ou `listings.shape[1]`
- Pour afficher les noms de variables (colonnes de dataframe), on utilise la fonction `print(Dataframe.columns)`

Avant de traiter les quatre questions de la partie 1, on a créé une fonction de nettoyage préliminaire des données `nettoyage1()` dans notre module `subPart1_2` qui est nécessaire pour manipuler et faciliter les opérations suivantes. Il suffira de faire : `import subPart1_2 as s` et d'écrire `s.nettoyage1()` pour exécuter ce nettoyage.

Cette fonction `nettoyage1()`, est composée de trois éléments principaux, correspondant à la question 1, à la question 2 et à la question 4 :

- Pour Q1 : il sert à transformer la variable "`price`" du dataframe `Listings` en type `float`
- Pour Q2 : il sert à transformer la variable "`date`" du dataframe `Calendar` en type `datetime`
- Pour Q4 : il sert à transformer la variable "`price`" du dataframe `Calendar` en type `float`

On évoquera en détail au fur et à mesure comment cette procédure de nettoyage a été réalisée.

Q1. Prix moyen des locations par quartier et représentation graphique

Pour étudier cette question, nous avons eu plusieurs interprétations : prix moyen par quartier, prix moyen par logement, ou prix moyen par quartier et par logement. Après réflexion, nous avons retenu les deux premières possibilités.

On constate qu'il existe une variable "price" dans les deux fichiers, mais celui dans **Calendar** a beaucoup de valeurs manquantes pour cette colonne-là. Par conséquent, on a pris la colonne "price" dans le fichier **Listings**, où les valeurs sont plus intuitives et complètes. Cela nous facilite aussi la tâche puisque les colonnes des quartiers et types de logement se trouvent aussi dans **Listings**.

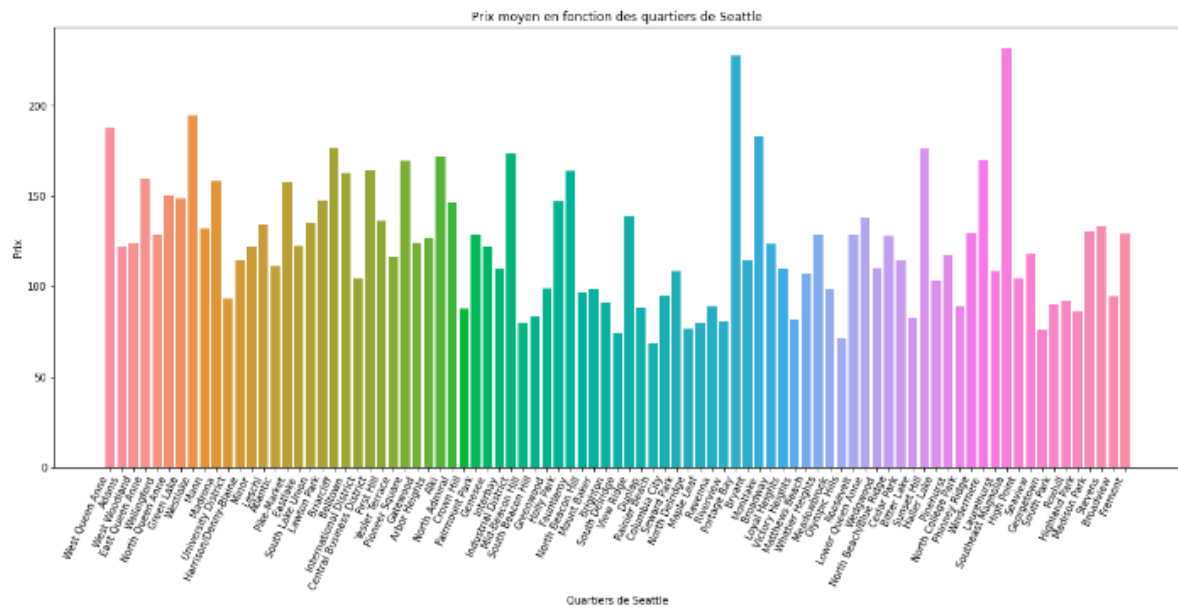
Cependant, les prix ne sont pas de type `float`, il faudrait donc les transformer en `float` pour ensuite les utiliser comme indicateur quantitatif. Pour cela, on a créé une nouvelle variable nommée "**price_dollar**" en enlevant les '\$'. Ensuite, on a utilisé la fonction **apply(float)** pour les transformer en types `float`. Toutefois, en exécutant les codes ci-dessus, on observe que l'itération **3122** pose un problème particulier : elle est la seule donnée avec une virgule, on l'a donc remplacé à la main.

A l'aide de la nouvelle variable "**price_float**", on peut ainsi calculer la moyenne par quartier en utilisant la fonction **groupby(["neighbourhood_cleansed"]).mean()**

Pour représenter graphiquement les résultats obtenus ci-dessus, on choisit la librairie **Seaborn** qui permet de créer des graphiques statistiques sophistiqués facilement et qui facilite l'interaction avec des dataframe **Pandas**.

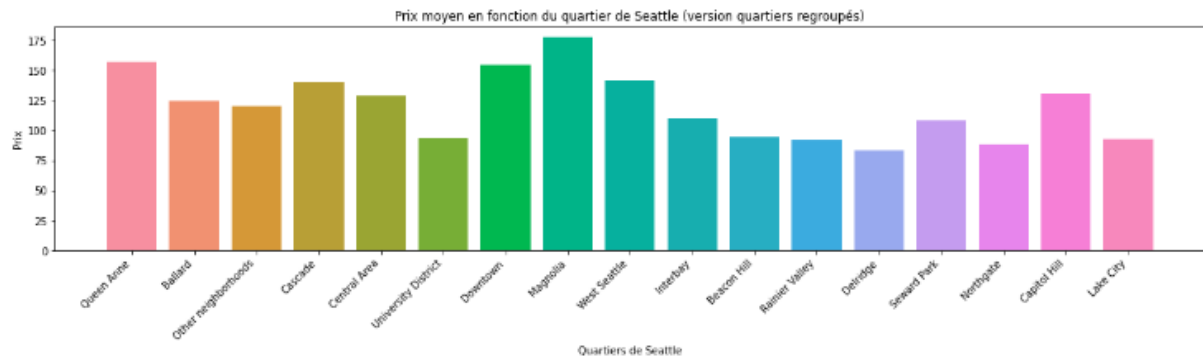
On a fait deux graphiques pour le prix moyen en fonction des quartiers à l'aide des fonctions suivantes: **plt.subplots()**, **sns.barplot()** et **axes.set_xticklabels()** qui a pour but de définir les étiquettes x-tick avec une liste d'étiquettes de type chaîne de caractères (ici, l'axe x = "**neighbourhood_cleansed**").

Graphique (1) avec $x = \text{"neighbourhood_cleansed"}$:

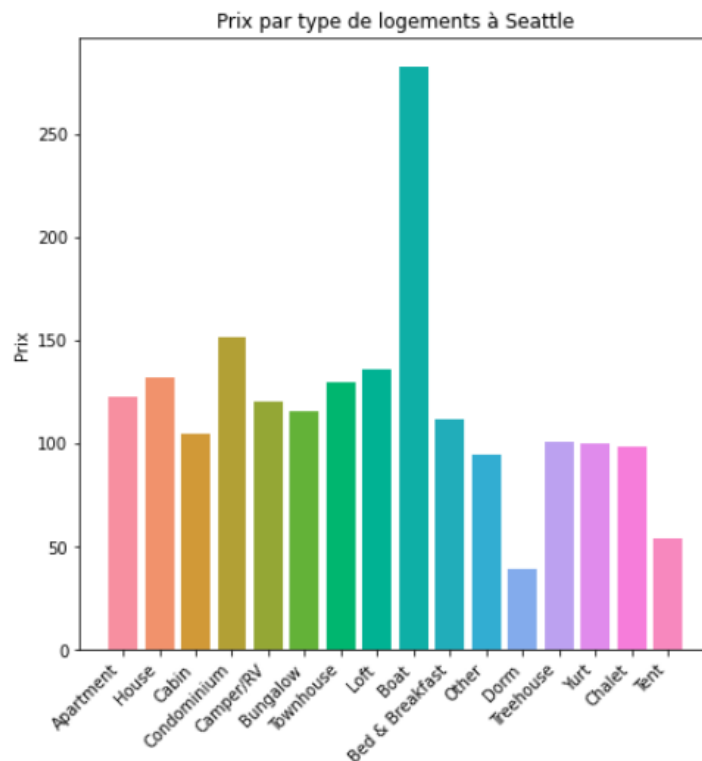


Néanmoins, comme il y a trop de quartiers listés sur l'axe des abscisses, on a aussi fait un deuxième graphique avec $x = \text{"neighbourhood_group_cleansed"}$ (une autre variable de **Listings**) afin d'avoir une meilleure visibilité :

Graphique (2) avec $x = \text{"neighbourhood_group_cleansed"}$



Graphique (3) avec $x = \text{"property_type"}$: (le prix moyen en fonction du type de logement)



Q2. Logements disponibles

Pour traiter cette question liée au temps, on a aperçu que la variable “date” dans **Calendar** n’est pas du bon type — `datetime`, on a donc appliqué la fonction `to_datetime(calendar["date"], format="%Y-%m-%d")` qui se trouve dans la fonction `nettoyage1()` pour la transformer en `datetime`. Ensuite, afin de chercher les mois de l’année où il y a le plus et le moins de logements disponibles, on a créé deux nouvelles variables “**year**” et “**month**” en type `str` et les a ajouté dans le dataframe **Calendar**.

Quant à l’interprétation de la question “Quels sont les mois de l’année où il y a le plus de logements disponibles ? Le moins ?” : on a pensé à deux versions différentes.

Au début, on a considéré qu’un logement qui est disponible 3 fois au mois de janvier 2016 sera compté 3 fois. On a donc fait une somme pour le nombre de disponibilités d’un même logement du mois en utilisant la fonction `groupby(["month"]).agg({"dispo": "sum"})`.

De cette façon, le mois de l’année 2016 où il y a le plus de logements disponibles est le mois de décembre avec un nombre de disponibilité du mois de 87061 ; et celui où il y a le moins de logements disponibles est le mois de janvier 2016 avec 59239 logements.

Néanmoins, on pense que cela est assez incohérent avec la réalité : normalement, les mois de décembre et janvier sont les mois de pointe de l’année pour le tourisme où il y a beaucoup de

demandes de logements Airbnb. Par ailleurs nous n'avons pas de mesure précise, la disponibilité de 87061 représente les jours disponibles en janvier et non le nombre de logements disponibles.

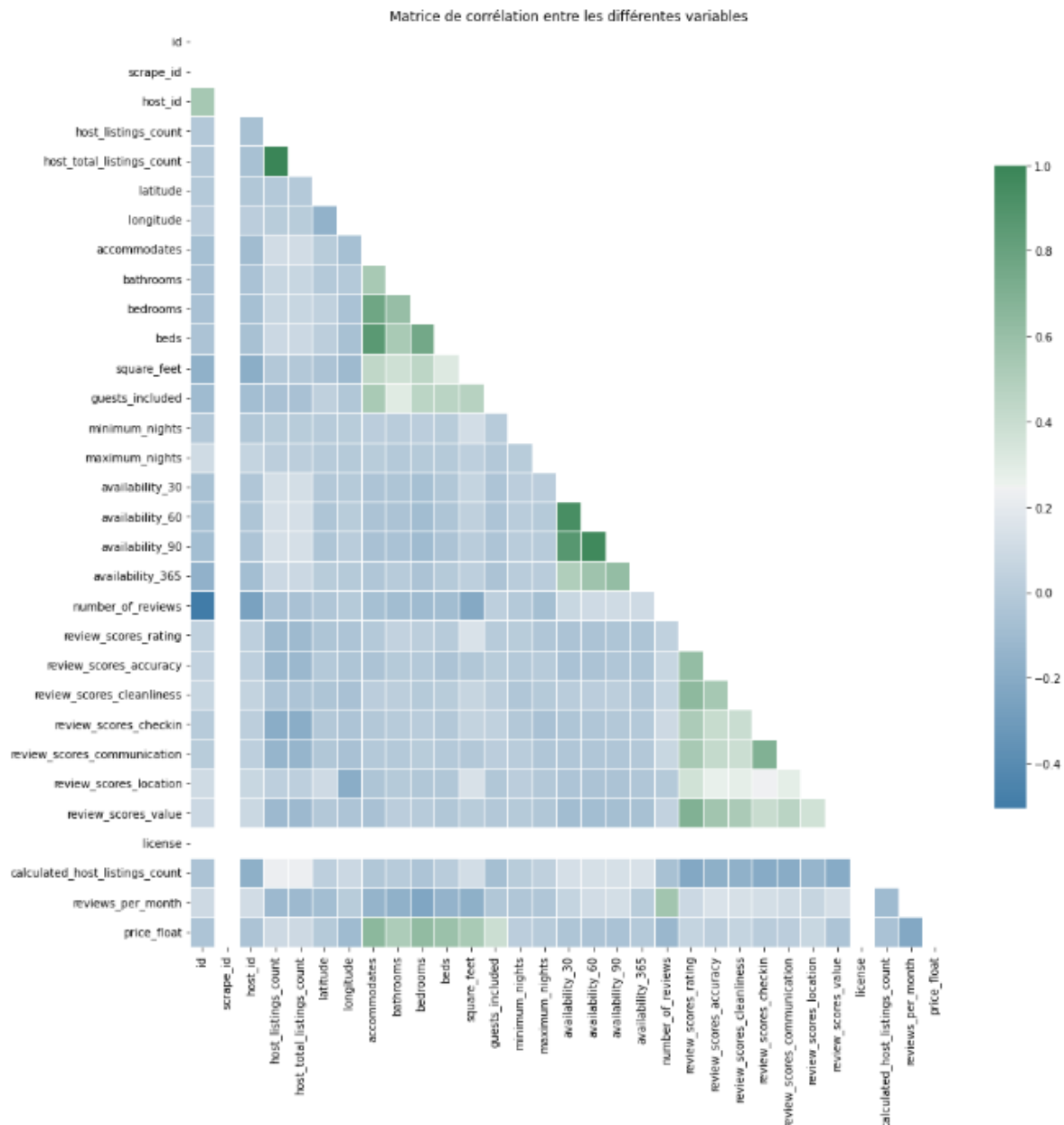
Par contre, selon notre deuxième interprétation : si on compte une fois pour un logement disponible à partir du moment où celui-ci est disponible au moins 1 jour dans le mois, c'est-à-dire si le logement d'id 3335 est disponible 3 jours du mois de janvier, on le compte 1 fois quand même, et non 3 fois. Pour cela, on a d'abord créé une nouvelle dataframe **df_inter** où on considère que pour un même logement si il est disponible au moins un jour du mois il sera marqué **dispo** pour ce mois-ci en faisant **groupby(["listing_id", "year", "month"]).max()**. Ensuite, on cherche à compter le nombre de logements disponibles pour chaque mois de l'année 2016 et l'année 2017 séparément avec la fonction **groupby(["year", "month"]).sum()**. Ce qui nous amène à trouver qu'en 2016, le mois où il y a le plus de logements disponibles est le mois d'avril, tandis que le mois avec le moins de logement disponible (le plus prisé) est celui de juillet. (Plus conforme à la réalité). Comme il y a qu'un seul mois en 2017 dans la base de données, on s'intéresse seulement à l'année 2016.

Q3. Déterminants du prix de location

Pour étudier l'influence des différentes variables sur le prix des locations(**price_float**), on a créé une matrice de corrélation à l'aide des fonctions suivantes :

- **plt.subplots()** pour superposer différents calques de graphique.
- **corr()** pour trouver la corrélation entre les colonnes du data frame(**listings**).
- **np.triu(np.ones_like())** où **np.ones_like()** renvoie un tableau de forme et de type données comme un tableau donné, avec des « 1 » ; et **np.triu()** renvoie une copie du tableau avec la partie supérieure du triangle (triangular upper).
- **sns.diverging_palette** pour générer une palette de couleurs divergentes.
- **sns.heatmap** pour permettre d'écrire les relations entre les variables sous forme de couleurs plutôt que de chiffres, ce qui rend alors plus intuitif.

Toutes les variables concernées sont représentées sur les deux axes. Les changements de couleur décrivent la relation entre deux valeurs en fonction de l'intensité de la couleur dans un bloc particulier.



A travers cette matrice ci-dessus, on remarque que le prix est fortement influencé de façon positive par le nombre d'accommodations, de chambres, de salles de bains et de lits. En revanche il est corrélé négativement par le nombre de reviews. Cela fait sens, en effet plus il y a de lits, de chambres et de salles de bains, plus la location est grande et spacieuse et vaudra cher.

Q4. Faire des économies et locations chères

On décide de séparer les deux années afin de chercher le mois pour lequel les locations sont les moins et les plus chères (en fonction du prix par nuit).

Cette fois-ci, on a pris la variable “**price**” du data frame **Calendar** qui n’était aussi pas en bon type (`float`), on a fait la même opération que pour la question 1 : `.str.replace('$', '')` et `str.replace(' ', '').astype(np.float64)` pour enlever tous les caractères spéciaux et les transformer en type `float`.

Par ailleurs, on a remarqué que la variable “**available**” dans **Calendar** sont en types `str` : les “`t`” (disponible) et “`f`” (indisponible). Afin de faciliter le comptage, on a créé une fonction **dispo()** pour avoir des valeurs qui valent 1 (si disponible) et 0 (si indisponible). On a créé une nouvelle colonne nommée “**dispo**” dans **Calendar** en appliquant la fonction **apply(dispo)**.

Ensuite, on a créé deux nouveaux dataframes pour chacune des deux années : **calendar_dispo_meilleur_2016** et **calendar_dispo_meilleur_2017**, qui récupèrent toutes les colonnes où `["dispo"]==1`.

Pour chercher le prix moyen des logements le moins cher et le plus cher dans l’année, on a fait **groupby(["month"]).mean()** dans les nouvelles dataframes.

Enfin, pour faire référence au mois pour lequel le prix moyen des logements est le moins cher ou le plus cher, on a d’abord créé une liste nommée “**mois**” avec tous les mois de l’année, et puis on a utilisé la fonction **df.idxmin()** et **df.idxmax()** qui permet de renvoyer l’indice de la première occurrence du minimum/maximum sur l’axe demandé. Comme les fonctions **df.idxmin()** et **df.idxmax()** ne renvoient pas des valeurs entières, alors on convertit le résultat en **int()** auquel on diminue de 1 pour faire correspondre le bon chiffre au bon mois de la liste **mois**. En effet, si **int(df.idxmin())** renvoie 1, cela signifie que c’est “janvier” mais **mois[1]** est égal à février d’où la nécessité du -1.

Les difficultés liées à cette partie :

Dans cette partie, on s’est rendu compte de l’importance d’avoir des données propres et en bon type pour nous permettre de manipuler les opérations qu’on souhaitait sur les variables. On rencontrait toujours des problèmes avec les types de variables : certaines des variables qu’on voulait utiliser ne sont pas en bon type, il fallait alors les convertir au bon type. Par exemple, pour la variable “**price**”, comme on voulait l’utiliser en tant qu’indicateur quantitatif, il fallait donc le transformer en type `float` en supprimant tous les caractères spéciaux et en faisant

attention au cas particulier de certaines observations (par exemple, l'itération 3122 dans **Listings** qui est la seule avec une virgule).

De plus, on devait aussi se mettre d'accord sur le choix du dataframe à utiliser : par exemple, la variable “**price**” existe dans tous les deux dataframe : **Listings** et **Calendar**. Même dans un même dataframe **Listings**, il y a des cas où plusieurs colonnes différentes décrivent une même caractéristique : **neighbourhood**, **neighbourhood_cleansed** et **neighbourhood_group_cleansed** qui sont tous liés à “quartier”.

On a donc appris qu'il est important de comprendre la signification de chaque variable, bien que la différence soit parfois très subtile.

Partie 2 : Moteur de recherche

Dans cette deuxième partie on s'intéresse à la recherche ciblée de logements en fonction de critères donnés. En somme, on espère à la fin de cette partie créer un moteur de recherche. Nous avons procédé comme le sujet nous guidait en créant des fonctions qui pour chaque option retourne les annonces répondant aux critères. Enfin nous avons centralisé ces fonctions dans une seule fonction "finale" qui propose à l'utilisateur de personnaliser sa recherche en fonction de tous les critères. Ainsi il suffira d'appeler la fonction **recherche()** du module **SubPart1_2** pour lancer la recherche complète. Celle-ci nous demandera d'abord si nous voulons choisir des options à personnaliser (ex: Voulez-vous choisir un type de logement en particulier ? Oui/Non) puis dans un second temps nous demandons quelles sont ces options (ex: Quel type de logement désirez-vous ? Un Loft). La fonction finit par afficher si elles existent les locations répondant au critère et invite à changer ses critères si aucune annonce n'est trouvée.

Cette partie nécessite de faire un nettoyage préliminaire des données. On exécutera la fonction **nettoyage2()** provenant de notre module **SubPart1_2** pour réaliser ce nettoyage. Nous détaillerons au fur et à mesure ce que cette procédure de nettoyage a facilité.

Pour sonder l'avis du client, cette partie se reposera essentiellement sur la fonction **input()** afin de récupérer la réponse à la question souhaitée. On retournera à chaque fois un data frame qui comportera au minimum l'id de la location, le titre de l'annonce, un résumé, le type de logement, le prix par nuit, le quartier, le nombre de chambres ainsi que le nom de l'hôte. Nous travaillerons majoritairement dans le dataset listings mais ferons appel à **calendar** pour l'option de sélection d'une plage de dates (Q9).

Q5. Contrainte au niveau du prix par nuit

Dans cette sous-partie nous voulons sélectionner des annonces qui répondent à un prix par nuit en particulier. Pour plus de clarté on s'interroge sur ce qu'on entend par "prix", pour le client il s'agit d'un budget qu'il ne veut pas dépasser. Nous avons donc eu l'idée de proposer au client de choisir une *fourchette de prix* (ex: j'aimerais trouver une location qui soit entre 40 et 80\$ la nuit).

Retourner une fourchette de prix correcte, c'est le but de notre fonction **budget()**. La fonction demande au client d'indiquer un budget minimum et un budget maximum qu'il convertit en type **float** pour qu'il soit plus aisément manipulable. Ensuite nous devons confirmer qu'il s'agisse bien d'une fourchette de prix: le budget minimum ne peut excéder le budget maximum, c'est incohérent. Dans ce cas, nous demandons au client de renseigner à

nouveau son budget minimum et maximum. Pour éviter de faire une boucle **while**, mais aussi pour avoir un code plus propre nous avons eu l'idée d'utiliser une *fonction récursive*. C'est une fonction qui a la particularité de s'appeler elle-même, couplée à une condition cela signifie que le client ne peut avancer tant qu'il n'a pas rempli une certaine condition. Tant que le budget renseigné sera incorrect, la "boucle" sera répétée. Parler de boucle serait une erreur mais la fonction récursive fait bel et bien office de boucle puisqu'elle continuera de tourner tant que la condition "budget minimum inférieur au budget maximum" ne sera pas remplie. Enfin lorsque le budget rentré est au bon format la fonction retournera une liste avec pour éléments la fourchette de prix.

La fonction **pricenight()** est la fonction principale de cette sous-partie, elle récupère un budget à l'aide de **budget()** et cherche les locations répondant aux demandes du client. Nous stockons dans la variable **z** (choisi arbitrairement) le budget et assignons respectivement à **a1** et **a2** le budget minimum et maximum. A partir de là, à l'aide de **loc** nous recherchons les annonces dans la fourchette de prix. Deux cas s'offrent à nous:

(a). Nous parvenons sans encombre à trouver des annonces, on retourne alors le résultat en triant par prix croissant avec **sort_values()** et en faisant attention à afficher pour chaque location les informations nécessaires précisées en début de partie.

(b). Aucune annonce n'a été trouvée, cela se traduit par un data frame comportant zéro ligne (**DataFrame.shape[0] == 0**). Sachant que notre budget est dans le bon format cela signifie que:

- la fourchette de prix donnée est trop basse (budget maximum trop faible)
- la fourchette de prix donnée est trop haute (budget minimum trop grand)

Pour remédier à cela, on demande au client de choisir à nouveau un budget en lui indiquant au préalable par une instruction **print** si sa fourchette de prix était trop haute ou trop basse. On arrive ainsi à revenir au cas (a).

En voici une illustration :

What is your minimum budget per night?

50

What is your maximum budget per night?

90

There are 1236 housings available with the selected budget only.

	id	name	summary	property_type	price_float	neighbourhood_cleansed	bedrooms	host_name
3041	149489	Quiet room in Northeast Seattle	Located on 2 dead-end streets near the Burke-G...	House	50.0	Matthews Beach	1.0	Kim
2438	6005222	Travelers' Comfy Futon	This is the perfect hook up for travelers! My ...	Apartment	50.0	Maple Leaf	1.0	Kristina
2492	8253758	Private Bed/Bath room North Seattle	-Beautiful, private upstairs room in shared ho...	House	50.0	Ravenna	1.0	Jeremy
649	9628972	Room in quite central district home	Enjoy your own private room in this Craftsman ...	House	50.0	Madrona	1.0	Oriana
1930	5873892	Beacon hill clean bedroom(可讲中文)	Queen size bed, beautiful view! Near by bus st...	House	50.0	Mid-Beacon Hill	1.0	Jennifer
...
116	9387189	Party Central Studio on Ballard Ave	Be in the heart of all that Ballard has to off...	Apartment	90.0	Adams	0.0	Alianna
304	6202603	Nice One bedroom in Fremont!	This cute 1 bedroom place is safe and quiet, b...	Apartment	90.0	Wallingford	1.0	Molly
3539	5738982	Ideal Location in N. Capitol Hill	Ample on street parking and access via metro t...	House	90.0	Stevens	1.0	Angela
3535	8689375	Master Bedroom with Private Bath	Located on a quiet street full of single famil...	House	90.0	Stevens	1.0	Gary
236	1799255	Cozy Queen Anne Cottage	Stay in one of the best neighborhoods in Seatt...	House	90.0	East Queen Anne	1.0	Sharon

Q6. Choix du type de location (appartement, chambre...)

Nous voulons maintenant choisir un type de logement. Mais quels sont les différents types de logements ? Plus tôt nous avons utilisé la fonction **nettoyage2()**, celle-ci comporte une partie qui nous est utile: elle récupère dans une liste nommée **logement** les différents types de logement parmi lesquels le client peut choisir. Pour construire cette liste on a utilisé la colonne qui recense les types de logements (**property_type**), nous n'avons pas besoin de doublons ni de valeurs nulles, nous nous en débarrassons avec les fonctions **drop_duplicates()** et **dropna()**. Enfin puisque nous voulons une liste à laquelle nous référer on utilise **tolist()**, pour une meilleure manipulation des chaînes de caractère on transforme cette liste en minuscule avec **lower()**. En effet, si tous les mots sont en minuscule nous n'aurons pas à nous soucier de la syntaxe.

Nous avons donc la liste **logement** qui comporte tous les types de logement. Nous nous en servons dans la fonction **bontype()** qui s'assure que le logement demandé par le client figure parmi les propositions. On utilise un **input** puis met en minuscule la proposition avant de vérifier si celle-ci est valide, si la demande est correcte, c'est-à-dire si elle est dans la liste de logements disponibles (**if a in logement**) on retourne la proposition dans le format reconnu par le data frame. On entend par là que la première lettre du mot doit être en majuscule, on utilise la fonction **capitalize()** pour mettre en majuscule la première lettre du mot. Il

y a cependant des exceptions à traiter: **Bed & Breakfast** et **Camper/RV** qui possèdent des caractères spéciaux et sont susceptibles de ne pas être bien exprimés par le client. C'est pourquoi nous préférons nous en occuper manuellement à l'aide d'une instruction **if** qui retourne dans le bon format la demande du client (ex: si celui-ci écrit "bed and breakfast" au lieu de "Bed & Breakfast" notre fonction doit comprendre de quoi parle l'utilisateur). Ici **bontype()** est aussi une fonction récursive, si la réponse entrée par le client ne fait pas partie des réponses acceptées (si aucune des boucles **if** n'est satisfaite) alors on considère et c'est le cas, que la réponse est incorrecte. Nous lui demandons de recommencer en rappelant **bontype()**, la méthode de la fonction récursive présente l'avantage pratique de ne pas avoir à réécrire les instructions conditionnelles dans une boucle.

Grâce à la fonction **bontype()** nous avons le type de logement désiré dans le bon format, la suite est similaire à la sous-partie précédente. Avec **z** (choisi arbitrairement dans notre code) le logement choisi par le client nous recherchons les annonces correspondantes et retournons par prix croissant les résultats.

Voici un exemple :

Type of housing : Apartment, House, Cabin, Condominium, Camper/RV, Bungalow, Townhouse, Loft, Boat, Bed & Breakfast, Other, Dorm, Treehouse, Yurt, Chalet, Tent

What type of housing do you want? Choose in the list above.

House

There are 1733 housings available with the selected property type only.

	id	name	summary	property_type	price_float	neighbourhood_cleansed	bedrooms	host_name
79	7011773	Pull out couch in garage	By Special Request Only	House	25.0	Adams	1.0	Elin
2251	7411863	Room E Single Bed in the conner	Single Bed in the conner of laundry room Clea...	House	25.0	Columbia City	1.0	Julian
1983	10299195	bright quiet room close to downtown	1 mins walk to bus stop, which get you to down...	House	26.0	Greenwood	1.0	Richard
3526	3994601	Crayola Home Green Room-Bunk Bed 5	Our space is a mix of a hostel and a home. We ...	House	27.0	Stevens	1.0	Michael
2443	7074024	View Seattle university washington	Graduate students Share this large craftsman h...	House	28.0	Ravenna	1.0	Daniel James
...
2518	2459519	Fabulous Views of Lakes & Mountains	Centrally located, minutes to downtown. Views ...	House	750.0	Portage Bay	6.0	Megan
2566	2350464	2700sqft 4BR Capitol Hill home	Our home sits on a quiet street close to Volun...	House	750.0	Montlake	4.0	Bojana
3443	2720963	Beautiful Home near Downtown	Beautiful home that is located quiet neighborh...	House	950.0	Southeast Magnolia	3.0	Dan
2	3308979	New Modern House-Amazing water view	New modern house built in 2013. Spectacular s...	House	975.0	West Queen Anne	5.0	Jill
3122	4825073	Cute Basement Apartment	2 bedroom fully finished basement apartment wi...	House	1000.0	Roosevelt	2.0	Christina

Q7. Choix du quartier

L'objectif ici est de filtrer les annonces en fonction du quartier de Seattle désiré. Grâce à la fonction **nettoyage2()** nous récupérons une liste (**quartier**) comprenant tous les quartiers de Seattle obtenu sur le même principe que la liste **logement** précédemment. Cette

liste utilise les noms de quartiers qu'il a obtenus via la colonne **neighbourhood_cleansed**, nous utilisons cette colonne car elle ne possède pas de valeur nulle.

Une fois la liste des quartiers récupérée nous faisons comme dans la Q6 avec notre fonction **bonquart()** qui vérifie que la réponse de l'utilisateur correspond bel et bien à un quartier existant. On traite à nouveau les exceptions qui comportent des caractères spéciaux et l'on utilise encore une fois une fonction récursive pour le cas où le client écrirait une réponse ne figurant pas dans la liste. Lorsque la bonne réponse est saisie nous utilisons la fonction **espace(a)** (située en dessous de la fonction **nettoyage2()**) pour que la fonction renvoie le choix dans un format acceptable (une majuscule au début de chaque mot ex: "south delridge" doit nous renvoyer "South Delridge"). Ici nous passons par une fonction car la liste des quartiers comporte souvent plusieurs "mots" et des espaces, ce qui n'était pas le cas précédemment. Avec cette fonction on compte le nombre d'espaces, si le nombre d'espaces est différent de zéro (il y a au moins un espace) alors on sépare la chaîne de caractère en une liste de mots sans les espaces (**split(' ')**). Puis on met en majuscule la première lettre et on les concatène à nouveau avec une boucle dans une chaîne de caractère en ajoutant un espace. Pour éviter d'avoir un espace en trop il nous suffit de prendre tous les indices de la chaîne de caractère sauf le dernier.

La fonction **quartype()** est la fonction principale de la sous-partie, avec **bonquart()** elle récupère le quartier sélectionné par le client, nous stockons cette réponse dans **z** (choisi arbitrairement). Puis comme à chaque fois nous recherchons (avec **loc**) les annonces puis retournons celles qui répondent au critère par prix croissant et en affichant les informations nécessaires.

Par exemple, si on choisit "Loyal Heights" :

```
Type of Neighbourhood :
['west queen anne', 'adams', 'west woodland', 'east queen anne', 'wallingford', 'north queen anne', 'green lake', 'westlake', 'mann', 'madrona', 'university district', 'harrison/denny-blaine', 'minor', 'leschi', 'atlantic', 'pike-market', 'eastlake', 'south lake union', 'lawton park', 'briarcliff', 'belltown', 'international district', 'central business district', 'first hill', 'yesler terrace', 'pioneer square', 'gatewood', 'arbor heights', 'alki', 'north admiral', 'crown hill', 'fairmount park', 'genesee', 'interbay', 'industrial district', 'mid-beacon hill', 'south beacon hill', 'greenwood', 'holly park', 'fauntleroy', 'north beacon hill', 'mount baker', 'brighton', 'south delridge', 'view ridge', 'dunlap', 'rainier beach', 'columbia city', 'seward park', 'north delridge', 'maple leaf', 'ravenna', 'riverview', 'portage bay', 'bryant', 'montlake', 'broadway', 'loyal heights', 'victory heights', 'matthews beach', 'whittier heights', 'meadowbrook', 'olympic hills', 'roosevelt', 'lower queen anne', 'wedgwood', 'north beach/blue ridge', 'cedar park', 'bitter lake', 'sunset hill', 'haller lake', 'pinehurst', 'north college park', 'phinney ridge', 'windermere', 'laurelhurst', 'southeast magnolia', 'high point', 'seaview', 'georgetown', 'south park', 'roxbill', 'highland park', 'madison park', 'stevens', 'broadview', 'fremon t']
```

What type of Neighbourhood do you want? Choose in the list above.

Loyal heights

There are 52 housings available with the selected neighbourhood.

	id	name	summary	property_type	price_float	neighbourhood_cleansed	bedrooms	host_name
3018	9563749	Spacious room	This is a large bedroom in a three bedroom ups...	Apartment	38.0	Loyal Heights	1.0	Daminda
3016	6884820	Ballard, Seattle Sunny Bedroom	Light filled room in Ballard neighborhood of S...	House	45.0	Loyal Heights	1.0	Daminda
2981	9619737	Bright by Ballard Bridge & Buses II	Two rooms upstairs, each with a bed that can a...	House	46.0	Loyal Heights	1.0	Jas
2983	8866331	Bright by Ballard Bridge and buses!	Two rooms upstairs, each with a bed that can a...	House	46.0	Loyal Heights	1.0	Jas
3017	6884107	Simple private room near	Come enjoy Seattle! This is a private	House	47.0	Loyal Heights	1.0	Daminda

Q8. Choix des installations (Wifi, Climatisation...)

Pour cette sous-partie nous souhaitons créer une fonction qui prend en compte un ou plusieurs choix d'installations et affiche les annonces qui possèdent toutes ces installations. Nous avons choisi de proposer à l'utilisateur de choisir parmi cinq différentes installations : la télévision, le wifi, la climatisation, le parking et la présence d'un ascenseur. Aussi nous avons fait un choix quant à l'implémentation de cette fonction. Lorsque l'utilisateur choisit une installation par exemple la climatisation, nous considérons qu'il veut absolument l'air conditionné (nécessaire) et se fiche du reste. Cela ne le dérange pas si le wifi ou la télévision est présent dans sa location pour peu qu'il y ait la climatisation.

La première difficulté de cette partie est de stocker et traiter les données de la colonne **amenities**. Chaque ligne est présentée sous la forme d'une seule chaîne de caractère avec des caractères spéciaux (ex: { , " / }) il faut donc nettoyer la colonne pour la rendre manipulable. L'idée est de transformer chaque ligne (**apply**) de la colonne en une liste ne contenant que les installations disponibles sans caractères spéciaux puis de créer des variables pour chaque critère qui viendront confirmer ou non la présence de l'installation dans la location. C'est le rôle de la fonction **nettoyage2()**, tout d'abord nous créons la colonne **instal** qui transforme chaque ligne de la colonne **amenities** en une liste avec la commande **apply(list)**. A ce stade la colonne **instal** avec laquelle nous travaillerons désormais ressemble plus ou moins à ceci:

```
instal[i]=ex: [{"{","E","l","e","v","a","t","o","r"," ","i","n"," ","b","u","i","l","d","i","n","g"," ","etc}]
```

Nous avons une liste qui sépare chaque lettre, espace ou caractère spécial. Maintenant il faut nettoyer cette liste, avec **espace 2()**. Nous faisons une boucle de la taille de la ligne qui va parcourir les éléments de cette ligne et effectuer des changements qu'on stockera dans une chaîne de caractère **b** (nommée arbitrairement). A chaque indice de la ligne on commence par supprimer les caractères spéciaux tels que { " et } avec l'instruction **replace("caractère spécial","")**: on remplace le caractère spécial par du vide ce qui revient à supprimer de la chaîne de caractère. Si l'élément n'est pas remplacé par du vide il est ajouté à **b**. De cette façon les lettres sont ajoutées une à une formant des mots. La chaîne de caractère **b** est le résultat de la transformation de la ligne (une liste de lettres) en une seule chaîne de caractère avec des virgules (ex: **b="Elevator in building,Wifi,Tv"**). Il faudrait désormais se débarrasser des virgules dans **b**. Pour faire d'une pierre deux coups on utilise l'instruction **b.split(",")**: on utilise la virgule (",") qui était le séparateur pour split la ligne et en faire une liste avec seulement les mots

(ex: **b="Elevator in building,Wifi,Tv"** deviendra **c=["Elevator in building","Wifi","Tv"]**).

Il suffit d'appliquer **espace2()** à notre colonne **instal** pour faire ce changement par ligne:
listings["instal"]=listings["instal"].apply(espace2).

Maintenant que les options sont sous une forme plus maniable nous commençons à créer nos colonnes options: la colonne **wifi** indiquera **1** si la location possède du wifi et **0** sinon. Pour cela on va créer une fonction **ouiwifi(a)** qui recensera si dans la location figure le wifi (ici présent sous la forme de “**Wireless Internet**” ou encore de “**Internet**”). Si le mot “**Wireless Internet**” ou encore “**Internet**” figure dans la ligne de la colonne **instal** cela signifie que la location possède le wifi, on pourra retourner comme valeur **1**, si ce n'est pas le cas on retourne **0**: il n'y a pas de wifi indiqué dans cette location. Sur le même principe on crée la colonne **TV** avec la fonction **ouitv(a)** et ainsi de suite pour la climatisation, le parking, et l'ascenseur.

Nous avons après avoir effectué **nettoyage2()** les colonnes **TV**, **wifi**, **clim**, **parking** et **elevator** qui nous renseignent sur les accommodations que possède la location. Vient alors la fonction de vérification comme à chaque sous-partie : **boninsta2()** demande à l'utilisateur d'écrire les options qu'il souhaite séparer par des espaces. Cette fois-ci nous procédons différemment : nous cherchons si les mots clés figurent dans ce qu'a écrit l'utilisateur. Auparavant nous vérifions si ce qui avait été écrit figurait dans notre liste des mots à entrer mais cette fois c'est l'inverse. Pour stocker les différents choix de l'utilisateur on utilise une liste **z** (nommée arbitrairement), on indexe **z** comme une liste composée de zéros. Chaque indice correspond à une option, si le client souhaite une télévision et entre le mot “tv” alors le premier indice de **z** vaudra **1** (**z[0]=1**). On procède de manière analogue pour les autres options. Nous ajoutons aussi l'option “rien” qui est une option lorsque le client ne veut ni tv, ni wifi, ni climatisation, ni parking, ni ascenseur (pas très utile mais facile à implémenter). Au final, **z=[tv,wifi,clim,parking,elevator,nothing]** avec un **1** si il désire cette option. Si toutefois il a choisi l'option rien on remplace directement **z=[0,0,0,0,0,1]** il serait contradictoire de vouloir une télévision si on choisit de ne pas avoir de télévision.

La fonction principale de cette sous-partie **instatype()** cherche les annonces qui possèdent les installations demandées par le client. Cette fois le principe est différent des parties précédentes puisque nous n'avons pas un critère sur une seule colonne mais cinq critères sur cinq colonnes.

Par ailleurs nous avons aussi un problème qui peut être illustré par un exemple : si le client demande une location avec wifi et clim (**z=[0,1,1,0,0,0]**) il nous faut trouver les locations qui satisfont ces deux conditions, en d'autres termes, parmi les locations avec le wifi et les locations avec la climatisation, nous voulons celles qui possèdent la climatisation ET le wifi. Spontanément on serait tenté de faire:
listings.loc[(1["TV"]==0) & (1["wifi"]==1) & (1["clim"]==1) & (1["parking"]==0) & (1["elevator"]==0)]

Seulement le sens serait différent ! En utilisant cela on chercherait les locations avec climatisation et wifi **mais sans** télévision, parking et ascenseur ! Ce qui n'est pas pareil que de chercher des locations avec climatisation et wifi tout court, dans ce dernier on se fiche de si la location possède un parking ou encore une télévision. Comme nous le disions en début de partie, si le client désire la climatisation et le wifi, c'est qu'il est prêt à accepter n'importe quelles options pour peu que celles qu'il a choisies soient présentes.

Après avoir expliqué cette démarche on essaie de l'appliquer au code, faire cela revient à faire un inner join, en termes d'ensembles ($A \cap B$), donc si le client choisit la tv, le parking et l'ascenseur il faudra faire l'intersection des trois conditions. Seulement nous n'avons pas le loisir de faire la jointure en fonction de chaque option choisie (il y aurait trop de possibilités ex: Wifi & Tv, Wifi & Clim & Tv, Parking, Clim & Wifi etc.). Bien que cela soit peu efficace, nous optons pour une autre manière. Dans le cas où l'option n'est pas choisie, nous allons faire un inner join avec l'ensemble entier: l'intuition est que pour A un ensemble de R faire $A \cap R$ nous donnera toujours A. Ainsi au lieu de forcer les autres options à ne pas être présentes on utilise la jointure de l'ensemble entier (ici l'ensemble est le data frame listings).

On commence par indexer autant de copies du data frame listings que d'options possibles (5 options donc 5 data frames qu'on nomme de **a** à **e**). Nous récupérons les options choisies (**z=boninsta2()**), si l'indice **i** de la liste **z** est égal à 1 cela signifie que l'option est désirée. ex: si **z[1]==1** cela signifie que l'utilisateur souhaite le wifi dans sa location, on change la valeur de **b** pour qu'elle réponde à la condition "avoir le wifi" **b=b.loc[b["wifi"]==1]** de cette façon on prend de manière confondue parmi les locations qui ont le wifi les locations qui ont une climatisation et celles qui n'en n'ont pas, le problème est résolu. Il suffira ensuite de faire un **merge** de tous les data frames pour avoir les locations qui répondent aux conditions indiquées et qui ne prêtent pas attention aux autres conditions. On utilise la fonction **pd.merge(a,b,how="inner")** en précisant le mode de jointure pour créer notre data frame final, il ne nous reste qu'à trier par prix croissant et tout sera en ordre.

Par exemple, si on veut sélectionner uniquement la télévision et le wifi :

Please choose one or more amenities among : TV, Wifi , Clim, Parking, Elevator, Nothing

What type of amenities do you need ? (give an answer separated by spaces)

TV clim

There are 563 housings available with the selected amenities.

	id	name	summary	property_type	price_float	neighbourhood_cleanse	bedrooms	host_name	TV	wifi	clim	parking	elevator
346	8919070	West Seattle, like Seattle but West	You don't come to Seattle to stay inside, ther...	Apartment	30.0	Genesee	1.0	Conor	1	1	1	0	1
337	3656508	Cozy 1 BR, Private Bath, Comfy Bed	Our cool and comfy one bedroom is located just...	House	35.0	Crown Hill	1.0	Augustin	1	1	1	1	0
13	9374365	Gorgeous Townhome Private Room	Heart of Ballard new townhome. Cozy room with ...	House	35.0	West Woodland	1.0	Varun	1	1	1	1	0
396	9030929	Cozy Wedgwood/Ravenna House	Private bedroom with queen size bed in quiet W...	House	35.0	Ravenna	1.0	Darren	1	1	1	1	0
470	4718820	Large room, bed, desk, shared bath	Our spacious room with a king size bed	House	35.0	Whittier Heights	1.0	Augustin	1	1	1	1	0

Q9. Sélection de plage de dates.

Cette fois-ci nous voulons les locations qui sont disponibles sur une plage horaire définie, on entend par là qu'elle doit être disponible tous les jours de la période sélectionnée. Pour ce faire on va utiliser en premier lieu le data frame **calendar** qui comporte par jour et par location la disponibilité. Dans la fonction **demanderddate()** on vérifie que la date adopte le bon format et nous retournons au passage la liste des identifiants des locations (**listing_id**) qui sont disponibles sur cette période.

Comme à chaque fois, nous faisons une fonction récursive afin que l'utilisateur recommence s'il se trompe. Pour vérifier le format on impose déjà que la date soit écrite sous la forme **yyyy-mm-dd** avec les premiers **if**. Une fois que le format est respecté, on peut transformer la date du début de séjour et de la fin en **datetime**. Grâce à ce module, on peut calculer la différence de jour entre les deux, ce qui nous donnera la durée du séjour.

Lorsqu'on fait **duree = str(diff)**, on obtient un résultat du type **"1 day, 00:00:00"**. Seul le premier élément nous intéresse donc on split **duree** puis on récupère son premier élément avec **"duree[0]"**. C'est donc ce dernier qui nous donne la durée du séjour. Ensuite il faut gérer le cas où l'utilisateur aurait échangé le début et la fin de son séjour: il faut cependant que les dates échangées soient comprises entre **<2016-01-04>** et **<2017-01-02>** car il n'y a pas de location existantes sinon. Si les dates ont été échangées (vérifiable grâce à la durée

: exemple: **duree[0]<0**) mais qu'elles restent dans la période correcte nous acceptons la réponse et lui indiquons que nous poursuivons la recherche en échangeant les dates de début et fin pour qu'elles coïncident avec leur véritable sens.

Après avoir rempli ces conditions nous avons une date en bonne et due forme, nous avons récupéré dans **duree** la durée en jour du séjour. On cherche les locations disponibles pendant la période entrée avec la commande:

```
ligne_date=calendar.loc[(calendar["date"]<=fin)&(calendar["date"]>=debut)&(calendar["dispo"]==1)]
```

Mais ce n'est pas suffisant pour autant ! En réalité, avec cette commande nous avons récupéré toutes les locations qui étaient disponibles au moins un jour pendant cette période, ce qui est différent d'être disponible **tous** les jours de cette période. C'est pourquoi nous allons récupérer dans une liste (**liste_id**), l'identifiant de ces locations. Avec ceci nous allons pour chaque identifiant créer un data frame (**df**). On parcourt chaque élément de **liste_id** et on récupère les lignes du data frame **ligne_date** qui correspondent au i-ème élément sélectionné de **liste_id**. On peut accepter le logement comme répondant au critère si **df.shape[0] == int(duree[0])** c'est-à-dire si la location est disponible pendant toute la durée du séjour, on stocke l'identifiant de la location dans une liste (**liste_id_definitive**) lorsque c'est le cas.

La fonction **logement_dispo()** utilise la fonction précédente afin de récupérer la liste **liste_id_definitive**. Ensuite, elle récupère uniquement les lignes de listings dont les identifiants des logements se trouvent dans **liste_id_definitive**. Elle garde les colonnes **"id"**, **"name"**, **"summary"**, **"property_type"**, **"price_float"**, **"neighbourhood_cleansed"**, **"bedrooms"**, **"host_name"**. Enfin, elle retourne un dataframe trié par ordre croissant des prix.

Par exemple, si on veut les logements entre le 04/05/2016 et 10/05/2016 :

Choose a date between <2016-01-04> and <2017-01-02>.

When will your vacation start ? (yyyy-mm-dd)

2016-05-04

When will your vacation end ? (yyyy-mm-dd)

2016-05-10

There are 23 housings available between 2016-05-04 and 2016-05-10 .

	id	name	summary	property_type	price_float	neighbourhood_cleansed	bedrooms	host_name
2319	1758935	Ethiopian cottage #1/Private Bath	Qwn bed in light-filled, private room. Privat...	House	62.0	Mount Baker	1.0	Beth
1065	936484	Marketside Flats next to Pike! MS1	The Marketside Flats is a fantastic home base ...	Apartment	76.0	Pike-Market	1.0	Jordan
1580	9922140	Luxury Seattle Downtown Residence	I am a 5 Star host with Airbnb. This apartment...	Apartment	85.0	First Hill	1.0	Cousin
1441	365550	contemporary art loft downtown [17]	** Please read our entire listing description ...	Loft	89.0	International District	1.0	Dirk & Jaq
2292	7562331	Modern apt near Lake Washington	New listing! Newly remodeled, modern apartmen...	Apartment	95.0	Mount Baker	1.0	Lisa
1242	19611	1 Bedroom Downtown Seattle Oasis	This central unit is perfect for anyone lookin...	Apartment	107.0	Belltown	1.0	Darik

Q10. Barre de Recherche

Maintenant que nous avons toutes les options, nous souhaitons faire une fonction qui centralise les fonctions précédentes pour en faire une véritable barre de recherche. Notre idée se présente comme suit, nous demandons au client quelles options souhaite-t-il personnaliser et ensuite nous lui proposons de personnaliser parmi les options choisies. ex: si le client n'a pas de budget précis en tête, il peut très bien refuser de choisir un budget en particulier.

```
s.recherche()
```

Answer by <yes> or <no>

Do you want to select a specific budget?

Pour ce faire nous créons les fonctions **reponseBudget()**, **reponseLogement()**, **reponseQuart()**, **reponseInsta()** et **reponsePeriode()** qui retourne 1 si l'utilisateur souhaite personnaliser l'option et 0 sinon. Nous stockons ces réponses dans une liste (**stock**).

Ainsi si le client souhaite des locations dans une fourchette de prix en particulier, avec un type de logement et une période en tête on aura **stock=[1,1,0,0,1]**. A l'aide de cette variable nous savons quelles fonctions appeler pour récupérer les résultats. Dans ce cas, les fonctions

appelées seront **pricenight()** pour le budget, **proptype()** pour le type de logement et **logement_dispo()** pour la plage de date. Nous réalisons cela à l'aide d'une instruction **if**, si l'option est choisie alors la fonction sera exécutée (si **stock[1]==1** on exécute **proptype()**).

Un problème se pose à nous : si l'utilisateur choisit de personnaliser son *budget* et son *logement* ou ses *installations* et son *quartier*, nous ne faisons pas la même chose. Dans un cas nous faisons l'intersection entre les résultats de **pricenight()** et **proptype()** et dans l'autre nous faisons l'intersection entre les résultats de **instatype()** et **quartype()**. Ce problème ressemble au problème que nous avons trouvé lorsque nous avons fait la partie sur les installations.

Nous résolvons ce souci de la même façon, en utilisant un inner join avec la liste elle-même. On indexe des variables de *a* à *e* auxquelles on assigne la liste entière (on reprend listings avec les colonnes qui nous intéressent). Elles seront modifiées si le client décide de personnaliser une option et on finira par faire un **merge** avec tous les data frames pour obtenir le data frame final (**final=pd.merge(a,b,how="inner")**) qui comportera les locations qui correspondent à tous ces critères croisés. De cette manière nous n'avons pas à nous préoccuper des combinaisons possibles

Pour finir, nous ajoutons une condition finale, si aucune annonce n'est trouvée il faudra indiquer à l'utilisateur de changer sa recherche. On utilise la fonction shape: **final.shape[0]==0** , si le data frame final ne comporte aucune ligne on recommence la fonction **recherche()** . Sinon cela signifie qu'il existe au moins une annonce, notre fonction de recherche peut s'arrêter là et retourner les résultats triés par prix croissants.

Difficultés liées à cette partie :

Si les premières options pouvaient sembler triviales elles sont devenues problématiques car nous avons progressé tout au long du projet. Les fonctions des trois premières questions de cette partie ont été modifiées maintes fois. En effet, entre le début et la fin du projet nous avons progressé et nous revenions sur notre ancien code pour l'optimiser.

Un souci était de trouver comment ne pas faire fonctionner le code, en d'autres termes : chercher les failles de notre code et ensuite les résoudre. Par exemple : que faire si le budget est inversé? Comment prendre en compte les soucis d'orthographe, les caractères spéciaux et autres.

Les plus grandes difficultés rencontrées dans cette partie interviennent réellement pour les deux dernières options (les installations et les plages de dates personnalisables). Tout d'abord, les installations : au départ nous n'avions pas saisi la différence entre choisir une option et ne pas se soucier des autres et choisir une option en forçant les autres à ne pas être présente. Ce n'est que plus tard, lorsqu'on considérait la fonction de recherche comme terminée que nous avons remarqué ce détail. Comme expliqué dans la sous-partie en question la différence est majeure

puisque'elle ne donne pas le même nombre de résultats. Il a fallu trouver une solution pour éviter de faire toutes les combinaisons possibles, autrement cela aurait résulté en un nombre interminable d'instructions if.

Concernant les plages de dates, il fallait vérifier que l'utilisateur écrivait dans le bon format : vérifier la longueur du mot saisi dans l'input, vérifier qu'il a bien mis des tirets pour séparer année-mois-jour. Le plus compliqué était de calculer la durée du séjour, car on se retrouvait avec un format lié à **datetime**. Ce qui nous intéressait était uniquement un entier de type **int** donc il fallait faire plusieurs conversions et passer par un **split**.

Un dernier problème concerne la fonction **recherche()**. Nous avons essayé de faire une jointure des tableaux avec nos variables *a* à *e* assigné à listings avec ses 100 colonnes (92 de base et 8 créées). Le problème était que le tableau renvoyé par nos fonctions (**pricenight()**, **proptype()**...) ne comportait pas le même nombre de colonnes(100 colonnes contre 13 dans le tableau trié), ce qui avec l'instruction **merge** concatène les colonnes et crée une erreur d'affichage lorsqu'on essaie de tout fusionner. La solution était simplement d'assigner aux variables *a* à *e* le tableau avec les colonnes souhaitées. De cette manière les tableaux dits de base comportent le même nombre de colonnes que les futurs tableaux remplissant des conditions.

Partie 3 : Interface

Tout d'abord, pour pouvoir lancer l'interface, il faut se mettre dans une cellule du notebook

« **MainFinal** » et écrire « **import InterfacePart3** ».

Afin de réaliser notre interface, nous avons utilisé la librairie **Tkinter** dont nous n'avions aucune connaissance dessus. Nous avons donc dû, en premier lieu, comprendre les bases de ce module, notamment à l'aide des liens de référence dans le sujet et d'autres sites internet.

Passé cette étape d'introduction, après avoir exécuté les fonctions **nettoyage1()** et **nettoyage2()** qui nous rendent les données utilisables (cf. partie 2), nous avons eu l'idée de réaliser notre interface dans la même logique que notre moteur de recherche : d'abord demander à l'utilisateur s'il veut choisir telle ou telle option, puis selon ses réponses positives, lui proposer les différents choix, et enfin, afficher le résultat.

Pour cela, nous avons initialisé une fenêtre principale appelée « **Interface Airbnb** » qui existera tout au long de l'interface qui est composée de 3 pages.

Première page :

Dans cette première page, nous avons créé une fenêtre imbriquée « **frame1** » à la fenêtre principale dans laquelle nous demandons à l'utilisateur si :

- Il veut sélectionner un budget
- Il veut sélectionner un type de location
- Il veut sélectionner un quartier
- Il veut sélectionner une ou plusieurs installations
- Il veut sélectionner une date de début et une date de fin de séjour

Pour enregistrer ses choix, nous avons opté pour des **tk.Radiobutton()** et non pas des **tk.Entry()** car cela empêche l'utilisateur de se tromper dans sa réponse, il ne peut répondre que par « oui » ou « non » et de façon unique, c'est-à-dire qu'il peut choisir soit l'un, soit l'autre. Aucune autre possibilité ne devait être tolérée afin de réduire les chances que le code bug.

Chaque question était donc sous la forme :

- **tk.Label()** pour indiquer la question, puis **label.pack()** pour afficher
- **tk.IntVar()** pour récupérer la valeur (vaut 1 si « Yes », 0 si « No »)
- **tk.Radiobutton()** de « Yes » puis **pack()**
- **tk.Radiobutton()** de « No » puis **pack()**

A la fin de cette première page, nous avons décidé de mettre un `tk.Button()` pour que l'on puisse changer de page. En effet, en cliquant sur celui-ci, cela active la fonction `etape_2()` qui détruit la `frame1 (.destroy())` et affiche `frame2` qui est une autre fenêtre imbriquée de la fenêtre principale.



Remarque : les lignes de codes relatives à la première page et aux définitions de frame ont été mises dans une fonction appelée *reset* afin que l'on puisse relancer l'interface lorsqu'on clique sur le bouton dédié, qui est situé en dernière page d'interface. Toutes les frames et les `IntVar()` qui permettent de récupérer la valeur du « Yes » / « No » ont été mises en global pour qu'elles soient utilisables à l'extérieur de la fonction.

Deuxième page :

S'affiche donc sur cette deuxième page les fenêtres dont l'utilisateur a souhaité choisir les options.

Par exemple, s'il veut choisir le budget et le quartier il aura :

The screenshot shows a window titled 'Interface Airbnb' with a 'Welcome' header. It contains two vertical listboxes for budget selection, each with values from 20 to 110 in increments of 10. Below each listbox is a button labeled 'Click here to confirm your min' and 'Click here to confirm your max' respectively. To the right, there is a 'Neighbourhood' listbox with options: West Queen Anne, Adams, West Woodland, East Queen Anne, Wallingford, North Queen Anne, Green Lake, Westlake, Mann, and Madrona. Below this listbox is a button labeled 'Click here to confirm your neighbourhood choice'. At the bottom right, there is a 'Next' button.

S'il veut tout choisir :

The screenshot shows a more complete version of the Airbnb interface. It includes the same budget and neighborhood selection options as the previous screenshot. Additionally, it features a 'Housing' section with radio buttons for Apartment, House, Cabin, Condominium, Camper/RV, Bungalow, Townhouse, Loft, Boat, Bed & Breakfast, Other, Dorm, Treehouse, Yurt, Chalet, and Tent. There is also a 'What type of Installation do you need?' section with radio buttons for TV, WiFi, Air Conditioning (A/C), Parking place, and Elevator. At the bottom right, there is a 'Date' calendar for January 2016, with a button labeled 'Click to confirm your debut'. A 'Next' button is located at the bottom center.

Nous avons récupéré les valeurs des « Yes » / « No » en faisant appel à la fonction **get()** .

· Si l'utilisateur choisit le budget i.e. « *if a == 1 :* » :

On crée une sous-fenêtre de **frame2** dont on **pack(side=tk.LEFT)** (car on veut que les sous-fenêtres soient alignées horizontalement) et qui contient elle-même deux sous-fenêtres. Pour chacune de ces sous-fenêtres, on a une **tk.Listbox()** qui contient toutes les valeurs comprises entre le prix minimal du logement (20\$) et le prix maximal du logement (1000\$), par pas de 10. L'utilisateur doit cliquer sur le bouton « Click here to confirm » de la sous-fenêtre respective afin que son choix soit enregistré. Nous avons opté pour une **Listbox** car nous voulons afficher une liste assez large de nombres, qui soit donc déroulable pour que l'on voit tout sur notre écran et dont l'utilisateur ne peut choisir qu'un seul nombre. Cela n'était pas

possible avec les **Radiobutton**. On met en global son choix de budget afin de pouvoir l'utiliser plus tard.

- Si l'utilisateur choisit le type de logement i.e. « `if b == 1 :` » :

On met dans une deuxième sous-fenêtre un certain nombre de **Radiobutton** qui contient chacun un type de logement. L'utilisateur peut sélectionner qu'un seul choix et ce dernier est automatiquement enregistré contrairement à la sous-fenêtre du budget. On utilise **tk.StringVar()** pour récupérer la valeur du **Radiobutton**. Exemple : cliquer sur « House » renvoie « House ». On sauvegarde son choix en le mettant en global.

Remarque : on aurait pu utiliser une boucle for, mais puisque qu'on voulait voir comment il fallait écrire, on l'a fait à la main.

- Si l'utilisateur choisit le quartier i.e. « `if c == 1 :` » :

On met dans une troisième sous-fenêtre une **Listbox** de tous les quartiers de Seattle. On veut que toute la liste soit lisible sur l'écran et qui soit scrollable d'où la **Listbox**. On récupère le choix de l'utilisateur à l'aide de la fonction **recup_var()**. On enregistre son choix grâce au bouton « Click here to confirm ». On met également en global **variable_quartier2** qui correspond au quartier sélectionné, afin de pouvoir utiliser cette variable aussi bien à l'intérieur qu'à l'extérieur de la fonction.

- Si l'utilisateur veut choisir une ou plusieurs installations i.e. « `if d == 1 :` » :

On met dans une quatrième sous-fenêtre un certain nombre de **Radiobutton**. Là aussi, on se base sur notre code du moteur de recherche. Pour chaque installation (TV/Wi-Fi/Climatisation/Parking/Ascenseur/Rien de cela), on demande à l'utilisateur s'il la veut ou non. S'il sélectionne « Yes », cela renvoie 1 pour chaque question respective. Si ça ne lui importe pas, alors cela renvoie 0. On récupère bien évidemment chaque valeur de chaque réponse dans un **tk.IntVar()**, et on les met en global.

- Si l'utilisateur veut choisir une date de début et une date de fin du séjour i.e. « `if e == 1 :` » :

Pour cette dernière sous-fenêtre, qui est par ailleurs également divisée en sous-fenêtres, on utilise le module **tkcalendar**. La première sous-fenêtre contient le calendrier qui correspond à la sélection du début de séjour. L'utilisateur ne peut pas sélectionner avant le 4 janvier 2016, date pour laquelle les premiers logements sont réservables. Une fois qu'il a sélectionné la date de début, il clique sur « Click to confirm » et cela lance la fonction **calen_2** et affiche la deuxième sous-fenêtre en dessous de la première sous-fenêtre pour un soucis de visibilité. L'utilisateur est donc invité à sélectionner sa date de fin de séjour, qui est obligatoirement différente de la date de début et qui soit après, d'où l'utilisation d'un **timedelta** de 1. Il ne peut

pas sélectionner une date au-delà du 2 janvier 2017. Pour configurer le deuxième calendrier, nous avons récupéré la date de début que nous avons split selon les tirets et nous avons obtenu une liste du type ['2016','01','04']. Cela permet de régler **mindate()** en **mindate=dt.date(2016,1,4)**.

Après avoir fait ses choix, l'utilisateur clique sur le bouton « **Next** » situé en bas de la page 2. Cela active alors la fonction **etape_3()** qui va détruire la **frame2** et afficher une **frame3**.

Troisième page :

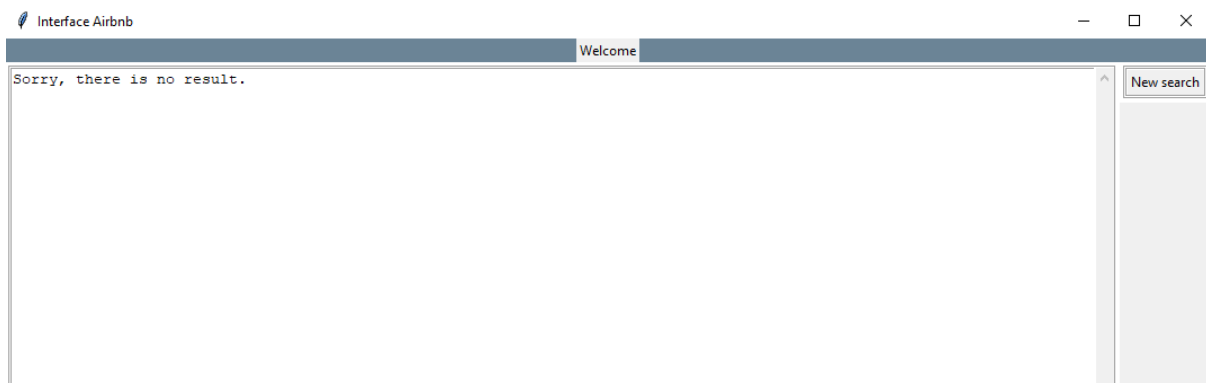
Dans cette dernière fenêtre, on affiche un **ScrolledText** qui nous permet d'afficher les résultats de la recherche par le biais de l'interface. On reprend aussi le même principe que le moteur de recherche.

On stocke dans une liste appelée stock la valeur des « Yes »/ « No » de notre première page.

Exemple : si stock = [a,b,c,d,e] = [1,0,1,0,0] : l'utilisateur veut choisir le budget et le quartier. S'il veut sélectionner le budget, alors on lance la fonction **budget()** ; s'il veut choisir le quartier alors on lance **quartype()**, etc.

Une fois qu'on obtient le data frame final comme expliqué dans la partie 2, on récupère tous les éléments de chaque colonne pour les transformer sous forme de listes. Par exemple, **property_txt = final['property_type'].tolist()** permet de mettre dans **property_txt** tous les éléments de « **property_type** ».

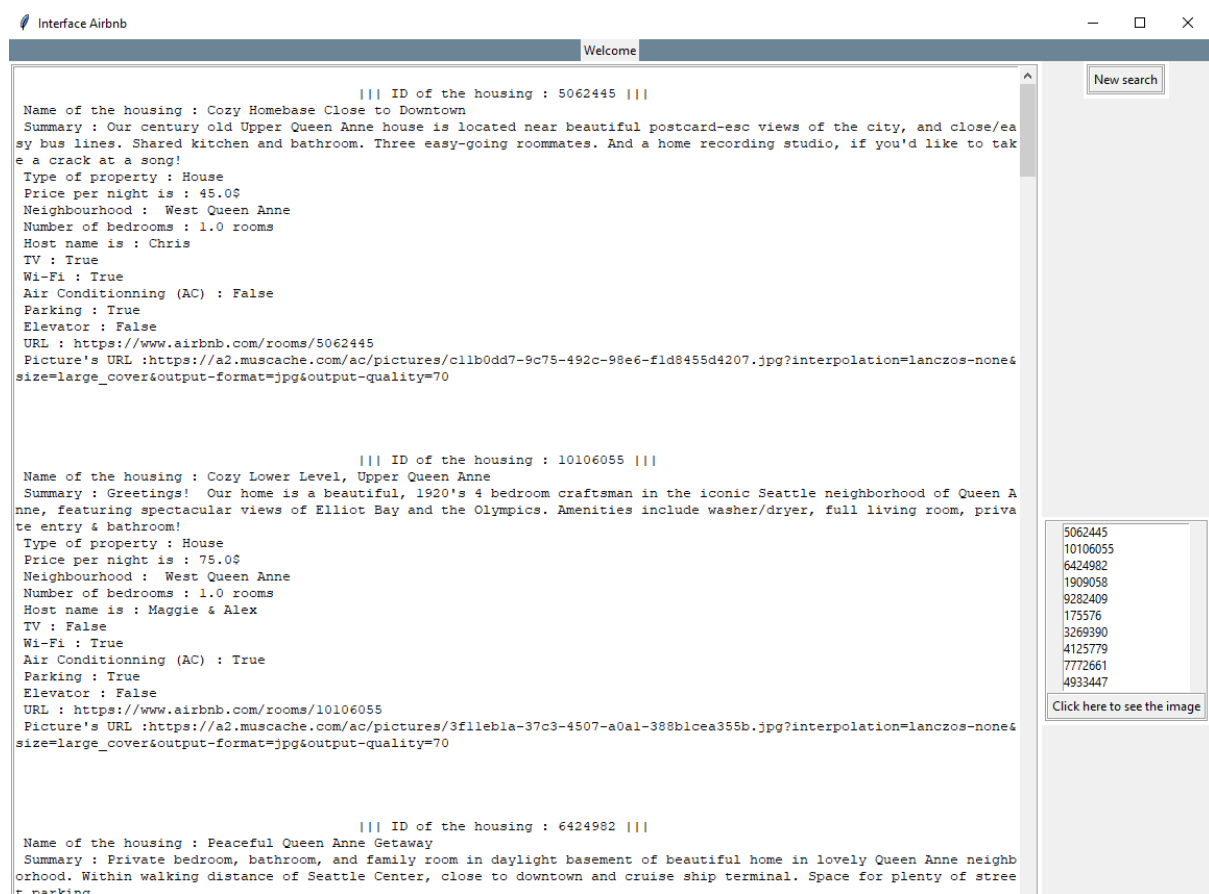
Si le data frame « **final** » ne contient aucune ligne, cela veut dire qu'aucun logement ne correspond aux critères sélectionnés. Dans ce cas on indique dans le **ScrolledText** qu'il n'y a pas de résultats.



Sinon, on affiche les résultats de la recherche à l'aide d'une boucle **for**. On met dans une variable appelée **findelafin** ce qu'on veut afficher. Comme tous les listes finissant par « **_txt** » ont toutes la même longueur, alors il n'y a pas de problème de longueur.

Enfin, on utilise la fonction **insert()** dans laquelle on réutilise la variable **findelafin** pour afficher notre texte.

Voici un résultat lorsque le data frame est non vide :



On peut remarquer à droite une **Listbox**. Elle contient tous les ID des logements qui correspondent aux critères sélectionnés. Si la liste est longue, on peut la faire défiler et on clique « click here to see the image » qui déclenche la fonction **afficher** et affiche la photo associée au logement. Cette photo s'ouvre dans une nouvelle fenêtre grâce à **Toplevel()**, de manière à en ouvrir plusieurs pour par exemple, les comparer.

Si on veut lancer une nouvelle recherche, il suffit juste de cliquer sur le bouton “**New search**” situé en haut à droite. Il lancera la fonction **destroy_frame3()** qui détruit la **frame3** et **window** et lance la fonction **reset()** qui est l'initialisation de notre interface. Cette fonction rouvrira une nouvelle en affichant la première page. Cela forme donc une “boucle”.

Les difficultés liées à cette partie :

On avait souvent le problème du code qui fonctionnait la veille et le lendemain ça ne marchait plus : il y avait toujours une erreur, donc il fallait passer beaucoup de temps pour résoudre le problème.

Il fallait comprendre la logique du fonctionnement de **Tkinter** avec la création des frames notamment.

Il fallait aussi adapter les fonctions du moteur de recherche pour qu'elles soient compatibles avec l'interface, mais on a remarqué que ça les simplifiait car les boutons empêchaient de se tromper et donc facilitaient la réécriture des fonctions.

Nous avons aussi remarqué beaucoup de problèmes d'affichage, tout ce qu'on voulait afficher ne rentrait pas dans l'écran. C'est pour cela qu'on a utilisé les **Listbox** et « joué » avec les frames pour optimiser le plus possible l'espace.

On avait souvent le problème des variables non définies donc on a opté pour la fonction **global**.

Ici, nous n'avons pas réussi à afficher un data frame tel que Pandas l'affiche mais dans **ScrolledText**, donc nous avons décidé de récupérer les informations une à une et de les afficher de façon remaniée.

On s'est rendu compte à la toute fin, en comparant notre moteur de recherche et notre interface, qu'on avait un problème au niveau des installations. Lorsqu'on lançait pour la première fois notre module de l'interface et qu'on choisissait de choisir les installations, cela ne nous sortait aucun résultat mais seulement au deuxième lancement. Cela est dû à la fonction **instatype()** qui avait une partie qui était écrite au mauvais endroit, donc on l'a déplacé dans la fonction **reset()** et tout est rentré dans l'ordre.

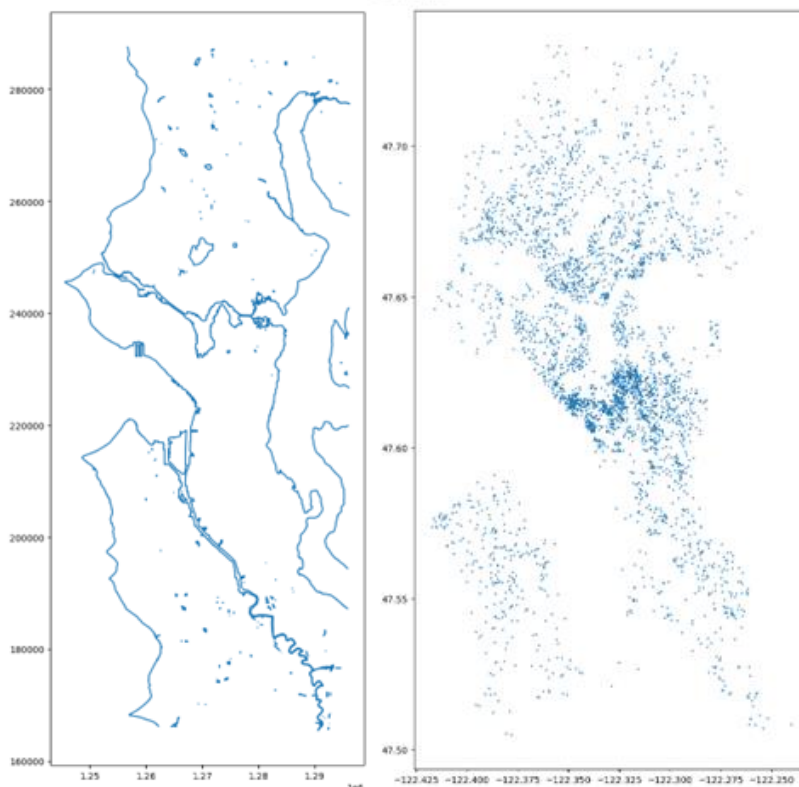
Partie 4 : Représentation graphique de données cartographiques

Pour la représentation graphique nous avons dans un premier temps regardé les liens de référence pour nous familiariser avec **Plotly** et **Geopandas** mais aussi pour récupérer notre fond de carte de Seattle.

Pour cette partie nous avons utilisé les modules : **Matplotlib.pyplot**, **Geopandas**

Une fois la carte récupérée et affichée avec `gpd.read_file()`, nous devions récupérer les coordonnées dans chaque logement qui se trouvaient dans les colonnes “longitude” et “latitude”. Cependant, elles n’étaient pas adaptées pour le format de Geopandas donc il fallait les convertir en geometry. (`geometry = gpd.points_from_xy(listings['longitude'], listings['latitude'])`).

La première difficulté rencontrée c’est l’affichage de la carte avec des points qui représentent la localisation des logements, nous avons réussi à afficher la carte d’un côté (grâce à Geopandas : `gpd.read_file()`) et les points de l’autre mais pas les points sur la carte. Nous avons vite compris que l’affichage des points avait une échelle différente de l’affichage de la carte.



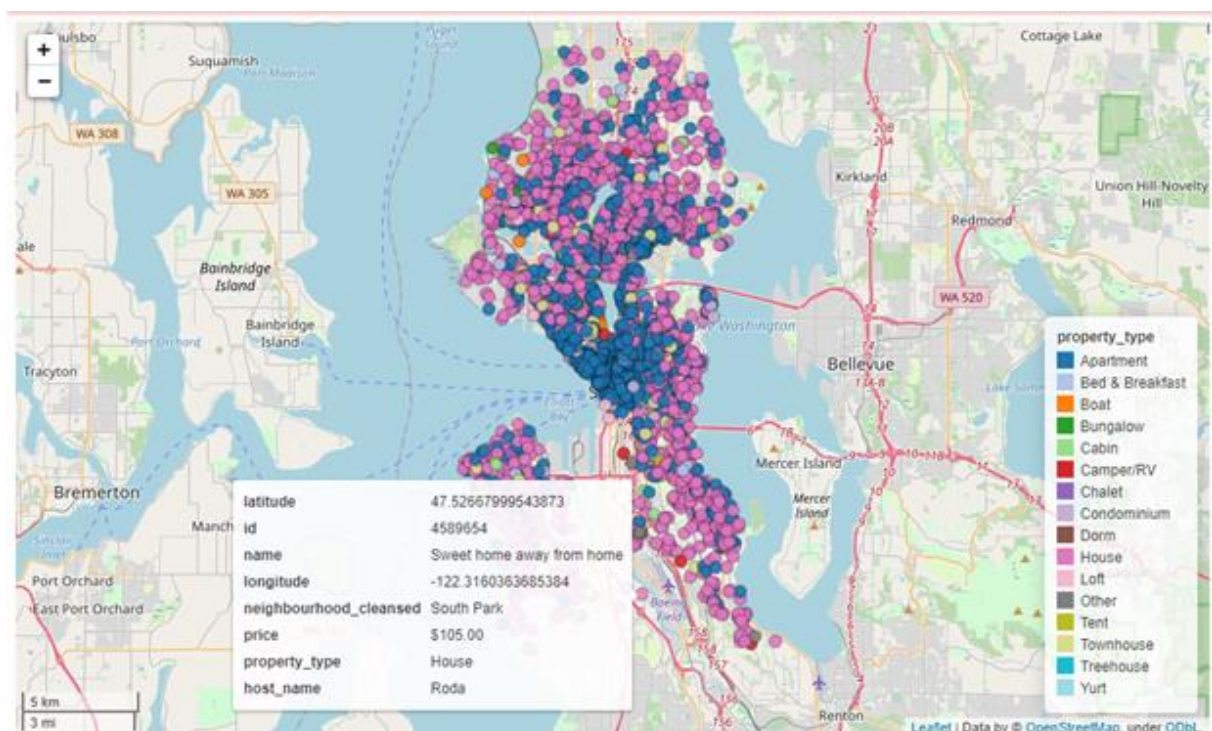
Pour surmonter ce problème nous avons téléchargé les autres formats de carte présents dans le lien. Puis nous avons trouvé que la carte en format **.geojson** était compatible avec les points.

Nous avons donc pu afficher chaque point de chaque logement mais nous avons décidé de différencier les points par type de logement pour que ce soit plus lisible.



Un autre problème que nous avons rencontré : on voulait faire une carte interactive comme nous avons pu le voir sur les cartes de Plotly mais nous n'avons pas réussi à le faire sur la carte téléchargée.

Pour aller plus loin dans cette partie nous avons créé une carte interactive qui affiche des informations lorsqu'on passe la souris sur les points de localisation grâce à **explore()** de Geopandas qui affichait toutes les informations du dataframe `Listings`. Cependant pour plus de lisibilité nous avons sélectionné les informations à afficher.



Prolongements et applications possibles

Prolongements :

En ce qui concerne le prolongement dans la partie 2 et 3 nous avons créé une fonction recherche (fonction **etape_3()** dans la partie interface) qui permet en plus de regrouper toutes les options de laisser le choix à l'utilisateur de ne pas sélectionner toutes les options. On lui propose des critères de sélection mais c'est à l'utilisateur de choisir s'il veut préciser ou non certains critères. Par exemple, il a le droit de ne pas vouloir préciser le quartier où il cherche son logement.

Pour la partie 4 nous avons créé une carte interactive avec un meilleur fond de carte et plus de lisibilité.

Nous avons pris l'initiative d'écrire des fonctions de nettoyage et le mettre dans un script. Ce qui en un appel de fonction, de nettoyer le dataframe pour les différentes parties et de gagner du temps.

Nous avons amélioré la présentation des images sur Tkinter en affichant dans une `Listbox` les numéros d'ID ce qui nous permet de choisir, l'affichage des images des logements choisis.

Pour la partie interface, si l'utilisateur souhaite effectuer une nouvelle recherche pour une quelconque raison, il le peut grâce à un bouton que nous avons implémenté dans la dernière page, à côté du **ScrolledText**. Cela ferme la fenêtre et affiche la première page.

Applications possibles :

De plus, on aurait pu mettre des hyperliens dans notre `ScrolledText` afin de rediriger vers l'annonce en question.

Ce que nous avons appris

Cécile :

Ce projet a été mon premier projet sur la programmation et n'ayant jamais fait de programmation auparavant, cela m'a permis d'en apprendre beaucoup que ce soit sur le travail de groupe ou encore sur la programmation en elle-même.

A travers ce projet j'ai pu observer les autres membres du groupe et constater que chacun a sa propre façon de coder et de penser. J'ai pu ainsi constituer instinctivement ma propre méthode. En ce qui concerne la programmation, même les choses les plus simples sont plus compliquées qu'il n'y paraît et demande beaucoup de recherche et de temps, et surtout de bien nettoyer les données sur lesquelles on travaille, ce projet nous a montré selon moi les étapes à suivre pour travailler sur des données.

Ce que je retiens de ce projet c'est qu'il faut énormément de patience, et que personnellement je ne me trouve pas assez indépendante, j'ai eu du mal à trouver les solutions par moi-même. Cependant, m'appuyer sur le groupe m'a permis de voir ce qu'est le vrai travail de groupe et le soutien qu'il apporte.

Alice :

Tout au long du projet, j'ai pu revoir les bases de Python, et de les consolider car j'en avais appris en Licence mais avait beaucoup de difficultés. Ce fut stimulant car il fallait mélanger toutes nos connaissances acquises. Mais j'ai surtout appris à manipuler les données, et compris qu'il fallait avoir des données propres et manipulables sinon ça nous causerait beaucoup de problèmes et nous ferait perdre beaucoup de temps. Travailler sur ce projet m'a fait comprendre l'importance d'avoir les bons types d'éléments (si c'est un int, un str, un float...) de vérifier l'homogénéité en général, et m'a fait développer une "logique" de programmation à avoir c'est-à-dire avoir l'idée de savoir quoi faire, puis de savoir comment le coder. J'ai aussi appris à manipuler des nouveaux modules comme Pandas, Seaborn, Tkinter, Geopandas, Datetime, ce qui m'a donné une formation assez générale de Python. Puisque je ne savais pas les utiliser, j'ai appris à chercher des solutions sur internet et je les ai adaptées à notre projet. De plus, j'ai beaucoup appris sur l'organisation lors de la réalisation des projets, la coordination entre les membres du groupe puisqu'on ne travaillait pas toujours en même temps.

Jingyi :

Ce projet de groupe m'a donné une bonne occasion d'appliquer avec souplesse ce que j'avais appris en classe et de réaliser ce projet en recherchant sur Internet des modules et des formules que je n'avais pas rencontrés (Tkinter, Geopandas, PIL, etc).

Au cours du projet, différentes erreurs se sont produites de temps à autre : variables non définies, divergences de type de variables. Mais en discutant et en cherchant les solutions avec les membres de mon groupe, on a surmonté toutes les difficultés rencontrées et cela m'a permis de m'améliorer beaucoup sur la mise en pratique de ce langage de programmation.

Grâce à ce projet, j'ai acquis une meilleure compréhension de l'utilisation pratique de Python, et j'ai également pris conscience de l'importance de type adéquat de variables et de propriétés et des caractéristiques de différentes structures de données (mutabilité et immutabilité des listes,

dictionnaires, tuples, sets). En outre, j'ai beaucoup appris à travailler avec de nouveaux modules comme Pandas, Numpy, Seaborn, Tkinter, PIL et Geopandas à l'aide des manuels sur Internet.

Yoan:

Ce projet a été divertissant et m'a permis d'acquérir une aisance dans l'utilisation des dataframes. J'ai pu voir qu'apprendre par soi-même des nouveaux modules est bien plus difficile que lorsqu'on nous l'enseigne en cours mais aussi bien plus intuitif. Apprendre par soi-même c'est toucher à tout, progresser à son rythme et se débrouiller. Cela permet d'acquérir une expérience qu'un cours n'aurait pas pu nous apporter mais cela signifie aussi passer beaucoup plus d'heures sur son travail. J'ai pu apprendre qu'un travail de groupe en programmation est plus amusant dans le sens où chacun partage sa vision et son interprétation. On peut voir la manière de coder de chacun et apprendre des habitudes des autres. On apprend aussi sur l'erreur des autres, face à un problème ou un bug, chacun à une façon d'appréhender la chose différente et il s'agit souvent d'une vision à laquelle je n'aurais jamais pu penser si j'avais été seul.