

Projet final Tkinter

Anri Kennel* (L2-A)
Module Programmation d'interfaces · Paris 8

Année universitaire 2021-2022

Table des matières

1	Consigne	2
1.1	Dépendances	2
1.2	Cahier des charges	2
2	Code	7
2.1	main.py	7
2.2	db.py	19
2.3	users.py	20
2.4	stock.py	21
2.5	stats.py	23

Les explications sont en commentaire du code.

*Numéro d'étudiant : 20010664

1 Consigne

Ici ce trouve le cahier des charges du programme. Toutes les améliorations, apportés au programme sont rangés à côtés du champs correspondant, en gras.

Pour les éléments ajoutés au programme qui ne rentre dans aucune cases, il y a une catégorie "À savoir" à la fin du cahier des charges qui les précise. Il y a aussi des informations complémentaire par rapport au projet.

1.1 Dépendances

Les modules externes utilisés sont :

- tkinter pour la GUI
 - .ttk pour la liste déroulante et les lignes qui séparent les cases du tableau
 - .messagebox pour les messages pop-up
 - .filedialog pour la boîte de dialogue du fichier
- sqlite3 pour la base de donnée SQLite
- datetime pour la date
- re pour le regex
- csv pour la gestion du fichier CSV
- random pour la génération du stock (prix et quantité)

1.2 Cahier des charges

- Page de login /1.5

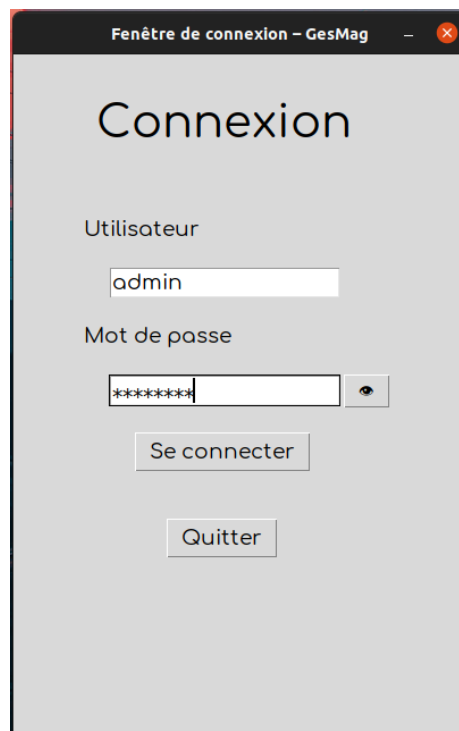


Figure 1: Page de login

- Nom d'utilisateur ne contient que des lettres et des chiffres

- ☑ Mot de passe de minimum 8 caractères dont 1 caractère spécial, une majuscule et une minuscule
⇒ **possibilité d'afficher ou non le mot de passe en clair**
 - ☑ Un bouton de connexion ⇒ **possibilité aussi d'utiliser la touche Entrer (pour aller plus vite) qui permet de se rendre sur l'interface Caissier ou Manager**
 - ☑ Un bouton pour quitter l'application
- ☑ Page de manager (défini par un nom d'utilisateur et un mot de passe) /7.5

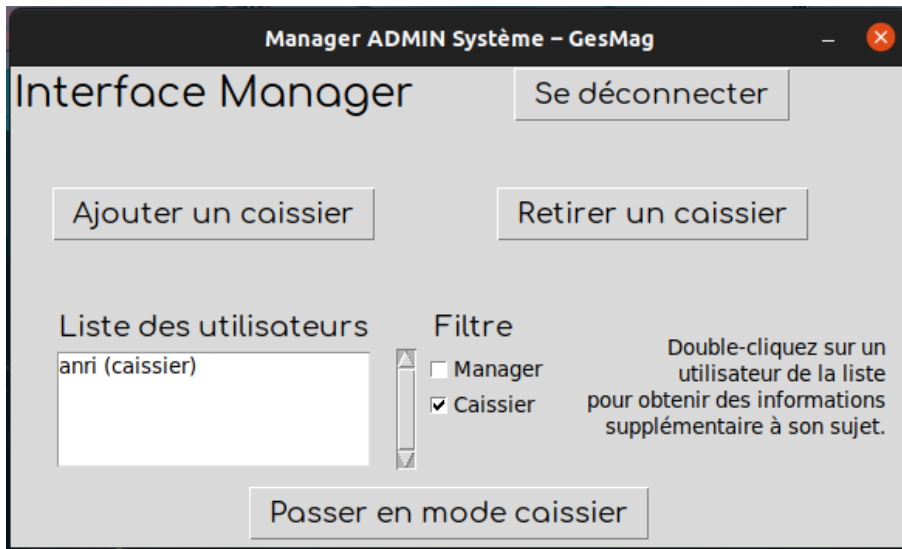


Figure 2: Page du Manager

- ☑ Peut ajouter et supprimer un caissier ⇒ **lisibilité accru pour les champs mal renseignés, l'ID n'est pas à renseigné car assigné automatiquement par la base de données**

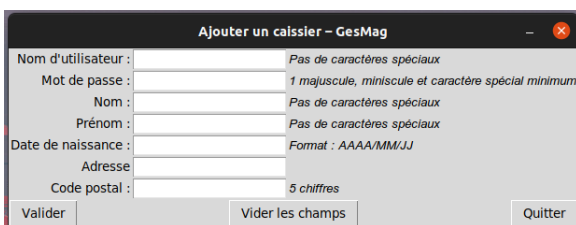


Figure 3: Ajout d'un caissier

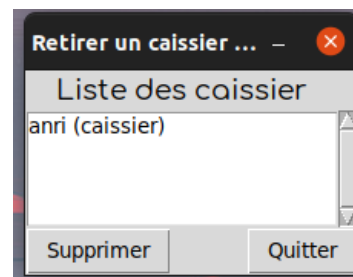


Figure 4: Suppression d'un caissier

- ☑ Peut voir la liste des caissiers ⇒ **possibilité d'ouvrir des informations étendues sur un utilisateur, ainsi que de filtrer les utilisateurs (manager et caissiers) mais impossible de tout désélectionner (caissier par défaut)**

- ☑ Un histogramme présentant l'évolution des sommes totales des ventes journalières de la semaine passée d'un utilisateur ⇒ **accessible au double-clic dans la fenêtre des informations étendues d'un utilisateur**

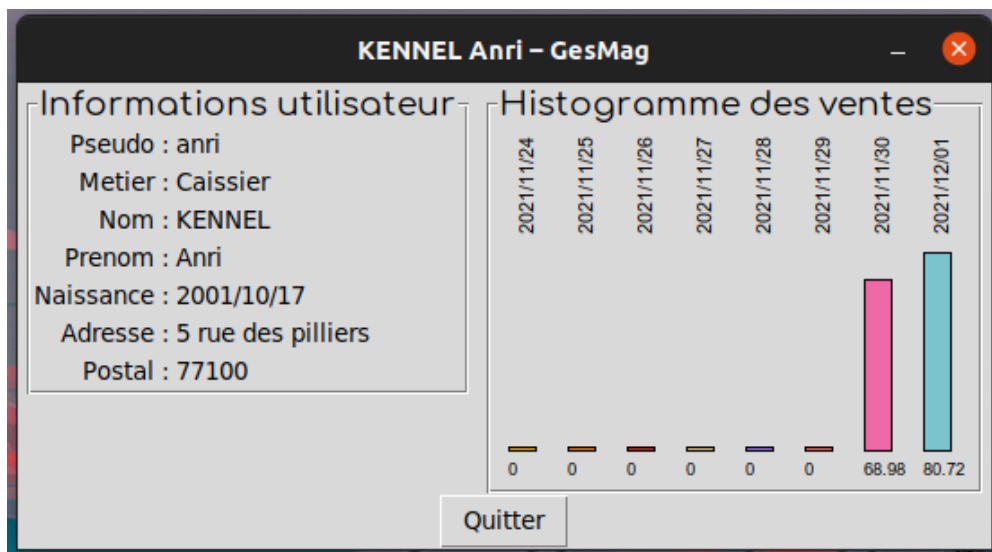
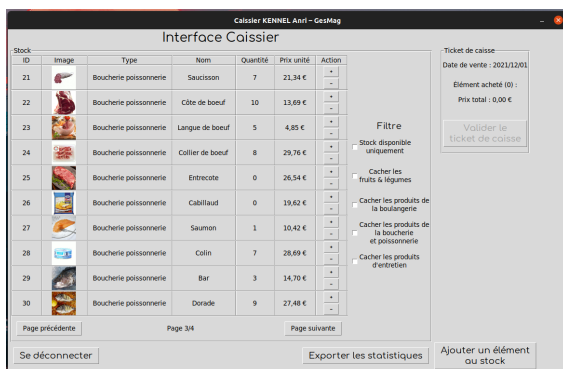
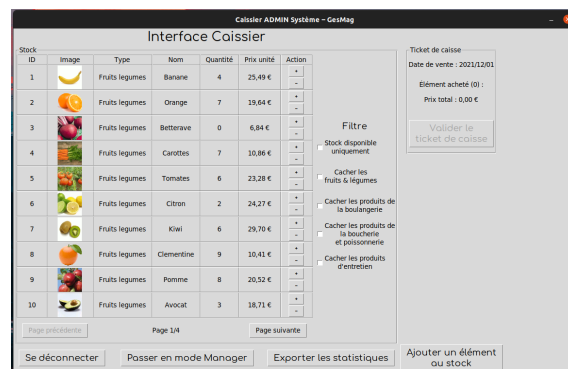


Figure 5: Informations étendues sur un utilisateur

- ☑ Un bouton pour vider tous les champs de saisie
- ☒ Un bouton pour quitter l'application ⇒ **j'ai préféré mettre un bouton pour se déconnecter**
- ☑ Un bouton pour se mettre en "mode caissier"
- ☑ Page de caissier (défini par un identifiant, un nom d'utilisateur, un mot de passe, un nom, un prénom, une date de naissance, une adresse et un code postal) /6



(a) En tant que Caissier



(b) En tant que Manager

Figure 6: Page du Caissier

- ☑ Afficher le stock disponible
 - ☒ 4 rayons de chacun au moins 10 articles de votre choix (fruits/légumes, boulangerie, boucherie/poissonnerie ou produits d'entretien) ⇒ **toutes les images sont aux dimensions 50x50 et ont été converties avec le logiciel Gimp**
 - ☒ Au clic sur le produit, l'identifiant, le nom, la quantité en stock et le prix s'affichent ⇒ **tout est affiché directement, pas besoin de cliquer sur le produit, il y a aussi un système de pages pour une meilleure lisibilité (10 éléments par page au maximum)**

- Possibilité de rajouter des produits en stock

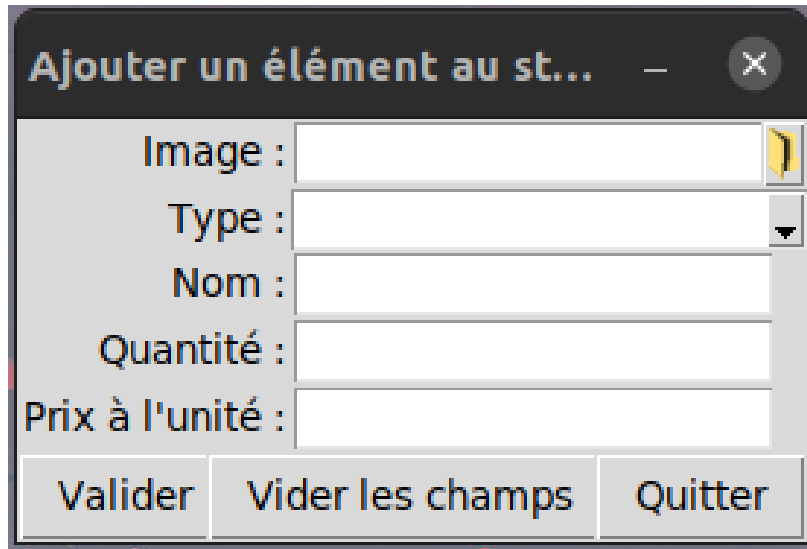


Figure 7: Ajout d'éléments au stock

- Affichage d'un ticket de caisse



Figure 8: Ticket de caisse avec 2 éléments au panier

- Date de vente
- ID, nom, quantité, prix des produits achetés
- Prix total
- Un bouton pour valider
- Interface d'export des statistiques (stock le montant total de vente par jour) ⇒ **export au format CSV**

Avec à savoir :

- Ergonomie /2
- Utilisation de Frame et peu de TopLevel, ainsi qu'une seule fenêtre Tk pour éviter de multiples ouverture/fermeture de fenêtre durant l'utilisation de l'application

- Utilisateurs stockés dans la base de donnée /2
 - Possibilité de recréer la base de donnée automatiquement si elle n'existe plus
 - Utilisation, en plus de SQLite, d'un fichier CSV pour exporter les statistiques des caissiers, et ainsi pouvoir traiter ces informations dans un tableur (outil externe) à l'avenir
- Ajout d'autres fonctionnalités /1
 - J'ai pas vraiment ajouter une toute nouvelle fonctionnalité, mais améliorer ce qui était demandé pour une plus grande souplesse à l'utilisation (cf. les cases cochés avec des ✕)
- Lisibilité du code
 - Toutes les fonctions sont commentés et typés (quand possible car j'utilises Python 3.9.7)
 - Tous le code est dans une classe et non directement dans le code (donc aucune variable globale)
 - Plusieurs fichiers pour une meilleur lisibilité
- Affichage sous forme de tableau
 - J'ai évité d'utiliser le widget Treeview du module ttk de tkinter car je le trouve que peu pratique/flexible (exemple : impossibilité de mettre des images dans les colonnes du tableau) alors j'ai développé une alternative (cf. l'affiche du stock avec un système de page)

2 Code

2.1 main.py, fichier principal

```
1 # Tkinter
2 from tkinter import Canvas, IntVar, Checkbutton, LabelFrame, PhotoImage, Scrollbar, Listbox, Entry,
   Button, Label, Frame, Tk, Toplevel
3 from tkinter.ttk import Combobox, Separator
4 from tkinter.messagebox import showerror, showinfo, showwarning, askyesno
5 from tkinter.filedialog import askopenfile, asksaveasfile
6 # Regex
7 from re import sub
8 # Date
9 from datetime import date
10
11 # Import des fichiers pour gérer la base de donnée et l'export en CSV
12 from users import Utilisateurs
13 from stock import Stock
14 from stats import Stats
15
16 class GesMag:
17     """Programme de Gestion d'une caisse de magasin."""
18     def __init__(self, presentation: bool = False) -> None:
19         """Instancie quelques variables pour plus de clareté."""
20         Utilisateurs().creationTable(presentation) # on créer la table utilisateurs si elle n'existe
   pas déjà
21         Stock().creationTable(presentation) # on créer la table du stock si elle n'existe pas déjà
22         Stats().creationCSV() # on créer le fichier CSV qui stockera les statistiques des
   utilisateurs
23
24         self.nomApp = "GesMag" # nom de l'application
25         self.parent = Tk() # fenêtre affiché à l'utilisateur
26         self.parent.resizable(False, False) # empêche la fenêtre d'être redimensionnée
27         self.f = Frame(self.parent) # 'Frame' "principale" affiché à l'écran
28         self.tableau = Frame() # 'Frame' qui va afficher le tableau des éléments présents dans le
   stock
29         self.imagesStock = [] # liste qui va contenir nos images pour l'affichage du stock
30         self.dossierImage = PhotoImage(file = "img/dossier.gif") # image pour l'icone de selection
31         self.panierAffichage = Frame() # 'Frame' qui va afficher le panier
32         self.panier = [] # liste des éléments "dans le panier"
33
34     def demarrer(self) -> None:
35         """Lance le programme GesMag."""
36         self.font = ("Comfortaa", 14) # police par défaut
37
38         self._interfaceConnexion() # on créer la variable 'self.f' qui est la frame a affiché
39         self.f.grid() # on affiche la frame
40
41         self.parent.mainloop() # on affiche la fenêtre
42
43     def motDePasseCorrect(self, motDPasse: str) -> tuple:
44         """Détermine si un mot de passe suit la politique du programme ou non."""
45         if len(motDPasse) == 0: # si le champs est vide
46             return (False, "Mot de passe incorrect.")
47         if len(motDPasse) < 8: # si le mot de passe est plus petit que 8 caractères
48             return (False, "Un mot de passe doit faire 8 caractères minimum.")
49         """
50         - Pour le regex, la fonction 'sub' élimine tout ce qui est donné en fonction
51           du pattern renseigné, alors si la fonction 'sub' renvoie pas exactement
52           la même chaîne de caractère alors c'est qu'il y avait un caractère interdit.
53         - J'utilises pas 'match' parce que je suis plus à l'aise avec 'sub'.
54         """
55         if not sub(r"[A-Z]", '', motDPasse) != motDPasse:
56             return (False, "Un mot de passe doit au moins contenir une lettre majuscule.")
57         if not sub(r"[a-z]", '', motDPasse) != motDPasse:
58             return (False, "Un mot de passe doit au moins contenir une lettre minuscule.")
59         if not sub(r" *?[\^w\s]+", '', motDPasse) != motDPasse:
60             return (False, "Un mot de passe doit au moins contenir un caractère spécial.")
61
62         return (True,) # si aucun des tests précédents n'est valide, alors le mot de passe est valide
63
64     def utilisateurCorrect(self, utilisateur: str) -> tuple:
65         """Détermine si un nom d'utilisateur suit la politique du programme ou non."""
66         """
67         Pour le nom d'utilisateur on vérifie si le champs n'est pas vide
68         et si il y a bien que des lettres et des chiffres.
69         """
70         if len(utilisateur) == 0:
71             return (False, "Utilisateur incorrect.")
72         if sub(r" *?[\^w\s]+", '', utilisateur) != utilisateur:
73             return (False, "Un nom d'utilisateur ne doit pas contenir de caractère spécial.")
74         return (True,)
75
```

```

76 def nomCorrect(self, nom: str) -> bool:
77     """Détermine si un nom suit la politique du programme ou non."""
78     if len(nom) == 0:
79         return False
80     if sub(r" *?[^\\w\\s]+", '', nom) != nom: # pas de caractères spéciaux dans un nom
81         return False
82     return True
83
84 def prenomCorrect(self, prenom: str) -> bool:
85     """Détermine si un prénom suit la politique du programme ou non."""
86     if len(prenom) == 0:
87         return False
88     if sub(r" *?[^\\w\\s]+", '', prenom) != prenom: # pas de caractères spéciaux dans un prénom
89         return False
90     return True
91
92 def naissanceCorrect(self, naissance: str) -> bool:
93     """Détermine si une date de naissance suit la politique du programme ou non."""
94     if len(naissance) == 0:
95         return False
96     # lien pour mieux comprendre ce qui se passe : https://www.debuggex.com/r/hSD-6BfSqDD1It5Z
97     if sub(r"[0-9]{4}\\/(0[1-9]|1[0-2])\\/(0[1-9]|1[1-2][0-9]|3[0-1])", '', naissance) != '':
98         return False
99     return True
100
101 def adresseCorrect(self, adresse: str) -> bool:
102     """Détermine si une adresse suit la politique du programme ou non."""
103     if len(adresse) == 0:
104         return False
105     return True
106
107 def postalCorrect(self, code: str) -> bool:
108     """Détermine si un code postal suit la politique du programme ou non."""
109     if len(code) == 0:
110         return False
111     if sub(r"\\d{5}", '', code) != '':
112         return False
113     return True
114
115 def connexion(self, utilisateur: str, motDePasse: str) -> None:
116     """Gère la connexion aux différentes interfaces de l'application."""
117     """
118     Vérification nom d'utilisateur / mot de passe correctement entré
119     avec leurs fonctions respectives.
120     """
121     pseudoOk = self.utilisateurCorrect(utilisateur)
122     if not pseudoOk[0]:
123         showerror(f"Erreur {self.nomApp}", pseudoOk[1])
124         return
125     mdpOk = self.motDePasseCorrect(motDePasse)
126     if not mdpOk[0]:
127         showerror(f"Erreur {self.nomApp}", mdpOk[1])
128         return
129
130     # Redirection vers la bonne interface
131     utilisateurBaseDeDonnee = Utilisateurs().verificationIdentifiants(utilisateur, motDePasse)
132     if utilisateurBaseDeDonnee[0] > 0:
133         if utilisateurBaseDeDonnee[1] == 0: # si le métier est "Manager"
134             self._interfaceManager(utilisateurBaseDeDonnee[0])
135         elif utilisateurBaseDeDonnee[1] == 1: # si le métier est "Caissier"
136             self._interfaceCaissier(utilisateurBaseDeDonnee[0])
137         else:
138             showerror(f"Erreur {self.nomApp}", "Une erreur est survenue : métier inconnue.")
139     else:
140         showerror(f"Erreur {self.nomApp}", "Utilisateur ou mot de passe incorrect.")
141
142 def dimensionsFenetre(self, fenetre, nouveauX: int, nouveauY: int) -> None:
143     """Permet de changer les dimensions de la fenêtre parent et la place au centre de l'écran."""
144     largeur = fenetre.winfo_screenwidth()
145     hauteur = fenetre.winfo_screenheight()
146
147     x = (largeur // 2) - (nouveauX // 2)
148     y = (hauteur // 2) - (nouveauY // 2)
149
150     fenetre.geometry(f"{{nouveauX}}x{{nouveauY}}+{{x}}+{{y}}")
151
152 def _interfaceConnexion(self) -> None:
153     """Affiche l'interface de connexion."""
154     # Paramètres de la fenêtre
155     self.dimensionsFenetre(self.parent, 400, 600)
156     self.parent.title(f"{{fenetre}} de connexion {self.nomApp}")
157
158     # Suppression de la dernière Frame
159     self.f.destroy()
160     # Instanciation d'une nouvelle Frame, on va donc ajouter tout nos widgets à cet Frame

```



```

161     self.f = Frame(self.parent)
162     self.f.grid()
163
164     # Affichage des labels et boutons
165     tentativeDeConnexion = lambda _ = None: self.connexion(utilisateur.get(), motDpasse.get()) #
166     lambda pour envoyer les informations entrés dans le formulaire
167     ecart = 80 # écart pour avoir un affichage centré
168     Label(self.f).grid(row=0, pady=50) # utilisé pour du padding (meilleur affichage)
169
170     Label(self.f, text="Connexion", font=(self.font[0], 30)).grid(column=1, row=0, columnspan=2)#
171     titre
172     Label(self.f, text="Utilisateur", font=self.font).grid(column=0, row=1, columnspan=2, padx=
173     ecart - 20, pady=20, sticky='w')
174     utilisateur = Entry(self.f, font=self.font, width=18)
175     utilisateur.grid(column=1, row=2, columnspan=2, padx=ecart)
176
177     Label(self.f, text="Mot de passe", font=self.font).grid(column=0, row=3, columnspan=2, padx=
178     ecart - 20, pady=20, sticky='w')
179     motDpasse = Entry(self.f, font=self.font, show='', width=18)
180     motDpasse.grid(column=1, row=4, columnspan=2, padx=ecart)
181     motDpasse.bind("<Return>", tentativeDeConnexion)
182
183     def __afficherMDP(self) -> None:
184         """Permet de gérer l'affichage du mot de passe dans le champs sur la page de connexion.
185         """
186         if self.mdpVisible == False: # si mot de passe caché, alors on l'affiche
187             self.mdpVisible = True
188             motDpasse.config(show='')
189             boutonAffichageMDP.config(font=("Arial", 10, "overstrike"))
190
191         else: # inversement
192             self.mdpVisible = False
193             motDpasse.config(show='')
194             boutonAffichageMDP.config(font=("Arial", 10))
195
196     boutonAffichageMDP = Button(self.f, text='', command=lambda: __afficherMDP(self))
197     boutonAffichageMDP.grid(column=2, row=4, columnspan=2)
198     self.mdpVisible = False
199
200     bouton = Button(self.f, text="Se connecter", font=self.font, command=tentativeDeConnexion)
201     bouton.grid(column=0, row=5, columnspan=3, padx=ecart, pady=20)
202     bouton.bind("<Return>", tentativeDeConnexion)
203
204     Button(self.f, text="Quitter", font=self.font, command=quit).grid(column=0, row=6, columnspan
205     =4, pady=20)
206
207 def _interfaceCaissier(self, id: int) -> None:
208     """Affiche l'interface du caissier."""
209     caissier = Utilisateurs().recuperationUtilisateur(id=id)
210     self.parent.title(f"Caissier {caissier['nom']} {caissier['prenom']} {self.nomApp}")
211     self.dimensionsFenetre(self.parent, 1160, 710)
212
213     self.panier = [] # remet le panier à 0
214
215     # Suppression de la dernière Frame
216     self.f.destroy()
217     # Instanciation d'une nouvelle Frame, on va donc ajouter tout nos widgets à cet Frame
218     self.f = Frame(self.parent)
219     self.f.grid()
220
221     Label(self.f, text="Interface Caissier", font=(self.font[0], 20)).grid(column=0, row=0) #
222     titre de l'interface
223
224     def __formatPrix(prix: str) -> str:
225         """
226         Renvoie un string pour un meilleur affichage du prix :
227         - ',' au lieu de '.'
228         - Symbole '€'
229         - 2 chiffres après la virgule
230         """
231         return f"{float(prix):.2f} ".replace('.', ',')
232
233     # -> Partie affichage du Stock
234     stock = LabelFrame(self.f, text="Stock")
235     stock.grid(column=0, row=1, sticky='n', padx=5)
236
237     # Variables pour les filtres du tableau
238     stockDisponibleVerif = IntVar(stock) # controle si on affiche que les éléments en stocks ou
239     non
240     # Cache un certain type de produit
241     fruitsLegumesVerif = IntVar(stock)
242     boulangerieVerif = IntVar(stock)
243     boucheriePoissonnerieVerif = IntVar(stock)
244     entretienVerif = IntVar(stock)
245

```

```

238     def __affichageTableau(page: int = 1) -> None:
239         """Fonction qui va actualiser le tableau avec une page donnée (par défaut affiche la
première page)."""
240         # On supprime et refais la frame qui va stocker notre tableau
241         self.tableau.destroy()
242         self.tableau = Frame(stock)
243         self.tableau.grid(column=0, row=1, columnspan=7)
244
245         # Filtre pour le tableau
246         filtres = Frame(stock) # Morceau qui va contenir nos checkbutton
247         ecartFiltre = 10 # écart entre les champs des filtres
248         Label(filtres, text="Filtre", font=self.font).grid(column=0, row=0) # titre
249         Checkbutton(filtres, text="Stock disponible\nuniquement", variable=stockDisponibleVerif,
command=__affichageTableau).grid(sticky='w', pady=ecartFiltre)
250         Checkbutton(filtres, text="Cacher les\nfruits & légumes", variable=fruitsLegumesVerif,
command=__affichageTableau).grid(sticky='w', pady=ecartFiltre)
251         Checkbutton(filtres, text="Cacher les produits de\nla boulangerie", variable=
boulangerieVerif, command=__affichageTableau).grid(sticky='w', pady=ecartFiltre)
252         Checkbutton(filtres, text="Cacher les produits de\nla boucherie\net poissonnerie",
variable=boucheriePoissonnerieVerif, command=__affichageTableau).grid(sticky='w')
253         Checkbutton(filtres, text="Cacher les produits\n'd'entretien", variable=entretienVerif,
command=__affichageTableau).grid(sticky='w', pady=ecartFiltre)
254         filtres.grid(column=7, row=1, sticky='w')
255
256         stockListe = Stock().listeStocks() # stock récupéré de la base de données
257
258     def __miseAJourPanier(element: dict, action: bool) -> None:
259         """
260         Permet d'ajouter ou de retirer des éléments au panier
261         -> Action
262             -> Vrai : Ajout
263             -> Faux : Retire
264         """
265         # On compte combien de fois l'élément est présent dans le panier
266         nombreDeFoisPresentDansLePanier = 0
267         index = None
268         for idx, elementDansLePanier in enumerate(self.panier):
269             if elementDansLePanier[0] == element:
270                 index = idx # On met à jour l'index
271                 nombreDeFoisPresentDansLePanier = elementDansLePanier[1]
272                 break # on peut quitter la boucle car on a trouvé notre élément
273
274         # On vérifie que on peut encore l'ajouter/retirer
275         if nombreDeFoisPresentDansLePanier == 0 and not action: # pop-up seulement si on veut
retirer un élément pas présent
276             showerror(f"Erreur {self.nomApp}", "Impossible de retirer cet élément au panier
.\nNon présent dans le panier.")
277             return
278         if nombreDeFoisPresentDansLePanier >= element["quantite"] and action: # pop-up
seulement si on veut en rajouter
279             showerror(f"Erreur {self.nomApp}", "Impossible de rajouter cet élément au panier
.\nLimite excédée.")
280             return
281
282         if index != None: # on retire l'ancienne valeur du panier si déjà présente dans le
panier
283             self.panier.pop(index)
284         else: # sinon on définit un index pour pouvoir ajouté la nouvelle valeur à la fin de
la liste
285             index = len(self.panier)
286
287         # On change la valeur dans le panier
288         if action: # si on ajoute
289             nombreDeFoisPresentDansLePanier += 1
290         else: # si on retire
291             nombreDeFoisPresentDansLePanier -= 1
292
293         # On rajoute l'élément avec sa nouvelle quantité seulement s'il y en a
294         if nombreDeFoisPresentDansLePanier > 0:
295             self.panier.insert(index, (element, nombreDeFoisPresentDansLePanier))
296
297         __affichagePanier() # met-à-jour le panier
298
299         for i in range(0, len(stockListe)): # on retire les éléments plus présent dans la liste
300             if stockDisponibleVerif.get() == 1 and stockListe[i]["quantite"] < 1:
301                 stockListe[i] = None
302             elif fruitsLegumesVerif.get() == 1 and stockListe[i]["type"] == "fruits legumes":
303                 stockListe[i] = None
304             elif boulangerieVerif.get() == 1 and stockListe[i]["type"] == "boulangerie":
305                 stockListe[i] = None
306             elif boucheriePoissonnerieVerif.get() == 1 and stockListe[i]["type"] == "boucherie
poissonnerie":
307                 stockListe[i] = None
308             elif entretienVerif.get() == 1 and stockListe[i]["type"] == "entretien":
309                 stockListe[i] = None

```

```

310
311     # Supprime toutes les valeurs 'None' de la liste
312     stockListe = list(filter(None, stockListe))
313
314     ecart = 10 # écart entre les champs
315
316     elementsParPage = 10 # on définit combien d'élément une page peut afficher au maximum
317     pageMax = -(len(stockListe) // elementsParPage) # on définit combien de page il y a
maximum
318
319     if pageMax <= 1:
320         page = 1 # on force la page à être à 1 si il n'y a qu'une page, peut importe l'
argument donnée à la fonction
321
322     limiteIndex = elementsParPage * page # on définit une limite pour ne pas afficher plus d'
éléments qu'il n'en faut par page
323     if len(stockListe) > 0: # si stock non vide
324         # Définition des colonnes
325         Label(self.tableau, text="ID").grid(column=0, row=0, padx=ecart)
326         Label(self.tableau, text="Image").grid(column=1, row=0, padx=ecart)
327         Label(self.tableau, text="Type").grid(column=2, row=0, padx=ecart)
328         Label(self.tableau, text="Nom").grid(column=3, row=0, padx=ecart)
329         Label(self.tableau, text="Quantité").grid(column=4, row=0, padx=ecart)
330         Label(self.tableau, text="Prix unité").grid(column=5, row=0, padx=ecart)
331         Label(self.tableau, text="Action").grid(column=6, row=0, padx=ecart)
332         Separator(self.tableau).grid(column=0, row=0, colspan=7, sticky="sew")
333         Separator(self.tableau).grid(column=0, row=0, colspan=7, sticky="new")
334         for j in range(0, 8):
335             Separator(self.tableau, orient='vertical').grid(column=j, row=0, colspan=2,
sticky="nsw")
336
337         curseur = limiteIndex - elementsParPage # on commence à partir du curseur
338         i = 1 # on commence à 1 car il y a déjà le nom des colonnes en position 0
339         self.imagesStock = [] # on vide la liste si elle contient déjà des images
340         for element in stockListe[curseur:limiteIndex]: # on ignore les éléments avant le
curseur et après la limite
341             Label(self.tableau, text=element["id"]).grid(column=0, row=i, padx=ecart)
342
343             """
344             L'idée est que on a une liste 'images' qui permet de stocker toutes nos images
345             (c'est une limitation de tkinter que de garder nos images en mémoire)
346             Une fois ajouté à la liste, on l'affiche dans notre Label
347             """
348             if Stock().fichierExiste(element["image_url"]): # si l'image existe, utilisation
de la fonction de 'db.py'
349                 self.imagesStock.append(PhotoImage(file = element["image_url"]))
350             else: # si l'image n'existe pas
351                 self.imagesStock.append(PhotoImage(file = "img/default.gif")) # image par
défaut
352             Label(self.tableau, image=self.imagesStock[i - 1]).grid(column=1, row=i, padx=
ecart)
353
354             Label(self.tableau, text=element["type"].capitalize()).grid(column=2, row=i, padx
=ecart)
355             Label(self.tableau, text=element["nom"].capitalize()).grid(column=3, row=i, padx=
ecart)
356             Label(self.tableau, text=element["quantite"]).grid(column=4, row=i, padx=ecart)
357             Label(self.tableau, text=element["prix"]).grid(column=5, row=i,
padx=ecart)
358             # boutons d'actions pour le panier
359             Button(self.tableau, text='+', font=("Arial", 7, "bold"), command=lambda e =
element: __miseAJourPanier(e, True)).grid(column=6, row=i, sticky='n', padx=ecart)
360             Button(self.tableau, text='-', font=("Arial", 7, "bold"), command=lambda e =
element: __miseAJourPanier(e, False)).grid(column=6, row=i, sticky='s', pady=2)
361             for j in range(0, 8):
362                 Separator(self.tableau, orient='vertical').grid(column=j, row=i, colspan
=2, sticky="nsw")
363                 Separator(self.tableau).grid(column=j, row=i, colspan=2, sticky="sew")
364             curseur += 1
365             i += 1
366
367             # Information sur la page actuelle
368             Label(self.tableau, text=f"Page {page}/{pageMax}").grid(column=2, row=i, colspan
=3)
369
370             # Boutons
371             precedent = Button(self.tableau, text="Page précédente", command=lambda:
__affichageTableau(page - 1))
372             precedent.grid(column=0, row=i, colspan=2, sticky='w', padx=ecart, pady=ecart)
373             suivant = Button(self.tableau, text="Page suivante", command=lambda:
__affichageTableau(page + 1))
374             suivant.grid(column=5, row=i, colspan=2, sticky='e', padx=ecart)
375             if page == 1: # si on est a la première page on désactive le bouton précédent
376                 precedent.config(state="disabled")
377             if page == pageMax: # si on est a la dernière page on désactive le bouton suivant

```

```

378         suivant.config(state="disabled")
379     else:
380         Label(self.tableau, text="Il n'y a rien en stock\nEssayez de réduire les critères
dans le filtre.").grid(column=0, row=0, columnspan=7)
381
382     __affichageTableau() # affichage du tableau
383
384     # -> Partie affichage du ticket de caisse
385     ecart = 10
386     ticket = LabelFrame(self.f, text="Ticket de caisse")
387     ticket.grid(column=1, row=1, sticky='n', padx=5)
388
389     Label(ticket, text=f"Date de vente : {date.today().strftime('%Y/%m/%d')}").grid(column=0, row
=0, pady=ecart)
390
391     def __affichagePanier() -> None:
392         """Affiche le panier actuel dans le ticket de caisse."""
393         self.panierAffichage.destroy()
394         self.panierAffichage = Frame(ticket)
395         self.panierAffichage.grid(column=0, row=1, pady=ecart)
396         elementsAchetes = Label(self.panierAffichage)
397         elementsAchetes.grid(column=0, columnspan=2)
398         prixTotal = 0
399         compteurElements = 0
400         for idx, element in enumerate(self.panier):
401             Label(self.panierAffichage, text=f"{{element[0]['id']}} -").grid(column=0, row=idx +
1, sticky='e')
402             if element[1] > 1:
403                 message = f"{{element[1]}}x {{element[0]['nom'].capitalize()}} ({{__formatPrix(element
[0]['prix'])}} | total: {{__formatPrix(element[0]['prix'] * element[1])}})"
404             else:
405                 message = f"{{element[1]}}x {{element[0]['nom'].capitalize()}} ({{__formatPrix(element
[0]['prix'])}})"
406             Label(self.panierAffichage, text=message).grid(column=1, row=idx + 1, sticky='w')
407             prixTotal += (element[0]["prix"] * element[1]) # ajout du prix
408             compteurElements += element[1]
409
410             elementsAchetes.config(text=f"Élément{'s' if compteurElements > 1 else ''} acheté{'s' if
compteurElements > 1 else ''} ({{compteurElements}} :)")
411
412             try: # désactive le bouton si rien n'est dans le panier
413                 if len(self.panier) <= 0:
414                     validationTicketDeCaisseBouton.config(state="disabled")
415                 else:
416                     validationTicketDeCaisseBouton.config(state="active")
417             except NameError: # si pas renseigné, alors = panier vide, déjà désactiver
418                 pass
419
420             Label(self.panierAffichage, text=f"Prix total : {{__formatPrix(prixTotal)}}").grid(column
=0, pady=ecart, columnspan=2)
421
422     __affichagePanier()
423
424     def __validationTicketDeCaisse() -> None:
425         """Lance plusieurs méthodes pour valider le ticket de caisse."""
426         # Met à jour la valeur dans le fichier 'CSV' (statistiques)
427         Stats().miseAJourStatsUtilisateur(id, sum([element[0]["prix"] * element[1] for element in
self.panier]))
428
429         # Informe l'utilisateur que tout est validé
430         showinfo(f"Validation {self.nomApp}", "Ticket de caisse validé !")
431
432         # Retire les éléments renseigné dans le panier du stock
433         for element in self.panier:
434             Stock().reduitQuantiteStock(element[0]["id"], element[1])
435
436         # Remet le panier à 0
437         self.panier = []
438
439         # Met-à-jour le panier et le tableau du stock
440         __affichagePanier()
441         __affichageTableau()
442
443     validationTicketDeCaisseBouton = Button(ticket, text="Valider le\nticket de caisse", font=
self.font, command=__validationTicketDeCaisse, state="disabled")
444     validationTicketDeCaisseBouton.grid(column=0, pady=ecart)
445
446     # -> Partie ajout élément au stock
447     def __ajouterElementStock() -> None:
448         """Ouvre une fenêtre qui permet d'ajouter un nouvel élément à la base de donnée."""
449         """
450         L'enfant ('TopLevel') dépend de la 'Frame' et non du parent ('Tk')
451         pour éviter de resté ouverte meme lorsque le caissier se déconnecte.
452         """
453         enfant = Toplevel(self.f)

```

```

454 enfant.resizable(False, False)
455 enfant.title(f"Ajouter un élément au stock {self.nomApp}")
456
457 def __verification() -> None:
458     """Vérifie si les champs renseignés sont valides."""
459     """
460     La variable 'ok' sert à savoir si la vérification est passée
461     si elle vaut 'True' alors tout est bon,
462     Par contre si elle vaut 'False' alors il y a eu une erreur.
463     Les valeurs 'Entry' qui ne sont pas passés seront dans
464     la liste 'mauvaisChamps'.
465     """
466     ok = True
467     mauvaisChamps = []
468     # vérification pour l'image, on utilise la fonction du fichier 'db.py'
469     if Stock().fichierExiste(image.get()) == False:
470         ok = False
471         mauvaisChamps.append(image)
472     # vérification pour le type
473     if typeElement.get() not in Stock().listeTypes():
474         ok = False
475     # Pas de coloration orange si le type est mauvais parce que on ne peut pas changé
la couleur de fond d'une ComboBox
476     # vérification pour le nom
477     def __nomValide(nom: str) -> bool:
478         """
479         Vérifie si un nom est valide pour le stock.
480         (non vide et pas déjà présent dans la base de donnée)
481         """
482         if len(nom) <= 0:
483             return False
484         if Stock().stockExistant(nom) == True:
485             return False
486         return True
487     if __nomValide(nom.get()) == False:
488         ok = False
489         mauvaisChamps.append(nom)
490     # vérification pour la quantité
491     try:
492         int(quantite.get()) # conversion en int
493     except ValueError: # si la conversion a échoué
494         ok = False
495         mauvaisChamps.append(quantite)
496     # vérification pour le prix
497     try:
498         float(prix.get()) # conversion en float
499     except ValueError: # si la conversion a échoué
500         ok = False
501         mauvaisChamps.append(prix)
502
503     if ok == False:
504         """
505         Tous les champs qui n'ont pas réunies les conditions nécessaires
506         sont mis en orange pendant 3 secondes pour bien comprendre quelles champs
507         sont à modifié.
508
509         La fonction lambda 'remettreCouleur' permet de remettre la couleur initial
510         après les 3 secondes.
511         """
512         remettreCouleur = lambda widget, ancienneCouleur: widget.configure(bg=
ancienneCouleur)
513         for champs in mauvaisChamps:
514             couleur = champs["background"] # couleur d'avant changement
515             champs.configure(bg="orange") # on change la couleur du champs en orange
516             # dans 3 secondes on fait : 'remettreCouleur(champs, couleur)'
517             champs.after(3000, remettreCouleur, champs, couleur)
518         else:
519             """
520             Tous les tests sont passés, on peut ajouter l'utilisateur à la base de donnée
521             Pas besoin de gérer les erreurs lors des casts car on a déjà vérifié que c'était
bien les bons types avant
522             """
523             Stock().ajoutStock(
524                 typeElement.get(),
525                 nom.get(),
526                 int(quantite.get()),
527                 float(prix.get()),
528                 image.get()
529             )
530             __affichageTableau() # met à jour le tableau
531
532     # Champs de saisie
533     # Image
534     Label(enfant, text="Image :").grid(column=0, row=0, sticky='e')
535     image = Entry(enfant)

```

```

536         image.grid(column=1, row=0, sticky='w')
537         def __selectionImage() -> None:
538             """Fonction qui permet de choisir une image dans l'arborescence de fichiers de l'
utilisateur."""
539             try:
540                 chemin = askopenfile(title=f"Choisir une image {self.nomApp}", filetypes=[("
Image GIF", ".gif")])
541                 image.delete(0, "end")
542                 image.insert(0, chemin.name)
543             except AttributeError: # si l'utilisateur n'a pas choisit d'image
544                 pass
545
546         Button(enfant, image=self.dossierImage, command=__selectionImage).grid(column=1, row=0,
sticky='e')
547         # Type (ComboBox)
548         Label(enfant, text="Type :").grid(column=0, row=1, sticky='e')
549         typeElement = Combobox(enfant, values=Stock().listeTypes())
550         # typeElement.current(0) # valeur 0 par défaut
551         typeElement.grid(column=1, row=1, sticky='w')
552         # Nom
553         Label(enfant, text="Nom :").grid(column=0, row=2, sticky='e')
554         nom = Entry(enfant)
555         nom.grid(column=1, row=2, sticky='w')
556         # Quantité
557         Label(enfant, text="Quantité :").grid(column=0, row=3, sticky='e')
558         quantite = Entry(enfant)
559         quantite.grid(column=1, row=3, sticky='w')
560         # Prix à l'unité
561         Label(enfant, text="Prix à l'unité :").grid(column=0, row=4, sticky='e')
562         prix = Entry(enfant)
563         prix.grid(column=1, row=4, sticky='w')
564
565         def __viderChamps() -> None:
566             """Vide tout les champs de leur contenu"""
567             # On récupère toutes les 'Entry' de la fenêtre et on change leur contenu
568             for champ in [widget for typeElement, widget in enfant.children.items() if "entry" in
typeElement]:
569                 champ.delete(0, "end")
570                 champ.update()
571
572         # Boutons
573         Button(enfant, text="Valider", command=__verification).grid(column=0, row=8, columnspan
=3, sticky='w')
574         Button(enfant, text="Vider les champs", command=__viderChamps).grid(column=0, row=8,
columnspan=3)
575         Button(enfant, text="Quitter", command=enfant.destroy).grid(column=0, row=8, columnspan
=3, sticky='e')
576
577         Button(self.f, text="Ajouter un élément\neu stock", font=self.font, command=
__ajouterElementStock).grid(column=1, row=2)
578
579         # -> Partie export des statistiques
580         def __exportation() -> None:
581             """Exporte dans un fichier choisie par l'utilisateur ses statistiques de la journée."""
582             chemin = asksaveasfile(title=f"Exportation des statistiques de {caissier['nom']} {
caissier['prenom']} {self.nomApp}", filetypes=[("Fichier CSV", ".csv")])
583             if chemin == None: # si rien n'a été spécifié on arrête l'exportation
584                 return
585             Stats().exporteCSV(chemin.name, id)
586             Button(self.f, text="Exporter les statistiques", font=self.font, command=__exportation).grid(
column=0, row=2, sticky='e', padx=ecart)
587
588             Button(self.f, text="Se déconnecter", font=self.font, command=self._interfaceConnexion).grid(
column=0, row=2, sticky='w', padx=ecart)
589
590         # -> Boutton pour passer en mode manager si la personne est un manager
591         if caissier["metier"] == 0:
592             Button(self.f, text="Passer en mode Manager", font=self.font, command=lambda: self.
_interfaceManager(id)).grid(column=0, row=2, sticky='w', padx=220)
593
594         def _interfaceManager(self, id: int) -> None:
595             """Affiche l'interface du manager."""
596             manager = Utilisateurs().recuperationUtilisateur(id=id)
597             # Dans le cas où un utilisateur réussi à trouvé cette interface alors qu'il n'a pas le droit,
il sera bloqué
598             if manager["metier"] != 0:
599                 showerror(f"Erreur {self.nomApp}", "Vous ne pouvez pas accéder à cette interface.")
600                 return
601             self.parent.title(f"Manager {manager['nom']} {manager['prenom']} {self.nomApp}")
602             self.dimensionsFenetre(self.parent, 580, 310)
603
604             # Suppression de la dernière Frame
605             self.f.destroy()
606             # Instanciation d'une nouvelle Frame, on va donc ajouter tout nos widgets à cet Frame
607             self.f = Frame(self.parent)

```

```

608     self.f.grid()
609
610     Label(self.f, text="Interface Manager", font=(self.font[0], 20)).grid(column=0, row=0)
611
612     Button(self.f, text="Se déconnecter", font=self.font, command=self._interfaceConnexion).grid(
        column=1, row=0, padx=50)
613
614     Label(self.f).grid(row = 1, pady=10) # séparateur
615
616     def __ajouterUtilisateur(metier: int) -> None:
617         """Permet de créer un nouvel utilisateur, manager ('metier = 0') et caissier ('metier =
        1')."""
618         """
619         L'enfant ('TopLevel') dépend de la 'Frame' et non du parent ('Tk')
620         pour éviter de resté ouverte meme lorsque le manager se déconnecte.
621         """
622         enfant = Toplevel(self.f)
623         enfant.resizable(False, False)
624         enfant.title(f"Ajouter un {'manager' if metier == 0 else 'caissier'} {self.nomApp}")
625
626     def __verification() -> None:
627         """Vérifie si les champs renseignées sont valides."""
628         """
629         Les valeurs 'Entry' qui ne sont pas passés seront dans
630         la liste 'mauvaisChamps'.
631         Si la liste 'mauvaisChamps' contient un élément alors un test n'est pas ok.
632         """
633         mauvaisChamps = []
634         # vérification pour le nom d'utilisateur
635         if self.utilisateurCorrect(pseudo.get())[0] == False or Utilisateurs().
        utilisateurExistant(pseudo.get()) == True:
636             mauvaisChamps.append(pseudo)
637         # vérification pour le mot de passe
638         if self.motDePasseCorrect(passe.get())[0] == False:
639             mauvaisChamps.append(passe)
640         # vérification pour le nom
641         if self.nomCorrect(nom.get()) == False:
642             mauvaisChamps.append(nom)
643         # vérification pour le prénom
644         if self.prenomCorrect(prenom.get()) == False:
645             mauvaisChamps.append(prenom)
646         # vérification pour la date de naissance
647         if self.naissanceCorrect(naissance.get()) == False:
648             mauvaisChamps.append(naissance)
649         # vérification pour l'adresse
650         if self.adresseCorrect(adresse.get()) == False:
651             mauvaisChamps.append(adresse)
652         # vérification pour le code postal
653         if self.postalCorrect(postal.get()) == False:
654             mauvaisChamps.append(postal)
655
656         if len(mauvaisChamps) != 0:
657             """
658             Tous les champs qui n'ont pas réunies les conditions nécessaires
659             sont mis en orange pendant 3 secondes pour bien comprendre quelles champs
660             sont à modifié.
661
662             La fonction lambda 'remettreCouleur' permet de remettre la couleur initial
663             après les 3 secondes.
664             """
665             remettreCouleur = lambda widget, ancienneCouleur: widget.configure(bg=
        ancienneCouleur)
666             for champs in mauvaisChamps:
667                 couleur = champs["background"] # couleur d'avant changement
668                 champs.configure(bg="orange") # on change la couleur du champs en orange
669                 # dans 3 secondes on fait: 'remettreCouleur(champs, couleur)'
670                 champs.after(3000, remettreCouleur, champs, couleur)
671             else:
672                 # Tous les tests sont passés, on peut ajouter l'utilisateur à la base de donnée
673                 Utilisateurs().ajoutUtilisateur(
674                     pseudo.get(),
675                     passe.get().strip(),
676                     metier,
677                     nom.get(),
678                     prenom.get(),
679                     naissance.get(),
680                     adresse.get(),
681                     int(postal.get()), # pas besoin de gérer d'erreur lors du cast car on a
        vérifié avant que c'était bien une suite de chiffre
        )
682                 __ajouterUtilisateursListe(listeUtilisateurs) # met à jour la liste
683
684         # Champs de saisie
685         # Nom d'utilisateurs
686         Label(enfant, text="Nom d'utilisateur :").grid(column=0, row=0, sticky='e')
687

```

```

688     Label(enfant, text="Pas de caractères spéciaux", font=("Arial", 10, "italic")).grid(
column=2, row=0, sticky='w')
689     pseudo = Entry(enfant)
690     pseudo.grid(column=1, row=0, sticky='w')
691     # Mot de passe
692     Label(enfant, text="Mot de passe :").grid(column=0, row=1, sticky='e')
693     Label(enfant, text="1 majuscule, miniscule et caractère spécial minimum", font=("Arial",
10, "italic")).grid(column=2, row=1, sticky='w')
694     passe = Entry(enfant)
695     passe.grid(column=1, row=1, sticky='w')
696     # Nom
697     Label(enfant, text="Nom :").grid(column=0, row=2, sticky='e')
698     Label(enfant, text="Pas de caractères spéciaux", font=("Arial", 10, "italic")).grid(
column=2, row=2, sticky='w')
699     nom = Entry(enfant)
700     nom.grid(column=1, row=2, sticky='w')
701     # Prénom
702     Label(enfant, text="Prénom :").grid(column=0, row=3, sticky='e')
703     Label(enfant, text="Pas de caractères spéciaux", font=("Arial", 10, "italic")).grid(
column=2, row=3, sticky='w')
704     prenom = Entry(enfant)
705     prenom.grid(column=1, row=3, sticky='w')
706     # Date de naissance
707     Label(enfant, text="Date de naissance :").grid(column=0, row=4, sticky='e')
708     Label(enfant, text="Format : AAAA/MM/JJ", font=("Arial", 10, "italic")).grid(column=2,
row=4, sticky='w')
709     naissance = Entry(enfant)
710     naissance.grid(column=1, row=4, sticky='w')
711     # Adresse
712     Label(enfant, text="Adresse").grid(column=0, row=5, sticky='e')
713     adresse = Entry(enfant)
714     adresse.grid(column=1, row=5, sticky='w')
715     # Code postal
716     Label(enfant, text="Code postal :").grid(column=0, row=6, sticky='e')
717     Label(enfant, text="5 chiffres", font=("Arial", 10, "italic")).grid(column=2, row=6,
sticky='w')
718     postal = Entry(enfant)
719     postal.grid(column=1, row=6, sticky='w')
720
721     def __viderChamps() -> None:
722         """Vide tout les champs de leur contenu"""
723         # On récupère toutes les 'Entry' de la fenêtre et on change leur contenu
724         for champ in [widget for typeElement, widget in enfant.children.items() if "entry" in
typeElement]:
725             champ.delete(0, "end")
726             champ.update()
727
728     # Boutons
729     Button(enfant, text="Valider", command=__verification).grid(column=0, row=8, columnspan
=3, sticky='w')
730     Button(enfant, text="Vider les champs", command=__viderChamps).grid(column=0, row=8,
columnspan=3)
731     Button(enfant, text="Quitter", command=enfant.destroy).grid(column=0, row=8, columnspan
=3, sticky='e')
732
733     def __retirerUtilisateur(metier: int) -> None:
734         """Permet de supprimer un utilisateur existant, manager ('metier = 0') et caissier ('
metier = 1')."""
735         enfant = Toplevel(self.f) # cf. l'explication dans '__ajouterUtilisateur'
736         enfant.resizable(False, False)
737         enfant.title(f"Retirer un {'manager' if metier == 0 else 'caissier'} {self.nomApp}")
738
739     # Liste des utilisateurs
740     Label(enfant, text=f"Liste des {'manager' if metier == 0 else 'caissier'}", font=self.
font).grid(column=0, row=0) # titre
741     # On définit une barre pour pouvoir scroller dans la liste
742     scroll_retirer = Scrollbar(enfant)
743     scroll_retirer.grid(column=1, row=1, sticky="nse")
744     # On définit notre liste et on la lie à notre 'Scrollbar'
745     listeUtilisateurs_retirer = Listbox(enfant, width=25, height=4, yscrollcommand=
scroll_retirer.set)
746     scroll_retirer.config(command=listeUtilisateurs_retirer.yview) # scroll à la verticale
dans notre liste
747     # On ajoute nos utilisateurs à notre liste
748     __ajouterUtilisateursListe(listeUtilisateurs_retirer, metier)
749     listeUtilisateurs_retirer.grid(column=0, row=1)
750     # On affiche l'utilisateur quand on double-clique dessus
751
752     def __suppressionUtilisateur() -> None:
753         """Supprime l'utilisateur actuellement sélectionné dans la liste"""
754         element = listeUtilisateurs_retirer.curselection()
755         if len(element) == 0: # si aucun élément n'est sélectionné
756             showwarning(f"Attention {self.nomApp}", "Aucun utilisateur n'a été sélectionné."
)
757         else:

```



```

758         utilisateur = listeUtilisateurs_retirer.get(listeUtilisateurs_retirer.
curselection() [0]).split('(')[0][:-1]
759         reponse = askyesno(f"Confirmation {self.nomApp}", f"Voulez vous supprimer {
utilisateur} ?")
760         if reponse == True:
761             Utilisateurs().suppressionUtilisateurs(utilisateur)
762             __ajouterUtilisateursListe(listeUtilisateurs_retirer) # met à jour la liste
dans la fenêtre de suppression
763             __ajouterUtilisateursListe(listeUtilisateurs) # met à jour la liste dans l'
interface principale
764             showinfo(f"Information {self.nomApp}", f"Utilisateur {utilisateur} supprimé.
")
765
766         # Boutons
767         Button(enfant, text="Supprimer", command=___suppressionUtilisateur).grid(column=0, row=8,
columnspan=3, sticky='w')
768         Button(enfant, text="Quitter", command=enfant.destroy).grid(column=0, row=8, columnspan
=3, sticky='e')
769
770     def __afficherInformationsUtilisateur(_) -> None:
771         """Permet d'afficher les informations d'un utilisateur"""
772         element = listeUtilisateurs.curselection()
773         if len(element) == 0: # si aucun élément n'est sélectionné
774             return
775         """
776         On split le champs car dans la liste on affiche le métier entre
777         parenthèses et on doit donner que le nom d'utilisateur à
778         la fonction 'recuperationUtilisateur', aussi on retire le dernier
779         caractère avec[:-1] car c'est un espace.
780         """
781         utilisateur = Utilisateurs().recuperationUtilisateur(pseudo=listeUtilisateurs.get(element
[0]).split('(')[0][:-1])
782         enfant = Toplevel(self.f) # cf. l'explication dans '__ajouterUtilisateur'
783         enfant.resizable(False, False)
784         enfant.title(f"{utilisateur['nom']} {utilisateur['prenom']} {self.nomApp}")
785
786         # Informations sur l'utilisateur
787         frameInfos = LabelFrame(enfant, text="Informations utilisateur", font=self.font)
788         frameInfos.grid(column=0, row=0, sticky='n', padx=5)
789         utilisateur["metier"] = "Manager" if utilisateur["metier"] == 0 else "Caissier"
790         del utilisateur["passe"] # le manager ne doit pas connaître le mot de passe de l'
utilisateur
791         for idx, cle in enumerate(utilisateur):
792             if cle == "id": # on ignore l'ID
793                 continue
794             cleAffichage = cle.capitalize()
795             Label(frameInfos, text=f"{cleAffichage} :").grid(column=0, row=idx + 1, sticky='e')
796             Label(frameInfos, text=utilisateur[cle]).grid(column=1, row=idx + 1, sticky='w')
797
798         frameSuivi = LabelFrame(enfant, text="Histogramme des ventes", font=self.font)
799         frameSuivi.grid(column=1, row=0, sticky="ns", padx=5)
800
801     def ___actualisationCanvas() -> None:
802         """Affiche l'histogramme des vente d'un utilisateur dans un canvas."""
803         donnees = Stats().recuperationDonneesCSV(utilisateur['id'])
804         if len(donnees) <= 0:
805             histogramme.create_text(10, 10, anchor='w', text="Aucun résultat récemment
enregistré")
806         else:
807             # Les dates dans le fichier CSV ne sont pas dans l'ordre
808             # On retire l'ID et le pseudo du dictionnaire
809             donnees.pop("id")
810             donnees.pop("pseudo")
811             ecart = 10
812             couleurs = [
813                 "CadetBlue3",
814                 "HotPink2",
815                 "IndianRed1",
816                 "MediumPurple2",
817                 "burlywood2",
818                 "brown3",
819                 "chocolate1",
820                 "goldenrod2"
821             ]
822
823         # On remplace tous les prix 'None' par '0.' (au cas où il y est des valeurs vide
dans le 'CSV')
824         for date, prix in donnees.items():
825             if prix == None:
826                 donnees[date] = 0.
827
828         # On récupère la plus grosse vente
829         maxVente = 0 # par défaut la meilleur vente est de 0
830         for prix in donnees.values():
831             prix = float(prix)

```

```

832         if prix > maxVente: # si on trouve une valeur plus grande
833             maxVente = prix
834
835         for date in sorted(donnees.keys()): # on regarde les dates dans l'ordre
836             # Affichage de la date
837             histogramme.create_text(ecart + 10, 60, anchor='w', text=date, font=("Arial",
838             8), angle=90)
839             # Affichage de la barre
840             hauteur = 190 - (float(donnees[date]) * 100) / maxVente # calcul de la
hauteur en fonction de la plus grosse vente
841             # On fait '- 20' au résultat pour allonger la barre, aussi on met une barre
de '2' pixel quand valeur petite
842             histogramme.create_rectangle(ecart, 180, ecart + 15, hauteur - 20 if hauteur
< 180 else 178, fill=couleurs.pop())
843             # Affichage du montant
844             histogramme.create_text(ecart, 190, anchor='w', text=donnees[date], font=("
Arial", 7))
845             ecart += 33
846
847             histogramme = Canvas(frameSuivi, width=270, height=200)
848             histogramme.grid()
849
850             ___actualisationCanvas()
851
852             Button(enfant, text="Quitter", command=enfant.destroy).grid(column=0, row=1, columnspan
=2)
853
854             Button(self.f, text="Ajouter un caissier", font=self.font, command=lambda:
__ajouterUtilisateur(1)).grid(column=0, row=2)
855             Button(self.f, text="Retirer un caissier", font=self.font, command=lambda:
__retirerUtilisateur(1)).grid(column=1, row=2)
856
857             Label(self.f).grid(row = 3, pady=10) # séparateur
858
859             # Liste des utilisateurs
860             managerVerif = IntVar(self.f) # filtre pour afficher ou non les managers dans la liste
caissierVerif = IntVar(self.f) # filtre pour afficher ou non les caissiers ou non dans la
liste
861
862             caissierVerif.set(1) # par défaut on affiche que les caissiers
863
864             def __ajouterUtilisateursListe(liste: Listbox, force: int = None) -> None:
865                 """
866                 Ajoute des utilisateurs à la liste du Manager.
867                 -> metier = 0 : manager uniquement
868                 -> metier = 1 : caissier uniquement
869                 -> metier = 2 : manager et caissier
870                 """
871                 liste.delete(0, "end") # vidé la liste des utilisateurs
872                 if force: # si 'force' n'est pas 'None', alors on force l'utilisation d'un métier
873                     metier = force
874                 else: # sinon on fait une vérification normale en fonction des filtres de l'interface
manager
875                     if managerVerif.get() == 1:
876                         if caissierVerif.get() == 1:
877                             metier = None # on affiche les 2
878                         else:
879                             metier = 0 # on affiche seulement les managers
880                     else:
881                         metier = 1 # on affiche les caissiers
882                         if caissierVerif.get() == 0: # rien est coché, on revient à la configuration par
défaut (caissiers uniquement)
883                             metier = 1
884                             caissierVerif.set(1)
885
886                     if metier == None: # on ajoute tous les utilisateurs
887                         for idx, utilisateur in enumerate(Utilisateurs().listUtilisateurs()):
888                             liste.insert(idx, f"{utilisateur[0]} ({'manager' if utilisateur[1] == 0 else '
caissier'})")
889                     elif metier == 0: # on ajoute que les managers
890                         for idx, utilisateur in enumerate(Utilisateurs().listUtilisateurs()):
891                             if utilisateur[1] == metier:
892                                 liste.insert(idx, f"{utilisateur[0]} ({'manager' if utilisateur[1] == 0 else
'caissier'})")
893                     elif metier == 1: # on ajoute que les caissiers
894                         for idx, utilisateur in enumerate(Utilisateurs().listUtilisateurs()):
895                             if utilisateur[1] == metier:
896                                 liste.insert(idx, f"{utilisateur[0]} ({'manager' if utilisateur[1] == 0 else
'caissier'})")
897                     else: # ce cas est là au cas où mais n'est pas sensé être appelé
898                         raise NameError("Métier inconnu.")
899
900             # Label d'information
901             Label(self.f, text="")
902             Double-cliquez sur un

```

```

903         utilisateur de la liste
904         pour obtenir des informations
905         supplémentaire à son sujet.
906         """ , justify="right").grid(column=1, row=4, rowspan=2, sticky="e")
907
908     Label(self.f, text="Liste des utilisateurs", font=self.font).grid(column=0, row=4) # titre
909     # On définit une barre pour pouvoir scroller dans la liste
910     scrollbar = Scrollbar(self.f)
911     scrollbar.grid(column=0, row=5, sticky="nse")
912     # On définit notre liste et on la lie à notre 'Scrollbar'
913     listeUtilisateurs = Listbox(self.f, width=25, height=4, yscrollcommand=scrollbar.set)
914     scrollbar.config(command=listeUtilisateurs.yview) # scrollbar à la verticale dans notre liste
915     # On ajoute nos utilisateurs à notre liste
916     __ajouterUtilisateursListe(listeUtilisateurs)
917     listeUtilisateurs.grid(column=0, row=5)
918     listeUtilisateurs.bind('<Double-Button>', __afficherInformationsUtilisateur) # on affiche l'
utilisateur quand on double-clique dessus
919
920     # Filtre pour la liste
921     Label(self.f, text="Filtre", font=self.font).grid(column=1, row=4, sticky='w', padx=10) #
titre
922     filtres = Frame(self.f) # Morceau qui va contenir nos checkbutton
923     filtres.grid(column=1, row=4, rowspan=2, sticky='w')
924     Checkbutton(filtres, text="Manager", variable=managerVerif, command=lambda:
__ajouterUtilisateursListe(listeUtilisateurs)).grid(sticky='w')
925     Checkbutton(filtres, text="Caissier", variable=caissierVerif, command=lambda:
__ajouterUtilisateursListe(listeUtilisateurs)).grid(sticky='w')
926
927     Button(self.f, text="Passer en mode caissier", font=self.font, command=lambda: self.
_interfaceCaissier(id)).grid(column=0, row=6, columnspan=3, pady=10)
928
929 if __name__ == "__main__":
930     """Application "GesMag" pour le module de Programmation d'interfaces (2021-2022)"""
931     """
932     Si presentation = True alors une base de donnée par défaut sera généré.
933     Si presentation = False ou n'est même pas mentionné, alors aucune base de donnée par défaut ne
sera généré.
934     """
935     GesMag(presentation = True).demarrer()

```

2.2 db.py, gère la communication avec la base de donnée en sa globalité

```

1  import sqlite3
2
3  class BaseDeDonnees:
4      """Gère la base de donnée."""
5      def __init__(self, urlBaseDeDonnee: str):
6          self.connexion = self.creerConnexion(urlBaseDeDonnee)
7
8      def creerConnexion(self, path: str):
9          """Connexion à une base de donnée SQLite."""
10         if not self.fichierExiste(path): # si l base de donnée n'existe pas
11             open(path, 'x') # on la créer
12         try:
13             connexion = sqlite3.connect(path)
14         except sqlite3.Error as e:
15             print(e) # on affiche l'erreur
16             connexion = None # et renvoie None
17         return connexion
18
19     def fichierExiste(self, path: str) -> bool:
20         """Vérifie qu'un fichier existe."""
21         try: # on essaie d'ouvrir le fichier
22             open(path, 'r')
23         except FileNotFoundError: # si le fichier n'existe pas
24             return False
25         else: # si le fichier existe
26             return True
27
28     def requete(self, requete: str, valeurs = None) -> tuple:
29         """Envois une requête vers la base de données."""
30         try:
31             curseur = self.connexion.cursor()
32             if valeurs: # s'il y a des valeurs alors on lance la commande 'execute' avec ses
dernières
33                 if type(valeurs) not in [list, tuple]: # si la valeur c'est juste une chaîne de
caractère (par exemple), alors la converti en liste
34                     valeurs = [valeurs]
35                 curseur.execute(requete, valeurs)
36             else: # sinon on lance juste la requête
37                 curseur.execute(requete)
38             self.connexion.commit() # applique les changements à la base de donnée

```

```

39         return (curseur, curseur.lastrowid) # renvoie le curseur et l'ID de l'élément modifié
40     except sqlite3.Error as e: # s'il y a eu une erreur SQLite
41         print(e)
42
43     def affichageResultat(self, curseur: tuple) -> list:
44         """Affiche le résultat d'une requête."""
45         tableau = []
46         if curseur == None: # si le curseur est vide il n'y a rien a affiché (tableau vide)
47             return tableau
48         lignes = curseur[0].fetchall() # sinon on récupère les éléments
49         for ligne in lignes:
50             tableau.append(ligne) # on les ajoute au tableau
51         return tableau # on le renvoie
52
53     def affichageResultatDictionnaire(self, cles, curseur: sqlite3.Cursor) -> dict:
54         """
55         Même but que 'affichageResultat()' mais avec
56         les clés qui correspondent aux valeurs.
57         """
58         valeurs = self.affichageResultat(curseur)
59         if len(valeurs) == 0:
60             valeurs = []
61         else:
62             valeurs = valeurs[0]
63         if type(cles) not in [list, tuple]:
64             cles = [cles]
65         if len(cles) != len(valeurs):
66             raise IndexError # il y a pas autant de clés que de valeurs
67         return dict(zip(cles, valeurs))

```

2.3 users.py, implante la base de donnée pour les utilisateurs

```

1  from db import BaseDeDonnees
2
3  class Utilisateurs(BaseDeDonnees):
4      """Gère une table "utilisateurs" pour une base de donnée donné."""
5      def __init__(self):
6          super().__init__(r"db.sqlite3") # connexion à la base de donnée
7
8      def creationTable(self, presentation: bool = False) -> None:
9          """Créer la table qui stocker les utilisateurs."""
10         requete = """
11             CREATE TABLE IF NOT EXISTS utilisateurs (
12                 id INTEGER PRIMARY KEY,
13                 pseudo TEXT,
14                 passe TEXT,
15                 metier INTEGER,
16                 nom TEXT,
17                 prenom TEXT,
18                 naissance TEXT,
19                 adresse TEXT,
20                 postal INTEGER
21             );
22         """
23         self.requete(requete)
24         # Ajout d'un utilisateur par défaut si aucun utilisateur n'existe dans la base de donnée
25         if len(self.listUtilisateurs()) == 0 and presentation:
26             self.ajoutUtilisateur(
27                 pseudo="admin",
28                 passe="P@ssword",
29                 metier=0,
30                 nom="Admin",
31                 prenom="Système",
32                 naissance="2000/10/09",
33                 adresse="12 Rue de Montmartre",
34                 postal=46800
35             )
36             print("-- Compte par défaut --\nNom d'utilisateur: admin\nMot de passe: P@ssword")
37
38         def ajoutUtilisateur(self, pseudo: str, passe: str, metier: int, nom: str, prenom: str, naissance
39         : str, adresse: str, postal: int) -> list:
40             """Ajoute un utilisateur et retourne l'ID de ce dernier."""
41             requete = """
42                 INSERT INTO utilisateurs (
43                     pseudo, passe, metier, nom, prenom, naissance, adresse, postal
44                 ) VALUES (
45                     ?, ?, ?, ?, ?, ?, ?, ?
46                 );
47             """
48             self.requete(requete, [pseudo.lower(), passe, metier, nom.upper(), prenom.capitalize(),
49             naissance, adresse, postal])
50             return self.affichageResultat(self.requete("SELECT last_insert_rowid();"))

```

```

49
50 def suppressionUtilisateurs(self, pseudo: str) -> None:
51     """Supprime un utilisateur."""
52     requete = """
53         DELETE FROM utilisateurs
54         WHERE pseudo = ?
55     """
56     self.requete(requete, pseudo)
57
58 def verificationIdentifiants(self, pseudo: str, motDePasse: str) -> tuple:
59     """
60     Retourne l'ID de l'utilisateur si trouvé dans la base de donnée ainsi
61     que son métier ('tuple'), sinon renvoie '(0,)'
62     """
63     requete = """
64         SELECT id, metier FROM utilisateurs
65         WHERE pseudo = ? AND passe = ?
66     """
67     reponseBaseDeDonnee = self.affichageResultat(self.requete(requete, [pseudo.lower(),
68 motDePasse]))
69     if len(reponseBaseDeDonnee) == 0: # si les identifiants renseignés sont mauvais
70         return (0,)
71     return reponseBaseDeDonnee[0]
72
73 def listUtilisateurs(self) -> list:
74     """Retourne la liste des nom d'utilisateurs (avec leur métier)."""
75     requete = """
76         SELECT pseudo, metier FROM utilisateurs
77     """
78     return self.affichageResultat(self.requete(requete))
79
80 def recuperationUtilisateur(self, id: int = None, pseudo: str = None) -> dict:
81     """Retourne les informations d'un utilisateur grâce à son ID ou son pseudo (ID en priorité).
82     """
83     recuperation = [
84         "id",
85         "pseudo",
86         "passe",
87         "metier",
88         "nom",
89         "prenom",
90         "naissance",
91         "adresse",
92         "postal",
93     ]
94     if not id: # si la variable 'id' n'est pas définie
95         if not pseudo: # si seul la variable 'pseudo' n'est pas définie
96             raise ValueError # Aucun utilisateur renseigné
97         else: # si un pseudo est renseigné, c'est ce qu'on va utilisé
98             requete = f"""
99                 SELECT {"", ".join(recuperation)} FROM utilisateurs
100                WHERE pseudo = ?
101            """
102             utilisateur = pseudo
103         else: # si un id est renseigné, c'est ce qu'on va utilisé
104             requete = f"""
105                 SELECT {"", ".join(recuperation)} FROM utilisateurs
106                WHERE id = ?
107            """
108             utilisateur = id
109     return self.affichageResultatDictionnaire(recuperation, self.requete(requete, utilisateur))
110
111 def utilisateurExistant(self, utilisateur: str) -> bool:
112     """Vérifie si l'utilisateur donnée existe déjà dans la base de donnée."""
113     requete = """
114         SELECT EXISTS (
115             SELECT 1 FROM utilisateurs
116             WHERE pseudo = ?
117         )
118     """
119     return True if self.affichageResultat(self.requete(requete, utilisateur.lower()))[0][0] == 1
120     else False

```

2.4 stock.py, implante la base de donnée pour le stock

```

1 from random import randint, uniform
2
3 from db import BaseDeDonnees
4
5 class Stock(BaseDeDonnees):
6     """Gère une table "stock" pour une base de donnée donné."""
7     def __init__(self):

```

```

8     super().__init__(r"db.sqlite3") # connexion à la base de donnée
9
10    def creationTable(self, presentation: bool = False) -> None:
11        """Créer la table qui stocker les stocks."""
12        requete = """
13            CREATE TABLE IF NOT EXISTS stocks (
14                id INTEGER PRIMARY KEY,
15                type TEXT,
16                nom TEXT,
17                quantite INTEGER,
18                prix REAL,
19                image_url TEXT
20            );
21        """
22        self.requete(requete)
23        # Ajout d'un stock par défaut si aucun stock n'existe dans la base de donnée
24        if len(self.listeStocks()) == 0 and presentation:
25            # Créer un dictionnaire d'éléments pour mieux voir ce que l'on ajoute à la base de donnée
26            default = {
27                "fruits legumes": [
28                    ("banane", "img/banane.gif"),
29                    ("orange", "img/orange.gif"),
30                    ("betterave", "img/betterave.gif"),
31                    ("carottes", "img/carottes.gif"),
32                    ("tomates", "img/tomates.gif"),
33                    ("citron", "img/citron.gif"),
34                    ("kiwi", "img/kiwi.gif"),
35                    ("clementine", "img/clementine.gif"),
36                    ("pomme", "img/pomme.gif"),
37                    ("avocat", "img/avocat.gif")
38                ],
39                "boulangerie": [
40                    ("brownie", "img/brownie.gif"),
41                    ("baguette", "img/baguette.gif"),
42                    ("pain au chocolat", "img/pain_au_chocolat.gif"),
43                    ("croissant", "img/croissant.gif"),
44                    ("macaron", "img/macaron.gif"),
45                    ("millefeuille", "img/millefeuille.gif"),
46                    ("paris-brest", "img/paris-brest.gif"),
47                    ("opera", "img/opera.gif"),
48                    ("fraisier", "img/fraisier.gif"),
49                    ("eclair", "img/eclair.gif")
50                ],
51                "boucherie poissonnerie": [
52                    ("saucisson", "img/saucisson.gif"),
53                    ("côte de boeuf", "img/cote_de_boeuf.gif"),
54                    ("langue de boeuf", "img/langue_de_boeuf.gif"),
55                    ("collier de boeuf", "img/collier_de_boeuf.gif"),
56                    ("entrecote", "img/entrecote.gif"),
57                    ("cabillaud", "img/cabillaud.gif"),
58                    ("saumon", "img/saumon.gif"),
59                    ("colin", "img/colin.gif"),
60                    ("bar", "img/bar.gif"),
61                    ("dorade", "img/dorade.gif")
62                ],
63                "entretien": [
64                    ("nettoyant air comprimé", "img/nettoyant_air_comprime.gif"),
65                    ("nettoyage anti-bactérien", "img/nettoyage_anti-bacterien.gif"),
66                    ("nettoyant pour écran", "img/nettoyant_pour_ecran.gif"),
67                    ("nettoyant pour lunettes", "img/nettoyant_pour_lunettes.gif"),
68                    ("pioche", "img/pioche.gif"),
69                    ("pelle", "img/pelle.gif"),
70                    ("lampe torche", "img/lampe_torche.gif"),
71                    ("gants", "img/gants.gif"),
72                    ("éponge", "img/eponge.gif"),
73                    ("essuie-tout", "img/essuie-tout.gif")
74                ]
75            }
76
77            # Ajoute le dictionnaire précédemment créer à la base de donnée avec un prix et une
78            # quantité aléatoire
79            for type in default:
80                for element in default[type]:
81                    self.ajoutStock(type, element[0], randint(0, 10), round(uniform(2., 30.), 2),
82                    element[1])
83
84    def ajoutStock(self, typeElement: str, nom: str, quantite: int, prix: float, imageURL: str) ->
85    list:
86        """Ajoute un élément dans le stock et retourne l'ID de ce dernier."""
87        requete = """
88            INSERT INTO stocks (
89                type, nom, quantite, prix, image_url
90            ) VALUES (
91                ?, ?, ?, ?, ?
92            );

```

```

90         """
91         self.requete(requete, [typeElement.lower(), nom.lower(), quantite, prix, imageURL])
92         return self.affichageResultat(self.requete("SELECT last_insert_rowid();"))
93
94     def reduitQuantiteStock(self, id: int, quantiteARetirer: int) -> None:
95         """Retire une quantité d'un élément du stock et met-à-jour la base de donnée."""
96         requeteA = """
97             SELECT quantite FROM stocks
98             WHERE id = ?
99             """
100         quantiteActuelle: int = self.affichageResultat(self.requete(requeteA, id))[0][0]
101         if quantiteActuelle <= quantiteARetirer: # il ne reste plus rien
102             quantiteFinale = 0
103         else: # il reste quelque chose
104             quantiteFinale = quantiteActuelle - quantiteARetirer
105         # On met à jour la quantité de l'élément dans la base de donnée
106         requeteB = """
107             UPDATE stocks
108             SET quantite = ?
109             WHERE id = ?
110             """
111         self.requete(requeteB, [quantiteFinale, id])
112
113     def listeStocks(self) -> list:
114         """Retourne la liste des éléments en stock sous forme de dictionnaire."""
115         recuperation = [
116             "id",
117             "type",
118             "nom",
119             "quantite",
120             "prix",
121             "image_url"
122         ]
123         requete = f"""
124             SELECT {", ".join(recuperation)} FROM stocks
125             """
126         return [dict(zip(recuperation, element)) for element in self.affichageResultat(self.requete(
127             requete))]
128
129     def stockExistant(self, stock: str) -> bool:
130         """Vérifie si le stock donnée existe déjà dans la base de donnée."""
131         requete = """
132             SELECT EXISTS (
133                 SELECT 1 FROM stocks
134                 WHERE nom = ?
135             )
136             """
137         return True if self.affichageResultat(self.requete(requete, stock.lower()))[0][0] == 1 else
138             False
139
140     def listeTypes(self) -> list:
141         """Renvoie la liste des types disponibles dans la base de donnée."""
142         requete = """
143             SELECT type FROM stocks
144             """
145         res = []
146         for i in self.affichageResultat(self.requete(requete)):
147             if i[0] not in res:
148                 res.append(i[0])
149         return res

```

2.5 stats.py, implante la gestion des statistiques et son export en format CSV

```

1  import csv
2
3  from datetime import date, timedelta
4
5  from users import Utilisateurs
6
7  class Stats():
8      """Gère les statistiques et son export en format CSV."""
9      def __init__(self):
10         self.formatDate = "%Y/%m/%d"
11
12     def datesDisponibles(self) -> list:
13         """Renvoie les dates disponibles pour l'entête du fichier 'CSV'."""
14         datesPossibles = []
15         dateAujourdHui = date.today() - timedelta(days=7)
16         for _ in range(0, 8):
17             datesPossibles.append(dateAujourdHui.strftime(self.formatDate))
18             dateAujourdHui = dateAujourdHui + timedelta(days=1)
19         return datesPossibles

```

```

20
21 def creationCSV(self, force: bool = False) -> None:
22     """
23     Créer le fichier 'CSV' qui stockera les statistiques pour tous les utilisateurs.
24
25     Possibilité de forcer la création (c-à-d même si le fichier existe déjà) en renseignant
26     'force = True'
27     """
28     if not Utilisateurs().fichierExiste("stats.csv") or force:
29         with open("stats.csv", 'w') as f:
30             fichier = csv.writer(f)
31             fichier.writerow(["id", "pseudo"] + self.datesDisponibles())
32
33 def miseAJourStatsUtilisateur(self, utilisateurID: int, prix: float) -> None:
34     """
35     Récupère le prix d'une transaction et l'ajoute au total d'un utilisateur.
36
37     - s'il y a déjà une valeur dans la base de donnée correspondant à la date du jour,
38       on met à jour cette valeur en l'additionnant avec le nouveaux prix
39     - s'il n'avait pas de valeur à cette date:
40       - si l'utilisateur n'est pas dans le fichier, on rajoute une ligne avec le prix
41       - si l'utilisateur est présent mais aucun prix n'est fixé pour la date du jour,
42         on rajoute le prix sur la ligne de l'utilisateur déjà existante
43
44     On remplit les espaces vides par la valeur '0' (car aucun chiffre n'a été fait ce jour là,
45     car aucune information n'était renseignée).
46     """
47     self.miseAJourDatesCSV() # met-à-jour les dates du fichier 'CSV'
48
49     # Mets-à-jour le 'CSV' avec le nouveau prix...
50     aujourdHui = date.today().strftime(self.formatDate)
51     with open("stats.csv", 'r') as f:
52         fichier = list(csv.reader(f))
53         # On récupère la colonne pour aujourd'hui
54         index = 0
55         locationDate = None # note l'index de la colonne de la date dans le fichier
56         for nomColonne in fichier[0]: # on regarde l'entête
57             if nomColonne == aujourdHui: # on regarde si la colonne correspond à la date du jour
58                 locationDate = index # on note l'entête
59                 index += 1
60         if locationDate == None: # ne devrait pas arrivé car on mets à jour les dates du 'CSV'
61             avant de lire le fichier
62                 raise IndexError("Date du jour non trouvé dans le fichier csv.")
63
64         utilisateur = Utilisateurs().recuperationUtilisateur(utilisateurID) # on récupère les
65         infos de l'utilisateur
66         # Vérification si l'utilisateur est déjà présent dans le fichier 'CSV'
67         locationUtilisateur = None # note l'index de la ligne de l'utilisateur dans le fichier
68         for idx, location in enumerate(fichier):
69             if location[0] == str(utilisateurID):
70                 locationUtilisateur = idx
71             if locationUtilisateur == None: # si l'utilisateur n'est pas présent dans le fichier
72                 # on rajoute la ligne
73                 fichier += [[utilisateurID, utilisateur["pseudo"]] + ['0' for _ in range(0,
74                 locationDate - 2)] + [prix]]
75             else: # si déjà présent dans le fichier
76                 try:
77                     ancienPrix = float(fichier[locationUtilisateur][locationDate]) # on récupère l'
78                     ancien prix
79                 except IndexError: # si il n'y avait pas de prix définie avant
80                     ancienPrix = 0
81                     # On rajoute la case
82                     fichier[locationUtilisateur] += ['0' for _ in range(0, locationDate - 1)]
83                     ancienPrix += prix # on y ajoute le nouveaux prix
84                     fichier[locationUtilisateur][locationDate] = f"{float(ancienPrix):.2f}" # on met à
85                     jour le fichier
86
87         with open("stats.csv", 'w') as f: # on applique les changements
88             ecriture = csv.writer(f)
89             ecriture.writerows(fichier)
90
91 def exporteCSV(self, chemin: str, utilisateurID: int) -> None:
92     """
93     Exporte les statistiques d'un utilisateur dans un fichier 'CSV'.
94     - N'exporte que les statistiques du jour.
95     """
96     donnees = self.recuperationDonneesCSV(utilisateurID)
97     aujourdHui = date.today().strftime(self.formatDate)
98     with open(chemin, 'w') as f:
99         fichier = csv.writer(f)
100        fichier.writerow(["ID Utilisateur", f"Totales des ventes du jour ({aujourdHui})"])
101        if len(donnees) > 0: # si il y a des données enregistrées
102            fichier.writerow([utilisateurID, donnees[aujourdHui]])
103        else:

```



```

100         fichier.writerow([utilisateurID, "Aucune ventes enregistrée"])
101
102     def recuperationDonneesCSV(self, utilisateurID: int) -> dict:
103         """Renvoie les informations contenu dans le fichier 'CSV' globale."""
104         self.miseAJourDatesCSV() # met à jour les dates du fichier 'CSV'
105         with open("stats.csv", 'r') as f:
106             fichier = list(csv.DictReader(f)) # lecture du fichier sous forme d'une liste de
            dictionnaire
107             for utilisateur in fichier: # on regarde tous les utilisateurs stockés dans le fichier
108                 if utilisateur["id"] == str(utilisateurID): # si utilisateur trouvé
109                     return utilisateur # renvoie des infos de l'utilisateur
110             return {} # ne retourne rien si l'utilisateur n'était pas présent dans le fichier
111
112     def miseAJourDatesCSV(self) -> None:
113         """
114         Mets-à-jour les dates trop anciennes du fichier globales 'CSV'.
115
116         On remplit les espaces vides par la valeur '0' pour les jours qui remplacent les dates
117         trop vieilles (âgées de plus d'une semaine) car soit aucun chiffre n'a été fait ce jour là,
118         car aucune information n'était renseignée.
119         """
120         besoinDeModification = False
121         with open("stats.csv", 'r') as f:
122             fichier = list(csv.reader(f))
123             if len(fichier) == 1: # si fichier ne contient que l'entête
124                 self.creationCSV(True) # on recrée le fichier dans le doute (avec les bonnes dates)
125             else:
126                 index = 2 # variable qui permet de savoir quel index on regarde (on commence à 2 car
on ignore les 2 premières valeurs [id et pseudo])
127                 mauvaisIndex = [] # liste qui va stocker les index trop vieux (> 1 semaine)
128                 datesPresentes = [] # liste qui stock les dates valides présentes dans le fichier (<
1 semaine)
129                 datesDisponibles = self.datesDisponibles() # liste des bonnes dates
130                 for dateFichier in fichier[0][index:]: # on regarde toutes les dates du fichier
131                     if dateFichier not in datesDisponibles: # si trop vieux
132                         mauvaisIndex.append(index) # on ajoute l'index à la liste
133                         besoinDeModification = True
134                     else:
135                         datesPresentes.append(dateFichier) # on ajoute la date à la liste
136                         index += 1
137
138                 if not besoinDeModification: # vérification si on a besoin de rien faire
139                     return # on quitte la fonction
140
141                 datesARajouter = [date for date in datesDisponibles if date not in datesPresentes] #
liste des dates à rajouter
142                 if len(datesARajouter) != len(mauvaisIndex): # ne devrais pas arrivé mais on sait
jamais
143                     raise IndexError("Problème, pas autant de dates à rajouter que de de dates
périmés dans le fichier.")
144                 for idx in mauvaisIndex: # pour tous les mauvais index
145                     for numLigne, ligne in enumerate(fichier): # on regarde toutes les lignes du
fichier
146                         if idx < len(ligne): # s'il y a un élément dans la ligne à l'index donnée ou
si elle est vide de toute façon
147                             if numLigne == 0: # si c'est la ligne d'entête
148                                 ligne[idx] = datesARajouter[0] # on change la ligne avec la nouvelle
date
149                             datesARajouter.pop(0)
150                             else: # si c'est une ligne de donnée
151                                 ligne[idx] = '0' # on change la ligne avec une valeur vide
152                                 fichier[numLigne] = ligne # on applique les changements
153                 if besoinDeModification: # vérification si on a besoin de faire des changements
154                     with open("stats.csv", 'w') as f: # on applique les changements
155                         ecriture = csv.writer(f)
156                         ecriture.writerows(fichier)

```