Créer un dossier avec le fichier mod_temperature.py.

1 Problématique

La *Banque des Périgourdins (BdP)* est une institution nouvellement implantée en Dordogne. Outre un compte courant classique, elle propose également des comptes rémunérés à différents taux. À une date fixée, la banque calcule les intérêts et crédite les comptes rémunérés des clients. Elle vous demande de créer un logiciel pour administrer correctement sa clientèle.

Réflexion en commun : En utilisant les principes de la programmation orientée objet (POO), quels objets pouvons-nous créer pour modéliser le problème ?

2 Modularité

Un jeu comme Fortnite est composé de centaines d'objets et emploie des centaines d'intervenants. Un programmeur n'a pas besoin de connaître l'implémentation de chaque objet. Ce dernier lui fournit par contre une interface qui définit ce que peut faire l'objet. La classe garantit ce que va faire l'objet (mais pas comment le programmeur va s'en servir).

Un intérêt important de programmer en POO est de rendre les structures indépendantes et réutilisables. Le programmeur peut importer une classe sans connaître son implémentation. Il n'a affaire qu'à son *interface* : elle peut être définie comme un contrat entre la classe et le programmeur : La classe garantit ce que vont faire les méthodes. Ainsi l'utilisateur peut se référer à la documentation de la classe grâce à l'instruction suivante :

```
1 help(nom_de_la_classe)
```

Il faut bien entendu au préalable importer la classe dans le programme.

Activité 1 :

1. Depuis l'EDI créer un fichier *modularite_poo.py* dans le même dossier que *mod_temperature.py*.
2. Importer la classe Temperature située dans le fichier mod_temperature.py.
3. Afficher la documentation de la classe. Quel est le rôle de la méthode *convertir* ? Doit-on lui transmettre des paramètres ?
4. Créer une instance de la classe Temperature (un objet).
5. Définir la température à 11.9°C.
6. Afficher le renvoi de la méthode *convertir* sur cet objet.
7. Afficher la température en degré Fahrenheit.

environnement de développement « intégré » (abrégé EDI en français ou IDE en anglais, pour integrated development environment)
réponse : 285.05K / 53.42°F

Christophe Viroulaud

3 Définir les classes

Notre banque est définie par 3 classes : *Banque, Client, Compte*. Dans un premier temps chaque groupe devra implémenter une des classes au choix, en respectant la documentation ci-après. Chaque classe sera construite dans un fichier indépendant avec les noms *mod_banque.py, mod_client.py, mod_compte.py*. Ensuite, il devra récupérer les deux autres classes créées par les autres groupes et construire le programme de gestion de la banque.

NB : Les attributs précédés d'un _ sont considérés privés. Lors de la création de l'objet, ils sont crées avec une valeur par défaut indépendamment des variables fournies par l'utilisateur.

déclaration fonction avec type : def maf(d:int, t:str) -> bool :
Il est possible de créer d'autres méthodes dites *internes*.
comment gérer le pb des erreurs qui s'affichent tant qu'on n'a pas récupéré le module de l'autre ? créer classe "vide" avec **pass**

3.1 Module banque

La banque possède les attributs suivants :

— *nom* : str ; le nom de la banque,

— *taux* : float ; le taux de rémunération des comptes,

— *_clients* : list ; liste des clients de la banque.

— *_id_libre* : int ; prochain identifiant unique donné au nouveau compte, initialisé à 1.

Les méthodes fournies à l'utilisateur sont :

— *cree_client(nom : str)* $\rightarrow$ *None* : crée un nouveau client, lui crée un compte courant et un compte rémunéré par défaut,

— *trouve_client(nom : str)* $\rightarrow$ *Client* : renvoie le client avec le *nom* spécifié s'il existe False sinon,

— *remunere()* $\rightarrow$ *None* : applique le taux de rémunération à tous les comptes rémunérés des clients.

3.2 Module client

Le client possède les attributs suivants :

— *nom* : str ; le nom du client,

— *_comptes* : dict ; regroupe les comptes du client ; l'identifiant du compte est la clé.

Les méthodes fournies à l'utilisateur sont :

— *ouvre_compte(remunere : bool, i : int)* $\rightarrow$ *None* : crée un nouveau compte dans *_comptes* avec *i* pour clé,

— *ferme_compte(i : int)* $\rightarrow$ *str* : clôture le compte ayant *i* pour clé et renvoie "Le compte est supprimé" ; sinon renvoie "Ce compte n'existe pas.",

— *get_comptes()* $\rightarrow$ *dict* : renvoie le dictionnaire des comptes.

3.3 Module compte

Le compte possède les attributs suivants :

— *_credit* : float ; argent restant sur le compte, initialisée à 0,

— *est_remunere* : bool ; si vrai le compte est rémunéré,

Christophe Viroulaud

Les méthodes fournies à l'utilisateur sont :

- *credite(s : float)* → *None* : crédite le compte de la somme *s*,
- *debite(s : float)* → *str* : débite le compte de la somme *s* seulement si le crédit restant est supérieur ou égal à *s* ; renvoie le message "Opération effectuée" ou "Crédit insuffisant" sinon,
- *get_credit()* → *float* : renvoie la somme sur le compte.

4 Créer la banque

Il faut utiliser les méthodes telles que définies même si elle ne semble pas les plus adaptées.

Après avoir récupéré les trois classes, il faut créer un programme *banque_des_perigourdins.py* qui simulera l'activité bancaire. Réaliser le déroulement suivant :

- Créer la banque *Banque des Périgourdins* avec le taux de rémunération 1,01.
- Créer les clients *Jay, Laure, Bertran.*
- Créditer les comptes de Jay :
  - 1000 s'il s'agit d'un compte courant,
  - 200 s'il s'agit d'un compte rémunéré.
- Afficher les sommes sur les comptes de Jay en précisant s'il s'agit d'un compté rémunéré ou non.
- Rémunérer les comptes.
- Afficher les sommes sur les comptes de Jay en précisant s'il s'agit d'un compté rémunéré ou non.

pour aller + loin :

- créer un nouveau compte
- get_info_client() donne nom et numéro des comptes
- ferme_compte() ne pas supprimer compte courant ; reverser argent du compte sur compte courant

Christophe Viroulaud