TRAVAUX DIRIGÉS
DOCUMENTATION - RAPPELS SUR LES CLASSES, MÉTHODES

Ce TD va permettre de rappeler la notion de classe Python. Il est important à chaque fois de préparer le schéma UML de votre classe avant d'entamer toute écriture de code. Vous commenterez votre code en utilisant les éléments suivants

- le # démarre un commentaire programmeur [1] sur une seule ligne
- le commentaire sur plusieurs lignes

  """ ... """

  La docstring utilise des tag pour identifier des éléments importants (retour et type de retour, arguments et type d'argument, erreurs, attribut

Vous mettrez aussi vos codes sur github afin de conserver une trace de votre travail, voir de versionner entre 2 TD/TP

1. Rappel sur les fonctions

1.1. **rappel.** une fonction est un objet python défini par le mot clé **def** suivi par son nom déclaré dans l'espace de nommage d'un module. Elle possède aussi un ensemble d'arguments et peut générer un résultat.

La liste des arguments d'une fonction comporte 4 type d'arguments dans l'ordre :

(1) les arguments positionnels, situé en premier dans la liste

(2) les arguments nommés, qui possèdent une valeur par défaut, permettant de ne pas les renseigner lors de l'appels de la fonction

(3) la liste d'arguments *args : permet de passer un ensemble de valeur non définie au préalable

(4) le dictionnaire d'arguments **kwargs : qui permet de donner une liste de valeur nommée

Ces différents types d'arguments sont facultatifs, mais s'il y a des arguments positionnels, alors ils sont situés au début de la liste des arguments, puis suivent les arguments nommées, la liste d'arguments et pour finir le dictionnaire d'arguments. Ces deux derniers peuvent être vide à l'appel de la fonction.

Enfin, les arguments positionnels et nommés peuvent recevoir une annotation de type sous la forme

`nom de l'argument : type`

Le type de retour de la fonction peux aussi être annoté sous la forme

`def fonction(arguments) -> type :`

```
def addition (a : int, b : int) -> int :
  """
    fonction d'addition de deux nombres entiers
  """
  return a + b
```

1.2. **exercice.** Vous allez créer les fonctions suivantes :

- la fonction qui retourne le plus grand de 2 nombres réels. Vous penserez à annoter les types.
- la fonction qui indique si la valeur passée est supérieure à un seuil. Ce seuil est fixé à 10, mais peux être modifié dans un second argument
- la fonction qui retourne la plus grande valeur d'une liste de valeur fournie
- la fonction qui retourne le nombre de valeurs d'une liste inférieure à un seuil qui est défini par défaut à 3 mais qui peux être modifié lors de l'appel de la fonction
- Une fonction qui affiche l'ensemble des données d'un dictionnaire passé en argument. Cette fonction prendra aussi comme argument une chaine de caractère qui précédera chaque affichage.

1. il faut consulter le code pour voir ces commentaires programmeurs

2. RAPPEL SUR LES CLASSES

2.1. **Rappel.** Une classe est définie par le mot clé **class**. On crée des instances d'une classe en faisant appel au nom de la classe avec entre parenthèse des valeurs initialisant les attributs d'instance

Une classe peux posséder des attributs de classe, définis directement au début du bloc de classe ou des attributs d'instance, définis dans la méthode __init__. La classe possède son propre espace de nommage distinct des espaces de nommage des instances.

```
6  class MaClasse:
7    monAttributClasse : str = "bonjour"
8    def __init__(self, valeur : int):
9      self.monAttributInstance = valeur
10
11 mc = MaClasse(12)
12 print(f"{mc}, {mc.monAttributClasse},{mc.monattributInstance}")
```

monAttributClasse est défini dans l'espace de nommage de la classe quand monAttributInstance est défini dans l'espace de nommage de l'instance, donc ici dans l'espace de nommage de mc.

On accéde à la liste des variables définies dans l'espace de nommage à l'aide de la fonction builtin **vars**

```
13 >>> vars(Maclasse) #donne l'espace de nommage de la classe
14 >>> vars (mc) # donne l'espace de nommage de l'instance mc de la classe MaClasse
```

Une classe peux implémenter un certain nombre de méthodes spéciales :

- → `__init__`
  initialisation des attributs d'instance
- → `__str__`
  formatage de la chaine caractères dans les appels print
- → `__repr__`
  formatage de la chaine caractères dans l'affichage en mode interactif

Vous pouvez aussi créer des méthodes liées à votre classe ou a une de ses instances. Les méthodes d'instances admettent comme premier argument le mot clé *self* qui fait référence à l'objet qui appelle la méthode. Si des attributs d'instances sont utilisé dans la méthode, ce sont les attributs de l'objet qui seront utilisés

2.2. **exercice.** Pour illustrer tous cela, vous aller créer la classe **Tasse** :

(1) La classe **Tasse** possède un attribut de classe qui est matière qui pointe vers la valeur chaine de caractère "céramique". **La classe Tasse possède une méthode d'initialisation de 3 attributs d'instance, la couleur, la contenance en ml et la marque.**

(2) Elle posséde aussi une méthode `__str__` qui permettra de retourner la chaine de caractère formaté suivante *"la tasse de matière céramique, de couleur bleur et de marque duralex a une contenance de 50 ml"*

(3) Elle possède aussi une méthode qui permet de définir le contenu comme un nouvel attribut d'instance

(4) et une autre méthode qui permet d'éliminer cet attribut si la boisson est bue. (pour cela vous utiliserez l'instruction buildtin **del**

Créer cette classe et créer un programme instanciant plusieurs objets de cette classe et utilisant les différentes méthodes que vous avez créer.

3. PREMIER PAS DANS LA DOCUMENTATION DE CODE

```
15 def bonjour(nom : str) -> None:
16     ''' fonction permettant de dire bonjour \'a une personne dont
17     le nom est pass\'e en argument
18     '''
19     print(f"bonjour {nom}")
20 if __name__ == "__main__":
21     bonjour("Arnauld")
22     print(f" documentation fonction bonjour {bonjour.__doc__}")
```

Les lignes 2 à 4 du code sont un commentaire sur plusieurs lignes. Comme il est situé au début d'une fonction il correspond aussi à la docstring de la fonction.

4. la documentation

Nous allons utiliser le format REST et sphinx https://www.sphinx-doc.org/fr/master/index.html pour créer notre documentation à publier ensuite.

Pour illustrer par l'exemple, vous allez créer dans un même fichier **example1.py** une classe, une fonction et le main qui va utiliser ces différents éléments.

4.1. **fonction affiche.** Cette fonction prend en argument une chaine de caractère et effectue l'affichage de cette chaine précédée de "texte à afficher :"

4.2. **classe vélo.** cette classe défini un vélo par sa marque, sa taille de pneu en pouce, sa couleur, son nombre de vitesse. Elle possède une méthode __init__ avec une valeur pour chacun des attributs, une méthode gear_up qui augmente la valeur de la vitesse courante, et gear_down, qui diminue la valeur de la vitesse courante. Ces deux méthodes renvoient la valeur de vitesse courante. Elles vérifient bien sur de rester entre 1 et le nombre de vitesses.

4.3. **la fonction principale.** Cette fonction qui sera appelée pour l'exécution du programme devra :

(1) Définir une chaine de caractère **str1**
(2) Faire appel à la fonction affiche en lui passant la chaine **str1**
(3) Créer une instance de Vélo nommée **v1**
(4) Faire appel aux méthodes gear_up et gear_down

Pour chacun de ces éléments vous écrirez de la documentation. Dans votre EDI, vous devriez voir apparaitre cette documentation lorsque vous survolez les éléments (classe, fonction, ...). Utilisez les tags définis dans https://www.sphinx-doc.org/fr/master/usage/restructuredtext/domains.html#the-python-domain

4.4. **génération de la documentation en ligne.** Nous allons utiliser sphinx pour générer la documentation en ligne. Dans un premier temps vous allez charger les packages nécessaires, à savoir,

(1) sphinx
(2) furo
(3) sphinx-autobuild

Ensuite vous allez effectuer les étapes suivantes.

— Dans votre projet placez le fichier python que vous venez de créer dans un répertoire nommé *src* (prenez l'habitude de créer ce répertoire *src* surtout sur des projets conséquents)
— Créer aussi un répertoire *docs* au même niveau que *src*
— positionnez vous dans le terminal dans le répertoire *docs* et exécutez la commande sphinx-quickstart. Cette commande pose un certain nombre de questions. Laisser les valeurs par défaut quand elles sont proposées et renseigner les autres en fonction de votre projet (auteur, nom du projet, version). Cette commande crée un ensemble de fichiers et de sous-répertoires dans le répertoire *docs*
— éditer le fichier conf.py
  (1) ajouter les extensions suivantes dans la liste vide extensions : "sphinx.ext.autodoc","sphinx.ext.viewcode",
  (2) ajouter en début de fichier les lignes suivantes

```
23 import os
24 import sys
25 sys.path.insert(0,os.path.abspath("../src"))
```

  cela permet d'indiquer à sphinx où sont les fichiers pythons à analyser pour produire la documentation
  (3) modifier le théme html en le remplaçant par furo.
— Au niveau du répertoire projet, exécutez la commande

```
26 sphinx-apidoc -o docs src
```

cela créer différents répertoire pour tenir compte du contenu de vos fichier source.

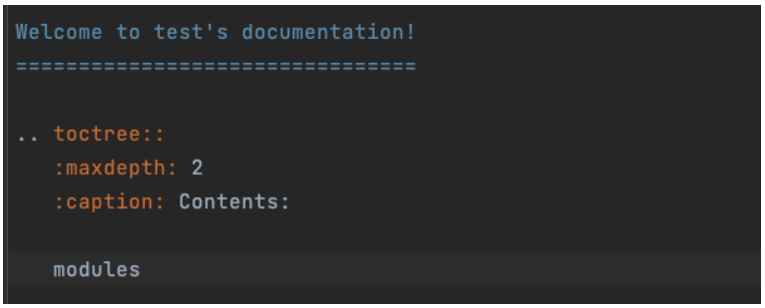

FIGURE 1. fichier index.rst

— ensuite, dans le fichier *index.rst* qui permet de définir la page d'accueil de votre documentation, vous allez ajouter "modules" dans la partie ".. toctree : :" à la fin de ce bloc en laissant une ligne vide. (cf figure 1)

— et pour finir au niveau du repértoire *docs*, exécutez la commande

```
27 make html
```

qui permet de générer une arborescence de fichier html disponible dans le répertoire **docs/-build/html/**