Feuille d'exercices

0. Hello world

Démarrer Thonny (ou votre éditeur/IDE préféré)

Écrire et lancer le programme suivant :

```
print("Hello World!")
```

1. Variables

1.1. : Écrire, lancer et analyser l'execution du programme suivant :

```
message = "Je connais la réponse à l'univers, la vie et le reste"
reponse = 6 * 7

print(message)
print(reponse)
```

1.2: À l'aide de python, calculer le résultat des opérations suivantes :

- 567×72
- 33^4
- 98.2/6
- ((7×9)^4)/6

2. Interactivité

2.1.1 : Demander l'âge de l'utilisateur, puis calculer et afficher l'âge qu'il aura dans deux ans.

2.1.2 : Même chose, mais en demandant l'année de naissance (approximativement, sans tenir compte du jour et mois de naissance. . .)

3. Chaînes de caractères

3.1.1 : Demander un mot à l'utilisateur. Afficher la longueur du mot avec un message tel que "Ce mot fait X caractères !"

3.1.2 : Remplacer les lettres A et E dans le mot, et afficher le mot ainsi modifié.

3.1.3 : Encadrer le mot modifié avec des `####`.

4. Fonctions

4.1.1 : Ecrire une fonction `centrer` qui permet de centrer une chaîne de caractère sur 80 caractères. Par exemple 'print(centrer(Pikachu)) affichera :

```
|                                       Pikachu                                        |
```

Afficher également la longueur du résultat.

4.1.2 : Ajouter un argument optionnel pour gérer la largeur au lieu du "80" fixé. Par exemple `print(centrer("Pikachu", 40)) affichera :`

```
|                Pikachu                |
```

4.1.3 : Créer une fonction `encadrer` qui utilise la fonction `centrer` pour produire un texte centré et encadré avec des `####`. Par exemple, `print(encadrer("Pikachu", 40))` affichera :

```
########################################
#               Pikachu                #
########################################
```

4.1.4 : Au lieu d'utiliser des `#` pour encadrer le texte, passer le caractère en argument optionnel. (Par exemple : `encadrer("Pikachu", 20, '@')`)

5. Conditions

5.1.1 : Reprendre la fonction `centrer` de l'exercice 4.1 et gérer le cas où la largeur demandée est -1 : dans ce cas, ne pas centrer. Par exemple, `print(encadrer("Pikachu", -1))` affichera :

```
###########
# Pikachu #
###########
```

5.1.2 : Gérer le cas où le caractère d'encadrement est vide. Dans ce cas, ne pas encadrer. Par exemple : `encadrer("Pikachu", 40, '')` affichera juste :

```
|                Pikachu                |
```

5.1.3 : Dans la fonction `centrer`, gérer également le cas où le texte dépasse la largeur demandée. Dans ce cas, tronquer le texte. Par exemple : `encadrer("Pikachu", 8)` affichera :

```
########
# Pika #
########
```

Trouvez de nouvelles façon de faire bugger la fonction. Conclusion ?

6. Exceptions, assertions

6.1 : Ecrire une fonction `demander_nombre_pair()` qui demande un entier pair à l'utilisateur, puis vérifie que la réponse est bien un entier pair. Si ce n'est pas le cas, déclencher une exception expliquant le problème (en francais). Si tout est bon, la fonction renvoie l'entier entré par l'utilisateur.

6.2 : Reprendre les fonctions `centrer` et `encadrer` et vérifier au début des fonctions certaines hypothèses faites, comme : - la longueur doit être un entier positif ou égal à -1 - le caractère d'encadrement doit être un caractère (chaine de longueur 1) ou vide - le texte donné doit effectivement être une chaîne de caractère

7. Boucles

7.1.1 : Écrire une fonction qui, pour un nombre donné, renvoie la table de multiplication. Par exemple `print(table_du_7())` affichera :

```
Table du 7
----------
 1 x 7 = 7
 2 x 7 = 14
 [..]
 10 x 7 = 70
```

7.2 : Cette fois, passer le nombre en argument. La fonction devient par exemple `table_multiplication(7)`

7.3 : En utilisant cette fonction, afficher les tables de multiplication pour tous les nombres entre 1 et 10.

7.4 : Protéger l'accès à toute cette connaissance précieuse en demandant, au début du programme, un "mot de passe" jusqu'à ce que le bon mot de passe soit donné.

7.5 : Écrire une fonction qui permet de déterminer si un nombre est premier. Par exemple `is_prime(3)` renverra True, et `is_prime(10)` renverra False.

7.6 : Écrire une fonction qui permet de générer les n premiers nombres de la suite de Fibonnaci

7.7 : Jeu des allumettes

Le jeu des allumettes est un jeu pour deux joueurs, où n allumettes sont disposées, et chaque joueur peut prendre à tour de rôle 1, 2 ou 3 allumettes. Le perdant est celui qui se retrouve obligé de prendre la dernière allumette.

- Écrire une fonction `afficher_allumettes` capable d'afficher un nombre donné d'allumettes (donné en argument), par exemple avec le caractère `|`

- Écrire une fonction `choisir_nombre` qui demande à l'utilisateur combien d'allumette il veut prendre. Cette fonction vérifiera que le choix est valide (en entier qui est soit 1, 2 ou 3).
- Commencer la construction d'une fonction `partie_allumettes` qui pour le moment, se contente de :
  - Initialiser le nombre d'allumette sur la table
  - Afficher des allumettes avec `afficher_allumettes`
  - Demander à l'utilisateur combien il veut prendre d'allumettes avec `choisir_nombre`
  - Propager ce choix sur le nombre d'allumette actuellement sur la table
  - Afficher le nouvel état avec `afficher_allumettes`
- Modifier `partie_allumettes` pour gérer deux joueurs (1 et 2) et les faire jouer à tour de rôle jusqu'à ce qu'une condition de victoire soit détectée (il reste moins d'une allumette...) moins d'une allumette.

7.8 : Intelligence artificielle

Reprendre le jeu précédent et le modifier pour introduire une "intelligence" artificielle qui soit capable de jouer en tant que 2ème joueur. (Par exemple, une stratégie très simple consiste à prendre une allumette quoiqu'il arrive)

Produire un fichier unique contenant l'"intelligence" artificielle : le fichier devra au moins contenir une fonction `play(allumettes_restantes)` qui renvoie le nombre d'allumettes que l'IA décide de prendre. Partager le fichier pour faire combattre les différentes IA dans l'arène !

8. Structures de données

8.1.1 : Écrire une fonction qui retourne le plus grand élément d'une liste (ou d'un set) de nombres, et une autre fonction qui permet retourner le plus petit. Par exemple, `plus_grand([5, 9, 12, 6, -1, 4])` retournera 12.

8.1.2 : Dans la fonction précédente, vérifier que l'argument donné est bien une liste non vide.

8.2 : Écrire une fonction qui calcule la somme d'une liste de nombres. Par exemple, `somme([3, 4, 5])` renverra 12.

8.3 : Comme 8.2, mais cette fois sans utiliser de variable intermédiaire (utiliser la récursivité)

8.4 : Écrire une fonction qui retourne seulement les entiers pairs d'une liste

8.5 : Écrire une fonction qui permet de trier une liste (ou un set) d'entiers

8.6 : Écrire une fonction qui prends en argument un chemin de fichier comme "/usr/bin/toto.py" et extrait le nom du fichier, c'est à dire "toto". On pourra utiliser la méthode `chaine.split(caractere)` des chaînes de caractère.

8.7 : Écrire un “générateur de formule de compliment / encouragement”. Voir : http://www.nioutaik.fr/Felicitron/ . Pour cela, construire des chaines de caractère avec la structure suivante : {superlatif} + {chose/personne} + {adjectif} + {verbe} + {beneficiaire} + {duree}. Chaque élément sera choisi aléatoirement dans une liste de choix possibles...

9. Fichiers

9.1 Écrire une fonction qui prends un nom de fichier en argument et retourne le contenu si elle a été capable de le récupérer. Sinon, elle doit déclencher une exception qui explique en français pourquoi elle n'a pas pu.

9.2 : Écrire une fonction qui remplace un mot par un autre dans un fichier. On pourra pour cela se servir de `une_chaine.replace("mot", "nouveau_mot")` qui renvoie une version modifiée de `une_chaine` en ayant remplacé “mot” par “nouveau mot”.

9.3 : Écrire une fonction qui permet d'afficher un fichier sans les commentaires et les lignes vides. Spécifier le caractère qui symbolise le début d'un commentaire en argument de la fonction. (Ou pourra utiliser la méthode `strip()` des chaînes de caractère pour identifier plus facilement les lignes vides)

9.4 : Écrire une fonction qui trouve et retourne tous les utilisateurs dont le shell de login (?) est `/bin/bash` dans `/etc/passwd`

10. Librairies

Les énoncés des exercices suivants sont un peu plus compliqués et techniques que les précédents, et ont aussi pour objectifs de vous inciter à explorer la documentation des librairies pour trouver les outils dont vous avez besoin. Il existe de nombreuse façon de résoudre chaque exercice.

10.1.1 : Télécharger le fichier `https://app.yunohost.org/community.json` (avec votre navigateur ou `wget` par exemple). Écrire une fonction qui lit ce fichier, le charge en tant que données json. Écrire une autre fonction capable de filter le dictionnaire pour ne garder que les apps d'un level `n` donné en argument. Écrire une fonction similaire pour le status (`working`, `inprogress`, `broken`).

10.1.2 : Améliorez le programme précédent pour récupérer la liste directement depuis le programme avec `requests`. Gérer les différentes exceptions qui pourraient se produire (afficher un message en français) : syntaxe json incorrecte, erreur 404, time-out du serveur, erreur SSL

10.1.3 : Écrire le résultat d'un tri (par exemple toutes les applications cassées) dans un fichier json.

10.2 : Écrire une fonction qui vérifie si un utilisateur système donné a le droit d'accéder à un fichier. On précisera en argument si il s'agit de droit de lecture, d'écriture ou d'execution.

10.3.1 : Écrire une fonction qui permet de trouver récursivement dans un dossier tous les fichiers modifiés il y a moins de 5 minutes.

10.3.2 : À l'aide d'une deuxième fonction permettant d'afficher les `n` dernières lignes d'un fichier, afficher les 10 dernières lignes des fichiers récemment modifiés dans `/var/log`

10.4 : Écrire des fonctions qui permettent de tester automatiqmenet les fonctions écrites en 10.3. Pour cela, la première fonction de test pourra par exemple créer un dossier temporaire contenant un fichier récemment modifié et un vieux fichier, puis utiliser `assert` pour vérifier ce que renvoie la fonction.

10.5 : Écrire une fonction qui renvoie les 3 processus les plus gourmands actuellement en CPU, et les 3 processus les plus gourmands en RAM (avec leur consommation actuelle, chacun en CPU et en RAM)