Feuille d'exercices : bases de Python

0. Hello world

0.1 : Démarrer Thonny (ou votre éditeur/IDE préféré)

0.2 : Écrire et lancer le programme suivant :

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

Créez un fichier `hello.py` directement dans la console (par exemple via `nano hello.py`) et mettez dedans :

```
#!/usr/bin/env python3
print("Hello, world!")
```

0.3 : Executez ensuite ce script à l'aide de `python3 hello.py` ou `./hello.py` dans un terminal

0.4 : Encore dans un terminal, lancez l'interpréteur en interactif et tapez directement dedans `print("Hello, world!")`

1. Variables

1.1. : De retour dans Thonny, écrire, lancer et analyser pas à pas 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
- $98.2/6$
- $((7 \times 9)^4)/6$

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 une message tel que `"Ce mot fait X caractères !"`

3.1.2 : Afficher le mot encadré avec des `####` . Par exemple:

```
##########
# Python #
##########
```

4. Fonctions

4.0.1 : Écrire une fonction `annee_naissance` qui prends en argument un age et retourne l'année de naissance (+/- 1) sachant que nous sommes ne 2022. Par exemple, `annee_naissance(29)` retounera l'entier `1990` .

4.0.2 : Inspecter l'execution du code pas à pas à l'aide de Thonny.

4.1.1 : Ecrire une fonction `centrer` prend en argument une chaîne de caractère, et retourne une nouvelle chaîne centrée sur 40 caractères. Par exemple `print(centrer("Python"))` affichera :

```
|                Python                |
```

4.1.2 : Ajouter un argument optionnel pour gérer la largeur au lieu du 40 "codé en dur". Par exemple `print(centrer("Python", 20)) affichera :`

```
|      Python      |
```

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("Python", 20))` affichera :

```
####################
|      Python      |
####################
```

5. Conditions

5.0.1 : Reprendre la fonction `annee_naissance` et afficher un message d'erreur et sortir immédiatement de la fonction si l'argument fourni n'est pas un nombre entre 0 et 130. Valider le comportement en appelant votre fonction avec comme argument `-12`, `158`, `None` ou `"toto"`.

5.0.2 : Inspecter l'execution du code pas à pas à l'aide de Thonny.

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

```
##########
# Python #
##########
```

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 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. Dans un premier temps, on pourra se contenter d'afficher les résultats. Par exemple `print(table_du_7())` affichera :

```
7
14
21
```

```
...
70
```

puis ensuite on peut améliorer la présentation pour obtenir le résultat :

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

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

7.1.3 : En appelant cette fonction plusieurs fois, afficher les tables de multiplication pour tous les nombres entre 1 et 10.

7.1.4 : 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.

7.2 : (Optionnel) 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.

7.3 : (Optionnel) Écrire une fonction qui permet de générer les n premiers nombres de la suite de Fibonnaci

7.4.1 : 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`

7.4.2 : (Optionnel) 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...).

7.4.3 : (Optionnel) 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)

8. Structures de données

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

```
assert plus_grand([5, 9, 12, 6, -1, 4]) == 12
assert plus_grand([-6, -19, -2]) == -2

assert plus_petit([5, 9, 12, 6, -1, 4]) == -1
assert plus_petit([-6, -19, -2]) == -19
```

8.1 : Écrire une fonction qui retourne le mot le plus long parmis une liste de mot donnée en argument.

```
assert plus_long(["Paris", "Amsterdam", "Londres"]) == "Amsterdam"
assert plus_long(["Choucroute", "Pizza", "Tarte flambée"]) == "Tarte flambée"
```

8.2 : Écrire une fonction qui calcule la somme d'une liste de nombres.

```
assert somme([3, 4, 5]) == 12
assert somme([0, 7, -3]) == 4
```

8.3 : 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.

8.4.1 : Récuperer le dictionnaire d'exemple auprès du formateur (`example_dict.py`) et boucler sur ce dictionnaire pour afficher quelque chose comme:

```
Sebastian est né.e en 1979
Barclay est né.e en 2000
```

```
Vivien est né.e en 1955
...
```

8.4.2 : Transformer le programme précédent pour n'afficher que les personnes ayant une adresse mail finissant par `.edu` .

8.5 : Ecrire une fonction `compte_lettres` qui prends en argument une (grande) chaîne de caractère et retourne un dictionnaire avec un compte des occurences des lettres. Par exemple `compte_lettres("hello")` retournera `{"h":1, "l": 2, "o": 1, "e":1 }` . Utiliser cette fonction sur Lorem Ipsum ("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt [...]")

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

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

8.8 : En **une seule** ligne de code, générer la matrice suivante :

```
[ [ 0, 1, 2, 3,  4  ],
  [ 0, 2, 4, 6,  8  ],
  [ 0, 3, 6, 9,  12 ],
  [ 0, 4, 8, 12, 16 ] ]
```

8.9 : Réécrire la fonction `somme` du 8.2, mais cette fois sans utiliser de variable intermédiaire (utiliser la récursivité)

8.10 : Ecrire un générateur `carre()` qui genere la suite 1, 4, 9, 16, ... Utiliser ce générateur pour afficher les carrés jusqu'à ce qu'une valeur dépasse 200.

8.11 : Ecrire un générateur `fibonnaci` qui genere la suite de fibonnaci 0, 1, 1, 2, 3, 5, 8, ... Utiliser ce générateur pour afficher les valeurs jusqu'à ce qu'elles dépassent 500.

9. Fichiers

9.1 : Créer un fonction `liste_users` qui lit le fichier `/etc/passwd` et retourne la liste des utilisateurs ayant comme shell de login `/bin/bash` .

9.2 : Dans le code Python, écrire un modèle d'email comme:

```
modele = """
Bonjour {prenom} !
```

```
Voici en pièce jointe les billets pour votre voyage en train vers {destination}.
"""
```

Ecrire une fonction `generer_email` qui remplace dans `modele` les chaines `{prenom}` et `{destination}` par des arguments fourni à la fonction, et enregistre le résultat dans un fichier `email_{prenom}.txt`. Par exemple, `generer_email("Alex", "Strasbourg")` générera le texte et sauvegardera le résultat dans `email_Alex.txt`.

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)

10. Librairies

Les énoncés des exercices suivants peuvent être un peu plus ouverts que les précédents, et ont aussi pour objectifs de vous inciter à explorer la documentation des librairies (ou Internet en général...) pour trouver les outils dont vous avez besoin. Il existe de nombreuse façon de résoudre chaque exercice.

JSON, requests et argparse

10.1.1 : Télécharger le fichier `https://app.yunohost.org/default/v2/apps.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 supérieur à `n` donné en argument. Écrire une fonction similaire pour le status (`working` , `inprogress` , `notworking`).

10.1.2 : Améliorer le programme précédent pour récupérer la liste directement depuis le programme avec `requests`. (Ajoutez une instruction pour s'assurer que le code du retour est bien 200 avant de continuer).

10.1.3 : Exporter le résultat d'un filtre (par exemple toutes les applications avec level >= 7) dans un fichier json.

10.1.4 : À l'aide de la librairie `argparse`, paramétrez le tri à l'aide d'un argument donné en ligne de commande. Par exemple: `python3 filtre_apps.py --level 7` exportera dans "result.json" seulement les apps level >= 7.

CSV

10.2.1 : Récupérer le fichier de données CSV auprès du formateur, le lire, et afficher le nom des personnes ayant moins de 24 ans. Pour ce faire, on utilisera la librarie `csv`.

10.2.2 : Trier les personnes du fichier CSV par année de naissance et enregistrer une nouvelle version de ce fichier avec seulement le nom et l'année de naissance. Pour trier, on pourra utiliser `sorted` et son argument `key` .

Random

10.3 : Écrire une fonction `jets_de_des(N)` qui simule N lancés de dés 6 et retourne le nombre d'occurence de chaque face dans un dictionnaire. Par exemple : `{1: 13, 2:16, 3:12, ... }` . Calculer ensuite la frequence (`nb_occurences / nb_lancés_total`) pour chaque face. Testez avec un N grand et en déduire si votre dé virtuel est pipé ou non.

10.4 : Écrire un fonction `create_tmp_dir` qui choisi un nombre au hasard entre 0 et 100000 puis créer le dossier `/tmp/tmp-{lenombre}` et retourne le nom du dossier ainsi créé. On pourra utiliser la librairie `random` pour choisir un nom aléatoire,
et `os.system` ou `subprocess.check_call` pour créer le dossier.

Interaction avec le systeme de fichier

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