L'objectif de ce TD est d'introduire les générateurs Python, c'est à dire le mot-clef `yield`, et le concept fondamental d'itérateur qui se cache derrière ainsi que le concept d'itérable spécifique à Python.

Exercice 1 : quel est le problème ?

Question 1

!!! question " " Analysez attentivement le code ci-dessous. Combien de lignes du fichier d'entrée faut-il lire pour que le programme affiche la note de l'étudiant situé en toute première position dans le fichier ?

```
#!/usr/bin/env python3

import sys

def traite_fichier(nom_fichier):
    fichier = open(nom_fichier, "r")
    resultats = []
    for ligne in fichier:
        ligne_decoupee = ligne.split(" ")
        prenom = ligne_decoupee[0]
        note = int(ligne_decoupee[1])
        resultats.append((prenom, note))
    fichier.close()
    return resultats

def cherche_note(prenom, prenoms_notes):
    for resultat in prenoms_notes:
        if resultat[0] == prenom:
            return resultat[1]
    return None

def get_note():
    if len(sys.argv) != 3:
        print("Usage :", sys.argv[0], "nom_fichier prenom")
        return
    prenoms_notes = traite_fichier(sys.argv[1])
    prenom = sys.argv[2]
    print("La note de", prenom, "est", cherche_note(prenom, prenoms_notes))

if __name__ == "__main__":
    get_note()
```

Correction question 1

Cliquez ici pour révéler la correction.

Le problème de ce code vient du fait que **tout** le fichier doit être traité avant de commencer la recherche. Si ce dernier est gigantesque alors que l'étudiant cherché se trouve au début, on perd du temps pour rien.

Question 2

!!! question " " Cette fois, on cherche à calculer la moyenne des notes se trouvant dans le fichier. Que peut-on dire sur l'espace mémoire occupé par ce programme ?

```
#!/usr/bin/env python3

import sys

def traite_fichier(nom_fichier):
    fichier = open(nom_fichier, "r")
    resultats = []
    for ligne in fichier:
        ligne_decoupee = ligne.split(" ")
        prenom = ligne_decoupee[0]
        note = int(ligne_decoupee[1])
        resultats.append((prenom, note))
    fichier.close()
    return resultats

def calcule_moyenne(prenoms_notes):
    somme = 0
    nb_etudiants = 0
    for resultat in prenoms_notes:
        somme += resultat[1]
        nb_etudiants += 1
    return somme / nb_etudiants

def teste():
    """Teste les fonctions ci-dessus."""
    if len(sys.argv) != 2:
        print("Usage :", sys.argv[0], "nom_fichier")
    else:
        prenoms_notes = traite_fichier(sys.argv[1])
        print("La note moyenne est", calcule_moyenne(prenoms_notes))

if __name__ == "__main__":
    teste()
```

Correction question 2

Cliquez ici pour révéler la correction.

Ici il n'y a plus le problème de perte de temps car dans **tous les cas** l'intégralité du fichier doit être traité. Néanmoins, une `list` de tous les étudiants est construite alors que cela n'est pas nécessaire. On utilise donc de la mémoire pour stocker tous les étudiants dans cette liste alors que cela n'est pas nécessaire.

Dans le **programme précédent** cherchant un étudiant dans un fichier, il y a **également de l'utilisation mémoire inutile.**.

Question 3

!!! question " " Comment corriger le problème de calculs inutiles et de mémoire dans le cas de la recherche d'un étudiant uniquement avec ce que nous avons vu jusqu'ici, c'est-à-dire sans avoir recours à `yield` ?

Correction question 3

Cliquez ici pour révéler la correction.

Il suffit de fusionner la recherche avec le traitement du fichier.

```
#!/usr/bin/env python3

"""Cherche la note d'un étudiant depuis son prénom"""

import sys

def cherche_note(nom_fichier, prenom_a_chercher):
    """Traitement fichier et recherche fusionnés"""
    fichier = open(nom_fichier, "r")
    for ligne in fichier:
        ligne_decoupee = ligne.split(" ")
        prenom = ligne_decoupee[0]
        note = int(ligne_decoupee[1])
        if prenom == prenom_a_chercher:
            fichier.close()
            return note
    fichier.close()
    return None

def get_note():
    if len(sys.argv) != 3:
        print("Usage :", sys.argv[0], "nom_fichier prenom")
        return
    fichier = sys.argv[1]
    prenom = sys.argv[2]
    print("La note de", prenom, "est", cherche_note(fichier, prenom))
```

```
if __name__ == "__main__":
    get_note()
```

Question 4

!!! question " " Quel est l'inconvénient de cette solution ?

Correction question 4

Cliquez ici pour révéler la correction.

Le code est moins bien structuré car le traitement du fichier et la recherche sont mélangés dans une même fonction. Ici ce n'est pas trop grave car le code est simple, mais dans du vrai code ça peut vite devenir très gênant pour la lisibilité/maintenance.

Exercice 2 : il n'y a pas de problème, il n'y a que des solutions

Voici donc comment, à l'aide de l'utilisation d'un `yield` Python, avoir une recherche d'étudiant dans laquelle :

- nous ne faisons aucun calcul pour rien ;
- nous ne créons pas de `list` d'étudiants, donc utilisation mémoire constante ;
- le traitement du fichier et la recherche sont séparés dans deux fonctions.

```
#!/usr/bin/env python3

import sys

def traite_fichier(fichier):
    for ligne in fichier:                                   # tf1 (bof, non ?)
        ligne_decoupee = ligne.split(" ")                   # tf2
        prenom = ligne_decoupee[0]                          # tf3
        note = int(ligne_decoupee[1])                       # tf4
        yield (prenom, note)                                # tf5

def cherche_note(prenom, resultats):
    for resultat in resultats:                              # cn1
        if resultat[0] == prenom:                           # cn2
            return resultat[1]                              # cn3
    return None                                             # cn4

def get_note():
    if len(sys.argv) != 3:                                  # gn1
        print("Usage :", sys.argv[0], "nom_fichier prenom")  # gn2
        return                                              # gn3
```

```
    fichier = open(sys.argv[1], "r")                          # gn4
    iterateur_prenoms_notes = traite_fichier(fichier)         # gn5
    prenom = sys.argv[2]                                      # gn6
    note = cherche_note(prenom, iterateur_prenoms_notes)      # gn7
    print("La note de", prenom, "est", note)                  # gn8
    fichier.close()                                           # gn9

if __name__ == "__main__":                                    # 1
    get_note()                                                # 2
```

Question 1

!!! question " " Analyser attentivement le code ci-dessus. En essayant de "deviner" ce que fait le `yield`, prendre quelques minutes pour dérouler le programme sur papier en écrivant les numéros de lignes successivement exécutées dans le cas d'un appel correct, c'est à dire avec un fichier qui existe et un nom d'étudiant qui existe en troisième position du fichier.

Correction question 1

Cliquez ici pour révéler la correction.

Voici l'ordre d'exécution :

- 1 2 gn1 gn4 gn5 gn6 gn7
- cn1 tf1 tf2 tf3 tf4 tf5 cn2
- cn1 tf1 tf2 tf3 tf4 tf5 cn2
- cn1 tf1 tf2 tf3 tf4 tf5 cn2 cn3
- gn8 gn9

Maintenant que nous avons vu le mot clef `yield` ainsi que son fonctionnement, il nous faut définir la notion fondamentale d'itérateur qui se cache derrière.

DÉFINITION D'UN ITÉRATEUR : état + fonction permettant d'avoir le prochain élément. Ce concept existe on delà du langage python.

DÉFINITION D'UN ITÉRABLE : instance Python à partir de laquelle on peut récupérer un itérateur, et donc que l'on peut parcourir. Les boucles for Python travaillent sur un itérable. Et les itérateurs sont eux-mêmes des itérables.

DÉFINITION D'UN GÉNÉRATEUR : fonction Python avec un `yield`, permettant de créer très facilement des itérateurs (contrairement à d'autres langages comme Java, C++ ou encore plus difficile C).

Les générateurs Python sont aussi appelés des *fonctions génératrices*. À chaque fois qu'une telle fonction est appelée, son code n'est pas exécuté ; un nouvel itérateur est simplement créé et renvoyé. Conceptuellement, l'état de cet itérateur contient le numéro de la prochaine ligne de code à exécuter. Initialement, ce numéro désigne la première ligne de la fonction génératrice.

Ensuite, à chaque fois que le prochain élément de l'itérateur est demandé, ce qui est fait par exemple dans notre dos quand on écrit `for elem in mon_iterateur`, alors le code de la fonction génératrice est exécuté jusqu'au prochain `yield` et son état est mis à jour de telle sorte que le numéro de la prochaine ligne de code à exécuter soit le numéro de la ligne suivant le `yield`.

Voici également deux vidéos qui peuvent vous éclairer. Elles présentent de deux manières différentes le mot-clef `yield` et les concepts qui se cachent derrière :

Question 2

!!! question " " À l'aide d'un générateur, réécrire le programme qui calcule la moyenne des notes. Quel est l'avantage par rapport au calcul de moyenne de l'exercice 1 ?

Correction question 2

Cliquez ici pour révéler la correction.

Voici le code de correction :

```
#!/usr/bin/env python3

import sys

def traite_fichier(nom_fichier):
    fichier = open(nom_fichier, "r")
    for ligne in fichier:
        ligne_decoupee = ligne.split(" ")
        prenom = ligne_decoupee[0]
        note = int(ligne_decoupee[1])
        yield (prenom, note)
    fichier.close()

def calcule_moyenne(resultats):
    somme = 0
    nb_etudiants = 0
    for resultat in resultats:
        somme += resultat[1]
        nb_etudiants += 1
    return somme / nb_etudiants

def teste():
    """Teste les fonctions ci-dessus."""
    if len(sys.argv) != 2:
        print("Usage :", sys.argv[0], "nom_fichier")
    else:
        iterateur_prenoms_notes = traite_fichier(sys.argv[1])
```

```
        print("La note moyenne est", calcule_moyenne(iterateur_prenoms_notes))

if __name__ == "__main__":
    teste()
```

L'avantage est que dans cette version il n'y pas de création de la `list` de tous les étudiants en mémoire.

Vidéo de correction complète des exercices 1 et 2 :

Exercice 3 : générer les jours de la semaine.

Question 1

!!! question " " Écrire une fonction génératrice renvoyant un itérateur sur les chaînes de caractères représentant les jours de la semaine.

Correction question 1

Cliquez ici pour révéler la correction.

L'objectif de cet exercice est d'enfoncer encore le clou sur le `yield`, en montrant qu'il n'y a pas forcément de boucle dans une fonction génératrice (c'est à dire une fonction avec des `yield`). La preuve en regardant la correction ci-dessous :

```
#!/usr/bin/env python3

"""On peut avoir uen fonction génératrice n'utilisant pas de boucles."""

def get_jours_semaine():
    """Fonction génératrice des jours de la semaine."""
    yield "lundi"
    yield "mardi"
    yield "mercredi"
    yield "jeudi"
    yield "vendredi"
    yield "samedi"
    yield "dimanche"

def affiche_jours():
    """Affiche les jours de la semaine."""
    iterateur_jours = get_jours_semaine()
    for jour in iterateur_jours:
        print(jour)
```

Vidéo de correction de l'exercice 3 :

Question 2

!!! question " " Qu'affiche le programme suivant ?

```
#!/usr/bin/env python3

"""Petit exemple pour illustrer le contexte de CHAQUE itérateur 1 et 2"""

def get_first_five():
    """Fonction génératrice des 5 premiers nombres."""
    yield "one"
    yield "two"
    yield "three"
    yield "four"
    yield "five"

def affiche():
    """Affiche des trucs en utilisant DEUX itérateurs."""
    iterateur_1 = get_first_five()
    iterateur_2 = get_first_five()

    print("De l'itérateur 1", next(iterateur_1))
    print("De l'itérateur 1", next(iterateur_1))
    print("De l'itérateur 2", next(iterateur_2))
    print("De l'itérateur 1", next(iterateur_1))

if __name__ == "__main__":
    affiche()
```

Correction question 2

Cliquez ici pour révéler la correction.

Pour répondre à la question il faut :

- savoir que la fonction `next` permet de récupérer le prochain élément d'un itérateur ;
- avoir compris le principe d'une fonction génératrice qui renvoie un **nouvel itérateur** à chaque fois qu'elle est appelée et qui sera exécutée morceaux par morceaux quand les éléments de cet itérateur seront demandés via là fonction `next` appelée explicitement ou implicitement si l'itérateur est utilisé dans une boucle.

Voici donc ce qu'affiche le code ci-dessus :

```
De l'itérateur 1 one
De l'itérateur 1 two
De l'itérateur 2 one
De l'itérateur 1 three
```

Exercice 4 : analyse de code

Question 1

!!! question " " Qu'affiche le programme ci-dessous ?

```
#!/usr/bin/env python3

"""Train reading someone else (bad) code"""

def mystery_function(first_parameter, second_parameter, everything):
    """What am I doing?"""
    for variable in first_parameter:
        everything.append(variable[second_parameter])
        yield variable[second_parameter]

def main():
    """Script's entry point"""
    a_list = []
    something = mystery_function([("to", 12), ("ti", 17), ("ta", 47)],
                                 1, a_list)
    print("type of something is", type(something))
    for element in something:
        print(element, end=' ')
    print()
    something_else = mystery_function(("to", "ti", "\ufdfd what is that?"),
                                      0, a_list)
    print("type of something_else is", type(something_else))
    for element in something_else:
        print(element, end=' ')
    print()
    print(a_list)
    other_thing = mystery_function({(1, 2, 3), (4, 5, 6), (7, 8)},
                                   2, a_list)
    print("type of other_thing is", type(other_thing))
    for element in other_thing:
        print(element, end=' ')
    print()

if __name__ == "__main__":
    main()
```

Correction question 1

Cliquez ici pour révéler la correction.

Voici ce qu'affiche ce programme :

```
type of something is <class 'generator'>
12 17 47
type of something_else is <class 'generator'>
t t
12], 17, 47, 't', 't', ' '[
type of other_thing is <class 'generator'>
3 Traceback (most recent call last):
  File "./mystery.py", line 32, in <module>
    main()
  File "./mystery.py", line 27, in main
    for element in other_thing:
  File "./mystery.py", line 8, in mystery_function
    everything.append(variable[second_parameter])
IndexError: tuple index out of range
```

Et voici ce qu'il faut en retenir :

- pour bien comprendre la notion des fonctions génératrice il faut dérouler le code ligne par ligne ;
- on voit bien qu'un appel à une fonction génératrice **n'exécute pas la fonction mais renvoie simplement un itérateur** (qui en pratique est de type `class 'generator` comme l'indique le programme, mais ceci est un détail d'implémentation, il faut juste retenir que c'est un itérateur tel que nous les avons définit dans ce TD) ;
- les `list`, les `tuple` et les `set` sont tous des itérables et donc "parcourables" avec une boucle `for` comme nous le savons déjà. Cela veut donc dire que l'on peut récupérer un itérateur sur leur contenu.
- il faut continuer à s'entraîner à lire les messages d'erreur ;
- il existe de très nombreux caractères unicode.

Vidéo de correction de l'exercice 4 :

Exercice 5 : quand aurions nous pu/dû utiliser `yield` ? (pour aller plus loin)

Question 1

!!! question " " Chercher dans ses notes, sa mémoire, son ordinateur, à quels endroits nous aurions pu/dû utiliser `yield` dans le cadre des TD et TP BPI en justifiant pourquoi ?

Correction question 1

Cliquez ici pour révéler la correction.

Voici une liste (à compléter), des endroits ou on aurait du faire du `yield` :

- dans nos listes chaînées pour récupérer toutes les cellules ;
- dans blobwar quand on récupère les voisins pour ne pas créer toute la `list` des voisins ;
- dans les sous-suite monotones pour avoir du code plus lisible ;
- dans mots suivants pour ne pas créer la `list` de tous les voisins.