MOOC Python 3

Session 2018

Corrigés de la semaine 3

comptage - Semaine 3 Séquence 2

```
def comptage(in_filename, out_filename):
    """
    retranscrit le fichier in_filename dans le fichier out_filename
    en ajoutant des annotations sur les nombres de lignes, de mots
    et de caractères
    """
    # on ouvre le fichier d'entrée en lecture
    with open(in_filename, encoding='utf-8') as in_file:
        # on ouvre la sortie en écriture
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            lineno = 1
            # pour toutes les lignes du fichier d'entrée
            # le numéro de ligne commence à 1
            for line in in_file:
                # autant de mots que d'éléments dans split()
                nb_words = len(line.split())
                # autant de caractères que d'éléments dans la ligne
                nb_chars = len(line)
                # on écrit la ligne de sortie; pas besoin
                # de newline (\n) car line en a déjà un
                out_file.write(f"{lineno}:{nb_words}:{nb_chars}:{line}")
                lineno += 1
```

comptage_bis - Semaine 3 Séquence 2

```
def comptage_bis(in_filename, out_filename):
    """
    un peu plus pythonique avec enumerate
    """
    with open(in_filename, encoding='utf-8') as in_file:
        with open(out_filename, 'w', encoding='utf-8') as out_file:
            # enumerate(.., 1) pour commencer avec une ligne
            # numérotée 1 et pas 0
            for lineno, line in enumerate(in_file, 1):
                # une astuce : si on met deux chaines
                # collées comme ceci elle sont concaténées
                # et on n'a pas besoin de mettre de backslash
                # puisqu'on est dans des parenthèses
                out_file.write(f"{lineno}:{len(line.split())}:"
                               f"{len(line)}:{line}")
```

comptage_ter - Semaine 3 Séquence 2

```
def comptage_ter(in_filename, out_filename):
    """
    pareil mais avec un seul with
    """
    with open(in_filename, encoding='utf-8') as in_file, \
         open(out_filename, 'w', encoding='utf-8') as out_file:
        for lineno, line in enumerate(in_file, 1):
            out_file.write(f"{lineno}:{len(line.split())}:"
                           f"{len(line)}:{line}")
```

comptage_quater - Semaine 3 Séquence 2

```
def comptage_quater(in_filename, out_filename):
    """
    si on est sûr que les séparateurs restent tous identiques,
    on peut écrire cette fonction en utilisant la méthode join
    en conjonction avec un tuple qui est un itérable
    pour ne pas répéter le séparateur
    """
    with open(in_filename, encoding="UTF-8") as in_file, \
         open(out_filename, mode='w', encoding="UTF-8") as out_file:
        for line_no, line in enumerate(in_file, 1):
            out_file.write(":".join((str(line_no), str(len(line.split())),
              str(len(line)), line)))
```

surgery - Semaine 3 Séquence 2

```
def surgery(liste):
    """
    Prend en argument une liste, et retourne la liste modifiée:
    * taille paire: on intervertit les deux premiers éléments
    * taille impaire >= 3: on fait tourner les 3 premiers éléments
    """
    # si la liste est de taille 0 ou 1, il n'y a rien à faire
    if len(liste) < 2:
        pass
    # si la liste est de taille paire
    elif len(liste) % 2 == 0:
        # on intervertit les deux premiers éléments
        liste[0], liste[1] = liste[1], liste[0]
    # si elle est de taille impaire
    else:
        liste[-2], liste[-1] = liste[-1], liste[-2]
    # et on n'oublie pas de retourner la liste dans tous les cas
    return liste
```

graph_dict - Semaine 3 Séquence 4

```
def graph_dict(filename):
    """
    construit une stucture de données de graphe
    à partir du nom du fichier d'entrée
    """
    # un dictionnaire vide normal
    graph = {}

    with open(filename) as feed:
        for line in feed:
            begin, value, end = line.split()
            # c'est cette partie qu'on économisera
            # dans la deuxième solution avec un defaultdict
            if begin not in graph:
                graph[begin] = []
            # remarquez les doubles parenthèses
            # car on appelle append avec un seul argument
            # qui est un tuple
            graph[begin].append((end, int(value)))
            # si on n'avait écrit qu'un seul niveau de parenthèses
            # graph[begin].append(end, int(value))
            # cela aurait signifié un appel à append avec deux arguments
            # ce qui n'aurait pas du tout fait ce qu'on veut
    return graph
```

graph_dict_bis - Semaine 3 Séquence 4

```
from collections import defaultdict

def graph_dict_bis(filename):
    """
    pareil mais en utilisant un defaultdict
    """
    # on déclare le defaultdict de type list
    # de cette façon si une clé manque elle
    # sera initialisée avec un appel à list()
    graph = defaultdict(list)

    with open(filename) as feed:
        for line in feed:
            # on coupe la ligne en trois parties
            begin, value, end = line.split()
            # comme c'est un defaultdict on n'a
            # pas besoin de l'initialiser
            graph[begin].append((end, int(value)))
    return graph
```

index - Semaine 3 Séquence 4

```
def index(bateaux):
    """
    Calcule sous la forme d'un dictionnaire indexé par les ids
    un index de tous les bateaux présents dans la liste en argument
    Comme les données étendues et abrégées ont toutes leur id
    en première position on peut en fait utiliser ce code
    avec les deux types de données
    """
    # c'est une simple compréhension de dictionnaire
    return {bateau[0] : bateau for bateau in bateaux}
```

index_bis - Semaine 3 Séquence 4

```
def index_bis(bateaux):
    """
    La même chose mais de manière itérative
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        resultat[bateau[0]] = bateau
    return resultat
```

index_ter - Semaine 3 Séquence 4

```
def index_ter(bateaux):
    """
    Encore une autre, avec un extended unpacking
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        # avec un extended unpacking on peut extraire
        # le premier champ; en appelant le reste _
        # on indique qu'on n'en fera en fait rien
        id, *_ = bateau
        resultat[id] = bateau
    return resultat
```

merge - Semaine 3 Séquence 4

```
def merge(extended, abbreviated):
    """
    Consolide des données étendues et des données abrégées
    comme décrit dans l'énoncé
    Le coût de cette fonction est linéaire dans la taille
    des données (longueur commune des deux listes)
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # pour les données étendues
    # on affecte les 6 premiers champs
    # et on ignore les champs de rang 6 et au delà
    for id, latitude, longitude, timestamp, name, country, *_ in extended:
        # on crée une entrée dans le résultat,
        # avec la mesure correspondant aux données étendues
        result[id] = [name, country, (latitude, longitude, timestamp)]
    # maintenant on peut compléter le résultat avec les données abrégées
    for id, latitude, longitude, timestamp in abbreviated:
        # et avec les hypothèses on sait que le bateau a déjà été
        # inscrit dans le résultat, donc result[id] doit déjà exister
        # et on peut se contenter d'ajouter la mesure abrégée
        # dans l'entrée correspondante dans result
        result[id].append((latitude, longitude, timestamp))
    # et retourner le résultat
    return result
```

merge_bis - Semaine 3 Séquence 4

```
def merge_bis(extended, abbreviated):
    """
    Une deuxième version, linéaire également
    mais qui utilise les indices plutôt que l'unpacking
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # on remplit d'abord à partir des données étendues
    for ship in extended:
        id = ship[0]
        # on crée la liste avec le nom et le pays
        result[id] = ship[4:6]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    # pareil que pour la première solution,
    # on sait d'après les hypothèses
    # que les id trouvées dans abbreviated
    # sont déja présentes dans le résultat
    for ship in abbreviated:
        id = ship[0]
        # on ajoute un tuple correspondant à la position
        result[id].append(tuple(ship[1:4]))
    return result
```

merge_ter - Semaine 3 Séquence 4

```
def merge_ter(extended, abbreviated):
    """
    Une troisième solution
    à cause du tri que l'on fait au départ, cette
    solution n'est plus linéaire mais en O(n.log(n))
    """
    # ici on va tirer profit du fait que les id sont
    # en première position dans les deux tableaux
    # si bien que si on les trie,
    # on va mettre les deux tableaux 'en phase'
    #
    # c'est une technique qui marche dans ce cas précis
    # parce qu'on sait que les deux tableaux contiennent des données
    # pour exactement le même ensemble de bateaux
    #
    # on a deux choix, selon qu'on peut se permettre ou non de
    # modifier les données en entrée. Supposons que oui:
    extended.sort()
    abbreviated.sort()
    # si ça n'avait pas été le cas on aurait fait plutôt
    # extended = extended.sorted() et idem pour l'autre
    #
    # il ne reste plus qu'à assembler le résultat
    # en découpant des tranches
    # et en les transformant en tuples pour les positions
    # puisque c'est ce qui est demandé
    return {
        ext[0] : ext[4:6] + [ tuple(ext[1:4]), tuple(abb[1:4]) ]
        for (ext, abb) in zip (extended, abbreviated)
        }
```

read_set - Semaine 3 Séquence 5

```
# on suppose que le fichier existe
def read_set(filename):
    """
    crée un ensemble des mots-lignes trouvés dans le fichier
    """
    # on crée un ensemble vide
    result = set()

    # on parcourt le fichier
    with open(filename) as feed:
        for line in feed:
            # avec strip() on enlève la fin de ligne,
            # et les espaces au début et à la fin
            result.add(line.strip())
    return result
```

read_set_bis - Semaine 3 Séquence 5

```
# on peut aussi utiliser une compréhension d'ensemble
# (voir semaine 5); ça se présente comme
# une compréhension de liste mais on remplace
# les [] par des {}
def read_set_bis(filename):
    with open(filename) as feed:
        return {line.strip() for line in feed}
```

search_in_set - Semaine 3 Séquence 5

```
# ici aussi on suppose que les fichiers existent
def search_in_set(filename_reference, filename):
    """
    cherche les mots-lignes de filename parmi ceux
    qui sont presents dans filename_reference
    """

    # on tire profit de la fonction précédente
    reference_set = read_set(filename_reference)

    # on crée une liste vide
    result = []
    with open(filename) as feed:
        for line in feed:
            token = line.strip()
            # remarquez ici les doubles parenthèses
            # pour passer le tuple en argument
            result.append((token, token in reference_set))

    return result
```

search_in_set_bis - Semaine 3 Séquence 5

```
def search_in_set_bis(filename_reference, filename):

    # on tire profit de la fonction précédente
    reference_set = read_set(filename_reference)

    # c'est un plus clair avec une compréhension
    # mais moins efficace car on calcule strip() deux fois
    with open(filename) as feed:
        return [(line.strip(), line.strip() in reference_set)
                for line in feed]
```

diff - Semaine 3 Séquence 5

```
def diff(extended, abbreviated):
    """Calcule comme demandé dans l'exercice, et sous formes d'ensembles
    (*) les noms des bateaux seulement dans extended
    (*) les noms des bateaux présents dans les deux listes
    (*) les ids des bateaux seulement dans abbreviated
    """

    ### on n'utilise que des ensembles dans tous l'exercice

    # les ids de tous les bateaux dans extended
    # avec ce qu'on a vu jusqu'ici le moyen le plus naturel
    # consiste à calculer une compréhension de liste
    # et à la traduire en ensemble comme ceci
    extended_ids = set([ship[0] for ship in extended])

    # les ids de tous les bateaux dans abbreviated
    # je fais exprès de ne pas mettre les []
    # de la compréhension de liste, c'est pour vous introduire
    # les expressions génératrices - voir semaine 5
    abbreviated_ids = set(ship[0] for ship in abbreviated)

    # les ids des bateaux seulement dans abbreviated
    # une difference d'ensembles
    abbreviated_only_ids = abbreviated_ids - extended_ids

    # les ids des bateaux dans les deux listes
    # une intersection d'ensembles
    both_ids = abbreviated_ids & extended_ids

    # les ids des bateaux seulement dans extended
    # ditto
    extended_only_ids = extended_ids - abbreviated_ids

    # pour les deux catégories où c'est possible
    # on recalcule les noms des bateaux
    # par une compréhension d'ensemble
    both_names = \
        set([ship[4] for ship in extended if ship[0] in both_ids])
    extended_only_names = \
        set([ship[4] for ship in extended if ship[0] in extended_only_ids])
    # enfin on retourne les 3 ensembles sous forme d'un tuple
    return extended_only_names, both_names, abbreviated_only_ids
```

diff_bis - Semaine 3 Séquence 5

```
def diff_bis(extended, abbreviated):
    """
    Même code mais qui utilise les compréhensions d'ensemble
    que l'on n'a pas encore vues - à nouveau, voir semaine 5
    mais vous allez voir que c'est assez intuitif
    """
    extended_ids = {ship[0] for ship in extended}
    abbreviated_ids = {ship[0] for ship in abbreviated}

    abbreviated_only_ids = abbreviated_ids - extended_ids
    both_ids = abbreviated_ids & extended_ids
    extended_only_ids = extended_ids - abbreviated_ids

    both_names = \
          {ship[4] for ship in extended if ship[0] in both_ids}
    extended_only_names = \
          {ship[4] for ship in extended if ship[0] in extended_only_ids}

    return extended_only_names, both_names, abbreviated_only_ids
```

diff_ter - Semaine 3 Séquence 5

```
def diff_ter(extended, abbreviated):
    """
    Idem sans les calculs d'ensembles intermédiaires
    en utilisant les conditions dans les compréhensions
    """
    extended_ids =     {ship[0] for ship in extended}
    abbreviated_ids =  {ship[0] for ship in abbreviated}
    abbreviated_only = {ship[0] for ship in abbreviated
                        if ship[0] not in extended_ids}
    extended_only =    {ship[4] for ship in extended
                        if ship[0] not in abbreviated_ids}
    both =             {ship[4] for ship in extended
                        if ship[0] in abbreviated_ids}
    return extended_only, both, abbreviated_only
```

diff_quater - Semaine 3 Séquence 5

```
def diff_quater(extended, abbreviated):
    """
    Idem sans indices
    """
    extended_ids =     {id for id, *_ in extended}
    abbreviated_ids =  {id for id, *_ in abbreviated}
    abbreviated_only = {id for id, *_ in abbreviated
                        if id not in extended_ids}
    extended_only =    {name for id, _, _, _, name, *_ in extended
                        if id not in abbreviated_ids}
    both =             {name for id, _, _, _, name, *_ in extended
                        if id in abbreviated_ids}
    return extended_only, both, abbreviated_only
```

fifo - Semaine 3 Séquence 8

```
class Fifo:
    """
    Une classe FIFO implémentée avec une simple liste
    """

    # dans cette première version on utilise
    # un object 'list' standard
    # on ajoute à la fin avec    queue.append(x),
    # et on enlève au début avec queue.pop(0)
    #
    # remarquez qu'on pourrait aussi
    # ajouter au début avec queue.insert(0, x)
    # enlever à la fin avec queue.pop()

    def __init__(self):
        # l'attribut queue est un objet liste
        self.queue = []

    def __repr__(self):
        contents = ", ".join(str(item) for item in self.queue)
        return f"[Fifo {contents}]"

    def incoming(self, item):
        # on insère au début de la liste
        self.queue.append(item)

    def outgoing(self):
        # pas la peine d'utiliser un try/except dans ce cas
        if self.queue:
            return self.queue.pop(0)
        # si on utilise pylint on va avoir envie de rajouter ceci
        # qui n'est pas vraiment indispensable..
        else:
            return None
```

fifo_bis - Semaine 3 Séquence 8

```
from collections import deque

class FifoBis:
    """
    une alternative en utilisant exactement la même stratégie
    mais avec un objet de type collections.deque
    en effet, l'objet 'list' standard est optimisé pour
    ajouter/enlever **à la fin** de la liste
    et on a vu dans la première version du code qu'il nous faut
    travailler sur les deux cotés de la pile, quel que soit le sens
    qu'on choisit pour implémenter la pile
    donc si la pile a des chances d'être longue de plusieurs milliers
    d'objets, il est utile de prendre un 'deque'
    'deque' vient de 'double-entry queue', et est optimisée
    pour les accès depuis le début et/ou la fin de la liste
    """
    def __init__(self):
        self.queue = deque()

    # ici pour faire bon poids on utilise la stratégie inverse
    # de la première version de la pile, on insère au début et on
    # enlève de la fin
    # du coup on les affice dans l'autre sens
    def __repr__(self):
        contents = ", ".join(str(item) for item in reversed(self.queue))
        return f"[Fifo {contents}]"

    def incoming(self, item):
        self.queue.insert(0, item)

    def outgoing(self):
        if self.queue:
            return self.queue.pop()

```