

MOOC Python 3

Session 2018

Corrigés de la semaine 3

comptage - Semaine 3 Séquence 2

```
1 def comptage(in_filename, out_filename):
2     """
3     retranscrit le fichier in_filename dans le fichier out_filename
4     en ajoutant des annotations sur les nombres de lignes, de mots
5     et de caractères
6     """
7     # on ouvre le fichier d'entrée en lecture
8     with open(in_filename, encoding='utf-8') as in_file:
9         # on ouvre la sortie en écriture
10        with open(out_filename, 'w', encoding='utf-8') as out_file:
11            lineno = 1
12            # pour toutes les lignes du fichier d'entrée
13            # le numéro de ligne commence à 1
14            for line in in_file:
15                # autant de mots que d'éléments dans split()
16                nb_words = len(line.split())
17                # autant de caractères que d'éléments dans la ligne
18                nb_chars = len(line)
19                # on écrit la ligne de sortie; pas besoin
20                # de newline (\n) car line en a déjà un
21                out_file.write(f"{lineno}:{nb_words}:{nb_chars}:{line}")
22                lineno += 1
```

comptage (bis) - Semaine 3 Séquence 2

```
1 def comptage_bis(in_filename, out_filename):
2     """
3     un peu plus pythonique avec enumerate
4     """
5     with open(in_filename, encoding='utf-8') as in_file:
6         with open(out_filename, 'w', encoding='utf-8') as out_file:
7             # enumerate(.., 1) pour commencer avec une ligne
8             # numérotée 1 et pas 0
9             for lineno, line in enumerate(in_file, 1):
10                 # une astuce : si on met deux chaines
11                 # collées comme ceci elle sont concaténées
12                 # et on n'a pas besoin de mettre de backslash
13                 # puisqu'on est dans des parenthèses
14                 out_file.write(f"{lineno}:{len(line.split())}: "
15                               f"{len(line)}:{line}")
```

comptage (ter) - Semaine 3 Séquence 2

```
1 def comptage_ter(in_filename, out_filename):
2     """
3     pareil mais avec un seul with
4     """
5     with open(in_filename, encoding='utf-8') as in_file, \
6         open(out_filename, 'w', encoding='utf-8') as out_file:
7         for lineno, line in enumerate(in_file, 1):
8             out_file.write(f"{lineno}:{len(line.split())}: "
9                             f"{len(line)}:{line}")
```

```
1 def surgery(liste):
2     """
3     Prend en argument une liste, et retourne la liste modifiée:
4     * taille paire: on intervertit les deux premiers éléments
5     * taille impaire >= 3: on fait tourner les 3 premiers éléments
6     """
7     # si la liste est de taille 0 ou 1, il n'y a rien à faire
8     if len(liste) < 2:
9         pass
10    # si la liste est de taille paire
11    elif len(liste) % 2 == 0:
12        # on intervertit les deux premiers éléments
13        liste[0], liste[1] = liste[1], liste[0]
14    # si elle est de taille impaire
15    else:
16        liste[-2], liste[-1] = liste[-1], liste[-2]
17    # et on n'oublie pas de retourner la liste dans tous les cas
18    return liste
```

```
1 from collections import defaultdict
2
3 def graph_dict(filename):
4     """
5     construit une stucture de données de graphe
6     à partir du nom du fichier d'entrée
7     """
8     # un dictionnaire vide normal
9     graph = {}
10
11     with open(filename) as feed:
12         for line in feed:
13             begin, value, end = line.split()
14             # c'est cette partie qu'on économisera
15             # dans la deuxième solution avec un defaultdict
16             if begin not in graph:
17                 graph[begin] = []
18             # remarquez les doubles parenthèses
19             # car on appelle append avec un seul argument
20             # qui est un tuple
21             graph[begin].append((end, int(value)))
22             # si on n'avait écrit qu'un seul niveau de parenthèses
23             # graph[begin].append(end, int(value))
24             # cela aurait signifié un appel à append avec deux arguments
25             # ce qui n'aurait pas du tout fait ce qu'on veut
26     return graph
```

```

1 def graph_dict_bis(filename):
2     """
3     pareil mais en utilisant un defaultdict
4     """
5     # on déclare le defaultdict de type list
6     # de cette façon si une clé manque elle
7     # sera initialisée avec un appel à list()
8     graph = defaultdict(list)
9
10    with open(filename) as feed:
11        for line in feed:
12            # on coupe la ligne en trois parties
13            begin, value, end = line.split()
14            # comme c'est un defaultdict on n'a
15            # pas besoin de l'initialiser
16            graph[begin].append((end, int(value)))
17    return graph

```

```

1 def index(bateaux):
2     """
3     Calcule sous la forme d'un dictionnaire indexé par les ids
4     un index de tous les bateaux présents dans la liste en argument
5     Comme les données étendues et abrégées ont toutes leur id
6     en première position on peut en fait utiliser ce code
7     avec les deux types de données
8     """
9     # c'est une simple compréhension de dictionnaire
10    return {bateau[0] : bateau for bateau in bateaux}

```

```
1 def index_bis(bateaux):
2     """
3     La même chose mais de manière itérative
4     """
5     # si on veut décortiquer
6     resultat = {}
7     for bateau in bateaux:
8         resultat[bateau[0]] = bateau
9     return resultat
```

```
1 def index_ter(bateaux):
2     """
3     Encore une autre, avec un extended unpacking
4     """
5     # si on veut décortiquer
6     resultat = {}
7     for bateau in bateaux:
8         # avec un extended unpacking on peut extraire
9         # le premier champ; en appelant le reste _
10        # on indique qu'on n'en fera en fait rien
11        id, *_ = bateau
12        resultat[id] = bateau
13    return resultat
```

```
1 def merge(extended, abbreviated):
2     """
3     Consolide des données étendues et des données abrégées
4     comme décrit dans l'énoncé
5     Le coût de cette fonction est linéaire dans la taille
6     des données (longueur commune des deux listes)
7     """
8     # on initialise le résultat avec un dictionnaire vide
9     result = {}
10    # pour les données étendues
11    # on affecte les 6 premiers champs
12    # et on ignore les champs de rang 6 et au delà
13    for id, latitude, longitude, timestamp, name, country, *_ in extended:
14        # on crée une entrée dans le résultat,
15        # avec la mesure correspondant aux données étendues
16        result[id] = [name, country, (latitude, longitude, timestamp)]
17    # maintenant on peut compléter le résultat avec les données abrégées
18    for id, latitude, longitude, timestamp in abbreviated:
19        # et avec les hypothèses on sait que le bateau a déjà été
20        # inscrit dans le résultat, donc result[id] doit déjà exister
21        # et on peut se contenter d'ajouter la mesure abrégée
22        # dans l'entrée correspondante dans result
23        result[id].append((latitude, longitude, timestamp))
24    # et retourner le résultat
25    return result
```

```
1 def merge_bis(extended, abbreviated):
2     """
3     Une deuxième version, linéaire également
4     mais qui utilise les indices plutôt que l'unpacking
5     """
6     # on initialise le résultat avec un dictionnaire vide
7     result = {}
8     # on remplit d'abord à partir des données étendues
9     for ship in extended:
10         id = ship[0]
11         # on crée la liste avec le nom et le pays
12         result[id] = ship[4:6]
13         # on ajoute un tuple correspondant à la position
14         result[id].append(tuple(ship[1:4]))
15     # pareil que pour la première solution,
16     # on sait d'après les hypothèses
17     # que les id trouvées dans abbreviated
18     # sont déjà présentes dans le résultat
19     for ship in abbreviated:
20         id = ship[0]
21         # on ajoute un tuple correspondant à la position
22         result[id].append(tuple(ship[1:4]))
23     return result
```



```

1 def merge_ter(extended, abbreviated):
2     """
3     Une troisième solution
4     à cause du tri que l'on fait au départ, cette
5     solution n'est plus linéaire mais en  $O(n \cdot \log(n))$ 
6     """
7     # ici on va tirer profit du fait que les id sont
8     # en première position dans les deux tableaux
9     # si bien que si on les trie,
10    # on va mettre les deux tableaux 'en phase'
11    #
12    # c'est une technique qui marche dans ce cas précis
13    # parce qu'on sait que les deux tableaux contiennent des données
14    # pour exactement le même ensemble de bateaux
15    #
16    # on a deux choix, selon qu'on peut se permettre ou non de
17    # modifier les données en entrée. Supposons que oui:
18    extended.sort()
19    abbreviated.sort()
20    # si ça n'avait pas été le cas on aurait fait plutôt
21    # extended = extended.sorted() et idem pour l'autre
22    #
23    # il ne reste plus qu'à assembler le résultat
24    # en découpant des tranches
25    # et en les transformant en tuples pour les positions
26    # puisque c'est ce qui est demandé
27    return {
28        ext[0] : ext[4:6] + [ tuple(ext[1:4]), tuple(abb[1:4]) ]
29        for (ext, abb) in zip (extended, abbreviated)
30    }

```

```

1  # on suppose que le fichier existe
2  def read_set(filename):
3      """
4      crée un ensemble des mots-lignes trouvés dans le fichier
5      """
6      # on crée un ensemble vide
7      result = set()
8
9      # on parcourt le fichier
10     with open(filename) as feed:
11         for line in feed:
12             # avec strip() on enlève la fin de ligne,
13             # et les espaces au début et à la fin
14             result.add(line.strip())
15     return result

```

```

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

```

```

1  # ici aussi on suppose que les fichiers existent
2  def search_in_set(filename_reference, filename):
3      """
4      cherche les mots-lignes de filename parmi ceux
5      qui sont presents dans filename_reference
6      """
7
8      # on tire profit de la fonction précédente
9      reference_set = read_set(filename_reference)
10
11     # on crée une liste vide
12     result = []
13     with open(filename) as feed:
14         for line in feed:
15             token = line.strip()
16             # remarquez ici les doubles parenthèses
17             # pour passer le tuple en argument
18             result.append((token, token in reference_set))
19
20     return result

```

```

1  def search_in_set_bis(filename_reference, filename):
2
3      # on tire profit de la fonction précédente
4      reference_set = read_set(filename_reference)
5
6      # c'est un plus clair avec une compréhension
7      # mais moins efficace car on calcule strip() deux fois
8      with open(filename) as feed:
9          return [(line.strip(), line.strip() in reference_set)
10                  for line in feed]

```

```

1 def diff(extended, abbreviated):
2     """Calcule comme demandé dans l'exercice, et sous formes d'ensembles
3     (*) les noms des bateaux seulement dans extended
4     (*) les noms des bateaux présents dans les deux listes
5     (*) les ids des bateaux seulement dans abbreviated
6     """
7
8     ### on n'utilise que des ensembles dans tous l'exercice
9
10    # les ids de tous les bateaux dans extended
11    # avec ce qu'on a vu jusqu'ici le moyen le plus naturel
12    # consiste à calculer une compréhension de liste
13    # et à la traduire en ensemble comme ceci
14    extended_ids = set([ship[0] for ship in extended])
15
16    # les ids de tous les bateaux dans abbreviated
17    # je fais exprès de ne pas mettre les []
18    # de la compréhension de liste, c'est pour vous introduire
19    # les expressions génératrices - voir semaine 5
20    abbreviated_ids = set(ship[0] for ship in abbreviated)
21
22    # les ids des bateaux seulement dans abbreviated
23    # une difference d'ensembles
24    abbreviated_only_ids = abbreviated_ids - extended_ids
25
26    # les ids des bateaux dans les deux listes
27    # une intersection d'ensembles
28    both_ids = abbreviated_ids & extended_ids
29
30    # les ids des bateaux seulement dans extended
31    # ditto
32    extended_only_ids = extended_ids - abbreviated_ids
33
34    # pour les deux catégories où c'est possible
35    # on recalcule les noms des bateaux
36    # par une compréhension d'ensemble
37    both_names = \
38        set([ship[4] for ship in extended if ship[0] in both_ids])
39    extended_only_names = \
40        set([ship[4] for ship in extended if ship[0] in extended_only_ids])
41    # enfin on retourne les 3 ensembles sous forme d'un tuple
42    return extended_only_names, both_names, abbreviated_only_ids

```

```

1 def diff_bis(extended, abbreviated):
2     """
3     Même code mais qui utilise les compréhensions d'ensemble
4     que l'on n'a pas encore vues - à nouveau, voir semaine 5
5     mais vous allez voir que c'est assez intuitif
6     """
7     extended_ids = {ship[0] for ship in extended}
8     abbreviated_ids = {ship[0] for ship in abbreviated}
9
10    abbreviated_only_ids = abbreviated_ids - extended_ids
11    both_ids = abbreviated_ids & extended_ids
12    extended_only_ids = extended_ids - abbreviated_ids
13
14    both_names = \
15        {ship[4] for ship in extended if ship[0] in both_ids}
16    extended_only_names = \
17        {ship[4] for ship in extended if ship[0] in extended_only_ids}
18
19    return extended_only_names, both_names, abbreviated_only_ids

```

```

1 def diff_ter(extended, abbreviated):
2     """
3     Idem sans les calculs d'ensembles intermédiaires
4     en utilisant les conditions dans les compréhensions
5     """
6     extended_ids = {ship[0] for ship in extended}
7     abbreviated_ids = {ship[0] for ship in abbreviated}
8     abbreviated_only = {ship[0] for ship in abbreviated
9                          if ship[0] not in extended_ids}
10    extended_only = {ship[4] for ship in extended
11                    if ship[0] not in abbreviated_ids}
12    both = {ship[4] for ship in extended
13           if ship[0] in abbreviated_ids}
14    return extended_only, both, abbreviated_only

```

```

1 def diff_quater(extended, abbreviated):
2     """
3     Idem sans indices
4     """
5     extended_ids = {id for id, *_ in extended}
6     abbreviated_ids = {id for id, *_ in abbreviated}
7     abbreviated_only = {id for id, *_ in abbreviated
8                         if id not in extended_ids}
9     extended_only = {name for id, _, _, name, *_ in extended
10                     if id not in abbreviated_ids}
11     both = {name for id, _, _, name, *_ in extended
12            if id in abbreviated_ids}
13     return extended_only, both, abbreviated_only

```

```

1 class Fifo:
2     """
3     Une classe FIFO implémentée avec une simple liste
4     """
5
6     def __init__(self):
7         # l'attribut queue est un objet liste
8         self.queue = []
9
10    def incoming(self, item):
11        # on insère au début de la liste
12        self.queue.insert(0, item)
13
14    def outgoing(self):
15        # une première façon de faire consiste à
16        # utiliser un try/except
17        try:
18            return self.queue.pop()
19        except IndexError:
20            return None

```

```
1 class FifoBis:
2     """
3     une alternative en testant directement
4     plutôt que d'attraper l'exception
5     """
6     def __init__(self):
7         self.queue = []
8
9     def incoming(self, item):
10        self.queue.insert(0, item)
11
12    def outgoing(self):
13        # plus concis mais peut-être moins lisible
14        if self.queue:
15            return self.queue.pop()
16        # pour que pylint soit content on *peut* retourner None explicitement
17        return None
18
```