

# Corrigés de la semaine 3

index - Semaine 3 Séquence 2

```
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}
```

index - Semaine 3 Séquence 2

```
1 def index2(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 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 des listes)
7     """
8     # on initialise le résultat avec un dictionnaire vide
9     result = {}
10    # pour les données étendues
11    for ship in extended:
12        # on affecte les 6 premiers champs
13        # et on ignore les champs de rang 6 et au delà
14        id, latitude, longitude, timestamp, name, country = ship[:6]
15        # on crée une entrée dans le résultat,
16        # avec la mesure correspondant aux données étendues
17        result[id] = [name, country, (latitude, longitude, timestamp)]
18    # maintenant on peut compléter le résultat avec les données abrégées
19    for id, latitude, longitude, timestamp in abbreviated:
20        # et avec les hypothèses on sait que le bateau a déjà été
21        # inscrit dans le résultat, donc result[id] doit déjà exister
22        # et on peut se contenter d'ajouter la mesure abrégée
23        # dans l'entrée correspondant dans result
24        result[id].append((latitude, longitude, timestamp))
25    # et retourner le résultat
26    return result
```

```
1 def merge2(extended, abbreviated):
2     """
3     Une deuxième version, linéaire également
4     """
5     # on initialise le résultat avec un dictionnaire vide
6     result = {}
7     # on remplit d'abord à partir des données étendues
8     for ship in extended:
9         id = ship[0]
10        # on crée la liste avec le nom et le pays
11        result[id] = ship[4:6]
12        # on ajoute un tuple correspondant à la position
13        result[id].append(tuple(ship[1:4]))
14    # pareil que pour la première solution,
15    # on sait d'après les hypothèses
16    # que les id trouvées dans abbreviated
17    # sont déjà présentes dans le resultat
18    for ship in abbreviated:
19        id = ship[0]
20        # on ajoute un tuple correspondant à la position
21        result[id].append(tuple(ship[1:4]))
22    return result
```

```
1 def merge3(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        e[0] : e[4:6] + [ tuple(e[1:4]), tuple(a[1:4]) ]
29        for (e,a) in zip (extended, abbreviated)
30    }
```

```

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     ### on n'utilise que des ensembles dans tous l'exercice
8     # les ids de tous les bateaux dans extended
9     # une compréhension d'ensemble
10    extended_ids = {ship[0] for ship in extended}
11    # les ids de tous les bateaux dans abbreviated
12    # idem
13    abbreviated_ids = {ship[0] for ship in abbreviated}
14    # les ids des bateaux seulement dans abbreviated
15    # une difference d'ensembles
16    abbreviated_only_ids = abbreviated_ids - extended_ids
17    # les ids des bateaux dans les deux listes
18    # une intersection d'ensembles
19    both_ids = abbreviated_ids & extended_ids
20    # les ids des bateaux seulement dans extended
21    # ditto
22    extended_only_ids = extended_ids - abbreviated_ids
23    # pour les deux catégories où c'est possible
24    # on recalcule les noms des bateaux
25    # par une compréhension d'ensemble
26    both_names = \
27        {ship[4] for ship in extended if ship[0] in both_ids}
28    extended_only_names = \
29        {ship[4] for ship in extended if ship[0] in extended_only_ids}
30    # enfin on retourne les 3 ensembles sous forme d'un tuple
31    return extended_only_names, both_names, abbreviated_only_ids

```

```

1  # le module this est implémenté comme une petite énigme
2  # comme le laissent entrevoir les indices, on y trouve
3  # (*) dans l'attribut 's' une version encodée du manifeste
4  # (*) dans l'attribut 'd' le code à utiliser pour décoder
5  #
6  # ce qui veut dire qu'en première approximation on pourrait
7  # obtenir une liste des caractères du manifeste en faisant
8  #
9  # [ this.d [c] for c in this.s ]
10 #
11 # mais ce serait le cas seulement si le code agissait sur
12 # tous les caractères; comme ce n'est pas le cas il faut
13 # laisser intacts les caractères dans this.s qui ne sont pas
14 # dans this.d (dans le sens "c in this.d")
15 #
16 # je fais exprès de ne pas appeler l'argument this pour
17 # illustrer le fait qu'un module est un objet comme un autre
18 #
19
20 def decode_zen(this_module):
21     "décode le zen de python à partir du module this"
22     # la version encodée du manifeste
23     encoded = this_module.s
24     # le 'code'
25     code = this_module.d
26     # si un caractère est dans le code, on applique le code
27     # sinon on garde le caractère tel quel
28     # aussi, on appelle 'join' pour refaire une chaîne à partir
29     # de la liste des caractères décodés
30     return ''.join([code[c] if c in code else c for c in encoded])

```

```

1  # une autre version qui marche aussi, en utilisant
2  # dict.get(key, default)
3  def decode_zen2(this):
4      return "".join([this.d.get(c, c) for c in this.s])

```

```

1 def dispatch1(a, b):
2     """dispatch1 comme spécifié"""
3     # si les deux arguments sont pairs
4     if a%2 == 0 and b%2 == 0:
5         return a*a + b*b
6     # si a est pair et b est impair
7     elif a%2 == 0 and b%2 != 0:
8         return a*(b-1)
9     # si a est impair et b est pair
10    elif a%2 != 0 and b%2 == 0:
11        return (a-1)*b
12    # sinon - c'est que a et b sont impairs
13    else:
14        return a*a - b*b

```

```

1 def dispatch2(a, b, A, B):
2     """dispatch2 comme spécifié"""
3     # les deux cas de la diagonale \
4     if (a in A and b in B) or (a not in A and b not in B):
5         return a*a + b*b
6     # sinon si b n'est pas dans B
7     # ce qui alors implique que a est dans A
8     elif b not in B:
9         return a*(b-1)
10    # le dernier cas, on sait forcément que
11    # b est dans B et a n'est pas dans A
12    else:
13        return (a-1)*b

```