Corrigés de la semaine 3

index - Semaine 3 Séquence 2

```
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 - Semaine 3 Séquence 2

```
def index2(bateaux):
    """
    La même chose mais de manière itérative
    """
    # si on veut décortiquer
    resultat = {}
    for bateau in bateaux:
        resultat [bateau[0]] = bateau
    return resultat
```

merge - Semaine 3 Séquence 2

```
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 des listes)
    """
    # on initialise le résultat avec un dictionnaire vide
    result = {}
    # pour les données étendues
    for ship in extended:
        # on affecte les 6 premiers champs
        # et on ignore les champs de rang 6 et au delà
        id, latitude, longitude, timestamp, name, country = ship[:6]
        # 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 ls mesure abrégée
        # dans l'entrée correspondant dans result
        result[id].append((latitude, longitude, timestamp))
    # et retourner le résultat
    return result
```

merge - Semaine 3 Séquence 2

```
def merge2(extended, abbreviated):
    """
    Une deuxième version, linéaire également
    """
    # 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 resultat
    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 - Semaine 3 Séquence 2

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

diff - Semaine 3 Séquence 3

```
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
    # une compréhension d'ensemble
    extended_ids = {ship[0] for ship in extended}
    # les ids de tous les bateaux dans abbreviated
    # idem
    abbreviated_ids = {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 = \
          {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}
    # enfin on retourne les 3 ensembles sous forme d'un tuple
    return extended_only_names, both_names, abbreviated_only_ids
```

decode_zen - Semaine 3 Séquence 5

```
# le module this est implémenté comme une petite énigme
# comme le laissent entrevoir les indices, on y trouve
# (*) dans l'attribut 's' une version encodée du manifeste
# (*) dans l'attribut 'd' le code à utiliser pour décoder
#
# ce qui veut dire qu'en première approximation on pourrait
# obtenir une liste des caractères du manifeste en faisant
#
# [ this.d [c] for c in this.s ]
#
# mais ce serait le cas seulement si le code agissait sur
# tous les caractères; comme ce n'est pas le cas il faut
# laisser intacts les caractères dans this.s qui ne sont pas
# dans this.d (dans le sens "c in this.d")
#
# je fais exprès de ne pas appeler l'argument this pour
# illustrer le fait qu'un module est un objet comme un autre
#

def decode_zen(this_module):
    "décode le zen de python à partir du module this"
    # la version encodée du manifeste
    encoded = this_module.s
    # le 'code'
    code = this_module.d
    # si un caractère est dans le code, on applique le code
    # sinon on garde le caractère tel quel
    # aussi, on appelle 'join' pour refaire une chaîne à partir
    # de la liste des caractères décodés
    return ''.join([code[c] if c in code else c for c in encoded])
```

decode_zen - Semaine 3 Séquence 5

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

dispatch1 - Semaine 3 Séquence 7

```
def dispatch1(a, b):
    """dispatch1 comme spécifié"""
    # si les deux arguments sont pairs
    if a%2 == 0 and b%2 == 0:
        return a*a + b*b
    # si a est pair et b est impair
    elif a%2 == 0 and b%2 != 0:
        return a*(b-1)
    # si a est impair et b est pair
    elif a%2 != 0 and b%2 == 0:
        return (a-1)*b
    # sinon - c'est que a et b sont impairs
    else:
        return a*a - b*b
```

dispatch2 - Semaine 3 Séquence 7

```
def dispatch2(a, b, A, B):
    """dispatch2 comme spécifié"""
    # les deux cas de la diagonale \
    if (a in A and b in B) or (a not in A and b not in B):
        return a*a + b*b
    # sinon si b n'est pas dans B
    # ce qui alors implique que a est dans A
    elif b not in B:
        return a*(b-1)
    # le dernier cas, on sait forcément que
    # b est dans B et a n'est pas dans A
    else:
        return (a-1)*b
```