

# MOOC Python

## Corrigés de la semaine 4

dispatch1 - Semaine 4 Séquence 2

```
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
```

dispatch2 - Semaine 4 Séquence 2

```
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
```

```

1 def libelle(ligne):
2     # on enlève les espaces et les tabulations
3     ligne = ligne.replace(' ', '').replace('\t','')
4     # on cherche les 3 champs
5     mots = ligne.split(',')
6     # si on n'a pas le bon nombre de champs
7     # rappelez-vous que 'return' tout court
8     # est équivalent à 'return None'
9     if len(mots) != 3:
10        return
11    # maintenant on a les trois valeurs
12    nom, prenom, rang = mots
13    # comment présenter le rang
14    rang_ieme = "1er" if rang == "1" \
15                else "2nd" if rang == "2" \
16                else f"{rang}-ème"
17    return f"{prenom}.{nom} ({rang_ieme})"

```

```

1 def pgcd(a, b):
2     "le pgcd de a et b par l'algorithme d'Euclide"
3     # l'algorithme suppose que a >= b
4     # donc si ce n'est pas le cas
5     # il faut inverser les deux entrées
6     if b > a :
7         a, b = b, a
8     if b == 0:
9         return a
10    # boucle sans fin
11    while True:
12        # on calcule le reste
13        r = a % b
14        # si le reste est nul, on a terminé
15        if r == 0:
16            return b
17        # sinon on passe à l'itération suivante
18        a, b = b, r

```

pgcd (bis) - Semaine 4 Séquence 3

```
1  # il se trouve qu'en fait la première inversion n'est
2  # pas nécessaire
3  # en effet si a <= b, la première itération de la boucle
4  # while va faire
5  # r = a % b = a
6  # et ensuite
7  # a, b = b, r = b, a
8  # ce qui provoque l'inversion
9  def pgcd_bis(a, b):
10     # si l'un des deux est nul on retourne l'autre
11     if a * b == 0:
12         return a or b
13     # sinon on fait une boucle sans fin
14     while True:
15         # on calcule le reste
16         r = a % b
17         # si le reste est nul, on a terminé
18         if r == 0:
19             return b
20         # sinon on passe à l'itération suivante
21         a, b = b, r
```

pgcd (ter) - Semaine 4 Séquence 3

```
1  # une autre alternative, qui fonctionne aussi
2  # plus court, mais on passe du temps à se convaincre
3  # que ça fonctionne bien comme demandé
4  def pgcd_ter(a, b):
5     # si on n'aime pas les boucles sans fin
6     # on peut faire aussi comme ceci
7     while b:
8         a, b = b, a % b
9     return a
```

```

1  # la définition des différentes tranches
2  bands = [
3      # à partir de 0. le taux est nul
4      (0, 0.),
5      # jusqu'à 11 500 où il devient de 20%
6      (11_500, 20/100),
7      # etc.
8      (45_000, 40/100),
9      (150_000, 45/100),
10 ]
11
12
13 def taxes(income):
14     """
15     U.K. income taxes calculator
16     https://www.gov.uk/income-tax-rates
17
18     utilise un for avec un break
19     """
20     # on accumule les morceaux
21     amount = 0
22
23     # en faisant ce zip un peu étrange, on va
24     # considérer les couples de tuples consécutifs dans
25     # la liste bands
26     for (band1, rate1), (band2, _) in zip(bands, bands[1:]):
27         # le salaire est au-delà de cette tranche
28         if income >= band2:
29             amount += (band2-band1) * rate1
30         # le salaire est dans cette tranche
31         else:
32             amount += (income-band1) * rate1
33             # du coup on peut sortir du for par un break
34             # et on ne passera pas par le else du for
35             break
36     # on ne passe ici qu'avec les salaires dans la dernière tranche
37     # en effet pour les autres on est sorti du for par un break
38     else:
39         band_top, rate_top = bands[-1]
40         amount += (income - band_top) * rate_top
41     return(int(amount))

```

```
1  # Une version proposée par adrienollier
2  # qui contourne la difficulté en utilisant
3  # habilement math.inf
4  # nombre infini qui est supérieur à tous les nombres
5
6  import math
7
8  TaxRate = (
9      (0, 11_500, 0),
10     (11_501, 45_000, 20),
11     (45_001, 150_000, 40),
12     (150_001, math.inf, 45),
13 )
14
15 def taxes_bis(income):
16
17     due = 0
18     for floor, ceiling, rate in TaxRate:
19         due += (min(income, ceiling) - floor + 1) * rate / 100
20         if income <= ceiling:
21             return int(due)
```

```

1  # La même chose mais améliorée pour éviter les
2  # répétitions dans le tuple qui sert de base au calcul
3
4  import math
5  tax_rate_nodup = (
6      (11_500, 0),
7      (45_000, 20),
8      (150_000, 40),
9      (math.inf, 45),
10 )
11
12 # calculer ce qui s'appelle TaxRate dans la solution taxes_bis
13 b1, r1 = tax_rate_nodup[0]
14
15 tax_rate = [ (0., b1, r1) ]
16 tax_rate += [
17     (b1, b2, r2)
18     for (b1, _), (b2, r2) in zip(tax_rate_nodup, tax_rate_nodup[1:])
19 ]
20
21 # à ce stade on peut utiliser le code de taxes_bis
22 # presque à l'identique
23 def taxes_ter(income):
24
25     due = 0
26     for floor, ceiling, rate in tax_rate:
27         due += (min(income, ceiling) - floor) * rate / 100
28         if income <= ceiling:
29             return int(due)

```

```

1  import math
2
3  def distance(*args):
4      "la racine de la somme des carrés des arguments"
5      # avec une compréhension on calcule la liste des carrés des arguments
6      # on applique ensuite sum pour en faire la somme
7      # vous pourrez d'ailleurs vérifier que sum ([]) = 0
8      # enfin on extrait la racine avec math.sqrt
9      return math.sqrt(sum([x**2 for x in args]))

```

```
1 from operator import mul
2
3 def numbers(*liste):
4     """
5     retourne un tuple contenant
6     (*) la somme
7     (*) le minimum
8     (*) le maximum
9     des éléments de la liste
10    """
11
12    if not liste:
13        return 0, 0, 0
14
15    return (
16        # la builtin 'sum' renvoie la somme
17        sum(liste),
18        # les builtin 'min' et 'max' font ce qu'on veut aussi
19        min(liste),
20        max(liste),
21    )
```

```
1  # en regardant bien la documentation de sum, max et min,
2  # on voit qu'on peut aussi traiter le cas singulier
3  # (pas d'argument) en passant
4  #   start à sum
5  #   et default à min ou max
6  # comme ceci
7  def numbers_bis(*liste):
8      return (
9          # attention:
10         # la signature de sum est: sum(iterable[, start])
11         # du coup on ne peut pas passer à sum start=0
12         # parce que start n'a pas de valeur par défaut
13         sum(liste, 0),
14         # par contre avec min c'est min(iterable, *[, key, default])
15         # du coup on doit appeler min avec default=0 qui est plus clair
16         # l'étoile qui apparaît dans la signature
17         # rend le paramètre default keyword-only
18         min(liste, default=0),
19         max(liste, default=0),
20     )
```